FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Low-Precision Neural Network Training

Mariana Solange Monteiro Rocha



Master in Informatics and Computing Engineering

Supervisor: Prof. Ivone De Fátima Da Cruz Amorim

Second Supervisor: Prof. Eva Catarina Gomes Maia

October 21, 2026

Low-Precision Neural Network Training

Mariana Solange Monteiro Rocha

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

President: Prof. Sagar Pratapsi
Examiner: Prof. Jana Medková
Supervisor: Prof. Ivone Amorim

October 21, 2026

Abstract

Reducing numerical precision is a promising approach to lowering the computational and memory requirements of neural network training. This dissertation investigates low-precision training through a structured literature review, followed by an experimental comparison of selected frameworks with publicly available implementations identified during the review.

The review analyzes thirteen studies, examining the adaptations introduced to conventional neural network training, numerical representations and precision configurations used across different stages of the training process. It further analyzes the predictive performance, computational efficiency and resource usage, identifying the main trade-offs and evaluation differences.

One of the main gaps identified in the review is the lack of a consistent experimental setting for comparing low-precision training approaches. Existing studies use different datasets, network architectures, hardware, training configurations and efficiency metrics, making direct comparison difficult. To address this, the experimental study developed in this dissertation compares *NITI*, *NITRO-D* and *PocketNN* with an FP32 baseline under partially harmonized conditions, using common datasets, architectures and evaluation metrics whenever possible. The experiments include multilayer perceptrons on MNIST and CIFAR-10 datasets and convolutional neural networks on CIFAR-10, considering predictive performance, execution time, throughput, memory usage and, for the convolutional experiments, device-level GPU energy.

Results show that the evaluated low-precision approaches can maintain predictive performance close to FP32 in some configurations, but do not provide consistent reductions in execution time or memory usage. Across the final comparisons, the FP32 baseline achieved the highest mean final test accuracy and the shortest execution time. In the common-architecture convolutional comparison, FP32, *NITI* and *NITRO-D* achieved mean final test accuracies of 85.843%, 84.391% and 83.141%, respectively, while *NITI* and *NITRO-D* required approximately 1.92 and 6.71 times the FP32 execution time. Memory usage varied according to the framework, dataset and type of memory considered.

Overall, reducing numerical precision alone does not guarantee lower practical resource requirements. The observed trade-offs depend on the complete training approach and experimental setting, including the learning algorithm, software implementation, training configuration and hardware used. This dissertation therefore provides a comparative view of existing low-precision training approaches and experimental evidence on the capabilities and limitations of selected publicly available frameworks, while identifying directions for further research.

Keywords: Low-Precision Training; Neural Network Quantization; Integer-Only Training; Computational Efficiency; Resource Usage

Resumo

A redução da precisão numérica é uma abordagem promissora para diminuir os custos computacionais e de memória associados ao treino de redes neuronais. Esta dissertação investiga o treino em baixa precisão através de uma revisão estruturada da literatura, seguida de uma comparação experimental de abordagens identificadas durante a revisão e com código-fonte publicamente disponível.

A revisão analisa treze estudos, focando-se nas mudanças introduzidas no treino convencional de redes neuronais, bem como nas representações numéricas e configurações de precisão utilizadas nas diferentes etapas de treino. Analisa ainda o desempenho preditivo, a eficiência computacional e a utilização de recursos documentados nos estudos, identificando as principais vantagens e limitações das abordagens, bem como as diferenças na forma como estes benefícios são avaliados.

Uma das principais limitações identificadas é a falta de um contexto experimental consistente para comparar diferentes abordagens de treino em baixa precisão. Os estudos existentes usam diferentes conjuntos de dados, arquiteturas, *hardware*, configurações de treino e métricas de eficiência, dificultando a comparação direta entre resultados. Para responder a esta limitação, o estudo experimental compara as abordagens *NITI*, *NITRO-D* e *PocketNN* com uma referência de treino convencional em vírgula-flutuante (FP32), sob condições experimentais parcialmente harmonizadas, utilizando, sempre que possível, os mesmos conjuntos de dados, arquiteturas e métricas de avaliação. As experiências incluem *perceptrons* multicamada nos conjuntos de dados MNIST e CIFAR-10 e redes neuronais convolucionais no CIFAR-10. A comparação considera o desempenho preditivo, o tempo de execução, a taxa de processamento, a utilização de memória e, nas experiências com redes convolucionais, o consumo energético da GPU ao nível do dispositivo.

Os resultados mostram que as abordagens de baixa precisão avaliadas são capazes de manter um desempenho preditivo próximo do obtido com FP32 em algumas configurações, mas não apresentam reduções consistentes no tempo de execução nem na utilização de memória. Nas experiências finais, a referência FP32 obteve a maior exatidão média final no conjunto de teste e o menor tempo de execução.

De forma geral, os resultados obtidos mostram que a redução da precisão numérica, por si só, não garante menor utilização de recursos na prática. As vantagens e desvantagens observadas dependem da abordagem de treino como um todo e do contexto experimental, incluindo o algoritmo de aprendizagem, a implementação do mesmo, a configuração do treino e do *hardware* utilizado. Assim, esta dissertação apresenta uma visão comparativa das abordagens existentes para treino em baixa precisão e fornece evidência experimental sobre as capacidades e limitações das implementações selecionadas, identificando também possíveis direções para trabalho futuro.

Palavras-chave: Treino em Baixa Precisão; Quantização de Redes Neuronais; Treino em Inteiros; Eficiência Computacional; Utilização de Recursos

The United Nations Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework for achieving a better and more sustainable future. They comprise 17 goals (https://sdgs.un.org/goals) addressing social, economic and environmental challenges.

This dissertation focuses on investigating the practical resource requirements of low-precision neural network training and providing evidence to inform the development and selection of more resource-efficient implementations. For this reason, it contributes to **SDG 9:** Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation.

The connection to target 9.4 is established through its focus on resource-use efficiency. The experiments examine if reduced numerical precision results in lower memory requirements and shorter execution times while considering predictive performance. The convolutional experiments additionally assess estimated GPU energy consumption.

This dissertation also aligns with the scientific research component of target 9.5. The structured literature review, preliminary validation against published results, implementation adaptations and comparative experiments improve understanding of existing low-precision training approaches. By documenting their practical benefits and limitations, this work provides evidence for further research into efficient training methods.

SDG	Target	Contribution	Evidence and Metrics
9	9.4	Provides experimental evidence on the resource requirements of low-precision training, which can inform more efficient use of computing infrastructure.	Peak process RAM and GPU-memory; total execution time; training throughput; device-level GPU-energy for the CNN experiments; comparison with FP32 alongside final test accuracy.
	9.5	Contributes to research on low-precision training through a literature review, reproduction of published results, and evaluation of the benefits and limitations of available implementations.	Reproduction of published results for *NITI*, *NITRO-D* and *PocketNN*; partially harmonized evaluation against an FP32 baseline; identification of discrepancies between theoretical/reported efficiency gains and observed implementation-level behavior.

Acknowledgments

The development of this work, similarly to most things in life, had its ups and downs. I would like to express my gratitude towards all the people that accompanied and supported me through this, making the downs more manageable and celebrating the ups with me.

First, I would like to thank my supervisors, Professor Ivone Amorim and Professor Eva Maia, as well as researcher Ivan Silva, for their continuous guidance, support and insights throughout the development of this dissertation. Your guidance helped me keep the work focused and your support helped me bring it to completion.

To my friends, thank you for your support and encouragement that never failed to motivate me. I am very lucky to have friends like you. Some close, some separated by seas – but you always found ways to be there for me.

I also need to thank my high school teacher, Ângela Coimbra, for her encouragement and care during the years she taught me and for encouraging me to pursue this degree. I will always remember you with great fondness.

To my boyfriend, Miguel, thank you for never doubting me and for the help and affection you have given me throughout this journey. You have been one of my biggest sources of support and I am very lucky to have you.

To my sister, Inês, thank you for being my best friend and always giving me strength. To my mother, thank you for all the effort you have put into supporting my education and for believing in me. To my grandfather Joaquim, my grandmother Júlia and my aunt Bela, thank you for your constant support and for teaching me to face life with courage. To my father, thank you for teaching me resilience and the importance of persevering through life's challenges.

To all of you, my deepest gratitude. Your support was so much important to me and has helped me bring this chapter of my life to an end.

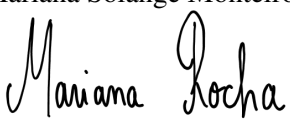
Mariana Rocha

Declaração de Honra

Declaro que a presente dissertação é de minha autoria e não foi previamente apresentada e avaliada nesta ou em outra instituição. As referências a outros autores (afirmações, ideias, pensamentos), assim como a utilização de trabalhos publicados respeitam escrupulosamente as regras da atribuição, e encontram-se devidamente indicadas no texto e nas referências bibliográficas, de acordo com as normas de referenciação. Tenho consciência de que a prática de plágio ou de autoplágio constituem ilícitos académicos, conforme estabelecido no Código de Ética da U.Porto.

Declaro, ainda, que utilizei ferramentas de Inteligência Artificial para a realização de partes da presente dissertação, nomeadamente para melhoramento de texto e como ferramenta auxiliar no processo de pesquisa e de *debug* do código produzido. Os resultados provenientes deste uso foram objeto de cuidada verificação para deteção de erros, imprecisões, atribuições infundadas ou outras formas de enviesamento de conteúdos e argumentos, bem como de determinação de autoria.

Mariana Solange Monteiro Rocha



"Caminante son tus huellas el camino y nada más
Caminante, no hay camino, se hace camino al andar."

Antonio Machado Ruiz

Contents

1 Introduction **1**
- 1.1 Context and Motivation . . . 1
- 1.2 Problem Statement . . . 2
- 1.3 Research Questions and Objectives . . . 3
- 1.4 Main Contributions . . . 4
- 1.5 Dissertation Structure . . . 4

2 Preliminaries **6**
- 2.1 Artificial Neural Networks . . . 6
  - 2.1.1 Fully Connected Neural Networks . . . 7
  - 2.1.2 Convolutional Neural Networks . . . 7
- 2.2 Batch Normalization . . . 8
- 2.3 Neural Network Training Algorithms . . . 9
  - 2.3.1 Backpropagation . . . 10
  - 2.3.2 Direct Feedback Alignment . . . 10
  - 2.3.3 Local Error Signals . . . 11
  - 2.3.4 Forward-Forward . . . 12
- 2.4 Quantization Methods . . . 13
  - 2.4.1 Post-Training Quantization . . . 14
  - 2.4.2 Quantization-Aware Training . . . 15
  - 2.4.3 Low-Precision Training . . . 16

3 Literature Review **18**
- 3.1 Research Methodology . . . 18
  - 3.1.1 Identification . . . 19
  - 3.1.2 Screening . . . 21
  - 3.1.3 Full-Text Assessment and Inclusion . . . 22
- 3.2 Review Findings . . . 23
  - 3.2.1 Backpropagation Adaptation Paradigms . . . 24
  - 3.2.2 Alternative Training Paradigms . . . 29
  - 3.2.3 Comparative Analysis of Reviewed Frameworks . . . 33

4 Experimental Methodology **42**
- 4.1 Problem Statement . . . 42
- 4.2 Experimental Design . . . 43
  - 4.2.1 Framework Selection and Preliminary Validation . . . 43
  - 4.2.2 Datasets and Data Splits . . . 45
  - 4.2.3 Framework Adaptations and Comparison Scenarios . . . 46

4.2.4 Hyperparameter Optimization . . . 47
4.3 Evaluation Protocol . . . 52
4.3.1 Predictive Performance and Execution Metrics . . . 52
4.3.2 Resource and Energy Measurements . . . 53
4.3.3 Experimental Environments and Methodological Limitations . . . 54

5 Results and Analysis **56**
5.1 MLP Experiments . . . 56
5.1.1 Common-Architecture Comparison . . . 56
5.1.2 Individually Tuned Comparison . . . 62
5.2 CNN Experiments . . . 64
5.2.1 Predictive Performance . . . 64
5.2.2 Execution Time and Resource Usage . . . 65
5.2.3 Accuracy-Efficiency Tradeoffs . . . 66
5.3 Discussion . . . 67
5.3.1 Predictive Performance and Configuration Effects . . . 67
5.3.2 Execution Efficiency and Resource Trade-offs . . . 68
5.3.3 Relationship with the Literature Review . . . 70
5.3.4 Answer to RQ3 . . . 70

6 Conclusions **72**
6.1 Summary and Achievement of Objectives . . . 72
6.2 Limitations . . . 74
6.3 Future Work . . . 75

A Appendix **81**
A.1 Database Search Queries . . . 81
A.2 Reported Accuracy of Reviewed Approaches . . . 84
A.3 Hyperparameter Optimization Details . . . 86

List of Figures

2.1 Example of a fully connected neural network with two hidden layers. 8
2.2 Comparison of the feedback paths used by BP, FA and DFA. Adapted from [22]. . 11
2.3 Overview of a typical Quantization-Aware Training workflow. 16

3.1 PRISMA-informed study identification and selection process 19

4.1 Example images from the CIFAR-10 dataset (each column corresponds to one of the ten object classes). 45
4.2 Example images from the MNIST dataset. 45
4.3 Optimization history of the refined *PocketNN* CIFAR-10 hyperparameter search. The y-axis represents accuracy and the x-axis represents the trial number. 49

5.1 Training and test accuracy evolution over epochs on MNIST. 57
5.2 Training and test accuracy evolution over epochs on CIFAR-10. 58
5.3 Training and test accuracy evolution over epochs on CIFAR-10 for the VGG-Small-8 CNN. 64

List of Tables

3.1 Search query for low-precision neural network training 20
3.2 Search fields for the query across the selected databases 20
3.3 Number of records retrieved from each database 21
3.4 Selected Inclusion and Exclusion Criteria 21
3.5 Comparison of numerical formats used by low-precision training frameworks. . . 37
3.6 Comparison of training adaptations used by the reviewed low-precision frameworks. 38
3.7 Top-1 classification accuracy and reported efficiency gains relative to full-precision training, unless otherwise noted. 39
3.8 ImageNet top-1 and top-5 classification accuracy and reported efficiency gains relative to full-precision training. 41

4.1 Comparison between the accuracy reported in the original works and the accuracy reproduced using the released implementations. 44
4.2 Convolutional architecture configurations. 47
4.3 Completed trials, pruning rates and best validation accuracy by number of layers. 48
4.4 Best validation accuracy obtained during hyperparameter optimization for the common-architecture MLP experiments. 50
4.5 Selected architectures and best validation accuracy by dataset and framework. . . 51
4.6 Best validation accuracy obtained during hyperparameter optimization for the common-architecture CNN experiments. 51
4.7 Experimental comparisons performed across datasets, architectures and frameworks. 52
4.8 Predictive performance and execution time metrics used in the experiments. . . . 53
4.9 Computational resource utilization metrics used in the evaluation. 54

5.1 Final and maximum observed test accuracy by framework for the common-architecture MLP on MNIST. Values are reported as mean ± standard deviation, rounded to three decimal places. 57
5.2 Final and maximum observed test accuracy by framework for the common-architecture MLP on CIFAR-10. Values are reported as mean ± standard deviation, rounded to three decimal places. 58
5.3 Total execution time, mean epoch time and throughput for the common-architecture MLP on MNIST. Values for repeated runs are reported as mean ± standard deviation. 59
5.4 Total execution time, mean epoch time and throughput for the common-architecture MLP on CIFAR-10. Values for repeated runs are reported as mean ± standard deviation. 59
5.5 Resource utilization across implementations for the common-architecture MLP on MNIST. 60

5.6 Resource utilization across implementations for the common-architecture MLP on CIFAR-10. 61
5.7 Final accuracy and resource ratios relative to FP32 for the common-architecture MLP on MNIST. 62
5.8 Final accuracy and resource ratios relative to FP32 for the common-architecture MLP on CIFAR-10. 62
5.9 Final accuracy and computational results for the individually selected MLP architectures and hyperparameters on MNIST. 63
5.10 Final accuracy and computational results for the individually selected MLP architectures and hyperparameters on CIFAR-10. 64
5.11 Final and maximum observed test accuracy by framework for VGG-Small-8 on CIFAR-10. Values are reported as mean ± standard deviation, rounded to three decimal places. 65
5.12 Total execution time, mean epoch time and throughput for VGG-Small-8 on CIFAR-10. Values are reported as mean ± standard deviation. 65
5.13 Resource utilization across implementations for VGG-Small-8 on CIFAR-10. . . 66
5.14 Final accuracy and computational ratios relative to FP32 for VGG-Small-8 on CIFAR-10. 67

A.1 Reported top-1 and top-5 classification accuracy (%) of the reviewed low-precision training approaches. 84
A.2 Hyperparameter search space used in the initial PocketNN CIFAR-10 search. . . 86
A.3 Hyperparameter search space used in the refined PocketNN CIFAR-10 search. . . 86
A.4 Hyperparameter search space used in the MLP NITRO-D search. 86
A.5 Hyperparameter search space used in the MLP NITI search. 87
A.6 Hyperparameter search space used in the MLP FP32 search. 87
A.7 Hyperparameter configurations selected for the common-architecture MLP experiments. 88
A.8 Hyperparameter configurations selected for the MLP architecture-search experiments. 88
A.9 Hyperparameter search space used for the NITRO-D CNN experiment. 89
A.10 Hyperparameter search space used for the NITI CNN experiment. 89
A.11 Hyperparameter search space used for the FP32 CNN experiment. 89
A.12 Hyperparameter configurations selected for the VGG-Small-8 CNN experiments. 89

Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
BN	Batch Normalization
BP	Backpropagation
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DFA	Direct Feedback Alignment
DFP	Dynamic Fixed Point
DNN	Deep Neural Network
FA	Feedback Alignment
FCNN	Fully Connected Neural Network
FP8	8-bit Floating Point
FP16	16-bit Floating Point
FP32	32-bit Floating Point
FPGA	Field-Programmable Gate Array
FPU	Floating-Point Unit
GPU	Graphics Processing Unit
INT8	8-bit Integer
INT16	16-bit Integer
INT32	32-bit Integer
IoT	Internet of Things
LES	Local Error Signals
MAC	Multiply-Accumulate
MLP	Multi-Layer Perceptron
NVML	NVIDIA Management Library
P95	95th Percentile
PRISMA	Preferred Reporting Items for Systematic Reviews and Meta-Analyses
PTQ	Post-Training Quantization
QAT	Quantization-Aware Training
RAM	Random Access Memory
RAPL	Running Average Power Limit
RQ	Research Question
RSS	Resident Set Size
SGD	Stochastic Gradient Descent
STE	Straight-Through Estimator
TPE	Tree-structured Parzen Estimator
VRAM	Video Random Access Memory

Chapter 1

Introduction

This chapter provides an overview of the context and motivation for this dissertation and defines the research problem addressed. It also presents the research objectives and research questions that guide the literature review and the subsequent experimental investigation.

1.1 Context and Motivation

Deep neural networks (DNNs) have achieved strong performance in domains such as computer vision, speech recognition and natural language processing. Their use has expanded across a wide range of application areas, including healthcare, finance, manufacturing and transportation [33].

Modern DNNs often contain millions or even billions of trainable parameters and their execution can require substantial computational resources. During training, these costs are compounded by the repeated forward and backward computations required to optimize the model parameters. This results in substantial memory requirements, long computation times and considerable energy consumption [3, 37].

The increasing computational complexity of modern DNNs creates challenges throughout their lifecycle. Training large models requires substantial computational resources, resulting in an increasing demand for energy and computing infrastructure. At the same time, there is a growing interest in executing and, in some cases, training DNNs directly on resource-constrained platforms, including embedded systems, Internet of Things (IoT) devices and edge devices, to enable low-latency processing, reduced communication costs and improved privacy. However, the limited memory, processing power, storage and energy available on these devices often make such applications challenging [12]. Furthermore, according to a 2026 report from the United Nations University [1], global data centers consumed approximately 448 TWh of electricity and had a water footprint of 4.5 trillion liters in 2025. Artificial Intelligence (AI) accounted for around 20% of this electricity demand, but its share is projected to double by 2030, reaching approximately 40% of total data center electricity consumption. These trends illustrate the broader environmental implications associated with the increasing computational demands of AI systems. Beyond electricity consumption, the rapid expansion of AI also increases the demand for critical minerals

that are required to produce specialized hardware and contributes to the growing generation of electronic waste, raising additional environmental and societal concerns.

These challenges motivate the search for techniques that improve the computational efficiency of DNNs, reducing their resource requirements while enabling more efficient training and deployment on resource-constrained devices. Several approaches have been proposed to address these challenges, including model compression techniques such as pruning, which removes redundant parameters; knowledge distillation, which transfers knowledge from a larger model to a smaller one; and quantization, which reduces the numerical precision of model parameters and/or computations [5].

Initially, quantization research focused primarily on inference. Approaches such as Post-Training Quantization (PTQ) and conventional Quantization-Aware Training (QAT) were primarily concerned with producing models suitable for reduced-precision inference, while training remained substantially dependent on higher-precision representations and computations [32]. Early attempts to extend quantization into the training process, including Binarized Neural Networks and DoReFa-Net, still retained full-precision parameter copies during optimization [9, 39]. More recent approaches have progressively extended reduced-precision computation to additional components of the training pipeline [4]. Unlike quantization applied exclusively for model deployment, low-precision training requires reduced numerical precision to be accommodated throughout the training process, rather than only in the final deployed model. Despite this, recent advances have demonstrated that training can be performed using low-precision or even integer-only arithmetic for substantial portions of the training pipeline, while maintaining competitive predictive performance in some settings [4, 25, 29, 30, 34].

1.2 Problem Statement

Training with reduced precision requires numerical representations and training procedures that account for the limited range and precision of intermediate values, gradients and parameter updates. In particular, gradients and intermediate values must fit within the limited numerical range of low-bitwidth representations, while small parameter updates must still be represented with sufficient precision. Rounding and quantization errors can also accumulate throughout training and affect convergence. As a result, low-precision training often requires modifications such as scaling and rounding mechanisms, wider representations for selected operations, modifications to optimization and normalization procedures or changes to the conventional backpropagation pipeline.

A growing number of frameworks have been proposed for low-precision training, but they differ in their learning algorithms, numerical representations and reported accuracy and efficiency. Direct comparison is difficult as studies use different architectures, datasets, training configurations, hardware platforms and evaluation metrics. As a result, reported differences in accuracy and efficiency are difficult to separate from these experimental and implementation choices. This diversity motivates a structured comparative analysis of existing approaches to better understand their trade-offs in predictive performance, execution efficiency and resource usage. Accordingly,

this dissertation focuses on approaches that incorporate reduced numerical precision directly into neural network training, rather than on quantization techniques intended exclusively for inference.

It combines a structured literature review with an experimental comparison of selected publicly available frameworks and an FP32 baseline under more comparable conditions.

1.3 Research Questions and Objectives

Based on the research problem outlined above, the dissertation is guided by the following main research question:

> *What trade-offs arise when neural networks are trained under reduced-precision constraints, particularly in terms of predictive performance, computational efficiency and resource usage?*

To address this broader question, three more specific research questions are considered:

- RQ1: How is reduced numerical precision integrated into the different components of neural network training and what adaptations are used to support training under these constraints?
- RQ2: What trade-offs between predictive performance, computational efficiency and resource usage are reported for existing low-precision training approaches?
- RQ3: When selected low-precision training frameworks are evaluated under more comparable experimental conditions, how do their predictive performance, execution time and resource usage compare with each other and with full-precision training?

The general objective of this dissertation is, therefore, to characterize low-precision neural network training approaches and to evaluate the trade-offs associated with their use, considering both how reduced numerical precision is incorporated into the training process and its implications for predictive performance, computational efficiency and resource usage.

More specifically, the work aims to:

- examine the numerical representations and precision configurations adopted across the different stages of neural network training;
- analyze the adaptations introduced to conventional training procedures in order to operate under reduced-precision constraints;
- assess the trade-offs in predictive performance, computational efficiency and resource usage reported in the existing literature;
- identify representative low-precision training frameworks that allow a meaningful experimental comparison;

- experimentally compare the selected frameworks under more comparable conditions, considering their predictive performance, execution efficiency and resource usage, as well as their behaviour relative to full-precision training.

Together, these objectives connect the literature review with the experimental study by first characterizing existing low-precision training approaches and then evaluating a selected subset under more comparable conditions.

1.4 Main Contributions

The main contributions of this dissertation are as follows:

- a structured review of low-precision neural network training approaches, comparing their numerical representations, training adaptations and learning mechanisms;
- an analysis of the predictive performance and efficiency results reported in the literature, showing how the reported benefits vary with the numerical representation, training adaptations, learning algorithm, implementation, hardware platform used and evaluation method and identifying the limitations these differences create for direct comparison;
- the adaptation of three publicly available low-precision training frameworks to a partially harmonized experimental methodology, including common-architecture and framework-specific comparisons, hyperparameter optimization and measurements of predictive performance, execution time and resource usage;
- an experimental comparison of the selected frameworks with an FP32 baseline under these partially harmonized conditions, providing implementation-level evidence on the trade-offs between predictive performance, execution time, memory usage and, where measured, device-level GPU energy.

1.5 Dissertation Structure

The remainder of this dissertation is organized as follows.

Chapter 2 introduces the theoretical background required to understand low-precision neural network training. It presents the main numerical representations used in neural network computation, reviews quantization concepts and techniques and discusses the challenges associated with training under reduced-precision constraints.

Chapter 3 presents the literature review on low-precision neural network training. It describes the study identification, screening and selection process and analyzes the selected approaches in terms of their numerical representations, training adaptations, reported predictive performance and computational characteristics.

Chapter 4 describes the experimental methodology adopted to compare selected low-precision training frameworks. It presents the frameworks considered, datasets, neural network architectures, training configurations, hyperparameter optimization procedure, evaluation metrics and experimental setup.

Chapter 5 presents and discusses the experimental results. The selected low-precision training frameworks are compared with each other and with a full-precision baseline in terms of predictive performance, execution efficiency and resource usage. The results are also interpreted in relation to the findings identified in the literature review.

Finally, Chapter 6 summarizes the main findings of the dissertation, discusses its limitations and presents directions for future work.

Chapter 2

Preliminaries

This chapter presents the fundamental concepts needed to understand the remainder of this dissertation. It first introduces the neural network architectures considered in this work, followed by the main training algorithms relevant to the reviewed low-precision approaches. Finally, it presents different quantization methods, ranging from inference-oriented quantization to approaches that apply reduced numerical precision directly to the training process.

2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are computational models originally inspired by the structure and functioning of the human brain. Their origins date back to 1943, when Warren McCulloch and Walter Pitts introduced an early conceptual model of an artificial neuron in their paper *A Logical Calculus of the Ideas Immanent in Nervous Activity* [21]. This work contributed to subsequent developments in neural network research, including Frank Rosenblatt's Perceptron [27], introduced in 1958 as an early trainable neural network model [13].

Two common classes of ANN architectures are feedforward and recurrent neural networks. During the forward pass, feedforward neural networks propagate information from the input layer towards the output layer without forming cycles or feedback connections. These architectures are commonly used for tasks such as image classification, regression and tabular data analysis. In contrast, recurrent neural networks contain feedback connections that allow information to persist across time steps, making them suitable for processing sequential data such as text, speech and time series. Since the training algorithms discussed in this dissertation are primarily evaluated on feedforward architectures, the following sections focus on this class of ANNs.

A basic feedforward ANN typically consists of three types of layers: an input layer, one or more hidden layers and an output layer. Each layer is composed of interconnected nodes commonly referred to as neurons. Connections between neurons are established using learnable parameters called weights, which determine the strength of the established link. Each neuron in a hidden or output layer typically computes an output – its activation – based on the inputs received, the bias and the chosen activation function, with the output layer producing the network prediction.

For example, in a 10-class classification problem, the output layer typically contains ten neurons – one for each class. The neuron with the highest activation or output score, depending on the output activation function, corresponds to the predicted class.

2.1.1 Fully Connected Neural Networks

Fully Connected Neural Networks (FCNNs), also known as Multi-Layer Perceptrons (MLPs), are one of the simplest types of feedforward ANN architectures. In this architecture, every neuron in each non-input layer is connected to all neurons in the preceding layer through weights. Layers in which every neuron is connected to all neurons in the preceding layer are referred to as fully connected, or dense, layers.

During the forward pass, each neuron first computes its pre-activation as the weighted sum of the activations from the previous layer plus a bias term. An activation function is then applied to obtain the neuron's activation. Figure 2.1 illustrates a fully connected neural network with two hidden layers.

The pre-activation $z_i^{(l)}$ of a neuron i in layer l can be expressed as

$$z_i^{(l)} = \sum_j w_{ij}^{(l)} a_j^{(l-1)} + b_i^{(l)}$$

where:

- $a_j^{(l-1)}$ is the activation of neuron j in the previous layer $l-1$;
- $w_{ij}^{(l)}$ is the weight connecting neuron j in layer $l-1$ with neuron i in layer l;
- $b_i^{(l)}$ is the bias associated with neuron i.

The output of a neuron i in layer l, referred to as its activation $a_i^{(l)}$, is obtained by applying an activation function to the pre-activation value:

$$a_i^{(l)} = \theta\left(z_i^{(l)}\right)$$

where $\theta(x)$ represents the activation function. When nonlinear activation functions are used, they allow the network to model relationships that cannot be captured by a purely linear model.

During training, the weights and biases are iteratively updated according to an optimization procedure that seeks to minimize a loss function, allowing the network to learn the relationship between inputs and outputs.

2.1.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a type of ANN well suited to processing data with a grid-like structure, such as images and videos. Unlike FCNNs, where each neuron is connected to every neuron in the previous layer, CNNs exploit local spatial relationships between neighboring

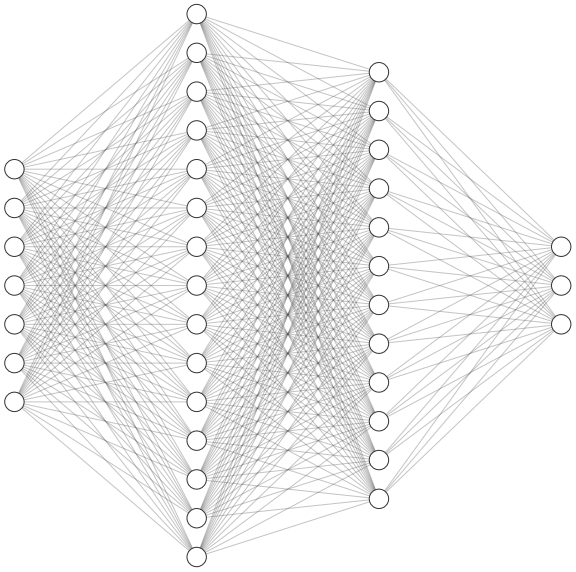


Figure 2.1: Example of a fully connected neural network with two hidden layers.

input values. This makes them particularly well suited for computer vision tasks, including image classification, segmentation and object detection.

The main building blocks of a CNN are convolutional layers. These layers apply small learnable filters, also referred to as kernels, across the input. Each filter operates only on a local region of the input at a time and produces a feature map encoding the response of that filter at different spatial locations. This allows CNNs to extract spatial features from images while typically requiring fewer parameters than an equivalently sized fully connected architecture.

In the first layers of a CNN, filters often learn simple features such as edges and textures. Deeper layers combine these features to learn more complex patterns. Convolutional layers are usually followed by activation functions, such as ReLU and may also be followed by pooling layers, which can reduce the spatial dimensions of the feature maps. For example, max pooling keeps the largest value from each local region, reducing the amount of information that needs to be processed by subsequent layers.

A typical CNN for image classification may contain several convolutional layers, optionally interleaved with pooling or other downsampling operations, followed by a classification head that produces the final prediction. In image classification tasks, the output layer typically contains one output value for each possible class.

2.2 Batch Normalization

Batch Normalization (BN) is a normalization technique introduced to facilitate the training of deep neural networks [16]. During training, BN normalizes intermediate values using statistics computed over a mini-batch. Given a mini-batch $\mathscr{B}$ containing m values $x_1, \ldots, x_m$, the mini-batch mean and variance are computed as

$$\mu_{\mathscr{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i$$

and

$$\sigma^2_{\mathscr{B}} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathscr{B}})^2.$$

Each value is then normalized according to

$$\hat{x}_i = \frac{x_i - \mu_{\mathscr{B}}}{\sqrt{\sigma^2_{\mathscr{B}} + \varepsilon}},$$

where ε is a small constant introduced for numerical stability. The normalized value is subsequently transformed using two learnable parameters, γ and β:

$$y_i = \gamma\hat{x}_i + \beta.$$

The learnable scale and shift parameters allow the network to adapt the normalized representation during training. By normalizing activations, BN can facilitate optimization and improve training stability [16].

During inference, the statistics of the current mini-batch are generally not used. Instead, estimates of the mean and variance accumulated during training are used to normalize the corresponding values. Therefore, BN involves not only the normalization itself but also the computation and storage of statistical quantities and learnable parameters.

These characteristics make BN particularly relevant in low-precision neural network training. The computation of normalization statistics and operations such as division and square root can be more difficult to implement under strict low-bitwidth or integer-only constraints. Consequently, low-precision training approaches adopt different strategies for handling BN: some retain higher precision for selected operations, others introduce quantized implementations, while some remove BN or replace it with simpler scaling mechanisms [36]. These alternatives are examined in the low-precision training frameworks discussed in Chapter 3.

2.3 Neural Network Training Algorithms

In conventional supervised learning, training an ANN is an iterative optimization process that seeks to minimize a loss function measuring the discrepancy between network predictions and target values. Training with backpropagation generally consists of three main steps: the forward pass, the backward pass and the parameter update.

During the forward pass, input data is propagated through the network, where each layer applies its weights, biases and activation functions to generate a prediction. The loss is then calculated from the network prediction and the corresponding target values. During the backward pass, information from the loss is used to compute gradients or other error signals associated with the

model parameters. Finally, the parameter update step adjusts the weights according to the selected optimization algorithm.

Conventional training generally relies on backpropagation to compute gradients, while alternative learning algorithms can modify or eliminate global backward error propagation. The following sections introduce Backpropagation (BP), Direct Feedback Alignment (DFA), Local Error Signals (LES) and Forward-Forward (FF), as these methods are used by the low-precision training approaches examined later in this dissertation.

2.3.1 Backpropagation

Backpropagation (BP), popularized by Rumelhart, Hinton and Williams [28], is the standard algorithm used to compute gradients during neural network training. BP propagates gradient information from the output layer towards the input layer using the chain rule of calculus. It computes the derivatives of the loss with respect to the network parameters, which are then used by optimization algorithms such as gradient descent to update those parameters.

A key characteristic of BP is its recursive layer-by-layer propagation of gradient information. The gradient associated with an earlier layer depends on quantities propagated from subsequent layers, creating a sequential dependency during the backward pass. This enables efficient computation of the gradients required for optimization and has been fundamental to the widespread use of deep neural networks across tasks such as image classification, speech recognition and natural language processing.

Despite its effectiveness, BP presents some practical limitations. During training, intermediate activations are typically maintained for use in the backward pass, increasing memory requirements, while gradient information must be propagated sequentially through multiple layers, introducing additional computational cost. These characteristics can also complicate low-precision and integer-only training. The limited dynamic range of low-bitwidth representations increases the risk of overflow in intermediate computations, while insufficient numerical resolution can cause small gradients or parameter updates to be lost through quantization [15, 30].

Nevertheless, BP remains the standard reference learning mechanism against which alternative training approaches are commonly compared due to its widespread use and effectiveness in training deep neural networks.

2.3.2 Direct Feedback Alignment

Nøkland introduced Direct Feedback Alignment (DFA) in 2016 as an alternative to BP motivated in part by the biological implausibility of symmetric forward and backward weight transport [22]. The DFA method builds upon the concept of Feedback Alignment (FA), introduced by Lillicrap et al. in 2014 [19]. In FA, error signals are propagated through fixed random feedback matrices rather than through the transposes of the forward weights. The results obtained with FA showed that learning can occur without requiring exact symmetry between the forward and backward weight matrices.

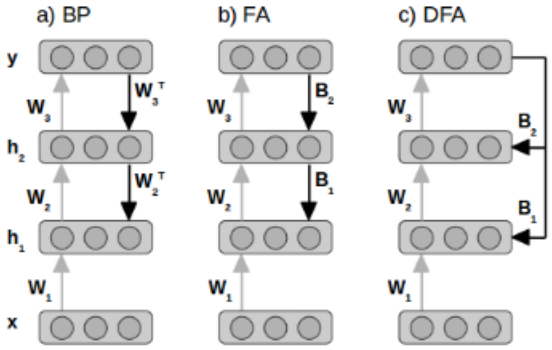


Figure 2.2: Comparison of the feedback paths used by BP, FA and DFA. Adapted from [22].

DFA extends this idea by removing the layer-by-layer propagation of error signals. Instead, each hidden layer receives an error signal derived directly from the output error through a fixed random feedback matrix. Once the output error is available, hidden-layer updates no longer depend on error signals propagated sequentially through deeper layers, increasing the potential for parallel computation during the backward phase. Figure 2.2 illustrates the different feedback paths used by BP, FA and DFA.

While originally proposed as an advance towards a better understanding of the biological mechanisms of error-driven learning in the brain, DFA was later explored in the context of low-precision training because some of its characteristics can be useful when training with integer arithmetic. Implementing standard BP using integer arithmetic can be challenging due to the repeated propagation and multiplication of error signals. By avoiding the recursive application of the chain rule across layers, DFA was exploited by integer-only approaches such as *PocketNN* [29] to reduce the risk of overflow during training.

Nøkland evaluated DFA on MNIST [18], CIFAR-10 and CIFAR-100 [17] using both fully connected and convolutional architectures [22]. The reported results showed relatively small differences from BP for the evaluated fully connected networks; on MNIST, for example, DFA achieved up to 98.98% accuracy and the reported difference in test error relative to BP did not exceed 0.22 percentage points. Larger differences were observed for the evaluated convolutional architectures, where DFA achieved accuracies 4.4 and 7.3 percentage points lower than BP on CIFAR-10 and CIFAR-100, respectively. These results indicate that the relative performance of DFA depends on the network architecture and task.

2.3.3 Local Error Signals

Local Error Signals (LES) [23], proposed by Nøkland and Eidnes, provide another alternative to conventional end-to-end BP by training hidden layers using locally generated error signals. In standard BP, error information is propagated layer by layer from the output to earlier layers, making hidden-layer updates dependent on the completion of the corresponding forward computations

and subsequent backward propagation. This dependency, referred to as backward locking, prevents earlier layers from being updated until the required backward information has propagated through subsequent layers [23].

LES addresses this dependency by associating hidden layers with local learning objectives. In the original formulation, auxiliary single-layer networks are used to generate local error signals, allowing each hidden layer to be optimized without relying on the global error propagated from the network output [23]. The local objective combines supervised information derived from the target labels with a representation-based loss that encourages useful structure in the hidden-layer representations.

In the original LES method, hidden layers are trained using layer-wise local losses. Each hidden layer uses a local objective to generate an error signal for the corresponding part of the network. Instead of propagating the final output error through all previous layers, gradients are computed locally for each hidden layer. Because hidden-layer weights can be updated locally once the computations required by that layer are available, their activations and gradients do not need to be retained for a later end-to-end backward pass. This structure can reduce memory requirements, enable earlier memory reuse and allow layer updates to be performed more independently than in conventional end-to-end BP [23].

The locality of gradient propagation is also relevant to reduced-precision training. By restricting gradient propagation to local blocks, LES can reduce the depth over which low-precision error signals must be propagated, a property exploited by integer-only training frameworks such as *NITRO-D* to reduce the risk of overflow [25].

2.3.4 Forward-Forward

Geoffrey Hinton (2022) [14] proposed the Forward-Forward (FF) training algorithm in the paper *The Forward-Forward Algorithm: Some Preliminary Investigations*. The motivation behind this work was, similarly to DFA and LES, to investigate whether ANNs could be trained without BP, driven by concerns regarding BP, including its biological implausibility and dependence on storing intermediate computations for the backward pass.

As its name suggests, the FF algorithm is based on two types of forward passes rather than a conventional forward and backward pass. Instead of propagating error signals backwards through the network, training consists of performing two forward passes: a *positive* and a *negative pass*. The *positive pass* is performed on *positive examples*, which correspond to valid training samples, while the *negative pass* uses *negative examples* that are created to represent either invalid inputs, in unsupervised learning, or invalid input-label pairs, in supervised learning.

Since this dissertation focuses on supervised learning, only this setting is considered in the remainder of the algorithm description. In a classification task, for example, a *positive example* may be an image paired with its correct label, while a *negative example* may consist of the same image paired with an incorrect label.

During each forward pass, every hidden layer computes a *goodness* value. Although different formulations are possible, the original work primarily defines *goodness* as the sum of the squared activations of the layer after the ReLU activation function is applied but before layer normalization.

Training encourages the *goodness* to exceed a predefined threshold during the *positive pass* and remain below the same threshold during the *negative pass*. Therefore, each layer learns locally how to distinguish compatible input-label pairs from incompatible ones without requiring backward error propagation. Before the activations are passed to the next layer, layer normalization is applied. This removes the magnitude of the activity vector, therefore reducing the information available about the layer's *goodness* score while preserving its direction. Consequently, subsequent layers cannot directly rely on the *goodness* computed by previous layers and must learn their own representations.

The author also proposes an optional procedure for generating hard negatives, that is, incorrect labels that the network is more likely to confuse with the correct class. Before the negative pass for a training sample, the network replaces the specific class label in the input with a neutral label and performs a forward pass to estimate the probability assigned to each class. Incorrect labels can then be sampled according to these probabilities, producing more informative negative examples.

The algorithm was evaluated on MNIST and CIFAR-10. On MNIST, an FCNN trained using FF achieved a test accuracy of 98.64% after 60 epochs. Hinton notes that BP achieved comparable performance after 20 epochs in the reported experimental setting, indicating slower training progress for FF in this comparison. On CIFAR-10, the algorithm was evaluated on a CNN-inspired architecture with local receptive fields but without weight sharing. In this setting, FF achieved 59% accuracy, which was 2 to 4 percentage points lower than BP depending on the number of hidden layers. The reported difference did not increase as deeper networks were evaluated.

The main limitation recognized by Hinton is that FF does not achieve the same performance as BP. In the experiments reported in the original work, FF required more training epochs in some settings and generally achieved lower predictive performance than BP. Hinton further argues that, given the lower training efficiency and predictive performance observed in these experiments, FF is unlikely to replace BP in conventional deep learning applications, while it may still be valuable in biologically inspired models or power-constrained settings. However, training time, computational complexity and energy consumption are not reported to support the latter potential advantage.

Another important limitation is that, due to FF's greedy layer-wise optimization, representations learned by earlier layers are not refined using error information propagated from deeper layers. The absence of conventional backward error propagation also makes FF relevant to reduced-precision training and subsequent work has explored integer implementations of the algorithm [20].

2.4 Quantization Methods

Quantization is the process of converting values represented in high-precision numerical formats into lower-precision representations. Lower-precision formats require less memory and can enable

more efficient computation and data movement, which may lead to faster processing, reduced energy consumption and deployment on resource-constrained devices.

In machine learning, quantization commonly refers to converting floating-point data, typically represented using 32-bit floating-point values, to lower-precision integer formats such as 8-bit integers. However, reduced-precision approaches may also involve lower-precision floating-point formats, such as mapping 32-bit floating-point values to 16-bit floating-point representations. Integer arithmetic can be implemented efficiently on hardware with strong integer support, particularly on hardware where floating-point operations are comparatively costly, making quantization relevant for resource-constrained deployment [32].

The main challenge of quantization is to maintain model accuracy while reducing numerical precision. For some quantization schemes, particularly post-training quantization, calibration is used to determine suitable quantization parameters for the activations. This is done by running the model on a representative subset of the dataset to observe the behaviour of activations and determine appropriate ranges for their quantized representation. While weight ranges can often be computed directly from model parameters, activation ranges depend on runtime behaviour and input data, making calibration useful for determining appropriate activation quantization parameters.

Quantization can be incorporated at different stages of the model lifecycle. Two widely used approaches are Post-Training Quantization, where quantization is applied after training and Quantization-Aware Training, where quantization effects are incorporated during training. Beyond these deployment-oriented approaches, more recent work has extended reduced-precision computation to the training process itself, which is the main focus of this dissertation.

2.4.1 Post-Training Quantization

PTQ is an approach for reducing the numerical precision of a neural network after training has been completed. In PTQ, a model is first trained, normally using floating-point precision and quantization is subsequently applied [32]. Tensors such as weights and, depending on the implementation, activations may be quantized to lower-precision representations, commonly INT8. This reduces model storage requirements and can improve inference speed and memory usage, making deployment more suitable for resource-constrained devices such as smartphones and IoT devices.

PTQ can be divided into static and dynamic quantization. Static PTQ quantizes weights and uses calibration data to determine activation quantization parameters before inference, while dynamic PTQ quantizes weights beforehand and determines activation quantization parameters during inference. During calibration, representative input samples are passed through the trained model so that the observed activation distributions can be used to determine suitable quantization ranges and scaling parameters. The quality of these ranges is important because values outside the selected interval may be clipped, while an excessively wide interval reduces the effective resolution available to represent values within that range.

Quantization can also differ in the granularity at which its parameters are defined. In per-tensor quantization, a single scale is used for an entire tensor, while per-channel quantization assigns separate quantization parameters to individual channels. The latter can better accommodate differences in the numerical ranges of different channels, particularly for weight tensors, at the cost of additional quantization parameters and implementation complexity [32].

Although simple to apply, PTQ can cause noticeable accuracy degradation, especially at very low bitwidths [32]. Because the original training process does not account for the quantization that will later be applied, the resulting model parameters may not be robust to the rounding and clipping errors introduced by quantization. This limitation motivates Quantization-Aware Training, discussed in the following subsection.

2.4.2 Quantization-Aware Training

QAT is used to reduce the accuracy degradation that can result from quantization by incorporating its effects during training [32]. QAT can be applied either when training from scratch or when fine-tuning a pre-trained model. During training, selected weights and activations are subjected to simulated quantization, while the underlying parameters are typically retained in a higher-precision representation for optimization.

This simulated quantization process is commonly referred to as fake-quantization. Fake-quantization simulates the mapping of values to a lower-precision representation through operations such as scaling, clipping and rounding, while the resulting values remain stored and processed using the underlying higher-precision data type. This allows the model to adapt to the numerical effects that will be present in the final quantized representation [32].

During QAT, fake-quantization operations are included in the forward pass for the relevant tensors. The resulting network output is used to compute the loss, after which backpropagation is performed. Quantization operations such as rounding have zero or undefined derivatives at most points, meaning that their derivatives cannot be used directly to propagate useful gradient information during training.

To address this problem, QAT commonly uses a Straight-Through Estimator (STE), which approximates the derivative of the quantization operation during backpropagation, typically by treating it as the identity function within the quantization range. The forward pass therefore incorporates simulated quantization, while the backward pass uses the surrogate gradient to propagate information through the quantization operation, allowing the underlying parameters to be updated in higher precision. During each forward pass, the relevant tensors are fake-quantized according to the target quantization scheme and this process is repeated throughout training.

At the end of training, the tensors required for deployment can be converted to the chosen lower-precision format, producing a final quantized model suitable for inference on resource-constrained devices.

QAT introduces additional training cost compared with PTQ because the model must undergo training or fine-tuning while simulating quantization effects. Therefore, choosing between the two approaches requires evaluating whether the potential accuracy benefits of QAT justify this

additional training cost [32]. Figure 2.3 summarizes the main stages of a typical QAT workflow, from fake-quantization during training to the generation of the final quantized model.

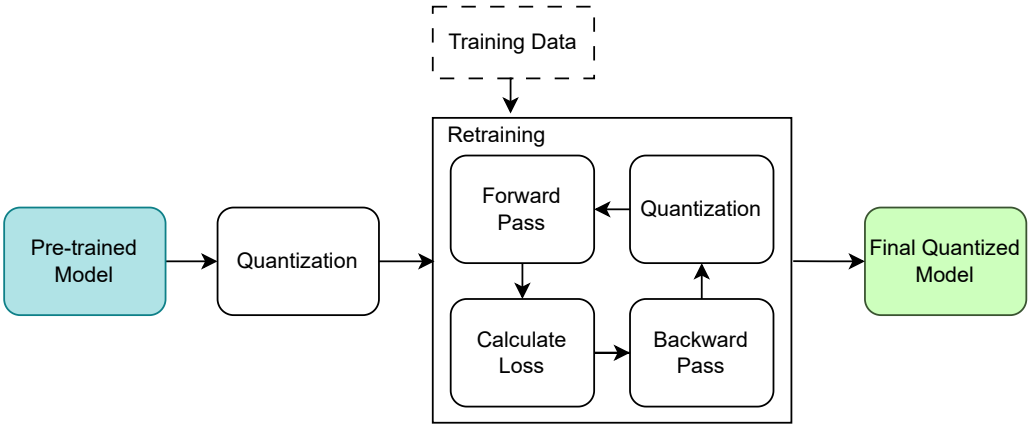


Figure 2.3: Overview of a typical Quantization-Aware Training workflow.

While PTQ applies quantization after training and conventional QAT incorporates quantization effects during training while typically retaining higher-precision parameters and optimization operations, more recent approaches extend reduced-precision representations and computations to the training process itself.

2.4.3 Low-Precision Training

Low-precision training applies reduced numerical precision directly to one or more components of the training process. In this setting, components such as weights, activations, gradients, error signals and parameter updates may be represented or computed using reduced-precision formats during training [4, 30].

This distinction is particularly important in relation to conventional QAT. QAT's quantization effects are typically simulated during the forward pass through fake-quantization, while the underlying parameters and optimization process remain substantially dependent on higher-precision representations. Low-precision training instead seeks to perform a larger portion of the actual training computations using reduced-precision arithmetic, therefore targeting not only the precision of the final model but also the computational and memory requirements of the training process itself.

Extending quantization to training introduces additional numerical challenges. Gradients and parameter updates can have substantially smaller magnitudes than weights and activations, making them more susceptible to rounding, underflow and loss of information. At the same time, intermediate results and accumulated values may require a wider numerical range to prevent overflow. Consequently, low-precision training methods often combine different bitwidths across training components and introduce mechanisms such as scaling, stochastic rounding or wider accumulators to preserve useful numerical information [15, 30].

The degree to which reduced precision is applied varies between approaches. Some methods retain higher precision for selected operations, such as accumulation, normalization or loss computation, while others aim to eliminate floating-point computation from most or all of the training pipeline. In this dissertation, the focus is on approaches that integrate reduced precision directly into training rather than only simulating quantization for inference. In particular, methods that depend on persistent full-precision master or shadow weights throughout training are outside the main scope of the literature review, although isolated higher-precision operations may still be present in otherwise low-precision training pipelines.

Chapter 3

Literature Review

This chapter reviews existing approaches for training neural networks under reduced-precision numerical constraints. It first describes the literature-review methodology and the study identification, screening and inclusion process, which was guided by the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) framework [24] guidelines. It then analyzes the selected approaches in terms of their numerical representations, training adaptations and reported predictive performance and efficiency trade-offs.

3.1 Research Methodology

This section describes the methodology adopted for the literature review, which examines existing research exploring the integration of reduced numerical precision into the training of feedforward neural networks, specifically methods that train from scratch using reduced-precision, including integer, fixed-point and low-bitwidth floating-point representations. Approaches that use reduced precision only during inference, simulate quantization while maintaining full-precision computations or address secondary objectives such as communication or hardware optimization without employing reduced-precision training are outside the scope of the review.

This review aims to provide a comprehensive understanding of existing approaches, the techniques commonly applied and the benefits and limitations of low-precision neural network training methodologies, as well as the challenges associated with their development and applications, while answering the research questions RQ1 and RQ2 formulated in Section 1.3.

The study identification and selection process was informed by the PRISMA guidelines, which provide guidance for transparently reporting the identification, screening and inclusion of studies. The process was organized into three stages: Identification, Screening and Inclusion. The procedures followed in each stage and their application to this review are described below. The study-selection flow is summarized in Figure 3.1 using a PRISMA-informed flow diagram.

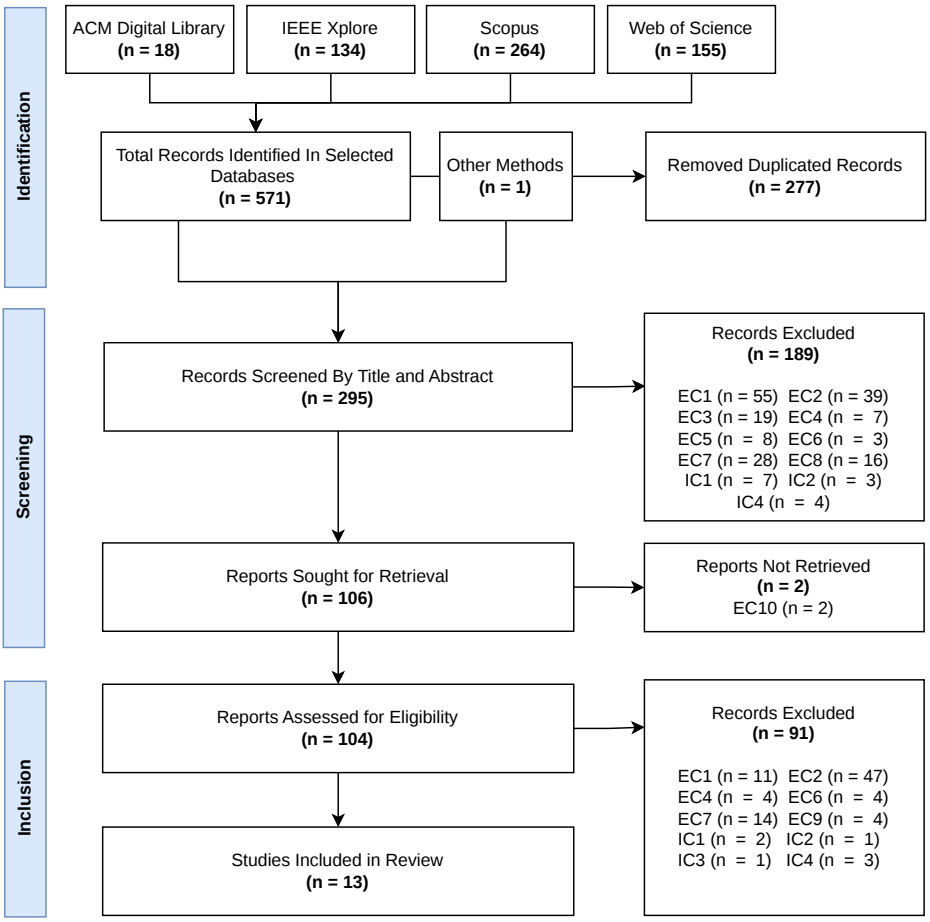


Figure 3.1: PRISMA-informed study identification and selection process

3.1.1 Identification

The Identification stage involved collecting potentially relevant studies from the selected scientific databases and removing duplicate records. For this review, four major scientific databases were selected to provide broad coverage of publications related to low-precision and quantized neural network training: the Association for Computing Machinery (ACM) Digital Library [1], the Institute of Electrical and Electronics Engineers (IEEE) *Xplore* [2], Scopus [3] and Web of Science [4] databases.

[1] https://dl.acm.org/
[2] https://ieeexplore.ieee.org/
[3] https://www.scopus.com/
[4] https://www.webofscience.com/

A search query was developed and iteratively refined to gather relevant studies from these databases. The process of defining the final query involved an initial exploratory literature search, including citation tracking of relevant studies, to identify influential works and terminology used within the field. Multiple query variations were iteratively tested across the selected databases to retrieve relevant low-precision training studies while limiting the number of unrelated results. The search terms presented in Table 3.1 were selected because they retrieved relevant works previously identified during the preliminary literature exploration, including native low-precision and integer-only training approaches, while also capturing a wide range of studies related to reduced-precision neural network training. The terms within each scope were combined using the *OR* operator, while the different scopes were combined using the *AND* operator, forming the final query. This query was applied to the corresponding title, abstract and keyword fields available in each database, as summarized in Table 3.2. The exact used queries are presented in Appendix A.1.

Table 3.1: Search query for low-precision neural network training

Scope	**Terms**
Low-precision	*integer-only* OR *integer only* OR *integer training* OR *quantized training* OR *quantize training* OR *low bit-width* OR *low bit width* OR *low bitwidth* OR *low precision training* OR *low-precision training*
Training	*training* OR *train*
Machine Learning	*neural network* OR *deep learning* OR *machine learning* OR *artificial intelligence*

Table 3.2: Search fields for the query across the selected databases

Database	**Search fields**
Web of Science	Topic (Title, Abstract, Author Keywords, Keywords Plus)
Scopus	Article Title, Abstract, Keywords
IEEE Xplore	Document Title, Author Keywords, Abstract
ACM Digital Library	Title, Abstract, Keywords

The search was performed on May 24, 2026, resulting in a total of 571 documents retrieved from the selected databases, as shown in Table 3.3. An additional study, *PocketNN* [29], was manually added to the review because it had been identified during the preliminary exploratory search but was not retrieved by the searches conducted in the selected databases. *PocketNN* was retained because it satisfied the same inclusion and exclusion criteria applied to the studies retrieved through the database search. Duplicate records were removed using the Zotero citation management software, with 277 duplicates identified. After including the additional study and removing the duplicates, 295 records proceeded to the Screening phase.

Table 3.3: Number of records retrieved from each database

Database	Results
ACM Digital Library	18
IEEE Xplore	134
Scopus	264
Web of Science	155
Total	571

3.1.2 Screening

The Screening stage consisted of reviewing the titles and abstracts of the identified records to determine whether each study should proceed to full-text assessment or be excluded. When eligibility could not be determined from the title and abstract, the record was retained for full-text assessment.

To support a consistent study-selection process, a set of inclusion and exclusion criteria was established, as shown in Table 3.4. These criteria were designed to identify studies that perform neural network training under reduced-precision numerical constraints. Specifically, the review focuses on methods in which reduced-precision representations, including integer, fixed-point and low-bitwidth floating-point representations, are integrated into the training process instead of being applied only during inference.

The inclusion criteria required the selected studies to train models from random initialization rather than relying exclusively on fine-tuning pre-trained checkpoints (*IC1*). Additionally, the reviewed methodologies were required to execute substantial parts of the training process using reduced-precision numerical formats, including both forward and backward propagation when applicable (*IC2*).

FP32 arithmetic was allowed in specific parts of the training process when necessary to maintain numerical stability, such as during accumulation or other numerically sensitive operations,

Table 3.4: Selected Inclusion and Exclusion Criteria

Inclusion Criteria	Exclusion Criteria
IC1: Training from Scratch	**EC1:** Inference-Only Methods
IC2: Reduced-Precision Training	**EC2:** Full-Precision-Dependent Quantized Training
IC3: No Persistent Full-Precision Parameter Copy	**EC3:** Out-of-Scope Learning Paradigms
IC4: Training-Oriented Frameworks	**EC4:** Secondary Optimization Objectives
	EC5: Data-Free Quantization
	EC6: Adversarial Methods
	EC7: Hardware-Focused Methods
	EC8: Publication Restrictions
	EC9: Insufficient Methodological Detail
	EC10: Full Text Unavailable

provided that the model parameters were not maintained in a separate full-precision representation throughout training (e.g., as FP32 master or shadow weights) (*IC3*). Finally, only training-oriented approaches were considered. Studies focused solely on inference, or using reduced precision exclusively for secondary objectives such as communication or hardware optimization, were excluded along with studies that simply applied existing training methods to a specific domain or application, without introducing a relevant reduced-precision training methodology (*IC4*). Federated-learning studies were retained when the development of a reduced-precision training methodology was itself a central contribution, rather than when quantization was used only to reduce communication costs.

The exclusion criteria removed studies that did not align with the objectives of this review. Inference-only approaches, such as PTQ, structural pruning and other compression techniques that do not modify the training process itself, were excluded (*EC1*).

Conventional FP32-based QAT and approaches that retain persistent full-precision model parameters as the underlying representation during training were excluded (*EC2*). This includes methods that maintain FP32 master or shadow copies of the model parameters, even when reduced precision is used for parts of the forward or backward computations.

Studies based on learning paradigms outside the scope of this review were excluded, including non-neural-network models (e.g., decision trees, random forests), spiking neural networks and neural network architectures that do not follow a feedforward structure *(EC3)*. Furthermore, studies primarily focused on neural architecture search, reinforcement learning or meta-learning were excluded when reduced-precision training was not the main focus of the proposed methodology (*EC4*); studies focused on adversarial attacks or adversarial robustness were excluded for the same reason *(EC6)*. Data-free quantization methods and hardware-focused contributions that did not introduce or evaluate a corresponding reduced-precision training methodology were similarly excluded (*EC5, EC7*). Results published before 2017, secondary literature (e.g., surveys or review papers) and non-English publications were excluded (*EC8*). Furthermore, studies for which insufficient methodological details were available to assess the proposed approach and determine whether it satisfied the inclusion criteria were also excluded (*EC9*). Finally, studies without publicly accessible full text were also discarded (*EC10*).

These criteria were defined to focus the review on approaches that integrate reduced-precision numerical constraints directly into the training process. Following the application of these criteria, the 295 records entering the Screening phase were assessed, of which 189 were excluded based on the criteria presented in Figure 3.1. This resulted in 106 records proceeding to full-text retrieval. Of these, the full text of two reports was not publicly accessible, resulting in 104 reports proceeding to full-text assessment. When the title and abstract did not provide sufficient information to determine eligibility, the study was retained for full-text assessment.

3.1.3 Full-Text Assessment and Inclusion

The full-text assessment stage consisted of examining the reports retained after Screening to determine whether they satisfied the inclusion and exclusion criteria. At this stage, the methodological

details of each study were examined in detail to assess aspects that could not be determined from the title and abstract alone.

Particular attention was given to the actual implementation of the training methodology, including the use of reduced-precision computations, the representation of model parameters, the presence of full-precision master or shadow weights and whether the study introduced a reduced-precision training methodology rather than simply applying an existing approach to a specific scenario. This assessment also served to distinguish approaches that integrate reduced precision directly into training from conventional FP32-based QAT, as the use and role of quantization during training were not always clearly described in the title or abstract.

After the full-text assessment, 91 of the 104 reports were excluded. The majority of these exclusions (47 out of 91) corresponded to *EC2*, mostly due to the use of persistent full-precision copies of the model parameters, such as FP32 master or shadow weights, or because the proposed training methods remained largely dependent on full-precision computations. The second most common reason was *EC7*, concerning hardware-focused contributions that did not introduce a new training method, followed by *EC1*. In several cases, studies described their approaches as low-precision training methods, but full-text assessment revealed that reduced precision was being applied only during inference or that the training process relied on simulated quantization rather than reduced-precision computation during training.

After applying all inclusion and exclusion criteria, 13 studies remained and were included in the final review. These studies were then analyzed comparatively according to their numerical representations, precision assigned to different training components, training algorithm and adaptations, supported architectures and datasets, reported predictive performance and reported efficiency or resource usage metrics. The extracted information was synthesized narratively and through comparative tables to identify common design choices, reported trade-offs and differences in evaluation practices across the reviewed approaches.

3.2 Review Findings

The 13 studies maintained in the literature review were organized into two categories based on their approach to train under low-precision constraints: *Backpropagation Adaptation Paradigms* and *Alternative Training Paradigms*. The first category, referred to here as *Backpropagation-Preserving Approaches*, includes methods that retain end-to-end backpropagation while modifying numerical representations or supporting training operations. The second, *Alternative Learning Paradigms*, includes approaches that replace conventional backpropagation with mechanisms such as Direct Feedback Alignment (DFA), Local Error Signals (LES) or the Forward-Forward (FF) algorithm.

The *Backpropagation Adaptation Paradigms* category includes low-precision training frameworks that seek to preserve end-to-end backpropagation as the underlying learning mechanism. These approaches modify the conventional training pipeline to operate under low-bitwidth constraints, addressing challenges such as numerical overflow, limited dynamic range and unstable

gradient propagation. The approaches in this category progressively reduced the dependence of training on floating-point arithmetic while attempting to preserve conventional end-to-end back-propagation. Their development can be understood as successive attempts to identify which parts of the standard training pipeline could be reduced to low-precision without harming convergence.

Alternative Training Paradigms, on the other hand, do not try to preserve conventional back-propagation. Instead, they replace it with alternative learning mechanisms that either avoid sequential layer-by-layer error propagation or remove the backward propagation of errors entirely. This can reduce the need to propagate quantized errors across the full network, which helps some integer implementations manage numerical range

The final results of the review, which are described in detail in the following subsections are systematically represented in Table 3.5.

3.2.1 Backpropagation Adaptation Paradigms

Prior to the approaches included in this section of the review, quantization was used mainly to reduce the computational and memory requirements of neural network inference, while training generally continued to rely heavily on floating-point arithmetic. Early low-precision training methods, such as DoReFa-Net by Zhou et al. [39] and Binarized Neural Networks (BNNs) by Courbariaux et al. [9], integrated quantization into the training loop. However, these approaches kept real-valued model parameters, from which quantized weights were generated and used during forward and backward computations. The real-valued parameters were updated during training to preserve the resolution needed to accumulate small parameter changes. Consequently, although these methods reduced the precision of several computationally intensive training operations, they maintained high-precision parameter storage and updates. This limited the memory savings that could be achieved and left part of the training process dependent on high-precision arithmetic. The works discussed below address this dependence while maintaining backpropagation as the underlying learning mechanism.

One of the earliest included approaches, *FxpNet* [4], introduced by Chen et al. in 2017, addresses persistent full-precision parameter storage by representing the model's master parameters, referred to as *primal parameters*, using a dynamic fixed-point representation. Its forward pass uses binarized weights and activations for most layers, while gradients are represented in fixed-point. Groups of fixed-point values are represented using signed integer mantissas and shared power-of-two scaling factors, that are adapted during training according to the numerical range of the values being represented. This scaling mechanism is important because low-precision representations have a limited numerical range, while different variables and layers may contain values of very different magnitudes. The same general problem appears repeatedly in later frameworks, although with different solutions.

WAGE [34] uses a 2-8-8-8 bitwidth configuration in most experiments: weights are ternarized to 2 bits during the forward pass, but retained as 8-bit integers for weight updates. The authors report that, for propagated errors, preserving their direction is more important for guiding the updates of earlier layers than preserving their exact magnitude. Before quantization, propagated

errors are rescaled using a power-of-two factor. Weight gradients are also rescaled and rounded using stochastic rounding to reduce the loss of small parameter updates. *WAGE* additionally replaces Batch Normalization with a simpler constant-scaling factor.

WAGEUBN [36] extends *WAGE* by restoring Batch Normalization and introducing a quantized Momentum optimizer. While *WAGE* replaced Batch Normalization with a constant-scaling layer to avoid floating-point computation and extra stored statistics, the authors of *WAGEUBN* argue that this simplification limits its applicability to large-scale and deeper networks.

For that reason, they introduce quantized implementations of Batch Normalization and Momentum, allowing these operations to be retained without relying on conventional floating-point computation for those components. Additionally, the framework uses different quantization functions according to the numerical requirements of each component: direct quantization for weights, activations and Batch Normalization values; constant quantization with stochastic rounding for weight gradients; and shift quantization with dynamically adapted scaling for propagated errors. For some propagated errors, *WAGEUBN* uses a 9-bit representation composed of one flag bit, one sign bit and seven data bits. The flag determines whether the data bits represent values below or above a layer-wise scaling factor, extending the representable range to approximately that of a 15-bit integer while keeping the arithmetic in INT8.

NITI [30] also uses shared power-of-two scaling, coupling low-bitwidth integer values with per-layer scaling exponents. Matrix multiplications are accumulated in INT32, since their results can require more bits than the low-bitwidth representation can store. *NITI* first determines how many bits are needed to represent the largest accumulated value in a layer. If this exceeds the target precision, the values are right-shifted until they fit within the available range. The layer scaling exponent is then adjusted accordingly so that the rescaled integers continue to represent values of approximately the same magnitude. Ghaffari et al. [11] maintain the conventional algorithm and instead modify how values are represented for layer computations. They use a shared-exponent representation, aligning the mantissas within each tensor before stochastically rounding them to signed 8-bit values for integer layer computation.

Together with the previous frameworks, this illustrates a common design pattern: low-bitwidth arithmetic is often supported by additional scaling information that adapts the limited representable range to the numerical magnitudes encountered in different tensors and layers.

Huang et al. (2020), in *Adaptive Precision Training for Resource Constrained Devices* [15], take a different approach to the same numerical limitation by adapting the bitwidth of each layer during training. For simplicity, this approach is referred to here as *Adaptive Precision Training* (*APT*). Rather than fixing a representation throughout training, it monitors a gradient-to-resolution metric and increases the bitwidth of a layer when small gradient updates can no longer be represented at the current precision. Precision can also be reduced when the metric exceeds an upper threshold, indicating that additional precision is unnecessary. This strategy treats precision as a dynamic training variable and highlights an important distinction between per-iteration cost and total training cost: reducing the arithmetic cost of each iteration may provide limited benefit if the lower precision also slows convergence, requiring more epochs. One limitation of *APT* is that the

precision-adjustment strategy relies on predefined thresholds $T_{\min}$ and $T_{\max}$, which are manually selected. The authors leave automatic selection of these values for future work.

The preservation of small gradients and parameter updates constitutes a second recurring challenge. Deterministic rounding can repeatedly map updates that are too small to reach the next quantization step to zero, preventing them from contributing to parameter updates. *WAGE* and *WAGEUBN* therefore use stochastic rounding for gradients, while *NITI* introduces a pseudo-stochastic rounding mechanism that derives its rounding decision from bits that would otherwise be discarded during quantization. The authors report 90.7% validation accuracy on CIFAR-10, compared with 90.8% using conventional stochastic rounding, while avoiding the need to generate an independent random value for every rounding operation.

LNS-MADAM [38] faces a related problem because the spacing between representable values in a logarithmic number system increases with magnitude. Small additive updates may therefore be too small to move a quantized weight to the next representable value. Instead of relying on persistent high-precision master weights to preserve these updates, *LNS-MADAM* combines the logarithmic representation with the multiplicative Madam optimizer, which is better matched to this numerical representation.

Fei et al. (2025), in *Latent Weight Quantization for Integerized Training of Deep Neural Networks* [10], address the storage overhead introduced when low-precision training retains full-precision master weights. For simplicity, their approach is referred to here as *Latent Weight Quantization* (*LWQ*). Instead of maintaining an FP32 latent weight for every parameter, it stores a quantized weight together with a low-bitwidth quantized residual (4-bit in the reported experiments) representing the information lost during quantization. The residual is defined as

$$r_t = w_t - \hat{w}_t, \qquad r_t \in \left[-\frac{s}{2}, \frac{s}{2}\right)$$

where w_t is the latent weight, $\hat{w}_t$ is its quantized value and s is the uniform quantization step. During an update, the latent weight is temporarily reconstructed from the quantized weight and stored residual, updated by the optimizer and then quantized again. Instead of always rounding the residual to the nearest codebook value, the method randomly selects between neighboring codebook values. This gives small updates a chance to be preserved rather than repeatedly rounded away, reducing the risk of weight freezing. This is relevant to the scope of this review because the method avoids maintaining a persistent FP32 master copy of the entire model, while still temporarily reconstructing full-precision weights during the update of each layer.

The treatment of normalization and optimization also varies substantially across the reviewed BP-preserving approaches. *FxpNet* introduces Integer Batch Normalization and FxpADAM, extending fixed-point arithmetic to normalization and optimization operations in order to further reduce the floating-point computation required during training. However, the backward computation of Integer BN remains in floating point as the evaluated fixed-point implementation caused considerable accuracy degradation. *FxpNet* also keeps loss computation in FP32. *WAGE* instead replaces Batch Normalization with a simpler constant-scaling operation to avoid floating-point

computation and relies on mini-batch SGD instead of optimizers such as Momentum or Adam, which require additional memory to store auxiliary variables from previous updates. *WAGEUBN* subsequently reintroduces quantized Batch Normalization and Momentum, showing that removing an operation is not the only way to avoid a floating-point dependency.

NITI also replaces the conventional SGD weight-update rule with an integer-compatible mechanism inspired by Resilient Backpropagation [26]. The gradient sign and effective bitwidth determine the update magnitude, after which the gradient is shifted and rounded to produce a low-bitwidth integer update. *LNS-MADAM* [38] represents a more substantial numerical redesign by combining a logarithmic number system with the multiplicative Madam optimizer [2]. Since multiplication in the linear domain becomes addition in logarithmic space, multiplicative parameter updates can be applied directly to the stored logarithmic representation without requiring persistent FP32 master weights. The framework uses 8-bit representations for weights, activations and gradients in its main configuration and 16-bit precision for weight updates, while Batch Normalization remains in full precision.

The logarithmic representation also illustrates that reduced-precision training is not limited to fixed-point or conventional integer formats. To reduce the spacing between adjacent representable values, *LNS-MADAM* uses fractional powers of two rather than restricting the representation to integer powers of two. While multiplication is simplified in the logarithmic domain, addition is more complex because values must be converted back to the linear domain before accumulation. To reduce the cost of converting logarithmic values back to the linear domain, *LNS-MADAM* separates the exponent into an integer part and a fractional part. The integer part can be handled with bit shifts, while the fractional part is obtained through a lookup table.

The extent to which reduced precision is applied across the complete training pipeline also differs across studies. *NITI* avoids conventional floating-point arithmetic in its principal training pipeline, but matrix-multiplication results are accumulated in INT32 before being rescaled, so it is integer-only rather than uniformly INT8. Ghaffari et al. perform the computations within each layer using low-precision integer arithmetic, but convert the layer output back to FP32 before quantizing it again for the next layer. *WAGE* retains the final classification layer and Softmax in FP32 for its ImageNet configuration, while *LNS-MADAM* retains full-precision Batch Normalization in its ResNet experiments. Similarly, *WAGEUBN* leaves the first and last network layers unquantized, as the authors report that quantizing these boundary layers causes greater accuracy degradation than quantizing the hidden layers.

LWQ temporarily reconstructs FP32 latent weights during updates. These differences show that low-precision training does not necessarily indicate that every operation or training component uses the same reduced-precision representation.

The predictive performance evidence also varies with representation and task. In *FxpNet*, gradient precision is reported to be particularly sensitive: performance remains relatively stable between 12 and 32 bits in the evaluated models, but degrades considerably below 12 bits. Its Model-L configuration with 12-bit *primal parameters* and gradients reaches 88.52% accuracy on CIFAR-10, close to the 88.6% reported for the BNN comparison. *WAGE* reports 98.08% on

SVHN compared with 97.70% for DoReFa, but on ImageNet its 48.4% top-1 accuracy remains below DoReFa's reported 53.0% and its 72.18% top-5 accuracy is below the 80.71% FP32 AlexNet baseline. The ImageNet exception in the final classification layer further indicates that scaling low-precision training to larger classification tasks may require selectively retaining higher precision.

WAGEUBN also shows increasing differences as precision is reduced. On ImageNet, the reported top-1 differences relative to full precision for ResNet-18, ResNet-34 and ResNet-50 are -1.30, -3.49 and -5.59 percentage points with 16-bit errors and -3.91, -4.36 and -6.71 percentage points with 8-bit errors, respectively. The authors additionally report sensitivity to smaller batch sizes, which is relevant to resource-constrained training because memory limitations can restrict larger batch sizes.

APT, by contrast, reports resource usage relative to target accuracy rather than only at a fixed epoch count. For a 91% accuracy CIFAR-10 target, the paper reports normalized training energy of approximately 38% of the FP32 baseline. For 92% target accuracy, *APT*'s reported energy is 26% of the required by the FP32 baseline. The authors report a similar trend for memory usage.

Other approaches report small predictive performance differences in their selected configurations. On MNIST, *NITI* achieves 99.1% accuracy with LeNet-5, compared with 99.2% for the corresponding FP32 baseline. On CIFAR-10, VGG-small-9 reaches 91.8%, slightly above the 91.5% obtained with FP32. On ImageNet, AlexNet achieves 48.3% top-1 accuracy with *NITI*, compared with 49.0% in the full precision baseline. Ghaffari et al. report small accuracy reductions relative to the floating-point baseline when using ResNet18. On CIFAR-10, the INT8 configuration reaches 94.84%, 0.50 percentage points below FP32; on CIFAR-100, it reaches 75.38%, a reduction of 0.76 percentage points and on ImageNet, it achieves 69.25% top-1 accuracy, 0.50 percentage points below FP32. In contrast, with MobileNetV2 on ImageNet, the INT8 configuration reaches 72.80% top-1 accuracy, outperforming the corresponding floating-point baseline by 0.93 percentage points.

LNS-MADAM reports 93.41% on CIFAR-10 compared with 93.51% for FP32 and 76.14% on ImageNet, 0.24 percentage points below the FP32 configuration. Because *LNS-MADAM* also changes the optimizer, however, this comparison does not isolate numerical representation as the only experimental difference. A limitation of *LNS-MADAM* is its use of uniform precision across the network, despite different layers potentially having different numerical requirements. This becomes more relevant on complex tasks: on ImageNet, reducing the weight-update precision from 14 to 11 bits lowers accuracy from about 70% to 55%.

LWQ shows that randomized residual quantization remains substantially closer to the baseline with full-precision latent weights than deterministic nearest rounding. On CIFAR-10, the proposed three-segment approximation achieves 91.23% with ResNet20, 92.52% with MobileNetV2 and 94.47% with InceptionV3, corresponding to reductions of 0.84, 0.91 and 0.39 percentage points from their respective baselines. In comparison, deterministic nearest rounding produces reductions of up to 9.68 percentage points. On ImageNet, this difference becomes more pronounced with ResNet18: deterministic nearest rounding reaches 44.39%, compared with 69.75% for full-precision training, while the proposed method reaches 70.46%. The first convolutional

layer remains in FP32 because quantizing it caused substantial degradation.

Resource and execution benefits are evaluated less consistently across the reviewed studies than predictive performance. *FxpNet* argues that reducing parameter precision can lower storage and memory-access costs. Its estimate of up to 100× greater energy efficiency, however, refers to on-chip versus off-chip memory accesses instead of to end-to-end training. *WAGEUBN* compares simulated arithmetic operations across numerical formats, showing potential advantages for INT8, although these results are not measured through complete training runs. Furthermore, some auxiliary values use wider formats, including 24-bit weight updates. *NITI* reports hardware results for individual GPU operations and a partial FPGA prototype rather than complete end-to-end training. *LNS-MADAM* reports approximately 2.2×, 4.6× and 11× higher energy efficiency than FP8, FP16 and FP32, respectively, per training iteration, consisting of one forward and one backward pass. However, the accuracy and energy results were obtained in different settings: accuracy was evaluated in software, while energy was estimated from a separate hardware design rather than measured during the same training runs. *LWQ* reduces persistent FP32 weight storage by avoiding a network-wide master copy, but for CNNs this benefit is evaluated through storage requirements rather than direct physical memory measurements. Ghaffari et al. do not report hardware-level efficiency metrics such as execution time, memory usage or energy consumption, so the potential resource benefits of their integer computations are not directly evaluated.

The reviewed BP-preserving approaches provide different forms of evidence for reduced storage or arithmetic cost, but their overall execution and energy benefits cannot be directly compared because they are evaluated using different methods and experimental settings.

3.2.2 Alternative Training Paradigms

The second category includes approaches that modify the learning mechanism itself rather than adapting conventional end-to-end BP alone. In conventional BP, error signals are propagated sequentially through the network, requiring low-precision representations to preserve useful information across successive layers. However, under reduced precision, quantization errors may accumulate during this process and the limited numerical range can increase the risk of overflow. The alternative approaches reviewed here reduce or remove this sequential dependency through DFA, LES or FF, previously introduced in Sections 2.3.2, 2.3.3 and 2.3.4, respectively. In this section, the focus is on how these mechanisms are employed in the context of reduced-precision training.

DFA is the learning mechanism used in *PocketNN* [29] and *TIFeD* [8] (Tiny Integer-based Federated learning with Direct feedback alignment). *PocketNN*, introduced by Song and Lin (2022), is implemented in C++ for the training of feedforward neural networks and operates directly with integer values using integer-friendly activation functions, including piecewise-linear approximations of Sigmoid and Tanh and an integer version of ReLU6. By replacing BP with DFA, *PocketNN* computes each hidden layer's update signal directly from the output error, avoiding the recursive layer-by-layer propagation of error signals that can increase their numerical range and contribute to overflow under integer arithmetic. Together with the pocket activation functions, this allows

the framework to operate directly with integer values, without relying on separate quantization procedures or customized fixed-point representations. On MNIST and Fashion-MNIST [35], the reported *PocketNN* best validation accuracies are 96.98% and 87.70%, corresponding to differences of 1.02 and 2.09 percentage points from the floating-point BP baselines reported by the authors, respectively. The main limitation of the *PocketNN* evidence is that the practical resource trade-offs are not measured directly. The study does not provide the corresponding execution time, memory or energy comparisons against the floating-point baseline. Furthermore, the implementation evaluated in the paper is limited to fully connected networks. Its importance within this review is that it demonstrates that changing the learning algorithm can provide a route to low-precision training without redesigning conventional BP.

TIFeD, introduced by Colombo et al. (2023) and inspired by the architecture of *PocketNN*, extends the use of DFA to federated training on resource-constrained IoT devices. In the evaluated implementation, client-side training uses 16-bit integer arithmetic and can follow either a *full-network* or a *single-layer* strategy. In the *full-network* setting, each client receives the global model, trains the complete network locally on its portion of the dataset and sends the updated parameters back to the central server for aggregation. In the *single-layer* setting, each client is responsible for training only one layer of the network, taking advantage of DFA's ability to train hidden layers independently. This reduces the amount of parameters that each client must store and update and the amount of information transmitted to the server. However, it also means that each layer is updated using data from only a subset of the participating devices. In the reported MNIST and Fashion-MNIST experiments, the *full-network* strategy achieved higher accuracy than the *single-layer* configuration. The authors associate this decrease in accuracy with the smaller number of devices contributing data to the training of each layer. Although *TIFeD* is motivated by resource-constrained federated learning, its claims regarding computational, memory and energy efficiency are mainly based on the structure of the proposed method rather than direct resource measurements. This includes the authors' claim that the *single-layer* strategy halves these demands. The paper also describes *TIFeD* as entirely integer-only, but does not specify the datatype or arithmetic precision used for server-side aggregation.

LES provides a different form of local learning and is adopted by *NITRO-D* [25] and by the federated CNN approach proposed by Colombo[7]. Colombo is also a co-author of both *TIFeD* and *NITRO-D*.

NITRO-D, by Pirillo et al., first released as a preprint in 2024 and revised in 2025, divides the network into independent segments called local-loss blocks. Each block contains forward layers, which propagate activations through the network and learning layers, which use the block activations to produce an intermediate prediction. This prediction is compared with the target using a Residual Sum of Squares loss, providing each block its own local learning signal. To maintain the activations within the INT8 range, *NITRO-D* introduces the *NITRO Scaling Layer* and *NITRO-ReLU*. The scaling layer rescales the integer pre-activations before the activation function, while *NITRO-ReLU* bounds the resulting activations to the symmetric INT8 range $[-127, 127]$. During backpropagation, gradients remain within their respective local-loss blocks rather than

being propagated through the complete network.

Parameter updates are performed using IntegerSGD, which adapts SGD and weight decay to integer arithmetic by representing their rates through integer reciprocals and performing the corresponding operations using integer division. Since multiplication by the learning-layer weights amplifies the gradient received by the forward layers, *NITRO-D* additionally introduces an Amplification Factor that adjusts their learning rate to maintain suitable update magnitudes. The independence of the local-loss blocks also prevents error signals from being recursively propagated across the complete network and allows their backward computations to be performed independently. Although activations and weights use low-bitwidth integer representations, intermediate results of matrix multiplications require wider integer representations before being rescaled to the INT8 activation range.

To evaluate the framework, the authors first compare *NITRO-D* with *PocketNN*, the full-precision LES baseline reported in the original LES paper [23] and standard full-precision BP using MLP architectures. On the architectures corresponding to the *PocketNN* experiments, *NITRO-D* achieves 97.36% on MNIST and 88.66% on Fashion-MNIST, outperforming *PocketNN* by 0.38 and 0.96 percentage points, respectively. Across the MNIST and Fashion-MNIST comparisons with the full-precision baselines, the accuracy gaps range from 0.64 to 2.50 percentage points. On CIFAR-10, however, the gap is larger: *NITRO-D* achieves 61.03%, compared with 67.70% for full-precision LES and 66.40% for full-precision BP, corresponding to reductions of 6.67 and 5.37 percentage points, respectively.

The evaluation is then extended to VGG-based CNN architectures. With VGG8B, *NITRO-D* achieves 99.45% on MNIST and 93.66% on Fashion-MNIST, corresponding to reductions of 0.15 and 0.68 percentage points relative to full-precision LES and 0.29 and 1.81 percentage points relative to full-precision BP, respectively. The differences become more evident on CIFAR-10: VGG8B achieves 87.96%, compared with 91.60% for full-precision LES and 94.01% for full-precision BP, while VGG11B achieves 87.39%, compared with 91.61% and 94.44%, respectively. The largest observed difference is 7.05 percentage points for VGG11B relative to full-precision BP.

Despite extending the approach to CNNs and maintaining relatively small accuracy differences on MNIST and Fashion-MNIST, the performance gaps are considerable on CIFAR-10 when compared with the full-precision baselines. The authors associate these differences with the use of the integer-compatible RSS loss instead of categorical cross-entropy and IntegerSGD instead of the more advanced Adam optimizer. Furthermore, the reported memory and energy reductions are estimated from operation costs rather than measured during complete training runs.

Colombo [7] extends *NITRO-D*'s LES-based integer training to federated CNNs, building on *TIFeD*'s approach of distributing model training across resource-constrained clients. While *TIFeD* supports full-network and single-layer training, the proposed method uses a server-side heuristic to determine how many layers each client trains according to its memory capacity, computational power and energy availability. By exploiting LES, the backward pass is restricted to the local loss blocks associated with the layers assigned to each client.

Partial-model allocation reduces the amount of computation and intermediate state required during training, as well as the amount of updated parameters transmitted to the server. However, for a fixed number of devices, assigning fewer layers to each device means that fewer clients contribute to the training of each individual layer, introducing a trade-off with predictive performance.

In the MLP experiments, the proposed method outperformed *TIFeD* by up to 8.44 percentage points on MNIST and 3.00 percentage points on Fashion-MNIST in the single-layer setting. For CNNs, full-model allocation achieved the highest accuracy, while single-layer allocation resulted in lower values. With 40 devices under full-model allocation, the method reported accuracies of approximately 98% on MNIST, 86% on Fashion-MNIST and 65% on CIFAR-10, around 1, 3 and 2 percentage points below the corresponding FP32 FedAvg baseline.

The method uses 16-bit integer model weights. During training, each client retains the weights up to its deepest assigned layer, while storing the intermediate values needed to update its assigned layers and their local blocks. However, performing inference with the final global model requires all layers to be available on the device.

The computational and memory advantages of partial allocation are supported through analytical comparisons, but the study does not provide direct end-to-end measurements of execution time, memory usage or energy consumption on the target hardware.

Ma et al. (2025) propose *FF-INT8* [20], which takes a different approach to low-precision training by using the Forward-Forward (FF) algorithm [14]. The authors motivate the approach with experiments showing that directly quantizing gradients to INT8 during BP leads to larger accuracy losses in deeper networks, which they attribute to the accumulation of quantization errors during backpropagation. By replacing end-to-end BP with the FF algorithm, *FF-INT8* avoids repeatedly propagating quantized gradients back through previous layers.

A limitation of the original FF algorithm is that earlier layers do not receive information from deeper layers, resulting in slower convergence, with lower accuracy. To address this limitation, *FF-INT8* introduces a *Look-Ahead* scheme that combines each layer's local loss with a weighted contribution from the losses computed using later layers' *goodness* values. Using the *Look-Ahead* scheme, the authors report improvements in both convergence and final accuracy, particularly for deeper architectures, when compared with the original FF formulation. However, it requires storing all network weights during the complete forward pass, increasing memory consumption relative to vanilla FF, although it still avoids storing the intermediate activations and complete gradient chain required by BP.

When evaluated on an NVIDIA Jetson Orin Nano with an INT8 engine, *FF-INT8* reports accuracy between 0.2 and 0.8 percentage points below FP32 BP across the evaluated models: an MLP on MNIST and MobileNet-V2, EfficientNet-B0 and ResNet-18 on CIFAR-10. Across these experiments, it reduces training time by an average of 28.6%, energy consumption by 46.4% and memory usage by 38.7% compared with FP32 BP. Relative to GDAI8 [31], the state-of-the-art INT8 BP method used by the authors for comparison, the corresponding reductions of *FF-INT8* are 4.6%, 8.3% and 27.0%, while average accuracy is 0.2 percentage points higher. These comparisons require some caution because *FF-INT8* changes the learning algorithm together with

numerical precision. *FF-INT8* is also not entirely integer-only: its matrix-multiplication inputs use INT8 precision with INT32 accumulation, while selected operations, including the loss and initial output gradient, are computed in FP32 before gradients are quantized for subsequent computation.

3.2.3 Comparative Analysis of Reviewed Frameworks

The reviewed works can now be compared in relation to RQ1 and RQ2 established in Section 1.3. RQ1 focuses on how reduced numerical precision is integrated into training and the adaptations required to support it, while RQ2 focuses on the predictive performance, execution-efficiency and resource usage trade-offs reported in the literature. Since the studies use different datasets, architectures, implementations, hardware platforms and evaluation procedures, the following comparison focuses on common patterns rather than treating the reported results as directly comparable.

Regarding RQ1, the reviewed frameworks employ different numerical representations and bitwidths adopted throughout the training pipelines. Despite this variability, 8-bit integer representations are frequently used to represent weights, activations, gradients and errors. In this context, gradients refer to the derivatives used to update the model parameters, whereas errors refer to the signals propagated between layers during the backward pass. However, reduced precision is not uniformly applied to every training component, as shown in Table 3.5, which summarizes the numerical formats used across different parts of the training process. Numerically sensitive operations, namely accumulation and parameter updates, often use wider representations. For instance, several frameworks rely on INT32 accumulators to prevent overflow, as MAC operations may produce values that cannot be represented using lower bitwidths. Similarly, parameter updates are often kept at higher precision to avoid discarding small updates that may accumulate into non-negligible changes over time.

The reviewed approaches also show that low-precision training is not restricted to conventional integer representations. *FxpNet* and *WAGE* use binary or ternary values in parts of the forward pass; Huang et al. dynamically adjust the bitwidth during training using runtime metrics and LNS-Madam uses a logarithmic number representation to replace costly multiplications with simpler additions in the logarithmic domain. Furthermore, most of the analyzed works still rely on floating-point arithmetic for certain parts of the training. Some of them use FP32 for operations such as BN, loss calculation, communication between layers or temporary weight reconstruction. However, unlike conventional QAT approaches, the reviewed frameworks generally avoid maintaining persistent FP32 master or shadow weights throughout training. Overall, the results show that most frameworks do not use a single precision across the entire training process: lower bitwidths are generally used to represent weights, activations, gradients and errors, while wider representations are applied when additional numerical range or precision is required.

The reviewed works were analyzed to identify the modifications applied to conventional training under reduced-precision constraints. These modifications mainly address the limited numerical range of low-bitwidth representations, the loss of small values during quantization and the difficulty of implementing some operations without floating-point arithmetic. Table 3.6 summarizes some of the most frequently used strategies, along with one of the main innovations introduced by

each work. One of the most common adaptations is the use of scaling mechanisms before values are quantized. Scaling factors are used to map values to the limited numerical range of low-bitwidth representations. Depending on the framework, the scaling factor can be calculated from the tensor's range, updated during training or kept fixed. It can also be restricted to powers-of-two, allowing scaling operations to be performed using shift operations. In some approaches, the scaling factor works similarly to a shared exponent. *NITI* and the framework presented by Ghaffari et al., for example, associate a group of integer values with a shared exponent that determines the scale of the tensor and changes according to the range of its values. In contrast, *NITRO-D* uses a fixed scaling factor for each layer as part of the mechanism that keeps activations within the INT8 range of $[-127, 127]$.

Rounding is another frequent modification, particularly when gradients or parameter updates become too small to be represented at the chosen bitwidth. With deterministic rounding, some small values may repeatedly be rounded to zero, causing parameter updates to be lost over time. To tackle this, some frameworks use stochastic rounding. When a value is scaled before quantization, the result can contain a fractional part that cannot be represented in the quantized format. Instead of discarding this part, stochastic rounding uses its size to determine the probability of rounding the value up or down.

Optimization and normalization are also modified in several frameworks. *FxpNet* introduces fixed-point implementations of ADAM and BN, although part of the BN backward pass remains in floating-point. *WAGE* instead removes BN and relies on mini-batch SGD, avoiding the extra memory required by optimizers such as Momentum or Adam to store information from previous updates. *WAGEUBN* later addresses these limitations by introducing quantized versions of both BN and Momentum. *NITI* also removes BN and replaces the conventional update rule with an integer one inspired by Resilient Backpropagation, while LNS-Madam uses a multiplicative optimizer that better fits its logarithmic representation.

Quantization sometimes depends on non-differentiable operations, such as rounding or binarization, whose derivatives may be zero or undefined. This creates a problem during backpropagation because their zero or undefined derivatives prevent useful gradient propagation. To deal with this, some frameworks use STE (described in Section 2) to approximate their derivatives during the backward pass. *FxpNet*, for example, uses an STE for the sign function used during binarization of weights and activations.

Some of the reviewed works take a different approach and avoid adapting conventional BP, replacing it with alternative training algorithms. In conventional BP, errors are propagated sequentially through the network, meaning that quantization errors can accumulate across layers and intermediate values become larger, increasing the risk of overflow. *PocketNN* and *TIFeD* use DFA, which avoids this sequential error propagation; *NITRO-D* and Colombo et al. use LES, where learning is performed locally instead of propagating errors through the entire network and *FF-INT8* uses the Forward-Forward algorithm, which also avoids conventional end-to-end error propagation.

In general, the reviewed frameworks show that supporting low-precision training requires

more than reducing the bitwidth of the values used during training. The proposed modifications include scaling and rounding mechanisms, changes to numerical representations, simplified optimizers and normalization, the use of STEs to tackle non-differentiability of some operations and, in some cases, the complete replacement of BP with alternative learning algorithms.

The reviewed literature therefore answers RQ1 not with a single preferred numerical format, but with a collection of design strategies that allocate precision according to the numerical role of each training component. The common objective is to preserve sufficient dynamic range and update resolution while reducing persistent full-precision state and the amount of high-precision arithmetic. The specific solution depends on the representation and learning algorithm: BP-preserving methods generally introduce scaling, rounding and optimizer or normalization adaptations, while alternative-learning methods additionally change how learning signals are propagated.

Regarding RQ2, Tables 3.7 and 3.8 compare the predictive performance of the low-precision approaches with their reported full-precision baselines, where available, and summarize the corresponding efficiency evidence. Appendix A.2 separately summarizes the classification accuracies reported for the low-precision approaches. These results were obtained under different experimental conditions and should therefore be interpreted as a structured summary of the literature rather than as a direct ranking of frameworks.

In some cases, relatively small accuracy differences are obtained together with considerable resource reductions. For example, *NITI* reports differences between -0.30% and +0.30% on CIFAR-10 while reducing GPU training memory by around 45% in both scenarios. *FF-INT8* also reports accuracy losses below 1% while reducing memory, energy consumption and training time. On the other hand, *NITRO-D* obtains larger memory reductions in some configurations, above 70%, but also presents larger accuracy drops on CIFAR-10.

It is important to consider that the reported efficiency gains are measured differently across the analyzed works. Some frameworks report training memory, time or energy measurements, while others rely on analytical estimates, simulations or only evaluate specific components such as weight storage. Even among the works that provide hardware measurements, the evaluated metrics are not always the same. For example, *FF-INT8* measures training time, energy and memory on an edge device, while *NITI* reports memory measurements and separate hardware efficiency results. *NITRO-D* reports estimated memory and energy reductions rather than end-to-end measurements of the complete implementation. Because of these differences, the trade-off between accuracy and efficiency cannot be fairly compared across all frameworks.

Another limitation is that numerical precision is often not the only factor that changes between the frameworks and their baselines. Learning algorithms, optimizers, normalization procedures and software implementations can also differ. For example, *PocketNN* and *NITRO-D* replace conventional BP with DFA and LES, respectively. Therefore, the reported differences characterize the complete training approaches rather than isolating the effect of numerical precision.

To conclude, the reviewed works show that low-precision training can reduce resource usage while maintaining competitive accuracy, but several limitations remain. The magnitude and type

of the reported benefits depend on the numerical representation, training adaptations, learning algorithm, implementation, hardware platform and measurement methodology. The frameworks are evaluated using different architectures, datasets, hardware and efficiency metrics, making comparisons between them difficult. Furthermore, several approaches still rely on FP32 precision operations in some parts of the training pipeline, while others only demonstrate their efficiency through simulations, analytical estimates or partial hardware implementations. In some cases, the learning algorithm is also changed together with the numerical precision, making it difficult to isolate the effect of quantization on accuracy. Finally, despite the motivation towards edge and resource-constrained training, end-to-end evaluations on this type of hardware remain limited. These differences make it difficult to determine the overall benefits of low-precision training independently of the implementation and experimental setup.

This limitation also motivates the experimental part of this dissertation. To make the comparison between frameworks more consistent, the selected implementations are evaluated using common datasets, architectures and metrics whenever possible. However, some framework-specific constraints cannot be harmonized and are therefore kept and documented. Chapter 4 describes the experimental methodology used to compare the selected frameworks under more comparable conditions. The results are therefore interpreted as evidence of the behaviour of the evaluated implementations, rather than as a direct measurement of the effect of numerical precision alone.

Table 3.5: Comparison of numerical formats used by low-precision training frameworks.

Work	Weights	Activations	Gradients	Errors	Weight storage / update precision	Accumulation precision	Batch normalization	Optimizer	Unquantized components
FxpNet [4]	1	1[a]	24/16/12[b]	24/16/12[b]	24/16/12[b]	Not specified[c]	Integer BN	FxpAdam	Backward computation of Integer BN and Softmax Cross-Entropy loss layer
WAGE [34]	2[d]	8	8	8	8	Widened MACs[d]	Constant scaling factor / BN removed	Mini-batch SGD without momentum	Last fully connected layer and Softmax (FP32) on ImageNet
APT [15]	6–13[h]	Not detailed	Not detailed	Not detailed	6–13[h]	Not specified	Standard FP32 BN	SGD with momentum	Batch Normalization
WAGEUBN [36]	8	8	8 and 15	8/9/16[e]	24	$[k_A + k_W - 1]$ (forward) $[k_{E_2} + k_A - 1]$ (backward)	Quantized to 16-bit integers	Quantized Momentum	First and last layers
NITI [30]	8	8	5[*]	8	8	INT32	Not used	Inspired by Resilient Backpropagation	None reported
LNS-Madam [38]	8	8	8	8	16	INT24	Standard FP32 BN	Madam	Batch Normalization Layers
Ghaffari et al. [11]	8	8	8	8	16 (SGD/update)	INT32	INT8 BN	INT16 SGD	FP32 inter-layer communication and Vision Transformers attention Softmax
PocketNN [29] [f]	8	8	32	32 [+]	8	INT32	Not used	Direct DFA update	None reported
TIFeD [8] [i]	16	16	16	16	16	Not detailed	Not detailed	Mini-batch SGD with DFA	Server-side aggregation
NITRO-D [25]	8	8	32	8 [j]	8	INT32	Not used	IntegerSGD	None reported
Colombo et al. [7] [i]	16	16	16	16	16	Not detailed	Not used	IntegerSGD with LES	None reported
FF-INT8 [20]	8	8	FP32 $\rightarrow$ 8-bit[g]	None	8	INT32	Not detailed	Not detailed	Loss calculation / initial output gradients
LWQ [10]	8 (or lower)	8 (or lower)	8 (or lower)	Not detailed	Quantized weight + 4-bit residual	FP32 (temporary latent-weight reconstruction)	Not detailed	Momentum; Adam 8-bit	First convolutional layer and temporary latent-weight reconstruction in FP32

Notes. "–" denotes information not specified in the reviewed publications.

[a] Activations are binary except for the 8-bit input to the first layer; forward-pass weights are binary.

[b] The values denote three Model-L configurations with matching primal-weight and gradient precisions of 24, 16 and 12 bits.

[c] FxpNet uses integer/fixed-point accumulation but does not report a universal MAC-accumulator width; Integer BN uses 32-bit integer intermediates.

[d] WAGE uses 2-bit ternary weights for computation while storing and updating weights in 8-bit integer precision. Its widened MAC outputs use $k_A + k_W - 1$, $k_E + k_W - 1$ and $k_E + k_A - 1$ bits for forward propagation, error propagation and weight-gradient computation, respectively.

[e] Weight/BN gradients use 8/15 bits. E_1 is 8-bit; E_2 is 16-bit or 8-bit plus a flag (9-bit storage). Weights and BN parameters use 24-bit storage/updates. The backward width shown is for weight gradients; error propagation uses $k_{E_2} + k_W - 1$ bits. BN statistics use 16 bits, Momentum uses a 13-bit accumulator and the first/last layers remain unquantized.

[f] The paper describes the framework globally rather than specifying each tensor separately.

[g] The raw loss derivative starts in FP32 but is immediately quantized to INT8 before any parameter updates are processed.

[h] APT initializes layers at 6 bits and reaches 13 bits in the reported ResNet20 experiment; separate bitwidths for activations, gradients and errors are not reported.

[i] TIFeD and Colombo report a global 16-bit integer implementation rather than separate widths for every tensor.

[*] Weight gradients are accumulated in INT32 and reduced to an effective update width of at most 5 bits.

[+] Although stored in 32-bit precision, the values are kept within a 16-bit range.

[j] Although stored in 8-bit precision, the loss gradient is kept within a 6-bit range and propagated using INT32 intermediates.

Table 3.6: Comparison of training adaptations used by the reviewed low-precision frameworks.

Work	Year	Main innovation	Learning mechanism	Stochastic rounding	Scaling-factor classification	Optimizer/update rule	Normalization	STE
FxpNet [4]	2017	Quantize Primal Parameters to Fixed-Point	BP	No	Dynamic power-of-two scaling	FxpADAM	Integer BN (forward in INT32, backward pass uses FP32)	Yes, for sign binarization
WAGE [34]	2018	Bidirectional Discrete Dataflow	BP	Yes	Static and dynamic power-of-two scaling	Mini-batch SGD without momentum; optional L_2 regularization	BN replaced by a fixed layer-wise scaling factor	Implicit; not called STE
APT [15]	2020	Dynamic Layer Wise Precision Allocation	BP	No	Dynamic range-based scaling	SGD with momentum and weight decay	Standard BN	No
WAGEUBN [36]	2020	Quantized BN and Momentum	BP	Yes	Static and dynamic power-of-two scaling	Fully quantized momentum SGD	Quantized BN	Yes
NITI [30]	2022	Eliminates all FP32 fallbacks	BP	Yes, pseudo stochastic	Dynamic power-of-two scaling	Discrete RPROP-inspired integer update	BN removed	No
Ghaffari et al. [11]	2022	Theoretical convergence guarantees for integer SGD	BP	Yes	Dynamic block-exponent scaling	Integer SGD with momentum and weight decay; AdamW for ViT experiments	Integer BN and LayerNorm; BN is frozen in some experiments	No
LNS-Madam [38]	2022	Logarithmic Number System	BP	No in the implementation; stochastic rounding appears only in theoretical analysis	Dynamic range-based scaling	Modified multiplicative Madam	Standard BN; no quantized/LNS BN is described	Yes
PocketNN [29]	2022	Discards BP, uses DFA instead	DFA	No	No explicit scaling mechanism	Direct integer DFA update using an inverse integer learning rate	None	No
TIFeD [8]	2023	Single Layer Federated with DFA	DFA	No	No explicit scaling mechanism; inherits PocketNN-style integer arithmetic	Integer DFA updates with server-side federated averaging	None	No
NITRO-D [25]	2025	Uses LES instead of BP	LES	No	Fixed layer-wise NITRO-Scaling factor	IntegerSGD with custom integer weight decay	Built-in centering in NITRO-ReLU; no BN	Yes, through NITRO-Scaling
Colombo et al. [7]	2025	Resource-Aware Federated Learning	LES	No	Inherits the fixed architecture-derived NITRO-D scaling	NITRO-D local integer updates with layer-wise server averaging	No BN reported; experiments use integer Leaky-ReLU	Yes, through NITRO-Scaling
FF-INT8 [20]	2025	Uses FF instead of BP	FF	Yes	Dynamic range-based scaling	Not Reported; direct local INT8 update	NR	Not reported
LWQ [10]	2025	Residual-based quantization of latent weights	BP (framework-dependent)	Yes	Static range-based scaling	Momentum or Adam; wraps the underlying optimizer	Framework-dependent; LWQ introduces none	Framework-dependent/inherited from underlying integerized training

Note: Years correspond to the publication year of the cited reference; earlier preprint or public-release dates are not shown.

Table 3.7: Top-1 classification accuracy and reported efficiency gains relative to full-precision training, unless otherwise noted.

Dataset	Architecture	Work	LP Top-1 (%)	FP32 Top-1 (%)	Δ Top-1 (pp)	Reported efficiency gain
MNIST	MLP (784–100–50–10)	PocketNN [29][1]	96.98	98.00	−1.02	Not quantitatively reported.
MNIST	MLP (784–100–50–10)	NITRO-D [25][2]	97.36 ± 0.23	BP: 98.00	BP: −0.64	Estimated memory reduction of 39.3% and energy reduction of 29.3% of proposed against FP32 BP.
MNIST	MLP (784–1024–1024–1024–10)	NITRO-D [25][2]	98.28 ± 0.08	LES: 99.32 BP: 99.25	LES: −1.04 BP: −0.97	Estimated memory reduction of 74.4% and energy reduction of 32.2% of proposed against FP32 BP.
MNIST	VGG8B	NITRO-D[25][2]	99.45 ± 0.05	LES: 99.60 BP: 99.74	LES: −0.15 BP: −0.29	No dataset-specific efficiency estimate reported.
MNIST	LeNet-5	NITI [30][3]	99.1	99.2	−0.10	GPU training-memory reduction of 23.8%; the FPGA kernel reports 12.1× lower latency and approximately 5× fewer DSPs.
MNIST	LeNet-5	WAGE [34][5]	99.6	–	–	4× lower weight storage than FP32; analytical rather than measured end-to-end.
MNIST	MLP (2 hidden layers; 1.79M parameters)	FF-INT8 [20][9]	94.3	94.5	−0.20	Measured reductions of 43.2% in memory, 52.6% in energy, and 35.2% in training time.
Fashion-MNIST	MLP (784–200–100–50–10)	PocketNN [29][1]	87.7	89.79	−2.09	Not quantitatively reported.
Fashion-MNIST	MLP (784–200–100–50–10)	NITRO-D [25][2]	88.66 ± 0.46	BP: 89.79	BP: −1.13	Estimated memory reduction of 35.2% and energy reduction of 29.6%.
Fashion-MNIST	MLP (784–1024–1024–1024–10)	NITRO-D [25][2]	89.13 ± 0.41	LES: 91.40 BP: 91.63	LES: −2.27 BP: −2.50	Estimated memory reduction of 74.4% and energy reduction of 32.2%.
Fashion-MNIST	VGG8B	NITRO-D [25][2]	93.66 ± 0.40	LES: 94.34 BP: 95.47	LES: −0.68 BP: −1.81	No dataset-specific efficiency estimate reported.
CIFAR-10	MLP (1024–3000–3000–3000–10)	NITRO-D [25][2]	61.03 ± 0.60	LES: 67.70 BP: 66.40	LES: −6.67 BP: −5.37	Estimated memory reduction of 73.0% and energy reduction of 32.4%.
CIFAR-10	VGG-small 7	NITI [30][3]	90.7 ± 0.1	91.0 ± 0.1	−0.30	Measured GPU training-memory reduction of 45.6%.
CIFAR-10	VGG8B	NITRO-D [25][2]	87.96 ± 0.39	LES: 91.60 BP: 94.01	LES: −3.64 BP: −6.05	Estimated memory reduction of 75.6% and energy reduction of 30.5%.
CIFAR-10	VGG-small 8	NITI [30][3]	91.5 ± 0.2	91.3 ± 0.2	+0.20	Measured GPU training-memory reduction of 44.6%.
CIFAR-10	VGG11B	NITRO-D [25][2]	87.39 ± 0.64	LES: 91.61 BP: 94.44	LES: −4.22 BP: −7.05	Estimated memory reduction of 76.1% and energy reduction of 30.5%.
CIFAR-10	VGG-small 9	NITI [30][3]	91.8 ± 0.2	91.5 ± 0.1	+0.30	Measured GPU training-memory reduction of 45.3%.
CIFAR-10	VGG-like	WAGE [34][5]	93.22	–	–	4× lower weight storage than FP32; analytical rather than measured end-to-end.
CIFAR-10	ResNet18	Ghaffari et al. [11]	94.84	95.34	−0.50	Not quantitatively reported.

Continued on the next page.

Table 3.7 continued from the previous page.

Dataset	Architecture	Work	LP Top-1 (%)	FP32 Top-1 (%)	Δ Top-1 (pp)	Reported efficiency gain
CIFAR-10	ResNet18	FF-INT8 [20][9]	93.1	93.5	−0.40	Measured reductions of 37.7% in memory, 47.1% in energy, and 30.0% in training time.
CIFAR-10	ResNet18	LNS-Madam [38][8]	93.41	93.51	−0.10	Estimated per-iteration energy reduction of 91.0% (0.54 mJ versus 5.99 mJ).
CIFAR-10	ResNet20	APT [15][6]	92.23	–	–	More than 50% combined energy and memory saving, based on a normalised cost model.
CIFAR-10	ResNet20	LWQ [10][7]	91.23 ± 0.27	–	–	Persistent FP32 latent weights are replaced by 4-bit representations; no measured end-to-end gain reported.
CIFAR-10	MobileNet v2	APT [15][6]	93.96	–	–	More than 50% combined energy and memory saving, based on a normalised cost model.
CIFAR-10	MobileNet v2	LWQ [10][7]	92.52 ± 0.21	–	–	Persistent FP32 latent weights are replaced by 4-bit representations; no measured end-to-end gain reported.
CIFAR-10	MobileNet v2	FF-INT8 [20][9]	91.1	91.5	−0.40	Measured reductions of 32.7% in memory, 46.7% in energy, and 28.1% in training time.
CIFAR-10	InceptionV3	LWQ [10][7]	94.47 ± 0.22	–	–	Persistent FP32 latent weights are replaced by 4-bit representations; no measured end-to-end gain reported.
CIFAR-10	Model-L (12-bit primal weight)	FxpNet [4][4]	88.52	–	–	Training-relative storage is 75% lower than the external BNN/QNN reference; the models have different parameter counts.
CIFAR-10	Model-L (16-bit primal weight)	FxpNet [4][4]	89.49	–	–	Training-relative storage is 67.5% lower than the external BNN/QNN reference; the models have different parameter counts.
CIFAR-10	Model-L (24-bit primal weight)	FxpNet [4][4]	89.70	–	–	Training-relative storage is 50% lower than the external BNN/QNN reference; the models have different parameter counts.
CIFAR-10	EfficientNet-B0	FF-INT8 [20][9]	88.6	89.4	−0.80	Measured reductions of 41.3% in memory, 39.4% in energy, and 20.9% in training time.
CIFAR-100	ResNet110	APT [15][6]	68.38	–	–	More than 50% combined energy and memory saving, based on a normalised cost model.
CIFAR-100	ResNet18	Ghaffari et al. [11]	75.38	76.14	−0.76	Not quantitatively reported.

LP denotes the low-precision approach reported in each work and FP32 denotes its reported full-precision baseline. $\Delta = \text{Accuracy}_{LP} - \text{Accuracy}_{FP32}$, expressed in percentage points. "–" indicates that an exact matched value was not reported. For NITRO-D, *LES* denotes FP32 using LES as the backward pass and *BP* denotes conventional FP32 backpropagation. TIFeD and Colombo et al. are not present because their main accuracy results are reported graphically across multiple federated configurations rather than as directly tabulated benchmark values.

[1] PocketNN uses DFA, whereas its FP32 comparator uses backpropagation. The difference therefore reflects both the numerical representation and training algorithm.
[2] NITRO-D reports FP32 LES and FP32 BP baselines. Both are shown when available. Its efficiency values are estimates computed per batch and training step relative to FP32 BP. For MLP1 and MLP2, no FP32 LES result is reported.
[3] NITI's memory footprints were measured on an RTX 3090. The reported 12.1× latency reduction and approximately 5× DSP reduction concern an FPGA computation kernel rather than complete training.
[4] FxpNet does not provide a matched FP32 Model-L result. Its storage comparison uses an external BNN/QNN implementation with 14.02M parameters, whereas Model-L has 9.29M parameters.
[5] WAGE's storage reduction is analytical and concerns weight storage rather than total training memory.
[6] APT reports more than 50% combined energy and memory savings relative to FP32, based on normalised modelling. It does not provide exact matched final FP32 accuracies for the listed results.
[7] The results correspond to the proposed 3Seg implementation. Persistent latent weights are represented with 4 bits, but no measured end-to-end resource reduction is reported.
[8] LNS-Madam's energy values are obtained from post-synthesis simulation combined with an analytical workload model.
[9] FF-INT8 uses Forward-Forward training, whereas its FP32 comparator uses backpropagation. Its efficiency results were measured on a Jetson Orin Nano.

Table 3.8: ImageNet top-1 and top-5 classification accuracy and reported efficiency gains relative to full-precision training.

Architecture	Work	LP Top-1 (%)	LP Top-5 (%)	FP32 Top-1 (%)	FP32 Top-5 (%)	Δ Top-1 (pp)	Δ Top-5 (pp)	Reported efficiency gain
AlexNet	WAGE [34][1]	48.4	72.18	–	80.71	–	−8.53	4× lower weight storage than FP32; operation costs are analytical rather than measured end-to-end.
AlexNet	NITI [30][2]	48.3	–	49.0	–	−0.70	–	Measured GPU training-memory reduction of 13.8%.
ResNet18 (E_2: 8-bit)	WAGEUBN [36][3]	64.79	85.81	68.70	88.37	−3.91	−2.56	Approximately 4× lower memory requirements; not measured end-to-end.
ResNet18 (E_2: 16-bit)	WAGEUBN [36][3]	67.40	87.63	68.70	88.37	−1.30	−0.74	Approximately 4× lower memory requirements; the overhead relative to 8-bit E_2 is described as negligible.
ResNet18	LWQ [10][4]	70.46	–	–	–	–	–	Persistent FP32 latent weights are replaced by 4-bit representations; no measured end-to-end gain reported.
ResNet18	Ghaffari et al. [11]	69.25	88.79	69.75	89.07	−0.50	−0.28	Not quantitatively reported.
ResNet34 (E_2: 8-bit)	WAGEUBN [36][3]	67.63	87.70	71.99	90.56	−4.36	−2.86	Approximately 4× lower memory requirements; not measured end-to-end.
ResNet34 (E_2: 16-bit)	WAGEUBN [36][3]	68.50	87.96	71.99	90.56	−3.49	−2.60	Approximately 4× lower memory requirements; the overhead relative to 8-bit E_2 is described as negligible.
ResNet50 (E_2: 8-bit)	WAGEUBN [36][3]	67.95	88.01	74.66	92.13	−6.71	−4.12	Approximately 4× lower memory requirements; not measured end-to-end.
ResNet50 (E_2: 16-bit)	WAGEUBN [36][3]	69.07	88.45	74.66	92.13	−5.59	−3.68	Approximately 4× lower memory requirements; the overhead relative to 8-bit E_2 is described as negligible.
ResNet50	LNS-Madam [38][5]	76.14	–	76.38	–	−0.24	–	Estimated per-iteration energy reduction of 91.0% (0.99 mJ versus 11.03 mJ).
ResNet50	LWQ [10][4]	76.09	–	–	–	–	–	Persistent FP32 latent weights are replaced by 4-bit representations; no measured end-to-end gain reported.
MobileNet v2	LWQ [10][4]	70.23	–	–	–	–	–	Persistent FP32 latent weights are replaced by 4-bit representations; no measured end-to-end gain reported.
MobileNet v2	Ghaffari et al. [11]	72.80	90.83	71.87	90.28	+0.93	+0.55	Not quantitatively reported.

LP denotes the low-precision approach reported in each work and FP32 denotes its reported full-precision baseline. $\Delta = \text{Accuracy}_{LP} - \text{Accuracy}_{FP32}$, expressed in percentage points. "–" indicates that an exact matched value was not reported.

[1] WAGE's top-5 accuracies were calculated from its reported top-5 error rates. The paper does not report a matched FP32 top-1 result. Its storage reduction concerns weight storage rather than total training memory.

[2] NITI's memory footprint was measured on an RTX 3090.

[3] WAGEUBN claims approximately 4× lower memory requirements relative to vanilla DNNs. Runtime and energy reductions were not measured end-to-end.

[4] The Latent results correspond to the proposed 3Seg implementation. Persistent latent weights are represented with 4 bits, but no measured end-to-end resource reduction is reported.

[5] LNS-Madam's energy values are obtained from post-synthesis simulation combined with an analytical workload model.

Chapter 4

Experimental Methodology

This chapter presents the experimental methodology used to compare the selected low-precision training frameworks with a full-precision baseline. It describes the experimental design, framework adaptations, hyperparameter optimization procedure, evaluation metrics and execution environments used to support the empirical analysis of RQ3.

4.1 Problem Statement

The literature review presented in Chapter 3 identified heterogeneity in the experimental evaluation of low-precision neural network training approaches. The reviewed studies use different datasets, network architectures, learning algorithms, software implementations, hardware platforms and evaluation metrics. In addition, reported efficiency benefits are supported by different forms of evidence, including end-to-end measurements, simulations, analytical estimates and component-level evaluations. These differences limit the extent to which the reported results can be compared across frameworks. As a consequence, the reviewed studies do not provide a sufficiently consistent empirical basis for examining how publicly available low-precision training frameworks differ in predictive performance, execution efficiency and resource usage under more similar experimental conditions. This comparability problem is relevant as low-precision training approaches are frequently motivated by reductions in computational and resource requirements, yet those benefits are not evaluated under a common experimental protocol.

The experimental study presented in this chapter addresses this limitation by evaluating a selected set of publicly available low-precision training frameworks under partially harmonized conditions. Common datasets, network architectures and evaluation metrics are used whenever allowed by the constraints of each implementation. This comparison reduces some of the experimental differences identified in the literature, while maintaining framework-specific differences in learning algorithms, software implementation and supported hardware.

This experimental design provides the empirical basis for addressing RQ3, which examines the predictive performance, execution efficiency and resource usage of selected low-precision training frameworks under more comparable conditions and relative to full-precision training.

4.2 Experimental Design

4.2.1 Framework Selection and Preliminary Validation

The frameworks selected for the experimental comparison were *NITI*[1], *PocketNN*[2] and *NITRO-D*[3]. They were selected from the approaches identified in the literature review because their official implementations were publicly available and because reduced precision is integrated directly into training without relying on persistent FP32 master or shadow weights. Federated-learning approaches were not included as their distributed training setting introduces additional factors, such as client allocation, communication and aggregation, that are not directly comparable with the centralized training setting considered in this work.

The selected frameworks also represent different approaches to low-precision training. *NITI* preserves conventional backpropagation, while *PocketNN* and *NITRO-D* replace it with DFA and LES, respectively. Their software and hardware requirements also differ. *NITI* is implemented in Python using custom CUDA kernels and requires GPU execution. *PocketNN* is implemented in C++, supports only fully connected neural networks and executes exclusively on the CPU. *NITRO-D* is Python-based, supports both MLPs and CNNs and can execute on either CPU or GPU. Preliminary CPU executions of *NITRO-D* were substantially slower, so its GPU implementation was used in the final experiments. The three low-precision frameworks were compared with a conventional FP32 backpropagation baseline implemented in PyTorch[4]. The baseline uses 32-bit floating-point representations and is trained using backpropagation with cross-entropy loss and stochastic gradient descent, with gradients computed through PyTorch's automatic differentiation.

Before defining the main experimental comparison, preliminary experiments were conducted to verify that the released implementations could be executed successfully and that their predictive performance could be approximately reproduced under the available environments. These experiments were intended as implementation checks rather than as direct comparisons between frameworks, since they used the framework-specific configurations reported by the original authors and were executed in different hardware environments.

PocketNN was executed locally, while *NITI* and *NITRO-D* were executed on Kaggle [5]. The local system was not compatible with the original CUDA and dependency configurations required by *NITI* and *NITRO-D* without modifying their released software environments. Kaggle was therefore used to preserve those environments as closely as possible.

The local experiments were performed on a laptop with a 13th Gen Intel Core i7-13650HX processor, 32 GB of RAM and an NVIDIA GeForce RTX 5060 Laptop GPU with 8 GB of VRAM. The system used Ubuntu 24.04.4 LTS with Linux kernel 7.0.0-31-generic. The Kaggle experiments used an Intel Xeon CPU with four cores, approximately 33 GB of RAM and two NVIDIA

[1] https://github.com/wangmaolin/niti
[2] https://github.com/jaewoosong/pocketnn
[3] https://github.com/AI-Tech-Research-Lab/nitro-d
[4] https://pytorch.org/
[5] https://www.kaggle.com/

Tesla T4 GPUs, each with 16 GB of VRAM. The environment used Ubuntu 22.04.5 LTS with Linux kernel 6.12.90+ and NVIDIA driver version 580.159.04.

Not all experiments reported in the original works could be reproduced from the released source code. Although the *NITI* paper reports experiments on MNIST with LeNet-5 and on ImageNet with AlexNet, these configurations are not provided in the official repository, which includes only a CIFAR-10 pipeline using VGG-small architectures. Preliminary validation of *NITI* was therefore restricted to the configurations supported by the released source code. In addition, the *NITRO-D* VGG11B experiment on CIFAR-10 could not be completed within the 12-hour maximum duration of a Kaggle session.

Each reproducible configuration was executed 10 times and the resulting accuracy was reported as the mean and standard deviation across runs. *PocketNN* produced the same final accuracy across the repeated preliminary executions, resulting in a standard deviation of zero. Table 4.1 compares the values reported in the original publications with those reproduced using the released implementations.

Table 4.1: Comparison between the accuracy reported in the original works and the accuracy reproduced using the released implementations.

Dataset	Framework	Architecture	Reported Accuracy (%)	Reproduced Accuracy (%)
MNIST	PocketNN	MLP1	96.98	96.80 ± 0.00
Fashion-MNIST	PocketNN	MLP2	87.7	87.87 ± 0.00
MNIST	NITRO-D	VGG8B	99.45 ± 0.05	99.45 ± 0.02
MNIST	NITRO-D	MLP1	97.36 ± 0.23	97.36 ± 0.12
MNIST	NITRO-D	MLP3	98.28 ± 0.08	98.28 ± 0.04
Fashion-MNIST	NITRO-D	VGG8B	93.66 ± 0.40	93.55 ± 0.22
Fashion-MNIST	NITRO-D	MLP2	88.66 ± 0.46	88.66 ± 0.24
Fashion-MNIST	NITRO-D	MLP3	89.13 ± 0.41	89.13 ± 0.22
CIFAR-10	NITRO-D	VGG8B	87.96 ± 0.39	87.86 ± 0.31
CIFAR-10	NITRO-D	MLP4	61.03 ± 0.60	61.03 ± 0.31
CIFAR-10	NITRO-D	VGG11B	87.39 ± 0.64	–
MNIST	NITI	LeNet	99.1	–
ImageNet	NITI	AlexNet	48.3	–
CIFAR-10	NITI	VGG-small 7	90.7 ± 0.1	90.66 ± 0.39
CIFAR-10	NITI	VGG-small 8	91.5 ± 0.2	91.32 ± 0.13
CIFAR-10	NITI	VGG-small 9	91.8 ± 0.2	91.72 ± 0.12

Note: MLP1: 784-100-50-10; MLP2: 784-200-100-50-10; MLP3: 784-1024-1024-1024-10; MLP4: 1024-3000-3000-3000-10. The reproduced values follow the accuracy reporting criterion used by each original work. "–" indicates configurations that were not reproduced using the released implementation.

For the configurations that could be executed, the obtained results were close to those reported by the original authors. This provided confidence that the released implementations were functioning as expected before using them in the subsequent experiments, although the validation was not intended as a formal replication of the original studies.

4.2.2 Datasets and Data Splits

The experimental evaluation used two image-classification datasets: MNIST [18] and CIFAR-10 [17]. MNIST contains 28×28 grayscale images of handwritten digits distributed across ten classes. The official dataset contains 60,000 training images and 10,000 test images. CIFAR-10 contains 32×32 RGB images distributed across ten mutually exclusive object classes and includes 50,000 training images and 10,000 test images.

Representative examples from each CIFAR-10 and MNIST class are shown in Figures 4.1 and 4.2, respectively.

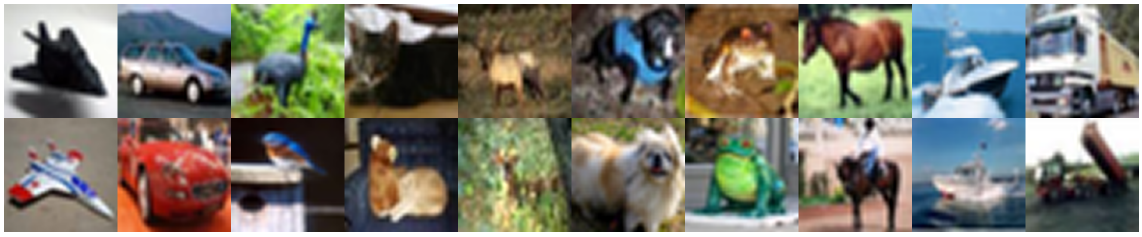


Figure 4.1: Example images from the CIFAR-10 dataset (each column corresponds to one of the ten object classes).



Figure 4.2: Example images from the MNIST dataset.

Both MNIST and CIFAR-10 were used in the MLP experiments, while the CNN experiments were performed only on CIFAR-10.

During hyperparameter optimization, the original training set was divided into training and validation subsets, with a 90/10 split used unless otherwise specified. The original test set was not used for hyperparameter or architecture selection. After hyperparameter optimization, each final model was trained from scratch using the selected configurations and the complete original training set. During these final runs, test accuracy was recorded after each epoch to characterize the learning evolution, while final-epoch test accuracy was used as the primary predictive performance metric.

Data augmentation was not applied in the MLP experiments because *PocketNN* does not support it and introducing framework-specific augmentation would have added another source of

variation between the implementations. The CNN experiments were also performed without data augmentation to keep this aspect consistent across the evaluated frameworks. Apart from data augmentation, the preprocessing procedures defined in the original *NITI* and *NITRO-D* implementations were preserved.

4.2.3 Framework Adaptations and Comparison Scenarios

Several modifications were required to evaluate the released implementations under a common experimental workflow. These adaptations were limited to the functionality required for comparison and metric collection, while the framework-specific training algorithms were preserved.

For instance, the released *NITI* implementation did not provide an MLP pipeline, although the integer layers and training operations required to construct one were already available. An MLP model and corresponding MNIST and CIFAR-10 data loaders were therefore added without changing the original integer-training method.

PocketNN also required an additional data loader because its released implementation did not support CIFAR-10. The added loader maps CIFAR-10 pixel values to the signed integer range $[-127, 127]$.

Because *PocketNN* is implemented in C++, it was additionally bound to Python using pybind11[6]. This allowed the original C++ training functions to be called directly from Python, making it possible to use Optuna [7] for hyperparameter optimization and to apply the same monitoring workflow used for the Python-based frameworks. The training procedure itself remained in the original C++ implementation.

To verify that the Python binding did not introduce a relevant execution time overhead, the same *PocketNN* configuration was executed directly in C++ and through the pybind11 interface. Using MNIST, the 784-100-50-10 architecture (corresponding to 784 input units, two hidden layers with 100 and 50 neurons and 10 output units), batch size 20, inverse learning rate 1000, PocketTanh and 10 training epochs, each configuration was executed three times. Direct C++ execution required 176.755 ± 0.436 s on average, whereas the Python-bound execution required 176.018 ± 0.529 s, an absolute difference of approximately 0.42%. The final accuracy was unchanged, so the binding was maintained for the experimental evaluation.

The empirical study was organized into three comparison scenarios. The first scenario used a common MLP architecture across all four training approaches. For MNIST, the 784-100-50-10 architecture used in the original *PocketNN* work was selected. This architecture had also been used in the *NITRO-D* comparison with *PocketNN*. The same network structure was implemented for *NITI* and the FP32 baseline, while the remaining tunable parameters were selected separately when required. For CIFAR-10, the MLP architecture originally used by *NITRO-D* was too large to be adopted by *PocketNN*. Since *PocketNN* showed stronger limitations as the network size increased, it was used as the reference when selecting a common architecture that could be supported by all frameworks. The architecture was selected through hyperparameter optimization with *PocketNN*,

[6] https://github.com/pybind/pybind11
[7] https://optuna.org/

resulting in a 3072-768-10 MLP, which was then used across *PocketNN*, *NITRO-D*, *NITI* and the FP32 baseline.

The second scenario allowed the MLP architecture and the remaining hyperparameters to be selected independently for each framework. Its main focus was predictive performance, assessing the accuracy each implementation could achieve with a configuration more closely adapted to its own training method. It is referred to as the *individually tuned* comparison for consistency throughout the dissertation, although not every configuration was tuned in this work: the original PocketNN MNIST configuration was retained, while the PocketNN CIFAR-10 configuration resulted from the search described below. *NITI*, *NITRO-D* and the FP32 baseline included architecture parameters in their respective optimization processes.

The third scenario compared CNN training using *NITI*, *NITRO-D* and the FP32 baseline. *PocketNN* was not included because its implementation supports only fully connected neural networks. VGG-Small-8 was selected as the common CNN architecture because it was supported by *NITI* and could be reproduced in the *NITRO-D* and FP32 implementations while keeping the model size moderate for repeated experiments. The architecture contains seven convolutional layers followed by one fully connected output layer, with max-pooling between convolutional blocks. It contains approximately 4.8 million parameters, considerably fewer than the VGG8B configuration originally used by *NITRO-D*, which contains approximately 8.9 million parameters, making it more practical for repeated comparative experiments. The configurations of both architectures are shown in Table 4.2.

Table 4.2: Convolutional architecture configurations.

Architecture	Configuration
VGG-Small-8 (NITI)	$3 \times \text{Conv}_{3\times3}(128) \rightarrow \text{MP2} \rightarrow 2 \times \text{Conv}_{3\times3}(256) \rightarrow \text{MP2} \rightarrow 2 \times \text{Conv}_{3\times3}(512) \rightarrow \text{MP2} \rightarrow \text{FC}(10)$
VGG8B (NITRO-D)	$\text{Conv}_{3\times3}(128) \rightarrow \text{Conv}_{3\times3}(256) \rightarrow \text{MP2} \rightarrow \text{Conv}_{3\times3}(256) \rightarrow \text{Conv}_{3\times3}(512) \rightarrow \text{MP2} \rightarrow \text{Conv}_{3\times3}(512) \rightarrow \text{MP2} \rightarrow \text{Conv}_{3\times3}(512) \rightarrow \text{MP2} \rightarrow \text{FC}(1024) \rightarrow \text{FC}(10)$

The common-architecture comparisons control the network topology but do not make the implementations identical. In particular, the learning algorithms remain different by design. In the CNN experiments, *NITRO-D* also retained its framework-specific activation function and corresponding weight initialization, while *NITI* and the FP32 baseline used their own configurations. These framework-specific choices were maintained because changing them would modify the original training method. Consequently, the experiments compare the complete framework implementations under more similar conditions, rather than isolating numerical precision or the learning algorithm as the only difference between them.

4.2.4 Hyperparameter Optimization

Hyperparameter optimization was performed using Optuna, with validation accuracy used as the objective to be maximized. The searches used the Tree-structured Parzen Estimator (TPE) sampler.

For *NITI*, *NITRO-D* and the FP32 baseline, a median pruner was additionally used to stop poorly performing trials. It was configured with 10 startup trials and 15 warm-up steps before pruning could occur. For each trial, the objective value was the maximum validation accuracy observed across the training epochs. These searches were stored in SQLite databases so that the same Optuna study could be resumed across Kaggle sessions when required.

For *PocketNN* on CIFAR-10, optimization was performed in two stages, both executed locally on the laptop described in Section 4.2.1. The first stage explored a wider search space, varying the number and size of hidden layers, inverse learning rate, batch size and activation function, as detailed in Table A.2. For *PocketNN*, the number of layers includes the input and output layers. Each configuration was trained for five epochs using 90% of the CIFAR-10 training split, with the remaining 10% used for validation. Configurations that resulted in overflow during training were pruned.

The initial search evaluated 286 configurations, of which 223 completed successfully and 63 were pruned due to overflow. The highest validation accuracy was 45.28%, obtained with the 3072-928-10 architecture, a batch size of 128, an inverse learning rate of 1512 and *PocketTanh*. The highest-performing completed configurations were concentrated around a smaller region of the search space: all configurations in the top 10% used *PocketTanh*, 22 of the 23 used three layers and 20 used a batch size of 128. The two-layer configuration, which contained no hidden layer, was included only as an exploratory case and did not produce competitive results, so it was not considered in the subsequent analysis. The behaviour by network depth is summarized in Table 4.3.

Table 4.3: Completed trials, pruning rates and best validation accuracy by number of layers.

Number of layers	Completed trials	Pruned	Best validation accuracy
3	176	18.22%	45.28%
4	18	41.94%	44.82%
5	15	31.82%	9.86%

Note: The number of layers includes the input and output layers.

The first-stage results were used to narrow the search space for the second stage. *PocketSigmoid* remained close to random classification accuracy and was excluded from the refined search. The only trial using *PocketLeakyReLU* in the initial search resulted in overflow, therefore both PocketTanh and PocketLeakyReLU were retained in the refined search space. The number of layers was restricted to three and four and the batch-size options were reduced to 32, 64 and 128. The ranges for hidden-layer size and inverse learning rate remained unchanged.

Because *PocketNN* required a substantial amount of time to train on the full CIFAR-10 training subset, the second stage used 25% of the training data used in the first stage. This reduced the cost of each epoch and allowed the number of training epochs to increase from 5 to 15. The validation set remained unchanged and validation accuracy continued to be used as the optimization objective. The refined search evaluated 500 trials and reached a highest validation accuracy of 43.93%.

The selected configuration used the 3072-768-10 architecture, an inverse learning rate of 677, a batch size of 32 and the activation function *PocketTanh*.

The validation accuracies obtained in the two *PocketNN* search stages are not directly comparable because the second stage used a smaller training subset and a different number of training epochs. The first stage was used to identify a narrower search region, while the configuration selected for the final experiments was taken from the refined second-stage search. Figure 4.3 shows the optimization history of that refined search.

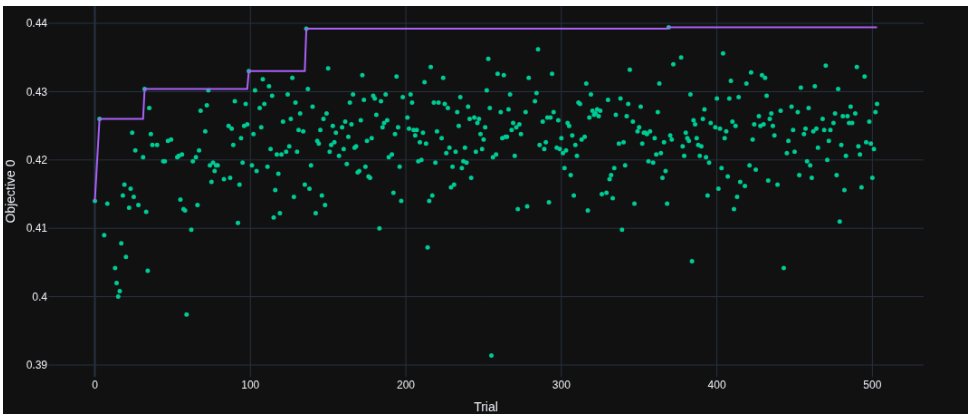


Figure 4.3: Optimization history of the refined *PocketNN* CIFAR-10 hyperparameter search. The *y*-axis represents accuracy and the *x*-axis represents the trial number.

Based on the refined search, the 3072-768-10 architecture was selected as the common CIFAR-10 MLP architecture and used across *PocketNN*, *NITRO-D*, *NITI* and the FP32 baseline. For *PocketNN*, the corresponding inverse learning rate of 677, batch size of 32 and *PocketTanh* activation were also retained in the final CIFAR-10 experiments.

4.2.4.1 Common-Architecture MLP Hyperparameter Tuning

After selecting the common MLP architectures for MNIST and CIFAR-10, the remaining hyperparameters were tuned separately for the configurations optimized in this work, while the network architecture remained fixed. These searches used 400 trials, with 60 epochs per trial and the same 90/10 training-validation split.

The framework-specific search spaces differed because each training approach exposes different parameters. For *NITRO-D*, the search included batch size, inverse learning rate, learning-rate amplification factor, forward and subnet decay factors, dropout rate and activation function. For *NITI*, the search included batch size, gradient-bitwidth schedule, rounding methods, weight initialization and use of dropout. The gradient-bitwidth schedule included three bitwidth values and the epochs at which transitions occurred. Deterministic, stochastic and pseudo-stochastic rounding were tested for activations, errors and gradients. For the FP32 baseline, the search included batch size, learning rate, weight decay, dropout rate, activation function and weight initialization. ReLU, Leaky ReLU and Tanh were considered together with uniform and normal initialization. The complete search spaces are presented in Appendix Tables A.4, A.5 and A.6.

For MNIST, additional hyperparameter optimization was not performed for *NITRO-D* because the 784-100-50-10 architecture had already been evaluated in the original work and the configuration reported by the authors was kept. Similarly, *PocketNN* kept the original author configuration for the common-architecture MNIST experiment. *NITI* and the FP32 baseline were tuned in this work. For CIFAR-10, *NITRO-D*, *NITI* and the FP32 baseline were tuned, while *PocketNN* used the configuration selected in the refined search described above. Consequently, the hyperparameter-selection procedure was not identical across all frameworks and this asymmetry is considered when interpreting the results.

The highest validation accuracies obtained during these searches are shown in Table 4.4. The corresponding selected configurations are provided in Appendix Table A.7. For each framework and dataset that was tuned in this work, the configuration achieving the highest validation accuracy was used in the final common-architecture experiments.

Table 4.4: Best validation accuracy obtained during hyperparameter optimization for the common-architecture MLP experiments.

Dataset	**Framework**	**Best validation accuracy**
MNIST	NITI	98.12%
MNIST	FP32 baseline	98.37%
CIFAR-10	NITRO-D	55.95%
CIFAR-10	NITI	55.50%
CIFAR-10	FP32 baseline	57.46%

4.2.4.2 MLP Architecture and Hyperparameter Tuning

In addition to the common-architecture MLP experiments, a second comparison allowed the architecture to vary together with the remaining hyperparameters. The purpose of this comparison was to evaluate the predictive performance each framework could achieve when allowed to use an architecture better adapted to its own training method, rather than being restricted to the common architecture. This architecture and hyperparameter search was performed for *NITI*, *NITRO-D* and the FP32 baseline. *PocketNN* retained the configurations already defined for each dataset: the original author configuration on MNIST and the refined-search configuration on CIFAR-10.

For *NITI*, *NITRO-D* and the FP32 baseline, the number and size of hidden layers were varied together with the framework-specific hyperparameters. As in the common-architecture searches, 400 trials were executed with 60 epochs per trial using the same dataset splits. The framework-specific search spaces in Appendix Tables A.4, A.5 and A.6 were retained, with the number and size of hidden layers added as optimization variables.

The number of hidden layers varied from one to five. The first hidden layer could contain between 512 and 1024 neurons in intervals of 32, while each subsequent hidden layer could contain between 32 neurons and the width of the preceding layer, also in steps of 32.

The highest validation accuracies obtained during the architecture searches, together with the selected architectures, are shown in Table 4.5. The selected hyperparameters are reported in Appendix Table A.8.

Table 4.5: Selected architectures and best validation accuracy by dataset and framework.

Dataset	Framework	Selected architecture	Best validation accuracy
MNIST	NITI	784-864-832-736-10	98.78%
MNIST	NITRO-D	784-960-960-960-960-960-10	98.82%
MNIST	FP32	784-864-640-384-320-10	98.77%
CIFAR-10	NITI	3072-864-10	55.12%
CIFAR-10	NITRO-D	3072-832-480-416-384-10	58.03%
CIFAR-10	FP32	3072-1024-1024-10	59.00%

4.2.4.3 CNN Hyperparameter Tuning

The CNN comparison included *NITI*, *NITRO-D* and the FP32 baseline; *PocketNN* was not included because its implementation supports only fully connected neural networks. For this experiment, VGG-Small-8 and a dropout rate of 0.5 were kept fixed.

ReLU was used for *NITI* and the FP32 baseline, while *NITRO-D* retained its *NITRO Leaky ReLU* activation. For *NITI* and the FP32 baseline, weight initialization was tuned between *Xavier normal* and *Xavier uniform*. *NITRO-D* kept the *Kaiming uniform* initialization associated with its activation function. These framework-specific choices were preserved rather than changed only to make the implementations identical. The remaining search spaces are shown in Appendix Tables A.10, A.9 and A.11.

Because CNN training required substantially more computation than the MLP experiments, the search budget was reduced from 400 trials with 60 epochs each to 100 trials with 30 epochs per trial. The same 90/10 training-validation split was used, the original test set remained untouched and data augmentation was not applied. The TPE sampler and median pruner were configured as described above. The selected configurations are presented in Appendix Table A.12 and the highest validation accuracies are shown in Table 4.6.

Table 4.6: Best validation accuracy obtained during hyperparameter optimization for the common-architecture CNN experiments.

Dataset	**Framework**	**Best validation accuracy**
CIFAR-10	NITRO-D	79.43%
CIFAR-10	NITI	84.68%
CIFAR-10	FP32 baseline	85.02%

The resulting experimental scenarios are summarized in Table 4.7.

Table 4.7: Experimental comparisons performed across datasets, architectures and frameworks.

Dataset	Comparison	Architecture	PocketNN	NITRO-D	NITI	FP32
MNIST	Common architecture	784-100-50-10	✓*	✓*	✓	✓
MNIST	Individually tuned	Tuned MLP	✓*	✓	✓	✓
CIFAR-10	Common architecture	3072-768-10	✓	✓	✓	✓
CIFAR-10	Individually tuned	Tuned MLP	✓	✓	✓	✓
CIFAR-10	Common architecture	VGG-Small-8	N/A	✓	✓	✓

Note: * The configuration reported by the original authors was used rather than a configuration selected through hyperparameter optimization in this work. N/A stands for "not applicable" as *PocketNN* does not support CNNs.

4.3 Evaluation Protocol

4.3.1 Predictive Performance and Execution Metrics

A common set of metrics was collected across the final experiments to evaluate predictive performance and observed execution behaviour. In this dissertation, *execution efficiency* refers to the observed execution time, epoch time and throughput of an implementation under the specified software and hardware environment. It is not intended as an implementation-independent measure of algorithmic computational efficiency.

Training and test accuracy were recorded after each epoch to analyze the learning evolution. Hyperparameter and architecture selection was based exclusively on validation performance during the tuning stage; test-set measurements were not used to select configurations. Final-epoch test accuracy is used as the main predictive performance metric in the comparative analysis. The maximum test accuracy reached during training is also reported to show how accuracy changed across epochs, but it was not used to select models, architectures or hyperparameters. Since the test set was evaluated after every epoch, the best test accuracy is reported only to describe training behaviour and is not used as the main performance result.

Execution times were measured using `perf_counter()` from Python's `time` module at batch, epoch and complete-experiment levels. For GPU-based implementations, CUDA operations were synchronized before and after measured operations so that the recorded wall-clock intervals included the corresponding GPU computation rather than only asynchronous kernel dispatch.

Batch-level timings, epoch-level metrics and total experiment time were recorded separately for each execution.

Throughput was calculated as the total number of training samples processed across all epochs divided by the total experiment time. Since the total experiment time also includes evaluation and other execution overheads, the reported throughput represents the overall rate at which the complete experimental procedure processed training samples rather than isolated training computation.

The metrics used in the final comparison are summarized in Table 4.8.

Table 4.8: Predictive performance and execution time metrics used in the experiments.

Metric	Description	Unit
Training accuracy per epoch	Classification accuracy obtained on the training set at each epoch	%
Test accuracy per epoch	Classification accuracy obtained on the test set at each epoch	%
Final test accuracy	Test accuracy recorded at the final training epoch; primary predictive performance metric	%
Maximum observed test accuracy	Highest test accuracy recorded across the training trajectory; reported descriptively and not used for model selection	%
Mean epoch time	Mean wall-clock time required to complete one epoch, including training and evaluation	s
Total experiment time	Total wall-clock time required to complete the experiment	s
Throughput	Total number of training samples processed across all epochs divided by total experiment time	samples/s

Each final MLP experiment was trained for 150 epochs, while each final CNN experiment was trained for 100 epochs. Final configurations were executed 10 times from scratch using predetermined seeds, except for *PocketNN*, which was executed once because of its substantially longer CPU execution time. Different seed sets were used for the different frameworks. For configurations with 10 runs, summary values are reported as the mean and standard deviation across runs. Although repeated preliminary executions of *PocketNN* produced the same final accuracy under the tested preliminary configurations, only one final run was performed; consequently, no estimate of run-to-run variability is available for *PocketNN* in the final experiments.

4.3.2 Resource and Energy Measurements

CPU and GPU utilization, RAM usage and GPU memory were sampled every 0.2 s to characterize computational resource usage. CPU and process-memory measurements were collected using `psutil`, while GPU utilization and memory measurements were obtained through NVIDIA NVML. Each resource sample was associated with the corresponding execution stage, such as the training forward pass, backward pass or evaluation. In addition to the sampled values, a baseline measurement and the maximum observed resource usage over the complete run were maintained for subsequent analysis.

The collected resource metrics include system and process CPU utilization, system RAM usage, process resident-set size, GPU utilization, GPU memory activity and GPU memory usage. Peak memory corresponds to the maximum sampled value observed over the complete run. When utilization summaries are reported, the mean is calculated across the relevant samples and P95 denotes the 95th percentile. Process CPU utilization may exceed 100% because a process can use more than one logical CPU core. The computational resource metrics used in the evaluation are summarized in Table 4.9.

Table 4.9: Computational resource utilization metrics used in the evaluation.

Metric	Description	Unit
Peak process RAM	Maximum sampled resident-set size (RSS) of the training process over the complete run	GB
Peak GPU memory	Maximum sampled GPU memory usage over the complete run	GB
Mean process CPU utilization	Mean CPU utilization of the training process during training	%
P95 process CPU utilization	95th percentile of the CPU utilization of the training process during training	%
Mean GPU utilization	Mean GPU utilization during training and evaluation, reported separately	%
P95 GPU utilization	95th percentile of GPU utilization during training and evaluation, reported separately	%

Energy-related metrics were monitored using CodeCarbon[8]. The tracker was started before the training loop and stopped after the experiment so that the complete training process was covered. CodeCarbon reported CPU, GPU and RAM energy and power values, together with total energy and carbon-emission estimates.

However, these values should not be interpreted as direct process-level hardware-energy measurements. In the CodeCarbon version used, CPU-energy reporting depends on access to RAPL energy counters, which was restricted in the experimental environments [6]. RAM energy is estimated using a model rather than measured directly. GPU energy is derived from device-level power measurements and therefore includes consumption that cannot be attributed only to the training process.

As *PocketNN* executes only on the CPU while the other MLP implementations use the GPU and because GPU device-level consumption can be recorded even when a particular process is not using the GPU, the CodeCarbon total-energy and carbon-emission estimates are not used for the MLP cross-framework comparison. In the CNN experiments, all evaluated implementations are GPU-based and were executed in the same environment. Therefore, only device-level GPU energy is used for the energy comparison. This value is interpreted as an estimate that includes baseline device consumption, while CPU and RAM energy are excluded.

To keep the instrumentation consistent across implementations, the same monitoring-library versions were used where applicable: CodeCarbon 2.1.4, `psutil` 5.9.1 and `pynvml` 11.4.1.

4.3.3 Experimental Environments and Methodological Limitations

The final experiments were executed in different hardware environments. All MLP experiments were performed on a remote server through Slurm jobs. Each job used an AMD EPYC 75F3 CPU, an NVIDIA A100 GPU with 40 GB of VRAM and 122 GB of allocated RAM. *PocketNN*

[8] https://codecarbon.io/

executed exclusively on the CPU because its implementation does not support GPU execution, whereas *NITI*, *NITRO-D* and the FP32 baseline were executed using GPU.

Because of limited compute time on the cluster, the final CNN experiments were executed on Kaggle. The available environment provided two NVIDIA Tesla T4 GPUs with approximately 16 GB of VRAM each and 33 GB of RAM, with a maximum session duration of 12 hours. Each CNN run used a single GPU.

These environmental differences limit how the experimental results can be compared. Absolute execution time, memory and energy values from the MLP and CNN experiments should not be compared directly because they were collected on different hardware. Within the MLP experiments, comparisons involving *PocketNN* are also not hardware-equivalent to those of the GPU-based frameworks, since *PocketNN* runs only on the CPU. Therefore, its execution time and resource results reflect the behaviour of the released CPU implementation rather than only differences between the training algorithms.

The experimental design does not control all implementation-level differences. *NITI*, *PocketNN* and *NITRO-D* use different learning algorithms, numerical mechanisms and software implementations, while the FP32 baseline was implemented in PyTorch. The common-architecture experiments control the network topology, but the complete training pipelines are not identical. In the CNN comparison, *NITRO-D* also retains a different activation function and initialization strategy from *NITI* and the FP32 baseline. Hyperparameters were selected independently, with some configurations taken from the original works and others obtained through searches performed in this dissertation. These differences between the implementations also make it difficult to separate the effect of numerical precision from the effect of the training method itself.

Additional measurement limitations also remain. *PocketNN* was executed only once in the final evaluation, so run-to-run variability could not be estimated for its final results. Test accuracy was recorded after every epoch; therefore, the maximum test accuracy observed during training may be optimistic and is reported only as a descriptive result, while final-epoch test accuracy is used as the main test metric. Energy-related measurements are also subject to the CodeCarbon limitations described above, including restricted CPU-energy measurement, model-based RAM energy and device-level GPU energy. CodeCarbon 2.1.4 was used to maintain a consistent monitoring setup across the experiments. This version was selected because *NITI* was initially run in an older Kaggle environment, which was the only environment available at that stage of the experiments. The older Python and package versions limited which monitoring-library versions could be used, so CodeCarbon 2.1.4 was kept for the remaining experiments for consistency.

The results should therefore be interpreted as evidence from the evaluated implementations under partially comparable experimental conditions. They reflect the behaviour of the released and adapted software in the specified environments and do not isolate the effect of numerical precision or the learning algorithm alone. Despite these limitations, the experimental setup provides a more consistent basis for answering RQ3 than the different evaluation settings found in the literature review.

Chapter 5

Results and Analysis

This chapter presents and analyzes the results of the experimental comparisons defined in Chapter 4. The evaluated frameworks are compared in terms of predictive performance, execution efficiency, computational resource utilization, memory usage and energy-related measurements. The analysis is divided into MLP and CNN experiments, following the comparison strategy described in Section 4.2.3.

5.1 MLP Experiments

The MLP experiments compare *PocketNN*, *NITRO-D*, *NITI* and the FP32 baseline on MNIST and CIFAR-10 under the two experimental settings described in Section 4.2.3: a common-architecture comparison and an individually tuned comparison. These settings allow the frameworks to be evaluated both under a shared architectural constraint and using framework-specific configurations.

Each final configuration was trained from scratch for 150 epochs. *NITI*, *NITRO-D* and the FP32 baseline were executed ten times using predetermined seeds, while *PocketNN* was executed once, as described in the methodology.

5.1.1 Common-Architecture Comparison

The common-architecture experiments hold the network architecture constant across the four training approaches for each dataset. The MNIST experiments used the 784-100-50-10 architecture, while the CIFAR-10 experiments used the 3072-768-10 architecture. Although the architectures were fixed, the remaining hyperparameters were selected independently, as the frameworks use different numerical representations and training algorithms.

5.1.1.1 Predictive Performance

Figure 5.1 shows the evolution of training and test accuracy during the common-architecture MLP experiments on MNIST. All approaches converged relatively quickly and remained stable during the later epochs. The FP32 baseline reached the highest final test accuracy, followed by *NITI* and

NITRO-D, while *PocketNN* obtained the lowest final accuracy. *NITI* showed more variation during the first epochs, but its accuracy stabilized as training progressed. Overall, the final-accuracy differences between the four approaches were relatively small on MNIST, as summarized in Table 5.1.

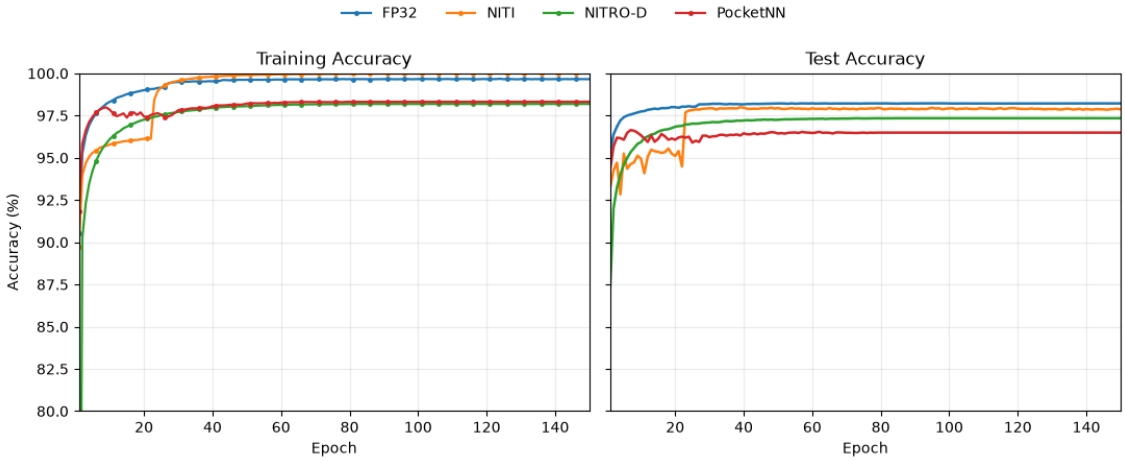


Figure 5.1: Training and test accuracy evolution over epochs on MNIST.

Table 5.1: Final and maximum observed test accuracy by framework for the common-architecture MLP on MNIST. Values are reported as mean ± standard deviation, rounded to three decimal places.

Framework	Final	Maximum observed	n
FP32	98.219 ± 0.069	98.274 ± 0.067	10
NITI	97.870 ± 0.103	98.126 ± 0.045	10
NITRO-D	97.341 ± 0.123	97.365 ± 0.126	10
PocketNN	96.490	96.640	1

Standard deviations are unavailable for PocketNN ($n = 1$).

The differences between the frameworks became more apparent on CIFAR-10. The FP32 baseline reached the highest final test accuracy, followed by *NITRO-D* and *NITI*, while *PocketNN* obtained a much lower final accuracy, as shown in Table 5.2. *NITRO-D* training accuracy continued increasing for most of the experiment, reaching close to 90%, while its test accuracy stabilized at around 55.8%. The FP32 baseline and *NITI* reached final test accuracies of approximately 56.6% and 53.9%, respectively.

PocketNN presented a different training trajectory from the other approaches. Its test accuracy initially increased to approximately 44%, but later decreased and stabilized at around 35% by epoch 50. Its training accuracy showed a similar decrease, indicating that the behaviour was not limited to the test set. PocketNN also showed a larger difference between its maximum observed test accuracy and its final test accuracy than the other approaches. The corresponding training and test trajectories are shown in Figure 5.2.

The final test accuracies of FP32, *NITI* and *NITRO-D* showed relatively small variation across the repeated runs, with standard deviations below 0.73 percentage points.

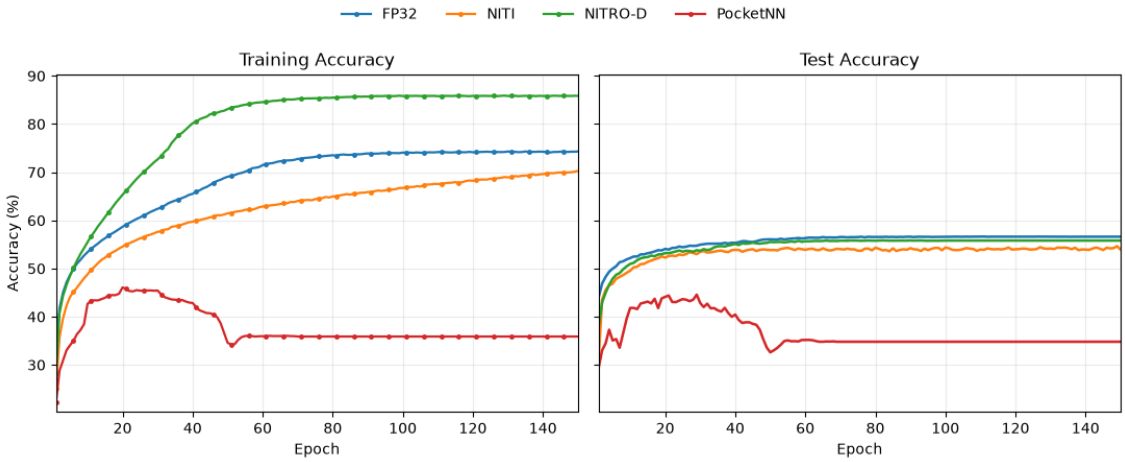


Figure 5.2: Training and test accuracy evolution over epochs on CIFAR-10.

Table 5.2: Final and maximum observed test accuracy by framework for the common-architecture MLP on CIFAR-10. Values are reported as mean ± standard deviation, rounded to three decimal places.

Framework	Final	Maximum observed	n
FP32	56.610 ± 0.262	56.785 ± 0.273	10
NITI	53.938 ± 0.726	55.224 ± 0.185	10
NITRO-D	55.804 ± 0.263	55.970 ± 0.235	10
PocketNN	34.800	44.580	1

Standard deviations are unavailable for PocketNN ($n = 1$).

5.1.1.2 Execution Time

For the MNIST dataset, *PocketNN* required the longest execution time, taking approximately $5.14\times$ as long as the FP32 baseline. *NITI* required approximately $1.98\times$ the FP32 execution time, while *NITRO-D* required approximately $1.36\times$. Mean epoch times followed the same general pattern, as shown in Table 5.3.

Throughput was calculated as the total number of training samples processed across all epochs divided by total experiment time. The FP32 baseline achieved the highest throughput, processing approximately 15063 samples/s, followed by *NITRO-D* and *NITI* with 11101 and 7622 samples/s, respectively. *PocketNN* presented the lowest throughput at 2930 samples/s, consistent with its longer total execution time.

Table 5.3: Total execution time, mean epoch time and throughput for the common-architecture MLP on MNIST. Values for repeated runs are reported as mean ± standard deviation.

Framework	Total execution time	Mean epoch time	Throughput
FP32	597.62 ± 9.74 s	3.93 ± 0.063 s	15063 ± 251 samples/s
NITI	1180.93 ± 14.33 s	7.44 ± 0.093 s	7622 ± 92 samples/s
NITRO-D	810.80 ± 5.68 s	5.35 ± 0.044 s	11101 ± 78 samples/s
PocketNN	3071.74 s	20.38 s	2930 samples/s

For CIFAR-10, the absolute execution times differed from those observed on MNIST. *NITRO-D* increased from 810.80 s on MNIST to 974.52 s on CIFAR-10, while *PocketNN* increased from 3071.74 s, less than an hour, to 86768.76 s, approximately 24 hours. Within the CIFAR-10 common-architecture comparison, *PocketNN* required approximately 405.87× the total execution time of the FP32 baseline. Mean epoch time followed the same pattern as total execution time and frameworks with shorter execution times achieved higher overall throughput, as shown in Table 5.4.

Because throughput includes evaluation and other execution overheads and the MNIST and CIFAR-10 experiments use different datasets and architectures, the throughput values are interpreted only within each corresponding comparison rather than directly across datasets.

Table 5.4: Total execution time, mean epoch time and throughput for the common-architecture MLP on CIFAR-10. Values for repeated runs are reported as mean ± standard deviation.

Framework	Total execution time	Mean epoch time	Throughput
FP32	213.78 ± 4.54 s	1.37 ± 0.031 s	35096 ± 750 samples/s
NITI	806.09 ± 10.74 s	4.84 ± 0.072 s	9306 ± 125 samples/s
NITRO-D	974.52 ± 3.47 s	6.44 ± 0.028 s	7696 ± 27 samples/s
PocketNN	86768.76 s	576.22 s	86 samples/s

5.1.1.3 Resource Usage

Tables 5.5 and 5.6 summarize selected resource-utilization measurements for the common-architecture MLP experiments on MNIST and CIFAR-10, respectively. Peak process RAM and peak GPU memory correspond to the maximum sampled values observed over the complete run. Mean utilization represents the average of the collected samples, while P95 corresponds to the 95th percentile. For FP32, *NITI* and *NITRO-D*, the reported values correspond to the mean across the 10 runs, while *PocketNN* values correspond to a single final run.

For MNIST, *NITI* presented the highest memory requirements, with peak process RAM and GPU memory of 5.27 GB and 2.24 GB, respectively. *NITRO-D* used slightly less GPU memory than FP32 (1.04 GB versus 1.14 GB), but required approximately 1.90× more process RAM. *PocketNN* displayed the lowest process RAM usage and did not use GPU memory, as expected according to its CPU-only implementation.

Mean process CPU utilization was similar across the four implementations. The P95 values differed more, with FP32 reaching 128.03%, compared with 110.90% for *NITI*, 106.80% for *NITRO-D* and 104.50% for *PocketNN*. These values describe CPU usage during training and should be considered together with execution time.

GPU utilization remained relatively low during MNIST training for the three GPU-based frameworks. *NITRO-D* achieved the highest mean training GPU utilization at 27.25%, followed by *NITI* at 24.71% and FP32 at 20.42%. During evaluation, *NITRO-D* again showed the highest mean GPU utilization, reaching 29.64%, while *NITI* decreased to 11.00%.

Table 5.5: Resource utilization across implementations for the common-architecture MLP on MNIST.

Activity	Metric	FP32	NITI	NITRO-D	PocketNN
Complete run	Peak process RAM (GB)	1.21	5.27	2.31	0.51
Complete run	Peak GPU memory (GB)	1.14	2.24	1.04	0.00
Training	Process CPU mean (%)	101.34	107.23	100.89	100.42
Training	Process CPU P95 (%)	128.03	110.90	106.80	104.50
Training	GPU utilization mean (%)	20.42	24.71	27.25	0.00
Training	GPU utilization P95 (%)	24.20	28.70	29.00	0.00
Evaluation	GPU utilization mean (%)	21.92	11.00	29.64	0.00

Note: Process CPU utilization may exceed 100% because a process can use more than one logical CPU core. PocketNN runs exclusively on the CPU, so its GPU-related values are zero.

On CIFAR-10, *NITRO-D* showed the highest peak process RAM usage at 6.56 GB, followed by *NITI* at 5.47 GB. FP32 and *PocketNN* used 1.23 GB and 1.64 GB, respectively. NITI showed the highest peak GPU-memory usage at 2.34 GB, while FP32 and *NITRO-D* required 1.69 GB and 1.63 GB.

Mean GPU utilization during training was 53.05% for *NITRO-D*, 26.73% for FP32 and 20.93% for *NITI*. For *NITRO-D*, this represented an increase from 27.25% on MNIST. During evaluation, *NITRO-D* reached a mean GPU utilization of 69.00%. For process CPU utilization, FP32 presented the highest P95 value at 153.49%. *NITI* showed lower mean and P95 CPU utilization on CIFAR-10 than on MNIST, with its mean decreasing from 107.23% to 87.87% and its P95 from 110.90% to 95.14%. These utilization measurements should be interpreted together with execution time.

Table 5.6: Resource utilization across implementations for the common-architecture MLP on CIFAR-10.

Activity	Metric	FP32	NITI	NITRO-D	PocketNN
Complete run	Peak process RAM (GB)	1.23	5.47	6.56	1.64
Complete run	Peak GPU memory (GB)	1.69	2.34	1.63	0.00
Training	Process CPU mean (%)	106.85	87.87	100.70	99.56
Training	Process CPU P95 (%)	153.49	95.14	106.60	104.30
Training	GPU utilization mean (%)	26.73	20.93	53.05	0.00
Training	GPU utilization P95 (%)	31.90	24.00	54.00	0.00
Evaluation	GPU utilization mean (%)	28.06	6.95	69.00	0.00

Note: Process CPU utilization may exceed 100% because a process can use more than one logical CPU core. PocketNN runs exclusively on the CPU, so its GPU-related values are zero.

Energy-related metrics were also tracked using CodeCarbon 2.1.4. However, they were not used for the MLP cross-framework comparison because the available measurements did not provide a sufficiently comparable estimate across CPU-only and GPU-based implementations. As described in Chapter 4, CPU-energy reporting depended on RAPL access, GPU energy was measured at device level and RAM energy was model-based. In particular, device-level GPU consumption could be recorded even when *PocketNN* itself did not use the GPU. Consequently, total energy and carbon-emission estimates are not analyzed comparatively for the MLP experiments, as they are not reliable enough to establish a fair comparison.

5.1.1.4 Accuracy-Efficiency Tradeoffs

Tables 5.7 and 5.8 summarize the trade-offs between final test accuracy and selected execution and memory measurements, using FP32 as the reference.

On MNIST, *NITI* achieved the closest final accuracy to FP32, with a decrease of 0.349 percentage points, but required approximately 1.98× the execution time, 4.34× the peak process RAM and 1.97× the peak GPU memory. *NITRO-D* showed a final-accuracy difference of 0.878 percentage points, required 1.36× the execution time and 1.90× the peak process RAM, while using slightly less peak GPU memory than FP32. *PocketNN* used the least process RAM and no GPU memory, but had the largest final-accuracy difference and required 5.14× the FP32 execution time.

On CIFAR-10, *NITRO-D* obtained the closest final accuracy to FP32, with a decrease of 0.806 percentage points, but required 4.56× the execution time and 5.35× the peak process RAM, while using slightly less peak GPU memory. *NITI* required 3.77× the execution time and 4.46× the peak process RAM, with a final-accuracy decrease of 2.672 percentage points. *PocketNN* showed the

largest difference, with a decrease of 21.810 percentage points in final accuracy and an execution time more than 400× that of the FP32 baseline.

These results show that reduced-precision implementations did not exhibit a uniform trade-off across predictive performance, execution time and memory usage. In some configurations, final accuracy remained close to FP32, but the measured execution or memory requirements were higher. The observed trade-offs therefore depend on the complete framework implementation and experimental setting.

Table 5.7: Final accuracy and resource ratios relative to FP32 for the common-architecture MLP on MNIST.

Framework	Final accuracy (%)	Accuracy difference	Time ratio	Process RAM ratio	GPU-memory ratio
FP32	98.219 ± 0.069	0.000	1.00×	1.00×	1.00×
NITI	97.870 ± 0.103	−0.349	1.98×	4.34×	1.97×
NITRO-D	97.341 ± 0.123	−0.878	1.36×	1.90×	0.92×
PocketNN	96.490	−1.729	5.14×	0.42×	—

Notes. Accuracy differences are relative to FP32 and expressed in percentage points. Time ratios are based on total execution time, while process RAM and GPU-memory ratios are based on peak memory usage over the complete run and are normalized to FP32. The dash indicates that no GPU-memory ratio is reported for *PocketNN*.

Table 5.8: Final accuracy and resource ratios relative to FP32 for the common-architecture MLP on CIFAR-10.

Framework	Final accuracy (%)	Accuracy difference	Time ratio	Process RAM ratio	GPU-memory ratio
FP32	56.610 ± 0.262	0.000	1.00×	1.00×	1.00×
NITI	53.938 ± 0.726	−2.672	3.77×	4.46×	1.38×
NITRO-D	55.804 ± 0.263	−0.806	4.56×	5.35×	0.96×
PocketNN	34.800	−21.810	405.87×	1.34×	—

Notes. Accuracy differences are relative to FP32 and expressed in percentage points. Time ratios are based on total execution time, while process RAM and GPU-memory ratios are based on peak memory usage over the complete run and are normalized to FP32. The dash indicates that no GPU-memory ratio is reported for *PocketNN*.

5.1.2 Individually Tuned Comparison

The previous comparison kept the network architecture constant across all frameworks. As described in Section 4.2.4.2, a second comparison allowed the architecture and remaining hyperparameters to be selected on a framework-specific basis. The selected architectures are presented in Table 4.5. The aim of this comparison was to evaluate the predictive performance each framework

could achieve when the architecture was included as an optimization variable along with the hyperparameters, with the final configuration selected according to the highest validation accuracy within the defined search space.

Tables 5.9 and 5.10 summarize final test accuracy, total execution time, peak process RAM and peak GPU memory for the individually selected MLP configurations, together with the change in final accuracy relative to the corresponding common-architecture experiment. *PocketNN* does not change between the two comparison settings because the retained MNIST configuration and refined CIFAR-10 configuration were already used in the common-architecture experiments.

On MNIST, the final test accuracy increased for all three GPU-based frameworks. *NITRO-D* showed the largest increase, at 1.005 percentage points, while *NITI* and FP32 had smaller improvements of 0.696 and 0.401 percentage points, respectively. These changes were accompanied by only small increases in peak memory usage. However, *NITRO-D* required approximately 10.63× the execution time of its common-architecture configuration. FP32 required approximately 131 s longer, while *NITI* showed only a small change.

On CIFAR-10, *NITRO-D* and the FP32 baseline increased their final accuracy by 2.227 and 2.804 percentage points, respectively. Both also required longer execution times. The selected *NITI* configuration ran slightly faster than its common-architecture counterpart, but achieved a final accuracy 3.635 percentage points lower. Its run-to-run variation also increased, with the standard deviation rising from 0.726 to 2.117 percentage points.

Table 5.9: Final accuracy and computational results for the individually selected MLP architectures and hyperparameters on MNIST.

Framework	Final test accuracy (%)	Change from common	Total time (s)	Peak process RAM (GB)	Peak GPU memory (GB)
FP32	98.620 ± 0.063	$+0.401$	728.39 ± 21.34	1.23 ± 0.01	1.16 ± 0.00
NITI	98.566 ± 0.059	$+0.696$	1173.97 ± 10.87	5.28 ± 0.01	2.28 ± 0.00
NITRO-D	98.346 ± 0.056	$+1.005$	8618.18 ± 331.77	2.45 ± 0.00	1.14 ± 0.05
PocketNN	96.490	0.000	3071.74	0.51	—

Note. Accuracy changes are relative to the corresponding common-architecture result of the same framework, in percentage points. Positive values indicate an increase in final accuracy. Standard deviations are unavailable for *PocketNN*, which was executed once. The dash indicates that *PocketNN* does not use the GPU.

Table 5.10: Final accuracy and computational results for the individually selected MLP architectures and hyperparameters on CIFAR-10.

Framework	Final test accuracy (%)	Change from common	Total time (s)	Peak process RAM (GB)	Peak GPU memory (GB)
FP32	59.414 ± 0.260	$+2.804$	555.75 ± 10.59	1.25 ± 0.01	1.71 ± 0.00
NITI	50.303 ± 2.117	-3.635	726.50 ± 9.24	5.49 ± 0.01	2.36 ± 0.00
NITRO-D	58.031 ± 0.296	$+2.227$	1520.39 ± 29.93	6.87 ± 0.00	1.70 ± 0.08
PocketNN	34.800	0.000	86768.76	1.64	—

Note. Accuracy changes are relative to the corresponding common-architecture result of the same framework, in percentage points. Positive values indicate an increase in final accuracy. Standard deviations are unavailable for PocketNN, which was executed once. The dash indicates that PocketNN does not use the GPU.

5.2 CNN Experiments

For the CNN experiments, the VGG-Small-8 architecture described in Section 4.2.4.3 was used as the common architecture for *NITI*, *NITRO-D* and the FP32 baseline. *PocketNN* was not included because its implementation does not support convolutional networks.

5.2.1 Predictive Performance

Figure 5.3 shows the evolution of training and test accuracy over the 100 training epochs. All three approaches reached training accuracies above 90%, while test accuracy remained lower. The FP32 baseline produced the highest final test accuracy at 85.843%, followed by *NITI* at 84.391% and *NITRO-D* at 83.141%, as shown in Table 5.11. All three approaches presented relatively small variation across the ten runs, with standard deviations below 0.32 percentage points for final test accuracy.

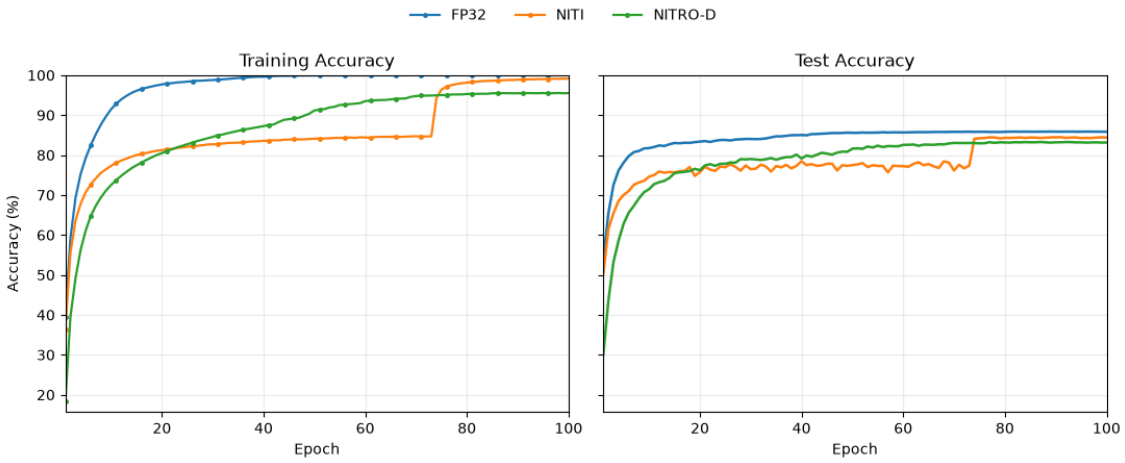


Figure 5.3: Training and test accuracy evolution over epochs on CIFAR-10 for the VGG-Small-8 CNN.

Table 5.11: Final and maximum observed test accuracy by framework for VGG-Small-8 on CIFAR-10. Values are reported as mean ± standard deviation, rounded to three decimal places.

Framework	Final	Maximum observed	n
FP32	85.843 ± 0.314	85.967 ± 0.260	10
NITI	84.391 ± 0.232	84.788 ± 0.174	10
NITRO-D	83.141 ± 0.239	83.534 ± 0.268	10

5.2.2 Execution Time and Resource Usage

The FP32 baseline required the shortest total execution time, completing the 100 epochs in approximately 1 hour and 6 minutes. *NITI* required approximately 2 hours and 7 minutes, while *NITRO-D* required approximately 7 hours and 25 minutes. Relative to FP32, the total execution time ratios were 1.92× for *NITI* and 6.71× for *NITRO-D*. The FP32 baseline also achieved the highest overall throughput, at approximately 1258 samples/s, followed by *NITI* at 655 samples/s and *NITRO-D* at 187 samples/s, as summarized in Table 5.12.

Table 5.12: Total execution time, mean epoch time and throughput for VGG-Small-8 on CIFAR-10. Values are reported as mean ± standard deviation.

Framework	Total execution time	Mean epoch time	Throughput
FP32	3984.55 ± 202.44 s	39.80 ± 2.02 s	1258 ± 62 samples/s
NITI	7640.28 ± 215.06 s	76.04 ± 2.13 s	655 ± 18 samples/s
NITRO-D	26728.03 ± 786.13 s	267.28 ± 7.86 s	187 ± 5 samples/s

Table 5.13 summarizes selected resource-utilization measurements. The FP32 baseline required the least peak process RAM and GPU memory. *NITI* presented the highest peak process RAM usage, at approximately 5.06 GB, while *NITRO-D* used the most peak GPU memory, at 3.58 GB.

Mean process CPU utilization was close to 100% for all three approaches during training. GPU utilization varied more substantially. *NITRO-D* showed the highest mean training GPU utilization at 99.17%, followed by FP32 at 92.82%, while *NITI* had the lowest at 76.20%. During evaluation, the corresponding values were 99.06%, 94.03% and 71.25%, respectively. Despite its higher GPU utilization, *NITRO-D* also had the longest execution time, illustrating that utilization alone does not indicate higher execution efficiency.

Table 5.13: Resource utilization across implementations for VGG-Small-8 on CIFAR-10.

Activity	Metric	FP32	NITI	NITRO-D
Complete run	Peak process RAM (GB)	1.36 ± 0.00	5.06 ± 0.01	1.55 ± 0.03
Complete run	Peak GPU memory (GB)	1.54 ± 0.00	1.97 ± 0.01	3.58 ± 0.12
Training	Process CPU mean (%)	101.16 ± 0.09	104.10 ± 0.16	101.37 ± 0.05
Training	Process CPU P95 (%)	107.91 ± 1.38	109.90 ± 0.05	108.39 ± 0.03
Training	GPU utilization mean (%)	92.82 ± 0.52	76.20 ± 2.21	99.17 ± 0.16
Training	GPU utilization P95 (%)	93.60 ± 0.52	79.40 ± 2.17	100.00 ± 0.00
Evaluation	GPU utilization mean (%)	94.03 ± 0.50	71.25 ± 2.34	99.06 ± 0.07

Note: Process CPU utilization may exceed 100% because a process can use more than one logical CPU core.

5.2.3 Accuracy-Efficiency Tradeoffs

Table 5.14 combines final test accuracy with selected execution, memory and device-level GPU-energy ratios relative to the FP32 baseline. FP32 reached the highest final test accuracy while also requiring the shortest execution time, the least peak process RAM and GPU memory and the lowest measured GPU energy.

NITI achieved a final test accuracy 1.452 percentage points below FP32, while *NITRO-D* differed by 2.702 percentage points. *NITI* required $1.92\times$ the FP32 execution time, $3.71\times$ the peak process RAM, $1.28\times$ the peak GPU memory and $1.89\times$ the device-level GPU energy. *NITRO-D* required $6.71\times$ the execution time, $1.14\times$ the peak process RAM, $2.33\times$ the peak GPU memory and $6.46\times$ the GPU energy.

Between the two low-precision implementations, *NITI* obtained 1.25 percentage points higher final accuracy and required less execution time, GPU memory and GPU energy than *NITRO-D*. *NITRO-D*, however, required considerably less process RAM. The two frameworks therefore exhibited different resource usage profiles even when evaluated with the same CNN architecture. Therefore, although both low-precision approaches kept final accuracy relatively close to FP32, it did not result in lower time, memory or GPU-energy requirements in this experiment.

The GPU-energy ratios followed a pattern similar to the execution time ratios. However, these values represent device-level GPU energy and include baseline device consumption, as discussed in Chapter 4. They should therefore be interpreted as execution-level measurements rather than as isolated measurements of the energy cost of reduced-precision operations.

Table 5.14: Final accuracy and computational ratios relative to FP32 for VGG-Small-8 on CIFAR-10.

Framework	Final accuracy (%)	Accuracy difference	Time ratio	Process RAM ratio	GPU-memory ratio	GPU-energy ratio
FP32	85.843 ± 0.314	0.000	1.00×	1.00×	1.00×	1.00×
NITI	84.391 ± 0.232	−1.452	1.92×	3.71×	1.28×	1.89×
NITRO-D	83.141 ± 0.239	−2.702	6.71×	1.14×	2.33×	6.46×

Notes. Accuracy differences are relative to FP32 and expressed in percentage points. Time ratios are based on total execution time, while process RAM and GPU-memory ratios are based on peak memory usage over the complete run. The GPU-energy ratio considers only the device-level GPU energy estimated by CodeCarbon; CPU and RAM energy are excluded.

5.3 Discussion

The experimental results showed that the evaluated low-precision frameworks can achieve predictive performance close to the FP32 baseline in the evaluated configurations; however, this does not consistently lead to lower execution time or measured resource usage. Across the final comparisons, FP32 obtained the highest mean final test accuracy and the shortest execution time. However, the differences between the frameworks varied with the dataset, architecture and resource considered. These findings complement the literature review by showing how the selected approaches behave in practice when implemented and evaluated under more comparable experimental conditions. Because the evaluated approaches differed in their learning algorithms, numerical mechanisms and software implementations, the observed results illustrate the complete training implementation rather than the effect of numerical precision alone.

5.3.1 Predictive Performance and Configuration Effects

In the common-architecture MLP experiments, *NITI* and *NITRO-D* retained final test accuracy relatively close to the FP32 baseline on MNIST. *NITI* differed from FP32 by 0.349 percentage points and *NITRO-D* by 0.878 percentage points. *PocketNN* showed a larger difference of 1.729 percentage points. On CIFAR-10, the differences became more noticeable. *NITRO-D* remained within 0.806 percentage points of FP32, while *NITI* differed by 2.672 percentage points and *PocketNN* by 21.810 percentage points.

These results indicate that the differences in predictive performance under reduced-precision training are not uniform across frameworks or datasets. In particular, the relatively small differences obtained on MNIST were not similarly observed on CIFAR-10.

The individually selected MLP configurations further showed that architecture and hyperparameter selection can considerably affect the results, changing the observed trade-offs. On MNIST, *NITI* reached a final accuracy of 98.566%, only 0.054 percentage points below the FP32 baseline, while *NITRO-D* improved by 1.005 percentage points relative to its common-architecture result. However, the selected *NITRO-D* configuration also required substantially longer execution time,

up to 10 times more. These results show that framework-specific configuration can alter the balance between predictive performance and computational cost.

The effect of framework-specific configuration was not consistently positive. On CIFAR-10, *NITRO-D* and the FP32 baseline improved their final accuracy relative to the common-architecture experiments, while the selected *NITI* configuration decreased by 3.635 percentage points and exhibited greater run-to-run variation. The hyperparameter searches optimized the maximum validation accuracy observed during 60 training epochs, while the final MLP experiments used 150 epochs. Consequently, the configuration that performed better during the shorter search was not necessarily the one that would perform better after longer training. The selected configurations therefore represent the best results found within the defined search space, rather than the best possible configuration for each framework.

PocketNN presented a different behaviour, specifically on CIFAR-10. Its maximum observed test accuracy reached 44.580%, but its final accuracy decreased to 34.800%. A similar decrease was observed in training accuracy, indicating that the behaviour was not restricted to generalization on unseen data. CIFAR-10 was not among the datasets considered in the original *PocketNN* evaluation and the experiment required a larger input representation and network than its MNIST configuration. The present measurements do not identify the cause of the observed degradation and it should therefore be interpreted as the behaviour of the released and adapted implementation under this experimental setting rather than as a general limitation of DFA or low-precision training.

In the CNN experiments, both integer-only approaches achieved final test accuracies close to the FP32 baseline, with differences of 1.452 percentage points for *NITI* and 2.702 percentage points for *NITRO-D*. *NITI* obtained a lower accuracy than that reported in the original work. However, the final experiments followed a different setup, including no data augmentation, a common VGG-Small-8 architecture and independently selected hyperparameters. Therefore, these results should not be interpreted as failures to reproduce the original works, since reproduction was evaluated separately during the preliminary validation described in Chapter 4.

5.3.2 Execution Efficiency and Resource Trade-offs

Although the accuracy differences relative to the FP32 baseline were relatively small, the low-precision approaches often required considerably longer execution times. In the common architecture MLP experiments, *NITI* and *NITRO-D* required 1.98 and 1.36 times the FP32 execution time on MNIST, respectively. On CIFAR-10, these ratios increased to 3.77 and 4.56. In the CNN comparison, *NITI* required 1.92 times the FP32 execution time and *NITRO-D* required 6.71 times as long.

As discussed in Chapter 3, *NITI* uses custom integer operations together with scaling, rounding and rescaling mechanisms, while *NITRO-D* introduces local-learning operations and framework-specific numerical procedures, such as its own scaling and optimization mechanisms. These adaptations are required to enable reduced-precision training, but they may also introduce additional

computational overhead. The measurements collected in this work do not isolate the cost of individual operations and therefore do not establish which components are responsible for the observed execution time differences.

The FP32 baseline may also benefit from a well-established and more optimized implementation. Therefore, the potential computational advantages of lower-precision arithmetic may not be directly reflected in the observed execution times. These results suggest that the measured efficiency depends not only on numerical precision, but also on the characteristics and optimization of the complete implementation.

Memory usage also depended on both the framework and the type of memory considered. In the common-architecture MLP experiments, *NITRO-D* used slightly less peak GPU memory than FP32, but more process RAM. *NITI* required more process RAM and GPU memory than the FP32 baseline in both MLP datasets. *PocketNN* required no GPU memory because it executes exclusively on the CPU and used less process RAM than FP32 on MNIST, although this advantage was not maintained on CIFAR-10.

The CNN comparison showed a different resource profile. *NITI* required 3.71 times the peak process RAM of FP32 and 1.28 times its peak GPU memory, while *NITRO-D* remained close to FP32 in process RAM but required 2.33 times its peak GPU memory. This shows that reduced precision did not consistently lead to lower memory usage across the different memory metrics. Batch size also needs to be considered when interpreting execution time and memory usage, since it was selected independently for each framework. For instance, in the CNN experiments, FP32 used a batch size of 16, *NITI* used 32 and *NITRO-D* used 64. The larger batch used by *NITRO-D* may have contributed to its higher peak GPU-memory usage, as more samples were processed together. However, memory usage did not increase uniformly with batch size: *NITI* required the most process RAM despite using a smaller batch than *NITRO-D*. Larger batches also reduced the number of batches processed per epoch, but did not lead to shorter execution times than FP32. Furthermore, in the common-architecture CIFAR-10 MLP comparison, *NITRO-D* and FP32 both used a batch size of 32, yet *NITRO-D* required 4.56 times the execution time. Batch size may therefore explain part of the observed trade-offs, but its contribution cannot be separated from the other configuration and implementation differences in these experiments.

Similarly, GPU utilization should not be interpreted independently as a measure of execution efficiency. *NITRO-D* reached approximately 99% mean GPU utilization during CNN training while also having the longest execution time and lowest throughput. This means that although the GPU was active for most of the execution time, it was not processing training samples faster. Therefore, higher GPU utilization did not correspond to higher execution efficiency in this experiment.

The CNN GPU-energy measurements followed the same pattern as execution time. Relative to FP32, *NITI* required 1.89 times the estimated device-level GPU energy and *NITRO-D* 6.46 times. These values were close to their corresponding execution time ratios of 1.92 and 6.71. However, the energy measurements represent device-level estimates and include baseline GPU consumption; they should therefore be interpreted as characteristics of the complete execution

environment rather than as measurements of the energy cost of integer arithmetic itself.

5.3.3 Relationship with the Literature Review

The experimental findings also help put the efficiency results from the literature review into context. The reviewed studies used various forms of evidence, including parameter-storage calculations, analytical estimates, simulations, measurements of individual operations and end-to-end execution measurements. Therefore, the efficiency improvements reported in the literature do not always correspond directly to the execution time and resource usage metrics measured in this work.

For example, *NITRO-D* reports estimated memory and energy reductions relative to FP32 backpropagation, while the present experiments measured memory usage and GPU energy during complete executions. Similarly, *NITI* reports GPU memory reductions together with measurements of selected operations and a partial FPGA implementation. These results were therefore obtained using different measurements and experimental conditions from those used in this work.

The lack of consistent end-to-end reductions in the present experiments does not imply that lower-bitwidth representations cannot reduce storage requirements or that integer arithmetic cannot provide computational advantages on hardware optimized for such operations. Instead, the results show that, in the evaluated implementations, these potential advantages were not sufficient to reduce the complete measured costs relative to the FP32 baseline. Similarly, theoretical savings or improvements observed for individual operations do not necessarily result in the same improvements during complete training executions.

Another difference concerns how training cost is defined. One of the reviewed studies, [15], evaluates resource requirements in relation to the cost of reaching a target accuracy. In contrast, the present experiments use fixed training budgets of 150 epochs for MLPs and 100 epochs for CNNs. The reported execution costs therefore represent complete runs under these fixed budgets and do not estimate the minimum time or energy required for each framework to reach the same accuracy level.

5.3.4 Answer to RQ3

RQ3 asks how the selected low-precision training frameworks compare with each other and with full-precision training in terms of predictive performance, execution efficiency and resource usage when evaluated under more comparable experimental conditions.

Under the evaluated conditions, the FP32 baseline achieved the highest mean final test accuracy in every experimental comparison. Nevertheless, *NITI* and *NITRO-D* achieved predictive performance relatively close to the FP32 baseline in several MLP and CNN configurations, although the magnitude of the accuracy difference depended on the dataset and selected configuration. *PocketNN* also remained relatively close to FP32 on MNIST, but showed substantially lower final accuracy on CIFAR-10.

The reduced-precision implementations did not consistently provide lower observed execution time or memory usage. In fact, the FP32 baseline finished all the evaluated experiments in less time, while memory requirements varied according to framework, dataset and memory type. *NITI* generally required more process RAM, while *NITRO-D* showed different trade-offs between process RAM and GPU memory. *PocketNN* avoided GPU-memory requirements and used relatively little process RAM in some settings, but its CPU-only execution took substantially longer to complete training, particularly on CIFAR-10.

For the CNN experiments, the device-level GPU-energy measurements were also higher than those of the FP32 baseline for both *NITI* and *NITRO-D*. This followed the longer execution times of the low-precision implementations and should be interpreted considering the measurement limitations described in Chapter 4.

Among the low-precision frameworks, no single approach provided the most favourable combination of accuracy, execution time and memory usage across all experiments. Their advantages changed with the dataset, architecture and resource considered.

Overall, the answer to RQ3 is that, in these experiments, low-precision training showed different trade-offs, but not a consistent advantage over FP32. This conclusion applies to the evaluated implementations and fixed training budgets. Reduced precision was able to keep predictive performance close to FP32 in some cases, but it did not consistently reduce execution time, memory usage or measured GPU energy. These findings apply to the evaluated implementations and the number of training epochs used. However, differences in learning algorithms, configurations and software implementations prevent the observed results from being attributed to numerical precision alone.

Chapter 6

Conclusions

This dissertation investigated how neural networks can be trained under reduced-precision constraints and what trade-offs arise regarding predictive performance, computational efficiency and resource usage. The work combined a structured literature review, with a PRISMA-informed study identification and selection process and an empirical comparison of selected publicly available low-precision training frameworks.

The literature review was used to understand how reduced precision is integrated into training, the modifications it requires, its potential benefits and how these are evaluated, while the experimental study examined *NITI*, *PocketNN* and *NITRO-D* relative to an FP32 PyTorch baseline under partially harmonized conditions.

6.1 Summary and Achievement of Objectives

The general objective of this dissertation was to examine existing approaches on low-precision neural network training and evaluate their trade-offs in terms of predictive performance, computational efficiency and resource usage. This was addressed through the literature review presented in Chapter 3 and the experimental evaluation presented and conducted in Chapters 4 and 5.

The first objective was to investigate the numerical representations and precision configurations adopted across the different stages of training. The literature review showed considerable variation in both the numerical representations and bitwidths used by existing approaches. Although 8-bit integer representations are frequently used for weights, activations, gradients and error signals, the reviewed works also employ fixed-point, binary, ternary and logarithmic representations, as well as approaches in which the bitwidth changes during training. In the reviewed works, precision is also not applied uniformly throughout the training process. Accumulation and parameter updates often use wider representations, while some approaches maintain floating-point arithmetic for selected operations such as batch normalization, loss calculation or temporary weight reconstruction. Overall, the review showed that low-precision training cannot generally be described by a single bitwidth, since the precision used varies across training components and

wider representations are kept where additional numerical range or precision is required. Accordingly, for the scope of this dissertation, low-precision training did not require every operation to use the same reduced precision, but required reduced precision to be integrated directly into training without relying on a persistent FP32 master or shadow representation of the complete model.

Regarding the second objective, which was to analyze the adaptations introduced to conventional training procedures to operate under reduced-precision constraints, the literature review identified four recurring types of adaptation: scaling mechanisms, stochastic or related rounding methods, modifications to numerically sensitive operations and changes to the learning mechanism. Scaling mechanisms are used to manage the limited dynamic range of low-bitwidth representations, while stochastic, pseudo-stochastic or randomized rounding helps preserve small gradients and parameter updates that could otherwise be lost. Numerically sensitive operations, such as parameter updates or normalization, are also modified, removed or assigned a different precision. These adaptations are applied either while preserving conventional backpropagation or, in some cases, by replacing it with alternative learning mechanisms such as DFA, LES or Forward-Forward. Overall, these findings show that training under reduced-precision constraints generally requires adaptations to the training procedure itself, rather than only reducing the bitwidth of the values involved.

Regarding the third objective, which was to assess the trade-offs in predictive performance, computational efficiency and resource usage reported in the existing literature, the review showed that low-precision training can reduce memory usage, execution time or energy consumption while maintaining accuracy close to FP32 in some settings. However, the trade-offs varied considerably between the reviewed studies. This variation is influenced by differences in numerical representation, training adaptations, learning algorithms, implementations and hardware platforms. In addition, the reviewed works did not evaluate efficiency in the same way: some reported end-to-end measurements, while others relied on theoretical estimates, simulations or measurements of individual components of training. Overall, the literature shows that low-precision training can reduce resource requirements while maintaining accuracy close to FP32 in some cases, but the trade-offs differ according to the specific approach and experimental setting. Consequently, the literature does not support a single general estimate of the accuracy or efficiency benefits of reduced-precision training.

The fourth objective was to identify frameworks suitable for experimental comparison. *NITI*, *PocketNN* and *NITRO-D* were selected because their implementations were publicly available and represented different approaches to low-precision training.

Finally, regarding the fifth objective, which was to experimentally compare the selected frameworks under more comparable conditions, *NITI*, *PocketNN* and *NITRO-D* were evaluated against an FP32 baseline using common datasets, architectures and metrics whenever possible. The experimental results showed that the low-precision frameworks did not provide a consistent advantage over FP32 across predictive performance, execution time and resource usage. Across the final comparisons, FP32 obtained the highest mean final test accuracy and the shortest execution time.

Reduced precision maintained predictive performance close to FP32 in some experiments, but did not show consistent execution time or resource usage reduction. The results also varied between datasets, architectures and frameworks. No single low-precision approach provided the most favourable combination of accuracy, execution time and memory usage across all experiments. Generally, the comparison showed that the practical trade-offs of low-precision training depend on the complete training approach and experimental setting, including the learning algorithm, software implementation, training configuration, hyperparameters, hardware platform and measurement methodology, rather than on numerical precision alone.

To conclude, these findings allowed the dissertation to achieve its general objective of examining low-precision neural network training approaches and evaluating their trade-offs in predictive performance, computational efficiency and resource usage.

6.2 Limitations

The conclusions of the literature review are limited by the employed search and study-selection process. The results are constrained to the selected databases, search terms and eligibility criteria and may therefore not capture every relevant study on low-precision training. In addition, the review excludes inference-only quantization, conventional quantization-aware training based on persistent FP32 parameter copies and other approaches outside the defined feed-forward training scope. The conclusions should therefore not be interpreted as covering all low-precision machine-learning approaches.

Additionally, the conclusions of the literature review are also limited by differences between the included studies. The reviewed works use different datasets, architectures, training configurations, baselines, hardware platforms and reported metrics, which prevents direct numerical comparison across all approaches. In addition, efficiency is evaluated using different methods, including analytical estimates, simulations, hardware-level evaluations and end-to-end measurements. The review can therefore identify common patterns and summarize the reported trade-offs, but differences in evaluation settings limit how directly the reported results can be compared across studies.

The experimental comparison is also limited by differences between the evaluated implementations. *NITI*, *PocketNN*, *NITRO-D* and the FP32 baseline differ in their learning algorithms, numerical mechanisms, programming languages, computational kernels and software stacks. Although the common-architecture experiments use the same network architecture, the complete training pipelines are not identical. In the CNN comparison, *NITRO-D* also uses a different activation function and corresponding initialization strategy from *NITI* and the FP32 baseline. These differences mean that the observed results cannot be attributed to numerical precision alone, since they also reflect framework-specific implementation and training choices.

Hardware conditions were also not identical across all experiments. The MLP experiments were executed on a server equipped with an NVIDIA A100 GPU, while the CNN experiments were executed on NVIDIA T4 hardware through Kaggle. Within the MLP comparison, *PocketNN*

executes only on the CPU while the remaining implementations also use the GPU. Consequently, absolute measurements should not be compared directly between the MLP and CNN experiment groups and comparisons involving *PocketNN* do not constitute hardware-equivalent execution time comparisons.

The hyperparameter-selection procedure also introduced some limitations. Some configurations reported by the original authors were kept, while others were selected through searches performed in this dissertation, meaning that the frameworks were not tuned under exactly the same procedure. The optimization objective was validation accuracy only. Consequently, when architecture was part of the search, configurations with larger or more complex networks could be selected even if they required more computation or memory. In the fixed-architecture searches, parameters such as batch size could also affect memory usage and execution efficiency, but these factors were not considered by the optimization objective. In addition, the searches used shorter training runs than the final experiments, so the configuration with the highest validation accuracy during optimization was not necessarily the one that would perform best under the final training budget.

PocketNN was executed only once in the final experiments because of its substantially longer CPU execution time, so run-to-run variability could not be estimated for its final results.

Finally, the energy-related measurements have important instrumentation limitations. CPU-energy reporting depended on RAPL access, which was restricted in the experimental environments, while RAM energy was model-based. Total energy and carbon-emission estimates were consequently not used for the MLP cross-framework comparison and only device-level GPU energy was used comparatively for the CNN experiments.

6.3 Future Work

Future work could extend the experimental comparison to additional datasets and network architectures to determine if the trade-offs observed in this dissertation remain consistent across different model sizes and tasks. In particular, architectures used in the original framework publications, such as LeNet-based models and the *NITRO-D* VGG8B configuration, could be evaluated to understand how the results depend on the selected network architecture.

A second direction for future work is to extend the hyperparameter optimization beyond validation accuracy. The search could also consider factors such as network size, execution time, memory usage or energy consumption. This could help identify configurations that achieve a better balance between predictive performance and resource usage, since the configuration with the highest validation accuracy is not necessarily the most efficient one.

Another direction for future work is to include an FP32 baseline executed on CPU. Since *PocketNN* was evaluated on CPU while the remaining implementations used GPU acceleration, a CPU-based FP32 baseline would provide a more comparable reference for interpreting its execution time, memory usage and energy consumption.

More detailed profiling could also help explain the execution time and memory results observed in this work. Measuring the cost of operations such as matrix multiplication or convolution separately from quantization, scaling, rounding and parameter updates could help identify which parts of each framework contribute most to the observed overhead.

Finally, future work could also compare the frameworks based on the cost required to reach a given accuracy, instead of considering only a fixed number of training epochs. Measuring the execution time, memory usage or energy consumption required to reach predefined validation-accuracy thresholds could provide a better view of efficiency when the frameworks converge at different rates. More reliable energy measurements would also strengthen this analysis by reducing the uncertainty of the estimates used in this work.

Bibliography

[1] M. Aczel et al. *Environmental Cost of AI's Energy Use: Carbon, Water and Land Footprints*. Tech. rep. Richmond Hill, Ontario, Canada: United Nations University Institute for Water, Environment and Health (UNU-INWEH), 2026. DOI: 10.53328/INR26RMA002.

[2] Jeremy Bernstein et al. "Learning compositional functions via multiplicative weight updates". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 13319–13330. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/9a32ef65c42085537062753ec435750f-Paper.pdf.

[3] Kit Yan Chan et al. "Deep neural networks in the cloud: Review, applications, challenges and research directions". In: *Neurocomputing* 545 (2023), p. 126327. DOI: 10.1016/j.neucom.2023.126327.

[4] Xi Chen et al. "FxpNet: Training a deep convolutional neural network in fixed-point representation". In: *2017 International Joint Conference on Neural Networks (IJCNN)*. May 2017, pp. 2494–2501. DOI: 10.1109/IJCNN.2017.7966159.

[5] Yu Cheng et al. "Model Compression and Acceleration for Deep Neural Networks: The Principles, Progress, and Challenges". In: *IEEE Signal Processing Magazine* 35.1 (2018), pp. 126–136. DOI: 10.1109/MSP.2017.2765695.

[6] CodeCarbon. *Unable to read Intel RAPL files for CPU power: Permission denied*. https://github.com/mlco2/codecarbon/issues/244. Accessed: 2026-09-12. 2021.

[7] L. Colombo. "Federated On-Device Learning of Integer-Based Convolutional Neural Networks". In: *Proc. Int. Conf. AI-ML Syst., AIMLSystems*. Association for Computing Machinery, Inc, 2025. DOI: 10.1145/3703412.3703423.

[8] L. Colombo, A. Falcetta, and M. Roveri. "TIFeD: a Tiny Integer-based Federated learning algorithm with Direct feedback alignment". In: *ACM Int. Conf. Proc. Ser.* Association for Computing Machinery, 2023. DOI: 10.1145/3639856.3639867.

[9] Matthieu Courbariaux et al. *Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1*. 2016. arXiv: 1602.02830 [cs.LG].

[10] Wen Fei et al. “Latent Weight Quantization for Integerized Training of Deep Neural Networks”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 47.4 (Apr. 2025), pp. 2816–2832. DOI: 10.1109/TPAMI.2025.3527498.

[11] A. Ghaffari et al. “Is Integer Arithmetic Enough for Deep Learning Training?” In: *Advances in Neural Information Processing Systems*. Ed. by Koyejo S. et al. Vol. 35. 2022.

[12] Ramyad Hadidi et al. “Toward Collaborative Inferencing of Deep Neural Networks on Internet-of-Things Devices”. In: *IEEE Internet of Things Journal* 7.6 (2020), pp. 4950–4960. DOI: 10.1109/JIOT.2020.2972000.

[13] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. 2nd ed. Prentice Hall, 1999, p. 117.

[14] Geoffrey Hinton. *The Forward-Forward Algorithm: Some Preliminary Investigations*. 2022. arXiv: 2212.13345 [cs.LG].

[15] Tian HUANG, Tao LUO, and Joey Tianyi ZHOU. “Adaptive Precision Training for Resource Constrained Devices”. In: *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. Nov. 2020, pp. 1403–1408. DOI: 10.1109/ICDCS47774.2020.00185.

[16] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Vol. 37. PMLR, 2015, pp. 448–456.

[17] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. Tech. rep. University of Toronto, 2009. URL: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[18] Yann LeCun et al. “Gradient-Based Learning Applied to Document Recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[19] Timothy P. Lillicrap et al. *Random feedback weights support learning in deep neural networks*. 2014. arXiv: 1411.0247 [q-bio.NC].

[20] Jingxiao Ma, Priyadarshini Panda, and Sherief Reda. “FF-INT8: Efficient Forward-Forward DNN Training on Edge Devices with INT8 Precision”. In: *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. June 2025, pp. 1–7. DOI: 10.1109/DAC63849.2025.11133061.

[21] Warren S. McCulloch and Walter Pitts. “A Logical Calculus of the Ideas Immanent in Nervous Activity”. In: *The Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133. DOI: 10.1007/BF02478259.

[22] Arild Nøkland. “Direct feedback alignment provides learning in deep neural networks”. In: *Advances in neural information processing systems* 29 (2016).

[23] Arild Nøkland and Lars Hiller Eidnes. "Training Neural Networks with Local Error Signals". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, Sept. 2019, pp. 4839–4850. URL: https://proceedings.mlr.press/v97/nokland19a.html.

[24] Matthew J Page et al. "The PRISMA 2020 statement: an updated guideline for reporting systematic reviews". In: *BMJ* 372 (2021). DOI: 10.1136/bmj.n71.

[25] A. Pirillo, L. Colombo, and M. Roveri. "NITRO-D: Native Integer-Only Training of Deep Convolutional Neural Networks". In: *Frontiers in Artificial Intelligence and Applications*. Ed. by Lynce I. et al. Vol. 413. IOS Press BV, 2025, pp. 588–595. DOI: 10.3233/FAIA250855.

[26] M. Riedmiller and H. Braun. "A direct adaptive method for faster backpropagation learning: the RPROP algorithm". In: *IEEE International Conference on Neural Networks*. 1993, 586–591 vol.1. DOI: 10.1109/ICNN.1993.298623.

[27] Frank Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain". In: *Psychological Review* 65.6 (1958), pp. 386–408. DOI: 10.1037/h0042519.

[28] 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, pp. 696–699. ISBN: 0262010976.

[29] Jaewoo Song and Fangzhen Lin. "PocketNN: Integer-only training and inference of neural networks via direct feedback alignment and pocket activations in pure C++". In: *tinyML Research Symposium 2022*. San Jose, California, USA, Mar. 2022.

[30] Maolin Wang et al. "NITI: Training Integer Neural Networks Using Integer-Only Arithmetic". In: *IEEE Transactions on Parallel and Distributed Systems* 33.11 (Nov. 2022), pp. 3249–3261. DOI: 10.1109/TPDS.2022.3149787.

[31] Shuai Wang and Yi Kang. "Gradient distribution-aware INT8 training for neural networks". In: *Neurocomputing* 541.C (July 2023). ISSN: 0925-2312. DOI: 10.1016/j.neucom.2023.126269.

[32] Lu Wei et al. "Advances in the Neural Network Quantization: A Comprehensive Review". In: *Applied Sciences* 14.17 (2024). DOI: 10.3390/app14177445.

[33] Yijie Weng et al. *Comprehensive Overview of Artificial Intelligence Applications in Modern Industries*. 2024. arXiv: 2409.13059 [cs.LG].

[34] S. Wu et al. "Training and inference with integers in deep neural networks". In: *International Conference on Learning Representations*. 2018.

[35] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. 2017. arXiv: 1708.07747 [cs.LG].

[36] Y. Yang et al. “Training high-performance and large-scale deep neural networks with full 8-bit integers”. In: *Neural Networks* 125 (2020), pp. 70–82. DOI: 10.1016/j.neunet.2019.12.027.

[37] Penghang Yin et al. *Quantization and Training of Low Bit-Width Convolutional Neural Networks for Object Detection*. 2017. arXiv: 1612.06052 [cs.LG].

[38] Jiawei Zhao et al. “LNS-Madam: Low-Precision Training in Logarithmic Number System Using Multiplicative Weight Update”. In: *IEEE Transactions on Computers* 71.12 (Dec. 2022), pp. 3179–3190. DOI: 10.1109/TC.2022.3202747.

[39] Shuchang Zhou et al. *DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients*. 2018. arXiv: 1606.06160 [cs.NE].

Appendix A

Appendix

A.1 Database Search Queries

The following queries correspond to the searches performed on May 24, 2026. The syntax was adapted to the search fields and operators supported by each database.

IEEE Xplore

```
(
  ("Document Title":"integer-only" OR "Abstract":"integer-only" OR "Author Keywords
    ":"integer-only")
  OR
  ("Document Title":"integer only" OR "Abstract":"integer only" OR "Author Keywords
    ":"integer only")
  OR
  ("Document Title":"integer training" OR "Abstract":"integer training" OR "Author
    Keywords":"integer training")
  OR
  ("Document Title":"quanti*ed training" OR "Abstract":"quanti*ed training" OR "
    Author Keywords":"quanti*ed training")
  OR
  ("Document Title":"quanti*e training" OR "Abstract":"quanti*e training" OR "
    Author Keywords":"quanti*e training")
  OR
  ("Document Title":"low bit-width" OR "Abstract":"low bit-width" OR "Author
    Keywords":"low bit-width")
  OR
  ("Document Title":"low bit width" OR "Abstract":"low bit width" OR "Author
    Keywords":"low bit width")
  OR
  ("Document Title":"low bitwidth" OR "Abstract":"low bitwidth" OR "Author Keywords
    ":"low bitwidth")
  OR
```

```
  ("Document Title":"low precision training" OR "Abstract":"low precision training"
     OR "Author Keywords":"low precision training")
  OR
  ("Document Title":"low-precision training" OR "Abstract":"low-precision training"
     OR "Author Keywords":"low-precision training")
)

AND
(
  ("Document Title":"training" OR "Abstract":"training" OR "Author Keywords":"
    training")
  OR
  ("Document Title":"train" OR "Abstract":"train" OR "Author Keywords":"train")
)

AND
(
  ("Document Title":"neural network" OR "Abstract":"neural network" OR "Author
    Keywords":"neural network")
  OR
  ("Document Title":"deep learning" OR "Abstract":"deep learning" OR "Author
    Keywords":"deep learning")
  OR
  ("Document Title":"machine learning" OR "Abstract":"machine learning" OR "Author
    Keywords":"machine learning")
  OR
  ("Document Title":"artificial intelligence" OR "Abstract":"artificial
    intelligence" OR "Author Keywords":"artificial intelligence")
)
```

Scopus and Web of Science

```
(
    "integer-only" OR "integer only" OR "integer training" OR "quanti*ed training"
    OR "quanti*e training" OR "low bit-width" OR "low bit width" OR "low bitwidth"
    OR "low precision training" OR "low-precision training"
)
AND
(
    "training" OR "train"
)
AND
(
    "neural network" OR "deep learning" OR "machine learning" OR "artificial
    intelligence"
)
```

ACM Digital Library

```
(
    Title:(
        "integer-only" OR "integer training" OR "quantized training" OR "quantised
    training" OR "low bit-width" OR "low bit width" OR "low bitwidth" OR "low
    precision training" OR "low-precision training"
    )
    OR Abstract:(
        "integer-only" OR "integer training" OR "quantized training" OR "quantised
    training" OR "low bit-width" OR "low bit width" OR "low bitwidth" OR "low
    precision training" OR "low-precision training"
    )
    OR Keywords:(
        "integer-only" OR "integer training" OR "quantized training" OR "quantised
    training" OR "low bit-width" OR "low bit width" OR "low bitwidth" OR "low
    precision training" OR "low-precision training"
    )
)

AND
(
    Title:(training OR train) OR Abstract:(training OR train) OR Keywords:(training
     OR train)
)

AND
(
    Title:(
        "neural network" OR "deep learning" OR "machine learning" OR "artificial
    intelligence"
    )
    OR Abstract:(
        "neural network" OR "deep learning" OR "machine learning" OR "artificial
    intelligence"
    )
    OR Keywords:(
        "neural network" OR "deep learning" OR "machine learning" OR "artificial
    intelligence"
    )
)
```

A.2 Reported Classification Accuracy of Reviewed Low-Precision Training Approaches

Table A.1: Reported top-1 and top-5 classification accuracy (%) of the reviewed low-precision training approaches.

Dataset	Architecture	Work	Top-1	Top-5
MNIST	MLP (784–100–50–10)	PocketNN [29]	96.98	–
	MLP (784–100–50–10)	NITRO-D [25]	97.36 ± 0.23	–
	MLP (784–1024–1024–1024–10)	NITRO-D [25]	98.28 ± 0.08	–
	VGG8B	NITRO-D [25]	99.45 ± 0.05	–
	LeNet-5	NITI [30]	99.1	–
	LeNet-5	WAGE [34]	99.6	–
	MLP (2 hidden layers; 1.79M parameters)	FF-INT8 [20]	94.3	–
Fashion-MNIST	MLP (784–200–100–50–10)	PocketNN [29]	87.7	–
	MLP (784–200–100–50–10)	NITRO-D [25]	88.66 ± 0.46	–
	MLP (784–1024–1024–1024–10)	NITRO-D [25]	89.13 ± 0.41	–
	VGG8B	NITRO-D [25]	93.66 ± 0.40	–
CIFAR-10	MLP (1024–3000–3000–3000–10)	NITRO-D [25]	61.03 ± 0.60	–
	VGG-small 7	NITI [30]	90.7 ± 0.1	–
	VGG8B	NITRO-D [25]	87.96 ± 0.39	–
	VGG-small 8	NITI [30]	91.5 ± 0.2	–
	VGG11B	NITRO-D [25]	87.39 ± 0.64	–
	VGG-small 9	NITI [30]	91.8 ± 0.2	–
	VGG-like[1]	WAGE [34]	93.22	–
	ResNet18	[11]	94.84	–
	ResNet18	FF-INT8 [20]	93.1	–
	ResNet18	LNS-Madam [38]	93.41	–
	ResNet20	APT [15]	92.23	–
	ResNet20[2]	LWQ [10]	91.23 ± 0.27	–
	MobileNet v2	APT [15]	93.96	–
	MobileNet v2[2]	LWQ [10]	92.52 ± 0.21	–
	MobileNet v2	FF-INT8 [20]	91.1	–
	InceptionV3[2]	LWQ [10]	94.47 ± 0.22	–
	Model-L[3] (12-bit primal weight)	FxpNet [4]	88.52	–

Continued on the next page.

Table A.1 continued from the previous page.

Dataset	Architecture	Work	Top-1	Top-5
	Model-L[3] (16-bit primal weight)	FxpNet [4]	89.49	–
	Model-L[3] (24-bit primal weight)	FxpNet [4]	89.70	–
	EfficientNet-B0	FF-INT8 [20]	88.6	–
SVHN	VGG-like[1]	WAGE [34]	98.08	–
CIFAR-100	ResNet110	APT [15]	68.38	–
	ResNet18	[11]	75.38	–
ImageNet	AlexNet[4]	WAGE [34]	48.4	72.2
	AlexNet	NITI [30]	48.3	–
	ResNet18[5] (8-bit)	WAGEUBN [36]	64.79	85.81
	ResNet18[5] (16-bit)	WAGEUBN [36]	67.40	87.63
	ResNet18[2]	LWQ [10]	70.46	–
	ResNet18	[11]	69.25	88.79
	ResNet34[5] (8-bit)	WAGEUBN [36]	67.63	87.70
	ResNet34[5] (16-bit)	WAGEUBN [36]	68.50	87.96
	ResNet50[5] (8-bit)	WAGEUBN [36]	67.95	88.01
	ResNet50[5] (16-bit)	WAGEUBN [36]	69.07	88.45
	ResNet50	LNS-Madam [38]	76.14	–
	ResNet50[2]	LWQ [10]	76.09	–
	MobileNet v2[2]	LWQ [10]	70.23	–
	MobileNet v2	[11]	72.80	90.83

TIFeD and Colombo et al. are not present because their main accuracy results are reported graphically across multiple federated configurations rather than as directly tabulated benchmark values.

[1]VGG-like architecture: $2\times(128C3)$MP2–$2\times(256C3)$–MP2–$2\times(512C3)$–MP2–1024FC–10SSE.

[2]The reported Fei et al. results use the proposed three-segment quantization scheme.

[3]FxpNet [4] evaluates a seven-convolutional-layer architecture in three channel-width variants; only Model-L is reported here because it is the largest variant and the one used for comparison with prior work.

[4]The WAGE ImageNet AlexNet architecture is evaluated without dropout or local normalization.

[5]The number of bits corresponds to the bitwidth of the error propagated between the convolution and Batch Normalization layers.

A.3 Hyperparameter Optimization Details

Table A.2: Hyperparameter search space used in the initial PocketNN CIFAR-10 search.

Hyperparameter	**Search space**	**Type**
Number of layers	$[2, 5]$	Integer
First hidden layer	$[512, 1024]$, step 32	Integer
Subsequent hidden layers	$[64,$ previous layer size$]$, step 32	Integer
Inverse learning rate	$[100, 2000]$	Integer (log scale)
Batch size	$\{16, 32, 64, 128, 256\}$	Categorical
Activation	$\{PocketSigmoid, PocketTanh, PocketLeakyReLU\}$	Categorical
Training epochs	5	Fixed Integer

Table A.3: Hyperparameter search space used in the refined PocketNN CIFAR-10 search.

Hyperparameter	**Search space**	**Type**
Number of layers	$[3, 4]$	Integer
First hidden layer	$[512, 1024]$, step 32	Integer
Subsequent hidden layers	$[64,$ previous layer size$]$, step 32	Integer
Inverse learning rate	$[100, 2000]$	Integer (log scale)
Batch size	$\{32, 64, 128\}$	Categorical
Activation	$\{PocketTanh, PocketLeakyReLU\}$	Categorical
Training epochs	15	Fixed integer

Note: The refined search used 25% of the training data used in the initial search, while keeping the same validation set.

Table A.4: Hyperparameter search space used in the MLP NITRO-D search.

Hyperparameter	**Search space**	**Type**
Batch size	$\{16, 32, 64, 128, 256\}$	Categorical
Inverse learning rate	$[64, 4096]$	Integer (log scale)
Learning-rate amplification factor	$[32, 1024]$	Integer (log scale)
Forward decay inverse	$[1000, 50000]$	Integer (log scale)
Subnet decay inverse	$[1000, 50000]$	Integer (log scale)
Dropout rate	$[0.0, 0.5]$, step 0.05	Float
Activation	See note below	Categorical

Note: The activation options were NITRO Leaky ReLU, Pocket Leaky ReLU, Pocket Tanh, Pocket ReLU, Pocket Bipolar ReLU and Pocket Bipolar Leaky ReLU.

Table A.5: Hyperparameter search space used in the MLP NITI search.

Hyperparameter	**Search space**	**Type**
Batch size	$\{16, 32, 64, 128, 256\}$	Categorical
Initial gradient bit-width	$[5, 8]$	Integer
Second gradient bit-width	$[2, gb_0]$	Integer
Third gradient bit-width	$[2, gb_1]$	Integer
First bit-width transition epoch	$[1, 58]$	Integer
Second bit-width transition epoch	$[gb_epoch_1 + 1, 59]$	Integer
Activation rounding	{deterministic, stochastic, pseudo-stochastic}	Categorical
Error rounding	{deterministic, stochastic, pseudo-stochastic}	Categorical
Gradient rounding	{deterministic, stochastic, pseudo-stochastic}	Categorical
Initialization	{uniform, normal}	Categorical
Dropout	{enabled, disabled}	Categorical

Table A.6: Hyperparameter search space used in the MLP FP32 search.

Hyperparameter	**Search space**	**Type**
Batch size	$\{16, 32, 64, 128, 256\}$	Categorical
Inverse learning rate	$[64, 4096]$	Integer (log scale)
Inverse weight decay	$[1000, 50000]$	Integer (log scale)
Dropout rate	$[0.0, 0.5]$, step 0.05	Float
Activation	{ReLU, Leaky ReLU, Tanh}	Categorical
Initialization	{uniform, normal}	Categorical

Table A.7: Hyperparameter configurations selected for the common-architecture MLP experiments.

Dataset	Framework	Selected hyperparameters
MNIST	NITI	Batch size: 64; gradient bit-widths: $(5,2,2)$; transition epochs: $(22,32)$; activation rounding: stochastic; error rounding: pseudo-stochastic; gradient rounding: stochastic; initialization: uniform; dropout: disabled.
MNIST	FP32 baseline	Batch size: 16; inverse learning rate: 68; inverse weight decay: 1757; dropout rate: 0.05; activation: Leaky ReLU; initialization: normal.
CIFAR-10	NITRO-D	Batch size: 32; inverse learning rate: 356; learning-rate amplification factor: 342; forward decay inverse: 20227; subnet decay inverse: 1074; dropout rate: 0.05; activation: NITRO Leaky ReLU.
CIFAR-10	NITI	Batch size: 128; gradient bit-widths: $(5,2,2)$; transition epochs: $(1,14)$; activation rounding: stochastic; error rounding: pseudo-stochastic; gradient rounding: deterministic; initialization: uniform; dropout: enabled.
CIFAR-10	FP32 baseline	Batch size: 32; inverse learning rate: 305; inverse weight decay: 26670; dropout rate: 0.40; activation: Leaky ReLU; initialization: normal.

Table A.8: Hyperparameter configurations selected for the MLP architecture-search experiments.

Dataset	Framework	Selected hyperparameters
MNIST	NITRO-D	Batch size: 16; inverse learning rate: 78; learning-rate amplification factor: 63; forward decay inverse: 21987; subnet decay inverse: 18104; dropout rate: 0.05; activation: NITRO Leaky ReLU.
MNIST	NITI	Batch size: 64; gradient bit-widths: $(5,3,2)$; transition epochs: $(12,14)$; activation rounding: deterministic; error rounding: pseudo-stochastic; gradient rounding: stochastic; initialization: uniform; dropout: disabled.
MNIST	FP32 baseline	Batch size: 16; inverse learning rate: 69; inverse weight decay: 3636; dropout rate: 0.25; activation: Leaky ReLU; initialization: normal.
CIFAR-10	NITRO-D	Batch size: 64; inverse learning rate: 78; learning-rate amplification factor: 373; forward decay inverse: 16403; subnet decay inverse: 16239; dropout rate: 0.15; activation: Pocket Leaky ReLU.
CIFAR-10	NITI	Batch size: 256; gradient bit-widths: $(5,2,2)$; transition epochs: $(10,15)$; activation rounding: pseudo-stochastic; error rounding: deterministic; gradient rounding: deterministic; initialization: uniform; dropout: disabled.
CIFAR-10	FP32 baseline	Batch size: 16; inverse learning rate: 144; inverse weight decay: 1212; dropout rate: 0.30; activation: Leaky ReLU; initialization: uniform.

Table A.9: Hyperparameter search space used for the NITRO-D CNN experiment.

Hyperparameter	**Search space**	**Type**
Batch size	$\{16, 32, 64, 128, 256\}$	Categorical
Inverse learning rate	$[64, 4096]$	Integer (log scale)
Learning-rate amplification factor	$[32, 1024]$	Integer (log scale)
Forward decay inverse	$[1000, 50000]$	Integer (log scale)
Subnet decay inverse	$[1000, 50000]$	Integer (log scale)

Table A.10: Hyperparameter search space used for the NITI CNN experiment.

Hyperparameter	**Search space**	**Type**
Batch size	$\{16, 32, 64, 128, 256\}$	Categorical
Initial gradient bit-width	$[5, 8]$	Integer
Second gradient bit-width	$[2, gb_0]$	Integer
Third gradient bit-width	$[2, gb_1]$	Integer
First bit-width transition epoch	$[1, 28]$	Integer
Second bit-width transition epoch	$[gb_epoch_1 + 1, 29]$	Integer
Activation rounding	{deterministic, stochastic, pseudo-stochastic}	Categorical
Error rounding	{deterministic, stochastic, pseudo-stochastic}	Categorical
Gradient rounding	{deterministic, stochastic, pseudo-stochastic}	Categorical
Initialization	{uniform, normal}	Categorical

Table A.11: Hyperparameter search space used for the FP32 CNN experiment.

Hyperparameter	**Search space**	**Type**
Batch size	$\{16, 32, 64, 128, 256\}$	Categorical
Inverse learning rate	$[64, 4096]$	Integer (log scale)
Initialization	{uniform, normal}	Categorical

Table A.12: Hyperparameter configurations selected for the VGG-Small-8 CNN experiments.

Framework	**Selected hyperparameters**
NITRO-D	Batch size: 64; inverse learning rate: 365; learning-rate amplification factor: 1011; forward decay inverse: 44396; subnet decay inverse: 1526.
NITI	Batch size: 32; gradient bit-widths: $(5, 2, 2)$; transition epochs: $(22, 26)$; activation rounding: pseudo-stochastic; error rounding: deterministic; gradient rounding: pseudo-stochastic; initialization: uniform.
FP32 baseline	Batch size: 16; inverse learning rate: 69; initialization: normal.