TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Convolutional Neural Network Optimizations using Knowledge Distillation for Applications on Hardware Accelerators

Author:
Apostolos-Nikolaos VAILAKIS

Thesis Committee:
Prof. Apostolos DOLLAS
Associate Prof. Michail G. LAGOUDAKIS
Dr. Vassilis PAPAEFSTATHIOU (FORTH-ICS)

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

in the

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

March 8, 2022

Abstract

Convolutional Neural Network Optimizations using Knowledge Distillation for Applications on Hardware Accelerators

by Apostolos-Nikolaos VAILAKIS

Over the last decade, Convolutional Neural Networks have gained popularity amongst the scientific community, due to their versatility and performance in an all-growing domain of applications. Recent advances in computational power have enabled researchers to develop and train CNNs of exponential complexity, capable of solving problems previously considered unattainable. From facial recognition, to climate analysis and self-driving cars, CNNs constantly prove their value in the field of Machine Learning. Deploying however such models in real-world applications presents a significant challenge. While training complex CNNs requires high performance computing systems, inference may need to be performed at much tighter computational budgets. This has motivated the scientific community to develop both hardware architectures capable of efficiently executing CNNs, as well as methodologies for compressing networks. Hardware accelerators focused on edge applications opt for lower precision arithmetics (network quantization), which in turn simplifies the computational engines and greatly reduces the memory footprint of the models. This however can result in staggering accuracy losses. Recent advances in quantization-aware training techniques promise to mitigate these effects. Centered around DenseNet, a state-of-the-art CNN developed for image classification, this study performs an in-depth analysis of Quantization Aware Knowledge Distillation (QKD), a promising technique which combines quantization-aware training with knowledge distillation. Additionally, a comparison in inference performance between a CPU, a GPU and a Xilinx DPU is conducted, the latter of which employs 8-bit integer arithmetic. To achieve this, QKD is integrated in Xilinx's Vitis-AI workflow. Achieving a minimum of $9\times$ latency speedup and $4\times$ power efficiency compared to all other platforms using Xilinx's DPU, indicates that effective model compression and quantization, coupled with dedicated hardware architectures can produce highly capable systems for edge applications.

Acknowledgements

I would like to express my deepest appreciation to my supervisor, Prof. Apostolos Dollas, for his valuable support, irreplaceable guidance and patience during the procedure of this thesis. His lectures on digital logic design, a first semester course, immediately attracted me to the field of digital hardware architecture. Since then, he has been a source of inspiration and expertise, of which I will always be grateful.

Moreover, I would like to thank the rest of my thesis committee, Associate Prof. Michail G. Lagoudakis, and Dr. Vasilis Papaefstathiou, for evaluating my work.

Furthermore, I would like to deeply thank Dr Gregory Tsagkatakis (FORTH) for his excellent collaboration, and providing deep insight regarding the field of Deep Learning. Doing so gave me the opportunity to broaden my horizons beyond my current field of expertise in the domain of Artificial Intelligence. I would also like to thank the rest of the CARV group. Dr. Christos Kozanitis for his contributions during the course of this thesis and Dr. Aggelos Ioannou for his valuable insight in Xilinx tools and technologies.

A special thanks to my friend and colleague, Tzanis Fotakis, who generously provided his expertise in FPGA accelerated CNNs, and helped with deployment on the ZCU 102 evaluation board.

Last but not least i would like to express my deepest gratitude to my family and friends for their support throughout the years of my studies.

Apostolos-Nikolaos Vailakis,
Chania 2022

Contents

Abstract **iii**

Acknowledgements **v**

Contents **vii**

List of Figures **xiii**

List of Tables **xv**

List of Algorithms **xvii**

List of Abbreviations **xix**

1 Introduction **1**
- 1.1 Motivation 2
- 1.2 Scientific Contributions 2
- 1.3 Thesis Outline 3

2 Theoretical Background **5**
- 2.1 Machine Learning 5
  - 2.1.1 Common types of supervised learning algorithms . . . 6
    - Regression Algorithms 6
    - Decision Trees 6
    - Instance-Based Algorithms 6
  - 2.1.2 Common types of unsupervised learning algorithms . 6
    - Clustering Algorithms 6
    - Association Algorithms 6
- 2.2 Deep Learning 7
  - 2.2.1 Multi-Layer Perceptron (MLP) 8
  - 2.2.2 Artificial Neurons 8
  - 2.2.3 Activation (Transfer) Functions 9
- 2.3 Optimizing (Training) Neural Networks 11

2.3.1 Gradient Descent 12
2.3.2 Loss Functions 13
Regression Loss Functions 14
Classification Loss Functions 14
2.3.3 Backpropagation 15
2.3.4 Stochastic Gradient Descent (SGD) and Batching 17
2.3.5 Batch Normalization 17
2.4 Model Overfitting 18
2.4.1 Early stopping 19
2.4.2 Training with more data 19
2.4.3 Data augmentation 19
2.4.4 Regularization 20
Weight decay 20
Dropout 20
2.5 Convolutional Neural Networks (CNNs) 21
2.5.1 Convolution Layers 22
1D Convolution Layers 22
2D Convolution Layers 23
Multi-Channel Convolution 24
2.5.2 Subsampling Layers 26
Average Pooling 26
Max Pooling 26

3 Related Work **27**
3.1 Training Datasets 27
3.2 CNN Architectures 28
3.2.1 LeNet 28
3.2.2 AlexNet 28
3.2.3 VGG 29
3.2.4 ResNet 29
Pre-Activated Residual Blocks 30
3.2.5 DenseNet 31
3.3 Adaptive Learning Rate 32
3.3.1 Time-Based Decay 33
3.3.2 Step Decay 33
3.3.3 Exponential Decay 34
3.3.4 Learning Rate Warmup 34
3.4 Knowledge Distillation (KD) 34

3.5 Quantization 35
3.5.1 Post-Training Quantization 37
3.5.2 Quantization-Aware Training (QAT) 37
3.6 Quantization-Aware Knowledge Distillation 38
3.6.1 Phase 1: Self-Studying (SS) 38
3.6.2 Phase 2: Co-Studying (CS) 38
3.6.3 Phase 3: Tutor-Studying (TS) 38

4 Tools and Platforms 41
4.1 Deep Learning Software Tools 41
4.1.1 TensorFlow & Keras 41
4.1.2 PyTorch 41
4.1.3 TensorBoard 42
4.2 Neural Networks on Hardware 42
4.2.1 CPU 43
4.2.2 GPU 43
4.2.3 TPU 44
4.2.4 FPGA 46
4.3 Vitis AI 47
4.3.1 Xilinx DPU 47
4.3.2 Vitis-AI Quantizer 47
4.3.3 Vitis-AI Compiler 48
4.4 Specifications of Utilized Platforms 48
4.4.1 Intel i5-8600K 49
4.4.2 Nvidia RTX-2060 Super 8GB 49
4.4.3 Xilinx Zynq UltraScale+ MPSoC 50
ZCU102 Evaluation Board 50
Quad FPGA Daughter Board 51
4.5 Thesis Approach 52

5 Robustness Analysis of CNNs 53
5.1 Software 53
5.1.1 Datasets and Dataloaders 53
5.1.2 Optimizers 54
5.1.3 Learning Rate Schedulers 54
5.1.4 Loss Functions 54
Knowledge Distillation Loss Function 54
5.1.5 Quantization Schemes 55
Static Quantization 55

Fake Quantization . . . 55
5.1.6 Pytorch CUDA Integration . . . 55
5.1.7 Configuration Files . . . 56
5.1.8 Base Classes . . . 56
Validator . . . 56
TrainerSS . . . 56
TrainerTS . . . 57
TrainerCS . . . 58
5.1.9 Quantization . . . 59
8-bit Activation Quantizer . . . 60
8-bit Weights Quantizer . . . 60
4-bit Activation Quantizer . . . 60
4-bit Weights Quantizer . . . 60
5.1.10 Data Augmentation . . . 60
5.2 Methodology . . . 61
5.2.1 Models . . . 61
DNN . . . 61
KL_MLP . . . 61
DenseNet . . . 61
5.3 Baseline Experiments . . . 62
5.3.1 DNN . . . 62
5.3.2 KL_MLP . . . 63
5.3.3 DenseNet . . . 63
5.4 Knowledge Distillation Experiments . . . 64
5.4.1 DNN . . . 64
5.4.2 KL_MLP . . . 65
5.4.3 DenseNet . . . 66
5.5 Post-Training Quantization . . . 67
5.6 Quantization-Aware Training . . . 68
5.6.1 KL_MLP . . . 68
5.6.2 DenseNet . . . 68
5.7 Quantization-Aware Knowledge Distillation . . . 69
5.7.1 MNIST . . . 70
5.7.2 CIFAR-10 . . . 71
5.7.3 CIFAR-100 . . . 72
5.8 Memory Footprint . . . 73
5.9 Conclusions . . . 73

6 FPGA Implementation **75**
6.1 DenseNet Modifications 75
6.2 Vitis-AI Toolchain 77
6.2.1 Vitis-AI Quantizer 77
6.2.2 Xilinx DPUCZDX8G 78
DSP Double Data Rate 79
DPU Configuration 80
RAM Usage 81
Channel Augmentation 82
Depthwise Convolution 83
Average Pool 83
Relu Type 84
Softmax 84
DSP Usage 84
6.3 DPUCZDX8G Implementation 85
6.4 Evaluating The Model 85

7 Results **89**
7.1 Performance Metrics 89
7.1.1 Latency 89
7.1.2 Throughput 89
7.1.3 Power and Energy Consumption 89
7.2 Overall Performance 90
7.2.1 CPU FP Representation 90
7.2.2 CPU 8-bit Representation 92
7.2.3 GPU 93
7.2.4 Xilinx DPU 94
7.2.5 Comparisons 96

8 Conclusions and Future Work **101**
8.1 Conclusions 101
8.2 Future Work 102
8.2.1 Incremental Quantization 102
8.2.2 Quantization Scheme 102
8.2.3 Tested Platforms 102
8.2.4 Model Sizes 103

References **105**

List of Figures

2.1 Simplified representation of a Multi-Layer Perceptron 8
2.2 Artificial Neuron Model 9
2.3 Common Activation Functions 11
2.4 Illustration of Gradient Descent 12
2.5 Overfitting on data 19
2.6 Simple Dropout Illustration 21
2.7 Typical CNN Architecture 21
2.8 2-Dimentional Convolution Layer 23
2.9 Edge detection using convolution kernels 24
2.10 Multi-Channel 2D convolution with single-channel output . . 25
2.11 Multi-Channel 2D convolution with multi-channel output . . 25
2.12 Illustration of Max-Pooling and Average-Pooling Layers . . . 26

3.1 CNN Architectures 29
3.2 Residual Block 30
3.3 34 Layer ResNet 30
3.4 Pre-Activated Residual Block 31
3.5 A deep DenseNet with three dense blocks 31
3.6 Learning Rate Decay 33
3.7 Comparison of different learning rate decay methods. 34

4.1 TPU Cores and Chips 44
4.2 Architecture of a Matrix Multiplier Unit (MXU) 45
4.3 Vitis AI Stack 47
4.4 Vitis AI Compiler Framework 48
4.5 The ZCU102 Evaluation Kit 51
4.6 QFDB Architecture 52

5.1 QAT and QKD Performance on CIFAR-10 for 8-bit and 4-bit quantization schemes 72

6.1 Xilinx DPU Architecture 79
6.2 DSP Double Data Rate 80

6.3 DPU Computation Parallelism 81

7.1 CPU Latency 91
7.2 CPU Throughput 91
7.3 CPU Energy Consumption 91
7.4 Quantized CPU Latency 92
7.5 Quantized CPU Throughput 92
7.6 Quantized CPU Energy Consumption 93
7.7 GPU Latency 93
7.8 GPU Throughput 94
7.9 GPU Energy Consumption 94
7.10 DPU Latency 95
7.11 DPU Throughput 95
7.12 DPU Energy Consumption 95
7.13 DenseNet-21 Inference Performance 97
7.14 DenseNet-37 Inference Performance 97
7.15 DenseNet-63 Inference Performance 98

List of Tables

3.1 DenseNet Architectures 32
3.2 Common Quantization Data Types 35
3.3 Comparison of Quantization Techniques 37

4.1 Intel i5-8600K processor specifications 49
4.2 NVIDIA RTX 2060 Super specifications 49
4.3 Zynq UltraScale+ Features 50

5.1 KL_MLP model variants 61
5.2 Custom DenseNet Architectures 62
5.3 DNN test results on MNIST 63
5.4 KL_MLP test results on MNIST 63
5.5 DenseNet test results on CIFAR-10 64
5.6 DenseNet test results on CIFAR-100 64
5.7 DNN Knowledge-Distillation test results on MNIST 65
5.8 KL_MLP Knowledge Distillation test results on MNIST 66
5.9 DenseNet-21 Knowledge Distillation test results on CIFAR-10 66
5.10 Post-Training Quantization impact on baseline models 67
5.11 KL_MLP QAT test results on MNIST 68
5.12 DenseNet 8-bit QAT test results 69
5.13 DenseNet 4-bit QAT test results 69
5.14 KL_MLP Performance comparison of training methods. 70
5.15 DenseNet Performance comparison of training methods for CIFAR-10. 71
5.16 DenseNet Performance comparison of training methods for CIFAR-100. 72

6.1 Modified DenseNet evaluation on Cifar-10 76
6.2 Modified DenseNet evaluation on Cifar-100 76
6.3 VAI Quantizer Model Accuracy 78
6.4 DPUCZDX8G Architectures Specifications 81
6.5 DPUCZDX8G Memory Usage 82

6.6 DPUCZDX8G Channel Augmentation Resource Usage 82
6.7 DPUCZDX8G Depthwise Convolution Resource Usage 83
6.8 DPUCZDX8G Average Pooling Resource Usage 83
6.9 DPUCZDX8G Low DSP Resource Usage 84
6.10 DPUCZDX8G Resource Requirements on ZCU-102 85

7.1 DenseNet-21 Performance Results 96
7.2 DenseNet-37 Performance Results 96
7.3 DenseNet-63 Performance Results 97

List of Algorithms

1 Quantization-aware Knowledge Distillation 39

2 TrainerSS.train_epoch() 57
3 TrainerTS.train_epoch() 58
4 TrainerCS.train_epoch() 59

5 DPU Runner 86

List of Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
ASIC	Application Specific Integrated Circuit
BN	Batch Normalization
BRAM	Block Random Access Memory
CE	Cross Entropy
CNN	Convolutional Neural Network
CPU	Central Processor Unit
CS	Co Studying
DDR	double Data Rate
DDR4	Double Data Rate type 4 memory
DPU	Deep Processing Unit
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
GPU	Graphic Processor Unit
HBM	High Bandwidth Memory
HDL	Hardware Description Language
KD	Knowledge Distillation
KLD	Kullback Leibler Divergence
KNN	K Nearest Neighbor
MAC	Multiply and Accumulate
MAE	Mean Absolute Error
MPSoC	Multiprocessor system on a chip
MSE	Mean Square Error
MSLE	Mean Squared Logarithmic Error
MXU	Matrix Multiply Unit
NLP	Natural Language Processing
PC	Processor Cluster
PE	Processing Element
PL	Programmable Logic
PS	Processing System

QAT	Quantized Aware Training
QKD	Quantized aware Knowledge Distillation
ReLU	Rectified Linear Unit
SGD	Stochastic Gradient Descent
SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessor
SS	Self Studying
TPU	Tensor Processing Unit
TS	Tutor Studying
VAI	Vittis Artificial Intelligence
VART	Vittis-AI Runtime
XIR	Xilinx Internal Representation

Dedicated to my family and friends…

Chapter 1

Introduction

Artificial Intelligence (AI) and Machine Learning (ML) have become indispensable tools for solving complex problems in all fields of software engineering. Specifically, Deep Neural Networks (DNN), a type of machine learning algorithm, have increasingly gained popularity over the last decade due to their versatility and outstanding performance. However, as the complexity of the problem increases, so does the complexity of DNNs. Fortunately, the structure of these models comes with high data parallelism. Therefore, they can be expanded in the space domain, in other words, they can utilize more hardware resources to cut down on needs from the time domain.

While training DNNs requires high performance computing systems, inference may need to be performed at much tighter computational budgets. For example, facial recognition networks for image post-processing can be executed on high-power servers, but similar networks designed for edge applications, such as integrating facial recognition for enhanced auto-focus of smartphone cameras, should be capable of efficiently running in mobile hardware.

Deploying DNNs at the edge thus requires a series of optimizations, by compressing the model, and more tightly coupling its characteristics with the available hardware resources. Hardware however, can too be optimized for deploying such models.

Another possible application constraint is inference latency, the elapsed time between feeding a neural network some information, and receiving a result. For example, image recognition networks are the backbone of autonomous driving. Here, the delay introduced between an event being captured by the imaging system, and the hardware completing the required computations, contributes to the autopilot's reaction time.

1.1 Motivation

As previously mentioned, DNNs are best characterized by their exceptional performance and increased computational requirements, making hardware acceleration a necessity. Alhough inferencing such models using conventional Central Processing Units (CPUs) is possible, doing so is considered the least efficient solution.

Graphic Processing Units on the other hand are optimized for data parallel throughput computations. making them vastly superior for large vector operations required by DNNs. However, they can be costly to scale up, and their power consumption may prove prohibitive for edge applications.

Application Specific Integrated Circuits (ASICs) line Google's Tensor Processing Unit (TPU)[] can provide the best performance in terms of data parallelism and energy efficiency. Unfortunately, such systems are expensive to develop and trade flexibility for performance.

Field Programmable Gate Arrays (FPGAs) bridge the gap between flexibility and performance. Although their speed cannot be compared to a purpose-built ASIC, their flexibility allows for experimentation with different architectures that best fit task at hand.

Modern implementations of DNN accelerators for edge applications heavily rely on lower arithmetic precision, which in turn hinders the model's performance. To mitigate this phenomenon, recent advances in Quantization-Aware Training (QAT) yield promising results. Originally proposed by Kim et al. [1] in 2019, Quantization-Aware Knowledge Distillation (QKD) is one of the most promising methodologies to lower the arithmetic precision of a model, while retaining its accuracy.

1.2 Scientific Contributions

In this work, an in-depth analysis of the QKD approach is performed in order to recover key insights which include the impact of different network architecture complexities, as well as the impact of different weight quantization settings, focusing on DenseNet [2], a state of the art Convolutional Neural Network (CNN) designed for image classification.

Furthermore, a performance analysis of three different platforms is conducted. The platforms used in this work are an intel i5-8600K CPU, an RTX2060-Super GPU, and a Xilinx DPU based accelerator implemented in the ZCU-102 evaluation board, the latter of which is designed to perform calculations for quantized neural networks. To achieve this, QKD is integrated in Xilinx's Vitis-AI workflow, producing highly accurate models compatible with Xilinx's AI engine.

Final figures indicate that effective model compression and quantization, coupled with dedicated hardware architectures can produce highly capable systems. Designed to extract performance benefits associated with lower precision arithmetics, Xilinx's DPU presents both latency and energy efficiency advantages, despite being implemented on the oldest and slowest chip compared to all other platforms. All the while retaining comparable accuracy, on-par with full-precision based models.

1.3 Thesis Outline

- **Chapter 2 - Theoretical Background:** The theoretical background of Machine Learning, with emphasis on Convolutional Neural Networks, is described.
- **Chapter 3 - Related Work:** The related work on the field of Convolutional Neural Network training and various quantization techniques are explored. Additionally different hardware implementations of neural networks are presented.
- **Chapter 4 - Robustness Analysis of CNNs:** This chapter presents the work done for the evaluation and characterization of various model compression and quantization algorithms. A plethora of training methodologies as presented in chapters 2 and 3 are employed for building efficient and accurate image classification models, used as baseline for further experimentation.
- **Chapter 5 - FPGA Implementation:** In this chapter, a set of modified DenseNet models are prepared for execution on a Xilinx DPU using the Vitis-AI development stack.
- **Chapter 6 - Results:** Metrics, such as the throughput, latency, energy efficiency, are compared between the various available technologies and platforms.

- **Chapter 7 - Conclusions and Related Work:** This chapter aims to present the conclusions of this dissertation. Also, proposals for future work indicated by the research are suggested.

Chapter 2

Theoretical Background

2.1 Machine Learning

Machine learning is an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. Machine learning focuses on the development of computer programs that can access data and use it to learn for themselves.

Image recognition, speech recognition and synthesis, medical diagnosis and autonomous driving are just a few of the ever growing list of hard real-world problems that have experienced constant breakthroughs in recent years due to such algorithms.

In machine learning, algorithms are *trained* to find patterns and features in massive amounts of data in order to make decisions and predictions based on new data. Such algorithms can benefit from the increasing developments in big data and are proving more useful as computing becomes more powerful and affordable.

Machine learning is achieved through several methods, and can be divided in two distinct types, based on the training data being unlabeled or labeled. These learning methods are called unsupervised and supervised respectively. Although supervised learning tends to yield better results when sufficient labeled data are provided, generating such data can prove extremely difficult and time consuming. Semi-supervised learning occurs when only part of the given data are labeled, and can provide a good trade-of when supervised learning is not feasible.

2.1.1 Common types of supervised learning algorithms

Regression Algorithms

Used to understand relationships in data, regression algorithms like *"Linear Regression"* and *"Logistic Regression"* model dependencies and correlation between output and input features of a training set. This enables them to accurately predict the output value of new data.

Decision Trees

Based on a tree-like model, decision trees contain conditional control statements to classify using a set of hierarchical decisions on the features. They are used for applying those rules on a set of factors to recommend an action or decision.

Instance-Based Algorithms

These algorithms compare new problem instances with instances seen in training. For example K-Nearest Neighbor (KNN) stores all available cases and classifies new cases by a majority vote of its K more similar instances.

2.1.2 Common types of unsupervised learning algorithms

Clustering Algorithms

Clustering analysis, divides the population of data points in groups, based on a set of feature distinctions. It is a way to find meaningful structure inherent in a set of examples. A commonly used algorithm called K-Means is a technique which tries to minimize the distance of the points in a cluster with their centroid. Clustering can be used for example to classify plants and animals among different species, to categorize different books on the basis of topics and information and many more.

Association Algorithms

Association algorithms find patterns and relationships in data and identify frequent *if-then* relationships called association rules which can be used for prediction and decision making.

2.2 Deep Learning

Deep Learning is a subfield of machine learning concerned with algorithms inspired by the structure and function of the brain called artificial neural networks (ANNs). ANNs consist of many interconnected computing units, called neurons, and are functional approximates that map inputs to outputs. Individual neurons have little intrinsic convergence, but when many neurons work together, their combined effects can show remarkable learning performance.

Nodes are the little parts of the system, similar to the neurons of the human brain. When a stimulus hits them, a process takes place in these nodes. Some of them are connected and marked, and some are not, but in general, nodes are grouped into layers. The system must process layers of data between the input and output to solve a task. The more layers it has to process to generate the result, the deeper the network is considered.

The idea of neural networks was initially conceived in 1943 by neurophysiologist Warren McCulloch and mathematician Walter Pitts. They proposed the McCulloch-Pitts neuron, also known as the Threshold Logic Unit (TLU), which takes inputs and returns whether their weighted sum is above a given threshold. In 1949 in his book *"The Organization of Behaviour"* [3] Donald Hebb took the idea further, proposing that neural pathways strengthen over each succesive use. The first neural network by today's standards was proposed by Frank Rosenblatt. Named the Mark I Perceptron, it was a system based on the McCulloch-Pitts neuron, and its weights would be 'learned' through succesively passed inputs, while minimizing the difference between desired and actual output.

Although they showed promising results, research on ANNs experienced significant stagnation. The age now famously referred to as 'the AI winter' came with the realization that, as such networks deepen, they become exponentially harder to fine-tune using conventional methods. The concept that helped ANNs escape their early infancy is called Backpropagation. Along with Gradient-Descend, they form the backbone of neural networks and are the main methods through which networks adapt and 'learn' the features of a training set.

2.2.1 Multi-Layer Perceptron (MLP)

Multi-Layer perceptrons [4] (sometimes refered to as *vanilla* networks) are the simplest form of a deep artificial neural network. They consist of three or more layers, an input layer, an output layer and at least one hidden layer. Every layer is comprised of artificial neurons that take as input the output of each neuron from the previous layer, whith an exception of the input layer, which takes the input data of the network.

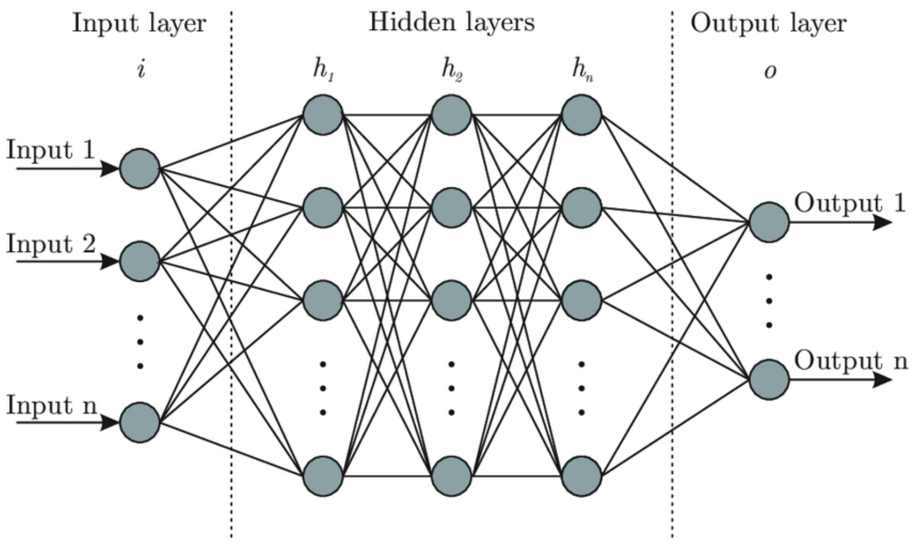


FIGURE 2.1: Simplified representation of a Multi-Layer Perceptron: URL.

2.2.2 Artificial Neurons

Based on the Threshold Logic Unit, artificial neurons [5] take the weighted sum of their inputs as well as a given bias, and use it as the input of a predetermined activation function (transfer function).

The output y of an artificial neuron is defined as:

$$y = F(b + \sum_{i=1}^{I} x_i * w_i) \tag{2.1}$$

Where I the number of neuron inputs x_i, w_i the weight of each input, b the neuron bias and F the activation function.

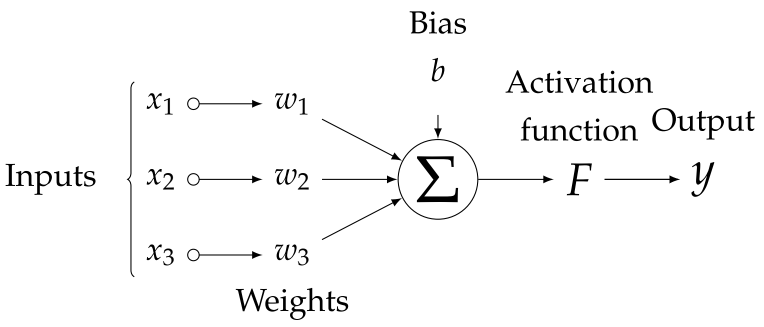


FIGURE 2.2: Artificial Neuron Model.

2.2.3 Activation (Transfer) Functions

The activation function [6] [7] [8] of a neuron is the component that introduces non-linearity to the system. This is necessary to gain any advantages from a multi-layer network architecture, since any multilayer perceptron not using an activation function, or using a linear one, has an equivalent single-layer network. Activation functions are chosen using a variety of criteria, and can significantly impact the performance of the network, as well as its complexity. The most common activation functions are:

- Binary Step function.

  Binary step function is a threshold-based activation function and is mainly used in binary classification problems. Originally used in the McCulloch-Pitts neuron, the output of the neuron is activated when a certain threshold is exceeded.

$$f(x) = \begin{cases} 0 & x <= 0 \\ 1 & x > 0 \end{cases} \tag{2.2}$$

- Sigmoid.

  Also known as the logistic function, the function's output ranges between 0 and 1. The function normalizes the output of each neuron, but does not change in very high or very low inputs.

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.3}$$

- TanH.

Similar to Sigmoid, the hyperbolic tangent (tanh) allows for negative values since it's output ranges between -1 and 1

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.4}$$

- Recti-Linear Unit (ReLU)

  ReLU is the combination of a linear and a binary step function. Due to it's simplistic implementation and non-linearity the function can yield good results while significantly simplifying the network.

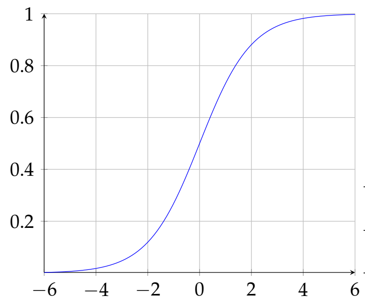
$$f(x) = \begin{cases} 0 & x <= 0 \\ x & x > 0 \end{cases} \tag{2.5}$$

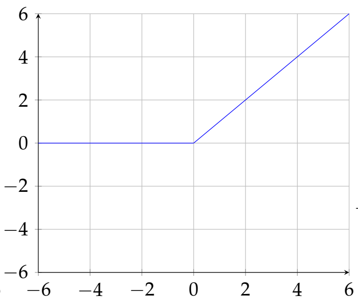
- Softmax

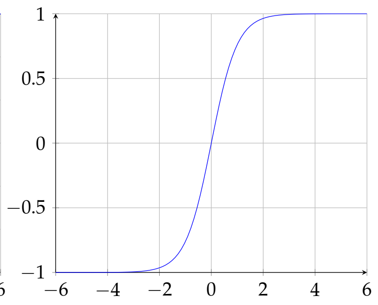
  Based on the logistic function, softmax is often used as the last activation function of a multi-class neural network. It is very similar to the sigmoid function, but normalizes it's output to a probability distribution, ensuring that the sum of all outputs is 1

$$f(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \tag{2.6}$$

  Where K is the number of neurons in this layer, and $\mathbf{x} = (x_1, \ldots, x_K)$ the input vector of the layer.


(A) Sigmoid


(B) ReLU


(C) TanH

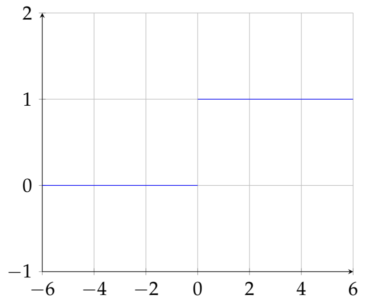

(D) Binary Activation

FIGURE 2.3: Common Activation Functions

2.3 Optimizing (Training) Neural Networks

Deep learning falls under the category of supervised learning. As such, a training dataset needs to be constructed, which plays a significant role to the network's final performance.

The methodology used for effectively optimizing a neural network consists of three main pillars, a loss function, backpropagation and an optimization algorithm. Using these main ingredients an iterative process can be constructed to train the network.

After initializing randomly the weights of each layer, the iterative process begins by feeding the training data through the network. This produces an output, which is then compared to the expected label with the help of a loss function. This loss function quantifies the error of the network, meaning the degree to which the network can accurately classify the input. The problem of training the network can now be described as the problem of minimizing this error.

Given the output of the loss function, a set of error gradients w.r.t each of the network's weights is calculated using the backpropagation algorithm. These gradients are then fed to an optimization algorithm, usually a derivative of gradient descent, which fine-tunes each weight to minimize the error, searching for a local optimum.

This procedure is called an *epoch*. To achieve convergence the algorithm may run for multiple epochs, preferably iterating through a random permutation of the training set each time.

2.3.1 Gradient Descent

Gradient Descent [9] is an optimization algorithm used for finding a local minimum of a differentiable function by iteratively moving in the direction of steepest descent. It is by far the most popular optimization strategy in machine learning. Each iteration is calculated using the gradient of the function and a given learning rate factor. The equation below describes each step the algorithm takes. The current position is θ_t , while θ_{t-1} is the previous one. The η is the learning rate and the gradient term is the direction of the steepest descent.

$$\theta_t = \theta_{t-1} - \eta \nabla_\theta F(\theta_{t-1}) \tag{2.7}$$

Each step is proportional to the learning rate, which plays a significant role in Gradient Descent. Taking small steps can result to the algorithm needing an unecessarily large ammount of iterations before convergence is achieved. On the other hand, increasing the learning rate may result in the algorithm oscilating around the local minima.

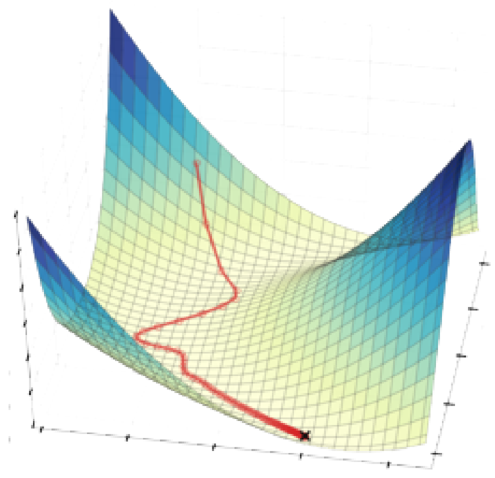


FIGURE 2.4: Illustration of Gradient Descent: URL.

Gradient Descent struggles navigating ravines. Once fallen into one, the algorithm oscillates across the slopes of the ravine, without making much progress towards the local optimum. A simple way to minimize these oscillations and accelerate in the relevant direction is the Momentum technique.

$$v_t = \gamma v_{t-1} - \eta \nabla_\theta F(\theta_{t-1}) \tag{2.8}$$

$$\theta_t = \theta_{t-1} + v_t \tag{2.9}$$

Here a fraction of the update vector of the past time step is added to the current update vector. This acts similarly to the momentum of a ball rolling down a hill, promoting steps in a similar direction. A later revision of this technique is called Nesterov Momentum. Here the gradient term of each step is computed after adding the previous steps momentum. This helps mitigate any large steps in the wrong direction the momentum may promote, since the gradient will now take it into account. The revised iteration step rule is:

$$v_t = \gamma v_{t-1} - \eta \nabla_\theta F(\theta_{t-1} + \gamma v_{t-1}) \tag{2.10}$$

$$\theta_t = \theta_{t-1} + v_t \tag{2.11}$$

2.3.2 Loss Functions

Training of any sorts, whether it occurs on the physical or artificial domain, requires sufficient evaluation of results. This is evident when for example a person is practising on an instrument, or when someone is playing chess. Simple repetition is not enough, people learn by their mistakes, and artificial neural networks are trained using the same principle.

To properly quantify the inaccuracy of a neural network when training it, a loss function needs to be defined. Given a set of inputs, the purpose of such function is to calculate the difference between the network's outputs and the ideal ones. When taining a model, the optimization algorithm tries to minimize the loss function, by updating the model's parameters in each iteration. The choice of loss function must match the framing of the specific predictive modeling problem, such as classification or regression. Some of the most common loss functions used in machine learning are listed below [8]. In the following examples we assume y_i^t, y_i^p as the target and prediction output of the model respectively.

Regression Loss Functions

A regression modeling problem involves predicting a real-valued quantity.

- Mean Squared Error Loss (MSE) [10].

  Is the most commonly used regression loss function. MSE is the sum of squared distances between our target variable and predicted values.

$$MSE = \frac{\sum_{i=1}^{n}(y_i^t - y_i^p)^2}{n} \tag{2.12}$$

- Mean Squared Logarithmic Error Loss (MSLE) [11].

  Similar to MSE, MSLE is the sum of squared distances between the logs of our target variable and predicted values. Here we assume $y^t > -1$ and $y^p > -1$.

$$MSLE = \frac{\sum_{i=1}^{n}(log(y_i^t + 1) - log(y_i^p + 1))^2}{n} \tag{2.13}$$

- Mean Absolute Error Loss (MAE) [12].

  MAE is the sum of absolute differences between our target and predicted variables.

$$MAE = \frac{\sum_{i=1}^{n}|y_i^t - y_i^p|}{n} \tag{2.14}$$

Classification Loss Functions

Classification are those predictive modeling problems where examples are assigned one of many predefined classes. To achieve this the model generates a probability distribution

- Cross-Entropy Loss (CE) [13].

  Cross-entropy loss measures the performance of a classification model whose output is a set of probabilities for each class. Each predicted class probability is compared to the desired output (0 or 1) and a loss is calculated based on the log of their difference.

$$CE = -\sum_{i=1}^{n} y_i^t \log((p_i)), \text{ for n classes} \tag{2.15}$$

  Where p_i is the predicted probability of the i^{th} class.

- Kullback Leibler Divergence Loss (KLD) [14].

  KLD is a measure of the divergence between two probability distributions. This loss function is more common with models that try to approximate a more complex function than simply multi-class classification, but is used in this manner as well.

$$KLD = \sum_{i=1}^{n} y_i^t(\log(y_i^t) - \log(y_i^p)), \text{ for n classes} \tag{2.16}$$

2.3.3 Backpropagation

As previously mentioned, training a neural network using gradient descent is an iterative process, which requires the calculation of the gradient of the loss function with respect to the weights and biases of the entire network. Traditional methods naively calculate the gradients with respect to each weight individually, which is an extremely cumbersome process, and a barrier to larger and more complex predictive models.

Popularized by David E. Rumelhart, Geoffrey E. Hinton & Ronald J. Williams in their work *"Learning representations by back-propagating errors"* [15], backpropagation is used as a general optimization method for performing automatic differentiation of complex nested functions. The name is short for *backward propagation of errors*, which indicates the fact that the gradient is calculated backwards through the network, beginning with the gradient of the final layer of weights and ending with the gradient of the first layer of weights.

For the basic case of a feedforward network, where nodes in each layer are connected only to nodes in the immediate next layer (without skipping any layers), and there is a loss function that computes a scalar loss for the final output, backpropagation can be understood simply by matrix multiplication [16] [17].

Denote:

x: Input vector.

y: Target output.

g: The network.

E: The loss function.

L: The number of layers.

W^l The weights of layer l, where w^l_{jk} is the weight between the k-th node in layer $l-1$ and the j-th node in layer l.

f^l Activation functions at layer l.

z^l: The weighted input of each layer.

a^l: The output of layer l.

Is should be noted that bias terms are not treated specially, since they can be represented as a weight with a fixed input of 1. Furthermore it is assumed that, for the purpose of backpropagation, the loss and activation functions along with their derivatives can be evaluated efficiently.

The network can now be represented as:

$$g(x) := f^L(W^L f^{L-1}(W^{L-1} \cdots f^1(W^1 x) \cdots)) \tag{2.17}$$

For each input, the loss function is dependent on the target output, and the actual output of the network:

$$E(y_i, g(x_i)) \tag{2.18}$$

Combining the above the error of a given input vector is:

$$E(y, f^L(W^L f^{L-1}(W^{L-1} \cdots f^2(W^2 f^1(W^1 x)) \cdots))) \tag{2.19}$$

During the forward pass of the network, the activation a^l as well as the derivatives $(f^l)'$ can be cached for use during the backwards pass. Using the chain rule, the derivatives of the loss in relation to the inputs can be constructed.

$$\frac{dE}{da^L} \cdot \frac{da^L}{dz^L} \cdot \frac{dz^L}{da^{L-1}} \cdot \frac{da^{L-1}}{dz^{L-1}} \cdot \frac{dz^{L-1}}{da^{L-2}} \cdots \frac{da^1}{dz^1} \cdot \frac{\partial z^1}{\partial x}. \tag{2.20}$$

The above represent the derivatives of the loss function, activation functions and the matrices of the weights.

$$\frac{dE}{da^L} \cdot (f^L)' \cdot W^L \cdot (f^{L-1})' \cdot W^{L-1} \cdots (f^1)' \cdot W^1. \tag{2.21}$$

To calculate the error gradient ∇, the matrices are transposed and the order of multiplication is reversed.

$$\nabla_x E = (W^1)^T \cdot (f^1)' \cdot \dots \cdot (W^{L-1})^T \cdot (f^{L-1})' \cdot (W^L)^T \cdot (f^L)' \cdot \nabla_{a^L} E. \tag{2.22}$$

This expression is the key to backpropagation. By evaluating it from right to left the gradient in respect to each layer's weights can be calculated stepwise.

2.3.4 Stochastic Gradient Descent (SGD) and Batching

As previously mentioned, training a neural network using Gradient Descent requires calculating the error gradient using all the training data points before updating the parameters of the model. The method is also called Batch Gradient Descent, where the term *batch* refers to the entire dataset. A variation of this scheme, called Stochastic Gradient Descent (SGD) replaces the actual gradient calculated by the entire dataset by an estimate, calculated from a randomly selected subset of the data.

In the simplest form of SGD, training a neural network involves calculating the error gradient for each data point separately, and updating the network's weights accordingly. A variation of this method involves subsampling the training set into *mini-batches*, which reduces the need for constantly updating the model.

Although using this method results in a more noisy training process, this variant of gradient descent can allow the model to avoid local minima, as well as promote faster learning. Additionally, subsampling into mini-batches makes the algorithm noticeably more memory efficient, suitable for cases where the use of bigger datasets is limited by the computer's memory capabilities.

2.3.5 Batch Normalization

A discussed previously, training a neural network is an iterative process. Consequently, the distribution of each layer's inputs change, as the parameters of the previous layer change during training. This phenomenon is referred to as *Internal Covariate Shift*, and is responsible for slower and more unstable training. To address this problem, Sergey Ioffe and Christian Szegedy

[18] proposed normalizing the weighted layer inputs for each training mini-batch.

The proposed solution calls for implementing this functionality in the form of discrete layers, called *Batch Normalization (BN)* layers, which can then be incorporated to the network's architecture. With x the input values over a mini-batch $B = \{x_1, \ldots, x_m\}$, the BN layer calculates the mean μ (2.23) and variance σ^2 (2.24) of the input values across the batch. It then normalizes the activation vector (2.25) and applies a linear transformation using γ and β, two trainable parameters (2.26). These allow the network to adjust and optimize each layer's output distribution [19] [20].

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.23}$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} \left(x_i - \mu_B\right)^2 \tag{2.24}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{2.25}$$

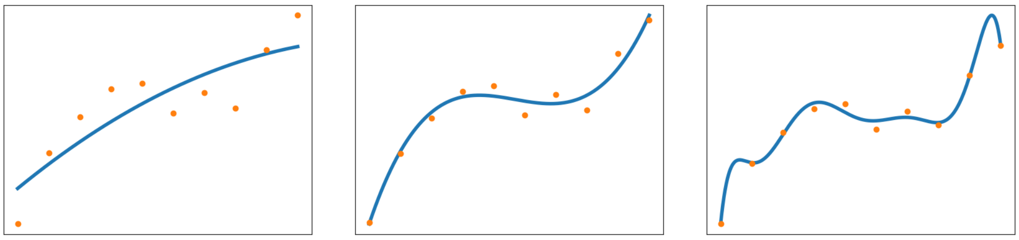
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,B}\left(x_i\right) \tag{2.26}$$

After the training process has been completed, the values μ_{pop} and σ_{pop}^2 are computed using the entirety of the dataset, and fed directly to equation 2.25 to inference the network.

2.4 Model Overfitting

Training networks to generalize well to new data is a challenging problem. The objective of a neural network is to have a final model that performs well not only on the data used for training, but also on previously unseen inputs. Simply increasing the complexity of the network, and thus it's representative capacity is not sufficient for a model to generalize well. A model should have the capacity to learn the key features of the draining data, but should avoid essentially memorizing the entire dataset. A network's tendency to *overfit* it's weights hinders it's ability to generalize on the problem and produce correct results when encountering new data [21].

The above intuition can become more evident in the simpler case of approximating a set of data points using a polynomial function. In fig. 2.5 a number of approximations using different degrees (and thus different complexity) functions on the same data are illustrated. The first attempt underfits the data using only a 2nd degree function, while the third one overfits the data using an 8th degree function. In both scenarios, the model cannot establish the dominant trend within the training dataset, and will not generalize well on new data.



(A) deg = 2 (B) deg = 4 (C) deg = 8

FIGURE 2.5: Trying to approximate a set of data points using a polynomial function. **A)** A 2nd degree function is not sufficient to effectively approximate the data (underfitting). **B)** Using a 4th degree function produces a good approximation. **C)** An 8th degree function overfits the data and is greatly effected by noise.

Mitigating overfitting is a major part of training efficient and useful neural networks. Although a plethora of methods and configurations exist to combat this phenomenon, a number of common methods can be found on most state-of-the-art designs.

2.4.1 Early stopping

Early stopping [22] is the simplest way to stop the model from learning the noise within the data. Although highly effective, it risks the possibility of stopping too early, and thus underfitting the model.

2.4.2 Training with more data

Training the network with more examples forces it to generalize on the features instead of relying on it's capacity to memorize. Unfortunately this requires the accumulation of more labeled data, which is not always be possible.

2.4.3 Data augmentation

Data augmentation [23] refers to the process of creating additional data by reasonably modifying the data in the training set. For example image data can be augmented by randomly flipping, rotating, zooming and cropping them. The introduced noise forces the model to become more stable and generalize better.

2.4.4 Regularization

Techniques that seek to reduce overfitting by keeping the network's weights small are referred to as regularization methods.

Weight decay

The most common regularization method, *weight decay* [24], penalizes the model during training based on the magnitude of the weights. To achieve this, the square of all weights is added to the loss function, multiplied by a small hyperparameter.

$$Loss = L(y', y) + wd * sum(w^2) \tag{2.27}$$

Where L the training loss function, y and y' the expected and produced outputs of the network respectively, w the weights of the network and wd the weight decay hyperparameter.

Dropout

Dropout was proposed by Nitish Srivastava et al. [25] in 2014 as a regularization method that approximates training a large number of neural networks with different architectures in parallel. To achieve this, during training, a number of nodes in the network are randomly ignored, or *dropped out* in each iteration. Although this method increases the time it takes for the model to converge, dropout can significantly reduce overfitting and improve the overall network performance. The probability of dropping a node in each training iteration is defined as a hyperparameter and the whole method can be implemented as a module to be placed after each weighted layer of the network.

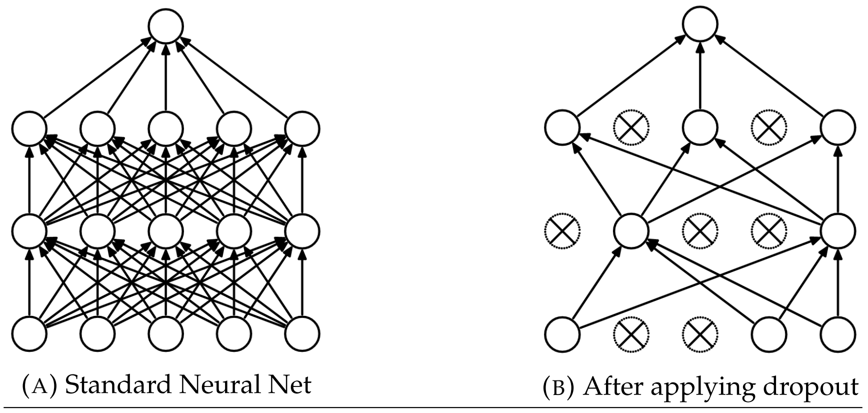


FIGURE 2.6: Dropout Neural Net Model. **A)** A standard neural net with 2 hidden layers. **B)** An example of a thinned net produced by applying dropout to the network on the left.

2.5 Convolutional Neural Networks (CNNs)

The versatility of neural networks and their capability to classify complex data structures have made them an integral part of computer vision. The technology is already being used to solve many problems such as face and object recognition, motion detection and analysis, image restoration and even disease diagnosis using data from medical imaging. These advancements have been achieved primarily over one particular algorithm, the Convolutional Neural Network [26] [27] [28].

Following the design of feed-forward networks, CNNs use a series of trainable convolution and subsampling filters to reduce the images into a form which is easier to process, without losing important features. The output of these convolution layers can then be fed to a Multi-Layered Perceptron for final processing.

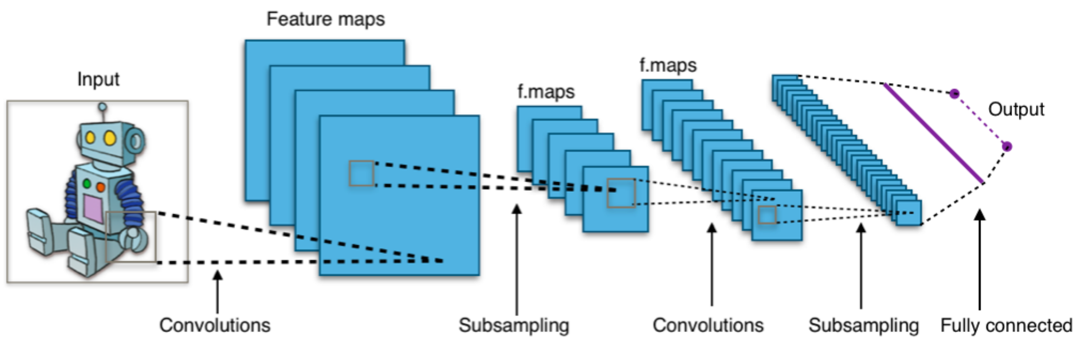


FIGURE 2.7: Typical CNN Architecture: URL.

2.5.1 Convolution Layers

Convolution layers (as the name suggests) are the building blocks of CNNs [29] [30]. They are responsible for reducing the amount of features while preserving the most useful information. To achieve this, each convolution layer uses a kernel, a small matrix of trainable parameters involved in carrying out the convolution operation. The kernel is the heart of each layer, and depends on its type and dimentionality, this will become more apparent when the most common types of convolution layers are explored.

1D Convolution Layers

The simplest form of convolution layers, useful for understanding how they work, is the one-dimentional. Assuming an input I of size 1×5 and a kernel K of 1×3 the convolution is defined as:

$$I = \begin{bmatrix} a & b & c & d & e \end{bmatrix} \tag{2.28}$$

$$K = \begin{bmatrix} x & y & z \end{bmatrix} \tag{2.29}$$

$$I * K = \begin{bmatrix} ax + by + cz & bx + cy + dz & cx + dy + ez \end{bmatrix} \tag{2.30}$$

Note that the above convolution produces a matrix of size 1×3. To keep the size of the features constant, the input is usually padded with zeros. In this example the input can be converted to I':

$$I' = \begin{bmatrix} 0 & a & b & c & d & e & 0 \end{bmatrix} \tag{2.31}$$

This way the convolution will produce an output of size 1×5:

$$I' * K = \begin{bmatrix} ya + zb & ax + by + cz & bx + cy + dz & cx + dy + ez & dx + ey \end{bmatrix} \tag{2.32}$$

It should be noted that each convolution layer defines a stride. This equals to the elements of the input the kernel skips before computing each partial convolution and influences the output's shape. For example, in the operations above, when a stride of 2 is used, the output size becomes 1×2:

$$I * K = \begin{bmatrix} ax + by + cz & cx + dy + ze \end{bmatrix} \tag{2.33}$$

2D Convolution Layers

Based on the aforementioned principles for 1-D convolution, one can easily understand how the method can be applied in the 2-D domain. In this domain, the application of the layers on images and how the kernels achieve feature extraction becomes more evident.

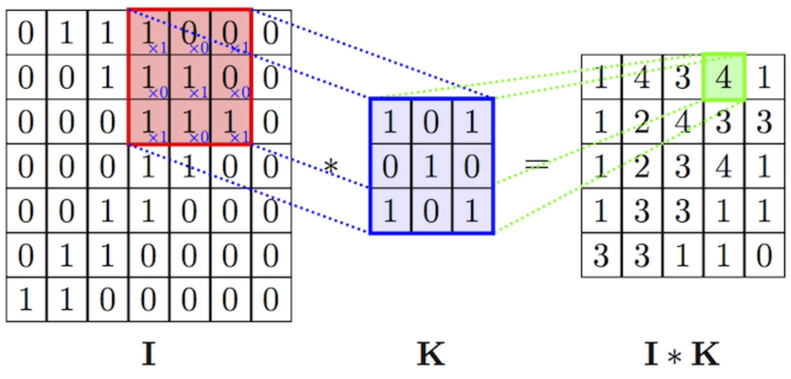


FIGURE 2.8: 2-Dimentional Convolution Layer: URL.

Assuming that a grayscale image can be represented as a two-dimentional matrix, the objective of the convolution operation is to extract the high-level features such as edges, from the input image. This can then be expanded to color images, using a 2-D matrix for each color to construct a 3-dimentional matrix of depth (or channels) $z = 3$. Although most common in computer vision, the input of the convolution network is not limited to 3 channels, and can be adapted for hyperspectral imaging, where multiple bands across the electromagnetic spectrum are used.

A single layer of convolution is usually limited to low-level features, such as edges, gradient orientation and color. By adding more layers, the network can adapt to higher level features as well, producing a model with deeper understanding for images.

To better visualize the capabilities of this technique, three different kernels designed for edge detection are applied to the same image:

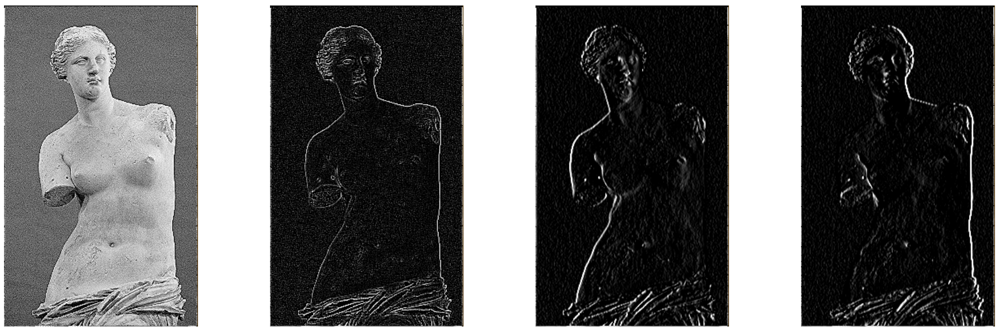

(A) Original (B) Outline (C) Left Edge (D) Right Edge

FIGURE 2.9: Edge detection using convolution kernels

Following are the kernels used for each different kind of edge detection:

$$K_{outline} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{2.34}$$

$$K_{left\_edge} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{2.35}$$

$$K_{right\_edge} = \begin{bmatrix} 1 & -0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \tag{2.36}$$

Multi-Channel Convolution

Although the input of most convolution layers is multi-channel, the equivalent 3D convolution (where the input, the kernel and the output are 3D matrices) is rarely used for computer vision applications. Multi-channel convolution is usually achieved with the use of multiple kernels, one per channel, continued by summing the outputs to a single channel. Usually this is denoted using 3D input and kernel, but a 2D output.

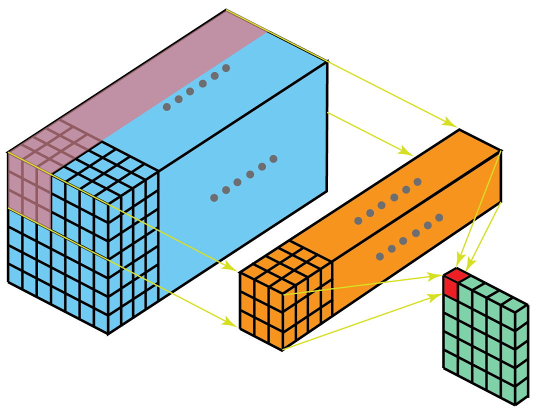


FIGURE 2.10: Multi-Channel 2D convolution with single-channel output: URL.

To increase the output channels of a multi-channel convolutional layer, a multiple of kernels can be used, one for each channel. This is usually denoted using 3D input and output, but a 4D kernel. Assuming the above, each convolution layer is defined using an input of size

$$[input\_height \times input\_width \times input\_channels] \tag{2.37}$$

, a kernel of size

$$[kernel\_height \times kernel\_width \times input\_channels \times output\_channels] \tag{2.38}$$

and an output of size

$$[output\_height \times output\_width \times output\_channels] \tag{2.39}$$

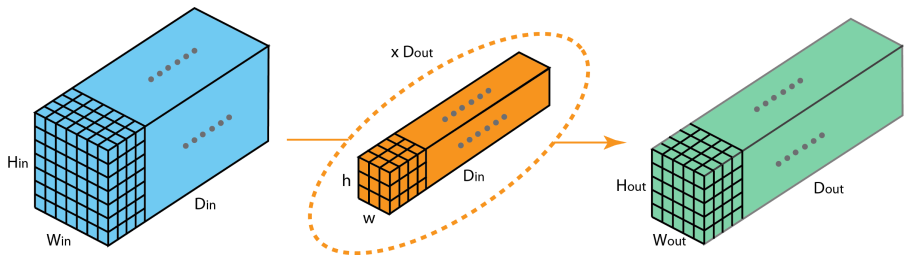


FIGURE 2.11: Multi-Channel 2D convolution with multi-channel output: URL.

2.5.2 Subsampling Layers

To further downsample (pool) the output of a convolution layer, a subsampling layer often immediately follows it [31]. This is done to decrease the size of the features along both spatial dimentions of height and width. Subsampling layers provide a way to effectively reduce the number of parameters to be learned by the network, while impeding overfitting and the reliance of precise positioning within feature maps.

The most commonly used subsampling layers are *"Average Pooling"* and *"Max Pooling"*

Average Pooling

Average pooling is an operation that as the name suggests, calculates the average for each subregion of a feature map. Subregions are usually non-overlapping and are defined by the filter's size and stride. This method of downsampling extracts features more smoothly.

Max Pooling

Similar to Average Pooling, Max Pooling applies a max filter to each subregion and is more suitable for better preserving pronounced features like edges.

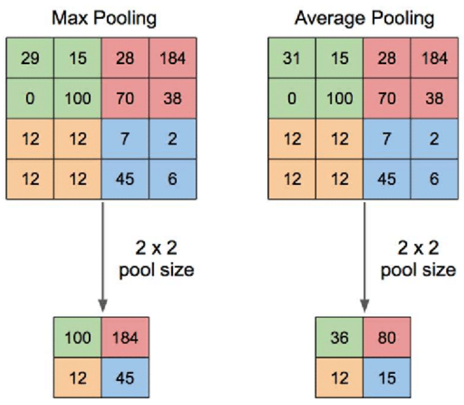


FIGURE 2.12: Illustration of Max-Pooling and Average-Pooling Layers : URL.

Chapter 3

Related Work

3.1 Training Datasets

Neural networks, like most computational systems, follow the rule of garbage in, garbage out. This effectively leads to the need of well formulated and useful datasets for the networks to be trained upon. A number of datasets have been generated over the years for research purposes, which are not only used for their qualities, but also introduce a common point of reference for different network architecture comparisons.

Usually, training datasets are separated into two distinct subsets. The first subset is used for training a neural network, while the second one is used for evaluating it's accuracy. This is done to prevent "overfitting", which is the tendency for the network to memorize the input dataset, instead of generalizing based on it's key features.

The most common datasets for image recognition are:

- MNIST

  A monochromatic database of handwritten digits, it is usu ally the first dataset someone encounters when learning about image recognition. The Dataset consists of $60,000$ training and $10,000$ evaluation data points which have been size normalized and centered in a fixed size image. Each training image is associated with a label from 10 classes.

- Fashion-MNIST

  Designed for clothing classification, Fashion-MNIST is intended as a harder to classify drop-in replacement of MNIST. It shares the same image size and structure of training and testing subsets, as well as the same number of classes.

- CIFAR-10

  One of the most used datasets in image classification research, CIFAR-10 consists of $50,000$ training and $10,000$ evaluation colour images of everyday objects. Each image is sized 32x32 pixels and belongs to one of 10 classes.

- CIFAR-100

  Similar to CIFAR-10, CIFAR-100 contains 100 classes of 500 training and 100 evaluation images each. It is a much harder dataset for a network to be trained upon not only for it's increased number of classes, but also for the reduced amount of images representing each class.

- ImageNet

  Imagenet is an accumulation of more than 14 classified million images, of which at least one million are annotated with bounding boxes. Since 2010 the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) is organized by the ImageNet project, where different network architectures compete on classifying a 'trimmed' subset of 1 thousand classes.

3.2 CNN Architectures

3.2.1 LeNet

One of the earliest convolutional neural networks, LeNet was proposed by Yann LeCun et al. [32] in 1989 as a method for hand-written zip code recognition. It is constructed using two convolution layers, each followed by an average pooling subsampling layer, with the final subsampling layer connected to a MLP of three layers. The network is trained on the MNIST dataset, having an input of 32x32 monochromatic pixels.

3.2.2 AlexNet

Alexnet, designed by Alex Krizhevsky et al. [33], was the best performing entry in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) on September 30, 2012. The increased accuracy of the network came from it's increased depth (5 convolution layers, 3 max-pooling layers and 3 linear layers), paving the way for progressively deeper networks. It's main disadvantage was it's increased complexity, making it much more computational intensive, and prohibitively time consuming when training. This was overcome by the researchers with the utilization of graphics processing units (GPUs) during training.

3.2.3 VGG

The VGG network architecture was introduced by Simonyan and Zisserman in 2014 [34]. It's most common variants are VGG-16 and VGG-19, where '16' and '19' stand for the total number of trainable layers in the network. As is evident, the upwards trend of depth is continued in this architecture, with the network favoring simpler convolution layers, with smaller kernels than AlexNet. The model was submitted to ILSVRC in 2014 and achieved 92.7% top-5 test accuracy.

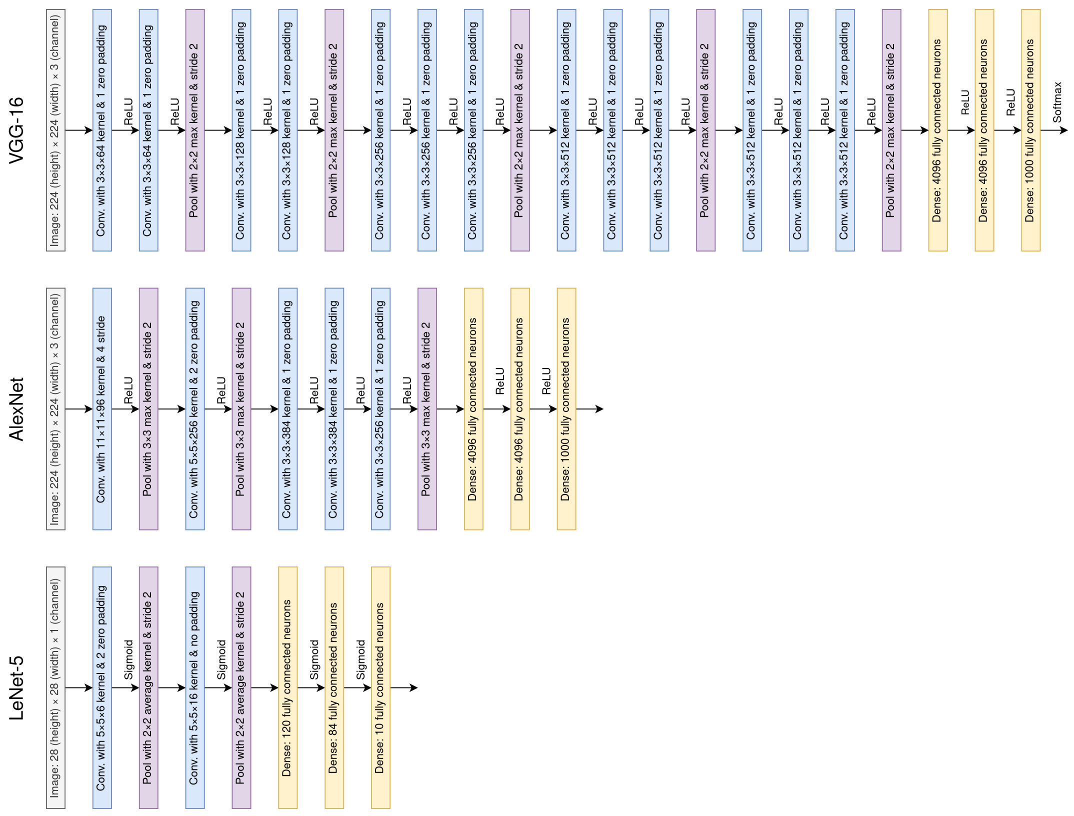

VGG-16
Image: 224 (height) × 224 (width) × 3 (channel)
Conv. with 3×3×64 kernel & 1 zero padding
ReLU
Conv. with 3×3×64 kernel & 1 zero padding
ReLU
Pool with 2×2 max kernel & stride 2
Conv. with 3×3×128 kernel & 1 zero padding
ReLU
Conv. with 3×3×128 kernel & 1 zero padding
ReLU
Pool with 2×2 max kernel & stride 2
Conv. with 3×3×256 kernel & 1 zero padding
ReLU
Conv. with 3×3×256 kernel & 1 zero padding
ReLU
Conv. with 3×3×256 kernel & 1 zero padding
ReLU
Pool with 2×2 max kernel & stride 2
Conv. with 3×3×512 kernel & 1 zero padding
ReLU
Conv. with 3×3×512 kernel & 1 zero padding
ReLU
Conv. with 3×3×512 kernel & 1 zero padding
ReLU
Pool with 2×2 max kernel & stride 2
Conv. with 3×3×512 kernel & 1 zero padding
ReLU
Conv. with 3×3×512 kernel & 1 zero padding
ReLU
Conv. with 3×3×512 kernel & 1 zero padding
ReLU
Pool with 2×2 max kernel & stride 2
Dense: 4096 fully connected neurons
ReLU
Dense: 4096 fully connected neurons
ReLU
Dense: 1000 fully connected neurons
Softmax
AlexNet
Image: 224 (height) × 224 (width) × 3 (channel)
Conv. with 11×11×96 kernel & 4 stride
ReLU
Pool with 3×3 max kernel & stride 2
Conv. with 5×5×256 kernel & 2 zero padding
ReLU
Pool with 3×3 max kernel & stride 2
Conv. with 3×3×384 kernel & 1 zero padding
ReLU
Conv. with 3×3×384 kernel & 1 zero padding
ReLU
Conv. with 3×3×256 kernel & 1 zero padding
ReLU
Pool with 3×3 max kernel & stride 2
Dense: 4096 fully connected neurons
ReLU
Dense: 4096 fully connected neurons
ReLU
Dense: 1000 fully connected neurons
LeNet-5
Image: 28 (height) × 28 (width) × 1 (channel)
Conv. with 5×5×6 kernel & 2 zero padding
Sigmoid
Pool with 2×2 average kernel & stride 2
Conv. with 5×5×16 kernel & no padding
Sigmoid
Pool with 2×2 average kernel & stride 2
Dense: 120 fully connected neurons
Sigmoid
Dense: 84 fully connected neurons
Sigmoid
Dense: 10 fully connected neurons

FIGURE 3.1: Representation of LeNet-5, AlexNet and VGG-16.

3.2.4 ResNet

While increasing network depth yields good results, training very deep neural networks becomes much harder. This is not only due to their increased complexity, but also due to the notorious vanishing gradient problem. As the error gradient is back-propagated through the network, repeated multiplication may make the gradient infinitely small, effectively stopping the weights of early layers from learning any new features.

To overcome this problem, Kaiming He et al. [35] proposed the idea of residual networks (ResNets), which introduce the use of 'identity shortcut connections'. These skip one or more layers, by summing the output of one layer, with the output of an earlier one, before feeding the data forward, this way each gradient has fewer layers to propagate through. The batch of layers between each identity connection is called a 'residual block'

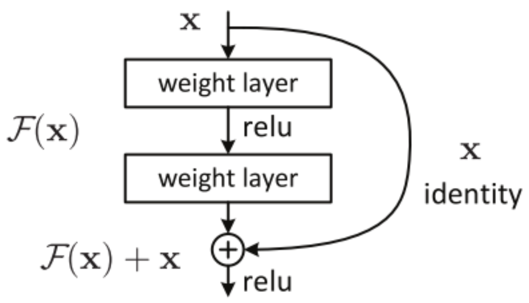

x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

FIGURE 3.2: A Residual Block.

By mitigating the vanishing gradient problem, ResNet quickly became one of the most popular architectures in various computer vision tasks.

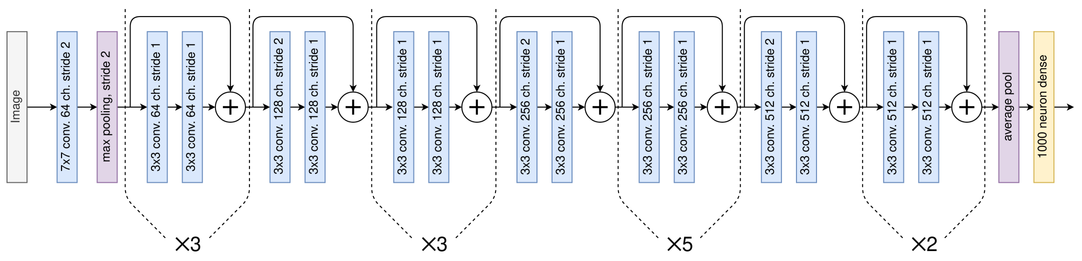


Image
7x7 conv. 64 ch. stride 2
max pooling, stride 2
3x3 conv. 64 ch. stride 1
3x3 conv. 64 ch. stride 1
3x3 conv. 128 ch. stride 2
3x3 conv. 128 ch. stride 1
3x3 conv. 128 ch. stride 1
3x3 conv. 128 ch. stride 1
3x3 conv. 256 ch. stride 2
3x3 conv. 256 ch. stride 1
3x3 conv. 256 ch. stride 1
3x3 conv. 256 ch. stride 1
3x3 conv. 512 ch. stride 2
3x3 conv. 512 ch. stride 1
3x3 conv. 512 ch. stride 1
3x3 conv. 512 ch. stride 1
average pool
1000 neuron dense
×3
×3
×5
×2

FIGURE 3.3: A Residual Network with 34 weighted layers.

Pre-Activated Residual Blocks

A year after their proposal, the researchers refined the residual block, focusing on creating a "direct" path for propagating information [36]. Their proposal demonstrated the advantages of a pre-activation variant of residual block, by successfully training a 1001-layer deep ResNet.

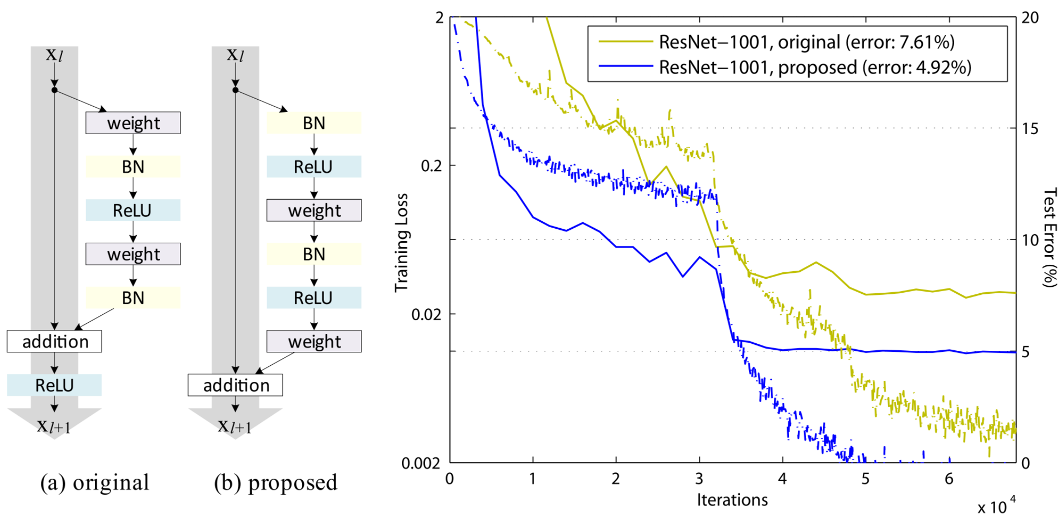


FIGURE 3.4: **Left** (a) original Residual Unit; (b) proposed Residual Unit. The grey arrows indicate the easiest paths for the information to propagate. **Right** training curves on CIFAR-10 of 1001-layer ResNet. Solid lines denote test error, and dashed lines denote training loss. The proposed unit makes ResNet-1001 easier to train.

3.2.5 DenseNet

Densely Connected CNN, or DenseNet, was proposed by Huang et al. [2] in 2016. It further exploits the effects of shortcut connections by connecting all layers directly with each other. This way, each layer obtains additional inputs from all preceding layers and passes on its own feature-maps to all subsequent layers. This architecture is then divided into discrete dense blocks comprising multiple modules each.

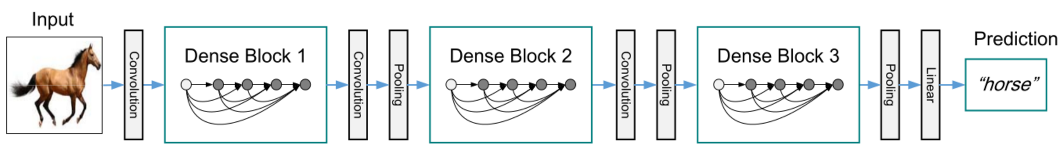


FIGURE 3.5: A deep DenseNet with three dense blocks

Motivated by He et al. [36] and their work on pre-activated residual blocks, the researchers define as a *Composition Module* the sequence of Batch Normalization (BN) followed by a Rectified-Linear Unit and a 3×3 Convolution layer. To reduce the number of inputs on composition modules, and thus improve computational efficiency, *Bottleneck Modules* are introduced to the design. These modules are composed of a Batch-Normalization (BN) followed by a Rectified-Linear Unit and a 1×1 Convolution layer. DenseNets are formulated by connecting multiple Dense Blocks in series using *Transition Blocks* in between, to decrease feature-map sizes. Each transition block is constructed using a bottleneck module, followed by an average pooling layer.

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112×112	7 x 7 conv, stride 2			
Pooling	56×56	3 x 3 max pool, stride 2			
DenseBlock	56×56	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 6$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 6$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 6$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 6$
Transition Layer	56×56	1 x 1 conv			
	28×28	2 x 2 average pool, stride 2			
DenseBlock	28×28	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 12$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 12$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 12$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 12$
Transition Layer	28×28	1 x 1 conv			
	14×14	2 x 2 average pool, stride 2			
DenseBlock	14×14	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 24$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 32$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 48$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 64$
Transition Layer	14×14	1 x 1 conv			
	7×7	2 x 2 average pool, stride 2			
DenseBlock	7×7	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 16$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 32$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 32$	$\begin{bmatrix}1\times1\text{ conv}\\3\times3\text{ conv}\end{bmatrix}\times 48$
Classification Layer	1×1	7 x 7 global average pool			
		1000D fully-connected, softmax			

TABLE 3.1: DenseNet architectures for ImageNet. The growth rate for all the networks is k = 32. Note that each "conv" layer shown in the table corresponds the sequence BN-ReLU-Conv.

Since each module receives feature maps from all preceding modules, the architecture promotes feature reuse, and the network can be thinner and compact, presenting higher computational and memory efficiency. To prevent the network from growing too wide, the authors used a hyper-parameter called growth-rate (k) which defines the additional number of channels after each convolution layer.

3.3 Adaptive Learning Rate

As previously mentioned, optimization algorithms for training neural networks are almost always dependent on a learning rate hyperparameter, which impacts the training process significantly. It has been empirically observed [37] that learning rate decay can help both optimization as well as generalization of the model. The method begins training the network using a large learning rate (usually larger than what would be used when training the network with a static learning rate), which is then reduced multiple times during the training procedure.

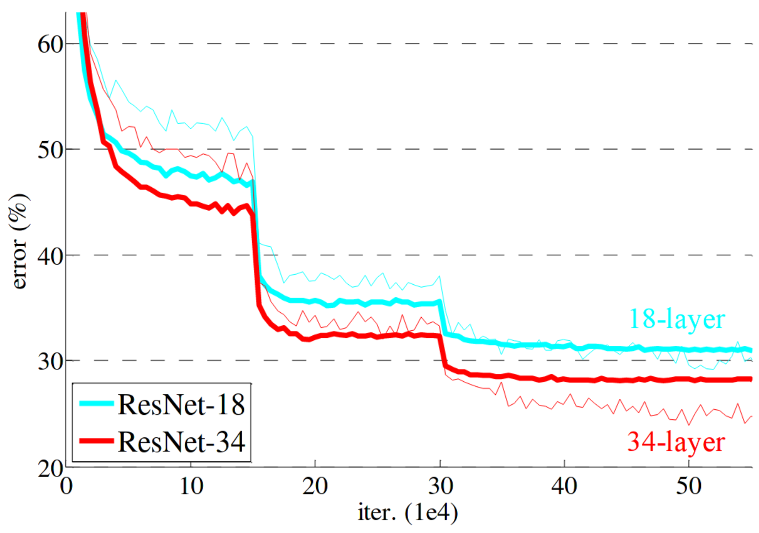


FIGURE 3.6: Figure taken by He et al [35]. Initial learning rate is $lr_0 = 0.1$, which is then divided by 10 at epochs 30 and 60. Training error is shown by thin curves, while test error by bold curves.

3.3.1 Time-Based Decay

Time based decay is defined as:

$$lr = \frac{lr_0}{K * E} \tag{3.1}$$

Where lr is the learning rate used by the optimization algorithm, lr_0 is the initial learning rate, E is the current epoch and K is used as a hyperparameter for tuning the decay rate.

3.3.2 Step Decay

Step decay schedule drops the learning rate by a factor every few epochs:

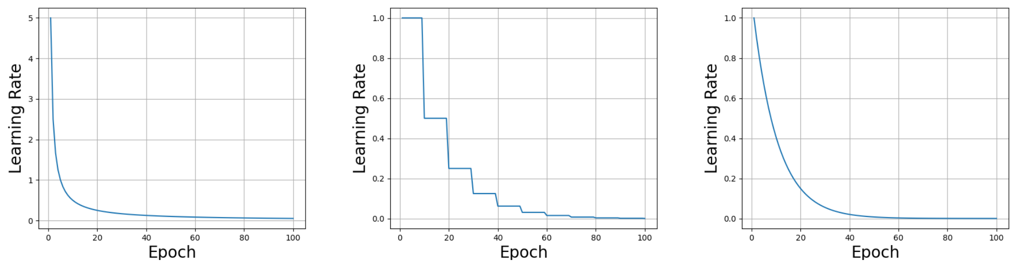
$$lr = lr_0 * K^{\lfloor \frac{E}{S} \rfloor} \tag{3.2}$$

Where S is the step size for each decay K is used as a hyperparameter for determining the decay size in each step. ()

3.3.3 Exponential Decay

Exponential decay is defined as:

$$lr = lr_0 * e^{-K*E} \tag{3.3}$$



(A) Time Decay (B) Step Decay (C) Expo Decay

FIGURE 3.7: Comparison of different learning rate decay methods. **A)** Time Decay using $lr_0 = 1$ and $K = 0.2$. **B)** Step Decay using $lr_0 = 1$, $K = 0.5$ and $S = 10$. **C)** Exponential Decay using $lr_0 = 1$ and $K = 0.1$.

3.3.4 Learning Rate Warmup

A specific type of adaptive learning rate, warmup is a way to reduce early overfitting by slowly ramping-up the learning rate in the course of one or two epochs. Without it, the model may tend to skew badly towards the first number of mini-batches it encounters, needing a few extra epochs to get the convergence desired, as the model un-trains those early superstitions.

3.4 Knowledge Distillation (KD)

A highly effective network compression technique, Knowledge distillation enables the simplification of cumbersome models without suffering large accuracy losses. Proposed by Hinton et al. [38], KD is a promising methodology to distill teacher models and partially transfer their knowledge to simpler student models. This is achieved by matching the softened probability distribution of each network's output classes.

The softmax q_i of the network's output logits z_i is calculated using the following equation:

$$q_i = \frac{exp(z_i/T)}{\sum exp(z_j/T)} \tag{3.4}$$

With higher temperature T the probability targets get softer. The knowledge learned from training a teacher model with normal softmax (i.e., $T = 1$) can then be distilled and partially transfered to a student network by minimizing the KD loss:

$$L_{KD}(W_{student}) = aT^2 * CrossEntropy(Q_S^\tau, Q_T^\tau) + (1 - a) * CrossEntropy(Q_S, y_{true}) \tag{3.5}$$

Q_S^τ and Q_T^τ are the softened targets of the student and teacher networks respectively using the same temperature T and a tunes the weighted average between the softened targets and the ground truth labels of the training set.

3.5 Quantization

Neural networks are traditionally developed using floating point arithmetic, since it most closely represents the original mathematical models, but are very resource intensive algorithms. Using lower precision representation of weights and activations and in consequence lower precision math can greatly increase the inference speed of a neural network while reducing it's memory footprint.

Lowering the precision of a neural network can be achieved in two ways, using a lower bit floating representation or migrating to integer arithmetic, the latter of which is called *quantization* and usually achieves the highest computational improvements, especially when coupled with appropriate hardware.

Data Type	**Bits**	**Minimum (α_q)**	**Maximum (β_q)**
int16	16	-32768	32767
uint16	16	0	65535
int8	8	-128	127
uint8	8	0	255
int4	4	-8	7
uint4	4	0	15

TABLE 3.2: Most common integer data types used in quantized networks.

Assuming a floating point value $x \in [\alpha, \beta] \in \mathbb{R}$, the quantization process maps the value to a b-bit integer $x_q \in [\alpha_q, \beta_q] \in \mathbb{Z}$.

$$x_q = clip(round(\frac{1}{s}x + z), \alpha_q, \beta_q) \tag{3.6}$$

$$x = s(x_q - z) \tag{3.7}$$

Where s and z are the scale and zero point respectively.

$$s = \frac{\beta - \alpha}{\beta_q - \alpha_q} \tag{3.8}$$

$$z = round(\frac{\beta\alpha_q - \alpha\beta_q}{\beta - \alpha}) \tag{3.9}$$

On the above equations α_q and β_q are dependent on the quantization data type (table 3.2) and α and β are usually acquired from the training dataset of the network. Depending on the implementation, α and β can represent the range of possible values for the entirety of the weighted layers, or a different range can be acquired per-layer, or even per-channel.

In practice, α and β are an estimation, and the network will have a chance to encounter values outside this range when in use, resulting in a quantized value outside the range of $[\alpha_q, \beta_q]$. To mitigate that in 3.6, a clipping function is necessary, where $clip(x, a, b)$ is defined as:

$$clip(x, a, b) = \begin{cases} a & x < a \\ x & a \leq x \leq b \\ b & x > b \end{cases} \tag{3.10}$$

A special case of the above called *symmetric quantization* is achieved when using signed integers for weight and activation representation and zero point $z = 0$. This means that the quantization process is mapping floating point numbers in the range $[-r, r]$ where:

$$r = max(|\alpha|, |\beta|) \tag{3.11}$$

Although very promising, quantization of a neural network presents some significant drawbacks. By lowering the precision of a model's weights and activations, it's representative power is diminished, which in turn can result to substantial losses in accuracy. To combat this phenomenon, a plethora of quantization techniques have been proposed by multiple researchers, aiming at retrieving the losses in accuracy while preserving any reduction in computational cost and memory footprint.

3.5.1 Post-Training Quantization

The simplest and most common way of quantizing a neural network is after the training process has been completed. Unfortunately this method can result in significant accuracy losses, since the model is not aware of the inherent quantization error while training.

3.5.2 Quantization-Aware Training (QAT)

Although a more complex, and as a result more computational intensive method, quantization-aware training aims to minimize the quantization error, by applying every quantization transformation in the training process. Here, the range $[\alpha_q, \beta_q]$ can be treated as a trainable parameter to be optimized. To speed up the training process, it is common for QAT to initialize the network by partially training it using floating point arithmetics before quantization.

Quantizing a neural network can induce regularization on its features, which as previously mentioned, NN models can benefit from. This phenomenon can result on some networks presenting small accuracy improvements when correctly trained using QAT.

Model	**Floating-Point Baseline**	**Quantization-Aware Training**	**Post-Training Quantization**
MobileNet V1 1.0 224	71.03%	71.06%	69.57%
MobileNet V2 1.0 224	70.77%	70.01%	70.2%
ResNet V1 50	76.3%	76.1%	75.95%

TABLE 3.3: Comparison of top-1 % accuracy between baseline, quantization-aware trained and post-training quantized models. Note the small improvement over the baseline model of Mobilenet V1 when using QAT (source).

3.6 Quantization-Aware Knowledge Distillation

As the name suggests, Quantization-Aware Knowledge Distillation (QKD) aims at combining Knowledge Distillation with Quantization-Aware Training, carefully coordinating the two techniques to achieve greater accuracy compared to simply applying them consecutively on the quantized model.

Originally proposed by Kim et al. [1], the process of QKD is divided in three phases:

3.6.1 Phase 1: Self-Studying (SS)

After training using it's full precision representation, the student network is quantized and fine tuned using Quantization Aware Training (QAT). All weights and activations are "fake quantized" during both the forward and backward passes of training: that is, float values are rounded to mimic integer values, but all computations are still done with floating point numbers. This serves as a good starting point for the next phases.

The authors of the original proposal, opted for trainable uniform quantization scheme, because of it's hardware-friendly characteristics. To achieve this, two trainable parameters for the interval values of each layer's weights and input activations are used (I_W and I_X respectively).

3.6.2 Phase 2: Co-Studying (CS)

In this phase the teacher network (full-precision) and student network (low-precision) are jointly trained in an online manner by minimizing $L_{KD}(W_{teacher})$ and $L_{KD}(W_{student})$ respectively:

$$L_{KD}(W_{teacher}) = aT^2 * CrossEntropy(Q_T^\tau, Q_S^\tau) + (1-a) * CrossEntropy(Q_T, y_{true}) \tag{3.12}$$

$$L_{KD}(W_{student}) = aT^2 * CrossEntropy(Q_S^\tau, Q_T^\tau) + (1-a) * CrossEntropy(Q_S, y_{true}) \tag{3.13}$$

3.6.3 Phase 3: Tutor-Studying (TS)

Finally the teacher network's state is freezed and only the student network is trained minimizing $L_{KD}(W_{student})$ in an offline manner.

Algorithm 1: Quantization-aware Knowledge Distillation

```
Input: Training Data;
       Pre-trained $FP$ weights for teacher model $T_F$;
       Pre-trained $FP$ weights for student model $S_F$;
       Low-bit student model weights $S_L$;
       Weight interval values $I_W$;
       Activation interval values $I_X$;
       Number of epochs for each phase $P_1$; $P_2$; $P_3$;
Output: Trained low-bit student weight and interval values $S'_L$, $I'_W$ and
        $I'_X$
1  Phase 1: Self-Studying;
2  Init $S_L$ with $S_F$ ; Init $I_W$ , $I_X$ using min-max values of weights and one
     batch of activations;
3  for $Epoch = 1, \dots, P1$ do
4      Update $S_L$ , $I_W$ and $I_X$ by minimizing $L^s_{ce}$;
5  end
6  Phase 2: Co-Studying;
7  Init $S'_L$, $I'_W$ and $I'_X$ with $S_L$, $I_W$ and $I_X$;
8  for $Epoch = 1, \dots, P2$ do
9      Update $T_F$ by minimizing $L_{KD}(W_{teacher})$;
10     Update $S'_L$, $I'_W$ and $I'_X$ by minimizing $L_{KD}(W_{student})$;
11 end
12 Phase 3: Tutoring;
13 for $Epoch = 1, \dots, P3$ do
14     Update $S'_L$, $I'_W$ and $I'_X$ by minimizing $L_{KD}(W_{student})$;
15 end
```

Chapter 4

Tools and Platforms

4.1 Deep Learning Software Tools

The popularization of DNNs has resulted in the creation of multiple software frameworks for easier and more efficient development. These frameworks provide a good abstraction, relieving the model designer from the tedious work of implementing all the aspects of their design, allowing for more experimentation. Some of the most popular ones are described below.

4.1.1 TensorFlow & Keras

As of this writing, TensorFlow and Keras are considered to be the most popular machine learning frameworks. Developed by the Google Brain Team and released in 2015, TensorFlow is an open-source framework not limited to, but predominantly used for machine learning applications. Offering a complete end-to-end solution, TensorFlow allows for easy deployment across a variety of platforms (CPUs, GPUs, TPUs) and devices such as desktops, clusters and edge devices. As a complementary tool, Keras API is designed on top of TensorFlow to enable fast experimentation with it's user friendliness and easy expandability.

4.1.2 PyTorch

Based on the Torch library [39], PyTorch is an open-source neural network framework developed by Facebook's AI Research Lab (FAIR) and released in 2016. Its highly polished and easy to use python interface, along with its flexibility, has increased the framework's popularity particularly in academia. This is due to the more 'pythonic' nature of the framework, offering a flatter learning curve compared to TensorFlow, especially to researchers who are already familiar with the language. Although PyTorch does not match the deployment capabilities of TensorFlow, mainly due to the latter's popularity in the tech industry, it too provides a plethora of compatible platforms and devices.

4.1.3 TensorBoard

Training a neural network requires effective visualization of a number of metrics such as loss, accuracy, learning rates etc. TensorBoard is a set of tools for tracking and visualizing metrics, visualizing model graphs and even displaying images, text and data, making it an indispensable tool for developers. Although it is a product of the TensorFlow ecosystem, TensorBoard can be utilized as a stand-alone tool by most frameworks in Python.

4.2 Neural Networks on Hardware

As previously mentioned, neural networks can achieve world-leading performance on various regression and classification tasks. As DNNs increase in size and capabilities, so does their appetite for computational power. DNNs are comprised by a variety of operations, the majority of which are series of Multiply-Accumulate (MAC) across large vectors of data. Choosing a platform to calculate these operations is very application dependent; CPUs, GPUs, TPUs and even FPGAs are viable options, but not all platforms are suitable in all situations, and no solution can be regarded as all-fitting.

Training a neural network is usually considered the most computationally intensive phase of its life-cycle. Here GPUs and purpose-built TPUs are considered the best solution, since both are designed for high-volume, parallel vector computations. Fortunattely most networks are trained once before their deployment, making the inference phase a greater point of interest for optimizations.

Although inferencing neural networks is much less computationally intensive, the requirements of each application can significantly influence the model's deployment strategy. An example of this can be found in modern A.I. assisted smartphones, where the user can "wake-up" the assistant using a set of simple voice commands, and then dictate an action or ask a question. Here, the relatively easy task of voice recognition is performed locally, using the device's onboard CPU. After that, the generated text is transmitted to a server for Natural Language Processing (NLP) [40], which is a much more complicated task, requiring bigger DNNs, which in turn require hardware like GPUs or TPUs to execute in reasonable time.

4.2.1 CPU

Although they offer excellent speed for sequential tasks, and can even support technologies such as Single Instruction Multiple Data (SIMD) [41], CPUs are not optimized for large vector operations required by DNNs. Nevertheless, due to their flexibility and availability, they offer easy deployment of DNNs on most systems.

4.2.2 GPU

As the name suggests, GPUs are designed to efficiently render graphics, which are essentially series of matrix calculations. To achieve that, GPUs are optimized for data parallel throughput computations. A GPU consists of multiple Processor Clusters (PC) that contain multiple Streaming Multiprocessors (SM), which are designed to support instruction-level parallelism using multiple small computational cores. Although these cores are limited in their capabilities compared to ones found in a CPU, they can be enumerated in the thousands in a single die, making them ideal for tasks with high parallelism characteristics.

Another design characteristic of GPUs is their increased memory bandwidth compared to CPUs which chat reach $10\times$ on modern hardware. Usually complemented by onboard V-RAM, High Bandwidth Memory (HBM) becomes indispensable when dealing with large datasets.

Most of modern deep learning frameworks support GPU acceleration, usually based on the CUDA architecture. Compute Unified Device Architecture is a prallel computing platform and API provided by NVIDIA to enable the use of GPUs for general-purpose applications. NVIDIA additionally provides the CUDA Deep Neural Network (cuDNN), a set of highly tuned implementations of common routines used in deep learning such as convolution, pooling and activation layers, to be used by other deep learning frameworks.

4.2.3 TPU

Developed by Google in conjunction with their machine learning framework TensorFlow, Tensor Processing Units (TPUs) are application specific integrated circuits (ASICs) designed to accelerate certain TensorFlow operations for deep learning. TPUs where internally deployed by the company in 2015 and later made available for third-party use in 2018, either through their cloud services or as stand-alone chips.

TPUs where originally intended for use in data centers for serving large amounts of inferencing requests or training very large models. Recently, the company has developed a line of edge-devices offered through their brand "Coral". These devices provide forward-pass acceleration for quantized networks aimed at IoT and low-power applications.

Compared to GPUs, TPUs use less flexible cores, which excel in large tensor operations making them better suitable for deep learning operations. More precisely, a TPU core consists of a Matrix Multiply Unit (MXU) for matrix multiplications of mixed-precision 16-32 bit floating point arithmetic and a Vector Processing Unit for tasks like activations etc. using float32 and int32 computations.

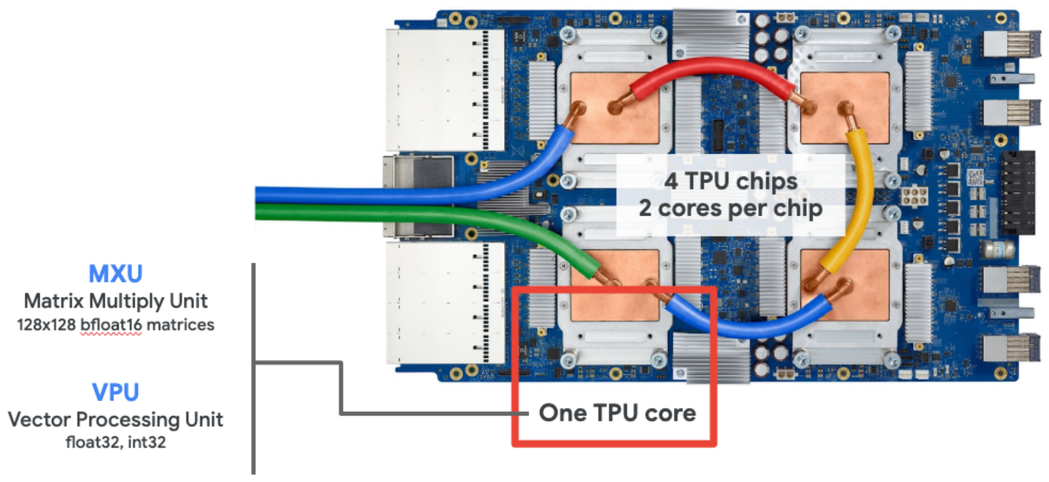


FIGURE 4.1: TPU Cores and Chips URL

Multiplication of two matrices can be defined as a series of dot-products between their lines and columns.

$$Z = X * Y \tag{4.1}$$

$$Z[A,B] = X[A,0] * Y[0,B] + X[A,1] * Y[1,B] + X[A,2] * Y[2,B] + \cdots + X[A,n] * Y[n,B] \tag{4.2}$$

Calculating the dot-products using a GPU is accomplished by assigning each one to a core. Assuming matrixes of size 128×128, the multiplication would require $128 * 128 = 16384$ (16K) dot-products, exceeding the available amount of cores in even the largest GPUs. On the contrary, MXUs consist of 16K simple MAC cores, the bare minimum required for dot-product operations, allowing them to calculate multiplications of 128×128 matrices in one go. Additionally, using a systolic-array architecture, MXUs can propagate the intermediate sums between adjacent compute units, without the need for storing/retrieving them to and from the memory, presenting significant speed and power advantages over GPUs.

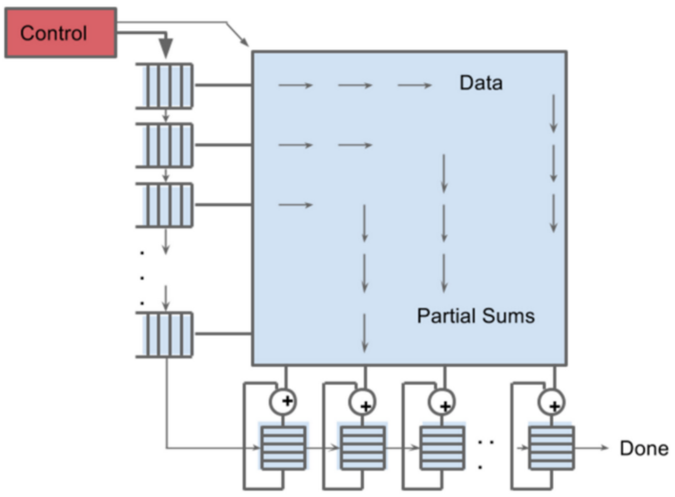


FIGURE 4.2: Architecture of a Matrix Multiplier Unit (MXU)
URL

Although TPUs present significant advantages over conventional methods for deep learning applications, they too suffer from most drawbacks of ASICs. Optimizing a system for a limited set of operations can quickly make it obsolete. As the field of deep learning progresses, new methods and requirements for state-of-the art networks are introduced. Unfortunately, ASICs are expensive to design and produce.

4.2.4 FPGA

Field Programmable Gate Arrays (FPGAs) are essentially devices comprising a large number of logic gates whose interconnection can be defined dynamically and hense be "programmed" to emulate a given logic architecture. These devices close the gap between flexible but not optimized CPUs and hard to develop and produce ASICs such as TPUs.

FPGAs often incorporate RAM blocks as well as digital signal processors (DSPs) on the same chip to accelerate most complex digital computations. Additionally, many FPGAs come with embedded processors, called "hard cores" to form a system-on-chip (SoC). These cores are usually in charge of communications, scheduling and data pre/post-processing, leaving the entirety of the FPGAs resources for accelerating the task in hand.

Unfortunately, FPGAs are usually limited by the size of their internal memory, leaving the need for slower DRAM when dealing with large datasets. This is a significant disadvantage compared to GPUs and TPUs which are provided with HBM. Additionally, their reprogrammable nature requires supplementary logic and extended datapaths, hindering their ability for higher clock speeds. Regardless their drawbacks, FPGAs can still present significant advantages when dealing with irregular architectures, enabling developers to structrure the hardware design to best suit the model. They also provide a fast and reliable way of emulating and validating hardware designs before moving to more expensive platforms such as ASICs.

Manually defining the connections between tens or hundreds of thousands of logic elements on an FPGA is not realistic. Most vendors provide solutions for developing harware application using higher level design tools. Designers can use structural representations of their design, describe hardware as functinal I/O behaviour, or a combination of both. Using Hardware Description Languages (HDL) a netlist can then be generated, which is a full representation of logical ports and their connections.

Nevertheless, designing complex accelerators for sophisticated algorithms using hardware description languages can be too time-consuming. To aid in development, many vendors offer tools for translating high-level functions into HDL code, which allow for faster design and better analysis of an architecture before translating it to a netlist. Additionally, highly optimized and verified hardware blocks of popular functions are made available as libraries.

4.3 Vitis AI

Aiming at faster and easier deployment of neural networks on their devices, Xilinx provides a design suite called Vitis-AI. The suit comprises a plethora of optimized IP cores, tools, libraries, models, and example designs, and provides a workflow to design and deploy networks on Xilinx Deep Learning Unit (DPU).

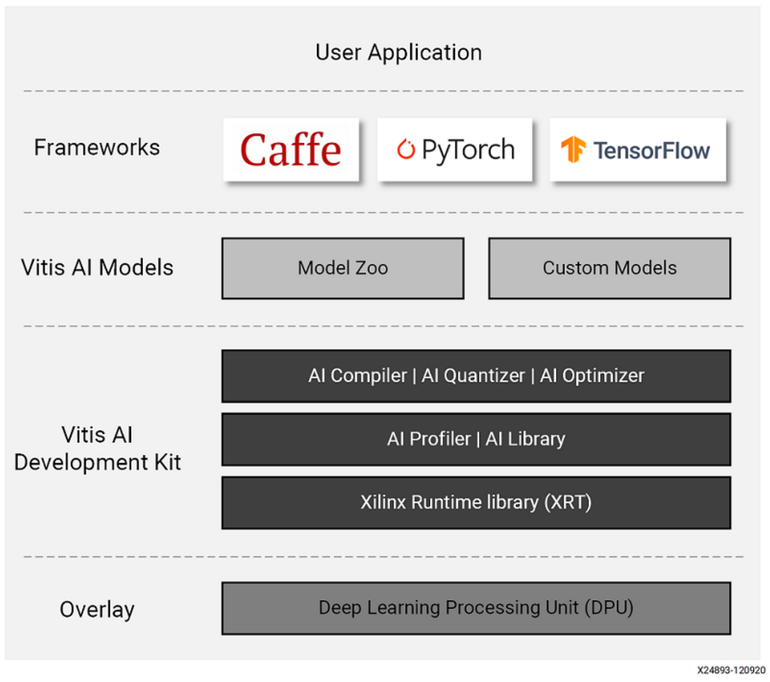


FIGURE 4.3: Vitis AI Stack.

4.3.1 Xilinx DPU

The DPU is designed as a programmable engine optimized for deep neural networks. It consists of several IP cores implemented on the hardware without requiring any place and route. Similar to a TPU, it accelerates common workloads for deep learning inference algorithms for computer vision applications. The DPU comes with its proprietary specialised Vitis-AI instruction set for efficient implementation of deep learning networks [42].

4.3.2 Vitis-AI Quantizer

The architecture of Xilinx DPU is based on quantized weights and activations for higher performance and energy efficiency. Vitis-AI provides tools for quantizing models of different deep learning frameworks (Caffe, Tensorflow and Pytorch). The Vitis AI quantizer takes a floating-point model as input, performs pre-processing like fusing batch-norm and weighted layers, and then quantizes the weights/biases and activations to the given bit width.

Calibrating the quantized layers is achieved using a small number of unlabeled data. Alongside simple model quantization, the toolkit provides various optimization functionalities such as Quantization-Aware Training for minimizing the quantization error using the entirety of the training dataset.

Finally, the quantizer generates a proprietary quantized network model which can then be compiled by the Vitis AI compiler and deployed to the DPU [43].

4.3.3 Vitis-AI Compiler

Vitis-AI provides a domain specific compiler optimized for neural network computations (VAI-C) for efficiently mapping the quantized network model on the DPU instruction sequence. Using the quantized model from the previous step, VAI-C generates internal computation graph (XIR), and corresponding control flow and data flow information. It then performs multiple kinds of compilation optimizations and transforming techniques, including computation nodes fusion, efficient instruction scheduling, full reuse of DPU on-chip data, etc. [44].

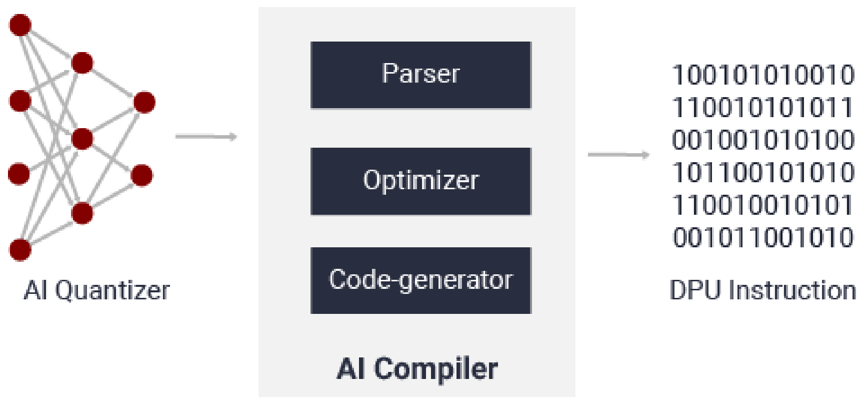


FIGURE 4.4: Vitis AI Compiler Framework.

4.4 Specifications of Utilized Platforms

This work focuses on the impact of quantization algorithms on the accuracy of models, as well as the performance of different computational platforms for inferencing them. This requires a set of different platforms to be evaluated, to better understand the impact of lower precision arithmetics, as well as utility of custom architecture. Following are the specifications of the platforms used for final experimentation.

4.4.1 Intel i5-8600K

The Intel i5-8600K CPU [45], released in late 2017, is a desktop processor targeted for medium to high workloads. Along with multi-threading capabilities, the device offers a set SIMD instructions (SSE4 [46]) to improve vector operation performance. In this work, the platform is evaluated with both floating-point and 8-bit integer arithmetic CNNs to compare the impact of its SIMD capabilities. Its specifications are presented in table 4.1.

Cores / Threads	6/6
Max Turbo Frequency	4.3 GHz
TDP	96W
Max Memory Bandwidth	41.6 GB/s
Lithography	14 nm
Instruction Set Extentions	SSE4.1, SSE4.2, AVX2

TABLE 4.1: Intel i5-8600K processor specifications.

4.4.2 Nvidia RTX-2060 Super 8GB

Released in 2019, the device is equipped with multiple CUDA cores and high-bandwidth memory, making it suitable for deep learning applications. The device's architecture is designed for floating-point arithmetic operations, and thus is only evaluated using full-precision models. It's specifications are represented in table 4.2.

CUDA Cores	2176
Tensor Cores	32
GPU Memory	8GB GDDR6
Boost Clock	1650 MHz
Memory Interface	256-bit
Memory Bandwidth	448GB/s
Power Consumption	175W

TABLE 4.2: NVIDIA RTX 2060 Super specifications [47].

4.4.3 Xilinx Zynq UltraScale+ MPSoC

Released in 2015 the Zynq UltraScale+ `XCZU9EG` MPSoC (Multi Processor System On Chip) combines a processing system (PS) and user programmable logic (PL) into the same device. The MPSoC provides 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. The processing system (PS) in a Zynq UltraScale+ MPSoC features the Arm Cortex-A53 MPCor 64-bit quad-core processor and Cortex-R5 dual-core real-time processor. Specifically, the MPSoC offers [48]:

Power Consumption	45W
HD banks	5 banks, total of 120pins
HP banks	4 banks, total of 208pins
MIO banks	3 banks, total of 78pins
PS-side GTR 6 Gb/s transceivers	4 PS-GTRs
PL-side GTH 16.3 Gb/s transceivers	24 PL-GTHs
Logic cells	599,550
CLB flip-flops	548,160
Max. distributed RAM	8.8 Mb
Total Block-RAM	32.1 Mb
DSP slices	2,520

TABLE 4.3: Zynq UltraScale+ MPSoC ZCU9EG Features and Resources.

ZCU102 Evaluation Board

The ZCU102, seen in figure 4.5, is a general purpose evaluation board for rapid prototyping based on the Zynq UltraScale+ XCZU9EG MPSoC. 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.

FIGURE 4.5: The ZCU102 Evaluation Kit. [49]

This thesis is focused on edge applications, where computational and power resources are usually limited. Since the platform features a single MPSoC, and due to the greater support provided by Xilinx for it's own evaluation kit, this work utilizes a ZCU102 board for further experimentation. However, transferring the resulting designs to devices of the same MPSoC family requires minimum effort. An example device designed for High Performance Computing (HPC), the Quad-FPGA Daughter Board is presented in 4.4.3.

Quad FPGA Daughter Board

Developed by the Foundation of Research and Technology Hellas (FORTH) [50], the QFDB, seen in figure 4.6, is the HPC Testbed Prototype built for ExaNeSt [51] project, which is now a part of the EuroExa research project. It is equipped with 4 Zynq UltraScale+ XCZU9EG MPSoCs, a part of the same family as the one used on the ZCU102 board, making the conveyance of a hardware design from one platform to the other a relatively easy task. Every FPGA node is connected to a 32MB QSPI memory and to a 16GB DDR4 SODIMM, able to transfer data at a rate of 160 Gbps, with the total available memory of 64 Gb on the board. The board is also equipped with a 256 GB SSD/NMVe (2 TB devices are available today), connected to one of the MPSoCs. The FPGA nodes are connected in an all-to-all intra-node topology with 2 High Speed Serial Links (HSSL) using GTH transceivers as well as 24

Low-voltage differential signaling (LVDS) pairs. Finally one of the MPSoCs is connected to the outside world with 10 HSSLs at 10.3125 Gbps, also using GTH transceivers.

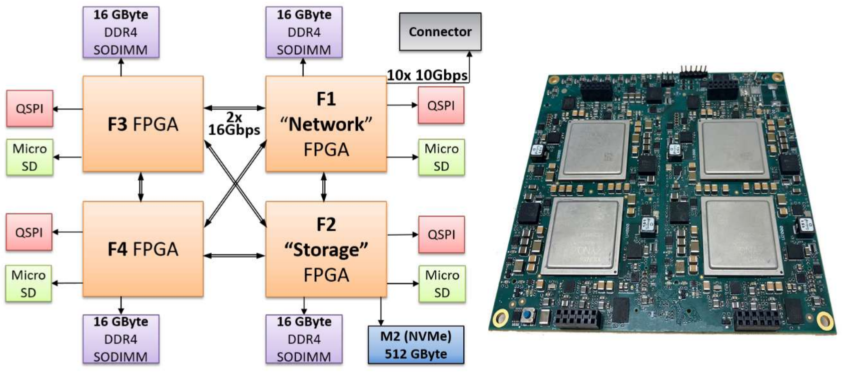


FIGURE 4.6: QFDB Architecture (**Left**) and actual board (**Right**). [52]

4.5 Thesis Approach

Network quantization for faster and less power demanding deep learning applications is frequently a core component in custom hardware AI accelerators. As mentioned above, Xilinx DPUs take advantage of quantized networks to better utilize the available hardware. Unfortunately, lowering the arithmetic precision of a network can result in significant accuracy loss, motivating the developers of Vitis-AI to include tools for automated quantization optimizations like QAT. This thesis aims to explore the performance of different CNN architectures on Xilinx DPUs, as well as implement QKD algorithms to further optimize the quantized models.

Chapter 5

Robustness Analysis of CNNs

This chapter presents the work done for the evaluation and characterization of various model compression and quantization algorithms. A plethora of training methodologies as presented in chapters 2 and 3 are employed for building efficient and accurate image classification models, used as baseline for further experimentation.

5.1 Software

All experiments are performed using the Pytorch framework (see 4.1.2). Pytorch is chosen for it's flexibility, as well as it's integration with the Xilinx Vitis-AI toolbox. For convenience, a set of utilities where developed for parsing configuration files. Defining the various experiments using configuration files promotes code reuse and simplifies experimentation.

5.1.1 Datasets and Dataloaders

To promote code readability and modularity, Pytorch provides two data primitives: `torch.utils.data.Dataset` and `torch.utils.data.DataLoader`. The former stores the samples and their corresponding labels, and the latter wraps an iterable around the Dataset to enable easy access to the samples [53].

Dataloaders provide a convenient way of effectively managing large datasets. Their functionality includes but is not limited to handling mini-batching, multi-process data loading and custom data loading order (e.g. shuffling) [54].

Using the Dataset primitive, Pytorch libraries provide numerous pre-loaded datasets for various applications (images, audio, text etc.) for faster prototyping and benchmarking. In this work, the datasets MNIST, CIFAR-10 and CIFAR-100 are used, provided by the framework's `Torchvision` package [55].

5.1.2 Optimizers

A variety of optimization algorithms are provided in the `torch.optim` package [56]. Based on the `Optimizer` class the algorithms provided include but are not limited to Stochastic Gradient Descent (SGD), Averaged Stochastic Gradient Descent (ASGD) and Adaptive Moment Estimation (ADAM). Each optimizer requires the model's trainable parameters, as well as a set of optimizer-specific options such as the learning rate, weight decay, etc. In this work, all models are trained using SGD, which provides options for Weight-Decay and Nesterov-Momentum (see 2.3.1)

5.1.3 Learning Rate Schedulers

For implementing adaptive learning rate methods (see 3.3), Pytorch provides a set of scheduler implementations to adjust the learning rate based on the number of epochs [57]. The available schedulers include but are not limited to Time-Decay, Step-Decay, Expo-Decay and Decay-On-Plateau, which reduce the learning rate when a metric has stopped improving. Each scheduler requires a `.step()` function to be called for each learning rate update, which is usually done after a training epoch has finished.

5.1.4 Loss Functions

Pytorch provides multiple implementations of standard loss functions to be used for training and evaluating a model [58]. These include but are not limited to Mean Square Error (MSE), Cross Entropy Loss (CE) and Kullback Leibler Divergence Loss (KLD). In this work, all models models are trained using Cross Entropy Loss, except for knowledge distillation applications.

Knowledge Distillation Loss Function

As previously indicated, Knowledge Distillation is achieved by matching the softened probability distribution of two network output classes. To achieve this, a custom loss function is required. This work's implementation follows the L_{KD} proposed by Haitong [59] where the first component is implemented using the built-in KL-Divergence loss function of Pytorch to better utilize the underlying C-backend for efficiency. This way L_{KD} is now defined as:

$$L_{KD}(W_{student}) = aT^2 * KLDiv(Q_S^\tau, Q_T^\tau) + (1-a) * CrossEntropy(Q_S, y_{true}) \quad (5.1)$$

5.1.5 Quantization Schemes

PyTorch supports multiple approaches to quantizing a deep learning model [60]. The framework distinguishes the quantization process in two different schemes: Static Quantization and Fake Quantization.

Static Quantization

Commonly used after a network has been trained using floating point weight representations, static quantization converts the model to integer arithmetic following a given configuration. The configuration class (QConfig) describes how to quantize a layer or a part of the network by providing settings for activations and weights respectively. Based on this configuration and a small evaluation dataset, the framework calibrates the quantization parameters (see 3.5) using Observer classes, which determine the scale s and zero point z.

Fake Quantization

For quantization-aware training purposes Pytorch supports fake quantization-modules that are inserted after each network layer and effectively model their output with lower arithmetic representation. Using this method, all computations are initially carried out in floating point. Then, the modules calculate quantization errors in both the forward and backward passes, which can be used for optimizing the network. At the end of quantization-aware training, PyTorch provides conversion functions to convert the trained model into lower precision.

5.1.6 Pytorch CUDA Integration

Pytorch provides integration for CUDA compatible GPUs. In this work, all experiments where implemented using an `NVIDIA RTX-2060 Super` [47]. The device is equipped with multiple CUDA cores and high-bandwidth memory, making it suitable for deep learning applications. More detailed specifications can be found in 4.4.

5.1.7 Configuration Files

Configuration files are written in YAML [61], a human friendly data serialization standard for all programming languages. The files contain all the information needed to run an experiment such as network configuration, dataset, epochs, optimizer, loss function etc. The same file can be used for training a network using different methodologies, or evaluating the resulting model. Finally, all algorithms are designed to receive a directory path containing the configuration file. This directory is used for storing the final model, intermediate checkpoints, as well as logging.

5.1.8 Base Classes

For the entirety of training and validating the models, a small set of base classes is required. These are:

Validator

The validator is responsible for evaluating the accuracy of a network without calculating the backproagation gradients for optimizing the model. It is most commonly used between training epochs. The class constructor requires a network object, the validation dataset, as well as the loss function. The class implements the `validate()` method, which evaluates the network using the provided dataset.

TrainerSS

The essentials of training a neural network have been covered in chapter 2.3. In this work, optimizing a model using only a training dataset is described using the term "Self-Studying" (SS). The TrainerSS constructor requires a network object, a dataloader, an optimizer object, a loss function, a Validator instance for evaluating the network's performance after each epoch and a warmup parameter. All trainer classes implement the `train_epoch()` method, which trains the network using the entirety of the given dataset.

Algorithm 2: TrainerSS.train_epoch()

```
Input: Network, Dataloader, Optimizer, Loss Function, Validator,
       Warmup, Current Epoch;
Output: Optimized Network, Training Loss, Training Accuracy,
        Validation Loss, Validation Accuracy;
1  $Correct \leftarrow 0$;
2  $Running\_Loss \leftarrow 0$;
3  $Total \leftarrow$ Total amount of training inputs;
4  for $Inputs, Labels \leftarrow Dataloader$ do
5      Zero the Optimizer parameters;
6      $Outputs \leftarrow Network(Inputs)$;
7      Increment $Correct$ whith the amount of $Inputs$ correctly labeled;
8      $Loss \leftarrow Loss\_Function(Outputs, Labels)$;
9      $Running\_Loss \leftarrow Running\_Loss + Loss$;
10     Use Backpropagation to calculate the Error Gradients;
11     Use the Error Gradients to optimize the network;
12     if $Current\_Epoch < Warmup$ then
13         Step Warmup scheduler;
14     end
15 end
16 $Training\_Loss \leftarrow Running\_Loss / Total$;
17 $Training\_Accuracy \leftarrow Correct / Total$;
18 $Validation\_Loss, Validation\_Accuracy \leftarrow Validator.validate()$ ;
```

TrainerTS

Following the proposal of Kim et al. [1], TrainerTS implements "Tutor-Studying" which is essentially Knowledge Distillation (see 3.4). The TrainerTS constructor requires a student network object, a set of corresponding outputs from the teacher network which are used as labels, a dataloader, an optimizer object, a Validator instance for evaluating the network's performance after each epoch, the hyperparameters used in L_{KD} and a warmup parameter. For optimizing the student network, the trainer utilizes a custom loss function as described in 5.1.4

Algorithm 3: TrainerTS.train_epoch()

```
Input: Student Network, Teacher Outputs, Dataloader, Optimizer,
       Validator, Warmup, $\alpha$, T, Current Epoch;
Output: Optimized Student Network, Training Loss, Training Accuracy,
        Validation Loss, Validation Accuracy;
1  $Correct \leftarrow 0$;
2  $Running\_Loss \leftarrow 0$;
3  $Total \leftarrow$ Total amount of training inputs;
4  for $Inputs, Labels, Indexes \leftarrow Dataloader$ do
5      Zero the Optimizer parameters;
6      $Outputs \leftarrow Student_Network(Inputs)$;
7      Increment $Correct$ whith the amount of $Inputs$ correctly labeled;
8      $Teacher\_Outputs\_Batch \leftarrow$ Get corresponding $Teacher\_Outputs$ using
         their $Index$;
9      $Loss \leftarrow L_{KD}(Outputs, Labels, Teacher\_Outputs\_Batch, \alpha, T)$;
10     $Running\_Loss \leftarrow Running\_Loss + Loss$;
11     Use Backpropagation to calculate the Error Gradients;
12     Use the Error Gradients to optimize the network;
13     if $Current\_Epoch < Warmup$ then
14         Step Warmup scheduler;
15     end
16 end
17 $Training\_Loss \leftarrow Running\_Loss/Total$;
18 $Training\_Accuracy \leftarrow Correct/Total$;
19 $Validation\_Loss, Validation\_Accuracy \leftarrow Validator.validate()$ ;
```

TrainerCS

TrainerCS is responsible for the "Co-Studying" phase of Quantization-Aware Knowledge Distillation (see 3.6). Compared to TrainerTS, the module needs to calculate both the student and teacher outputs to update their weights in an online manner. The TrainerCS constructor requires a student network object, a teacher network object, a dataloader, two different optimizer objects for the student and teacher respectively, a Validator instance for evaluating the student network's performance after each epoch, the hyperparameters used in L_{KD} and a warmup parameter.

Algorithm 4: TrainerCS.train_epoch()

```
Input: Student Network, Teacher Network, Dataloader, Student
       Optimizer, Teacher Optimizer, Student Validator, Warmup, $\alpha$, T,
       Current Epoch;
Output: Optimized Student Network, Optimized Teacher Network,
        Training Loss, Training Accuracy, Validation Loss, Validation
        Accuracy;
1  $Correct \leftarrow 0$;
2  $Running\_Loss \leftarrow 0$;
3  $Total \leftarrow$ Total amount of training inputs;
4  for $Inputs, Labels, Indexes \leftarrow Dataloader$ do
5      Zero the Student Optimizer parameters;
6      Zero the Teacher Optimizer parameters;
7      $Student\_Out \leftarrow Student_Network(Inputs)$;
8      $Teacher\_Out \leftarrow Teacher_Network(Inputs)$;
9      Increment $Correct$ whith the amount of $Inputs$ correctly labeled by
         the student network;
10     $Student\_Loss \leftarrow L_{KD}(Student\_Out, Labels, Teacher\_Out, \alpha, T)$;
11     $Teacher\_Loss \leftarrow L_{KD}(Teacher\_Out, Labels, Student\_Out, \alpha, T)$;
12     $Running\_Loss \leftarrow Running\_Loss + Student\_Loss$;
13     Use Backpropagation to calculate both the Student's and the
         Teacher's Error Gradients;
14     Use the Error Gradients to optimize both the Student and the Teacher
         networks;
15     if $Current\_Epoch < Warmup$ then
16         Step Warmup scheduler;
17     end
18 end
19 $Training\_Loss \leftarrow Running\_Loss/Total$;
20 $Training\_Accuracy \leftarrow Correct/Total$;
21 $Validation\_Loss, Validation\_Accuracy \leftarrow Validator.validate()$ ;
```

5.1.9 Quantization

In this work both weights and activations of all networks quantized to 8-bit representation use the following quantizers:

8-bit Activation Quantizer

Activations are quantized to unsigned integers in the range of $[0, 255]$. The scale factor for quantization as well as the zero-point are determined using the running average minimum and average maximum values of each Tensor.

8-bit Weights Quantizer

Weights are quantized to signed integers in the range of $[-128, 127]$. The scale factor for quantization as well as the zero-point are determined identically to the activations quantizer.

Additionally, all networks quantized to 4-bit representation use the following quantizers:

4-bit Activation Quantizer

Activations are quantized to unsigned integers in the range of $[0, 15]$. The scale factor for quantization as well as the zero-point are determined using the running average minimum and average maximum values of each Tensor.

4-bit Weights Quantizer

Weights are quantized to signed integers in the range of $[-8, 7]$. The scale factor for quantization as well as the zero-point are determined identically to the activations quantizer.

5.1.10 Data Augmentation

As described in 2.4.3, data augmentation can reduce model overfitting by introducing noise in the training process. This technique is employed when training on the `CIFAR-10` and `CIFAR-100` datasets using a series of transforms provided by the framework's `Torchvision` package [62]. These are:

1. **Random Crop:** 4 pixels of zero-padding is applied in each direction and then a random crop is performed to extract a 32×32 image.
2. **Random Horizontal Flip:** The image is randomly flipped horizontally using a probability value of $p = 0.5$.
3. **Random Rotation:** The image is randomly rotated between a range of $[-15, 15]$ degrees.

5.2 Methodology

Deep learning requires repetitive experimentation. Hyperparameter tuning, different training methods and multiple model variations can exponentially increase the amount of tests needed before a robust and accurate network is produced. Unfortunately, training a model is by itself a time-consuming process, especially when large networks and complex training methods are involved. This work approaches the problem by evaluating a training method on smaller networks, before gradually increasing both the model complexity, as well as the training dataset size.

5.2.1 Models

DNN

The simplest network used for experimentation, `DNN` is a deep neural network of a single hidden layer. The model comes in four variations: `DNN-1200`, `DNN-800`, `DNN-300` and `DNN-30`. As the name suggests, each variation's hidden layer has 1200, 800, 300 and 30 neurons respectively.

KL_MLP

`KL_MLP` is a deep multi-layer perceptron utilizing two hidden layers. Similar to `DNN`, the network comes in two variations for knowledge-distillation experimentations: `KL_MPL_Teacher` and `KL_MPL_Student`. Both `DNN` and `KL_MLP` networks are trained on the `MNIST` dataset (see 3.1)

Model	**Features**				
	Input Layer	Layer 1	Layer 2	Layer 3	Output
KL_MLP_Teacher	28 * 28	1200	1200	1200	10
KL_MLP_Student	28 * 28	300	300	300	10

TABLE 5.1: KL_MLP model variants.

DenseNet

Approaching state-of-the-art architectures, DenseNet (see 3.2.5) is the biggest network used in this work's experiments. Along with the configurations provided in the original paper, three additional smaller variants where created for exploring the impact of network compression and the effectiveness of Knowledge Distillation. Their specifications are presented in rable 5.2. All

DenseNet variants are trained on the CIFAR-10 and CIFAR-100 datasets (see 3.1).

Layers	Output Size	DenseNet-21	DenseNet-37	DenseNet-63
Convolution	112×112	7 x 7 conv, stride 2		
Pooling	56×56	3 x 3 max pool, stride 2		
DenseBlock	56×56	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 1$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 2$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 3$
Transition Layer	56×56	1 x 1 conv		
	28×28	2 x 2 average pool, stride 2		
DenseBlock	28×28	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 2$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$
Transition Layer	28×28	1 x 1 conv		
	14×14	2 x 2 average pool, stride 2		
DenseBlock	14×14	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$
Transition Layer	14×14	1 x 1 conv		
	7×7	2 x 2 average pool, stride 2		
DenseBlock	7×7	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 2$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 8$
Classification Layer	1×1	7 x 7 global average pool		
		1000D fully-connected, softmax		

TABLE 5.2: Custom DenseNet architectures. The growth rate for all the networks is k = 32.

5.3 Baseline Experiments

The following experiments aim to determine each network's accuracy using floating point arithmetic. All networks where trained on their corresponding datasets multiple times to determine the best hyperparameter combinations. The resulting models will then serve as a baseline for exploring quantization and knowledge distillation techniques.

5.3.1 DNN

All variants of DNN are trained using identical hyperparameters. The networks are trained for 160 epochs using 64 images per mini-batch. The learning rate is chosen to be $lr = 0.1$ which is then divided by 2 at epochs 40, 80 and 120, additionally a warmup period of 2 epochs is chosen. The optimizer uses Nesterov momentum $m = 0.9$ and weight decay $wd = 5 * 10^4$. The results are shown in table 5.3.

Model	Top-1 Train Acc %	Top-1 Test Acc %
DNN-1200	99.96	98.68
DNN-800	99.96	98.73
DNN-300	99.94	98.67
DNN-30	99.17	97.86

TABLE 5.3: DNN test results on MNIST.

As is evident, the best performing variant for MNIST classification is `DNN-800` exceeding the test accuracy of `DNN-1200`. This can be attributed to model over-fitting, since both `DNN-800` and `DNN-1200` produce the same train accuracy.

5.3.2 KL_MLP

Similar to `DNN`, all variants of `KL_MLP` are trained using identical hyperparameters. The networks are trained for 100 epochs using 64 images per mini-batch. The learning rate is chosen to be $lr = 0.1$ which is then divided by 2 at epochs 20, 40 and 60, additionally a warmup period of 2 epochs is chosen. The optimizer uses Nesterov momentum $m = 0.9$ and weight decay $wd = 5 * 10^4$. The results are shown in table 5.4.

Model	Top-1 Train Acc %	Top-1 Test Acc %
KL_MLP_Teacher	98.61	98.45
KL_MLP_Student	99.96	98.39

TABLE 5.4: KL_MLP test results on MNIST.

5.3.3 DenseNet

For the purposes of this work, five different variants of DenseNet are evaluated for classifying the CIFAR-10 and CIFAR-100 datasets. All variants are trained using identical hyperparameters for both datasets. The networks are trained for 200 epochs using 64 images per mini-batch. The learning rate is chosen to be $lr = 0.1$ which is then divided by 5 at epochs 60, 120 and 160, additionally a warmup period of 2 epochs is chosen. The optimizer uses Nesterov momentum $m = 0.9$ and weight decay $wd = 5 * 10^4$. The results are shown in tables 5.5 and 5.6.

Model	Top-1 Train Acc %	Top-1 Test Acc %
DenseNet-21	98.93	92.14
DenseNet-37	99.9	93.92
DenseNet-63	99.98	94.47
DenseNet-121	99.98	94.48
DenseNet-201	99.98	95.23

TABLE 5.5: DenseNet test results on CIFAR-10.

Model	Top-1 Train Acc %	Top-1 Test Acc %
DenseNet-21	82.59	68.86
DenseNet-37	96.04	73.98
DenseNet-63	99.24	76.76
DenseNet-121	99.73	77.66
DenseNet-201	99.83	77.60

TABLE 5.6: DenseNet test results on CIFAR-100.

5.4 Knowledge Distillation Experiments

Following are the different experiments conducted for fine-tuning the implementation of Knowledge Distillation using the `TrainerTS` algorithm (see 5.1.8).

5.4.1 DNN

Knowledge Distillation is performed on `DNN` by assigning the `DNN-800` variant as a teacher, and the variants `DNN-300` and `DNN-30` as students. All models are trained for 160 epochs using 32 images per mini-batch. The initial learning rate is set to $lr = 0.1$ and divided by 5 in epochs 40, 80 and 120 with a warmup period of 2 epochs. Additionally the optimizer uses Nesterov momentum $m = 0.9$ and weight decay $wd = 5 * 10^4$. For both networks, 5 different values for hyperparameter α in L_{KD} are tested, while T is set to $T = 20$ for `DNN-300` and $T = 3$ for `DNN-30`. The different hyperparameters used for knowledge distillation, along with the resulting accuracy of each model is presented in table 5.7.

Model	α	**Top-1 Train Acc %**	**Top-1 Test Acc %**
DNN-300	0.5	99.87	98.62
	0.4	99.91	98.58
	0.3	99.94	98.64
	0.2	99.95	98.62
	0.1	99.96	98.69
DNN-30	0.5	99.14	97.91
	0.4	99.26	97.79
	0.3	99.37	97.99
	0.2	99.50	98.06
	0.1	99.51	97.81

TABLE 5.7: DNN Knowledge-Distillation test results on MNIST.

Both networks present small improvements compared to their baseline performance. However, the advantages of Knowledge Distillation are clearer on `DNN-30`, where the maximum improvement is observed using $\alpha = 0.2$. This indicates that the method is most beneficial in models with small representative power.

5.4.2 KL_MLP

In this experiment, Knowledge Distillation is applied from `KL_MLP_Teacher` to `KL_MLP_Student`. The model is trained for 10 epochs using 64 images per mini-batch. The optimizer is configured using a learning rate value of $lr = 0.1$, a warmup period of 1 epoch, Nesterov momentum $m = 0.9$. L_{KD} is configured using the hyperparameters $\alpha = 0.7$ and $T = 20$. A comparison between the baseline networks and the resulting model is presented in table 5.8.

Model	Training	Top-1 Train Acc %	Top-1 Test Acc %
KL_MLP_Teacher	Baseline	98.61	98.45
KL_MLP_Student	Baseline	99.96	98.39
KL_MLP_Student	KD	99.87	98.46

TABLE 5.8: KL_MLP Knowledge Distillation test results on MNIST.

Although the observed improvement is small, it should be noted that the baseline models presented minor accuracy differences. Nonetheless, training the student model using knowledge distillation achieves higher accuracy compared to the baseline teacher model.

5.4.3 DenseNet

For CIFAR-10 classification, Knowledge Distillation is applied from `DenseNet-63` to `DenseNet-21`. All models are trained for 200 epochs using 64 images per mini-batch. The optimizer uses Nesterov momentum $m = 0.9$, weight decay $wd = 5 * 10^4$ and is initialized with learning rate $lr = 0.01$ using a warmup period of 2 epochs. Multiple combinations of hyperparameters are tested, with the best results presented in table 5.9.

LR Scheduler Steps	**T**	**a**	**Top-1 Train Acc %**	**Top-1 Test Acc %**
60, 120, 160	20	0.7	99.63	92.36
60, 100, 120, 140, 160	20	0.7	99.34	92.74
60, 100, 120, 140, 160	20	0.8	99.29	92.71
60, 100, 120, 140, 160	8	0.7	99.41	92.32

TABLE 5.9: DenseNet-21 Knowledge Distillation test results on CIFAR-10. Column 1 denotes the sequence of epochs where the learning rate is divided by 5.

Again, Knowledge Distillation presents a small but consistent performance increase over the baseline models.

5.5 Post-Training Quantization

In this section, the impact of post-training quantization on the baseline models is explored. All experiments are conducted using fake-quantization, utilizing the quantizers described in 5.1.9. The final results are depicted in table 5.10.

Model	**Dataset**	**Top-1 Test Accuracy**		
		Baseline	8-bit Quant.	4-bit Quant.
DNN-1200	MNIST	98.68	98.68	98.05
DNN-800	MNIST	98.73	98.74	97.92
DNN-300	MNIST	98.67	98.65	98.15
DNN-30	MNIST	97.86	97.88	95.91
KL_MLP_Teacher	MNIST	98.45	98.46	98.13
KL_MLP_Student	MNIST	98.39	98.34	97.74
DenseNet-21	CIFAR-10	92.14	91.90	62.69
DenseNet-37	CIFAR-10	93.92	93.76	72.51
DenseNet-63	CIFAR-10	94.47	94.46	74.59
DenseNet-121	CIFAR-10	94.98	94.93	75.80
DenseNet-201	CIFAR-10	95.23	95.15	78.08
DenseNet-21	CIFAR-100	68.86	68.41	17.75
DenseNet-37	CIFAR-100	73.98	73.76	23.54
DenseNet-63	CIFAR-100	76.76	76.62	39.93
DenseNet-121	CIFAR-100	77.66	77.50	32.97
DenseNet-201	CIFAR-100	77.60	77.48	38.30

TABLE 5.10: Post-Training Quantization impact on baseline models.

Examining the performance figures reveals two significant trends. As expected, 4-bit representation yields the greatest degradation of accuracy across all models. However 8-bit representation results vary from minuscule improvements, to small but significant deterioration of performance. This can be attributed to quantization's regularization characteristics, aiding some models to generalize better on the problem.

5.6 Quantization-Aware Training

Following are the different experiments conducted for fine-tuning the implementation of Quantization-Aware training using the `TrainerSS` algorithm (see 5.1.8). QAT is performed using the baseline models generated in 5.3,. The networks are fake-quantized and further trained for a small number of epochs.

5.6.1 KL_MLP

All models are trained for 30 epochs using 64 images per mini-batch. The optimizer is initialized with learning rate $lr = 0.01$ using a warmup period of 2 epochs. Additionally, Nesterov momentum is set to $m = 0.9$ and weight decay to $wd = 5 * 10^{-4}$. The resulting performance metrics are presented in table 5.11

Model	Bit-Width	Top-1 Train Acc %	Top-1 Test Acc %
KL_MLP_Teacher	8	98.72	98.46
KL_MLP_Teacher	4	98.21	98.13
KL_MLP_Student	8	99.94	98.35
KL_MLP_Student	4	99.10	97.74

TABLE 5.11: KL_MLP QAT test results on MNIST.

As is evident by the test results, QAT only improved the accuracy of `KL_MLP_Student` in 8-bit representation. However, it should be noted that the models did not suffer from the quantization process in the first place.

5.6.2 DenseNet

For 8-bit quantization all models are trained for 60 epochs using 64 images per mini-batch. The optimizer is initialized with learning rate $lr = 10^{-6}$ using a warmup period of 1 epoch which is then divided by 5 in epochs 20 and 40. Additionally, Nesterov momentum is set to $m = 0.9$ and weight decay to $wd = 5 * 10^{-4}$. The resulting performance metrics are presented in table 5.12.

Model	Dataset	Top-1 Train Acc %	Top-1 Test Acc %
DenseNet-21	CIFAR-10	98.68	92.13
DenseNet-37	CIFAR-10	99.84	93.97
DenseNet-63	CIFAR-10	99.99	94.50
DenseNet-121	CIFAR-10	99.98	94.92
DenseNet-21	CIFAR-100	82.05	68.78
DenseNet-37	CIFAR-100	95.55	73.84

TABLE 5.12: DenseNet 8-bit QAT test results.

Quantization-Aware training yields small but significant performance improvements across all cases. Furthermore, most quantized models show promising results, by surpassing the accuracy of their baseline counterparts.

For 4-bit quantization the variants are trained for 140 epochs using 64 images per mini-batch. The optimizer is initialized with learning rate $lr = 10^{-3}$ using a warmup period of 1 epoch which is then divided by 5 per 20 epochs. Additionally, Nesterov momentum is set to $m = 0.9$ and weight decay to $wd = 5 * 10^{-4}$. The resulting performance metrics are presented in table 5.13.

Model	Dataset	Top-1 Train Acc %	Top-1 Test Acc %
DenseNet-21	CIFAR-10	93.63	88.9
DenseNet-37	CIFAR-10	92.83	89.23
DenseNet-63	CIFAR-10	96.17	90.54
DenseNet-121	CIFAR-10	96.68	91.46

TABLE 5.13: DenseNet 4-bit QAT test results.

The usefulness of QAT is better showcased in 4-bit quantization schemes. Here, the algorithm achieves over 25% performance increase compared to post-training quantization.

5.7 Quantization-Aware Knowledge Distillation

To compare the performance between QKD and traditional QAT, each experiment consists of the following method combinations (See 3.6):

1. **Baseline:** The network is trained and evaluated in full-precision representation.
2. **Quantized:** The network is trained in full-precision and evaluated in low-precision representation.
3. **SS:** The network is trained using only the "SS" phase, effectively acting as "QAT".
4. **SS + TS:** The network is trained using the "SS" phase and then the knowledge from the original teacher network is distilled.
5. **QKD (SS + CS + TS):** The three phases are combined to implement Quantization aware Knowledge Distillation (QKD).

The following experiments use 8-bit quantization.

5.7.1 MNIST

In this experiment, MNIST classification is achieved using the `KL_MLP` networks. The optimizer uses learning rate $lr = 0.01$ and Nesterov momentum $m = 0.9$ in all experiments except the "TS" phase of QKD where the learning rate is set to $lr = 0.1$. Similarly all "TS" and "CS" phases use $L_{KD}()$ with $T = 20$ and $a = 0.7$. The student network is trained for 30 epochs in "SS" phase, 50 epochs in "CS" phase and 40 epochs in "TS" phase.

Method	**Top-1 Train Acc %**	**Top-1 Test Acc %**
Baseline	99.96	98.39
Quantized	99.89	98.34
SS	99.94	98.35
SS+TS	99.96	98.63
QKD (SS+CS+TS)	99.99	98.79

TABLE 5.14: KL_MLP Performance comparison of training methods.

The experimental results on MNIST are shown in table 5.14. The teacher network achieved top-1 accuracy of 98.45%. We can see that due to the regularization provided both by quantization and knowledge distillation the student performs better than the teacher network using QKD training.

5.7.2 CIFAR-10

For CIFAR-10 classification, the networks Densenet-21 and Densenet-63 are used as student and teacher networks respectively. All experiments used SGD optimizer with Nesterov momentum $m = 0.9$, weight decay $wd = 5 * 10^{-4}$ and mini-batch of size 64. The hyperparameters used in each phase are:

- **SS Phase**

  The quantized student network is trained for 60 epochs using learning rate $lr = 10^{-6}$, which is initialized using one warmup epoch and then divided by 5 in epochs 20 and 40.

- **CS Phase**

  The Co-Studying phase is executed for 100 epochs using $lr = 10^{-4}$ which is divided by 5 in epochs 40, 60 and 80. Knowledge Distillation is performed both from teacher to student and vice versa using the loss hyperparameters $T = 8$ and $\alpha = 0.7$.

- **TS Phase**

  Tutor-Studying is performed for 100 epochs using $lr = 10^{-3}$ which is divided by 5 in epochs 40, 60 and 80. Knowledge Distillation is performed from teacher to student using the loss hyperparameters $T = 8$ and $\alpha = 0.7$.

Method	**Top-1 Train Acc %**	**Top-1 Test Acc %**
Baseline	98.93	92.14
Quantized	98.41	91.90
SS	98.68	92.13
SS+TS	98.96	92.26
QKD (SS+CS+TS)	99.00	92.50

TABLE 5.15: DenseNet Performance comparison of training methods for CIFAR-10.

The experimental results on CIFAR-10 are shown in table 5.15. The teacher network achieved baseline top-1 accuracy of 94.47%. It is evident in this case that simple QAT is not sufficient to compensate for the quantization losses. However, SS + TS and QKD show significant improvements, surpassing the baseline accuracy of the student model, with QKD performing the best.

Using the hyperparameters as described above a set of experiments where carried out for model variants DenseNet-21, DenseNet-37, DenseNet-63 and DenseNet-121 to compare between QAT and QKD performance for 8-bit and 4-bit quantization. The resulting metrics are presented in figure 5.1, where it can be observed that QKD consistently outperforms QAT.

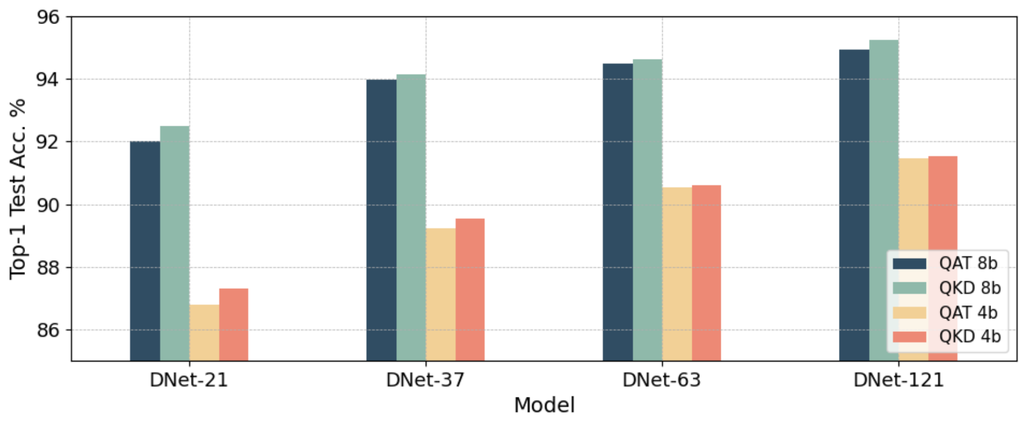


FIGURE 5.1: QAT and QKD Performance on CIFAR-10 for 8-bit and 4-bit quantization schemes

5.7.3 CIFAR-100

For CIFAR-100 classification, the networks Densenet-37 and Densenet-121 are used as student and teacher networks respectively. All experiments used SGD optimizer with Nesterov momentum $m = 0.9$, weight decay $wd = 5 * 10^{-4}$ and mini-batch of size 64. The hyperparameters used in each phase are identical to those used on CIFAR-10 experiments.

Method	**Top-1 Train Acc %**	**Top-1 Test Acc %**
Baseline	96.04	73.98
Quantized	95.20	73.76
SS	95.55	73.81
SS+TS	95.31	73.78
QKD (SS+CS+TS)	95.38	73.84

TABLE 5.16: DenseNet Performance comparison of training methods for CIFAR-100.

The experimental results on CIFAR-100 are shown in table 5.16. The teacher network achieved top-1 accuracy of 77.66%. Again SS + TS and QKD show similar improvements to CIFAR-10, with QKD performing the best.

5.8 Memory Footprint

In this section, the memory footprint of all quantized networks are evaluated and compared to the baseline floating-point models.

Model	**# Parameters**	**Model Size (MB)**		
		Floating Point	8-bit Quant.	4-bit Quant.
Densenet-21	419420	1.60	0.40	0.20
Densenet-37	1008964	3.85	0.96	0.48
Densenet-63	2353676	8.98	2.24	1.12
Densenet-105	5358244	20.44	5.11	2.56
Densenet-121	7048548	26.89	6.72	3.36
Densenet-201	18277220	69.72	17.43	8.72
KL_MLP_Teacher	2395210	9.14	2.28	1.14
KL_MLP_Student	328810	1.25	0.31	0.16
dnn-30	24790	0.09	0.02	0.01
dnn-300	328810	1.25	0.31	0.16
dnn-800	1276810	4.87	1.22	0.61
dnn-1200	2395210	9.14	2.28	1.14

5.9 Conclusions

Quantizing a neural network can significantly degrade it's representative power, and thus reduce it's effectiveness. In this work, we prove that both QAT, as well as QKD can minimize the negative effects of quantization on a neural network, with the latter proving the most effective. Additionally, all experiments on QKD suggest that the additional CS phase as originally proposed by Kim et al. [1] contributes significantly to the training process.

Chapter 6

FPGA Implementation

In chapter 5, different methodologies for model quantization and compression where evaluated. Utilizing the aforementioned techniques, a set of modified DenseNet models are prepared for execution on a Xilinx DPU using the Vitis-AI development stack (4.3).

The workflow consists of the following steps:

- Modify the original models to fit the constraints of Vitis-AI.
- Train the networks with QKD using the Vitis-AI quantizer.
- Create a Xilinx Internal Representation (XIR) of the trainied model using the Vitis-AI compiler.
- Load the DPU IP-Core in the ZCU-102 development board and inference the compiled model.

6.1 DenseNet Modifications

Deploying the different variants of DenseNet in a Xilinx DPU requires a set of modifications to be implemented. These modifications aim to prepeare the model for the Vitis-AI toolchain.

As mentioned in 4.3.2, Vitis-AI Quantizer performs a variety of optimizations. One of these optimizations is Batch-Norm "fusion", which is performed on the trained model as a last step to eliminate Batch-Norm layers. Since Batch-Norm layers utilize the statistics of the entire training dataset (which are immutable regardless the network's input) in inference time, they essentially perform a static linear translation of their input (2.3.5). This translation can be "fused" in the features of a previous weighted layer, reducing the final size of the model.

The original model proposed by Huang et al. [2] utilizes "pre-activated" residual blocks (3.2.4) by implementing each "conv" block as a sequence of BN-ReLu-Conv (3.1), which offers significant advantages when training extremely deep residual networks. Unfortunately such a sequence cannot be fused after the training process is completed to eliminate the Batch-Norm (BN) layers. Additionally the original model contains "Transition" layers which consist of BN-Conv sequences, which too cannot be fused.

Another complication emerges when attempting to implement this type of residual block in a Vitis-AI environment. Xilinx DPUs do not support standalone activation layers, but instead integrate them as a part of weighted layers. Trying to compile a sequence of BN-ReLu-Conv using the Vitis-AI compiler produces a series of subgraphs aimed at executing the ReLu operation in software, introducing significant bottleneck to the critical path.

To overcome the aforementioned issues, the original model is modified to utilize the more conventional approach of Conv-BN-ReLu sequences for dense-blocks and Conv-BN for transition layers. To compare the impact of these modifications in the final test accuracy of the model the baseline experiments where repeated using the same hyper-parameters as before. The results are displayed in tables 6.1 and 6.2.

Model Variant	**Top-1 Train Acc %**		**Top-1 Test Acc %**	
	Baseline	Modified	Baseline	Modified
DenseNet-21	98.93	98.8	91.87	92.27
DenseNet-37	99.9	99.98	93.92	94.49
DenseNet-63	99.98	99.89	94.47	94.18
DenseNet-121	99.99	99.99	94.48	95.52
DenseNet-201	99.98	100	95.23	95.58

TABLE 6.1: Modified DenseNet evaluation on Cifar-10.

Model Variant	**Top-1 Train Acc %**		**Top-1 Test Acc %**	
	Baseline	Modified	Baseline	Modified
DenseNet-21	82.59	81.42	68.86	68.9
DenseNet-37	96.04	94.27	73.08	73.66
DenseNet-63	99.24	98.89	76.76	76.12
DenseNet-121	99.73	99.71	77.66	77.9
DenseNet-201	99.83	99.91	77.06	78.21

TABLE 6.2: Modified DenseNet evaluation on Cifar-100.

The modified version of DenseNet proposed in this work achieves comparable accuracy metrics to the original. This is to be expected since the use of pre-activated dense blocks presented significant advantages in cases of extremely deep ResNet architectures, which are not present in our experiments.

Finally, the Vitis-AI quantizer requires all operations of the model to be defined as PyTorch modules. In our case, all dense block intermediate output additions, as well as flattening the output of the last convolution layer to be accepted from the final fully-connected layer where replaced with corresponding modules. This does not change the functionality of the network, but is necessary to proceed with quantization.

6.2 Vitis-AI Toolchain

Xilinx offers the Vitis-AI toolchain in the form of ’Docker containers’. A Docker container is a standard unit of software that packages code and all its dependencies so the application runs quickly and reliably from one computing environment to another [63]. Xilinx provides Vitis-AI in two major variants, ’CPU’ and ’GPU’. This refers to the capabilities of the various ML frameworks that are included in the containers, the latter of which can utilize CUDA enabled GPUs to accelerate the training process. Although the ’CPU’ variant is available pre-built by Xilinx, the ’GPU’ version must be built using the provided Docker ’recipe’ [64]. In this work, the ’GPU’ version of Vitis-AI is used, since the models must be trained using the Vitis-AI Quantizer.

6.2.1 Vitis-AI Quantizer

Vitis-AI provides a PyTorch plugin named ’vai_q_pytorch’ [65], designed to replace the original quantizers of the framework. In this work, the workflow of ’Quantize Finetuning’ [66] provided by Xilinx is extended to implement both QAT and QKD algorithms as described in chapter 5. Following are the necessary modifications to the original algorithms.

- Replace PyTorch fake quantizers with vai_q_pytorch quantizers. To do so, vai_q_pytorch is set to ’calibration’ and ’qat_proc’ mode, which models the error gradients to enable quantization-aware training. Additionally, the quantizer constructor requires the expected input shape to be provided, which is easily derived from a random training sample.

- Save the trained model, along with additional quantization info required by vai_q_pytorch.
- Export the trained quantized model along with the quantization info required by Vitis-AI compiler. To achieve this, vai_q_pytorch requires a final evaluation of the network using a single input vector and batch size of 1.

The resulting quantized models used in the following experiments are presented in table 6.3.

Model	Method	Dataset	Top-1 Train Acc %	Top-1 Test Acc %
DenseNet-21	QAT	CIFAR-10	98.25	92.01
DenseNet-21	QKD	CIFAR-10	99.05	92.47
DenseNet-37	QAT	CIFAR-100	91.97	73.97
DenseNet-37	QKD	CIFAR-100	95.11	74.27

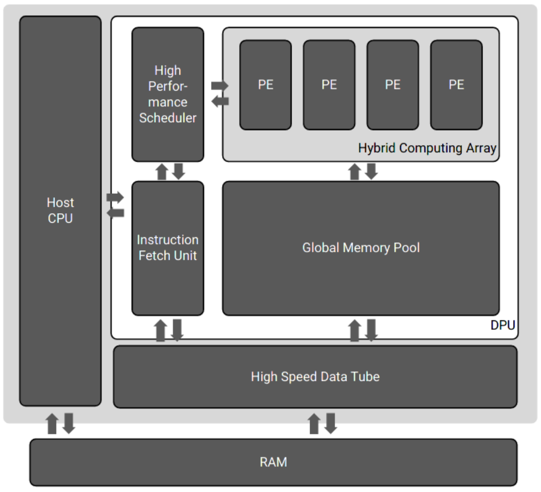
TABLE 6.3: VAI Quantizer Model Accuracy.

6.2.2 Xilinx DPUCZDX8G

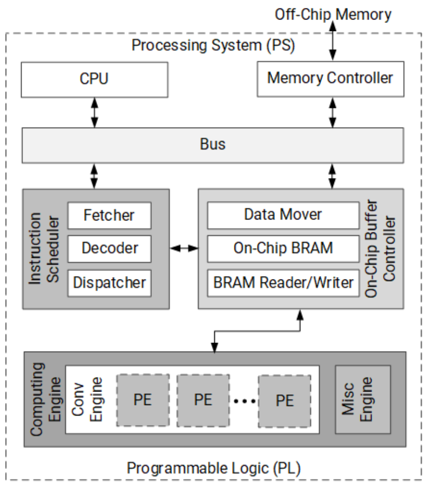
Xilinx provides a plethora of DPU IP cores. The DPUCZDX8G has been optimized for MPSoC devices and can be integrated as a block in the programmable logic (PL) of Zynq-7000 SoC and Zynq UltraScale+ MPSoCs with direct connections to the processing system (PS). It is designed to implement a neural network based on given instructions and accessible memory locations for input images as well as temporary and output data. The architecture is designed for 8-bit integer arithmetic. Additionally, the DPU is accompanied by software running on the application processing unit (APU) to service interrupts and coordinate data transfers [67].

The DPUCZDX8G has the following features:

- One AXI slave interface for accessing configuration and status registers.
- One AXI master interface for accessing instructions.
- Supports individual configuration of each channel.
- Supports optional interrupt request generation.



(A) DPU Top-Level Block Diagram URL



(B) DPU Hardware Architecture URL

FIGURE 6.1: Xilinx DPU Architecture

The detailed hardware architecture of the DPU is shown in figure 6.1. It is composed of a high performance scheduler module, a hybrid computing array module, an instruction fetch unit module, and a global memory pool module. Instructions for the operation of the computation engine are stored and fetched from the off-chip memory. To reduce the amount of external memory bandwidth and achieve higher throughput, the accelerator uses on-chip memory to buffer input, intermediate, and output data, reusing as much as possible. The computation engine comprises a deep pipeline of processing elements (PEs), which are in turn based on the fine-grained building blocks such as multipliers, adders, and accumulators found in Xilinx devices.

DSP Double Data Rate

To further improve the performance of the accelerator, the Double Data Rate (DDR) technique is used to double the throughput of the chip's DSPs. Therefore, two input clocks for the DPU are needed: One for general logic and another at twice the frequency for DSP slices [68].

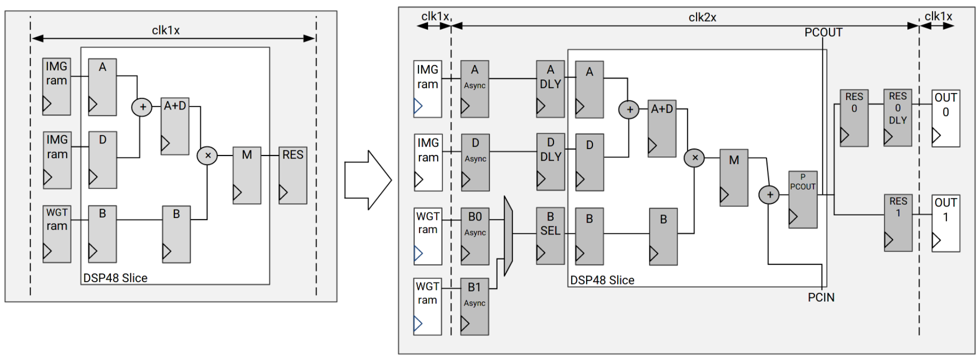


FIGURE 6.2: Difference between DPUCZDX8G without DSP DDR (**Left**) and DPUCZDX8G Enhanced Usage (**Right**).

DPU Configuration

The IP Core comes with a variety of user-configurable parameters to optimize resource usage. Different configurations can be selected for DSP slices, LUT, block RAM, and UltraRAM usage based on the amount of available programmable logic resources. Additionally, a selection of various features can be enabled such as average pooling, channel augmentation, depthwise convolution and softmax. Furthermore, multiple DPUCZDX8G cores (up to 4) can will be instantiated in a single DPUCZDX8G IP to achieve higher performance.

The DPUCZDX8G IP can be configured with a variety of convolution architectures, each of them offering different parallelism of the convolution unit. The convolution architecture utilizes parallelism in three distinct dimensions: pixel parallelism (PP), input channel parallelism (ICP), and output channel parallelism (OCP).

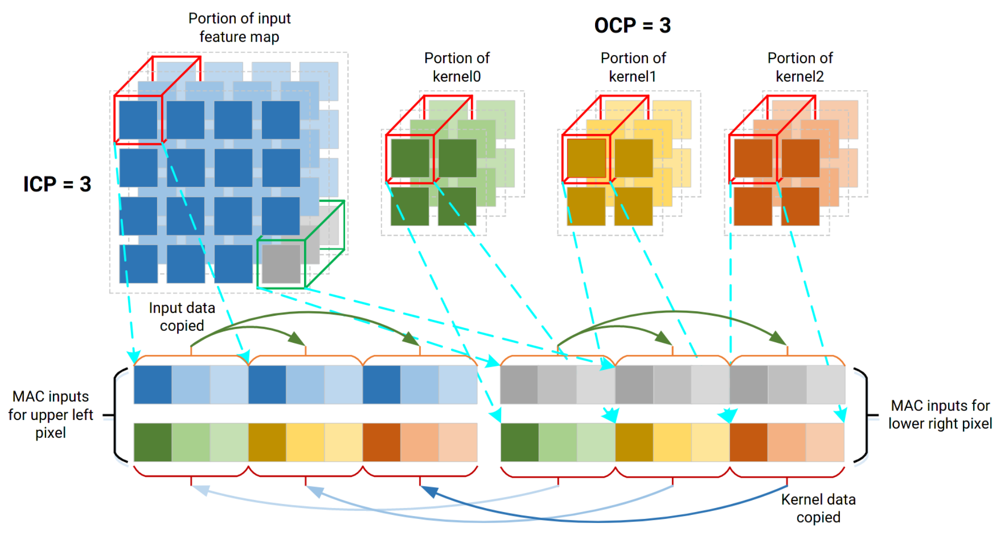


FIGURE 6.3: DPU Computation Parallelism.

The different variants of DPUCZDX8G architectures can be found in table 6.4. Each variant offers different levels of parallelism, peak operations per clock cycle and consequently resource requirements. It should be noted that in each clock cycle, the convolution array performs a multiplication and an accumulation, which are counted as two operations, so operations per cycle is equal to $PP * ICP * OCP * 2$.

Arch	PP	ICP	OCP	Ops/Cycle	LUT	Register	Block RAM	DSP
B512	4	8	8	510	27893	35435	73.5	78
B800	4	10	10	800	30468	42773	91.5	117
B1024	8	8	8	1024	34471	50763	105.5	154
B1152	4	12	12	1152	33238	49040	123	164
B1600	8	10	10	1600	38716	63033	127.5	232
B2304	8	12	12	2304	42842	73326	167	326
B3136	8	14	14	3136	47667	85778	210	436
B4096	8	16	16	4096	53540	105008	257	562

TABLE 6.4: DPUCZDX8G Architectures specifications and resource utilization for single core IP on ZCU-102 using Block-RAM.

RAM Usage

The DPU utilizes on-chip memory to buffer weights, bias, and intermediate features. DPUCZDX8G provides a RAM Usage option, which determines the total amount of on-chip memory used in different architectures for all cores in the IP. High RAM usage results to larger resource utilization, but implies higher performance in each core. The differences in resource requirements are illustrated in table 6.5.

Architecture	Low RAM Usage	High RAM Usage
B512	73.5	89.5
B800	91.5	109.5
B1024	105.5	137.5
B1152	123	145
B1600	127.5	163.5
B2304	167	211
B3136	210	262
B4096	257	317.5

TABLE 6.5: Number of BRAM36K blocks in different architectures for each DPUCZDX8G core.

Channel Augmentation

The DPUCZDX8G provides architecture variants with high input channel parallelism. It is not uncommon for networks to feature smaller number of channels in some layers, and thus not completely utilize the parallelism offered. However, when the number of input channels is larger than the channel parallelism, then channel augmentation may be utilized, to improve efficiency. Again, channel augmentation costs extra logic resources, as illustrated in table 6.6.

Architecture	Extra LUTs
B512	3121
B800	2624
B1024	3133
B1152	1744
B1600	2476
B2304	1710
B3136	1946
B4096	1701

TABLE 6.6: Extra LUTs of DPUCZDX8G with Channel Augmentation.

Depthwise Convolution

The DPUCZDX8G provides functionality for accelerating depth-wise convolutions [69]. This type of convolution is performed in two steps, with a separate convolution for each channel (using as many kernels as input channels), followed by a pointwise convolution (a standard convolution with kernel size 1x1). The parallelism of depthwise convolution is half that of the pixel parallelism. Depth-wise convolution costs extra logic resources, as illustrated in table 6.7.

Architecture	Extra LUTs	Extra Block RAMs	Extra DSPs
B512	1734	4	12
B800	2293	4.5	15
B1024	2744	4	24
B1152	2365	5.5	18
B1600	3392	4.5	30
B2304	3943	5.5	36
B3136	4269	6.5	42
B4096	4930	7.5	48

TABLE 6.7: Extra resources of DPUCZDX8G with Depthwise Convolution.

Average Pool

The AveragePool option determines whether the average pooling operation will be performed on the DPUCZDX8G or not. The supported sizes range from 2x2, 3x3, ..., to 8x8, with only square sizes supported. The extra resources with Average Pool is listed in table 6.8.

Architecture	Extra LUTs
B512	1507
B800	2016
B1024	1564
B1152	2352
B1600	1862
B2304	2338
B3136	2574
B4096	3081

TABLE 6.8: Extra LUTs of DPUCZDX8G with Average Pooling.

Relu Type

The ReLU Type option determines which kind of ReLU function can be used in the DPUCZDX8G. By default, the accelerator supports ReLu and ReLu6. Additionally LeakyRelu can be enabled with a small additional cost of resources (347 to 706 extra LUTs).

Softmax

This option enables the DPUCZDX8G to execute softmax layers, which can be more than 150x times faster than a software implementation. The maximum labels of hardware softmax is 1023. The extra resources with Softmax enabled are 9580 extra LUTs, 8019 extra FlipFlops, 4 extram BRAM blocks and 12 extra DSPs.

DSP Usage

The default configuration of DPUCZDX8G utilizes DSP48E slices for multiplication and accumulation operations in the convolution modules. The IP provides an additional option to disable the use of DSPs for accumulation named 'Low DSP Usage'. This reduces the usage of DSPs but requires an additional amount of LUTs and registers for accumulation operations. The extra logic utilization compared of high and low DSP usage is shown in table 6.9.

Architecture	Extra LUTs	Extra Block RAMs	Fewer DSPs
B512	1418	2515	-32
B800	1903	4652	-40
B1024	1445	3069	-64
B1152	2550	4762	-48
B1600	1978	3520	-80
B2304	3457	6219	-96
B3136	1661	3900	-112
B4096	2525	7359	-128

TABLE 6.9: Resources of Low DSP Usage Compared to High DSP Usage.

6.3 DPUCZDX8G Implementation

Xilinx offers a pre-compiled version of DPUCZDX8G with 3 DPU Cores configured at 300Mhz for deployment on a ZCU102 evaluation board. Additionally Channel Augmentation, Softmax, LeakyRelu enabled. The resource utilization for its implementation on a ZCU-102 is depicted in table 6.10. As mentioned in 6.2.2, the device is designed for 8-bit integer arithmetic, and thus only quantized versions of CNN models are used for performance evaluation.

PL/DSP Clock Frequency	300/600 MHz
LUT	58.60%
BRAM Usage	84.45%
DSP Usage	66.90%
Power Consumption	22.8W

TABLE 6.10: DPUCZDX8G Resource Requirements on ZCU-102.

Compiling the final model required by DPUCZDX8G is achieved using the XIR-based toolchain (`vai_c_xir`). The XIR based compiler takes the quantized model as the input. First, it transforms the input models into the XIR format. A unified representation that eliminates most of the variations among different frameworks. Then, various optimizations are applied on the graph. Additionally, the compiler breaks up the graph into several subgraphs on the basis of whether the operation can be executed on the DPU. The compiler requires an additional architecture file of the target DPU. This is exported by the configured DPU core and represents its capabilities.

6.4 Evaluating The Model

Xilinx provides a series of C++ and Python APIs called `VART` which stands for "Vitis-AI Runtime" [70]. The API features asynchronous submission and collection of jobs to the accelerator. In this work, the python version of the API is used to evaluate the final models.

The python script developed for model evaluation deserializes the generated XIR model, initializes a number of threads with the required subgraphs and data and waits for the threads to finnish their execution. This is achieved using Python's threading library [71]. Finally a set of performance metrics are calculated based on the time of execution and accuracy of the model.

Algorithm 5: DPU Runner

```
Input: Dataset, Model, Thread_Count;
Output: Throughput, Time, Test Accuracy;
1  $Images, Labels \leftarrow Dataloader(Dataset)$;
2  $Total \leftarrow Count(Images)$;
3  $Graph \leftarrow VART\_XIR\_Deserializer(Model)$;
4  $DPU\_Runners \leftarrow []$;
5  $Threads \leftarrow []$;
6  $Predictions \leftarrow []$;
7  foreach $i$ in $(1, \ldots, Thread\_Count)$ do
8      $runner \leftarrow VART\_Create\_Runner(Graph)$;
9      $DPU\_Runners.append(runner)$;
10 end
11 foreach $i$ in $(1, \ldots, Thread\_Count)$ do
12     $Threads.append(DPU\_Runners[i], Subset(Images, i))$;
13 end
14 $Start\_Time \leftarrow time()$;
15 foreach $i$ in $(1, \ldots, Thread\_Count)$ do
16     $Threads[i].start()$;
17 end
18 foreach $i$ in $(1, \ldots, Thread\_Count)$ do
19     $Threads[i].join()$;
20 end
21 $Stop\_Time \leftarrow time()$;
22 foreach $i$ in $(1, \ldots, Thread\_Count)$ do
23     $Labels.append(Threads[i].results)$;
24 end
25 $Correct \leftarrow Predictions = Labels$;
26 $Test\_Accuracy \leftarrow Correct/Total$;
27 $Time \leftarrow Stop\_Time - Start\_Time$;
28 $Throughput \leftarrow Total/Time$ ;
```

The image provided by Xilinx is based on PetaLinux [72], and comes with all the necessary VART libraries. It can be flashed in a micro-sd card and used to boot the evaluation board. Additionally, the necessary python script, compiled XIR models and datasets are transferred to the memory card. After the

Peta-Linux have booted, root access to the system can be acquired through SSH [73].

Chapter 7

Results

7.1 Performance Metrics

7.1.1 Latency

Latency, is the time required for accomplishing a single task. In this work, latency is defined as the time taken for a specific platform to process a single image.

7.1.2 Throughput

In general terms, throughput is defined as the number of tasks accomplished in a unit time. The higher the throughput, the higher the rate at which something in processed. In this work, throughput is defined as the number of images classified per second.

$$Throughput = \frac{Images}{Time(sec)} \tag{7.1}$$

7.1.3 Power and Energy Consumption

In physics, power is the amount of energy transferred or converted per unit time. In the International System of Units (SI) it is measured in Watts (W).

$$P = \frac{W}{\Delta t} \tag{7.2}$$

Where W is the work and Δt is the elapsed time.

Measured in Joules (J), energy consumption is defined as the energy required for accomplishing a specific task in a specific time amount. It is

$$E = P * T \tag{7.3}$$

Where P is the required power and T is the time needed to complete the task.

In this work, the Energy Consumption/Image metric is calculated as shown in equation 7.4. The minimum value of the energy consumption based on latency or throughput is selected to represent the optimal case for each platform.

$$\frac{EnergyConsumption}{Image} = min\{TotalPower * Latency, \frac{TotalPower}{Throughput}\} \tag{7.4}$$

7.2 Overall Performance

This sections aims to quantify and compare the performance of three different platforms. Although all models have been trained on both CIFAR-10 and CIFAR-100 datasets, the overall performance metrics are based only on the latter. Both datasets comprise images of identical dimensions and thus both perform similar.

Benchmarking on CPU and GPU is achieved using PyTorch. The framework can configure the number of workers to be used for its inference procedure. Workers are orchestration processes designed to run in parallel, loading data to RAM and inferencing the images. As a rule of thumb, the number of workers used to inference a network should be equal to the number of threads in a CPU. Alternatively, inferencing with a GPU requires only one CPU worker. Using multiple workers in this case can create communication bottlenecks and hinder performance. In this work, CPU experiments used six workers (as many threads provided in an i5-8600K) and a single worker for GPU experiments.

7.2.1 CPU FP Representation

Full-precision performance metrics on the CPU where acquired for models DenseNet-21, DenseNet-37, DenseNet-63, DenseNet-121 and DenseNet-201. Inference latency (fig. 7.1) is smallest for a batch size of single images. However throughput (fig. 7.2) is maximized in the range of 8 to 32 images per batch depending on the model. In that range, the minimum energy consumption per image is also observed (fig. 7.3).

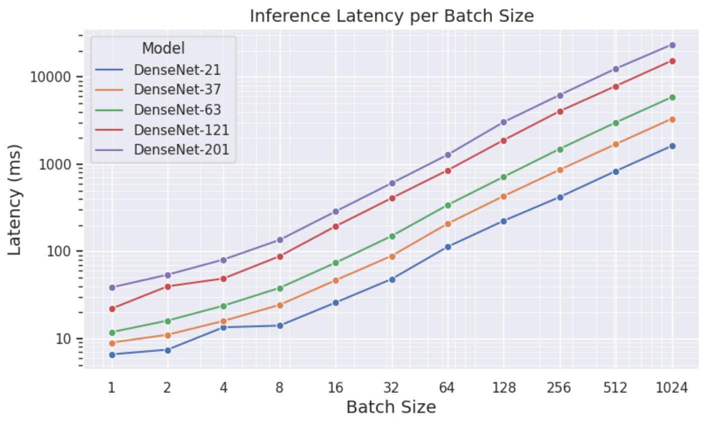


FIGURE 7.1: CPU Latency.

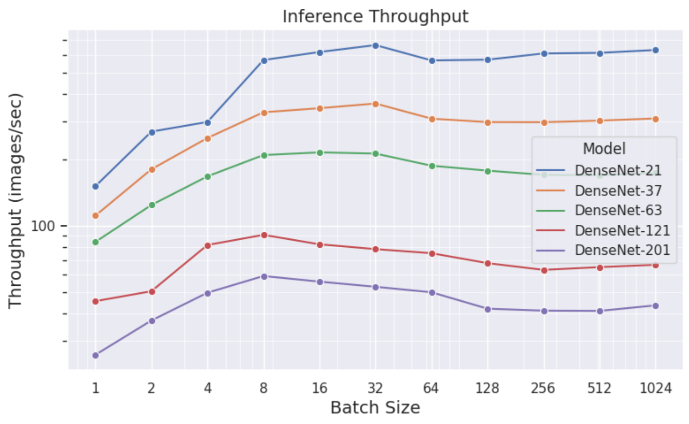


FIGURE 7.2: CPU Throughput.

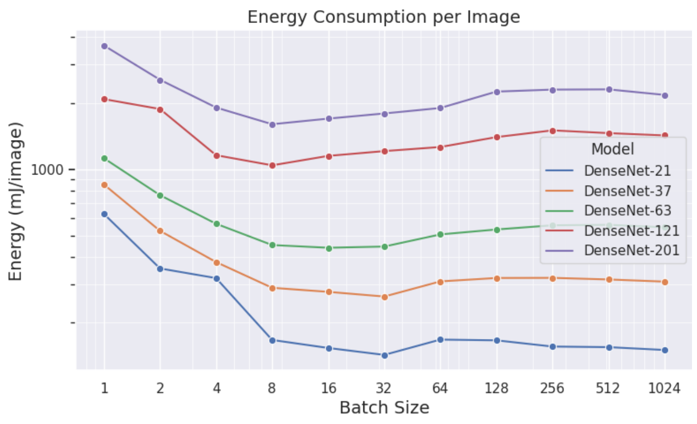


FIGURE 7.3: CPU Energy Consumption.

7.2.2 CPU 8-bit Representation

Performance metrics on the Quantized CPU where acquired for the modified versions of DenseNet-21, DenseNet-37, DenseNet-63, DenseNet-121 and DenseNet-201, as defined in 6.1. Inference latency (fig. 7.4) is smallest for a batch size of single images. However throughput (fig. 7.5) is maximized for batch sizes exceeding 128 images ber bach, depending on the model. In that range, the minimum energy consumption per image is also observed (fig. 7.6).

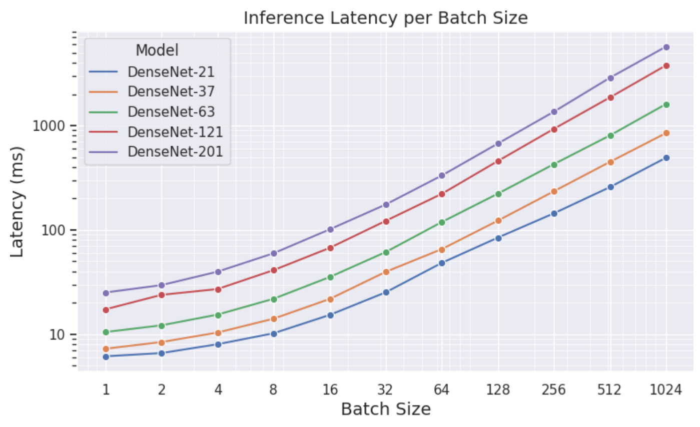


FIGURE 7.4: Quantized CPU Latency.

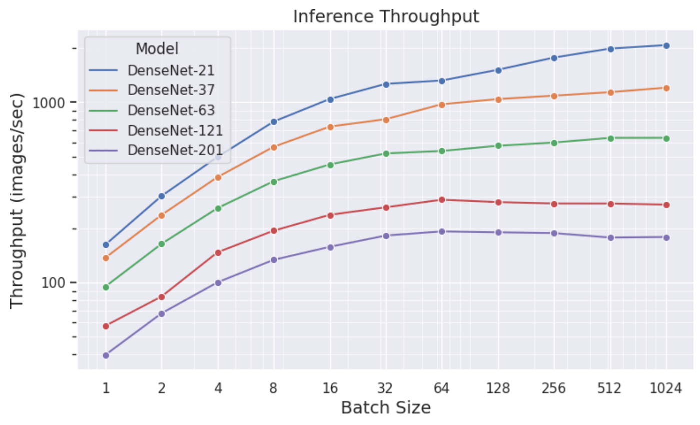


FIGURE 7.5: Quantized CPU Throughput.

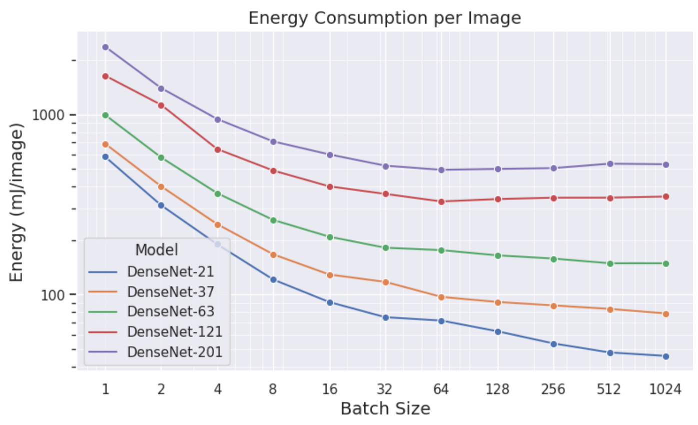


FIGURE 7.6: Quantized CPU Energy Consumption.

7.2.3 GPU

Performance metrics on the GPU where acquired for models DenseNet-21, DenseNet-37, DenseNet-63, DenseNet-121 and DenseNet-201. Inference latency (fig. 7.7) increases for batch sizes 16 and greater. Compared to CPU performance, the platform presents a lower limit in latency requirements, and thus decreasing the batch size below 8 does not yield faster inference.

As expected, throughput (fig. 7.8) benefits from the increase in batch sizes, where it plateaus in the range of 512 to 1024 images per batch. In that range, the minimum energy consumption per image is also observed (fig. 7.9).

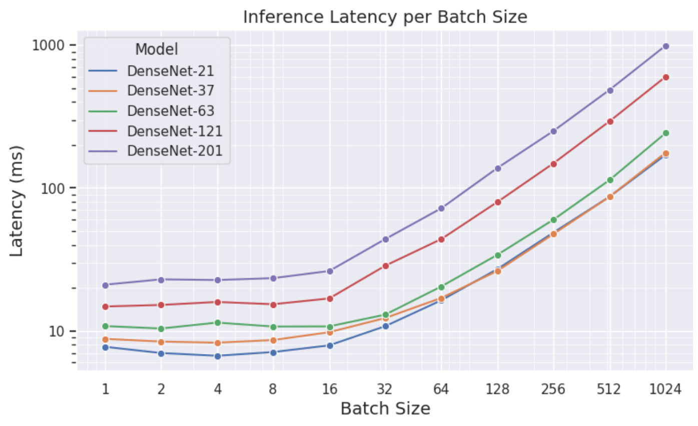


FIGURE 7.7: GPU Latency.

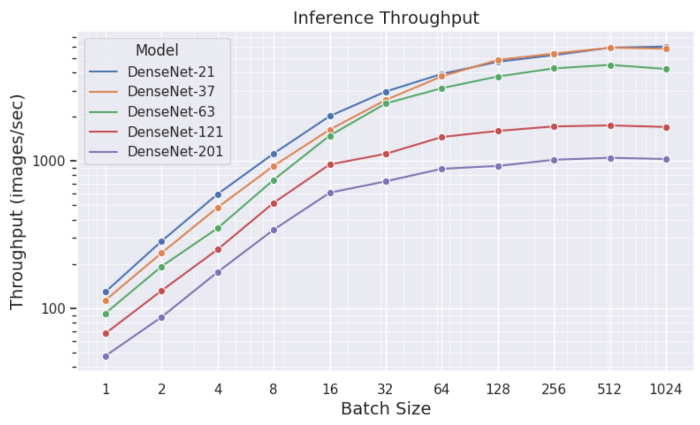


FIGURE 7.8: GPU Throughput.

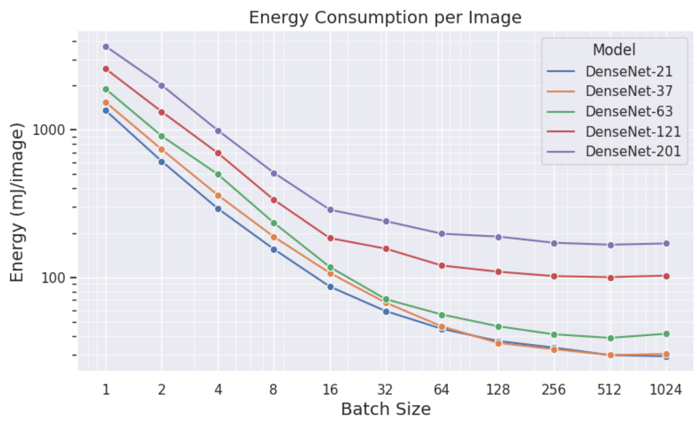


FIGURE 7.9: GPU Energy Consumption.

7.2.4 Xilinx DPU

Performance metrics on the DPU where acquired for the modified versions of DenseNet-21, DenseNet-37 and DenseNet-63 as defined in 6.1. Xilinx DPU cores are designed to operate with one image per batch.

As mentioned in **??**, the python script developed for inferencing the DPU features multi-threading capabilities using Python's threading library. Each thread orchestrates the necessary memory transactions and submits a job to the DPU scheduler.

Inference latency is presented in figure 7.10. Similar to CPU, minimum latency is observed with one thread. However, throughput (fig. 7.11) is maximized in the range of 5 to 8 images per batch depending on the model. In that range, the minimum energy consumption per image is also observed (fig. 7.12).

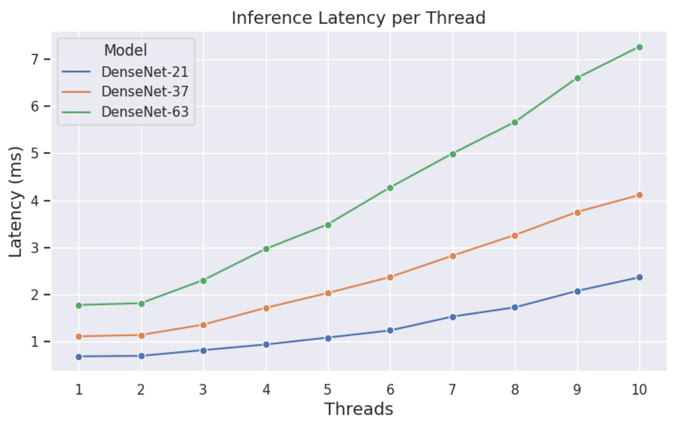


FIGURE 7.10: DPU Latency.

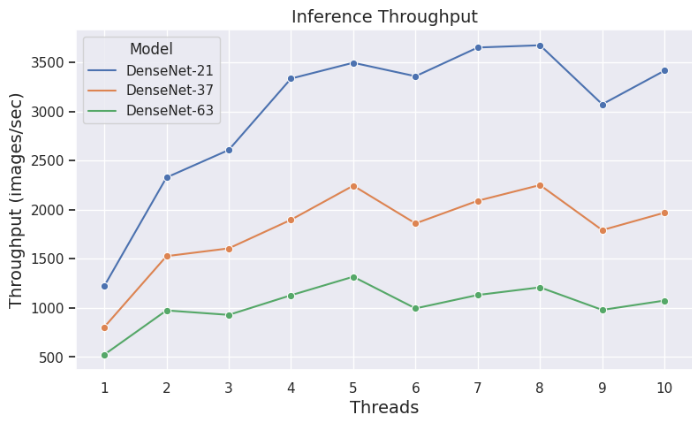


FIGURE 7.11: DPU Throughput.

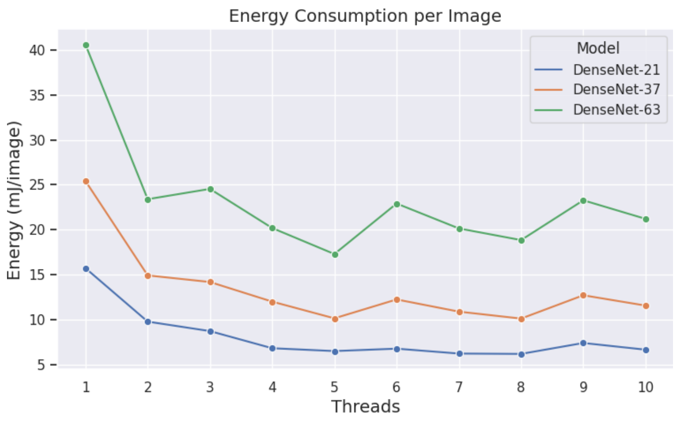


FIGURE 7.12: DPU Energy Consumption.

7.2.5 Comparisons

The throughput and latency speedups, as well as the energy efficiencies for every platform, are calculated compared to the full-precision CPU metrics. Since all platforms demonstrate optimal latency and throughput performance in different batch sizes or count of threads, energy efficiency is calculated for both cases. 'QCPU' denotes the evaluation of quantized networks using the CPU. The final results are presented in tables 7.1, 7.2 and 7.3.

Platform	CPU	QCPU	GPU	DPU
Throughput (images/s)	665.26	1717.07	5995.64	3674.32
Throughput Speedup	1.00x	2.58x	9.01x	5.52x
Latency (ms)	6.59	7.92	6.7	0.69
Latency Speedup	1.00x	0.83x	0.98x	9.55x
Energy Con/Img (best latency)	626.05	752.4	1172.5	15.73
Energy Efficiency (best latency)	1.00x	0.83x	0.53x	39.80x
Energy Con/Img (best throughput)	142.8	55.33	29.19	6.21
Energy Efficiency (best throughput)	1.00x	2.58x	4.89x	23.00x

TABLE 7.1: DenseNet-21 Performance Results.

Platform	CPU	QCPU	GPU	DPU
Throughput (images/s)	360.65	1020.56	5890.41	2250.92
Throughput Speedup	1.00x	2.83x	16.33x	6.24x
Latency (ms)	8.98	10.45	8.27	1.11
Latency Speedup	1.00x	0.86x	1.09x	8.09x
Energy Con/Img (best latency)	853.1	992.75	1447.25	25.42
Energy Efficiency (best latency)	1.00x	0.86x	0.59x	33.56x
Energy Con/Img (best throughput)	263.41	93.09	29.71	10.13
Energy Efficiency (best throughput)	1.00x	2.83x	8.87x	26.00x

TABLE 7.2: DenseNet-37 Performance Results.

Platform	CPU	QCPU	GPU	DPU
Throughput (images/s)	216.19	458.19	4499.56	1317.68
Throughput Speedup	1.00x	2.12x	20.81x	6.10x
Latency (ms)	11.83	13.74	10.39	1.78
Latency Speedup	1.00x	0.86x	1.14x	6.65x
Energy Con/Img (best latency)	1123.85	1305.3	1818.25	40.58
Energy Efficiency (best latency)	1.00x	0.86x	0.62x	27.69x
Energy Con/Img (best throughput)	439.43	207.34	38.89	17.3
Energy Efficiency (best throughput)	1.00x	2.12x	11.30x	25.40x

TABLE 7.3: DenseNet-63 Performance Results.

The final results of the various performance metrics are also depicted using bar charts in figures 7.13, 7.14 and 7.15 for better visibility.

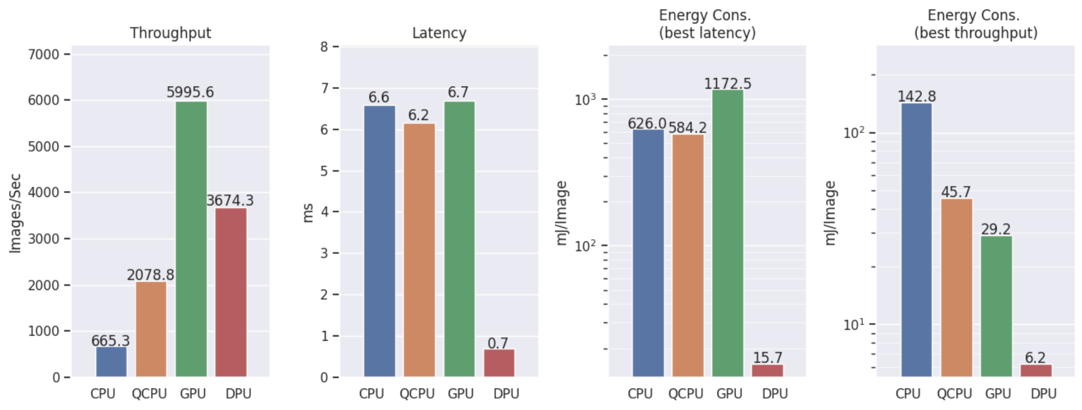


FIGURE 7.13: DenseNet-21 Inference Performance.

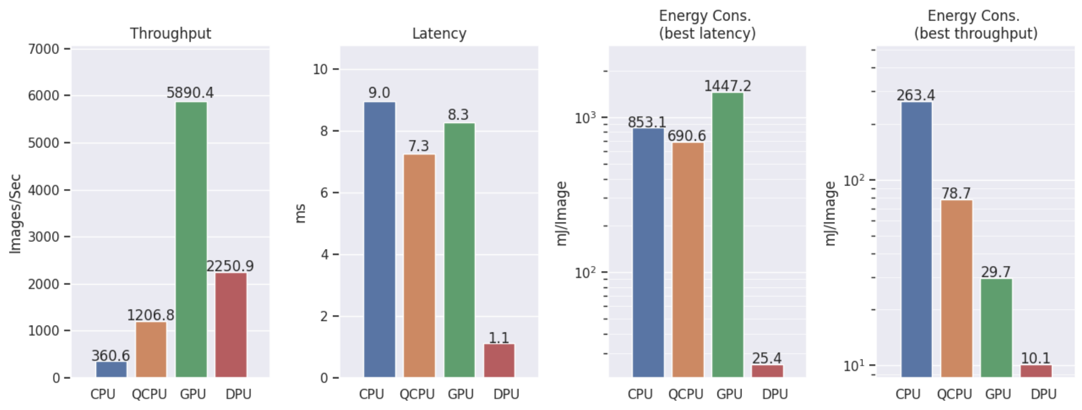


FIGURE 7.14: DenseNet-37 Inference Performance.

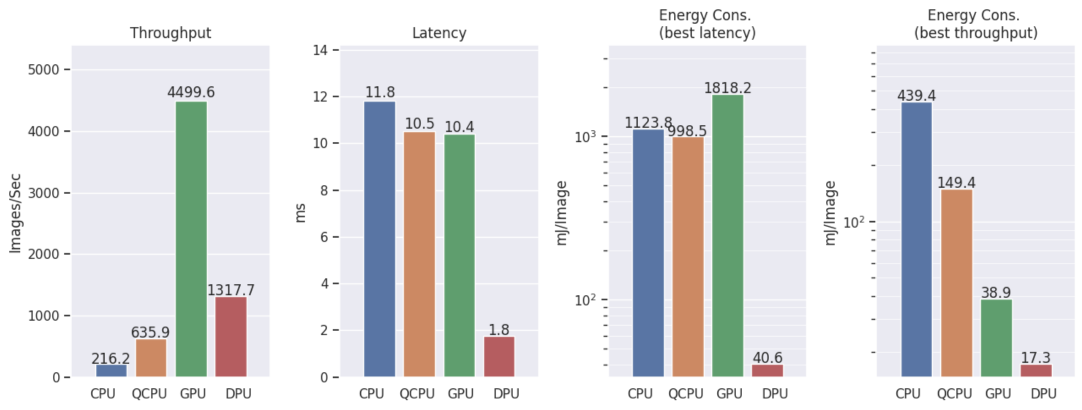


FIGURE 7.15: DenseNet-63 Inference Performance.

It can be observed that, for DenseNet-21 classification, the GPU performs best in terms of data throughput. However, both CPU and DPU yield better latency results, with the DPU exceeding a 9× speedup compared to all other platforms.

Energy consumption metrics paint an interesting picture. Here, the DPU performs the best on both cases (best throughput and best latency) by an order of magnitude. However, based on the throughput and latency requirements, CPU and GPU energy efficiencies differ significantly. Both platforms depend greatly on data parallelism to maximize their throughput, and thus demonstrate their highest efficiency in these working conditions. In this case, and although the GPU is the most power-hungry device, it exceeds the energy efficiency of the CPU due to its high-parallelism architecture.

Additionally, it should be noted that evaluation of quantized networks on CPU using SIMD instructions yields significant throughput speedups compared to full-precision, almost reaching 4×. Interestingly, although latency performance is positively effected, the speedup is minuscule.

By comparing the final results between the inference of DenseNet-21, DenseNet-37 and DenseNet63, it is evident that, as the model size and complexity increases, so does the overall performance of the GPU compared both to CPU and DPU. It is safe to assume that larger models will hinder the performance of all platforms. However, the GPU used in these experiments, with 8 gigabytes of high-speed memory, 2176 CUDA cores and 32 Tensor Cores will degrade with a smaller rate.

Although a few concrete observations can be made using the final results, the technology dissimilarity of the compared platforms should be noted. The

RTX 2060 Super is a fairly new GPU, launched in the mid 2019. The Intel i5-8600K CPU was released in late 2017, and the ZCU102's MPSoC, the Xilinx Zynq Ultrascale+ ZU9, launched in 2015. Hence, a fairer comparison between more similar platforms should be conducted.

Chapter 8

Conclusions and Future Work

This chapter aims to present the conclusions of this dissertation. Also, proposals for future work indicated by the research are suggested

8.1 Conclusions

The versatility of Convolution Neural Networks and their capability to classify complex data structures have made them an integral part machine learning applications, from image and sound recognition, to medical data analysis and many more. With the research community continuing to successfully explore new applications of such models in various industries, hardware capable of executing these models in a fast and energy efficient way is needed.

However, constraining the neural network to better fit certain hardware limitations can greatly increase the overall performance. This thesis' purpose was to explore a variety of methodologies for model compression and quantization, and their impact on accuracy. To achieve this, both Quantization Aware Training and Quantization Aware Knowledge Distillation algorithms where implemented and applied on a set of DenseNet variants. Training methodologies, quantization schemes and hyperparameter-tuning techniques where studied to produce quantized models, often exceeding the accuracy metrics of their non-quantized counterparts.

Finally, a comparison was made between three different platforms, a CPU, a GPU, and a Xilinx DPU, the latter of which was implemented on a ZCU102 FPGA evaluation board, and used quantized models. To achieve this, all quantization techniques had to be implemented on Xilinx's Vitis-AI framework to prepare the final models for evaluation.

The final performance metrics show that advanced quantization methodologies can produce neural networks of comparable capabilities, that enable cheaper and less power demanding hardware to outperform platforms of higher specifications.

8.2 Future Work

This thesis touches on a plethora of quantization training methods and available hardware implementations for neural network inference. However, a vast amount of literature on this subject exist, and continues to be expanded as of the time of this writing. Some the future work regarding this topic is presented below.

8.2.1 Incremental Quantization

Incremental Network Quantization, another quantization technique proposed by Zhou et al. [74] in 2017, promises lossless quantization of CNN networks by incrementally quantizing them in the training process. This could be used to augment Quantization Aware Knowledge distillation.

8.2.2 Quantization Scheme

Although this work touched briefly on 4-bit quantization, yielding promising results, further exploration of 2-bit and 1-bit quantized models is needed [75] [76]. Extremely lowering the arithmetic precision, and compensating with larger models can produce networks more suitable hardware acceleration. However, since most CNN accelerators in the market are based on 8-bit and 4-bit quantization, testing such a proposition requires custom hardware architecture.

8.2.3 Tested Platforms

In this work, quantized models where evaluated using a Xilinx DPU implementation, which is based in 8-bit quantization. However, as mentioned in 4.2.3, Google provides its own implementation of NN hardware acceleration. In addition to its main line of TPUs, google has developed a series for edge applications, called Coral [77], which too use 8-bit quantization for model inference. Evaluating the quantized models on this architecture can provide more useful data.

Additionally, a pre-compiled version of DPUCZDX8G offered by Xilinx is utilized for all experiments. However, as is mentioned in 6.2.2, the IP Core comes with a variety of user-configurable parameters to optimize resource usage. A customized implementation could be explored, reducing the amount of enabled features to only those required for inferencing DenseNet architectures, and increasing the total amount of available DPU cores.

Finally, although this dissertation draws inspiration from edge applications of CNNs, High Performance Computing (HPC) systems can too benefit from quantized neural networks. As described in 4.4.3, the Quad-FPGA Daughter Board (QFDB) utilizes the same MPSoC used in ZCU102 evaluation boards, and thus is compatible with the Xilinx DPU IP used for our experiments. The platform displays promising potential, and can use the benefits of quantized neural networks to meet HPC performance requirements, if the available resources are utilized efficiently.

8.2.4 Model Sizes

As mentioned in 7.2.5, it is evident that, as the model size and complexity increases, so does the overall performance of the GPU compared both to CPU and DPU. This phenomenon should be explored by evaluating larger models.

References

[1] Jangho Kim et al. *QKD: Quantization-aware Knowledge Distillation*. 2019. arXiv: 1911.12491 [cs.CV].

[2] Gao Huang et al. *Densely Connected Convolutional Networks*. 2018. arXiv: 1608.06993 [cs.CV].

[3] G. L. Shaw. "Donald Hebb: The Organization of Behavior". In: *Brain Theory*. Ed. by Günther Palm and Ad Aertsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 231–233. ISBN: 978-3-642-70911-1.

[15] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning Representations by Back-Propagating Errors". In: *Neurocomputing: Foundations of Research*. Cambridge, MA, USA: MIT Press, 1988, 696–699. ISBN: 0262010976.

[18] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[25] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

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

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'12. Red Hook, NY, USA: Curran Associates Inc., 2012, 1097–1105.

[34] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: 1409.1556 [cs.CV].

[35] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[36] Kaiming He et al. *Identity Mappings in Deep Residual Networks*. 2016. arXiv: 1603.05027 [cs.CV].

[37] Kaichao You et al. *How Does Learning Rate Decay Help Modern Neural Networks?* 2019. arXiv: 1908.01878 [cs.LG].

[38] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. *Distilling the Knowledge in a Neural Network*. 2015. arXiv: 1503.02531 [stat.ML].

[51] Fabien Chaix et al. "Implementation and Impact of an Ultra-Compact Multi-FPGA Board for Large System Prototyping". In: *2019 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. 2019, pp. 34–41. DOI: 10.1109/H2RC49586.2019.00010.

[52] Charisios Loukas. *Large Scale Design and Implementation of Convolutional Neural Networks based on Large FPGA Arrays*. en. 2020. DOI: 10.26233/HEALLINK.TUC.84315. URL: https://dias.library.tuc.gr/view/84315.

[59] Haitong Li. *Exploring Knowledge Distillation of Deep Neural Networks for Efficient Hardware Solutions*.

[74] Aojun Zhou et al. *Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights*. 2017. arXiv: 1702.03044 [cs.CV].

[75] Zhe Wang et al. *Towards Effective 2-bit Quantization: Pareto-optimal Bit Allocation for Deep {CNN}s Compression*. 2020. URL: https://openreview.net/forum?id=H1eKT1SFvH.

[76] Jiaxin Gu et al. *Bayesian Optimized 1-Bit CNNs*. 2019. arXiv: 1908.06314 [cs.CV].

External Links

[4] *Multilayer perceptron | Wikipedia*. URL: https://en.wikipedia.org/w/index.php?title=Multilayer_perceptron&oldid=1037690032.

[5] *Artificial neuron | Wikipedia*. URL: https://en.wikipedia.org/w/index.php?title=Artificial_neuron&oldid=1039675282.

[6] *Activation function | Wikipedia*. URL: https://en.wikipedia.org/w/index.php?title=Activation_function&oldid=1043520911.

[7] *Activation Functions in Neural Networks | Towards Data Science*. URL: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

[8] *How to Choose an Activation Function for Deep Learning | Machine Learning Mastery*. URL: https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/.

[9] *Gradient descent | Wikipedia*. URL: https://en.wikipedia.org/w/index.php?title=Gradient_descent&oldid=1043106649.

[10] *Mean squared error loss function | Peltarion Platform*. URL: https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/mean-squared-error.

[11] *Mean squared logarithmic error (MSLE) | Peltarion Platform*. URL: https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/mean-squared-logarithmic-error-(msle).

[12] *Mean absolute error loss function | Peltarion Platform*. URL: https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/mean-absolute-error.

[13] *Cross-Entropy | Wikipedia*. URL: https://en.wikipedia.org/w/index.php?title=Cross_entropy&oldid=1031879913.

[14] *Kullback–Leibler divergence | Wikipedia*. URL: https://en.wikipedia.org/w/index.php?title=Kullback%E2%80%93Leibler_divergence&oldid=1043560273.

[16] *Backpropagation | Wikipedia*. URL: https://en.wikipedia.org/w/index.php?title=Backpropagation&oldid=1032593246.

[17] *Backpropagation | Brilliant Math & Science Wiki*. URL: https://brilliant.org/wiki/backpropagation/.

[19] *Batch normalization | Wikipedia*. URL: https://en.wikipedia.org/w/index.php?title=Batch_normalization&oldid=1037955103.

[20] *A Gentle Introduction to Batch Normalization for Deep Neural Networks | Machine Learning Mastery*. URL: https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/.

[21] *Overfitting | Wikipedia*. URL: https://en.wikipedia.org/w/index.php?title=Overfitting&oldid=1041995989.

[22] *Early stopping | Wikipedia*. URL: https://en.wikipedia.org/wiki/Early_stopping.

[23] *Data augmentation | Wikipedia*. URL: https://en.wikipedia.org/wiki/Data_augmentation.

[24] *This Thing Called Weight Decay | Towards Data Science*. URL: https://towardsdatascience.com/this-thing-called-weight-decay-a7cd4bcfccab.

[26] *Convolutional neural network | Wikipedia*. URL: https://en.wikipedia.org/w/index.php?title=Convolutional_neural_network&oldid=1042909201.

[27] *A Comprehensive Guide to Convolutional Neural Networks | Towards Data Science*. URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[28] *CS231n Convolutional Neural Networks for Visual Recognition | Stanford Github*. URL: https://cs231n.github.io/convolutional-networks/.

[29] *A Comprehensive Introduction to Different Types of Convolutions in Deep Learning | Towards Data Science*. URL: https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215.

[30] *How Do Convolutional Layers Work in Deep Learning Neural Networks? | Machibe Learning Mastery*. URL: https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/.

[31] *A Gentle Introduction to Pooling Layers for Convolutional Neural Networks | Towards Data Science*. URL: https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/.

[39] *Torch*. URL: http://torch.ch/.

[40] *Natural Language Processing | Wikipedia*. URL: https://en.wikipedia.org/wiki/Natural_language_processing.

[41] *Single instruction, multiple data (SIMD) | Wikipedia*. URL: https://en.wikipedia.org/wiki/SIMD.

[42] *Tools Overview | Vitis-AI*. URL: https://www.xilinx.com/html_docs/vitis_ai/1_3/tools_overview.html.

[43] *Quantizer Flow | Vitis-AI*. concept. URL: https://www.xilinx.com/html_docs/vitis_ai/1_3/eku1570695929094.html.

[44] *Compiling the Model | Vitis-AI*. URL: https://www.xilinx.com/html_docs/vitis_ai/1_3/compiling_model.html#ztl1570696058091.

[45] *Intel Core i5-8600K Processor Specifications*. URL: https://ark.intel.com/content/www/us/en/ark/products/126685/intel-core-i58600k-processor-9m-cache-up-to-4-30-ghz.html.

[46] *Streaming SIMD Extensions 4 (SSE4) | Wikipedia*. URL: https://en.wikipedia.org/wiki/SSE4.

[47] *NVIDIA RTX-2060 Super*. URL: https://www.nvidia.com/en-eu/geforce/graphics-cards/rtx-2060-super/.

[48] *Zynq UltraScale+ MPSoC Data Sheet*. URL: https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf.

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

[50] *Foundation of Research and Technology Hellas (FORTH)*. URL: https://www.forth.gr/.

[53] *Datasets & DataLoaders | PyTorch Tutorials*. URL: https://pytorch.org/tutorials/beginner/basics/data_tutorial.html.

[54] *Pytorch Data | PyTorch documentation*. URL: https://pytorch.org/docs/stable/data.html.

[55] *Torchvision Datasets | Torchvision documentation*. URL: https://pytorch.org/vision/stable/datasets.html.

[56] *Pytorch Optimizers | PyTorch documentation*. URL: https://pytorch.org/docs/stable/optim.html.

[57] *Pytorch Schedulers | | PyTorch documentation*. URL: https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate.

[58] *Pytorch Loss Functions | PyTorch documentation*. URL: https://pytorch.org/docs/stable/nn.html#loss-functions.

[60] *Quantization | PyTorch documentation*. URL: https://pytorch.org/docs/stable/quantization.html.

[61] *YAML | Wikipedia*. URL: https://en.wikipedia.org/w/index.php?title=YAML&oldid=1042224518.

[62] *Torchvision Transforms | Torchvision documentation*. URL: https://pytorch.org/vision/stable/transforms.html.

[63] *Use containers to Build, Share and Run your applications*. URL: https://www.docker.com/resources/what-container.

[64] *Building Vitis-AI Docker from Recipe | GitHub*. URL: https://github.com/Xilinx/Vitis-AI/tree/1.3.2#building-docker-from-recipe.

[65] *Vitis AI Quantizer for Pytorch | GitHub*. URL: https://github.com/Xilinx/Vitis-AI/tree/1.3.2/tools/Vitis-AI-Quantizer/vai_q_pytorch.

[66] *Vitis AI, Finetune Quantized Model | GitHub*. URL: https://github.com/Xilinx/Vitis-AI/tree/1.3.2/tools/Vitis-AI-Quantizer/vai_q_pytorch#finetune-quantized-model.

[67] *Overview | Zynq DPU*. URL: https://www.xilinx.com/html_docs/vitis_ai/1_3/dpu_over.html.

[68] *Product Specification | Zynq DPU*. URL: https://www.xilinx.com/html_docs/vitis_ai/1_3/prod_spec.html.

[69] *A Basic Introduction to Separable Convolutions | Towards Data Science*. URL: https://towardsdatascience.com/a-basic-introduction-to-separable-convolutions-b99ec3102728.

[70] *Vitis-AI User Guide*. URL: https://www.xilinx.com/html_docs/vitis_ai/1_3/zmw1606771874842.html.

[71] *Python Threading*. URL: https://docs.python.org/3/library/threading.html.

[72] *Xilinx PetaLinux*. URL: https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html.

[73] *Secure Shell Protocol | Wikipedia*. URL: https://en.wikipedia.org/wiki/Secure_Shell.

[77] *Coral AI*. URL: https://coral.ai/.