Technical University of Crete

Diploma Thesis

Design and Implementation of an FPGA-Based Convolutional Neural Network Accelerator

Author:
Antonios Georgios Pitsis

Thesis Committee:
Prof. Apostolos Dollas
Prof. Dionisios Pnevmatikatos
Dr. Christos Kozanitis

A thesis submitted in fulfillment of the requirements for the DIPLOMA of Electrical and Computer Engineering

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Lab

October 10, 2018

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

DIPLOMA THESIS

Design and Implementation of an FPGA-Based Convolutional Neural Network Accelerator

by Antonios Georgios Pitsis

In recent years Convolutional Neural Networks (CNNs) have shown extremely growth due to their effectiveness at complex image recognition problems. They are currently adopted to solve an ever greater number of problems, ranging from speech recognition to image segmentation and classification. The continuing increasing amount of processing required by CNNs creates the field for hardware support methods. Moreover, CNN workloads have a streaming nature, well suited to reconfigurable hardware architectures such as FPGAs. The amount of research on the Machine Learning and especially on CNN (implemented on FPGA platforms) within the last 4 years demonstrates the tremendous industrial and academic interest. This study presents a CNN inference accelerator over FPGAs. The network we aim to accelerate was developed by Dr. Tsagatakis in the context of DEDALE project (Horizon 2020 [33]) for astrophysics subject. After carrying out Sensitivity Analysis computational workloads and memory accesses are analyzed, as well as compression methods and algorithmic optimizations to exploit FPGA parallelism. At the level of neurons, optimizations of the convolutional and fully connected layers are explained and compared. At the network level, approximate computing optimization methods are examined limited by not reducing the accuracy of the network. The platforms were used are ZCU102 and QFDB(a custom 4-FPGA platform developed at FORTH). The implemented accelerator was managed to achieve 20x latency speedup, 2.17x throughput speedup and 11.9x energy efficient over GPU NVIDIA-Quadro-K2200 in terms of EuroExa [19] project.

This work is partially supported by the European Union Horizon 2020 program through the EuroExa project.

Acknowledgements

First of all, I would like to thank my supervisor, Prof. Apostolos Dollas for his trust in my abilities and his valuable guidance during the procedure of this thesis, as well as for the chance he gave me to broaden my horizons beyond my current field of expertise in the domain of Artificial Intelligence.

Furthermore, I would like to thank my co-partner of this project Giannis Kalomoiris, for the collaboration on the successful completion of this project. Moreover, I would like to deeply thank Dr. Gregory Tsagatakis (FORTH), for the excellent collaboration and the confidence he showed us to continue his own work, as well as for his valuable advice and guidance on technical difficulties and especially on Deep Learning. I would also like to express my deepest gratitude to Dr. Christos Kozanitis (FORTH) for his significant contribution and for the excellent cooperation during the course of this thesis. In addition, I would like to thank Angelos Ioannou for his valuable contribution as far as technical issues are concerned with the QFDB.

I would also to thank all the MHL lab. More specifically Dr Anonis Nikitakis for our conversations and valuable knowledge about Machine Learning, Dr Kostas Georgopoulos and Dimitris Theodoropoulos for the time he spent on me to solve technical problems. In particular PHD Candidate Pavlos Malagonakis and Msc Student Vasilis Amourgianos for his valuable input on Vivado Tools and for the time they devoted to me to complete this project. Moreover I would like to thank Petros Toupas for the valuable support and the countless discussions we had for our projects.

Last but not least i would like to express my deepest gratitude to my family and friends for the support during this thesis and especially my friends Argirw Tsilia and Alexia Faidherbe for his valuable support in image design.

Antonios Georgios Pitsis,
Chania 2018

Contents

Abstract iii

Acknowledgements v

List of Figures xi

List of Tables xiii

List of Algorithms xv

List of Abbreviations xvii

1 Introduction 1

1.1 Motivation 1

1.2 Scientific Contributions 2

1.3 Thesis Outline 3

2 Theoretical Background 5

2.1 Machine Learning 6

2.2 Convolutional Neural Network 6

2.3 Structure of Convolutional Neural Network 8

2.3.1 Convolution Layer 8

2.3.2 Pooling 9

2.3.3 Activation Function 9

2.4 Typical Architecture of a 1-Dimensional CNN 14

2.4.1 Convolutional Layer 14

2.4.2 Fully-Connected Layer 15

2.4.3 Final Classification 15

2.4.4 Pooling Layer 15

3 Related Work 17

3.1 Google Brain Project 17

3.1.1 DistBelief 17

3.1.2 TensorFlow 17

3.1.3 Tensor Processing Unit (TPU) 18
3.2 GPU approach 18
3.3 The FPGA perspective 19
3.3.1 Training 20
3.3.2 Weight Reduction 20
Computional Reduction 21
3.3.3 Inference 21
Weight Reduction 21
Re-training approaches 22
3.4 Convolutional Neural Networks for Spectroscopic Red-shift Estimation on Euclid Data 22
Training and Evaluation 23
3.5 Thesis Approach 24

4 Robustness analysis of CNN **25**
4.1 MATLAB vs TensorFlow 25
4.1.1 Data types 28
4.2 Memory Footprint 29
4.3 Weight Distribution 30
4.4 Pruning 33
4.4.1 Using Fixed Point 34
4.4.2 Dynamic Fixed Point 35
4.5 Evaluating Results 36
4.6 Clustering 36
4.6.1 Pre-Clustering Techniques 36
4.6.2 Quantization with Codebook 37
4.6.3 Comparison of different Codebooks 38
Stohasting Clustering 39
4.6.4 Proposed methods for quantization with Codebook 40
4.7 Pair-Compression 42
4.7.1 Proposed Methods and Pair-Compression 43
4.8 Quad-Compression 44
4.9 Pair Compression and Hierarchical Clustering 45
4.10 Second Level Codebook (SLC) 48
4.10.1 SLC with higher weights-blocks 49
Decreasing error rate 50
Lowering bits/weights ratio 50
Golden Ratio 51

Lossless Method . . . 52
4.11 SLC comparison with Huffman . . . 55
4.12 SLC, Hierarchical Clustering and Pair Compression . . . 57
4.13 Proposed methods comparison with Binarized Approach . . . 59

5 FPGA Implementation **61**
5.1 Tools Used . . . 61
5.1.1 Vivado HLS . . . 61
Synthesis Report in Vivado HLS . . . 62
Optimizations in Vivado HLS . . . 62
5.1.2 Vivado IDE . . . 63
5.1.3 Vivado SDK . . . 64
5.2 FPGA platforms . . . 64
5.2.1 Xilinx Zynq UltraScale+ MPSoC ZCU102 . . . 64
5.2.2 QFDB . . . 65
5.3 Bottom-up Strategy . . . 65
5.3.1 Two Stage Architecture . . . 66
5.4 First Approach . . . 66
5.4.1 Memory and I/0 . . . 66
5.4.2 Transference to HLS . . . 67
5.4.3 Convolution (1-D) . . . 67
Array Partitioning . . . 68
Pipeline . . . 68
5.4.4 Convolution (2-D) . . . 69
Array Partitioning . . . 69
Pipeline . . . 69
5.4.5 Fully Connected . . . 70
Pipeline . . . 70
5.5 A better Approach . . . 71
5.5.1 Larger Streaming Buses . . . 71
5.5.2 Multiple DMAs . . . 72
DDR Bandwidth . . . 72
5.5.3 Convolution (1-D) and Convolution (2-D) . . . 73
5.5.4 Fully Connected . . . 73
5.6 Architecture v1.0 . . . 76
5.6.1 Embedding Compressed Weights . . . 77
5.6.2 Pipelining Convolution Module . . . 77
Shifted FIFO . . . 78

Task Level Parallelism 78
5.6.3 Pipelining two Modules 79
Convolution Transformations 80
Fully Connected Transformations 80
5.6.4 Resource Optimizations 82
5.7 Architecture v2.0 82
5.8 Resource Optimizations 83
5.8.1 Batching 83
5.9 Porting to QFDB 83

6 Results **85**
6.1 Specification of Compared Platforms 85
6.1.1 Intel i7 7700HQ 85
6.1.2 NVIDIA Quadro K2200 86
6.2 Power Consumption 86
6.3 Energy Consumption 86
6.4 Throughput and Latency Speedup 87
6.5 Architecture v1.0 87
6.5.1 ZCU-102 88
6.5.2 QFDB 89
6.6 Architecture v2.0 90
6.6.1 ZCU-102 90
6.6.2 QFDB 92
6.7 Final Performance 92

7 Conclusions and Future Work **97**
7.1 Conclusions 97
7.2 Future Work 97

References **99**

List of Figures

2.1 Standard Structure of Real Neural Network 7
2.2 Architecture of CNN 8
2.3 Pooling Comparison 9
2.4 Perceptron 11
2.5 Sigmoid Function 12
2.6 tanh(x) 13
2.7 ReLU 13
2.8 Simple 1-Dimensional CNN 16

3.1 Accuracy of Euclede 23

4.1 Weight Distribution of conv1 30
4.2 Weight Distribution of conv2 31
4.3 Weight Distribution of conv3 32
4.4 Weight Distribution of dense 32
4.5 Pruning Comparison 33
4.6 Weight Distribution after Pruning 34
4.7 Pre-Clustering Compression 37
4.8 Clustering sample 38
4.9 Clustering-16 example 39
4.10 Lloyds Clustering 40
4.11 Inverse Density 41
4.12 Clustering 16 42
4.13 Pair Compression example 43
4.14 Pair Compression 44
4.15 Quad Compression example 45
4.16 Comparison of Weight Distribution CDF v.2 46
4.17 Hierarchical Clustering And Pair Compression Procedure 46
4.18 Hierarchical Clustering And Quad Compression Procedure 47
4.19 SLC compression example 49
4.20 SLC in Clustering with 256 centroids 51
4.21 Weights-Block-2 on Clustering with 256 centroids 52

4.22 SLC in Clustering with 18 centroids 53
4.23 Weights-Block-2 on Clustering with 18 centroids 53
4.24 SLC Performance 54
4.25 Comparison of SLC compression on Codebook 18 and 17 versus Huffman Coding 56
4.26 Compression Flow using Pair Compression and SLC 56
4.27 Compression Flow using Pair Compression and SLC 57
4.28 Compression Flow using Quad Compression and SLC 58

5.1 Datapath of Two-Stage Architecture 66
5.2 Datapath of Convolution(1-D) 68
5.3 Convolutional Layers Time-Chart (FA) 69
5.4 Fully Connected Module 70
5.5 CNN Time-Chart (FA) 71
5.6 Datapath of the Architecture v1.0 76
5.7 Convolutional Layers Time-Chart (Arch1.) 77
5.8 CNN Time-Chart (Arch1) 80

6.1 Throughput Speedup for different batches 93
6.2 Throughput Speedup for different Number of Images 94
6.3 Power and Energy Efficiency over GPU 94
6.4 Utilization after the use of Resource Optimizations 95

List of Tables

3.1 Comparison of CPU and GPU training running time 24

4.1 Error rate 29
4.2 Weights Memory Footprint 29
4.3 Data Stages Memory Footprint 29
4.4 Fixed point formats 35
4.5 Dynamic Fixed Point 35
4.6 Pre-Clustering Compression Error 37
4.7 Lloyds Clustering 40
4.8 Compression Methods 45
4.9 SLC compression on Codebook 256 for different Weights Block . . . 50
4.10 SLC compression on Codebook 18 for different Weights Block . . . 51
4.11 SLC compression on Codebook 17 for different Weights Block . . . 54
4.12 Comparison of SLC compression on Codebook 18 and 17 with Huffman Coding 55
4.13 Final Compression Methods 59
4.14 Proposed methods comparison with Binarized Approach 59

5.1 ZCU102 Specifications 65
5.2 First Approach Performance 71
5.3 Memory Bandwidth 73
5.4 Second Approach Performance 75
5.5 Pipeline Convolutional Layers 78
5.6 Pipeline Convolutional with Fully Connected Layer 82
5.7 Architecture v1.0 Performance 82
5.8 Architecture v2.0 Performance 83
5.9 Pipeline Convolutional Layers 84

6.1 Intel i7 7700HQ Specifications 85
6.2 NVIDIA Quadro K2200 Specifications 86
6.3 Architecture v1.0 88
6.4 Architecture v1.0 comparison with CPU and GPU 88
6.5 Speedup over GPU and CPU 88

6.6 Energy and Power Efficiency over GPU and CPU 89
6.7 Architecture v1.0 comparison with CPU and GPU 89
6.8 Speedup over GPU and CPU 89
6.9 Energy and Power Efficiency over GPU and CPU 90
6.10 Architecture v2.0 90
6.11 Architecture v2.0 comparison with CPU and GPU 91
6.12 Speedup over GPU and CPU 91
6.13 Energy and Power Efficiency over GPU and CPU 91
6.14 Architecture v2.0 comparison with CPU and GPU 92
6.15 Speedup over GPU and CPU 92
6.16 Energy and Power Efficiency over GPU and CPU 92

List of Algorithms

1 Convolution (1-D) 26
2 Convolution (2-D) 27
3 Fully Connected 27
4 ReLU 28
5 SoftMax 28
6 Matrix Multiplication 74
7 Matrix Multiplication (Opt) 75
8 Sequential Parallel with MUTEX 79
9 Matrix Multiplication (Opt) 81

List of Abbreviations

CNN	**C**onvolutional **N**eural **N**etwork
FC	**F**ully **C**onnected
DNN	**D**eep **N**eural **N**etwork
ANN	**A**rtificial **N**eural **N**etwork
DP	**D**eep **L**earning
AI	**A**rtificial **I**ntelligence
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
CPU	**C**entral **P**rocessor **U**nit
GPU	**G**raphic **P**rocessor **U**nit
TPU	**T**ensor **P**rocessor **U**nit
ML	**M**achine **L**earning
QFDB	**Q**uad **F**PGA **D**aughter **B**oard
ReLU	**Re**ctified **L**inear **U**nit
HLS	**H**igh **L**evel **S**ynthesis
SDK	**S**oftware **D**evelopment **K**it
SLC	**S**econd **L**evel **C**odebook
RAM	**R**andom **A**ccess **M**emory
D-RAM	**D**ynamic **R**andom **A**ccess **M**emory
B-RAM	**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

Dedicated to my family and friends…

Chapter 1

Introduction

The exponential growth of data during the last years is increasing, leading to the need for proper management.We always knew it was big in 2010 cracking the zettabyte barrier. Extracting and analyzing these amounts of information makes it difficult or even impossible using conventional software tools and technologies. Digital information, sizes and dimensions, is growing at astonishing rates. For example in 2013, according to the National Security Agency, the Internet is processing 1.8 Petabytes of data per day ("National Security Agency", 2013) [1]. During 2006-2011, digital data has grown 9 times in volume [24]. Moreover, its amount in the world will be estimated to reach 35 trillion gigabytes [25]. More specifically, according to Forbes ([48]) we produce every day globally is 2.5 ExaBytes (quintillion of bytes) according to our current pace. This rapid explosion of digital data brings big opportunities for innovative methods and creates the field to explore ways to extract a high-level understanding of the low-level information given by raw data such as images, video and speech sequences. Among the proposed methods, Convolutional Neural Networks (CNNs) [81] have become the driving force by achieving accuracy even better than humans in many applications related to machine vision (e.g detection [26], classification [38], segmentation [82]) and speech recognition [58].

1.1 Motivation

State-of-the-art accuracy results in image , speech , language and many other tasks are being achieved via using convolutional neural networks (CNNs) (e.g. [9], [12], [42], [54],[55],[13]). This outstanding performance comes at the price of a huge computational cost. CNNs require up to 40 GOP/s to classify efficiently a single image [70]. As a result, dedicated hardware is required to accelerate their execution. Graphics Processing Units (GPUs), have been the most widely used platform to implement CNNs as they present the best performance in terms of pure computational throughput, reaching up to 11 TFLOP/s [22]. Nevertheless, in terms of power consumption, Field-Programmable Gate Array (FPGA) solutions are known to be more energy efficient (vs GPUs).

CNN training is highly compute-intensive, requiring hours, days or even weeks to achieve high accuracy using high-end graphics processing units (GPUs). Hardware acceleration is particularly suitable for inference, as training is typically done once off-line, whereas inference with a is applied repeatedly. As a result, plethora of FPGA-Based CNN accelerators have been proposed mostly for inference. They targeting both High-Performance Computing (HPC) i.e. data-centers [61] and embedded (low-power) applications [4]. Moreover, there is increased emphasis on performing CNN inference embedded-computing context (e.g. mobile applications, aerospace, etc), where low-power and low latency are the most important metrics.

1.2 Scientific Contributions

The scientific contribution of this work is focused on two aspects. Firstly several methods have been performed to scale down the memory footprint and computation complexity by reducing the redundancy of CNN models. These methods include pruning, lower floating point, static and dynamic fixed point, clustering algorithms Memory Layout Transformation etc was based on sensitivity analysis implemented in MATLAB. The results of this analysis show us the limitations of how aggressive we can be in each technique, having as a limiting factor to keep low error rate of the network. In typical neural networks, there are millions of parameters which define the model and requires a large amount of data to store them. This problem is especially intense in implementations over FPGA where we have limitations in memory. B-RAM is immensely fast but its size is too small (few MB), on the other hand, D-RAM has significantly bigger size (tens of GB) but limited bandwidth. In our network we have 22.776.272 (64-bit, double precision floating point) weights, meaning 173.77 MB, thus it is essential to reduce its size efficiently to accelerate the network. More specifically the hardware of choice is GPU as they present the best performance in terms of pure computational throughput, using a runtime such as TensorFlow. TensorFlow provides high-level of abstraction features as a general purpose solution suitable for GPUs and CPUs. According to a Google, there was a fear that if the over a billion Android users began to use Google's voice search for just three minutes a day it would require Google to double its number of data centers (using CPUs and GPUs leads to high energy and power demands) [27]. For this reason Google deployed **TPU [57]** targeting its data centers to accelerate the inference phase of neural networks (NNs) having lower power and energy consumption comparing to GPU and CPU solutions.

As a result, dedicated hardware is required to accelerate the execution of CNN applications efficiently leading to high throughput/Watt (or energy) results.

We managed to use FPGAs as CNN accelerators using some of the techniques mentioned above and exploit its parallelism. A pipeline was originally created at each layer separately and then it was expanded between the layers. In order to achieve this, we have to transform the order in which the layers export their results so that the next layers are able to start their process before the previous finish their own. Another challenge was to limit I/O transactions which are the main bottleneck in every FPGA implementation. Several architectures are examined but in order to split efficiently workload in each FPGA, eventually we managed to fit entire CNN in a single FPGA applied significant resource optimization in order to achieve this. Finally, We propose two architectures for a single FPGA (ZCU-102) and for QFDB (Quad FPGA Daughter Board) [19]. The implemented accelerator was managed to achieve 2.5x speedup and 10x energy efficient over GPU NVIDIA-Quadro-K2200. Both results are important when we are targeting satellite-based applications (Euclid satellite).

1.3 Thesis Outline

In this section we outline the organization of this thesis.

- **Chapter 2:** We describe in detail the theoretical background of Machine Learning and especially for CNN.
- **Chapter 3:** We describe in detail the related work in the field of CNN and more specifically for hardware implementations.
- **Chapter 4:** We present our Sensitivity Analysis of the given CNN 2.8 in MATLAB. Moreover, several techniques have been performed to limit the memory footprint and computation complexity by reducing the redundancy of the network based on our robustness analysis.
- **Chapter 5:** We develop two architectures for a single FPGA (ZCU-102) and for QFDB (Quad FPGA).
- **Chapter 6:** We present and compare both Architectures results in throughput,latency power and energy consumption with GPU (NVIDIA-Quadro-K2200) and CPU(i-7 7700HQ).

- **Chapter 7:** We conclude this thesis, and we provide directions for future work and possible extensions to our work.

  In chapter 3,4 the CNN that we analyze was developed by FORTH researchers (Dr. Tsagatakis and his team). The corresponding section has been obtained from the paper of Dr. Tsagatakis (currently is on submission stage) with his permission (private communication).

Chapter 2

Theoretical Background

In deep learning, the algorithms we use now are versions of the algorithms we were developing in the 1980s, the 1990s. People were very optimistic about them, but it turns out they didn't work too well."

Geoffrey Hinton

In this Chapter, we describe in detail the theoretical background of Machine Learning and especially for CNN.

The human and several animals have very complex visual recognition systems. We are able to distinguish and classify objects independently. The structure neural network is imitated from the biological brain structure. Each neuron in the network is connected to other neurons through synapses. When a neuron is "fired", sends to the next layer of neurons a chemical substance to change the potential "state" of these neurons. If the potential of a neuron exceeds a threshold, it is activate. Otherwise, it will not be activated. One of the most common research fields in computer science is image recognition. Convolutional Neural Networks (CNNs) neural networks aims to solve problems such as image recognition, by using significant process, such as Gradient Descent [14] and Backpropagation [37]. One of the earliest neural network was inspired in 1943. McCulloch and Pitts (MCP) [77] hve raised an artificial neural model, which was aimed to simulate the process of human neuron response using a computer. It simplifies neurons into two basic processes: linearization of the input signal, and non-linear activation (threshold method). The first use of MCP for machine learning was the perceptron algorithm invented by Rosenblatt in 1958 [65] constituting the foundations of further development in ML.

2.1 Machine Learning

Machine learning is a sub-field of computer science that uses techniques with statistical nature to develop in computer systems the ability to progressively learn (gradually improve performance on a specific task) with data, without being programmed in a predetermined manner [41].

The name "machine learning" first was given by Arthur Samuel in 1959 [66]. Machine Learning is the sub-field of Artificial Intelligence that explores and study the construction of algorithms that can "learn" from data and make predictions on them [64]. Firstly was developed and evolved from the fields of computational "learning" theory and pattern recognition in artificial intelligence. Machine learning is employed in a wide range of tasks related to computer. Designing and programming algorithms with these characteristics with high performance are difficult or even impossible. Machine learning promises to approach tasks such as include email filtering, network filtering or various insiders, learning to classify, and computer vision is problems that.

Machine learning is closely related to (and often overlaps with) computational statistics, which also focuses on prediction-making through the use of computers. It has strong ties to mathematical optimization, which delivers methods, theory and application domains to the field. Machine learning is sometimes conflated with data mining [46], where the latter sub-field focuses more on exploratory data analysis and is known as unsupervised learning [8]. Machine learning can also be unsupervised [31] and be used to learn and establish baseline behavior for various tasks and entities and then used to find meaningful disorders.

In data analytics, machine learning is a method used to manage complex models and algorithms that focus to prediction. Many applications in commercial use, known as predictive analytics. These analytical models allow engineers, researchers, and analysts to be able to produce reliable decisions and high-accuracy results and uncover hidden information and structures through learning from historical relationships and trends in the data [45].

2.2 Convolutional Neural Network

In machine learning, a convolutional neural network (CNN) is a sub-class of deep, feed-forward artificial neural networks, most commonly applied to analyze visual data.

CNNs use a variation of multi-layer perceptrons designed to require a minimum amount of preprocessing. Convolutional Neural Networks were inspired by biological processes. Thus the connectivity patterns between neurons relied on the study of the organization of the animal visual cortex. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. The receptive fields of different neurons partially overlap such that they cover the entire visual field.

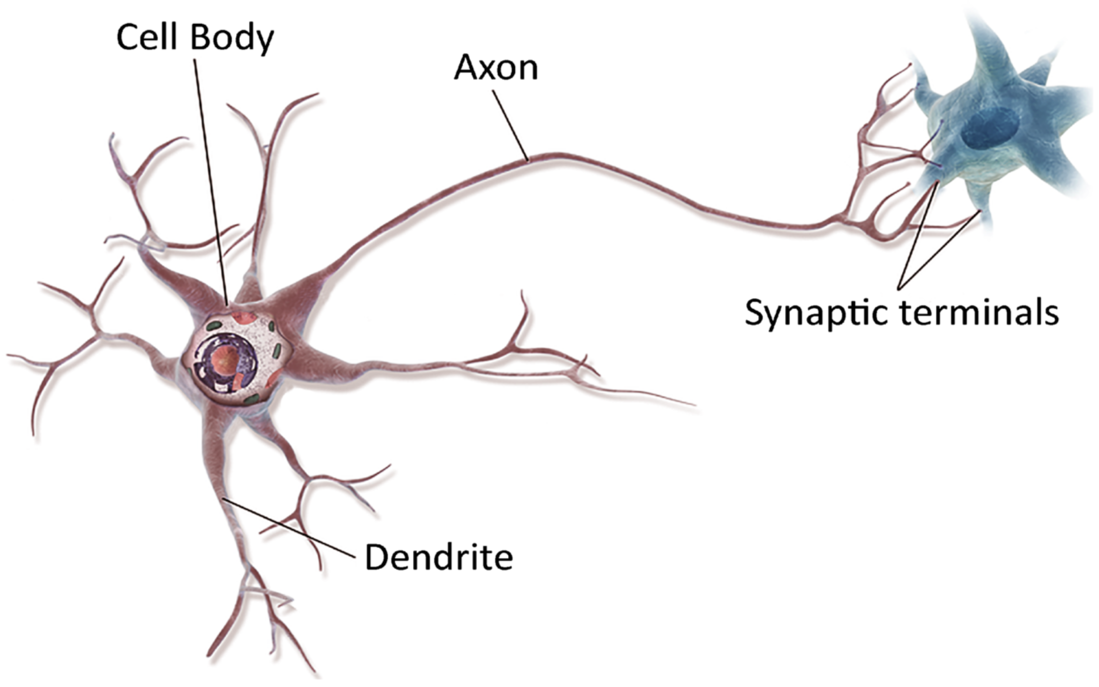


FIGURE 2.1: Standard Structure of Real Neural Network: URL

Convolutional Neural Networks are very similar to ordinary Neural Networks. They are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. And they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer.

DNNs can replace a Machine Learnign expert on Feauture Extraction using relatively little pre-processing compared to other image classification algorithms. This means that the network learns the filters that in traditional algorithms were hand-engineered. This independence from prior knowledge and human effort in feature design is a major advantage. They have applications in image and video recognition and natural language processing [13].

2.3 Structure of Convolutional Neural Network

A typical Convolutional Neural Network Figure 2.2 consists of millions of neurons where they are organized in several layers. The beginning layer is Convolution layer and the last few layers,1-5 depending on the application, are Fully Connected. The las Fully Connected Layer also named as Classifier. The layers between them are called hidden layers. The main purpose of the convolution layer is to extract image features, then drive them into the hidden layers of computing, and extract the results through the output layer. Layers among hidden layers usually, such as pooling layers (max, average etc), are sub-sampling layers, are partially connected, while the output layers are fully connected. Between hidden layers often there are activation functions that help to keep valuable information for next layers.

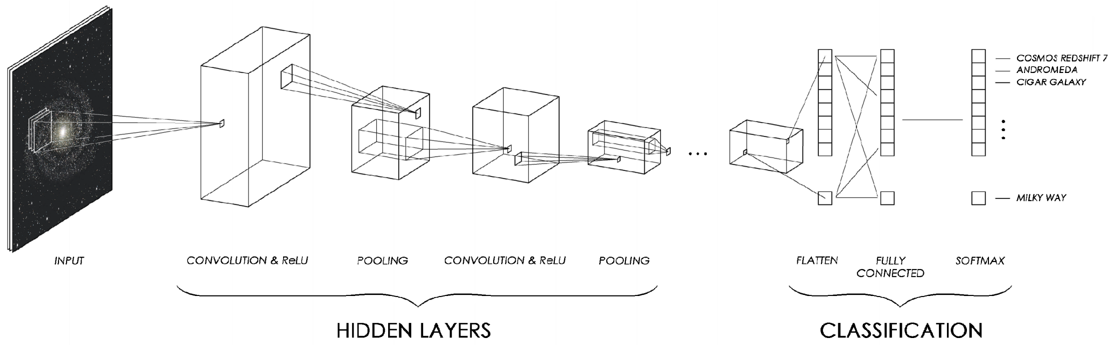


FIGURE 2.2: Architecture of CNN

2.3.1 Convolution Layer

Convolutional layers apply a convolution operation to the input, passing the result to the next layer. Therefore convolution emulates the response of an individual neuron to visual stimuli [32].It can be implemented in a variety of ways.
There are some hyper-parameters that are used to configure a convolution layer:

- **Kernel size**(K): Size of filter
- **Stride**(S): How many pixels the kernel window will slide (on each dimension). Normally 1, in conv layers, and 2 in pooling layers.
- **Zero Padding**(pad): 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** returns the same size of the input with appropriate zero padding.
  - **full** returns the full convolution with full zero-padded edges.

- **Number of filters**(F): Number of patterns and structures, known as "feature maps", that the conv layer will look for.

2.3.2 Pooling

A pooling layer is another building block of a CNN. They are used to progressively reduce the size of the representation, but not depth. By having less spatial information you gain computation performance. Also, less spatial information means fewer parameters, so less chance to over-fit the network. Convolutional networks may include local or global pooling layers combining the output of neurons from the previous layer into a single neuron to the next layer [56],[10] The most common approach used in pooling 2.3 is max pooling uses the maximum value of the output of neurons. Another differention of pooling is average , which uses the average value of each neuron at the prior layer and outputs the pooled neuron [11].

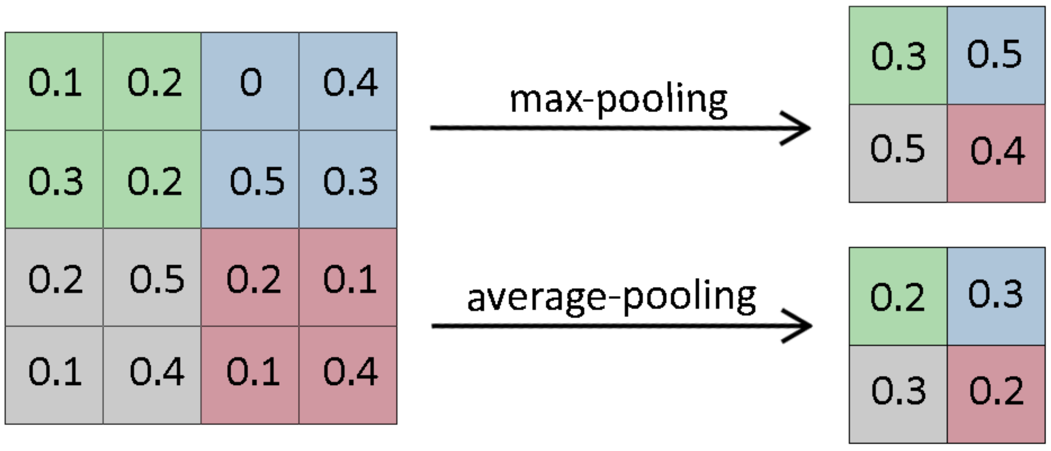


FIGURE 2.3: Pooling Comparison

2.3.3 Activation Function

In artificial neural networks, the activation function of a node defines the output of that node given an input or set of inputs. A standard computer chip circuit can be seen as a digital network of activation functions that can be "ON" (1) or "OFF" (0), depending on input. This is similar to the behavior of the linear perceptron in neural networks. However, only nonlinear activation functions allow such networks to compute nontrivial problems using only a small number of nodes. In artificial neural networks, this function is also called the transfer function.

Activation functions are used to determine the firing of neurons in a neural network. Given a linear combination of inputs and weights from the previous layer, the activation function controls how we'll process and pass that information on to the next layer.

An ideal activation function is both nonlinear and differentiable. The nonlinear behavior of an activation function allows our neural network to learn nonlinear relationships in the data. Differentiability is important because it allows us to backpropagate the model's error when training to optimize the weights.

Perceptron

While 2.1 is the original activation first developed when neural networks were invented, it is no longer used in neural network architectures because it's incompatible with backpropagation. Backpropagation allows us to find the optimal weights for our model using a version of (mainly stochastic) gradient descent. Unfortunately, the derivative of a perceptron activation function cannot be used to update the weights. . The step function is not "convex" and thus you cannot find a local min using Gradient Descent[67].

$$f(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases} \tag{2.1}$$

The sigmoid 2.2 function is commonly used when training CNNs, 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.

Most commonly sigmoid used is 2.2 2.5.There are several sigmoid such as tanh 2.3.3 2.6 and arctan .

Recall that we included the derivative of the activation function in calculating the "error" term for each layer in the backpropagation algorithm. The maximum value of the derivative for the sigmoid function is 0.25, thus, and you progress backwards each layer in backpropagation you're reducing the size of your "error" by at least 75 at each layer. This ends up limiting our ability to change the weights in layers close to the input layer for deep networks because so many of terms multiplied together in the derivative chain are less than or equal to 0.25.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

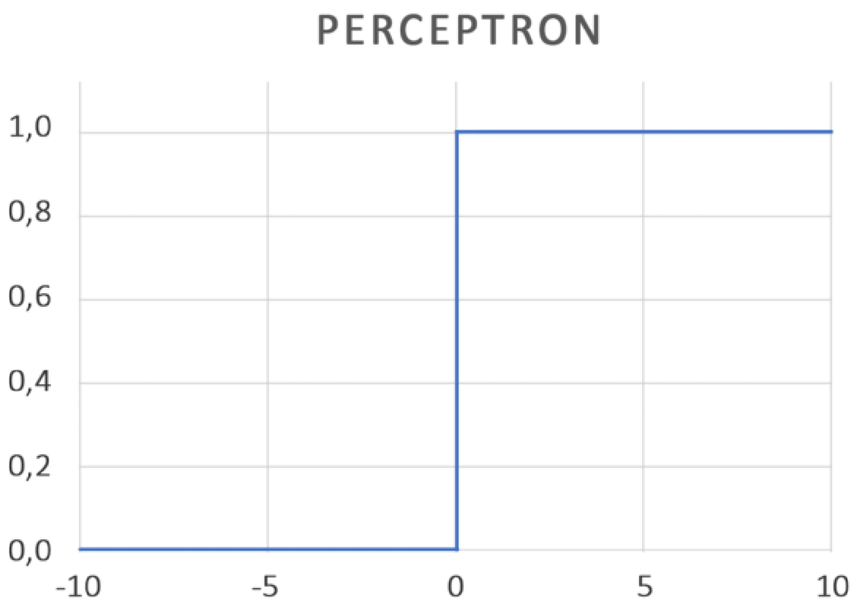


FIGURE 2.4: Perceptron

$$\tanh(\mathrm{x}) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

(2.3)

ReLU (rectified linear unit)

This is 2.4 one of the most popularly used activation functions of 2018. Due to its popularity, a number of variants have been proposed that provide an incremental benefit over standard ReLUs.

$$f(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases} \tag{2.4}$$

Softmax

The softmax function 4.2 is commonly used as the output activation function for

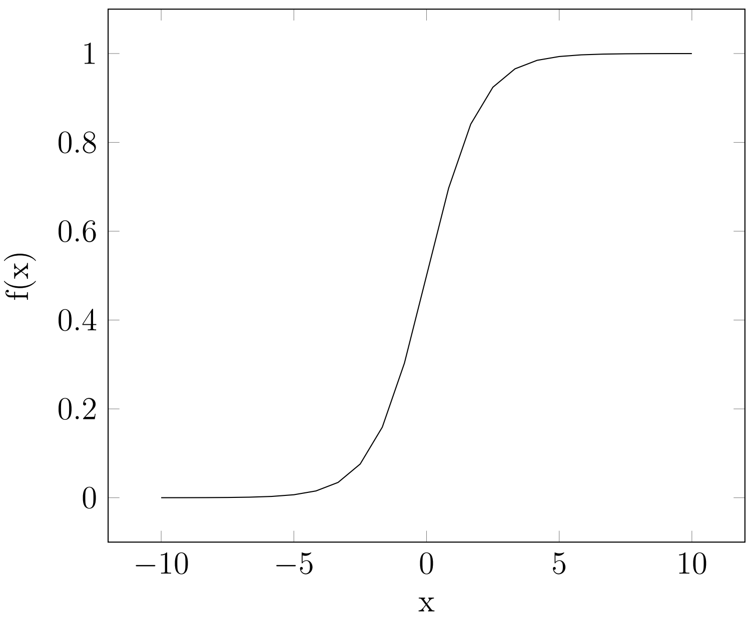


Figure 2.5: Sigmoid Function

multi-class classification because it scales the preceding inputs from a range between 0 and 1 and normalizes the output layer so that the sum of all output neurons is equal to one. As a result, we can consider the softmax function as a categorical probability distribution. This allows you to communicate a degree of confidence in your class predictions.

$$f_i(x) = \frac{e^{x_i}}{\sum_{j=1}^{J}} \tag{2.5}$$

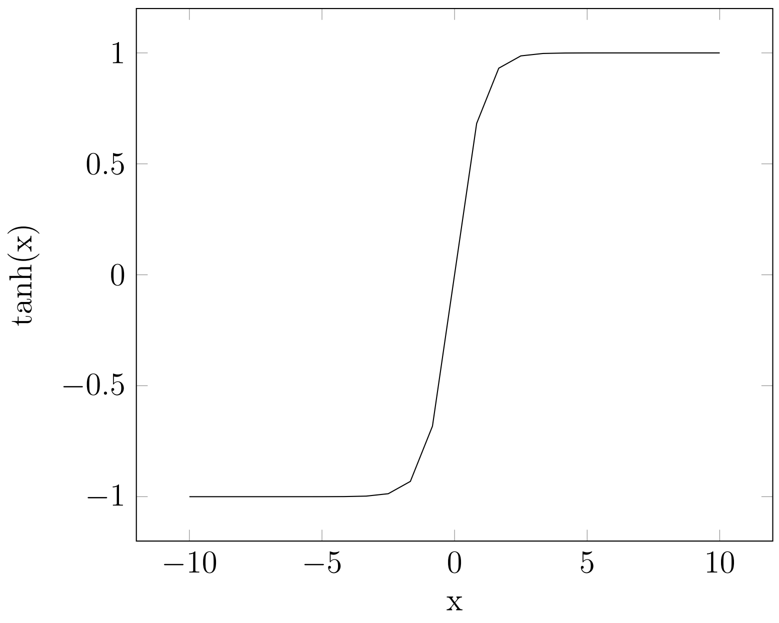


FIGURE 2.6: tanh(x)

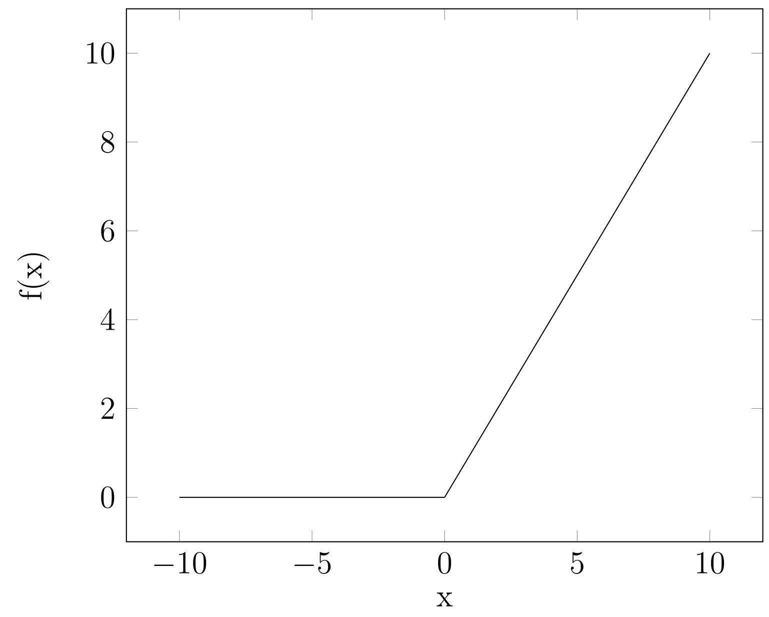


FIGURE 2.7: ReLU

2.4 Typical Architecture of a 1-Dimensional CNN

In this section, we analyze the Convolution Neural Network on which we'll study and worked on. The structure, informations, and the document below have been obtained after approval by K. Tsagtakakis. A typical 1-D CNN 2.8 is structured in a sequential manner, layer by layer, using a variety of different layer types. The foundational layer of a CNN is the Convolutional Layer. Given as an input vector of size (1 x N) and a trainable filter (1 x K), the convolution operation of the two entities will result in a new output vector with a size (1 x M), where $M = N - K + 1$. The value of M may vary based on the stride of the operation of convolution, with bigger strides leading to smaller outputs. In the entirety of this thesis, we assume the generic case of a stride value equal to 1. The trainable parameters of the network (incorporated in the filter), are initialized randomly [6] and, therefore, are totally unreliable, but as the training of the network advances, through the process of back-propagation [15], they are essentially optimized and are able to capture interesting features from the given inputs. The parameters(i.e.weights) of a certain kernel are considered to be shared [43], in the aspect that the same weights can be used throughout the convolution of the entirety of the input. This can consequently lead to a drastic decrease in the number of weights, enhancing the ability of the network to generalize and adding to its total robustness against over-fitting. To ensure that all different features can be captured in the process, more than one filters can be actually used. More specifically to be able to compose high level features we need to extract more than one feature from each input "pixel".

2.4.1 Convolutional Layer

In more difficult problems, using one Convolutional Layer is insufficient, if we want to construct a reliable and more complex solution. A deeper architecture, able to derive more detailed characteristics from the training examples, is a necessity. To cope with this issue, a non-linear function can be interjected between adjacent Convolutional Layers, enabling the network to act as a universal function approximator [34]. Typical choices for the non-linear function (known as activation function) include the logistic (sigmoid) function, the hyperbolic tangent (tanh) and the Rectified Linear Unit (ReLU). The most common choice in CNNs is ReLU (f(x) = max(0,x)) and its variations [30]. Compared to the cases of the sigmoid and hyperbolic tangent functions, the rectifier possesses the advantage that it is easier to compute (as well as its gradient) and is resistant to saturation conditions [56], rendering the training process much faster and less likely to suffer from the problem of vanishing gradients [65].

2.4.2 Fully-Connected Layer

Finally, one or more Fully-Connected Layers are typically introduced as the final layers of the CNN, committed to the task of the supervised classification. A Fully-Connected Layer is a typical layer met in Multilayer Perceptron and as the name implies, all its neuronal nodes are connected with all the neurons of the previous layer leading to a very dense connectivity. Given the fact that the output neurons of the CNN correspond to the unique classes of the selected problem, each of these neurons must have a complete view of the highest-order features extracted by the deepest Convolutional Layer, meaning that they must be necessarily associated with each of these features.

2.4.3 Final Classification

The final classification step is performed using the multiclass generalization of Logistic Regression known as Softmax Regression. Softmax Regression is based on the exploitation of the probabilistic characteristics of the normalized exponential (softmax) function below where x is the input of the Fully-Connected Layer, θ_j are the parameters that correspond to a certain class w_j and W is the total number of the distinct classes related to the problem at hand. It is fairly obvious that the softmax function reflects an estimation of the normalized probability of each class w_j, to be predicted as the correct class. As deduced from the previous equation, each of these probabilities can take values in the range of [0,1] and obviously, they all add up to the value of 1. This probabilistic approach composes a good reason for the transformation of the examined problem to a classification task, rendering possible to quantify the level of confidence for each estimation and providing a clearer view on what has been misconstrued in the case of mis-classification.

2.4.4 Pooling Layer

The use of Pooling Layers has been excluded from the pipeline, given the fact that pooling is considered, among others, a great method of rendering the network invariant to small changes of the initial input. This is a very important property in image classification, but in our case, these translations of the original rest-frame SEDs, almost define the different redshifted states. By using pooling, we suppress these transformations, “crippling” the network’s ability to identify each different redshift.

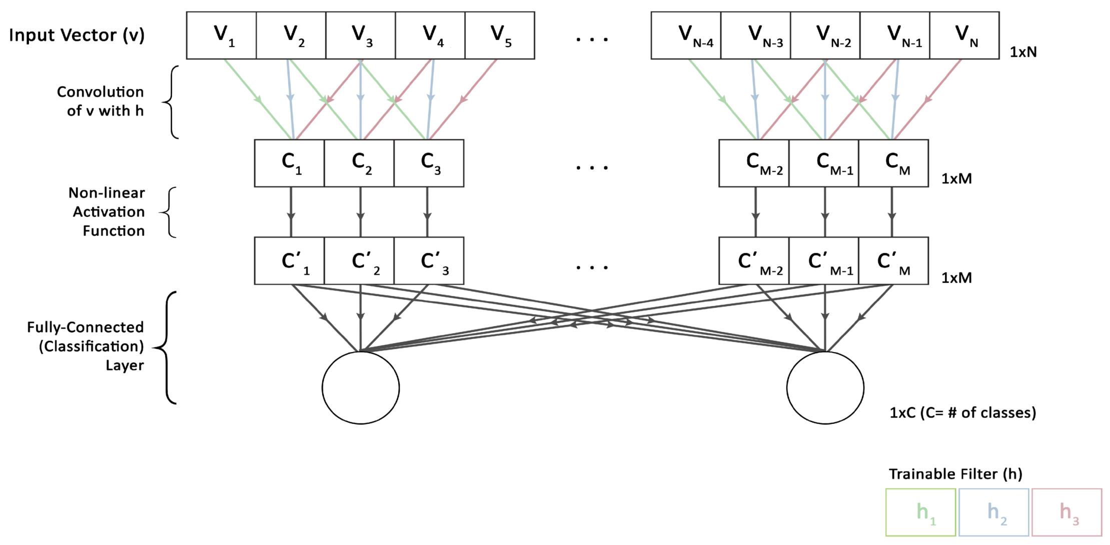

Input Vector (v)
V_1 V_2 V_3 V_4 V_5 . . . V_{N-4} V_{N-3} V_{N-2} V_{N-1} V_N
1xN
Convolution of v with h
C_1 C_2 C_3 . . . C_{M-2} C_{M-1} C_M
1xM
Non-linear Activation Function
C'_1 C'_2 C'_3 . . . C'_{M-2} C'_{M-1} C'_M
1xM
Fully-Connected (Classification) Layer
1xC (C= # of classes)
Trainable Filter (h)
h_1 h_2 h_3

FIGURE 2.8: Simple 1-Dimensional CNN: The input vector v is convolved with a trainable filter h (with a stride equal to 1), resulting in an output vector of size M = N - 2. Subsequently, a non-linear transfer function (typically ReLU) is applied element-wise on the output vector preserving its original size. Finally, a fully-connected, supervised layer is used for the task of classification. The number of the output neurons(C) is equal to the number of the distinct classes of the formulated problem (800 classes in our case).

Chapter 3

Related Work

3.1 Google Brain Project

TensorFlow is an open-source software library for data-flow programming across a range of tasks. It is an extended math library, developed specifically for machine learning applications such as Deep Neural Networks.[49] It is developed and used for both production and research at Google Labs, often extended its closed-source predecessor, DistBelief. TensorFlow was initially inspired and developed by the Google Brain team for internal Google use. It was released under the Apache 2.0 open source license on November 9, 2015, [73].

3.1.1 DistBelief

During 2011, Google Brain built DistBelief as a machine learning system based on Deep Learning neural networks. It has shown rapid growth across different Alphabet companies in both commercial and research applications. Google hired multiple computer scientists, including Jeff Dean, to simplify and reconstruct the base of DistBelief into a faster, more reliable and robust application library, which became TensorFlow. In 2009, Google Brain team, led by Geoffrey Hinton, had developed generalized back-propagation and other important improvements which allowed generation of neural networks to grow and achieve higher performance i.e. a significant 25% reduction in the error rate in speech recognition.

3.1.2 TensorFlow

TensorFlow [75] was originally a major project developed by researchers and engineers working on the Google Brain Team. They collaborate with Google's Machine Intelligence research organization for the purpose of conducting machine learning and deep neural networks research. The results of this project was applicable to a number of other domains, as well, says Google. TensorFlow is Google Brain's second-generation system. Version 1.0.0 was released on 2017 [74], while the reference implementation is able to run on single devices. Moreover, TensorFlow can run on multiple CPUs and GPUs (with optional CUDA extensions for general-purpose computing on graphics processing units). Its flexible architecture allows deployment of computation across a variety of systems (CPUs, GPUs, TPUs). TensorFlow computations could be expressed as stateful data flow graphs. The name TensorFlow derives from the operations that such neural networks perform on multidimensional data arrays. These arrays are referred to as "tensors". In June 2016, Dean stated that 1,500 repositories on GitHub mentioned TensorFlow, of which only 5 were from Google.

3.1.3 Tensor Processing Unit (TPU)

In May 2016, Google announced its Tensor processing unit (TPU), an application-specific integrated circuit (ASIC) developed specifically for machine learning and optimized for the use of TensorFlow. TPU is a programmable AI accelerator designed to provide high throughput of low-precision arithmetic (e.g., 8-bit), and oriented toward inference or running models rather than training them. Google announced, that they had been running TPUs inside their data centers for more than a year leading to 30x-80x higher TOPS/Watt compared to contemporary CPU and GPU [57]. Afterwards, it mentioned that they were able to deliver an order of magnitude better "optimized" performance per watt for machine learning.[39]

In May 2017, Google announced the evolution of the first generation, as well as the availability of the TPUs in Google Compute Engine [36]. The second-generation TPUs deliver up to 180 teraflops of performance, and when organized into clusters of 64 TPUs, provide up to 11.5 petaflops. In February 2018, Google announced that they were developed TPUs to be available in beta on the Google Cloud Platform.

3.2 GPU approach

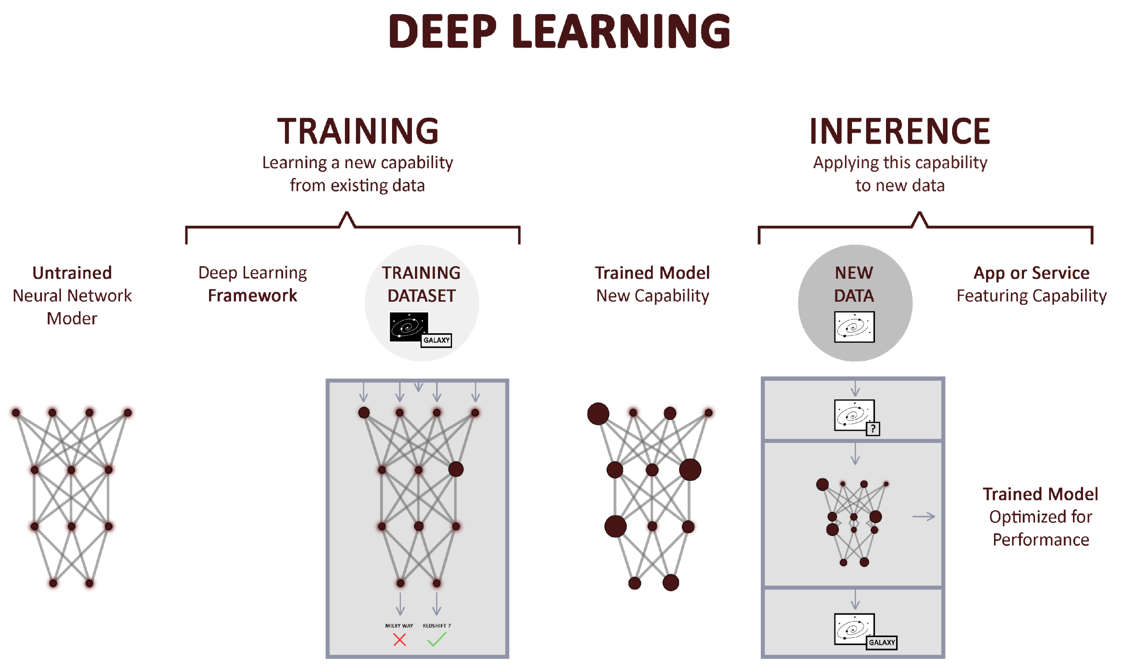
Deep learning frameworks allow researchers to develop and explore Convolutional Neural Networks (CNNs) and other Deep Neural Networks (DNNs), while delivering high throughput for both research and industrial deployment. The NVIDIA Deep Learning SDK accelerates deep learning frameworks such as Caffe, TensorFlow, Theano and Torch as well as many other machine learning applications. Neural Network applications run faster on GPUs and scale across multiple GPUs within a single node. To use the frameworks with GPUs for Convolutional Neural Network training and inference processes, NVIDIA provides toolbox and libraries such as, cuDNN and TensorRT respectively. cuDNN and TensorRT provide highly tuned implementations for standard operations in Deep Learning such as convolution, pooling, normalization, and activation functions.

3.3 The FPGA perspective

In recent years, modern Convolutional Neural Networks were being over-developed, but the biggest challenge was getting them to work efficiently. That meant that accuracy, speed during training and energy efficiency were the top priorities. Community mainly focuses on reducing the operands on training and inference. Orthogonal and complementary techniques for reducing redundancies like weight compression, pruning techniques [71] and compact architectures [51] are impressively efficient and they were proposed in the past years.

Until recently, the use of low-precision networks from fixed-point, lower floating points format to binary, in the extreme case, was believed to be highly destructive to the network performance [53] during training and inference procedure. Contrary by showing that good accuracy performance could be achieved (in training) even if a network was binarized to +-1 [16]. This was implemented using Expectation Back Propagation (EBP), a variational Bayesian approach, which infers networks with bi-Binarized Neural Networks: Training Neural Networks with Weights and Activations. Constrained to +-1 binary weights and neurons by updating the posterior distributions over the weights. These distributions are updated by differentiating their parameters (e.g., mean values, average values etc.) through the back propagation (BP) procedure. Implemented a fully binary network at run-time using a very similar approach to EBP, showing significant results in energy efficiency [72]. The drawback of EBP is that the binarized parameters were only used during the inference procedure.

Both procedures are very important and they started hierarchical separated. The first is a prerequisite for the second to start operating. The trend in recent years is to run together.


DEEP LEARNING
TRAINING
Learning a new capability from existing data
INFERENCE
Applying this capability to new data
Untrained Neural Network Moder
Deep Learning Framework
TRAINING DATASET
GALAXY
MILKY WAY
REDSHIFT ?
Trained Model New Capability
NEW DATA
?
GALAXY
App or Service Featuring Capability
Trained Model Optimized for Performance

3.3.1 Training

Training as a process is computationally intensive and requires a lot of resources in hardware implementations. On the other hand, it is an one-time procedure (not always), but it requires a lot of time to get a high-performance behavior (accuracy).

3.3.2 Weight Reduction

During the past years methods were proposed to train DNNs with binary weights (BC) and activations (BNN) successively [52] [51] . Initially they add noises to weights and activations as a input of regularization but gradients are calculated with real-valued variables, suggesting that high precision data-types formats accumulation is likely required for Stochastic Gradient Descent optimization. Many researchers have also explored training neural networks directly with fixed-point weights. In 1990 proposed a hardware architecture for on-chip learning with fixed-point operations. More recently, in [51], the authors train neural networks with 3 different data types (floating-point, fixed-point and dynamic fixed- point formats) demonstrating and compare its results. [23] demonstrate a Convolutional Neural Network training using 16-bit fixed-point weights rounded with stochastic scheme. XNOR-Net [56] has proposed a filter-wise scaling factor for weights trying to improve the performance of fixed point. XNOR-Net implements efficiently convolution operations using XNOR logical units and bit-count operations. However, these high-precision factors are calculated simultaneously during training, which generally aggravates the training effort. In TWN [20] and [68] were proposed two symmetric thresholds to retain the weights to be ternary-valued: -1,0,+1. They came up with a trade-off between model expressive ability and complexity of the network.

Computional Reduction

DoReFa-Net [69] quantizes gradients to low-bitwidth floating-point formats with static discrete states in the backward pass. Another quantization of gradient updates to ternary values proposed in [78] to reduce the overhead of gradient synchronization in distributed training. Nevertheless, in DoReFa-Net and TernGrad the weights were stored and updated with single-precision float during training like formal works. Besides, the quantization of batch normalization and its derivative were ignored. Thus, the overall computation graph for the training process is still presented with float(32 bit-width) and more complex with external quantization. Generally, it is impossible to apply DoReFa-Net training in an integer-based hardware directly. Therefore, it could lead to a potential opportunity to explore high-dimensional discrete spaces with discrete gradient descent.

3.3.3 Inference

As far as the part of the inference is concerned, which is a continuous procedure, while the training has been preceded. It is important to be able to achieve high performance (low error-rate), and throughput. Furthermore, an important factor in Hardware implementations is the energy efficiency.

Weight Reduction

The data type precision of weights and activations plays a major role in determining the speed accuracy and energy efficiency of any CNN implementations in hardware or software. Plethora of research focuses on how to efficiently replace the 32-bit floating-point data with reduced precision data for CNN inference. Several data-types have been proposed such as [28] representing both weights and activations using low-bit floating point, i.e., single or half. However, it is well known that the fixed-point arithmetic is much efficient than floating-point arithmetic. This state direct most research focuses on fixed-point quantization. Many papers present the impacts of different fixed-point rounding formats on the accuracy of various benchmark network [44]. Thereinafter researchers demonstrate that targeting the minimum required data precision not only varies across different networks but also across different layers of the same network [59]. [17] propose a fixed-point quantization technique to approaching the optimal data precision (range, resolution) for all layers of a network. [60] present a framework Ristretto for fixed-point quantization and re-training of CNN based on [80].

Re-training approaches

Many other approaches for memory compression of neural networks have been explored. [9] propose a combination of network pruning, weight quantization during training and compression based on Huffman coding to reduce the VGG-16 network size by 49X. In [84], store both 8-bit quantized floating-point weights and 32-bit full precision weights. At runtime, compressed weights or floating point weights are randomly fetched in order to reduce memory bandwidth. The continuous research effort to reduce the memory footprint has led to many interesting demonstrations reaching up to 2-bit weights [23] and even binary weights/activations [53]. [69] demonstrate AlexNet training with 1-bit weights, 2-bit activations and 6-bit gradients. These techniques require additional re-training of the network and can result in sub-optimal accuracy.

3.4 Convolutional Neural Networks for Spectroscopic Red-shift Estimation on Euclid Data

The Convolutional Neural Network was developed by FORTH researchers (Dr. Tsagatakis and his team). The following section has been taken from the paper of Dr. Tsagatakis (This is on submission stage) with his permission. [63] Modern astrophysical cosmological researches seek answers to questions like "what is the distribution of dark energy and dark matter in the Universe?" [7], [18]. In this paper, there was an extended study of performing accurate redshift estimation using realistic spectroscopic observations, modeled after Euclid. Redshift estimation is considered to be a regression task, given the fact that a galaxy redshift (z) can be measured as a non-negative real-valued number (with zero corresponding to the rest-frame). Given the specific characteristics of Euclid, we can focus our study on the redshift range of detectable galaxies. Subsequently, we can restrict the precision of our estimations to match the resolution of the spectroscopic instrument, meaning that split the chosen into evenly sized slots equal to Euclid 's required resolution. Hence, we can transform the problem at a regression task to a classification task using a set of ordinal classes, with each class corresponding to a different slot, and accordingly, we can utilize a classification model (Convolutional Neural Networks in our case) instead of a regression algorithm.

Training and Evaluation

The simulated dataset that was used is modeled after the upcoming Euclid satellite galaxy survey [62]. The training of the network was implemented in the GPU using TensorFlow tools. As far as the Inference part is concerned, there is space for other dedicated hardware like FPGA because energy efficiency is much more important than throughput in aerospace applications. Below are the results of the training 3.1 and the performance 4.16 of the network.

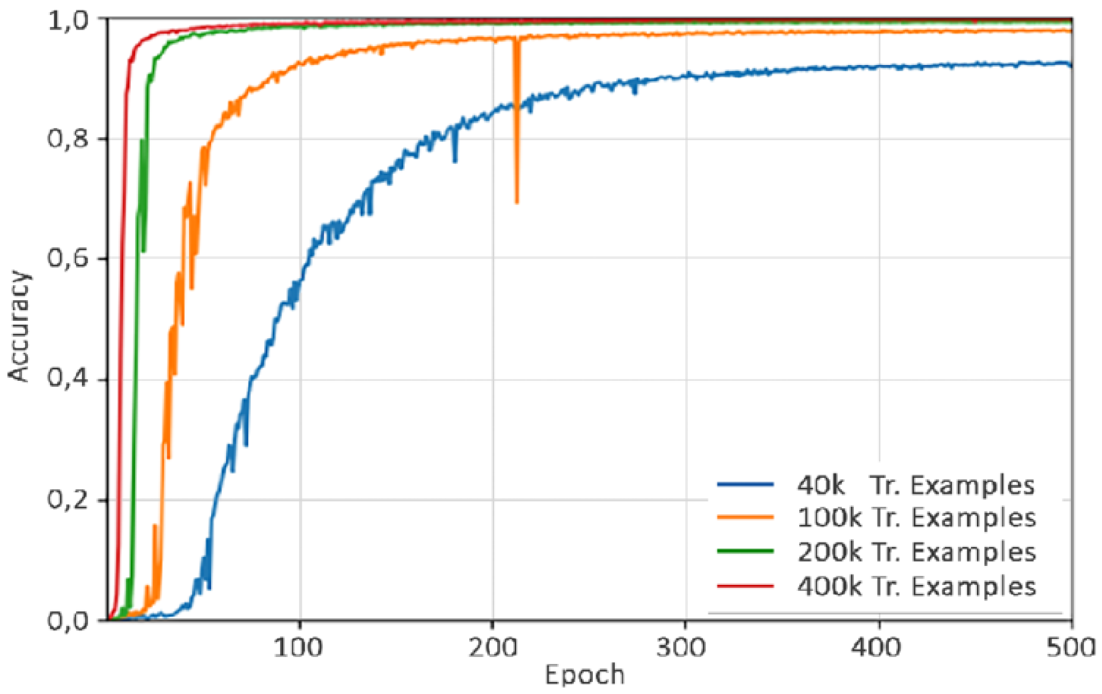


FIGURE 3.1: Accuracy of Euclede: Validation performance of a 3-layered network, using larger and more limited in size datasets. In all cases the training accuracy (not depicted here) can asymptotically reach near to 100% accuracy, after enough epochs. Copyrights from []

TABLE 3.1: Comparison of CPU and GPU training running time

Experiment #	CPU Time (per epoch)	GPU Time (per epoch)
1 4	75 sec.	11 sec.
2 4	735 sec.	107 sec.
3	158 sec. 4	20 sec.

Comparison of CPU and GPU training running time, in 3 different benchmark experiments. In the 1st and the 2nd experiments, we utilize 40,000 and 400,000 training observations, of the idealistic case, in a CNN with 1 Convolutional Layer. In the 3rd case, we deploy 40,000 realistic training examples for the training of a CNN with 3 Convolutional Layers.

3.5 Thesis Approach

In contrast to prior works,[2], [40], [9] and [21] this thesis comes to develop and propose new ways of compressing information into a specific CNN 3.4 implementation. For the proper use and implementation of clustering algorithms, we highlighted the characteristic problems when implemented on CNN applications and propose techniques to limit them. We also propose new compression methods (Pair compression, Hierarchical Clustering) combining it with existing ones, resulting in a large compression rate(36x) with high accuracy performance (0.7%).

Chapter 4

Robustness analysis of CNN

The key to artificial intelligence has always been the representation."

Jeff Hawkins

In this chapter, we intend to analyze the structure of the given Convolution Neural Network [63] 3.4. Afterwards, we aim to model the network in MATLAB [50] and evaluate the results with TensorFlow. The initial CNN has been built and pre-trained in TensorFlow. Therefore starts the inference procedure, where images are fed into the network and this predicts their (target) classes. We built our ToolBox in MATLAB, functions were created for each operation of the network, from importing and formatting image and the filters to the SoftMax layer for the final classification. Thereinafter, we are going to perform a Sensitivity Analysis to explore opportunities for hardware implementation. Finally, several compress and algorithmic optimizations have been studied and tested in order to reduce memory footprint and accelerate the network. Methods, relies on previous studies, such as pruning, different data-types(single floating point, half floating point, fixed point, dynamic-fixed point) and clustering algorithms have been implemented. Finally, we propose new techniques and methods that approach better the problems of already state-of-the-art optimizations.

4.1 MATLAB vs TensorFlow

TensorFlow uses high-level of abstraction, hence we manage to transform to MATLAB code for lower-level understanding of the network. Next step was to extract and compare the results between MATLAB and TensorFlow. This procedure will show us whether a TensorFlow's Convolutional Neural Network (CNN) can be implemented in MATLAB with our toolbox. In our experiments with TensorFlow, we used double-precision floating point (IEEE standard) and single-precision floating point (IEEE standard) for MATLAB implementation. After running inference procedure for both implemantations for 10000 images, the main dataset, and comparing the results we had fully matched top-1 classification.

Below we present the main building blocks of a typical 1-D Convolution Neural Network.

In Algorithm1 we perform an 1-D Convolution with stride 1. This is the first layer of our network. As inputs, it gets a one-dimensional vector(image from the dataset), the kernel which are the pre-trained filters on which the operation of convolution will be based and bias. The output of the layer will be a 2-Dimension vector.

Algorithm 1 Convolution (1-D)

```
1: procedure CONVOLUTION($image, kernel, bias$)
2:     $ImageSize \leftarrow size(image, 1)$            ▷ Size of $1^{st}$ Dimension
3:     $NumOfKernels \leftarrow size(kernel, 2)$        ▷ Number of kernels
4:     $KernelDim1 \leftarrow size(kernel, 1)$          ▷ Size of $1^{st}$ Dimension
5:     for k:=1 to NumOfKernels do
6:         for i:=1 to (ImageSize-KernelSize+1) do
7:             $Conv(i, k) \leftarrow 0$
8:     for k:=1 to NumOfKernels do
9:         for i:=1 to (ImageSize-KernelSize+1) do
10:            for j:=1 to KernelSize do
11:                $Conv(i, k) \leftarrow Conv(i, k) + image(i + j - 1) * kernel(k, j)$
12:            $Conv(i, k) \leftarrow Conv(i, k) + bias(k)$
13:    return $Conv$
```

In 2 we perform an 2-D Convolution with stride 1. This is the second and third layer of our network. As inputs it gets a two-dimensional vector($image_{stage}$ from the previous layers), the kernel which are the pre-trained filters on which the operation of convolution will be based and bias. The output of the layer will be a 2-Dimension vector.

Algorithm 2 Convolution (2-D)

```
1:  procedure CONVOLUTION($image, kernel, bias$)
2:      $ImageSize \leftarrow size(image, 1)$                       ▷ Size of $1^{st}$ Dimension
3:      $NumOfKernels \leftarrow size(kernel, 3)$                    ▷ Number of kernels
4:      $KernelDim1 \leftarrow size(kernel, 1)$                      ▷ Size of $1^{st}$ Dimension
5:      $KernelDim2 \leftarrow size(kernel, 2)$                      ▷ Size of $2^{nd}$ Dimension
6:      for k:=1 to NumOfKernels do
7:          for i:=1 to (ImageSize-KernelSize+1) do
8:              $Conv(i, k) \leftarrow 0$
9:      for k:=1 to NumOfKernels do
10:         for i:=1 to (ImageSize-KernelDim2+1) do
11:             for p:=1 to KernelDim2 do
12:                 for j:=1 to KernelDim1 do
13:                     $Conv(i, k) \leftarrow Conv(i, k) + image(i + j - 1, p) * kernel(k, p, j)$
14:             $Conv(i, k) \leftarrow Conv(i, k) + bias(k)$
15:     return $Conv$
```

In 3 we perform a Matrix Multiplication. This is the final layer of the network. As inputs it gets a flatten vector (transforming from 2 to 1 Dimension), $dense_{kernel}$ which are the pre-trained filters and $dense_{bias}$. The output of the layer will be a 1-Dimension vector in the size of Number of Classes.

Algorithm 3 Fully Connected

```
1:  procedure FULLY CONNECTED($conv, kernel_{dense}, bias$)
2:      $ConvSize \leftarrow size(conv)$                             ▷ Size of Conv
3:      $NoClasses \leftarrow size(kernel_{dense}, 1)$                ▷ Number of Classes
4:      for k:=1 to $NoClasses$ do
5:          $Classes(k) \leftarrow 0$
6:      for k:=1 to NoClasses do
7:          for i:=1 to ConvSize do
8:              $Classes(k) \leftarrow Classes(k) + conv(i) * kernel_{dense}(k, j)$
9:          $Classes(k) \leftarrow Classes(k) + bias(k)$
10:     return $Conv$
```

In 4 we perform ReLU activation function. This layer performed in the output of Convolution Layers.

Algorithm 4 ReLU

```
1: procedure ReLU($conv$)
2:     $ConvSize1 \leftarrow size(conv, 1)$                ▷ Size of $1^{st}$ Conv
3:     $ConvSize2 \leftarrow size(conv, 2)$                ▷ Size of $2^{nd}$ Conv
4:     for i:=1 to ConvSize1 do
5:         for k:=1 to ConvSize2 do
6:             if $conv(i, k) < 0$ then
7:                 $conv(i, k) \leftarrow 0$
8:     return $conv$
```

In 5 we perform SoftMax activation function. This layer performed in the output of Fully Connected Layer. It outputs the target class of the network with a probability distribution.

Algorithm 5 SoftMax

```
1: procedure SoftMax($Classes$)
2:     $NoOfClasses \leftarrow size(Classes)$                ▷ Number of Classes
3:     $sumC \leftarrow 0$
4:     $maxC \leftarrow -\infty$
5:     $posC \leftarrow -1$
6:     for i:=1 to NoOfClasses do
7:         $sumC \leftarrow sumC + e^{Classes(i)}$
8:         if $e^{Classes(i)} > maxC$ then
9:             $maxC \leftarrow e^{Classes(i)}$
10:            $posC \leftarrow i$                ▷ Find the position of the predicted class
11:    $P_{max} \leftarrow maxC/sumC$                ▷ Probality of the max Class
12:    return $P_{max}, posC$
```

4.1.1 Data types

In our experiments, we used 3 different data types: double, single, half precision floating point. All three formats are IEEE Floating Point, the first two are supported by MATLAB while for half Floating Point a mini-toolbox was used by MATH-WORKS. One first step is to understand what floating point datatype actually we need because Input/Output (I/O) has always played a crucial role in computer, industrial applications and especially in FPGAs which are main speed bottleneck. Below in table 4.1 is presented error rate in top-1 classification for 3 different floating point types.

Table 4.1: Error rate

Data type	**Error rate(%)**
double	0
single	0.02
half	0.04

It is easily observed that there isn't significant error rate for lower floating point formats, thus creating space to exploit their use.

4.2 Memory Footprint

An important issue that has been reported several times is memory footprint. Especially for implementing FPGA applications, memory is a performance inhibitor. Most of the time, the application is to fit into internal B-RAMs or even to accommodate a lot. Of course in modern FPGA there is the solution of D-RAM but speed is the main disadvantage. It is important to perceive how weights are distributed in the network stages and find ways to reduce the overall memory footprint. Tables (5.8,4.3) presents memory footprint of the weights and the stages of the image using double floating point.

Table 4.2: Weights Memory Footprint

Layer	**#Weights**	**Footprint**	**Memory(%)**
conv1	144	1.1KB	$6.33 * 10^4$
conv2	2064	16.1KB	$9.2 * 10^3$
conv3	2064	16.1KB	$9.2 * 10^3$
dense	22771200	173.7MB	99.98

Table 4.3: Data Stages Memory Footprint

Stage	**#Data**	**Footprint**	**Memory(%)**
image	1800	14KB	2
conv1	28688	224KB	32
conv2	28576	223.2KB	32
conv3	28464	222.4KB	32
dense	800	6.25KB	0.9

4.3 Weight Distribution

In typical neural networks, there are millions of parameters which define the model and requires a large amount of data to store them. Neural networks are typically over-parameterized, and there is significant redundancy for deep learning models. This results in a waste of both computation and memory. There have been various proposals to remove the redundancy. In our network, we have 22.776.272 parameters, so it is essential to make a weight distribution and analyze its results.

Weights can get values $(-\infty, 0)U(0, +\infty)$, but what we are interested in for our study is the absolute distribution of weights in the histogram chart. In Figures (4.1, 4.2, 4.3) we present the weight distribution of 3 Convolutional Layers grouping them into bins with adjacent values. In Figure 4.4 we present the weight distribution of Fully Connected (Dense) Layer grouping them into bins with adjacent values.

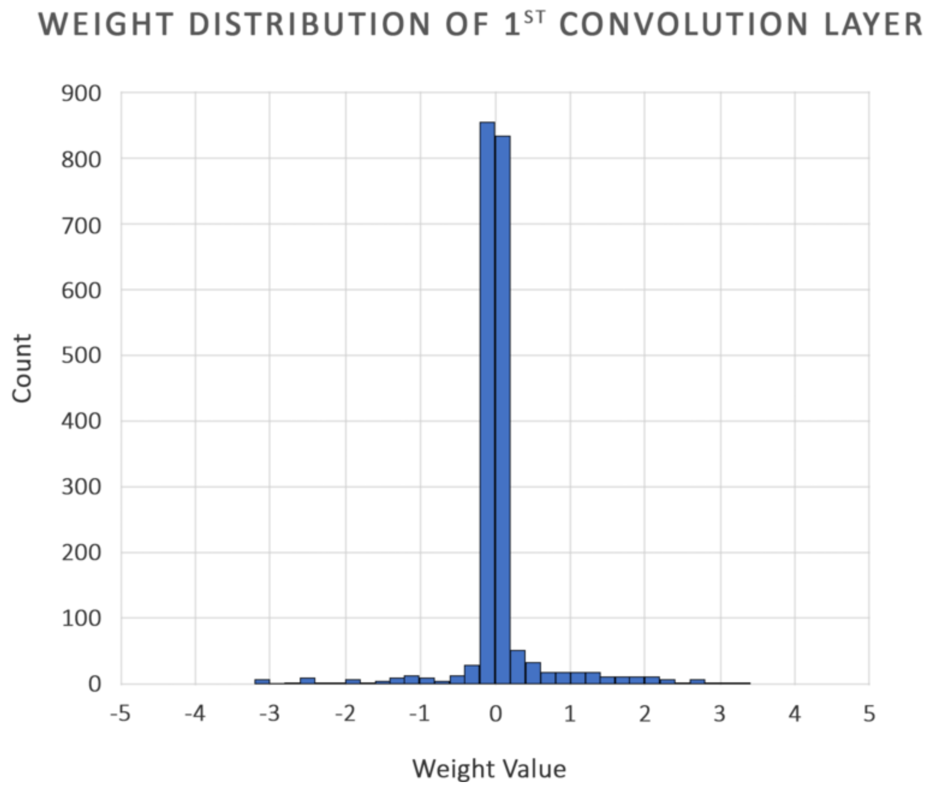


Figure 4.1: Weight Distribution of 1st Convolutional Layer

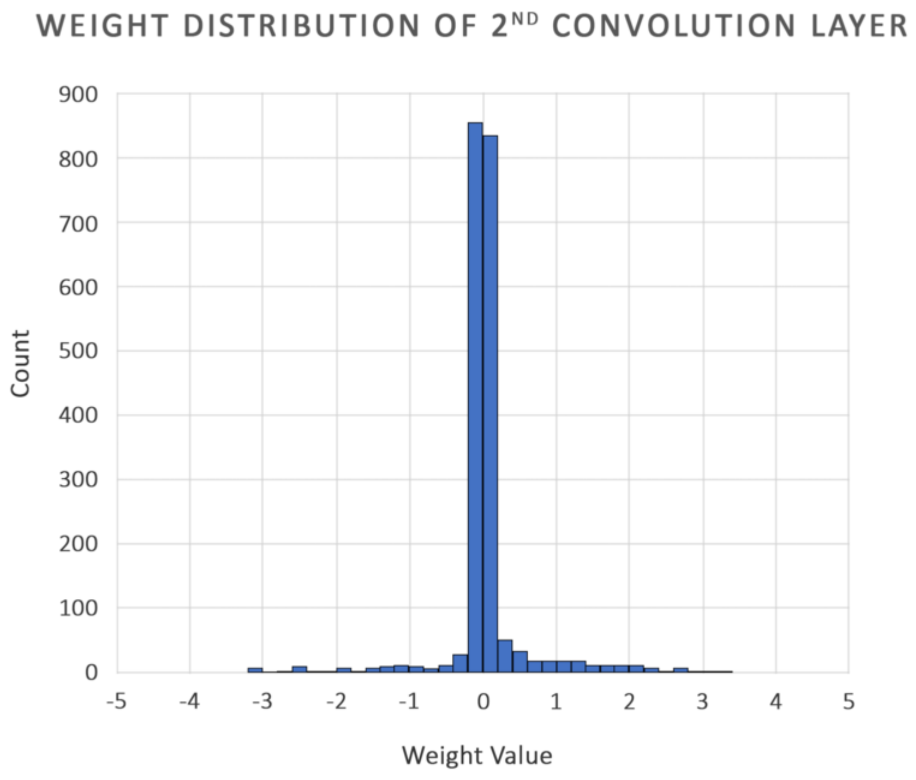


FIGURE 4.2: Weight Distribution of 2nd Convolutional Layer

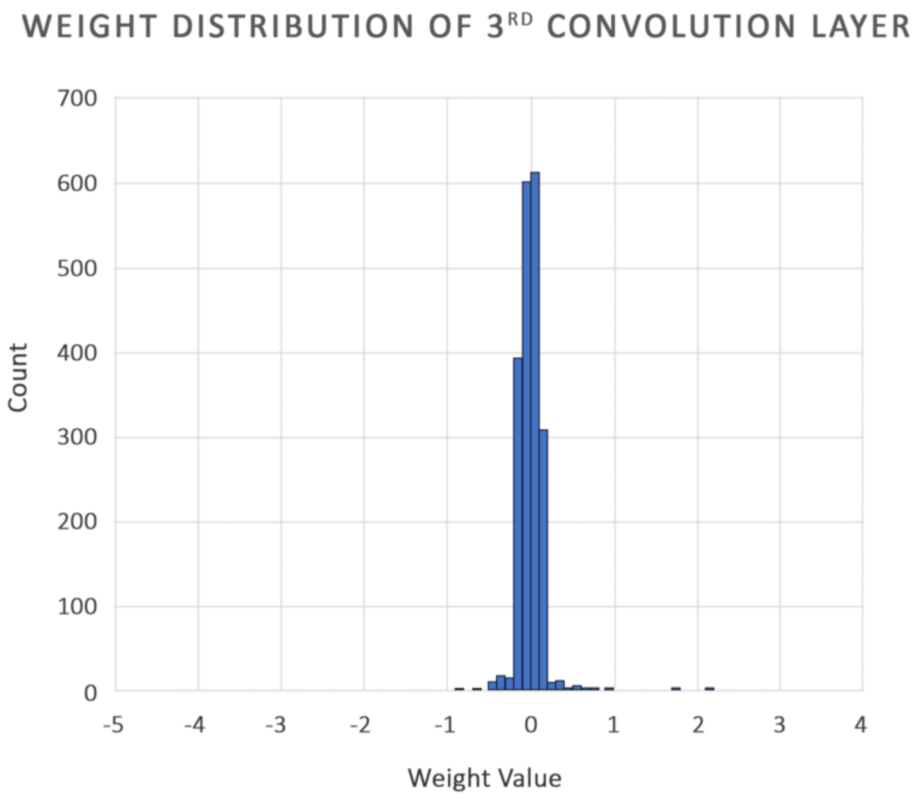


FIGURE 4.3: Weight Distribution of 3rd Convolutional Layer

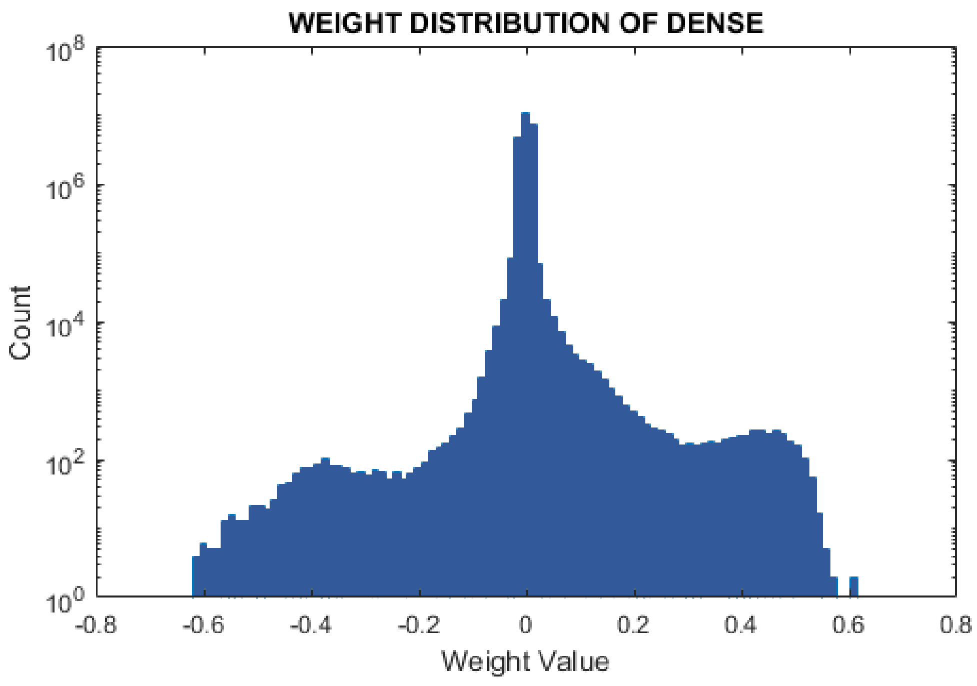


FIGURE 4.4: Weight Distribution of Dense Layer

4.4 Pruning

After weight and data distribution studies a resulting optimization is the pruning. Pruning is a technique in machine learning that reduces the size of the network by removing weights that provide little power to classify instances. We chose a factor and values that are absolute-smaller than this are zeroing. The question is to what extent we could prune values while having the least possible cost in classifying. Afterwards, it is essential to consider pruning only in weights of Fully Connected Layer because contains 99% of the network parameters. After implementation in MATLAB 4.5 for several pruning factors it was observed that there is a trade-off between network accuracy - prune factor. it is easy to perceive that selecting suitable prune factor (0.015) we can achieve high weights reduction (97.78%) having the least possible loss of network accuracy (error rate=0.34%). It is easily perceived that factor =0.015 were selected because it leads to low error rate with high weights reduction.

In Fig 4.6 we present the Weight Distribution of Fully Connected Layer after Pruning.

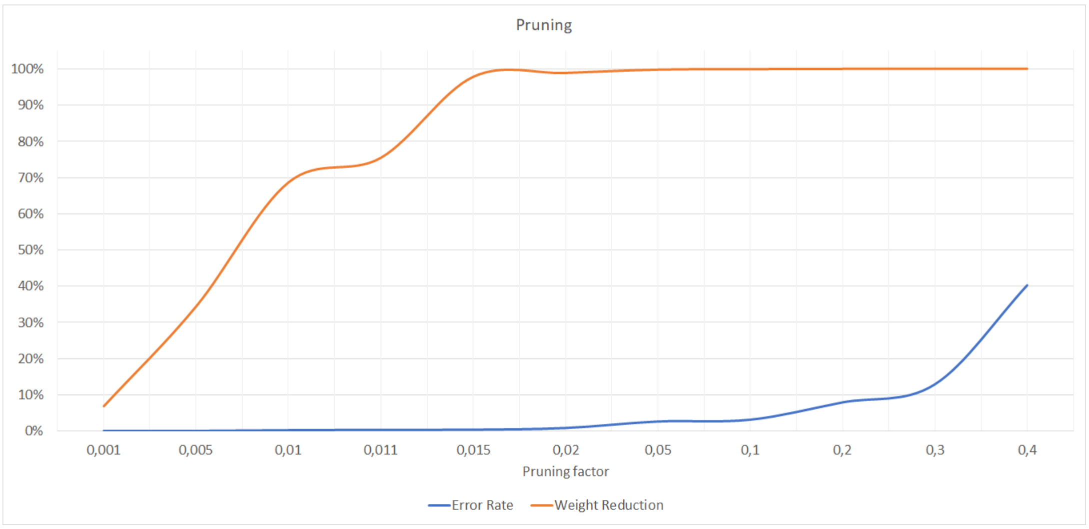


FIGURE 4.5: Pruning Comparison: Weight reduction rate (%): Percentage of weights that were pruned , Pruning Factor: The factor where values are absolute-smaller than this are pruned. , Error rate (%): Percentage of misclassification in top-1 Class.

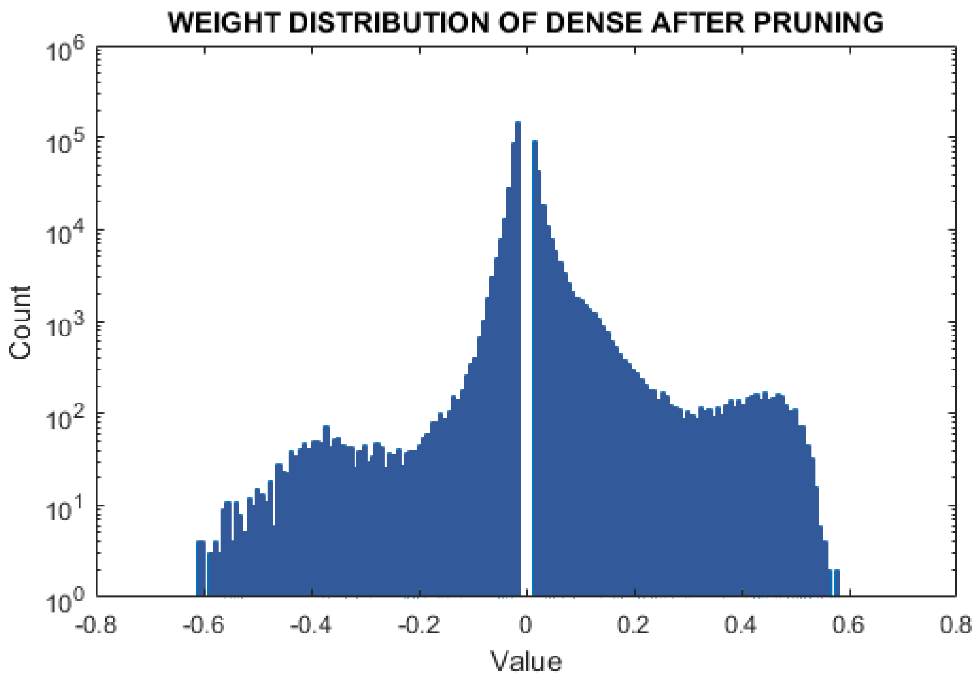


FIGURE 4.6: Weight Distribution of Dense Layer after Pruning: Pruning factor = 0.015 was selected because it leads to low error rate with high weights reduction. More specifically 97.78% weight reduction and 0.34% error rate.

4.4.1 Using Fixed Point

Using floating point was the safest way to guarantee high accuracy and performance on the results. Floating Point solutions in hardware are expensive computing and consume a lot of resources. They also have a big memory footprint. Hence an alternative solution that comes is the use of Fixed Point, where they are more FPGA-friendly computing (TPU uses 8-bit fxed-points). In FPGA designs, fixed-point formats are very efficient if we know beforehand the resolution and range of our data so that we can select the appropriate format. Compared to floating point, fixed point requires less resources in FPGAs (DSPs) and they are less computational complex. In addition, we can reduce the memory footprint to a percentage that the network allows.

Below is the table using various fixed-point format and their impact on performance 4.5. As error we define top-1 miss-classification.

Table 4.4: Fixed point formats

Format	Compression	Error rate (%)
32 bit	2x	0.8
24 bit	2.66x	3.8
16 bit	4	20.01

4.4.2 Dynamic Fixed Point

Using Fixed Point, we observe that there is indeed a significant improvement in memory footprint. Another solution to be considered is the Dynamic Fixed Point, which was first introduced by Williamson in 1991 [5]. The difference is that instead of using a global scaling factor, more can be used depending on the application's needs. The way this idea was applied to the network was to group the layers separately and each to have a different fixed point format using scaling factors.

We have noticed that the dense layer has the largest memory footprint compared to the others. So we tried to find the minimum number of representation bits having as a limit to the correct classification of the top classes. For the rest of the groups, the image is read by 32-bit for high precision, for weights and bias except dense 12-bit is the best lower limit and in addition for the final classification, 32-bit was used for large analysis.

Below is the table of various Dynamic Fixed Point Formats.

Table 4.5: Dynamic Fixed Point

Format	Compression	Error rate (%)
18 bit$_{(1)}$	3.55x	0.8
12 bit$_{(2)}$	5.33x	2.8
10 bit$_{(3)}$	6.4x	4.2
8 bit$_{(4)}$	8x	8.01
7 bit$_{(5)}$	9.14	10.01

1. Dynamic 18-bit : image: 18-bit FP for weights and every image stage except the except output Classification (32bit FP)

2. Dynamic 12-bit : image: 12-bit FP for weights and every image stage except the except output Classification (32bit FP)

3. Dynamic 10-bit :image: 12-bit FP for weights except dense:10-bit FP and for every image stage 12 bit FP except the output Classification (32-bit FP).

4. Dynamic 8-bit : image: 12-bit FP for weights except dense:8-bit FP and for every image stage 12 bit FP except output Classification (32-bit FP).

5. Dynamic 7-bit : image: 12-bit FP for weights except dense:7-bit FP for every image stage 12 bit FP except output Classification (32-bit FP).

4.5 Evaluating Results

In order to test the techniques and methods are followed we used a second stage training set of 2500 images. This data-set will provide us the opportunity to develop these methods and test their behavior. Then to evaluate compression methods and extract the error results we used an inference "unkown" dataset of 10000 images.

4.6 Clustering

Clustering is the task of grouping a set of objects in such a way that objects in the same group (called a centroid) are more similar (in some sense) to each other than to those in other groups (clusters). In CNN clustering works well as a small error occurs while we have large compression of the weights. Network quantization and weight sharing compress the network by reducing the number of bits required to represent each weight. We limit the number of effective weights we need to store by having multiple connections share the same weight, and then fine-tune those shared weights.

4.6.1 Pre-Clustering Techniques

As the clustering process is computational intensive it is important to study and determine the number of different values in the network. This will allow us to really understand how close the values are and how far we are allowed to group together. Afterwards, a function (named Pre-Clustering) is implemented. It gets as arguments sorted weights and a variable named tolerance. Finally, it groups weights where the absolute differences of values are smaller than this tolerance. We sort the initial weights and then weights that have an absolute difference less than the tolerance are grouped. In figure 4.7 and table 4.6 it is presented the number of different weights using Pre-Clustering algorithm for various tolerances.

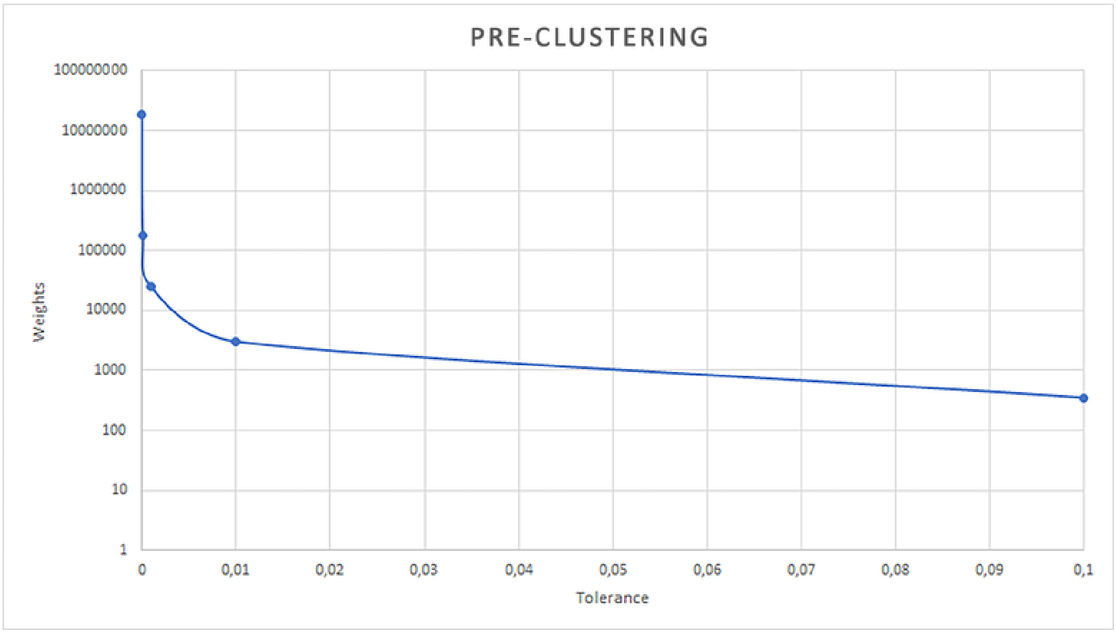


FIGURE 4.7: Pre-Clustering : Number of different weights for several tolerances.

TABLE 4.6: Pre-Clustering Compression Error

Tolerance	**#Weights**	**Frequency (%)**	**Error rate (%)**
0	18615045	81.7	0
10^{-4}	482910	0.8	0
10^{-2}	24995	0.2	0.01
10^{-3}	3047	0.01	0.02
10^{-1}	352	0.002	0.03

$$\#Weights \leq 22771200 \text{ , } Frequency = \frac{\#Weights}{22771200} * 100\%$$

4.6.2 Quantization with Codebook

Using Fixed Point (static or dynamic), we observe that there is indeed a significant improvement in memory footprint but as we reduce bit-width, the accuracy of the network decreases. So we are heading to a hybrid solution, which provides us floating point format for the kernels (using double or single) with much less memory footprint. The idea is to group weights according to their values in k centroids and store their values in a codebook 4.8. Thus, each weight instead of storing its value, it stores the index of the corresponding centroid in a shared code-book. Using this quantization, given k centroids, we only need log_2k bits to encode the index. There are several algorithms to cluster these centroids such as Lloyds, K-means. These optimizations in our network concern dense layer only because this is the main memory bottleneck.

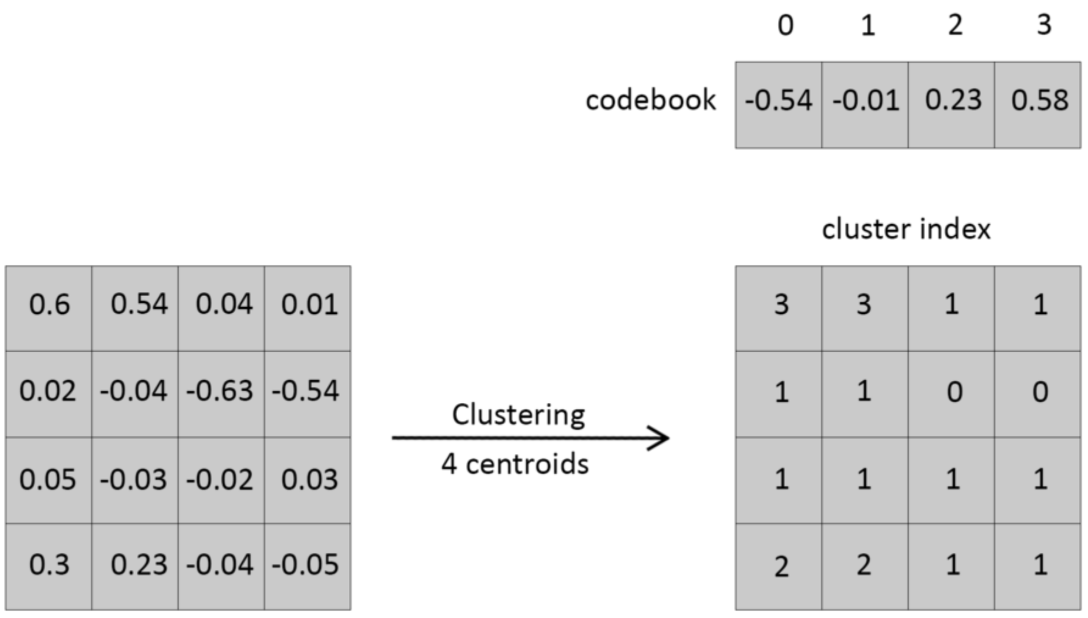


FIGURE 4.8: Clustering : A sample of Clustering 4.

4.6.3 Comparison of different Codebooks

In table 4.6 it is observed that with the pre-clustering algorithm we have a low error rate and at the same time high compression (few centroids). Thus we can start using the Lloyds algorithm from $\lfloor \log_2 352 \rfloor = 8$ and below and see what is the clustering limit of our network. In the following table 4.7 it is presented Lloyds clustering for several centroids. To calculate the compression rate, given k centroids, we only need (4.1) $\log_2(k)$ bits to encode the index. In general, for a network with n connections (weights) to have the only k shared weights (centroids) will result in a compression rate of 4.2:

Below in figure 4.9 we present an example of the use of clustering algortihm with 16 centroids.

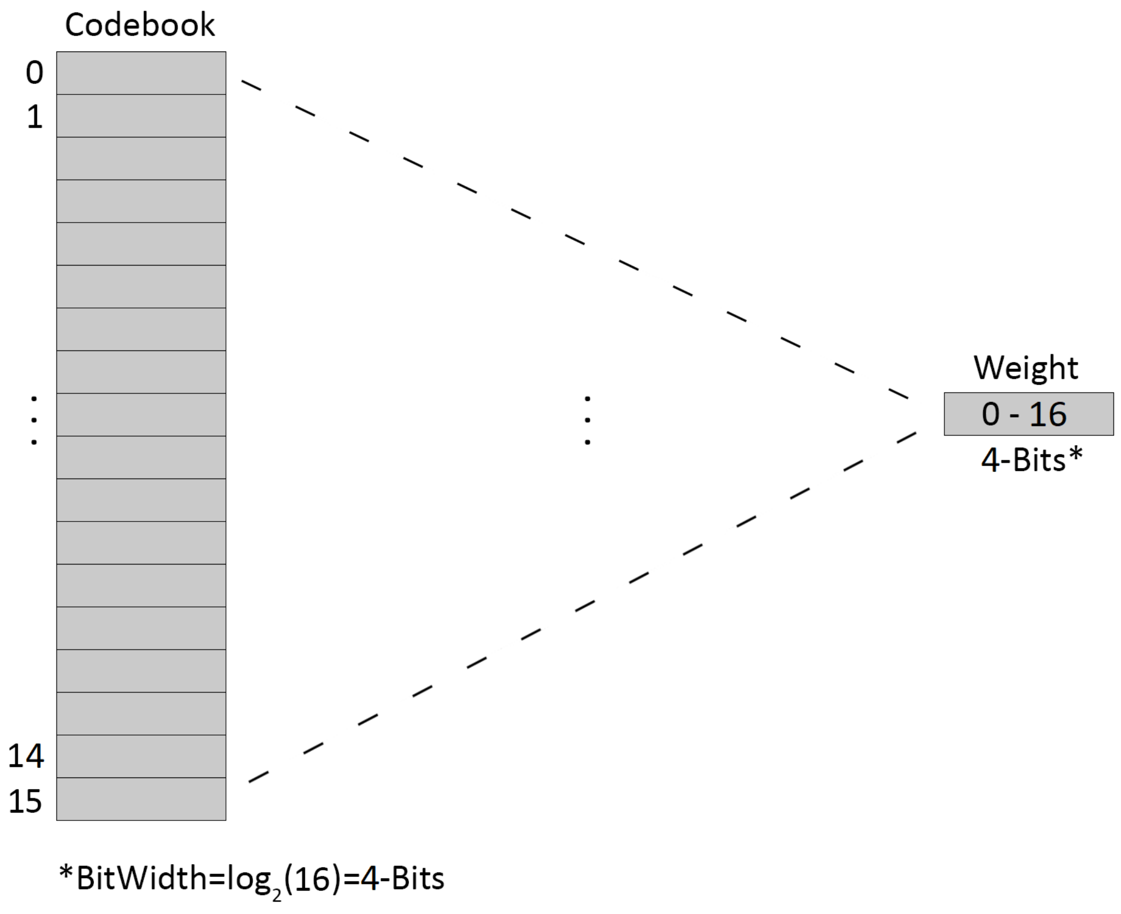


FIGURE 4.9: Clustering-16 example

Stohasting Clustering

The time Complexity of Lloyd's algorithm (and other clustering algorithms) is $O(n*k*d*i)$ [29] [47],where d is the number of different dimensions,n the number of the input dataset, k the number of target centroids and i the number of iterations needed until convergence. The [79] dictates that Lloyd's algorithm is often considered to be of "linear" O(n) complexity in practice. In the worst case scenario is super-polynomial. Hence we realize that when we use large n it takes a long time to complete. Consequently, we assume that it is acceptable to go to an estimated solution. The way to perform this is to introduce the stochasticity into the original dataset. Accordingly, we capture a representative sample of the original data-set (10-30%) stochastically. This can be theoretically supported for its correctness by the Pre-Clustering algorithm that we previously applied and showed us that values are concentrated around specific values.

$$Bit_{Width} = \log_2(k) \tag{4.1}$$

$$Compression = \frac{nb}{nBit_{Width} + kb} \tag{4.2}$$

TABLE 4.7: Lloyds Clustering

#Centroids	Bit_{Width}	**Error rate(%)**	**Compression**
256	8	0.03	8x
128	7	0.09	9.1x
64	6	0.16	10.7x
32	5	0.26	12.8x
16	4	1.37	16x
8	3	4.6	21.3x

$Bit_{Width} = \log_2(k), Compression = \frac{nb}{nBit_{Width}+kb}$

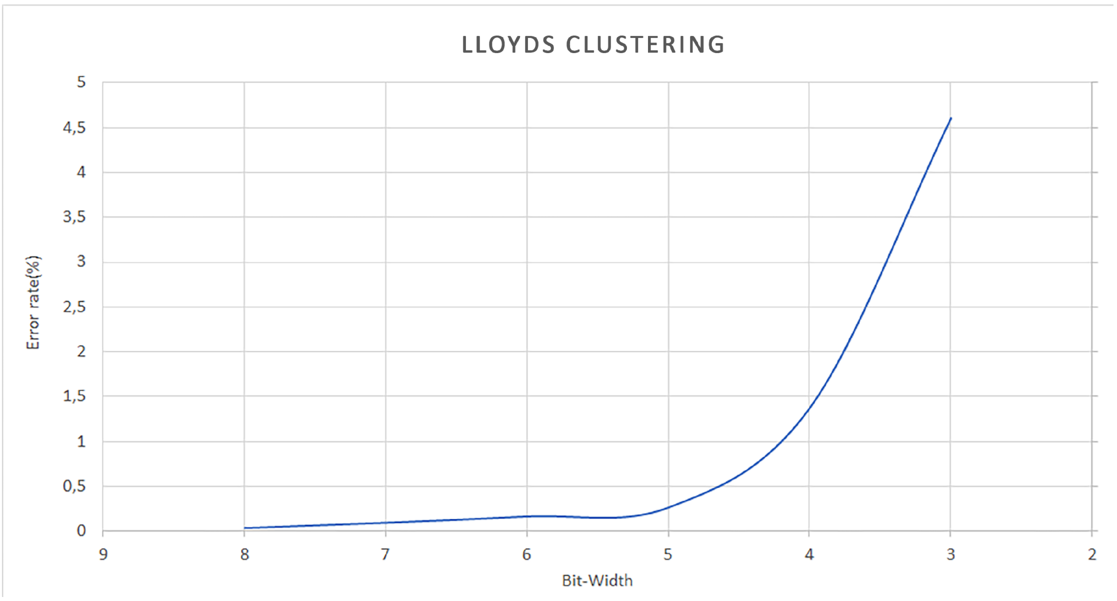


FIGURE 4.10: Lloyds Clustering : Error rate for several bit-widths.

4.6.4 Proposed methods for quantization with Codebook

An ideal case would be to use Lloyds with 16 centroids (ie, 4-bit). We will try to develop some techniques so that we can drop the error at smaller levels. A problem that seems to arise, is that Lloyds is trying to group weights without understanding their differential importance. For example, weight with value 0.69 is much more important for the network than weight with value 0.0023. Larger weights play a more important role than smaller weights (Han et al., 2015), but the density of them are inversely proportional.

Inverse Density: A very important factor is that we can give normalization to the algorithm by giving it the initial codebook. Knowing that density is inversely proportional to the importance of weights, it is necessary to give an initial code-book that takes this into account. So we propose a technique to initialize the code-book, starting from the minimum value and ending up to the maximum, trying to have a big resolution at the absolute big values and as we approach small values to reduce resolution linearly 4.11.

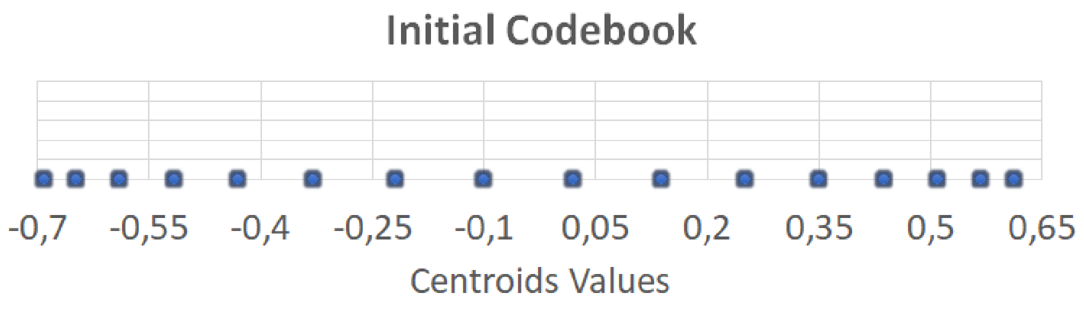


FIGURE 4.11: Inverse Density: Importance => more resolution

Hierarchical Clustering: The second method we propose is trying to solve the same problem from another point of view. As long as we use a larger number of centroids, we increase the resolution across all values (large and small). Thus, we force the algorithm to pay more attention to high values. Then if we apply the clustering algorithm hierarchically we will come up with a better resolution at weights that are of greater importance to us.

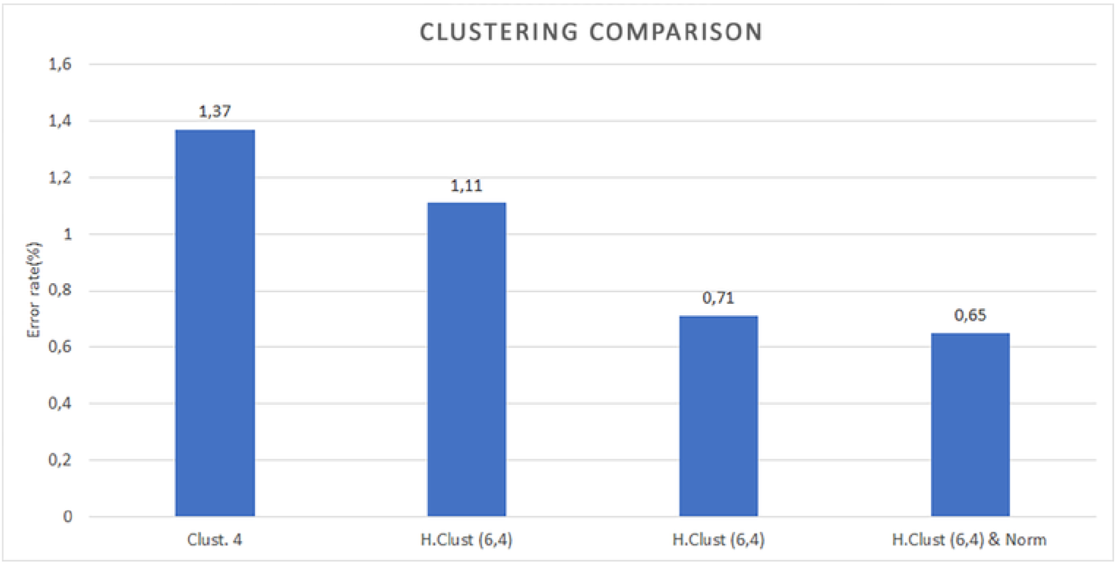


FIGURE 4.12: Clustering 16: Error rate for Clustering methods. Clust. = Clustering , H. Clust. = Hierarchical Clustering

4.7 Pair-Compression

Another technique we proposed for further compression is compression pairing 4.13.The pair compression is a lossy compression method very similar to Vector Quantization method. Knowing that there is a spatial relationship between the weights and weights that are spatial close have similar values, we have the idea of joining two consecutive weights so that we keep the information a single weight. A problem that arises is when the two consecutive weights w_1, w_2 have heteronymous values. The solution comes by storing at the unite weight $w_{1,2}$ their absolute average value keep the sign of w_1 and placing an extra bit for the decompression to understand whether we will keep or not sign for the w_2. Let b the number of bits required for a weight, the compression rate appears below in equation 4.3.

$$Compression_{rate} = \frac{2b}{b+1} \tag{4.3}$$

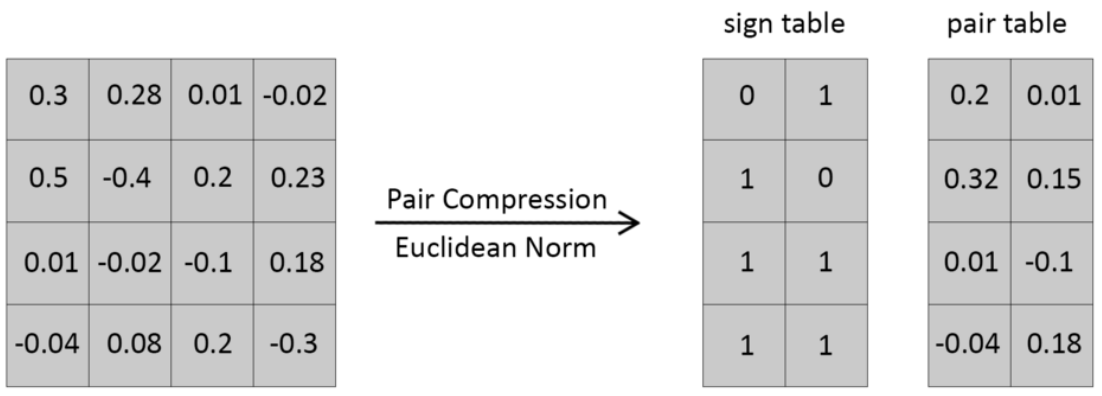


FIGURE 4.13: Pair Compression : An example of Pair Compression.

4.7.1 Proposed Methods and Pair-Compression

The initial error that resulted using the Pair-Compression was 0.38%. Below we propose some techniques that helps Pair-Compression have a better implementation on CNNs leading a further reduction of the error.

Clustering and Pair-Compression: An optimization was to apply the clustering algorithm first so that the weights are pooled into some centroids. In this way, we would help the Pair-Compression unite weights as the sparsity would have already decreased.

Normalization: Another factor in improving the behavior of the algorithm is to normalize the calculation of the united weight. Instead of calculating the average of the absolute values of weights, we will compute the Euclidean norm:

$$Weight_{12} = \sqrt{{Weight_1}^2 + {Weight_2}^2} \tag{4.4}$$

Below in Figure 4.14 we present error rate for the previous techniques.

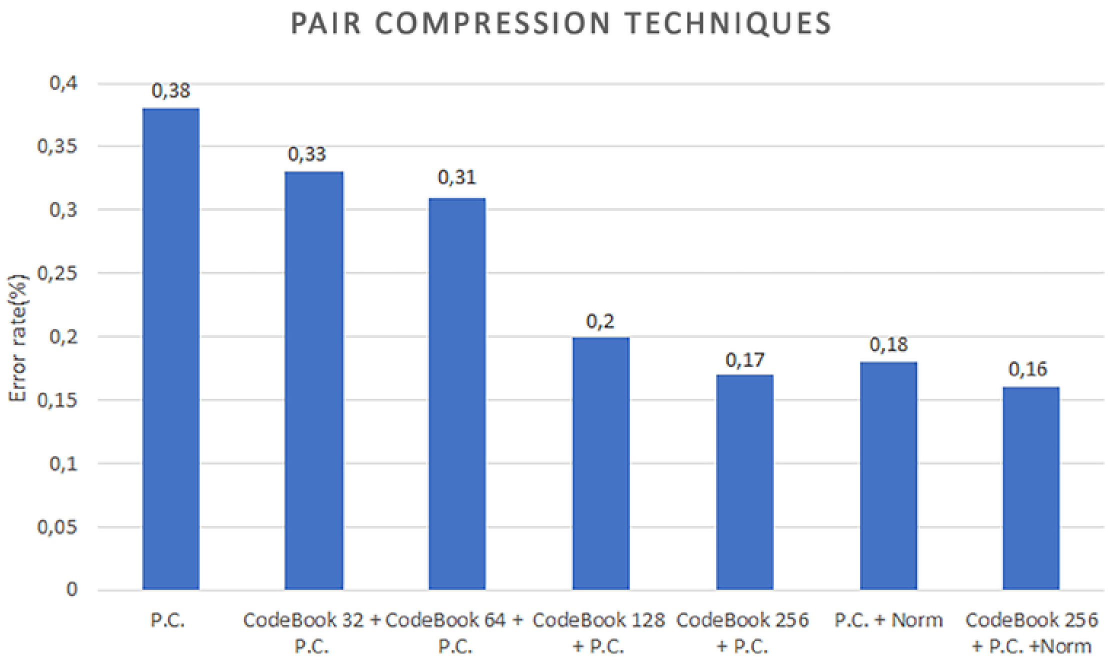


FIGURE 4.14: Pair Compression : Comparison of the above Techniques, P.C. = Pair Compression, Norm = Normalization

4.8 Quad-Compression

The logic of combining consecutive weights can be extended to more than 2. So we suggest Quad Compression 4.15 union 4 consecutive weights w_1, w_2, w_3, w_4 into a union weight w_{union}. The problem that contiguous weights might not have the same sign will be solved using 3-bits. In the same way as before the union weight w_{union} will retain the sign of the w_1 and each of the following 3 bits will show us the sign for w_2, w_3, w_4 during the decompression. Let b the number of bits required for a weight, the compression rate appears below in equation 4.3. Applying the techniques we mentioned before the error reached up to 0.6%

$$Compression_{rate} = \frac{4b}{b+3} \tag{4.5}$$

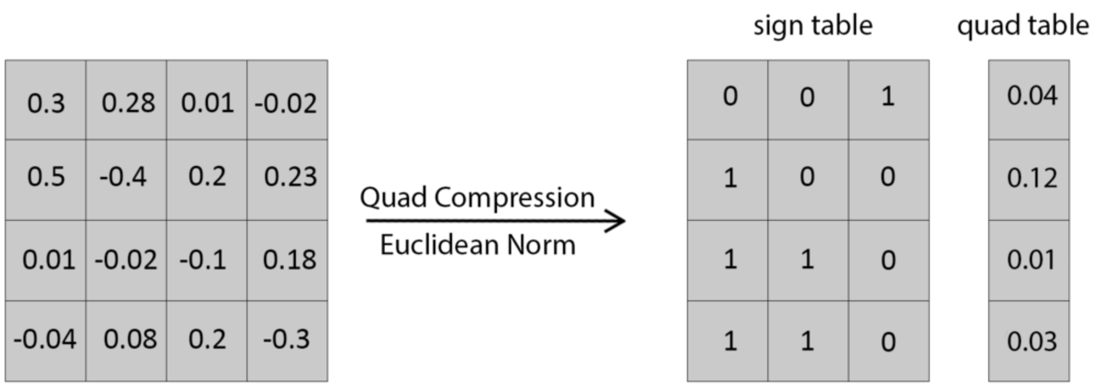


FIGURE 4.15: Quad Compression : An example of Quad Compression.

4.9 Pair Compression and Hierarchical Clustering

In order to be able to use the two main compression methods mentioned above, we need to follow the procedure below. We will first implement a Clustering algorithm with 256 centroids and then Pair Compression. At this point, values from the original centroids have changed, so we will reapply hierarchical clustering to result in the final codebook with 16 centroids.

Below table 4.14 presents final error rate and Compression rate using our proposed Compression methods.

TABLE 4.8: Compression Methods

Method	**Error rate(%)**	**Compression rate**
Clust. 4	1.37	16x
H.Clust. 4	0.65	16x
P.C.& H.Clust. 4	0.62	25.6x
Q.C.& H.Clust. 4	0.76	36.57x

Clust. = Clustering , H. Clust. = Hierarchical Clustering , P.C. = Pair Compression, Q.C. = Quad Compression

Below figure 4.16 presents the weight distribution in CDF format using our proposed Compression methods. We notice that techniques help us to spread the weight distribution to incrementally higher weights, giving them the better resolution.

Below on figures 4.17 4.18 we present the procedure followed to extract the final compressed weights using methods Hierarchical Clustering, Pair or Quad Compression.

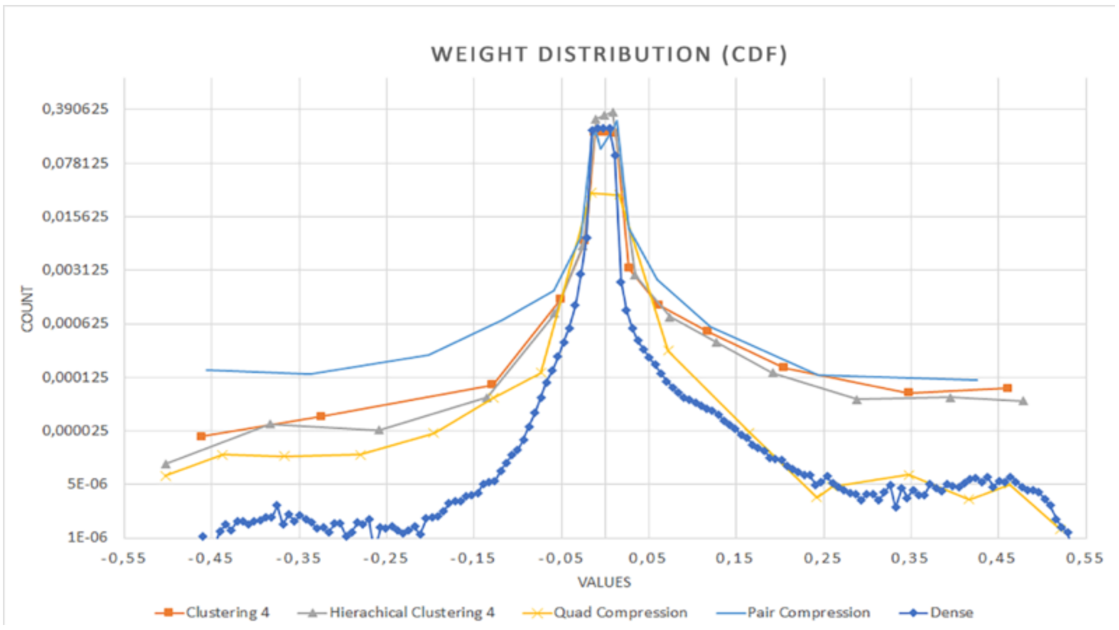


Figure 4.16: Comparison of Weight Distribution CDF

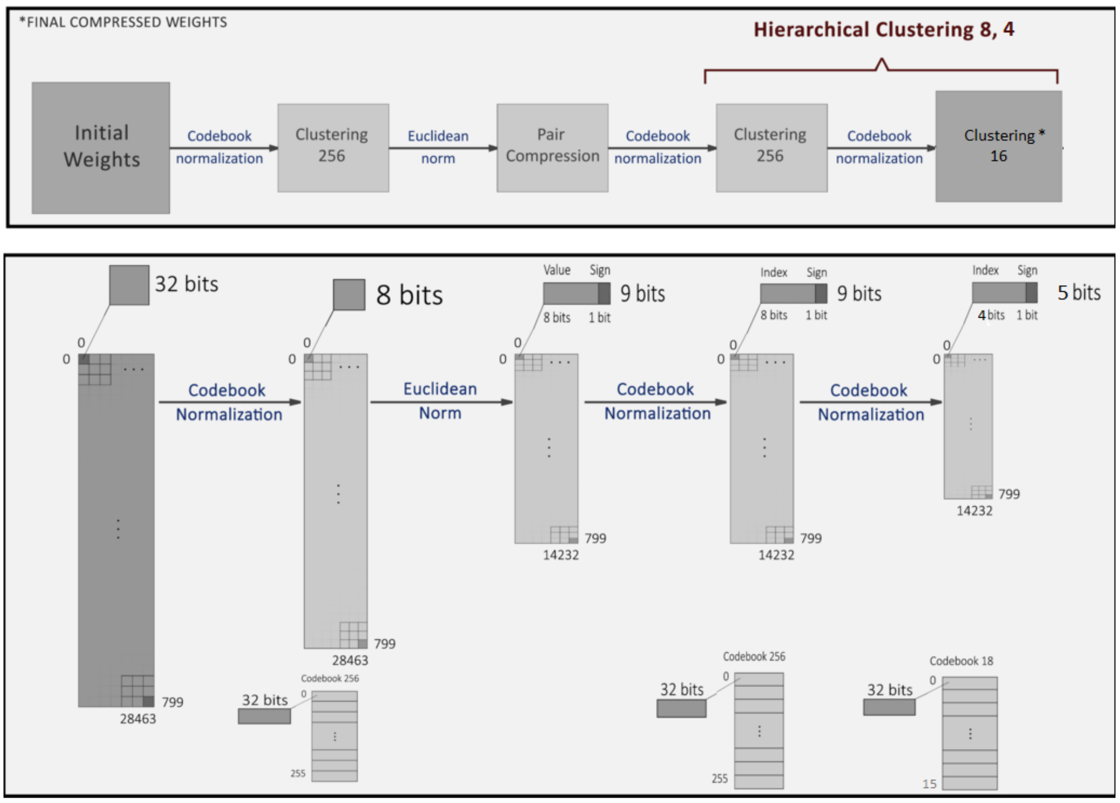


Figure 4.17: Hierarchical Clustering And Pair Compression Procedure

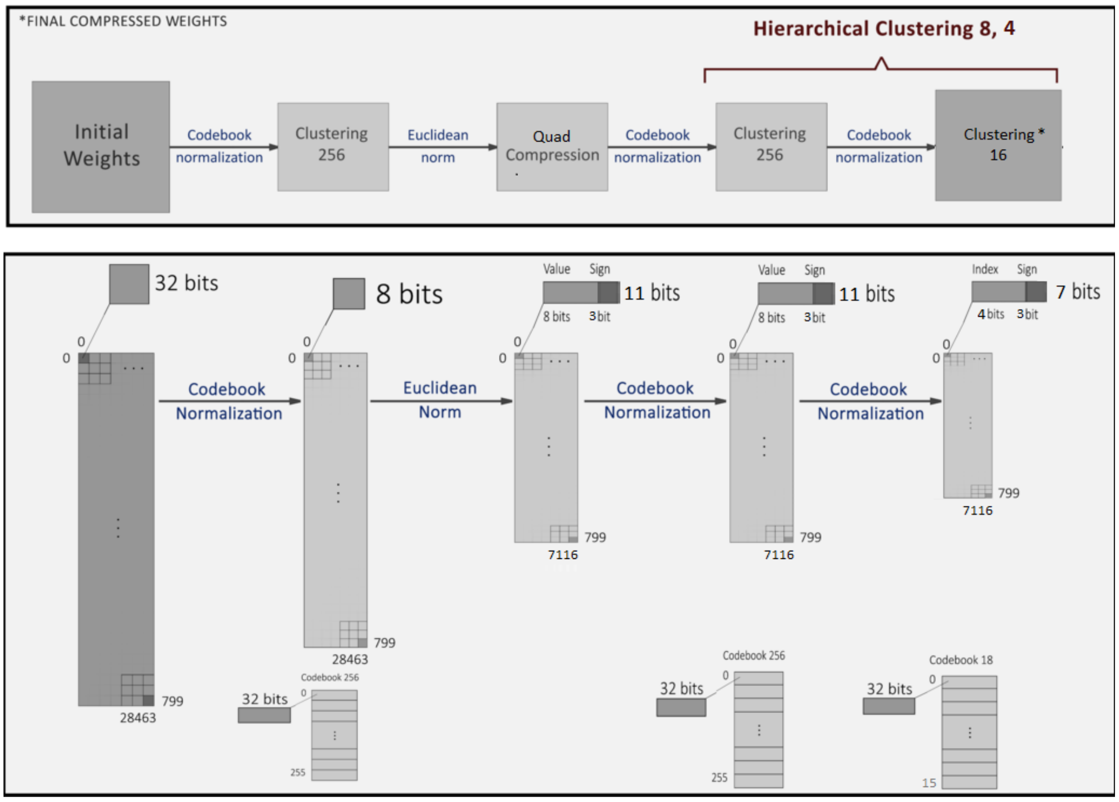


FIGURE 4.18: Hierarchical Clustering And Quad Compression Procedure

4.10 Second Level Codebook (SLC)

Next, we propose and analyze a new lossless method (Second Level Codebook) that can be applied to clustering algorithms. This method will help us to further reduce the error rate by keeping the memory footprint constant. We will begin by considering that we have implemented clustering with 16 centroids in the original data-set. As we have analyzed above, we would need $\lfloor \log_2 16 \rfloor = 4$ bits to represent this information. The difference, in this case, is that we will concatenate two compressed weights (4-bit) and create an 8-bit block. This 8-bit block will contain values from 0-256 and when decompression is applied will produce the 2 initial compressed weights. After an extensive study, we grouped all the weights in this way, and we concluded that the 8-bit block did not contain 256 different values (there was not any possible weight combination). Instead, we came up with 165 different combinations. Initially, this study was aimed at compressing the 2 weights with a smaller number of 8 bits. In order to achieve this, 8-bit blocks would have to end up with less than 127 different values so we would be able to perform a second lossless compression ending at 7 bits.

This could not be achieved, and hence we ended up applying another thought. We will try to exploit the fact that we do not use the whole range that 8-bits blocks can provide us. We came up with the following approach:

- **Clustering with more centroids**: Originally, we will operate the clustering algorithm with more centroids than the number of bits, we want to encode, would allow us. In this case, we want to reach a compressed weight ratio of 4-bit. Consequently, we will create more than 16 centroids, which 4-bit allow us, and we'll consult what is the maximum we can get.

- **Creation of 8-bit block**: To represent the 16+ centroids (eg 18 19) we would need at least 5 bits $\lceil \log_2 18 \rceil = 5$. Going one step further we will create an 8-bit weight-block concatenating 2 consecutive (loseless decompression) weights.

- **Creating Level Stage Codebook**(SLC): In order for this to be performed efficiently, we have to implement the following procedure. We assume that we will attempt to fit 18 centroids into the new block that will be built. The procedure is to concatenate the two compressed weights as follows: $Weight_{Block} = weight_1 * 18 + weight_2$ to be able to ensure that we have a unique decode. Consequently, the range of the resulting number is 0-323 (17*18 + 17). This number can be represented by $\lceil \log_2 323 \rceil = 9$bits instead of 10 if we were coded separately. Furthermore, we checked all the possible values, resulting in 245, that could occur. This information can be represented by 8 bits because $\lceil log_2 245 \rceil = 8$.

- **Decompression**: Hence we will create a second-level codebook consists of 245 indexes. This codebook contains the primary, uniquely decodable, 9-bit block. Finally, the optimized second-level codebook could be decompressed to the 2 primitive compressed weights.

Below in figure 4.19 is an example of SLC compression with weight block 2

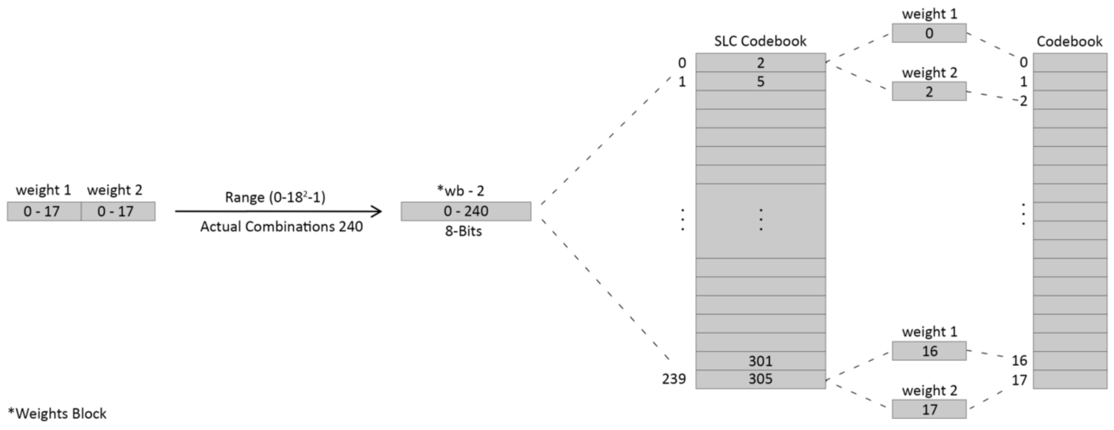


FIGURE 4.19: SLC compression example : Weights Block 2

To calculate the compression rate of the SLC, given k centroids, we only need (4.2) $\log_2(k)$ bits to encode the index. In general, for a network with n connections (weights) to have the only k shared weights (centroids) and a weights-block with p different indexes will result in a compression rate of 4.2:

$$Bit_{Width} = \log_2(k) \tag{4.6}$$

$$Compression = \frac{nb}{k * b + p * Bit_{Width}} \tag{4.7}$$

4.10.1 SLC with higher weights-blocks

This encoding can be expanded to a larger extent using larger weights-blocks. We can concatenate 4,8 or even 16 weights in larger blocks. This can lead to 2 results. We can compress the information even more (in smaller bits/weight ratio) or lower the error by keeping the bits/weight ratio stable.

To calculate the compression rate of the SLC, given k centroids, we only need (4.8) $\log_2(k)$ bits to encode the 1st Level Codebook index. In general, for a network with n connections (weights) to have the only k shared weights (centroids) and a weights-block(concatenating l different weights) with p different indexes will result in a compression rate of 4.10:

$$Bit_{Width} = \log_2(k) \tag{4.8}$$

$$SLC_{Footprint} = p * l * Bit_{Width} \tag{4.9}$$

$$Compression = \frac{nb}{k * b + SLC_{Footprint} + n * bits/weight} \tag{4.10}$$

Decreasing error rate

Originally we tried to reduce the error rate using large clusters and end up with a small error rate. In conjunction with SLC, we can exploit the extremely small error rate of a large clustering algorithm while compressing the information further. The following study has been done for the largest clustering we have studied of 256 centroids.

TABLE 4.9: SLC compression on Codebook 256 for different Weights Block

Method	**Bits/Weight**	**Error rate(%)**	**Compression rate**
Clust. 256 & 8	0.03	8x	
Clust. 256 & SLC 2-WB	6.5	0.03	9.84x
Clust. 256 & SLC 4-WB	5	0.03	10.79x
Clust. 256 & SLC 8-WB	3.5	0.03	5.57x

Clust. = Clustering , WB = Weights Block

Lowering bits/weights ratio

At this point, we used the Clustering with the 18 values and tried to compress it into as small as possible bits/weights by increasing the number of compressed weights-block.

We arranged the clustering codebook of 18 centroids and applied a pruning to the 2 closest to 0 centroids. After calculating their weighted average, we have joined them by storing this value. By implementing this we expected better compression rate because the centroids that are close to 0 have high frequency, so we will reduce the possible combinations for the SLC.

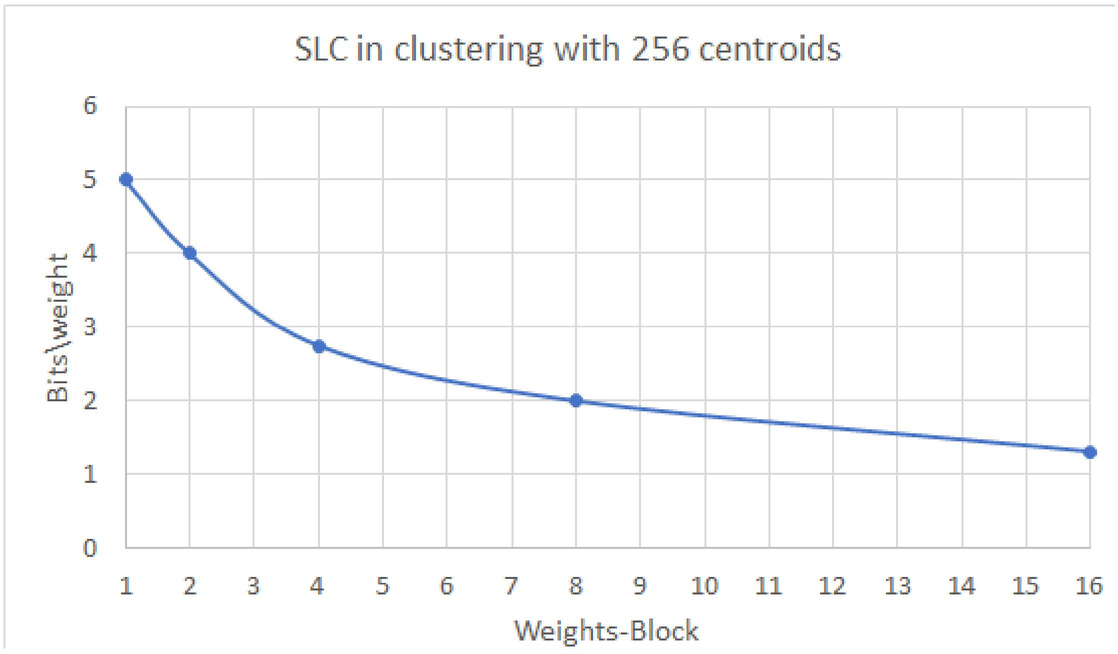


FIGURE 4.20: SLC in Clustering with 256 centroids

TABLE 4.10: SLC compression on Codebook 18 for different Weights Block

Method	Bits/Weight	Error rate(%)	Compression rate
H.Clust. 18 & 5	0.43	12.8x	
H.Clust. 18 & SLC 2-WB	4	0.43	15.99x
H.Clust. 18 & SLC 4-WB	2.75	0.43	20.5x
H.Clust. 18 & SLC 8-WB	2	0.43	30.96x
H.Clust. 18 & SLC 10-WB	1.5	0.43	30.82x
H.Clust. 18 & SLC 16-WB	1.31	0.43	10.24x

Clust. = Clustering , WB = Weights Block

Golden Ratio

By observing the tables and figures above, we realize that by using SLC we can notably reduce the bits/weight ratio. Attention must be drawn to the fact that as the weights-block (concatenate more weights) increases, linearity in compression is dropped. The golden ratio for the correct use of SLC is: for the 18 centroids using 8-weights blocks, for the 17 centroids using 12-weights blocks, while for the 256 centroids using 4-weights blocks 4.24.

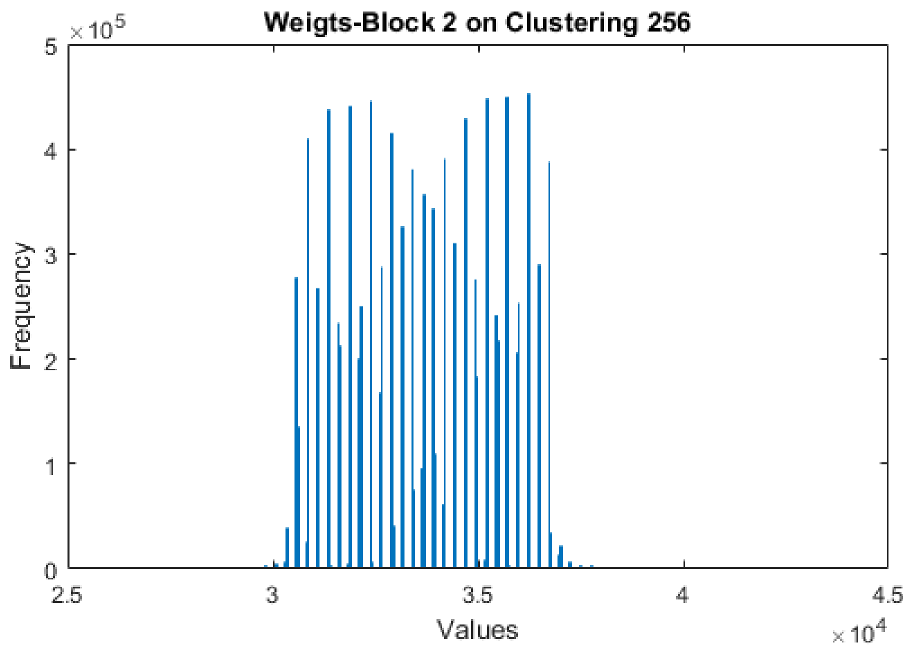


FIGURE 4.21: Weights-Block-2 on Clustering with 256 centroids

Lossless Method

A fact that needs to be reinstated is that this method is lossless. There is no loss of data in compression and therefore, after its use, we end up with the same error rate.

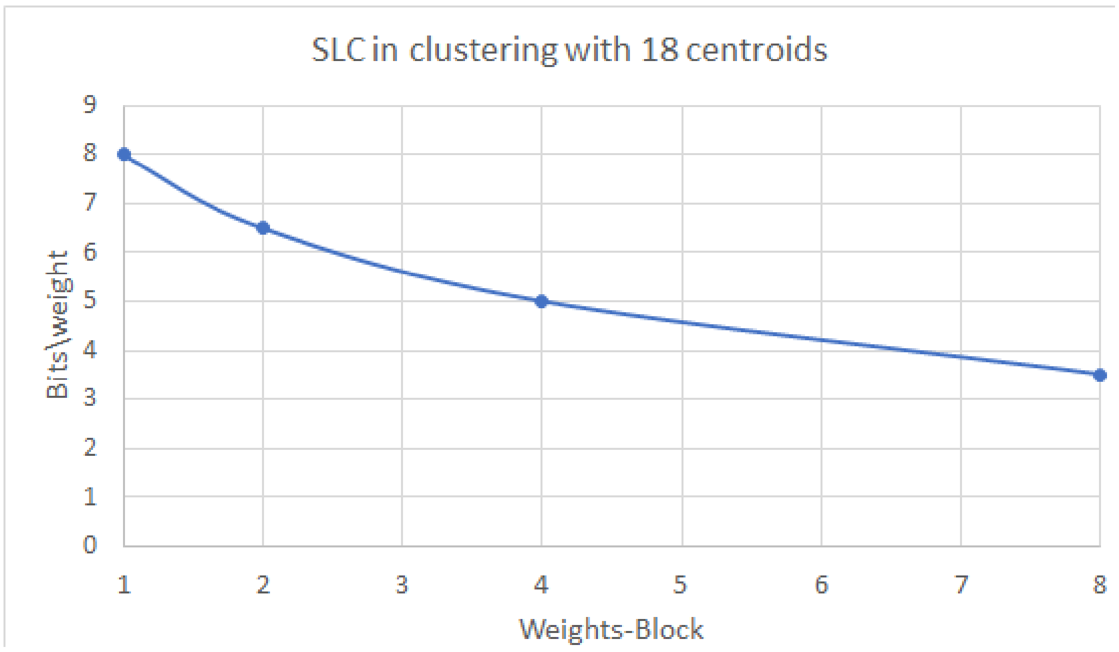


FIGURE 4.22: SLC in Clustering with 18 centroids

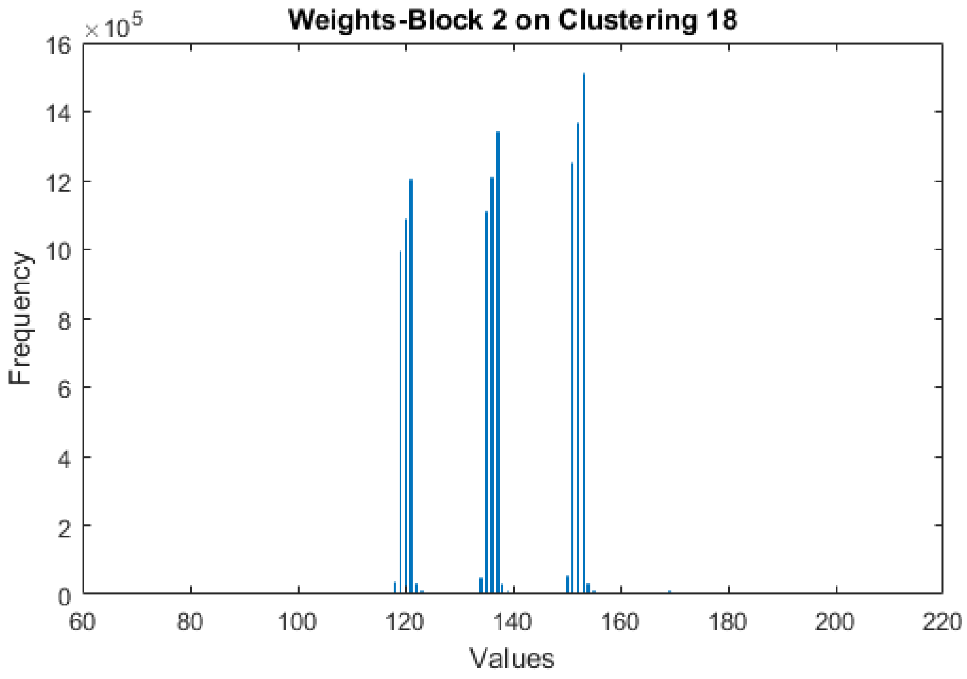


FIGURE 4.23: Weights-Block-2 on Clustering with 18 centroids

Table 4.11: SLC compression on Codebook 17 for different Weights Block

Method	**Bits/Weight**	**Error rate(%)**	**Compression rate**
H.Clust. 17	5	0.39	12.8x
H.Clust. 17 & SLC 2-WB	3.5	0.39	18.29x
H.Clust. 17 & SLC 4-WB	2.5	0.39	25.59x
H.Clust. 17 & SLC 8-WB	1.625	0.39	39.1x
H.Clust. 17 & SLC 10-WB	1.5	0.39	40.83x
H.Clust. 17 & SLC 12-WB	1.33	0.39	43.78x
H.Clust. 17 & SLC 15-WB	1.13	0.39	41.86x

Clust. = Clustering , WB = Weights Block

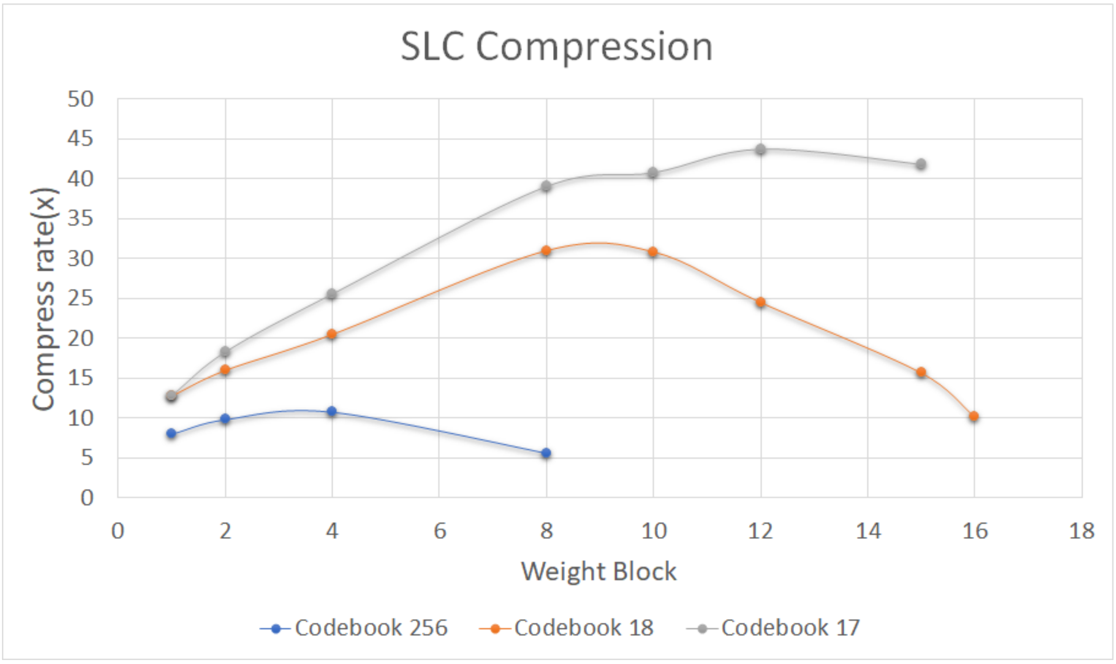


Figure 4.24: SLC Performance : For different codebooks an weights-blocks

4.11 SLC comparison with Huffman

In computer science, Huffman code is a commonly used lossless data compression algorithm. Huffman's coding algorithm is an prefix optimal code for a "symbol-by-symbol" coding with a known probability distribution of data. The algorithm developed by David A. Huffman while he was a PHD student at MIT, and published in the 1952 [35].

In this section and more specifically in table 4.12 and figure 4.25 we compare our lossless SLC algorithm with the optimal Huffman. The SLC method we have implemented is more FPGA-friendly than Huffman. The problem with Huffman and other prefix-code encoding are that they do not have a fixed Bits/weight ratio. This varies by weight in weight and e.g. the most likely weight can be represented by 1 Bit, which is unlikely by 14bits. The above to be effective in Hardware is prohibitive because enormous if-conversions will have to be implemented. This will dramatically increase the critical path and the use of many resources. In contrast to Huffman, SLC creates a fixed Bits/weight ratio Compression, by reading a certain number of bits leading to a specific number of weights.

TABLE 4.12: Comparison of SLC compression on Codebook 18 and 17 with Huffman Coding

Method	Bits/Weight	Error rate(%)	Compression rate
H.Clust. 18 & SLC 8-WB	2	0.43	30.96x
H. Clust. 18 & Huffman	1.98	0.43	32.39x
H.Clust. 17 & SLC 12-WB	1.33	0.39	43.78x
H. Clust. 17 & Huffman	1.34	0.39	47.75x

Clust. = Clustering , WB = Weights Block

In this table, we see that SLC achieves compression rates very close to those of Huffman. More specifically for the 17-codebook, it reaches 91.7%, while for the 18-codebook 95.6% of the optimal performance of Huffman.

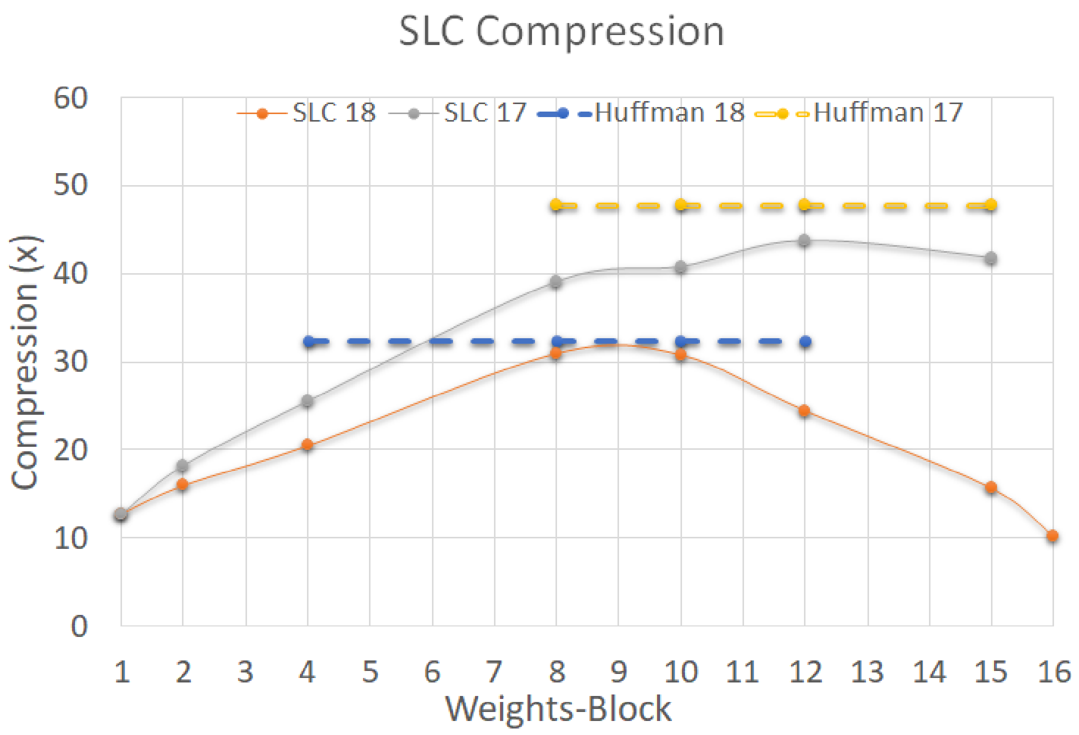


FIGURE 4.25: Comparison of SLC compression on Codebook 18 and 17 versus Huffman Coding: for different number of weights-block

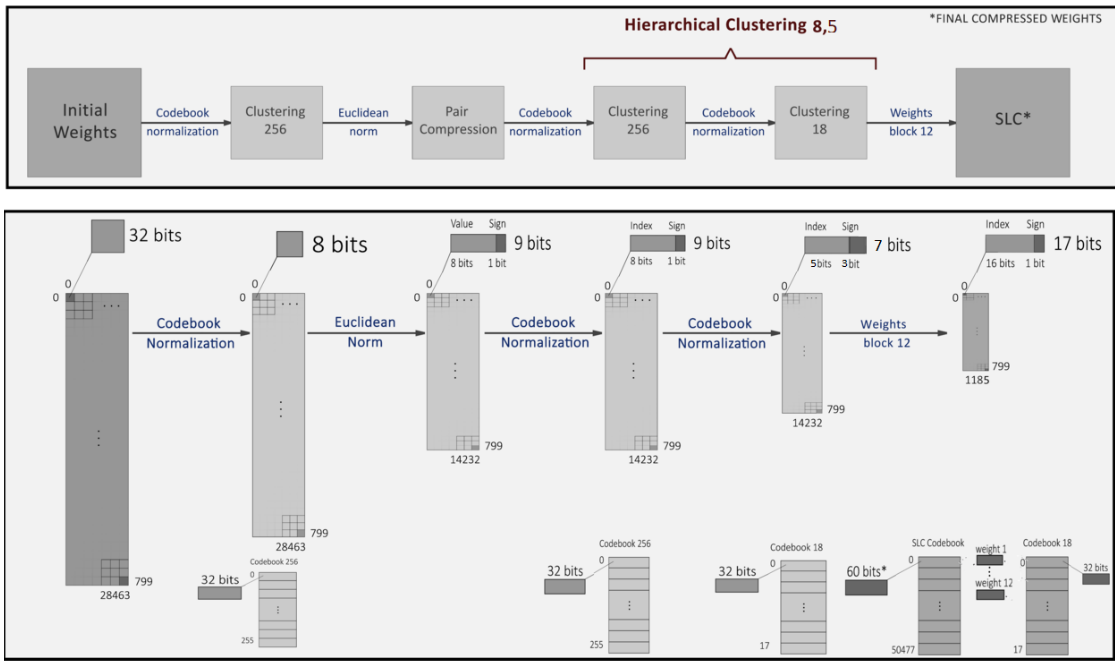


FIGURE 4.26: Compression Flow using Pair Compression and SLC: Weights-Block 12

4.12 SLC, Hierarchical Clustering and Pair Compression

The next step is to try to combine the SLC method with the existing techniques we have implemented. It is obvious that there is no conflict between the simultaneous use of these methods and SLC is because they try to achieve data compression from a different perspective. 4.27

Below in figures 4.27 4.28 we present compression flow using the proposed methods and techniques.

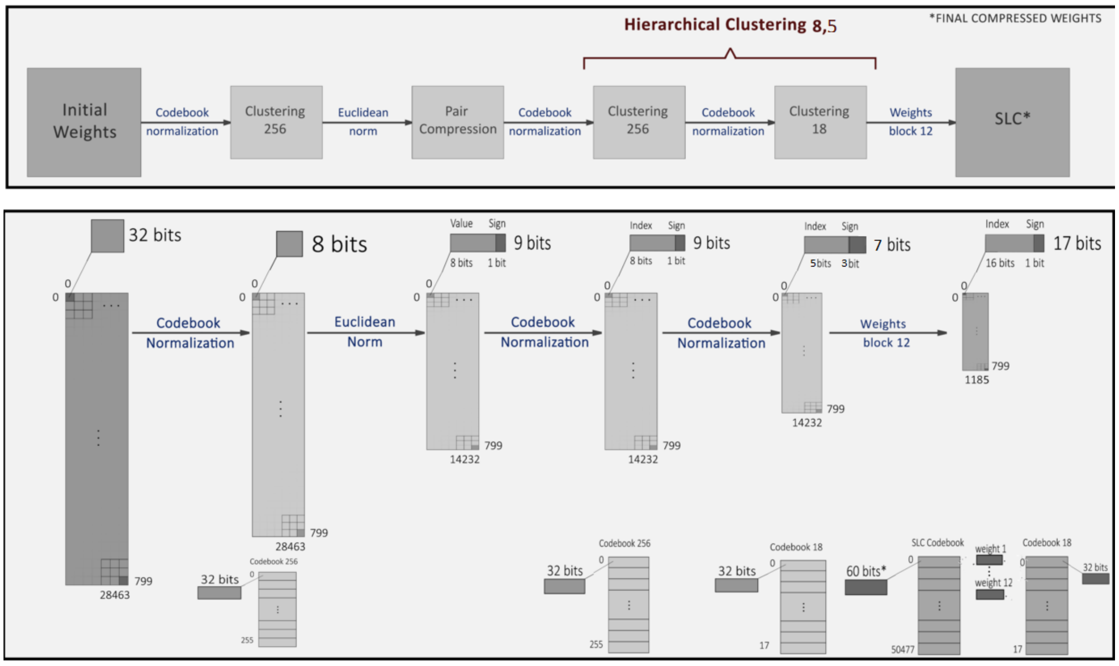


FIGURE 4.27: Compression Flow using Pair Compression and SLC: Weights-Block 12

To calculate the compression rate of the SLC alongside the other methods, given k centroids, we only need (4.8) $\log_2(k)$ bits to encode the 1st Level Codebook index. In general, for a network with n connections (weights) to have the only k shared weights (centroids) and a weights-block(concatenating l different weights) with p different indexes will result in a compression rate of 4.13, 4.14 applied Pair or Quad Compression:

$$Bit_{Width} = \log_2(k) \tag{4.11}$$

$$SLC_{Footprint} = p * l * Bit_{Width} \tag{4.12}$$

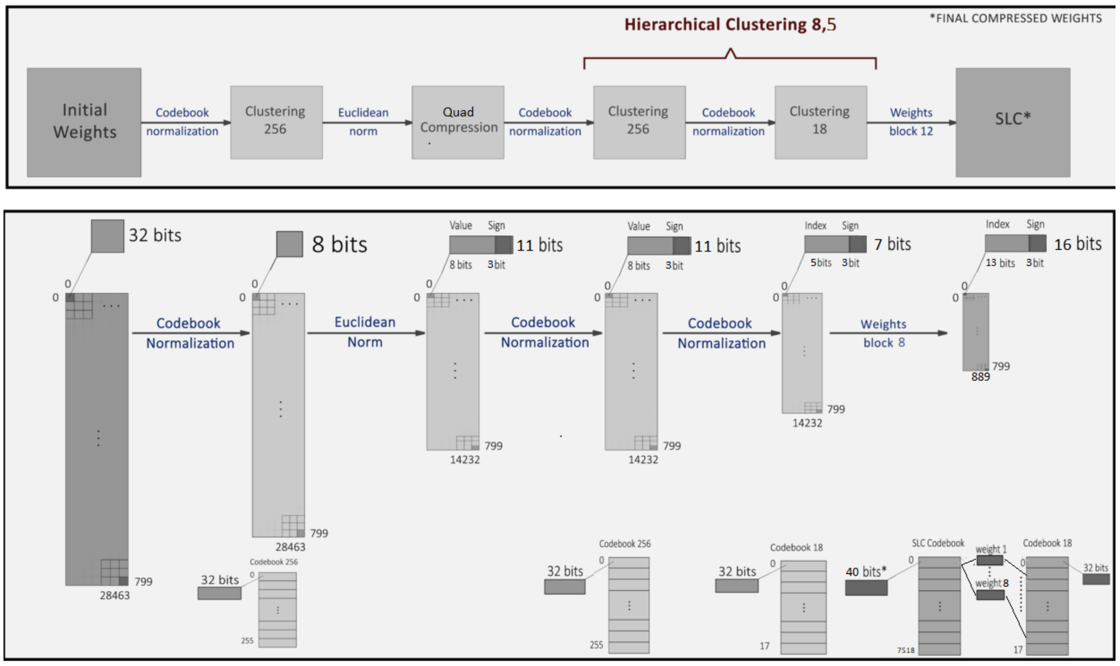


FIGURE 4.28: Compression Flow using Pair Compression and SLC: Weights-Block 8

$$Compression = \frac{nb}{k*b + SLC_{Footprint} + \frac{n*((bits/weight)+1)}{2}} \tag{4.13}$$

$$Compression = \frac{nb}{k*b + SLC_{Footprint} + \frac{n*((bits/weight)+3)}{4}} \tag{4.14}$$

In order to be able to unfify the proposed techniques, SLC Compression will be applied to the compressed weights (After Pair, Quad Compression). Hence in table we present our final compression and error results after every stage of our techniques and compression methods we propose.

In this case the maximum compression rate we could get if we managed to have q-Compression, where $q \to \infty$ is:

$$Compression = lim_{q\to\infty}\frac{nb}{k*b + SLC_{Footprint} + \frac{n*(bits/weight+q-1)}{q}} = lim_{q\to\infty}\frac{nb}{k*b+n} = 64$$

(4.15)

Table 4.13: Final Compression Methods

Method	Bits/Weight	Error rate(%)	Compression rate
Clust. 16	4	1.37	16x
H.Clust. 16	4	0.65	16x
P.C.& H.Clust. 16	2.5	0.62	25.6x
Q.C.& H.Clust. 16	1.75	0.76	36.57x
P.C.& H.Clust. 18 WB-12	1.3	0.5	49.24x
Q.C.& H.Clust. 18 WB-8	1.17	0.8	54.73x
P.C.& H.Clust. 256	3	0.16	18.46x
Q.C.& H.Clust. 256	2	0.2	28.65x

Clust. = Clustering , H. Clust. = Hierarchical Clustering , P.C. = Pair Compression, Q.C. = Quad Compression

4.13 Proposed methods comparison with Binarized Approach

We notice that after applying all the methods we proposed the compressing rate reaches the order of 54.73. This means that we need 1.17 bits to send a weight overall (calculating the stream of codebooks), or more specifically 13+3=16 bits to send 4*8=32 weights after send the 2 Codebooks for the decompression. This analogy is extremely close to a binarized approach. Hence we implemented a binarized approach for our network and compare the results with the previous methods.

Table 4.14: Proposed methods comparison with Binarized Approach

Method	Bits/Weight	Error rate(%)	Compression rate
Q.C.& H.Clust. 18 WB-8	1.17	0.8	54.73x
Binarized	1	40	64x

Clust. = Clustering , H. Clust. = Hierarchical Clustering, Q.C. = Quad Compression

We mention that a Binarized Network, having a 64x compression rate, reaches a 40% error which is huge and prohibitive for a CNN. On the other hand, our own network having a 57.38x compression rate, leading to a very small error of 0.7 %

Chapter 5

FPGA Implementation

5.1 Tools Used

The CNN (Inference) Hardware Accelerator was implemented using the Xilinx Vivado Design Suite - HL System Edition 2017.1. Vivado Design Suite is a software suite developed by Xilinx for synthesis and analysis of HDL designs, superseding Xilinx ISE with additional features for system on a chip development and high-level synthesis. The tools used are the Vivado HLS, Vivado IDE, and Xilinx SDK [76].

5.1.1 Vivado HLS

The Xilinx Vivado HLS tool [83] provides a higher level of abstraction for the user by synthesizing functions written in C,C++ (with constraints and feautures) into IP blocks, by generating the appropriate ,low-level, VHDL and Verilog code. Then those blocks can be integrated into a real hardware system. Vivado HLS is tightly integrated with the rest of the Xilinx design tools and provides comprehensive language support and features for lower level optimizations, making it possible to optimize the C code for hardware systems. The Vivado HLS tool allows for C functions written in C, C++, SystemC, or an OpenCL API C kernel. We decided to use C++, as there are several supported features in HLS design (template e.t.c.) . To debug the code, Vivado HLS uses a C test bench to simulate the C function prior to synthesis and to verify the RTL output using C/RTL Cosimulation.

The tool also adds some constraints to the exported IP block, like the clock period, clock uncertainty, and FPGA target. The clock uncertainty defaults to 12.5% of the clock period, but you have the option to change it. Finally, the tool allows for directives to be added to direct the synthesis process to implement a specific behavior or optimization. Directives are optional and do not change the behavior of the c code in the simulations, only in the synthesized IP block.

Synthesis Report in Vivado HLS

When synthesize an HLS code tool exports a synthesis report showing the performance metrics of the generated design. The report's information on performance metrics are presented below:

- **Area:** Amount of hardware resources required to implement the design based on the resources available in the target FPGA. The types of resources are, Look-Up Tables (LUT), Flip Flops (FF) , Block RAMs (BRAMs), and DSP48s.
- **Latency:** Number of clock cycles required for the function to compute all output values.
- **Iteration Interval (II):** Number of clock cycles before the function can accept new input data.
- **Loop Initiation Interval:** Number of clock cycle before the next iteration of the loop starts to process data.
- **Loop Latency:** Number of cycles to execute all iterations of the loop.
- **Loop Iteration Latency:** Number of clock cycles it takes to complete one iteration of the loop.

Optimizations in Vivado HLS

Vivado HLS also provides (optional) directives that can be used to optimize the design: reduce latency, improve throughput performance, and reduce area and device resource utilization of the resulting RTL code. These pragmas can be added directly to th1e source code for the kernel.

- **Pipeline**: The PIPELINE pragma reduces the initiation interval for a function or loop by allowing the concurrent execution of operations. A pipelined function or loop can process new inputs every N clock cycles, where N is the initiation interval (II) of the loop or function.
- **Array Partition**: Partitions an array into smaller arrays or individual elements.

  This partitioning:

  - Results in RTL with multiple small memories or multiple registers instead of one large memory.

  - Effectively increases the amount of read and write ports for the storage.
  - Potentially improves the throughput of the design. Requires more memory instances or registers.

- **Unroll**: Unroll loops to create multiple independent operations rather than a single collection of operations. The UNROLL pragma transforms loops by creating multiples copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel.

- **Stream**:By default, array variables are implemented as RAM. If the data stored in the array is consumed or produced in a sequential manner, a more efficient communication mechanism is to use streaming data as specified by the STREAM pragma, where FIFOs are used instead of RAMs.

- **Array Map**: Combines multiple smaller arrays into a single large array to help reduce block RAM resources.

- **Resource**: Specify that a specific library resource (core) is used to implement a variable (array, arithmetic operation or function argument) in the RTL. If the RESOURCE pragma is not specified, Vivado HLS determines the resource to use.

- **Dataflow**: The DATAFLOW pragma enables task-level pipelining, allowing functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation, and increasing the overall throughput of the design.

5.1.2 Vivado IDE

The Vivado IDE (was implemented from co-partner Giannis Kalomoiris) is the GUI for the Vivado Design Suite. All of the Vivado design Suite tools are written with a native Tcl interface, and all of those commands are available through the IDE either through the GUI or through the Tcl console. Tcl commands can be entered in the Tcl Console in the Vivado IDE or using the Vivado Design Suite Tcl shell. You can run analysis and assign constraints throughout the design process. Timing and power estimations are provided after synthesis, placement, and routing.

5.1.3 Vivado SDK

The Xilinx SDK is an IDE tool to develop embedded software applications targeted towards Xilinx ARM processors. The SDK works with hardware designs and bitstreams created with Vivado Design Suite. The SDK is based on the Eclipse standard. SDK includes an C/C++ code editor and a compilation environment with easy project management, application build configuration and automatic Makefile generation. It also provides an environment for debugging and profiling of software code as well as design performance. The SDK also provides it own tools to configure FPGAs and create bootable bitstrems with software extensions. We open the SDK (through IDE) using the preconfigured directories, including bitsreams after successful Vivado IDE implementation. The SDK automatically imports the project hardware wrapper and generates the files needed(memory porting , defines ,etc) for the software part. Therefore we create a new fuzzy project which generates the project files,and the needed BSP packages, which includes the suitable drivers for the software design that the PS has access to.

The SDK is fisrtsly used to create a simple program to run by the PS to test and debug the PL functionality. To be able to program the FPGA, the JTAG port has to be connected to the PC, and to monitor and debug it we use the UART port as well. Another very useful tool that is part of the Vivado IDE is the Hardware Manager. It connects to the ILAs that have been added to the Vivado project and allows us to monitor in real-time the values of the signals between our modules, while the program is running.

For the needs of the SDK, we have created functions where activate and initialize our IPs, DMA, memories, and caches. Additionally, functions for writing and reading data from the SD card were implemented. We made a study to be able to measure the real memory bandwidth. The data was stored in the DDR in 2 ways either by SD read or JTAG. Finally, time measurement functions were used to enable speedups to be evaluated.

5.2 FPGA platforms

Our architectures are targetting on 2 FPGA platforms we worked on.

5.2.1 Xilinx Zynq UltraScale+ MPSoC ZCU102

The ZCU102 is a general purpose evaluation board for rapid-prototyping based on the Zynq UltraScale+ XCZU9EG-2FFVB1156E-2-i MPSoC (multiprocessor system-on-chip). High-speed DDR4 SODIMM and component memory interfaces, FMC expansion ports, multi-gigabit per second serial transceivers, a variety of peripheral interfaces, and FPGA logic for user customized designs provides a flexible prototyping platform.

Below in table 5.1 we present main features of the FPGA

Table 5.1: ZCU102 Specifications

Logic Cells (K)	**B-RAM (MB)**	**DSP Slices**	**PS DDR (GB)**	**PL DDR (MB)**
600	4	25200	4	512

MB=Mbytes , GB=Gbytes

5.2.2 QFDB

QFDB is a 4-FPGA custom platform designed and implemented by FORTH. It has 4 ZCU102 FPGA,(meaning 4x powerful than ZCU102) thus our architecture transference between platforms are fully compatible.

5.3 Bottom-up Strategy

The first approach was implemented with the bottom-up strategy. A bottom-up approach is the unification of many "simple" systems to end up in a more "complex" system.

The algorithm was divided into small entities with specific behavior and hierarchical structure. Then they would be joined together to make the overall accelerator. The entities were separated as follows:

- **Convolution (1-D)** (conv1): 1^{st} Convolutional layer of the network (Input: image, kernel1, bias1)(Output:ConvStage1)
- **Convolution (2-D)** (conv2): 2^{nd} Convolutional layer of the network (Input: ConvStage1, kernel2, bias2)(Output:ConvStage2)
- **Convolution (2-D)** (conv3): 3^{rd} Convolutional layer of the network (Input: ConvStage2, kernel3, bias3)(Output:ConvStage3)
- **Fully Connected** (fc): Fully Connected of the network (Input: ConvStage3, $kernel_{dense}$, $bias_{dense}$)(Output:Classification)
- **ReLU** : ReLU is the main activation function that was performed after every convolutional layer. For this purpose, ReLU was inserted in every Convolutional Layer entity.

5.3.1 Two Stage Architecture

After the network was divided and depicted in small entities, they were reunited to make the two basic modules of our architecture. The first module is "Convolutional Layers" where it contains the three Convolutional Layer with the corresponding ReLU. The second module is the "Fully Connected Layer" where it receives the exit of the first module and makes the final classification. This leads to a 2-Stage Architecture designed to communicate 2 FPGAs. The first will implement Convolution and the second the Fully Connected module 5.1.

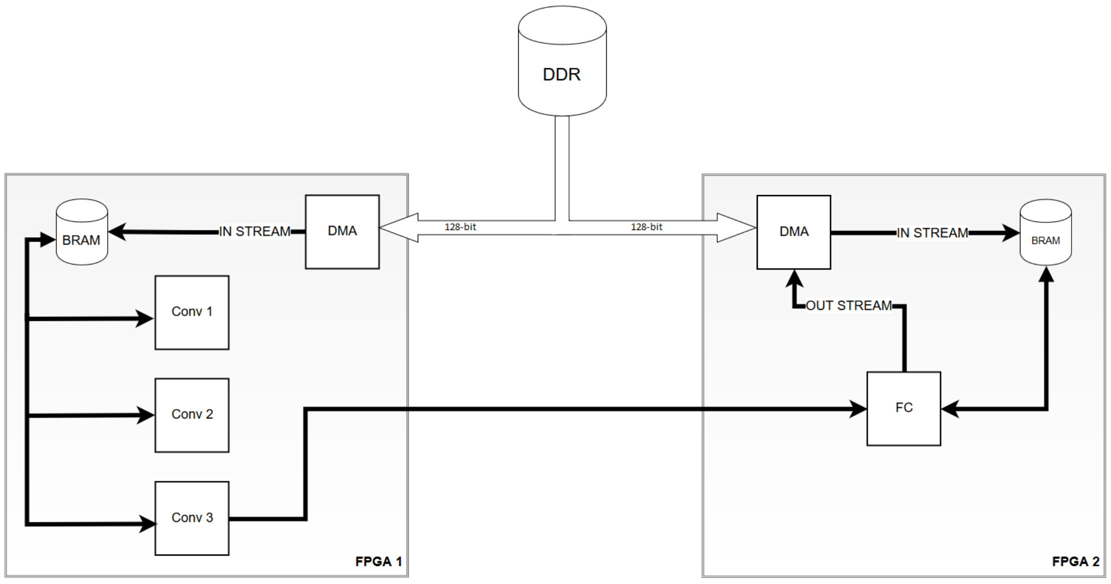


FIGURE 5.1: Datapath of Two-Stage Architecture

5.4 First Approach

5.4.1 Memory and I/0

Firstly we have to clarify how to insert the data into the different Building Blocks of the accelerator from the outside world. The first step is to pass the data to the DDR of the processor. Then we have three ways to link this data to FPGA:

- **Memory-mapped I/O** (MMIO): A complementary method of performing input/output (I/O) between the central processing unit (CPU) and peripheral devices such as FPGA. This method uses the same address space to address both memory and I/O extensions. The memory and registers of the I/O devices are mapped to (associated with) address values. This technique comes to emphasize the main DDR drawback, which is that it can not efficiently drive multiple requests because it is random access. This means that for each request we would have to "pay" the initial interval (30 -50 cycles) and so there would be no meaning for pipeline design in the FPGA. This technique can be implemented through using Xilinx's IP Data-movers.

- **Streaming** (AXI-4): The second method we can use is Streaming Interface using AXI-4 protocol. In fact, we are creating a continuous DDR communication channel with FPGA (a large FIFO) and we are sending the processor the data we want without requests. This relieves us from the delay of requests, and you create a huge pipeline for entering data by hiding the DDR interval. This technique can be implemented through using Xilinx's IP DMA.
- **B-RAM** : Another approach is to transfer the data in burst with memory mapped or stream and then store them in B-RAM to take advantage of the huge bandwidth. In order to achieve this, the data must have a small memory footprint. Typically the B-RAM is in the order of many KBs and a few MBs.

Using Streaming Interface is much more efficient because it allows us to fully utilize the DDR bandwidth (about 10 GBs). However, this method can be used only in cases where we know in advance the data-flow. CNNs have inherently streaming nature, so they are ideal for taking advantage of this.

5.4.2 Transference to HLS

A next step was to be able to integrate each Entity's algorithm parts from MATLAB into C++ and then into HLS synthesizable code. In order to successfully complete this, the limitations and capabilities of the tool had to be studied. Convolution was the most algorithmic complex part. On the other hand, Matrix Multiplication has less algorithmic complexity, but it requires much more computation time.

5.4.3 Convolution (1-D)

First, we grant the weights and the image into the FPGA and store them in the internal B-RAM. This approach came because we need to use that data several times. The reasoning in the use of B-RAM instead of sending the data with Stream interface was to be able to exploit its bandwidth (TBs).

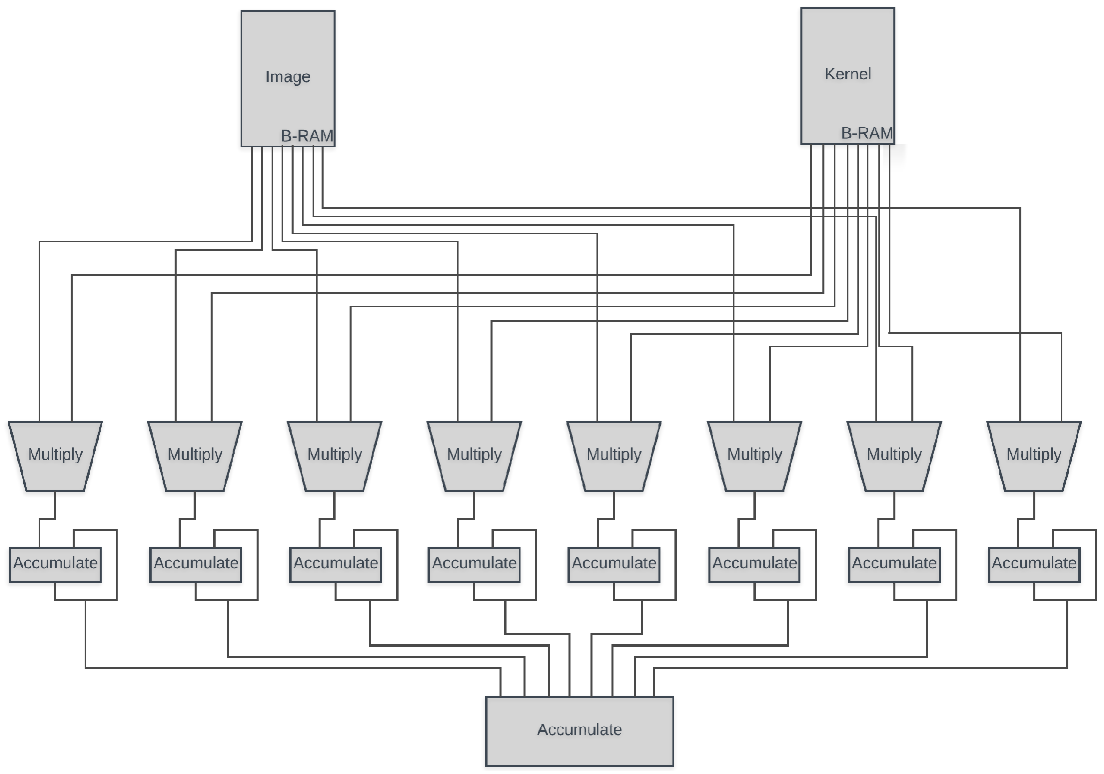


FIGURE 5.2: Datapath of Convolution(1-D)

Array Partitioning

B-RAM has memory access limitations because it has two memory channels. A feature that HLS provides you is the option to partition a B-RAM array. This allows you to have more than two memory accesses in a clock cycle. Essentially it creates copies of your original array with the use of multiplexer systems. The nature of the algorithm is access to specific arrays many times in the same cycle, so it is advisable to use this directive.

Pipeline

In FPGA designing, it is very important to manage running your algorithm in a large pipeline, thus taking advantage of FPGA's capabilities. However, in order to succeed Pipeline, a limitation that came up was algorithmic. We basically wanted to process an 8-part set, e.g. 0-7, of the image in a cycle. In the next cycle, we have to retain the last 7 parts and store the next part, having at the end of this cycle a new 8-part set e.g. 1-8 (stride=1 of the network). This was approached by creating a new dimension in the image and implementing it as a shifted B-RAM, achieving by that pipeline=1.

5.4.4 Convolution (2-D)

The complexity of this entity is 16x compared to Convolution(1-D), as it has an extra dimension size of 16. The weights and the image are granted into the FPGA and store them in the internal B-RAM with the same way. This approach came because we need to use that data several times. The reasoning in the use of B-RAM instead of sending the data with Stream interface was to be able to exploit its bandwidth (TBs).

Array Partitioning

B-RAM has memory access limitations because it has two memory channels. A feature that HLS provides you is the option to partition a B-RAM array. This allows you to have more than two memory accesses in a clock cycle. Essentially it creates copies of your original array with the use of multiplexer systems. The nature of the algorithm is access to specific arrays many times in the same cycle, so it is advisable to use this directive.

Pipeline

In order to succeed Pipeline a limitation that came up was also algorithmic. We basically wanted to process 16x8 parts of the image in a cycle and in the next cycle to retain the last 16x7 and store the next 16 part (stride=1 of the network). This was approached by creating a new dimension in the image and implementing it as a shifted B-RAM, achieving by that an overall pipeline=8.The time-chart of Convolutional Layers presents in figure 5.3.

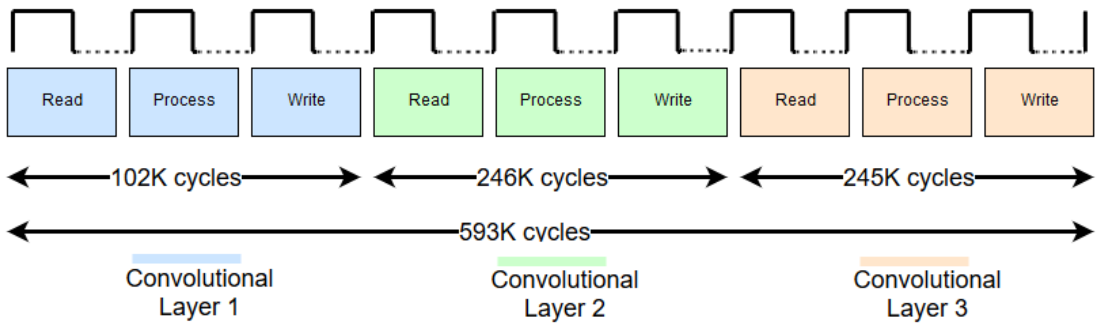


FIGURE 5.3: Convolutional Layers Time-Chart of First Approach

5.4.5 Fully Connected

The idea of the streaming interface was fully utilized to implement Fully Connected Layer. We store into B-RAM the least possible data(image stage, bias) and we basically stream the weights from DDR. The reason for this is because weights of the dense layers could not be stored into B-RAM. 5.4

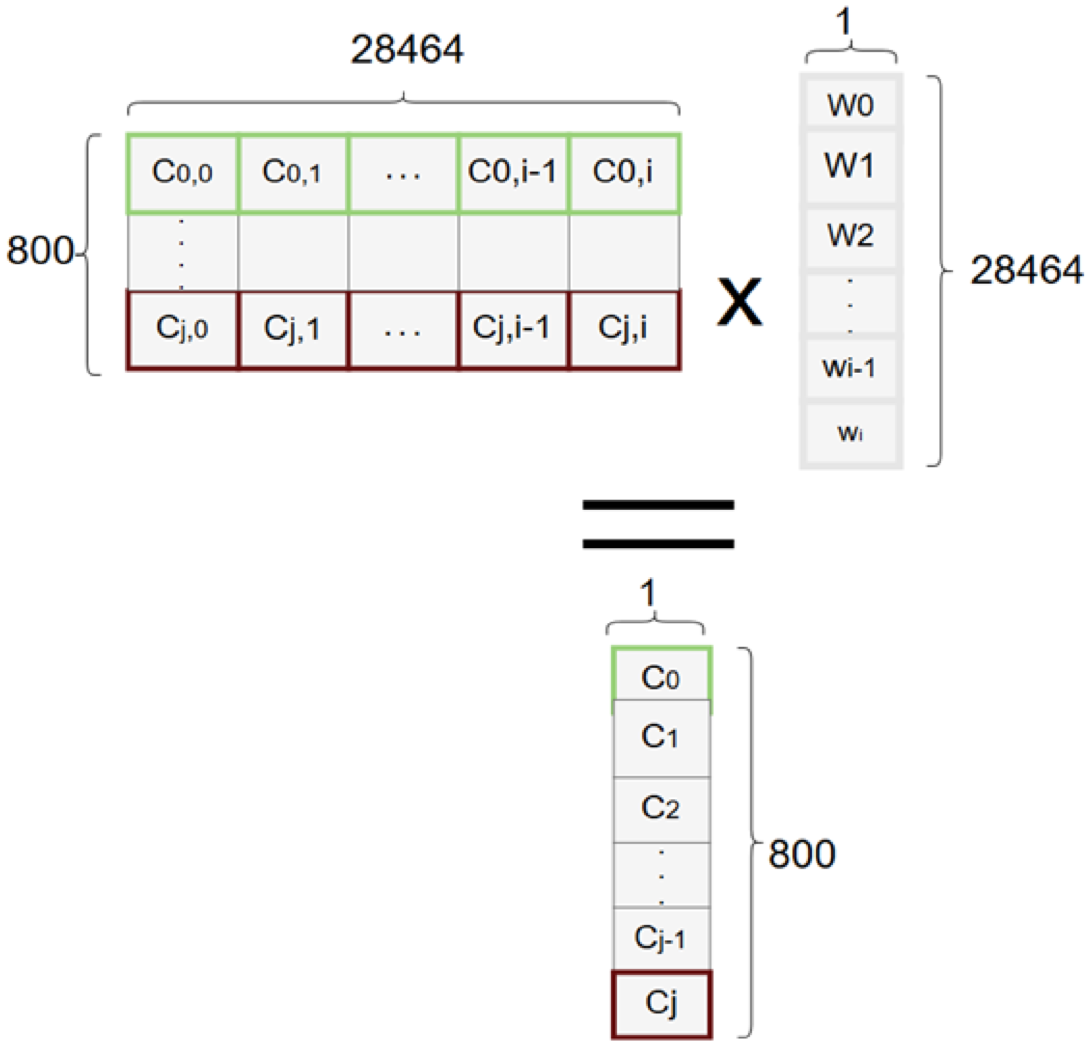


FIGURE 5.4: Fully Connected Module

Pipeline

A Pipeline (Iteration Interval=1) has been implemented essentially by export each partial result in each cycle. The time-chart of CNN presents in figure 5.5.

GB=Gbytes

TABLE 5.2: First Approach Performance

Modules	**Latency (cycles)**	**Comp. Performance (GFLOPS)**	**Bandwidth (GB/s)**
conv	593596	7.6	1
dense	22971200	0.6	1
conv+dense	23564796	0.77	1

In computing, floating point operations per second (FLOPS, flops or flop/s) is an important measure of computational performance, useful in fields of scientific computations that require floating-point calculations 5.1.

$$FLOPS = \frac{cycles}{second} * \frac{FLOPs}{cycle} \tag{5.1}$$

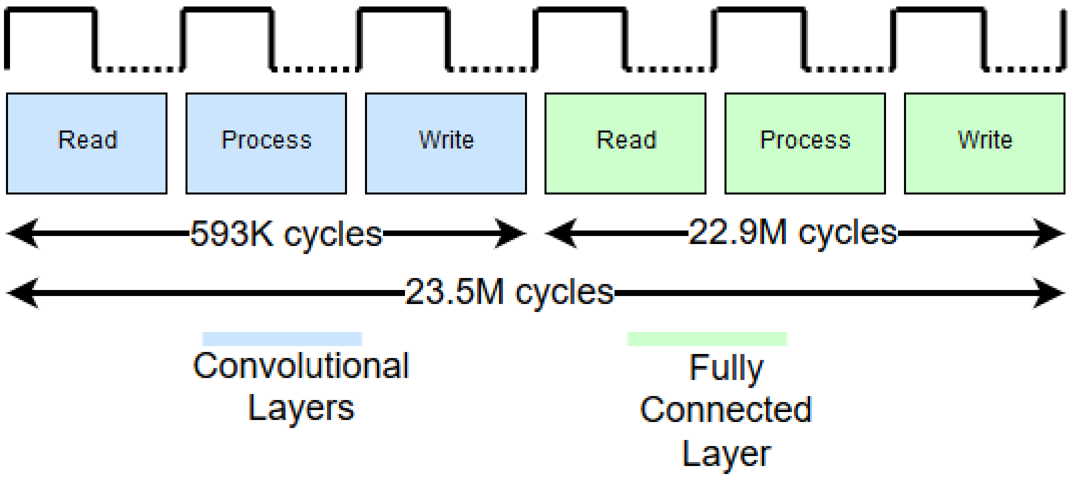


FIGURE 5.5: CNN Time-Chart of First Approach

5.5 A better Approach

After the first approach was realized how long we were from a possible speedup over the GPU. The main obstacle was that we did not use the maximum DDR Bandwidth (we used 1GB / s, meaning 6% of the maximum theoretical Bandwidth), resulting in the failure to exploit the processing power of the FPGA.

5.5.1 Larger Streaming Buses

The next action is to try to limit ourselves from the bandwidth of DDR. Consequently, we tried to use larger memory streaming buses (more than 32bits). The limitations here are three:

- **DMA**: The maximum memory bus that DMA can support is 1024 Bits in one cycle.
- **DDR Bandwidth**: The maximum bandwidth of the DDR is 16GB/s. So the maximum bus considering 300MHz clock we can support is 458 bit from 5.2.
- **HP ports** : The maximum memory bus that the CPU can transfer from memory to FPGA through DMA. This is done through HP ports and the limit is 128-bit.

$$Bus_{BitWidth} = \frac{Bandwidth}{Clock_{frequency}} \tag{5.2}$$

Bandwidth=Memory Bandwidth in b/s , $Clock_{frequency}$ of the FPGA

Therefore we end up using 128 bits, that is 4 times larger than the previous ones. This means we use 4 times more memory bandwidth. This could lead to a 4x speedup by implementing the ideal architecture.

5.5.2 Multiple DMAs

Next, we realize that even though we used 4GB/s bandwidth we have not reached its full potential(16GB/s). The next solution we can exploit is to use multiple DMAs where they will stream from different HP ports (maximum 6) data linking in the same DDR. Theoretically, 4 DMAs is the golden ratio because we will be totally restrained from the memory.

DDR Bandwidth

In order to be convinced how many DMAs (and HP ports) would be the best to use we did some fuzzy IPs (which reads and writes random values through DMAs) to be able to accurately count read, write and read/write bandwidth as showing table 5.3. Furthermore to achieve higher bandwidth we enabled caches and flushed them with the weights.

Read and Write channels are separated, so it makes sense not to see a linear increase in bandwidth for reading and write separately. Whereas when we use

TABLE 5.3: Memory Bandwidth

HP ports	Read BW (GB/s)	Write BW (GB/s)	Read/Write BW (GB/s)
1	4.15	4.15	4.18
2	8.24	8.23	8.3
3	9.8	9.9	12.4
4	10.2	10.4	16.8

GB=Gbytes

both channels we see linearity. Our algorithm is using the read channel much more because it requires to read more data than to write in the final classification. Therefore we conclude that the best possible case for not consuming resources is to use 2 HP ports and 2 DMAs.

5.5.3 Convolution (1-D) and Convolution (2-D)

Having 8 times higher bandwidth than the first approach we can store B-RAM 8 times faster. The rest and most of the implementation reads the data through B-RAM, so there is no significant optimization.

5.5.4 Fully Connected

On the other hand, the fully connected was clearly set up as streaming oriented IP. Now by bringing 8 times more data (2 DMA (128) -> 256 bit versus 32bit), we must take advantage of this feature and come up with 8x speedup relation to the first approach.

Multiply and Accumulate To be able to achieve this we need to observe from a different perspective the basic operation of multiply and accumulate and try to "hide" the Latency of the 10 cycles it needs. In order to be able to exploit the rate that weights are being streamed in, we have to increase the parallelism at the level of operations.

First Approach Trying to make the 8-fold multiplication and addition based on the current Fully Connected implementation end up in a Pipeline equal to the accumulative delay (10 in our case), something that does not satisfy us 6. This leads to a very poor exploit of the memory bandwidth. It is perceptible that we have to follow a different implementation trying to achieve a full pipeline module, managing to hide the MAC operation delay.

Algorithm 6 Matrix Multiplication

```
1: procedure MATRIX MULTIPLICATION($input, weights$)
2:     $NumOfClasses \leftarrow size(weights, 1)$            ▷ Number of Classes
3:     $NumOfWeights \leftarrow size(weights, 2)$         ▷ Number of Weights/Class
4:     $FaddLat = 10$                  ▷ Latency of Multiply and Accumulate
5:     for i:=1 to NumOfClasses do
6:         $Classes(i) \leftarrow 0$
7:     $sum \leftarrow 0$
8:     for i:=1 to NumOfClasses do
9:         $sum = 0$
10:        for j:=1 to NumOfWeights do
11:            $PIPELINE = FaddLat$
12:            $sum \leftarrow sum + input(j) * weights(i, j)$
13:        $Classes(i) \leftarrow sum$
14:    return $Classes$
```

Better Approach To achieve that we will perform the task in a different way. In fact, we will calculate different partial results in parallel for each class. We will end up having the partial results that need to be added to produce a result. This will be achieved by creating a deep adder tree. The appropriate number of different partial results must be equal to the adder delay, meaning 10 in 7.

Cyclic Array We originally solved the problem of how we process the 8-fold weights in a single clock cycle and exploit the resources to do the 8-fold multiplication and addition. Instantly the problem that arises is to explore how we will ensure access to 8-Images stages of the edited image (in a cycle). This could easily be achieved using the array partition of the tool by applying a cyclic partition. This allows us, depending on the factor we will set, in how many consecutive cells of the array we will have access (in a cycle). However, because the array was large (28464), the tool used a large multiplexer system to route access to the memory and thus did not allow us to reach a competent clock (it greatly increased the critical path).

Custom Cyclic Array So we ended up making our own "cyclic partition" helping the tool to achieve the behavior we wanted. To achieve this we created 4 subterranean arrays of the original one. We created 4 and not 8 because each B-RAM has 2 memory channels, meaning 2 memory accesses. We stored weight1,weight2 in the first table, weight3,weight4 in the second etc. in order to be able to have access in 8 consecutive image-stages. Hence we use a same sized B-RAM, but instead of reading from 1 array, we now read from 4 different subterranean arrays by striking the behavior we wanted. Finally, we ended up creating our own "cyclic partition" helping the tool to achieve the behavior we wanted.

Algorithm 7 Matrix Multiplication (Opt)

```
 1: procedure MATRIX MULTIPLICATION($input, weights$)
 2:     $NumOfClasses \leftarrow size(weights, 1)$                ▷ Number of Classes
 3:     $NumOfWeights \leftarrow size(weights, 2)$          ▷ Number of Weights/Class
 4:     $FaddLat = 10$                          ▷ Latency of Multiply and Accumulate
 5:     for i:=1 to NumOfClasses do
 6:         $UNROLL$
 7:         $Classes(i) \leftarrow 0$
 8:     for i:=1 to FaddLat do
 9:         $UNROLL$
10:         $partialSum(i) \leftarrow 0$
11:     for i:=1 to NumOfClasses step=FaddLat do
12:         $PIPELINEII = FaddLat$
13:         for k:=1 to FaddLat do
14:             $UNROLL$
15:             $partialSum(k) \leftarrow 0$
16:         for j:=1 to NumOfWeights do
17:             $partialSum(j) \leftarrow partialSum(j) + input(j) * weights(i, j)$
18:         for v:=1 to FaddLat do
19:             $UNROLL$
20:             $Classes(k) \leftarrow Classes(k) + partialSum(v)$
21:     return $Classes$
```

TABLE 5.4: Second Approach Performance

Modules	**Latency (cycles)**	**Comp. Performance (GFLOPS)**	**Bandwidth (GB/s)**
conv	593596	7.6	1
dense	2871400	4.8	8.23
conv+dense	3464996	5.25	9.23

GB=Gbytes

5.6 Architecture v1.0

In this architecture, we basically integrate the study implemented in Sensitivity Analysis 4. We have been able to reduce the memory footprint of weights to a satisfactory degree with small error. Hence we can speed the processing of our data. Then we managed to make our network a huge pipeline from the image entering and communication of Convolution Layers to the communication of the three Convolutional Layers with the Fully Connected Layer. Afterwards, we perform resource optimizations to be able to fit our network into an FPGA.

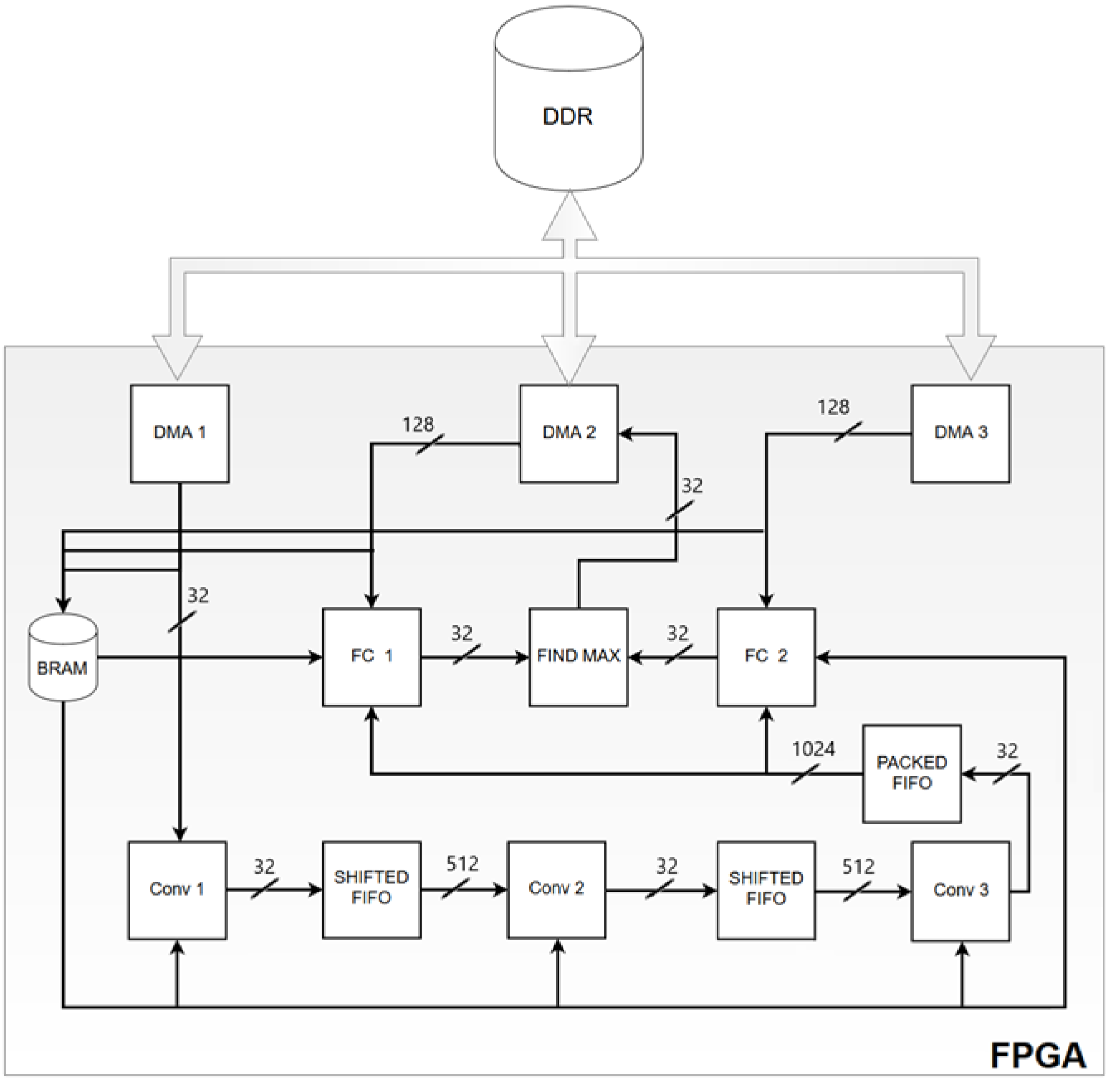


Figure 5.6: Datapath of the Architecture v1.0

5.6.1 Embedding Compressed Weights

Originally, the analysis was done to reduce the memory footprint of the weights that only applies to the fully connected, because this is the main memory bottleneck of the algorithm. Furthermore, compressed weights are used in the I/O streaming interface during the processing. We used 256-bit channel from the memory (2-DMA of 128 bits) based on the previous analysis on memory buses. Each compressed weight has a 4-bit size. Therefore we can stream 64 weights in a cycle(stream read). This gives us a possibility for a huge parallelism at the operation level. To achieve this we extend the previous architecture to operation level parallelism.

5.6.2 Pipelining Convolution Module

The next step is to try to get the most out of all available resources. To accomplish this, a pipeline must be created between convolutional layers as shown in figure 5.7, in such a way that different parts of the image are processed at the same time by the three convolutional entities.

To perform this we need to change the way we obtain the data and output them to the next entity in a way that is suitable for it, to process them in parallel with the previous one. FIFO must be placed between the layers. Thus we create a channel in which every entity will be able to obtain image-parts. We need to introduce some new entities with specific behavior that will satisfy this purpose and will enable pipeline processing.

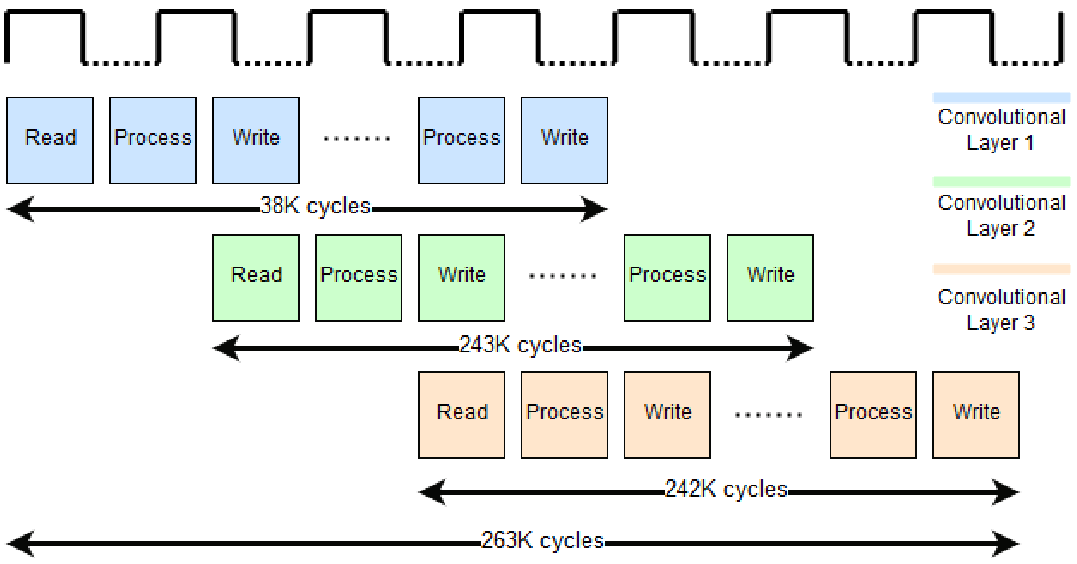


FIGURE 5.7: Convolutional Layers Time-Chart of Architecture v1.0

In table 5.9 we present performance results after the pipeline of Convolutional Layers.

TABLE 5.5: Pipeline Convolutional Layers

Modules	**FLOPS**	**MAC/cycle**
conv1	459K	8
conv2	7.3M	16
conv3	7.3M	16
CONV	15.1M	40(peak)

GB=Gbytes MAC = Multiply and Accumulate FLOPS = Floating Point Operations

Shifted FIFO

The utility of this entity is double. Originally, this implements a custom behavior of a Shifted FIFO. It receives data from the previous layer, and when it reaches 16x8 it pushes it in 16 packages and streams them through a FIFO (512 bits) to the next Layer to start the processing. Then in the next cycle, the next module will need the last 16x7 + the new 16 data. Therefore here comes the shift register. At the same time, we do not have to store some of the stages of the image (like before in B-RAM), we just go through FIFOs and read them there. The shift register functionality could be added to the next layer. However, this has led to great critical paths and that's why its behavior has taken place in a different entity.

Task Level Parallelism

Having created all the structures to activate the pipeline, the next step is to use the Xilinx dataflow pragma. However, trying to incorporate it, we realized that there was a feature, sequential and parallel processing in different modules at the same time, that was not supported. As a result, when the tool noticed any FIFO or some kind of stream behavior, it was trying to turn it into task level parallelism.

Sequential and Parallel Processing Accordingly, we should find a workaround to incorporate this feature. This problem was presented in the insertion of the initial data (image, weights, bias) in the 1st module. Essentially we were reading at several points from the same stream and stored in different B-RAMs. The solution was to create a custom mutual exclusion for reading in this stream (mutex). Therefore we forced the tool to read in a specific manner the information and store it in specific B-RAMs. On the other hand, the image is passed to an internal FIFO to continue the flow in the next modules. Below in 8 we present the behavior above.

Algorithm 8 Sequential Parallel with MUTEX

```
1: procedure SEQUENTIAL PARALLEL WITH MUTEX($InStream$)
2:     $SKernel_1 \leftarrow size(kernel_1)$        ▷ Size of $kernel_1$
3:     $SKernel_2 \leftarrow size(kernel_2)$        ▷ Size of $kernel_2$
4:     $SKernel_3 \leftarrow size(kernel_3)$        ▷ Size of $kernel_3$
5:     $SBias_1 \leftarrow size(bias_1)$        ▷ Size of $bias_1$
6:     $SBias_2 \leftarrow size(bias_2)$        ▷ Size of $bias_2$
7:     $SBias_3 \leftarrow size(bias_3)$        ▷ Size of $bias_3$
8:     $SImage \leftarrow size(Image)$        ▷ Size of Image
9:     $TotalSize \leftarrow SKernel_1 + SKernel_2 + SKernel_3 + SBias_1 * 3$ ▷ Total Size
10:    for i:=1 to TotalSize do
11:        if $i \leq SKernel_1$ then
12:            $Kernel_1(i) \leftarrow InStream.read$
13:        else if $i \leq SKernel_1 + SKernel_2$ then
14:            $k \leftarrow i - SKernel_1$
15:            $Kernel_2(k) \leftarrow InStream.read$
16:        else if $i \leq SKernel_1 + SKernel_2 + SKernel_3$ then
17:            $k \leftarrow i - SKernel_1 + SKernel_2$
18:            $Kernel_3(k) \leftarrow InStream.read$
19:        else if $i \leq SKernel_1 + SKernel_2 + SKernel_3 + SBias_1$ then
20:            $k \leftarrow i - SKernel_1 + SKernel_2 + SKernel_3$
21:            $Bias_1(k) \leftarrow InStream.read$
22:        else if $i \leq SKernel_1 + SKernel_2 + SKernel_3 + SBias_1 * 2$ then
23:            $k \leftarrow i - SKernel_1 + SKernel_2 + SKernel_3 + SBias_1$
24:            $Bias_2(k) \leftarrow InStream.read$
25:        else if $i \leq SKernel_1 + SKernel_2 + SKernel_3 + SBias_1 * 3$ then
26:            $k \leftarrow i - SKernel_1 + SKernel_2 + SKernel_3 + SBias_1 * 2$
27:            $Bias_3(k) \leftarrow InStream.read$
28:        else if $i \leq TotalSize$ then
29:            $StreamImage.Write(InStream.read)$
30:    return $Kernel_1, Kernel_2, Kernel_3, Bias_1, Bias_2, Bias_3, StreamImage$
```

5.6.3 Pipelining two Modules

The next step is to be able to pipeline the two main modules of the Convolution Layers and Fully Connected networks, as shown in figure 5.8. When we achieve this, we will essentially gain the latency of the smallest module in total latency 5.3.

$$Lat_{TOTAL} = max(Lat_{CONV}, Lat_{FC} + LatencyInterval) \tag{5.3}$$

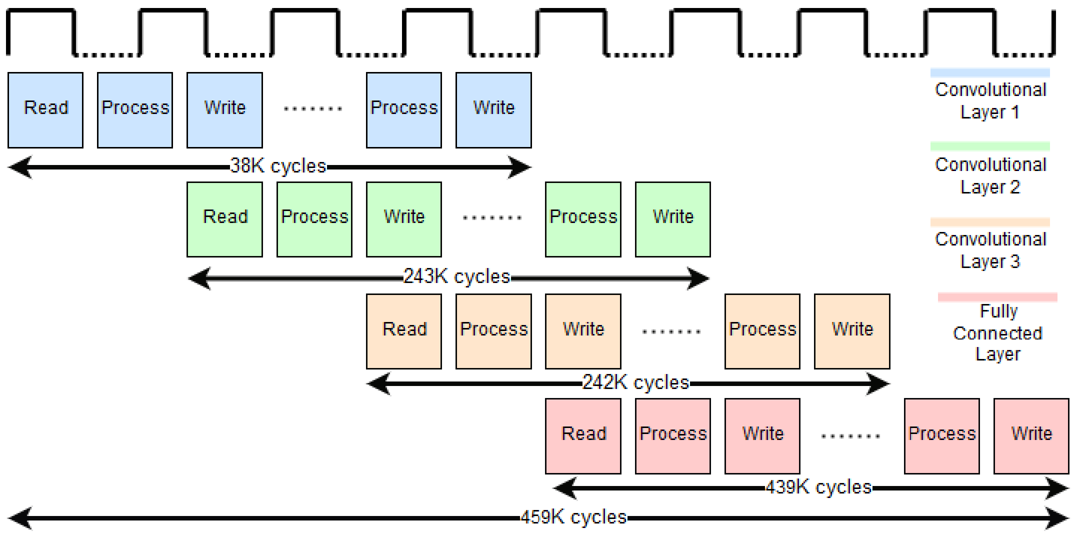


FIGURE 5.8: CNN Time-Chart of Architecture v1.0

To accomplish this, major modifications have to be made to the 2 modules.

Convolution Transformations

Convolution should transform the way we export the data. For these purposes, a new entity was created by FifoToPackedFifo. This entity gets the processed image and packs it into 32 items. That's why we would need a 1024bits FIFO. This feature is not supported by HLS, so we came with a workaround by having two 512 FIFOs each containing half of the data. After the initial latency interval, every 32 * 8 = 512 cycles, it feeds the Fully Connected Layer through FIFO.

Fully Connected Transformations

With regard to the Fully Connected Layer, the way the processed image is imported should be transformed according to the above changes. Next, we need to try to gain the most out of the data that comes, or else we can drive the system into huge stalls and thus destroy the Pipeline. The way the FC has so far operated was to calculate the results for each class individually. For each class, we had to read the whole processed image again. This cannot be continued because the whole processed image will be available only when the Conv module has finished its processing. This would mean the pipeline concept will be lost. Hence we have to change the way we calculate the classes. Another dimension of parallelism needs to be added. There has to be another important change in the structure of FC. We will create 2 FC Modules where each will calculate different classes. So in 400 cycles, we will have partial results for the 800 classes. To achieve this, we need to perform a Memory Layout Transformation where it will also transform the way that weights are streamed.

Algorithm 9 Matrix Multiplication (Opt)

```
procedure MATRIX MULTIPLICATION($InStream, ConvStream, input, weights$)
    $NumOfClasses \leftarrow size(weights, 1)$                ▷ Number of Classes
    $NumOfWeights \leftarrow size(weights, 2)$                ▷ Number of Weights/Class
    $FaddLat = 10$                                            ▷ Latency of Multiply and Accumulate
    for i:=1 to NumOfClasses do
        $UNROLL$
        $Classes(i) \leftarrow Dense_{bias}(i)$
    for i:=1 to FaddLat do
        $UNROLL$
        $partialSum(i) \leftarrow 0$
    for i:=1 to NumOfWeights/32 do
        $ConvStream.read()$
        for j:=1 to NumOfClasses step=FaddLat do PIPELINE=10
            for k:=1 to FaddLat do
                $InStream.read()$
                $sum64 = inp1 * w1 + inp2 * w2 + ... + inp63 * w63 + inp64 * w64$
                $Classes[j + k] += sum64$
    return $Classes$
```

In fact, a consumer-producer model has been created, where we feed at a rate of 1/256 cycles and consume at a rate of 1/ 400. This could create stalls in the first module. To avoid this speech in their communication, a large FIFO has been placed

Another problem that appeared was that we had to divide the weights to 32 sets. We possessed 64 weights and would calculate 2 classes in parallel in each module in a cycle (a total of 64 multiples in parallel). Hence the weights for each class are 28464 must be divided into 32 sets. To accomplish this, a small zero-padding of 16 elements has to be added. We also examined the method of processing them completely without using zero-padding and the rest of them sequentially but led worse results.

In table 5.6 we present performance results after the pipeline of Convolutional Layers.

Table 5.6: Pipeline Convolutional with Fully Connected Layer

Modules	FLOPS	MAC/cycle
conv	15.1M	40(peak)
fc	46.6M	64
CNN	60.7M	104(peak)

GB=Gbytes MAC = Multiply and Accumulate FLOPS = Floating Point Operations

5.6.4 Resource Optimizations

It is important to ensure that we use efficiently the resources at our disposal.

- We used the array map where implicitly many small B-RAMs combined to create large ones while maintaining their functionality.
- We have noticed that HLS UNROLL was bound many Flip Flops and LUTs. These decreased considerably by manually unrolling. The tool has this behavior because it can not be guaranteed the exact number of iterations it will run.
- The appropriate DSPs were used through the directive resource.
- Fixed calculations where they are performed many times we compute them once and assign them to defines variables, otherwise the tool binds resources to calculate them each time.

Table 5.7: Architecture v1.0 Performance

Modules	Latency (cycles)	Comp. Performance (GFLOPS)	Bandwidth (GB/s)
conv	263685	17.13	1
dense	439590	31.1	8.23
conv+dense	475590	38.3	9.23

GB=Gbytes

5.7 Architecture v2.0

After the successful completion of the first architecture, we realized that there was an opening for more parallelism. So we decided to introduce another dimension of parallelism, the use of batching. Instead of computing the results for an image, we calculate for two images in parallel.

5.8 Resource Optimizations

The first thought is to insert another instance of the already existing accelerator to implement Batch 2. This would, however, lead to a doubling of resources, which would make it impossible to routing to VIVADO IDE. Consequently, the solution is integrating batch 2 into a single architecture in HLS avoiding duplication of resources and helping the tool get better routing.

5.8.1 Batching

The weights will be streaming the same way. Instead of performing calculations for one image, we will arrange for both. The I/O will also not be increased significantly because the image is very small compared to the weights we stream in. More specifically, the I/O will grow to 0.00012%. The reason why we can not proceed to larger batches is that resources don't allow us.

Table 5.8: Architecture v2.0 Performance

Modules	**Latency (cycles)**	**Comp. Performance (GFLOPS)**	**Bandwidth (GB/s)**
conv	264685	34.25	1
dense	441590	62	8.23
conv+dense	477590	76.5	9.23

GB=Gbytes

5.9 Porting to QFDB

Architecture 1 and 2 have been implemented as described above to run on ZCU-102. To enable the two architectures to run at QFDB, we created 4 Instances of Accelerators, ending in Batch 4 and Batch 8 respectively. Porting didn't require many transformations because there was FPGA consistency. The only substantial differences are that there is no SD on the QFDB resulting in the use of JTAG to send the data. Finally, ARM communication with our machine was done through JTAG (instead of U-ART) using the coresight protocol.

In table 5.9 we present performance results for the QFDB. More specifically for the Architecture v2.0 (Batch 8).In peak performance we achieve 416 GFLOPS/s, when all modules are executed in parallel.

TABLE 5.9: Pipeline Convolutional Layers

Modules	**FLOPS**	**MAC/cycle**	**GFLOPS/s**
conv	120.8	8	57.4
fc	364.8M	16	104
CNN	485.6M	832(peak)	265

GB=Gbytes MAC = Multiply and Accumulate FLOPS = Floating Point Operations

Chapter 6

Results

In this chapter, we will present the results of our work. These results were obtained from the 2 different architectures we propose. These two architectures were tailored to the needs of each different ZCU-102 (released March 2015) and QFDB platforms (essentially 4 ZCUs in parallel) to make the most of our resources.

6.1 Specification of Compared Platforms

Comparisons were made with CPU and GPU platforms. The CPU used was the Intel i7 7700HQ (released January 2017) and the GPU was NVIDIA Quadro K2200 (released August 2014). Platforms of a similar generation were used. More specifically due to the need of the application, the most important factor was high performance with low power consumption as we aim to a satellite-based application.

6.1.1 Intel i7 7700HQ

Below we present tables with the specifications of both CPU 6.1 and GPU 6.2 platforms

TABLE 6.1: Intel i7 7700HQ Specifications

Cores	**Threads**	**Max Turbo Frequency**	**TDP**	**Max Memory Bandwidth**	**Lithography**
4	8	3800 MHz	45W	37.5 GB/s	14nm

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

Table 6.2: NVIDIA Quadro K2200 Specifications

CUDA Cores	GPU Memory	Clock Frequency	Memory Interface	Memory Bandwidth	Power Consumption
640	4GB GDDR5	1124 MHz	128-bit	80 GB/s	60W

6.1.2 NVIDIA Quadro K2200

CUDA

Compute Unified Device Architecture is a technology developed by Nvidia that accelerates GPU computation processes. With CUDA, researchers and developers can send high-level code (C, C++, and Fortran) directly to the GPU without using assembly code. This lets them take advantage of parallel computing in which thousands of threads, can execute simultaneously.

6.2 Power Consumption

Power consumption refers to the required energy per unit time, supplied to the current system to perform a task. Power consumption is usually measured in units of Watts (W) or kiloWatts (kW). To be exact, when we consider about power, we need to measure all the power that the machine on which the GPU the CPU and the FPGA runs. Therefore for the GPU, it runs on a machine that consumes 300 Watts, while the CPU is at 100 Watts.

6.3 Energy Consumption

Energy consumption refers to the energy was required to perform a task in a specific time. To calculate energy:

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

Power = required power , Time= required time to complete the task. Energy consumption is usually measured in units of Joule (J) or kiloJoule (kJ).

6.4 Throughput and Latency Speedup

In computer architecture, the notion speedup is the factor that measures the relative performance of two systems processing the same task. More specifically, it is the improvement in speed of execution of a problem executed on two different architectures. The notion of speedup was established by Amdahl's law [3], which was particularly focused on parallel processing. However, speedup can be used more generally to show the effect on performance after any resource enhancement.

Latency is the time that a systems requires to perform a single task.

$$Latency = \frac{1}{v} = \frac{T}{W}, \tag{6.2}$$

v is the execution speed of the task,
T is the execution time of the task,
W is the execution workload of the task

Throughput is the maximum rate of processing or production of a specific problem.

$$Throughput = r * v * A = \frac{r * A * W}{T} = \frac{r * A}{L}, \tag{6.3}$$

r is the execution density,
A is the execution capacity,

Speedup can be defined for two different types of quantities: latency and throughput.

$$S_{latency} = \frac{L_1}{L_2} = \frac{T_1 * W_2}{T_2 * W_1}, \tag{6.4}$$

$$S_{throughput} = \frac{Thr_2}{Thr_1} \tag{6.5}$$

6.5 Architecture v1.0

In this section, we will present the results of the Architecture v1.0 ported to both platforms single-FPGA(ZCU-102) and Quad-FPGA (QFDB). These results report resource utilization, performance over latency, throughput, power, and energy matters. Finally, we compare results with GPU and CPU.

Table 6.3: Architecture v1.0

Clock Frequency	LUT Usage (%)	FF Usage (%)	DSP Usage (%)	BRAM Usage (%)	BUFG
300	39	39	19	18	1

LUT = LookUp Table, FF= Flip Flop, DSP= Digital Signal Processor, BRAM=Block RAM, BUFG resource is one of the most expensive resources in FPGA. It is used for sending the clock across the clock network in the design.

6.5.1 ZCU-102

In the following tables, we present results for the architecture v1.0, ported on the ZCU-102 comparing to CPU and GPU. Comparing to CPU we get latency (2533x) and throughput (180x) speedup. Furthermore, we are much more power and energy efficient (9.1x and 1636x respectively). On the other hand, compared to the GPU we grant speedup in latency (20x) while we are worse in throughput (0.31x) due to the GPU's big batching. In addition, we are more efficient on power and energy metrics (27.2x and 8.5x respectively).

To extract the following results, we used the entire dataset that we had at our disposal, ie the 10000 images.

Table 6.4: Architecture v1.0 comparison with CPU and GPU

	ZCU-102	CPU	GPU
Clock Frequency(MHz)	300	3800	1124
Throughput(Images/s)	628	3.47	2000
Latency(s)	0.003	7.6	0.06
GFLOPS	38.3	0.21	122.55
Total On-chip Power(Watt)	11	100	300
Energy Consumption(Joule)	175	288K	1.5K
Images/Joule	56.8	0.035	6.66

We use the entire data-set 10K images.

Table 6.5: Speedup over GPU and CPU

ZCU102	GPU	CPU
Latency speedup	20x	2533x
Throughput speedup	0.33x	180x

We use the entire data-set 10K images.

Table 6.6: Energy and Power Efficiency over GPU and CPU

ZCU-102	GPU	CPU
Power Efficiency	27.2x	9.1x
Energy Efficiency	8.5x	1636x

We use the entire data-set 10K images.

6.5.2 QFDB

To complete porting to QFDB we got 4 instances from the existing implementation on ZCU-102. In the following tables, we perceive results for the architecture v1.0, ported on the QFDB comparing to CPU and GPU. Compared to the CPU we get latency (2533x) and throughput (712x) speedup. Furthermore, we are much more power and energy efficient (9.1x and 1618x respectively). Compared to the GPU we grant speedup in latency (20x) and throughput (1.24x) (now we use batch 4). In addition, we are more efficient on power and energy metrics (27.2x and 8.5x respectively).

To extract the following results, we used the entire dataset that we had at our disposal, ie the 10000 images.

Table 6.7: Architecture v1.0 comparison with CPU and GPU

	QFDB	CPU	GPU
Clock Frequency(MHz)	300	3800	1124
Throughput(Images/s)	2640	3.47	2000
Latency(s)	0.003	7.6	0.06
GFLOPS	163	0.21	122.55
Total On-chip Power(Watt)	44	100	300
Energy Consumption(Joule)	166	288 K	1.5 K
Images/Joule	60	0.035	6.66

We use the entire data-set 10K images.

Table 6.8: Speedup over GPU and CPU

QFDB	GPU	CPU
Latency speedup	20x	2533x
Throughput speedup	1.32x	760x

We use the entire data-set 10K images.

Table 6.9: Energy and Power Efficiency over GPU and CPU

QFDB	**GPU**	**CPU**
Power Efficiency	6.8x	2.27x
Energy Efficiency	8.4x	1618x

We use the entire data-set 10K images.

6.6 Architecture v2.0

In this section, we will present you the results of the Architecture v2.0 ported to both platforms single-FPGA(ZCU-102) and Quad-FPGA (QFDB). These results report resource utilization, performance over latency, throughput, power, and energy matters. Finally, we compare results with GPU and CPU.

Table 6.10: Architecture v2.0

Clock Frequency	**LUT Usage (%)**	**FF Usage (%)**	**DSP Usage (%)**	**BRAM Usage (%)**	**BUFG**
240	64	58	38	16	5

LUT = LookUp Table, FF= Flip Flop, DSP= Digital Signal Processor, BRAM=Block RAM, BUFG resource is one of the most expensive resources in FPGA. It is used for sending the clock across the clock network in the design.

6.6.1 ZCU-102

In the following tables, we present results for the architecture v2.0, ported on the ZCU-102 comparing to CPU and GPU. Comparing to CPU we get latency (2533x) and throughput (312x) speedup. Furthermore, we are much more power and energy efficient (7.32x and 2286x respectively). On the other hand, compared to the GPU we grant speedup in latency (20x) while we are worse in throughput (0.55x) due to the GPU's big batching (we use batch-2). In addition, we are more efficient on power and energy metrics (22x and 11.9x respectively).

Table 6.11: Architecture v2.0 comparison with CPU and GPU

	ZCU-102	**CPU**	**GPU**
Clock Frequency(MHz)	240	3800	1124
Throughput(Images/s)	1084	3.47	2000
Latency(s)	0.003	7.6	0.06
GFLOPS	76.5	0.21	122.55
Total On-chip Power(Watt)	13.66	100	300
Energy Consumption(Joule)	126	288 K	1.5 K
Images/Joule	79.36	0.035	6.66

We use the entire data-set 10K images.

Table 6.12: Speedup over GPU and CPU

ZCU102	**GPU**	**CPU**
Latency speedup	20x	2533x
Throughput speedup	0.55x	312x

We use the entire data-set 10K images.

Table 6.13: Energy and Power Efficiency over GPU and CPU

ZCU-102	**CPU**	**GPU**
Power Efficiency	7.32x	22x
Energy Efficiency	2286	11.9x

We use the entire data-set 10K images.

6.6.2 QFDB

In the following tables, we present results for the architecture v2.0, ported on the QFDB comparing to CPU and GPU. Compared to the CPU we get latency (2533x) and throughput (1249x) speedup. Furthermore, we are much more power and energy efficient (9.1x and 1618x respectively). Compared to the GPU we grant speedup in latency (20x) and throughput (2.17x) (now we use batch 4). In addition, we are more efficient on power and energy metrics (27.2x and 8.4x respectively).

TABLE 6.14: Architecture v2.0 comparison with CPU and GPU

	QFDB	**CPU**	**GPU**
Clock Frequency(MHz)	240	3800	1124
Throughput(Images/s)	4334	3.47	2000
Latency(s)	0.003	7.6	0.06
GFLOPS	265	0.21	122.55
Total On-chip Power(Watt)	54.64	100	300
Energy Consumption(Joule)	126	288K	1.5K
Images/Joule	79.36	0.035	6.66

We use the entire data-set 10K images.

TABLE 6.15: Speedup over GPU and CPU

QFDB	**GPU**	**CPU**
Latency speedup	20x	2533x
Throughput speedup	2.17x	1249x

We use the entire data-set 10K images.

TABLE 6.16: Energy and Power Efficiency over GPU and CPU

QFDB	**CPU**	**GPU**
Power Efficiency	1.83x	5.49x
Energy Efficiency	2286x	11.9x

We use the entire data-set 10K images.

6.7 Final Performance

Below we present throughput and latency speedups over different batches 6.1 and different number of images in the given data-set. 6.2. Furthermore in figure 6.3 we present power and energy efficiency over GPU. Figure 6.4 presents Architecture v1.0 ,v2.0 and optimized v2.0.

The first approach to be able to integrate the batch 2 is to instantiate the existing IPs. This, however, has led to a doubling of all resources. Hence it was decided to change the architecture of each IP to integrate the batch 2. After this step, we were able to implement resource optimizations in the HLS as presented and analyzed in the previous chapter. In figure 6.4 we present the results of the optimizations.

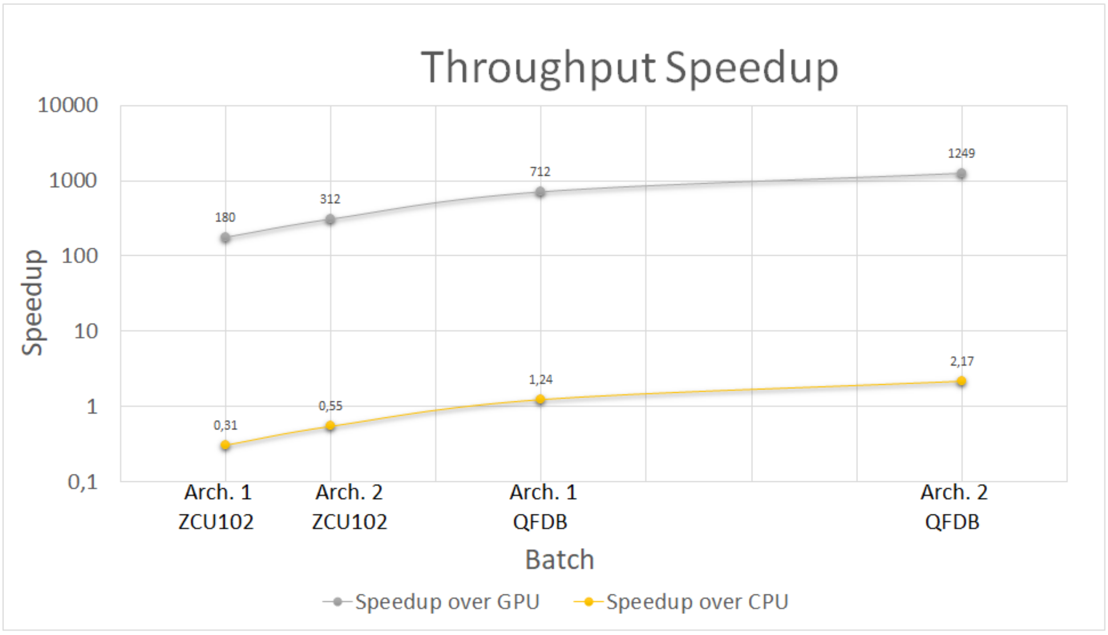


FIGURE 6.1: Throughput Speedup for different batches

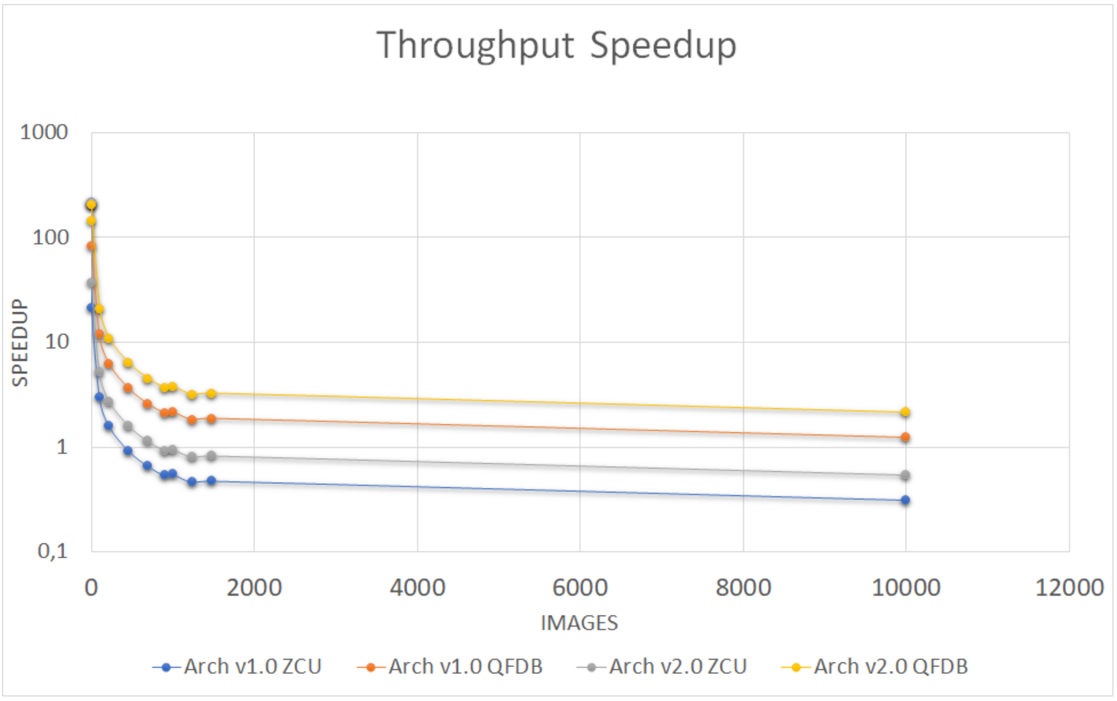


FIGURE 6.2: Throughput Speedup for different Number of Images

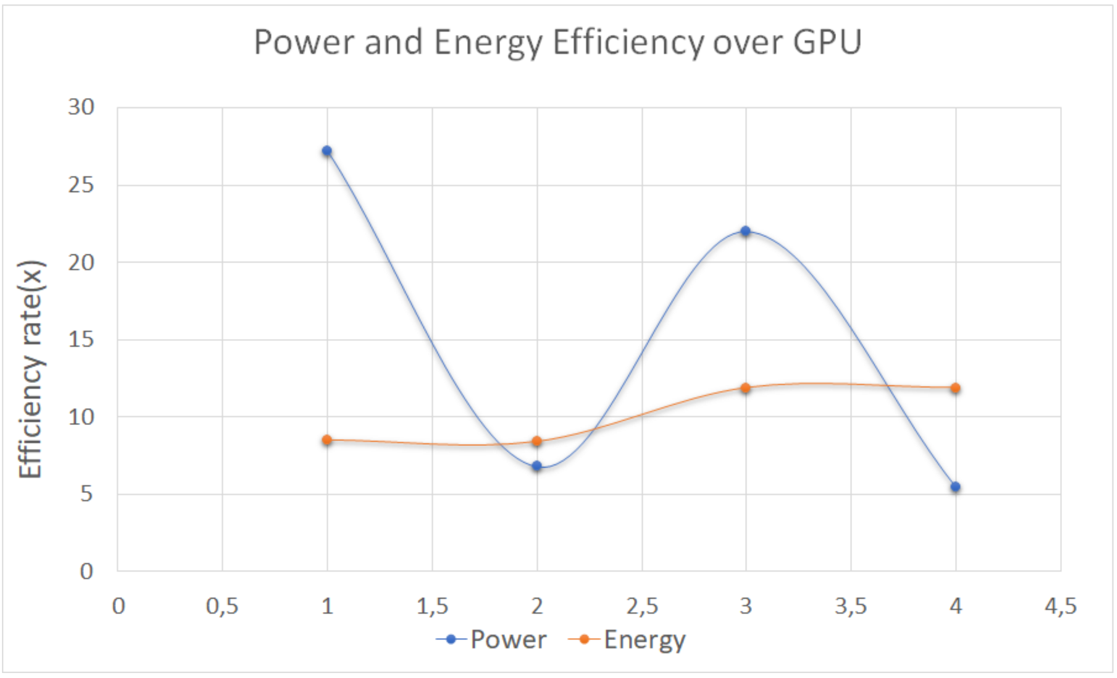


FIGURE 6.3: Power and Energy Efficiency over GPU: Both architectures

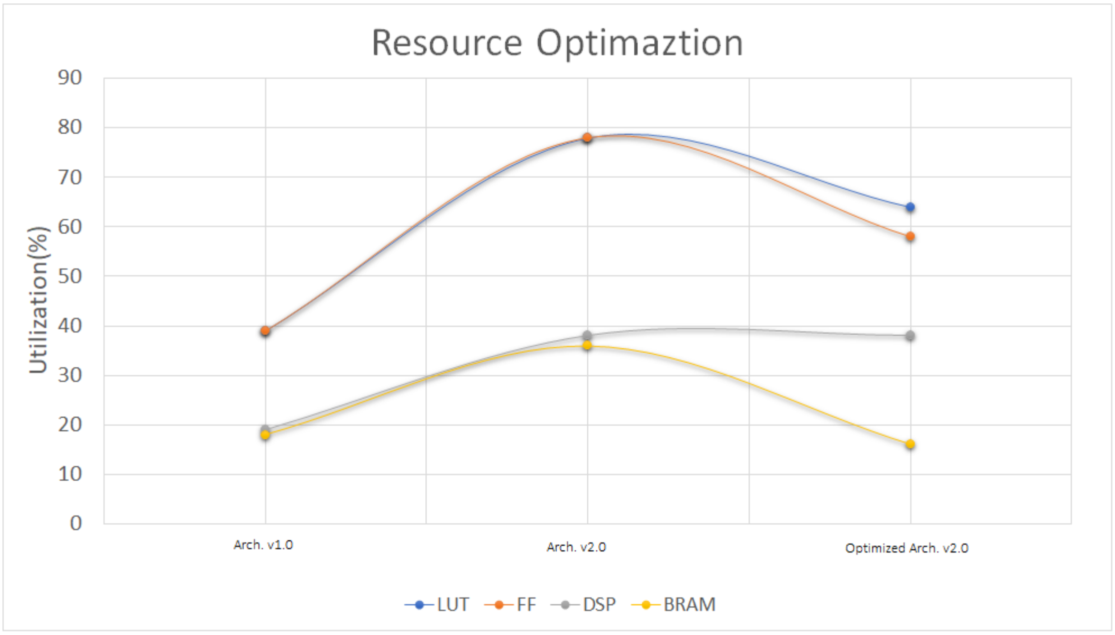


FIGURE 6.4: Utilization after the use of Resource Optimizations

Chapter 7

Conclusions and Future Work

This chapter will sum up and evaluate this Thesis' work. Furthermore opportunities for future work will be rising and how this Thesis's Robustness Analysis creates scope for further optimizations (for CNN) in FPGA designs.

7.1 Conclusions

In recent years Convolutional Neural Networks (CNNs) have been shown extremely growth due to their effectiveness at complex image recognition problems. The purpose of this Thesis was to accelerate a specific-CNN for aerospace subject using Reconfigurable Logic(FPGA). After carrying out Robustness Analysis computational workloads and memory accesses are analyzed, as well as compression methods and algorithmic optimizations to exploit FPGA parallelism. At the level of neurons, optimizations of the convolutional and fully connected layers are explained and compared. At the network level, approximate computing optimization methods are examined limited by not reducing the accuracy of the network. The platforms were used are ZCU102 and QFDB(a custom 4-FPGA platform developed at FORTH). The implemented accelerator was managed to achieve 20x latency speedup, 2.17x throughput speedup and 11.9x energy efficient over GPU NVIDIA-Quadro-K2200.

7.2 Future Work

As a future work, the methods that have been proposed in Robustness Analysis, Pair, Quad Compression and SLC, can be implemented in hardware to take full advantage of the huge compression rate they give us and further reduce I/O which is the main bottleneck of the most FPGAs' designs. Subsequently, it could work on algorithmic redundancies by exploiting the pruning that has been implemented. Furthermore, if we're concerned about High-Performance Computing, we could scale up to more FPGAs (eg Mezzanine 8-QFDB) and expect an 8x near speedup due to the parallelism of the application. On the other hand, if we moved to a larger

FPGA we could increase the parallelism internally to the FPGA by adding larger Batch to the images by presenting an almost linear speedup. Finally, the use-case of the application is to travel into space with the Euclid satellite, therefore it would be important to study FPGAs where they have resistance to space radiation and porting to one of them. The FPGA's suitability for space is that it more energy efficient than GPU and we also managed to get throughput speedup over a Nvidia GPU K2200.

References

[1] National Security Agency. *The National Security Agency: Missions, Authorities, Oversight and Partnerships*. Tech. rep. 2013. URL: http://www.nsa.gov/public_info/_?les/speeches_testimonies/2013_08_09_the_nsa_story.pdfintroduction.

[2] Geoffrey E. Hinton Alex Krizhevsky Ilya Sutskever. *ImageNet Classification with Deep Convolutional Neural Networks*. 2012. URL: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[3] Gene Amdahl. *Validity of the single processor approach to achieving large scale computing capabilities*. 1967. URL: www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf.

[4] Bo Chen-Dmitry Kalenichenko Weijun Wang Tobias Weyand Marco Andreetto Hartwig Adam Andrew G. Howard Menglong Zhu. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2011. URL: https://arxiv.org/pdf/1704.04861.

[5] Sergei Arthur David Vassilvitskii. *Dynamically scaled fixed point arithmetic*. 1991. URL: https://ieeexplore.ieee.org/document/160742/.

[6] Y. Bengio. *Practical recommendations for gradient-based train- ing of deep architectures*. 2012.

[7] G. Bertone. *Particle dark matter: Observations, models and searches*. 2010.

[8] C. M. Bishop. "Pattern Recognition and Machine Learning". In: (2006).

[9] Yangqing Jia Pierre Sermanet Scott Reed Dragomir Anguelov Dumitru Erhan Vincent Vanhoucke Andrew Rabinovich Christian Szegedy Wei Liu. *Going Deeper with Convolutions*. 2014. URL: https://arxiv.org/pdf/1409.4842.pdf.

[10] Dan; Ueli Meier; Jonathan Masci; Luca M. Gambardella; Jurgen Schmidhuber Ciresan. *Flexible, High Performance Convolutional Neural Networks for Image Classification*. 2011. URL: http://people.idsia.ch/~juergen/ijcai2011.pdf.

[11] Dan; Ueli Meier; Jonathan Masci; Luca M. Gambardella; Jurgen Schmidhuber Ciresan. *Multi-column deep neural networks for image classification.* 2012. URL: ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6248110.

[12] Ueli Schmidhuber Jürgen Ciresan Dan; Meier. *Multi-column Deep Neural Networks for Image Classification.* 2012. URL: https://arxiv.org/pdf/1202.2745.pdf.

[13] Ronan Collobert and Jason Weston. *A unified architecture for natural language processing: Deep neural networks with multitask learning.* 2008.

[14] Nan Cui. *Applying Gradient Descent in Convolutional Neural Networks.* 2016. URL: http://iopscience.iop.org/article/10.1088/1742-6596/1004/1/012027/pdf.

[15] G. E. Hinton D. E. Rumelhart and R. J. Williams. *Learning internal representations by error propagation.* 1985.

[16] Ron Meir Daniel Soudry Itay Hubara. *Expectation Backpropagation: Parameter-Free Training of Multilayer Neural Networks with Continuous or Discrete Weights.* 2014. URL: https://papers.nips.cc/paper/5269-expectation-backpropagation-parameter-free-training-of-multilayer-neural-networks-with-continuous-or-discrete-weights.pdf.

[17] V. Sreekanth Annapureddy Darryl D. Lin Sachin S. Talathi. *Fixed Point Quantization of Deep Convolutional Networks.* 2016. URL: https://arxiv.org/pdf/1511.05236.pdf.

[18] M. Sami E. J. Copeland and S. Tsujikawa. *Dynamics of Dark Energy.* 2006.

[20] Bin Liu Fengfu Li. *Ternary weight networks.* 2016. URL: https://arxiv.org/pdf/1605.04711.pdf.

[21] Matthew W. Moskewicz Forrest N. Iandola. *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size.* 2016. URL: https://arxiv.org/pdf/1602.07360.pdf.

[22] Intel FPGA. *Intel Stratix 10 Variable Precision DSP Blocks User Guid.* 2017.

[23] Debbie Marr Ganesh Venkatesh Eriko Nurvitadhi. *ACCELERATING DEEP CONVOLUTIONAL NETWORKS USING LOW-PRECISION AND SPARSITY.* 2016. URL: https://arxiv.org/pdf/1610.00324.pdf.

[24] J. Gantz and D. Reinsel. "The Digital Universe Decade—Are You Ready". In: *Hopkinton, MA, USA: EMC* (2010). URL: http://link.aip.org/link/?RSI/69/1236/1.

[25] John Gantz and David Reinse. “Extracting Value from Chaos”. In: *EMC Corporation* 62 (2011). URL: https://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf.

[26] Ross Girshick. *Fast R-CNN*. 2015.

[28] Philipp Gysel. “Ristretto: Hardware-Oriented Approximation of Convolutional Neural Networks”. 2017. URL: https://arxiv.org/pdf/1605.06402.pdf.

[29] Wong Hartigan. *Algorithm AS 136: A K-Means Clustering Algorithm*. 1979. URL: https://www.jstor.org/stable/2346830?seq=1#page_scan_tab_contents.

[30] S. Hochreite. *The vanishing gradient problem during learning recurrent neural nets and problem solutions*. 2013.

[31] Rajesh Ranganath Andrew Y. Ng Honglak Lee Roger Grosse. *Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations*. 2009. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.149.802&rep=rep1&type=pdf.

[32] Rajesh Ranganath Andrew Y. Ng Honglak Lee Roger Grosse. “Convolutional Neural Networks (LeNet) – DeepLearning 0.1 documentation”. In: (2013). URL: http://deeplearning.net/tutorial/lenet.html.

[34] K. Hornik. *Approximation capabilities of multilayer feedforward networks, Neural networks vol.4*. 1991.

[35] D. Huffman. *A Method for the Construction of Minimum-Redundancy Codes*. 1952. URL: https://ieeexplore.ieee.org/document/4051119/.

[36] Urs Hölzle Jeff Dean. *Build and train machine learning models on our new Google Cloud TPUs*. 2017. URL: https://www.blog.google/products/google-cloud/google-cloud-offer-tpus-machine-learning/.

[37] Jefkine. “Backpropagation In Convolutional Neural Networks”. In: (2016). URL: http://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/.

[38] Evan Shelhamer Jonathan Long and Trevor Darrell. *Fully Convolutional Networks for Semantic Segmentation*. 2015.

[39] Norm Jouppi. *Google supercharges machine learning tasks with TPU custom chip*. 2017. URL: https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html.

[40] Andrew Zisserman Karen Simonyan. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. URL: https://arxiv.org/pdf/1409.1556.pdf.

[41] John R.; Bennett Koza. “How can computers learn to solve problems without being explicitly programmed”. In: (1996). URL: https://link.springer.com/chapter/10.1007/978-94-009-0279-4_9.

[42] Andrew D. Back Lawrence Steve; C. Lee Giles; Ah Chung Tsoi. *Face Recognition: A Convolutional Neural Network Approach*. 1997. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.92.5813.

[43] Y. LeCun. *Generalization and network design strategies,” Connectionism in perspective*. 1989.

[44] Vikas Chandra Liangzhen Lai Naveen Suda. *Deep Convolutional Neural Network Inference with Floating-point Weights and Fixed-point Activations*. 2017. URL: https://arxiv.org/pdf/1703.03073.pdf.

[45] “Machine Learning: What it is and why it matters”. In: (2016). URL: https://www.sas.com/it_it/insights/analytics/machine-learning.html.

[46] Heikki Mannila. “Data mining: machine learning, statistics, and databases”. In: (1996).

[47] Christopher Manning. *Introduction to information retrieval*. 2008. URL: https://www.worldcat.org/title/introduction-to-information-retrieval/oclc/190786122.

[48] Bernard Marr. *How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read*. 2016. URL: https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/#63f264f160ba.

[49] Paul Barham Martın Abadi Ashish Agarwal. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2015. URL: http://download.tensorflow.org/paper/whitepaper2015.pdf.

[51] Daniel Soudry Matthieu Courbariaux Itay Hubara. *Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or 1*. 2016. URL: https://arxiv.org/pdf/1602.02830.pdf.

[52] Jean-Pierre David Matthieu Courbariaux Yoshua Bengio. *BinaryConnect: Training Deep Neural Networks with binary weights during propagations*. 2016. URL: https://arxiv.org/pdf/1511.00363.pdf.

[53] Yoshua Bengio Matthieu Courbariaux Jean-Pierre David. *TRAINING DEEP NEURAL NETWORKS WITH LOW PRECISION MULTIPLICATIONS.* 2016. URL: https://arxiv.org/pdf/1412.7024.pdf.

[54] Katsuhiko Mori-Yusuke Mitari Yuji Kaneda Matusugu Masakazu. *Subject independent facial expression recognition with robust face detection using a convolutional neural network.* 2011. URL: https://www.sciencedirect.com/science/article/pii/S0893608003001151?via\%3Dihub.

[55] Microsoft. *Learning Semantic Representations Using Convolutional Neural Networks for Web Search – Microsoft Research.* 2015. URL: https://www.microsoft.com/en-us/research/publication/learning-semantic-representations-using-convolutional-neural-networks-for-web-search/?from=http\%3A\%2F\%2Fresearch.microsoft.com\%2Fapps\%2Fpubs\%2Fdefault.aspx\%3Fid\%3D214617.

[56] Joseph Redmon Mohammad Rastegari Vicente Ordonez and Ali Farhadi. *XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks.* 2016. URL: https://arxiv.org/pdf/1603.05279.pdf.

[57] Cliff Young Norman P. Jouppi. *In-Datacenter Performance Analysis of a Tensor Processing Unit.* 2017. URL: https://arxiv.org/ftp/arxiv/papers/1704/1704.04760.pdf.

[58] Hao Su Jonathan Krause Sanjeev Satheesh Sean Ma Zhiheng Huang Andrej Karpathy Aditya Khosla Michael Bernstein Olga Russakovsky Jia Deng et al. *Subject independent facial expression recognition with robust face detection using a convolutional neural network.* 2015.

[59] Tayler Hetherington Patrick Judd Jorge Albericio. *Reduced-precision strategies for bounded memory in deep neural nets.* 2017. URL: https://arxiv.org/pdf/1511.05236.pdf.

[60] Mohammad Motamedi Soheil Ghiasi Philipp Gysel. *Hardware-oriented Approximation of Convolutional Neural Networks.* 2016. URL: https://arxiv.org/pdf/1604.03168.pdf.

[61] Sasanka Potluri. *CNN based high performance computing for real time image processing on GPU.* 2011. URL: https://ieeexplore.ieee.org/document/6024781/.

[62] S. Arduini J.-L. Augueres R. Laureijs J. Amiaux. *Euclid definition study report.* 2011.

[63] Bruno Moraes Panagiotis Tsakalides Radamanthys Stivaktakis Grigorios Tsagkatakis. *Convolutional Neural Networks for Spectroscopic Redshift Estimation on Euclid Data*. 2018. URL: DuringSubmissionprocedure.

[64] Foster Provost Ron Kohavi. "Glossary of terms". In: (1998). URL: http://ai.stanford.edu/~ronnyk/glossary.html.

[65] F. Rosenblat. *Principles of neurodynamics. perceptrons and the theory of brain mechanism*. 1961. URL: http://www.dtic.mil/dtic/tr/fulltext/u2/256582.pdf.

[66] Arthur Samuel. *Some Studies in Machine Learning Using the Game of Checkers*. 1959. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.368.2254.

[67] Fabian Schilling. *The Effect of Batch Normalization on Deep Convolutional Neural Networks*. 2016. URL: https://kth.diva-portal.org/smash/get/diva2:955562/FULLTEXT01.pdf.

[68] Feng Chen Shuang Wu Guoqi Li. *Training and Inference with integers in Deep Neural Networks*. 2018. URL: https://arxiv.org/pdf/1802.04680.pdf#cite.zhu2016trained.

[69] Zekun Ni Xinyu Zhou He Wen Yuheng Zou Shuchang Zhou Yuxin Wu. *DOREFANET: Traininf Low BitWidth Convolutional Neural Networks with low BitWdith Gradients*. 2017. URL: https://arxiv.org/pdf/1606.06160.pdf.

[70] Karen Simonyan and Andrew Zisserman. *Very deep convolutional networks for large-scale image recognition*. 2014. URL: https://arxiv.org/pdf/1409.1556.pdf.

[71] William J. Dally Song Han Huizi Mao. *Deep Compression: Compressing Deep Neural Networks with Pruning Trained Quantization and Huffman Coding*. 2016. URL: https://arxiv.org/pdf/1510.00149.pdf.

[72] Paul A. Merolla Steve K. Esser Rathinakumar Appuswamy. *Backpropagation for Energy-Efficient Neuromorphic Computing*. 2014. URL: https://papers.nips.cc/paper/5862-backpropagation-for-energy-efficient-neuromorphic-computing.pdf.

[73] Google Brain team. *Google Just Open Sourced TensorFlow, Its Artificial Intelligence Engine*. 2015. URL: https://www.wired.com/2015/11/google-open-sources-its-artificial-intelligence-engine/.

[74] Google Brain team. *TensorFlow, Google's Open Source AI, Signals Big Changes in Hardware Too*. 2015. URL: https://www.wired.com/2015/11/googles-open-source-ai-tensorflow-signals-fast-changing-hardware-world/.

[77] Walter H.Pitts Warren S. McCulloch. *A logical calculus of the ideas immament in nervous activity*. 1943. URL: http://www.cse.chalmers.se/~coquand/AUTOMATA/mcp.pdf.

[78] Feng Yan Wei Wen Cong Xu. *TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning*. 2017. URL: https://papers.nips.cc/paper/6749-terngrad-ternary-gradients-to-reduce-communication-in-distributed-deep-learning.pdf.

[79] D. Williamson. *How Slow is the K-means Method?* 2006. URL: https://dl.acm.org/citation.cfm?doid=1137856.1137880.

[80] Jeff Donahue Sergey Karayev Jonathan Long Ross Girshick Sergio Guadarrama Trevor Darrell Yangqing Jia Evan Shelhamer. *Caffe: Convolutional Architecture for Fast Feature Embedding*. 2016. URL: https://arxiv.org/pdf/1408.5093.pdf.

[81] Yoshua Bengio Yann LeCun and Georey Hinton. *Deep learning*. 2015.

[82] Philémon Brakel Saizheng Zhang Cesar Laurent Yoshua Bengio Ying Zhang Mohammad Pezeshki. *Towards end-to-end speech recognition with deep convolutional neural networks*. 2017.

[84] Qiong Cai Paolo Faraboschi Zhaoxia Deng Cong Xu. *Reduced-Precision Memory Value Approximation for Deep Learning*. 2017. URL: https://www.labs.hpe.com/techreports/2015/HPL-2015-100.pdf.

External Links

[19] *EuroExa*. URL: https://euroexa.eu/.

[27] *Google opens up about its Tensor Processing Unit*. URL: https://www.datacenterdynamics.com/news/google-opens-up-about-its-tensor-processing-unit/.

[33] *Horizon 2020*. URL: https://ec.europa.eu/programmes/horizon2020/.

[50] *MATLAB*. URL: https://www.mathworks.com/.

[75] *TensorFlow*. URL: https://www.tensorflow.org/.

[76] *Vivado Design Suite - HLx Editions*. URL: https://www.xilinx.com/products/design-tools/vivado.html.

[83] *ZCU102 User Guide*. URL: https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf.