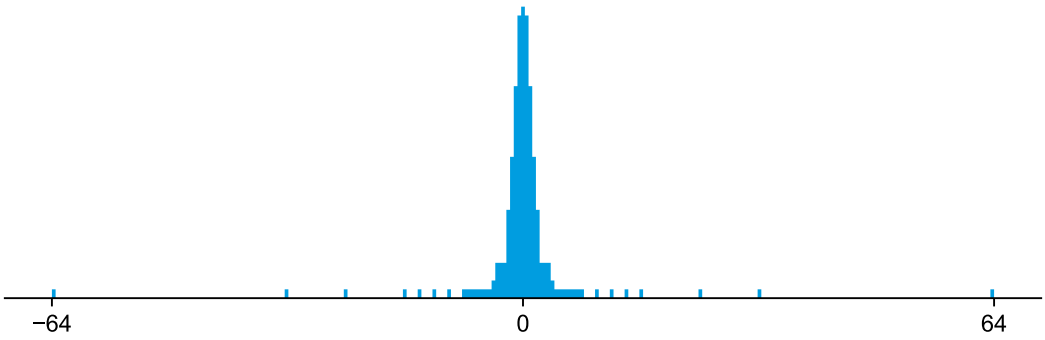
TÉCNICO LISBOA



Deep Learning with approximate computing: an energy efficient approach

Gonçalo Eduardo Cascalho Raposo

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisors: Prof. Nuno Filipe Valentim Roma
Prof. Pedro Filipe Zeferino Aidos Tomás

Examination Committee

Chairperson: Prof. Paulo Jorge Coelho Ramalho Oliveira
Supervisor: Prof. Nuno Filipe Valentim Roma
Members of the Committee: Prof. Gabriel Falcão Paiva Fernandes
Prof. João Paulo Salgado Arriscado Costeira

January 2021

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Agradecimentos

Gostaria de começar por agradecer à minha família mais próxima, mãe e irmãos, que estão sempre presentes e são, para mim, as pessoas mais importantes. Não posso deixar de realçar o meu irmão gémeo Afonso, o meu melhor amigo desde sempre, com o qual partilho várias horas de produtividade e de procrastinação. Refiro também que este período negativamente marcado pela COVID-19 acabou por me favorecer no sentido em que me permitiu regressar a casa, em Santiago do Cacém, e aproveitar quase um ano inteiro com a minha família.

Agradeço também à minha namorada Mariana, com a qual partilho já 3 anos repletos de memórias e de crescimento em vários níveis. Espero continuar esta frase na minha tese de doutoramento.

Um grande obrigado à minha avó Ana, que me acolheu a mim e aos meus irmãos durante os vários anos do nosso percurso académico. Cuidar de 3 jovens não é uma tarefa fácil, mas estudar em Lisboa não teria sido possível sem o lar e companhia que tive em Mem Martins. Obrigado ainda ao meu pai e avó Manuela, que apesar de distantes, me apoiaram sempre desde Sines.

Aos meus amigos mais chegados, obrigado por toda a companhia e espero continuar a viver vários momentos convosco durante muitos mais anos, pois a felicidade é melhor quando partilhada.

Agradeço bastante aos meus orientadores Prof. Nuno Roma e Prof. Pedro Tomás, os quais me deram um apoio essencial na elaboração deste trabalho e me motivaram sempre a ir mais longe. Estendo o agradecimento ao Instituto de Engenharia de Sistemas e Computadores, Investigação e Desenvolvimento (INESC-ID), no qual este trabalho foi elaborado.

Obrigado ao Técnico na sua generalidade, que tem sido literalmente uma 2ª casa e do qual visto a camisola com muito orgulho. O meu percurso nesta grande instituição foi marcado por vários bons e maus momentos que contribuíram para o meu desenvolvimento académico e, sobretudo, pessoal. Guardarei com carinho e saudade os últimos 5 anos.

Não posso deixar de agradecer à Fundação Calouste Gulbenkian (FCG), que desde que ingressei na universidade me apoiou financeiramente através da bolsa de mérito Gulbenkian *mais* e me desafiou sempre a fazer mais e melhor. Agradeço também às pessoas que conheci através da Fundação, todas elas extraordinárias e sempre disponíveis.

Por último, este trabalho foi realizado no âmbito dos projetos UIDB/50021/2020 e PTDC/EEI-HAC/30485/2017 da Fundação para a Ciência e a Tecnologia (FCT).

Resumo

Implementações de aprendizagem profunda mais rápidas e energeticamente eficientes beneficiarão certamente várias áreas, especialmente aquelas com sistemas que possuem limitações de energia e de carga, tais como dispositivos aéreos e espaciais. Sob esta premissa, os formatos de baixa precisão demonstram ser uma forma eficiente de reduzir não só a utilização de memória, mas também os recursos de *hardware* e respetivo consumo energético em aprendizagem profunda. Particularmente, o formato numérico *posit* parece ser um substituto altamente viável para o sistema de vírgula flutuante IEEE, mas ainda pouco explorado para o treino de redes neuronais. Alguns resultados preliminares mostram que *posits* de 8 *bits* (ou menos) podem ser utilizados para inferência e *posits* de 16 *bits* para treino, mantendo a precisão do modelo. O trabalho apresentado visa avaliar o treino de redes neuronais convolucionais com *posits* de precisão inferior ou igual a 16 *bits*. Para tal, foi desenvolvida uma *software framework* que permite utilizar *posits* e *quires* para aprendizagem profunda. Em particular, permitiu treinar e testar modelos com qualquer tamanho de *bits* e ainda com configurações de precisão mista, adequado a diferentes requisitos de precisão. Foi ainda avaliada uma variação do formato *posit* com *underflow*.

Os resultados obtidos sugerem que o *posit* de 8 *bits* consegue substituir o formato simples de vírgula flutuante de 32 *bits* numa configuração de treino mista com *posits* de baixa precisão, sem qualquer impacto na precisão resultante. Além disso, a precisão obtida para testes com *posits* de muito baixa precisão aumentou com a introdução de *underflow*.

Palavras-chave: Formato numérico *posit*, aritmética de baixa precisão, redes neuronais profundas, treino, inferência

Abstract

Faster and energy-efficient deep learning implementations will certainly benefit various application domains, especially those deployed in systems with energy and payload limitations, such as aerial and space devices. Under this premise, low-precision formats have proven to be an efficient way to reduce not only the memory footprint but also the hardware resources and power consumption of deep learning computations. For this purpose, the posit numerical format stands out as a highly viable substitute for the IEEE floating-point, but its application to neural networks training still requires further research. Some preliminary results have shown that 8-bit (and even smaller) posits may be used for inference and 16-bit for training while maintaining the model accuracy. The presented work aims to evaluate the feasibility to train deep convolutional neural networks using posits, focusing on precisions less than or equal to 16 bits. For such purpose, a software framework was developed to use simulated posits and quires in end-to-end deep learning training and inference. This implementation allowed to train and test deep learning models using any bit size, configuration, and even mixed-precision, suitable for different precision requirements in various stages. Additionally, a variation of the posit format able to underflow was also evaluated for low-precision posits.

The obtained results suggest that 8-bit posits can replace 32-bit floats in a mixed low-precision posit configuration for the training phase, with no negative impact on the resulting accuracy. Moreover, enabling posits to underflow increased their testing accuracy for very low precisions.

Keywords: Posit numerical format, low-precision arithmetic, deep neural networks, training, inference

Contents

List of Tables

List of Figures

Nomenclature

Greek symbols

$\beta_1,\ \beta_2$	Exponential decay rates for the moments estimates.
ϵ	Smoothing term that avoids dividing by 0.
η	Learning rate.
γ	Momentum or exponential decay rate.

Roman symbols

A	Sparse matrix for a convolution through a matrix multiplication.
a	Neuron output after activation function.
b	Bias.
C	Number of channels.
E	Expected/average value.
g	Activation function.
H	Height.
K	Kernel size.
L	Loss function.
ℓ	Last layer index. Corresponds to model depth.
m	First moment (mean).
$\hat{m}$	Bias-corrected first moment.
N	(Mini-)batch size.
N_t	Total of samples.
P	Padding.
p	Size of previous layer

q Size of current layer

$\mathcal{R}$ Empirical risk.

S Stride.

t Iteration.

v Velocity/momentum term or second moment (uncentered variance).

$\hat{v}$ Bias-corrected second moment.

W Width.

w Weight tensor.

x Input sample.

y Target output.

$\hat{y}$ Model output.

z Neuron output before activation function.

Subscripts

i Current layer neuron index.

j Previous layer neuron index.

k Kernel/weight index.

m Next layer neuron index.

Q Accumulating with th quire format from the most recent Posit Standard.

q Accumulating with the quire format.

u Underflow is enabled for the posit format.

w Window.

in Input.

out Output.

Superscripts

(n) Sample index.

l Layer index.

Abbreviations

1D 1-dimensional

2D 2-dimensional

3D 3-dimensional

Adam Adaptive Moment Estimation

AI Artificial Intelligence

ANN Artificial Neural Network

ASIC Application-Specific Integrated Circuit

CNN Convolutional Neural Network

CNTK Microsoft Cognitive Toolkit

CPU Central Processing Unit

DL Deep Learning

DNN Deep Neural Network

es exponent size

FC Fully Connected

FCNN Fully Connected Neural Network

FP Floating-Point

FPGA Field-Programmable Gate Array

FPU Floating-Point Unit

gprof GNU Profiler

GPU Graphics Processing Unit

IEEE 754 IEEE Standard for Floating-Point Arithmetic

ILSVRC ImageNet Large Scale Visual Recognition Challenge

LR Learning Rate

maxpos maximum positive number

minpos minimum positive number

MSc Master of Science

MSE Mean Squared Error

MTL Matrix Template Library

NAG Nesterov Accelerated Gradient

NaN Not a Number

NaR Not a Real

nbits number of bits

NGA Next Generation Arithmetic

NLL Negative Log Likelihood

NLP Natural Language Processing

NN Neural Network

PPU Posit Processing Unit

ReLU Rectified Linear Unit

RMSprop Root Mean Square Propagation

SGD Stochastic Gradient Descent

SIMD Single Instruction, Multiple Data

TanH Hyperbolic Tangent

TPU Tensor Processing Unit

VGG Visual Geometry Group

Chapter 1

Introduction

Contents

1.1 Motivation

Humankind has for long been curious about understanding the mind and what it means to be intelligent. Research about modern Artificial Intelligence (AI) began mid-1950s [1] and the Perceptron algorithm was introduced in 1958 [2], which then gave rise to the research about Neural Networks (NNs) [3]. Around 1960, different complexity cells were found in the cat's visual cortex, which later inspired DNN architectures [3]. Quoting Lex Fridman, "the best way to understand the mind is to build it" [4].

As mentioned by Thompson [5], in the past, as NNs got increasingly complex, the available computing power was starting to limit its growth. It was clear that deeper NNs were necessary to improve performance, but the technology of the 1960s could not deliver it [6]. This would become one of the reasons for what was later called AI Winter, which was only escaped after decades of improvement in hardware performance.

Nowadays, DL is one of the hottest topics in research, spanning across multiple scientific areas. It has shown human-like performance, or even better, in problems such as: image classification [7], object detection [8], machine translation [9], etc. Moreover, due to the digitization of the world, there are now large amounts of data that can be used to train deeper, more flexible, and general models [5].

To illustrate how DL algorithms are becoming more complex and more expensive, take for example the recently announced model GPT-3, from Open AI [10]. This model is composed of 175×10^9 parameters [10], which contrasts to the human brain, with 100×10^9 neurons and a total number of synapses estimated to be between 10^{14} and 10^{15} [1]. GPT-3 shows a strong performance on various Natural Language Processing (NLP) datasets and is even able to generate articles that evaluators have difficulty distinguishing from human written ones [10].

Although the brain internal structure may not be directly comparable to an Artificial Neural Network (ANN), one may wonder what performance level will be achieved when DNN models are implemented with as many parameters as the human brain. However, some may claim that DL is reaching its limits [11], since hardware performance is not being able to follow the DL growth, as shown in Figure 1.1 [5]. As it happened in the past, to achieve better performance in DL, improving the modern computing hardware may be more important than advances in algorithms [3, 12].

Most computations that are done in DL use IEEE 754 single-precision floating-point values [13], commonly known as float. However, recent research has shown that it is possible to achieve similar or comparable precision with smaller floating-point values [14, 15], or even with other data types, as mentioned in [16]: fixed-point [17], 8-bit integer [18], binary representations [19], etc. This alternative data formats may reduce the memory footprint and energy consumption by operation.

In 2017 [20], Gustafson proposed a new data type named posit, designed as a direct drop-in replacement for float that provides a larger dynamic range, higher accuracy, and simpler hardware and exception handling. Moreover, a Posit Processing Unit (PPU) takes less circuitry and uses less power than an IEEE float Floating-Point Unit (FPU) [20], which can be exploited in deep learning applications to obtain superior performance for a smaller cost.

Some research regarding the use of posits on DL has already been made, firstly about inference and,

more recently, about training. The obtained results show that when using low-precision 8-bit posits, one can achieve a better accuracy than other low-precision formats and comparable to 32-bits floats [21].

Under these premises, this thesis focuses on using low-precision posits (simulated via software) for end-to-end DNN training and inference, namely, its use in known Convolutional Neural Networks (CNNs) models. Moreover, it proposes a mixed precision configuration that allows to perform most of the computations using posits of 8 bits or less. Faster computations using 8-bit posits instead of 32-bit floats would result in much lower memory and power consumption [22]. As part of this work, a framework for DL with NNs was developed from scratch, giving the reader a better insight on the operations behind DNNs and on the posit numerical format.

If the models are able to achieve a similar performance using posits instead of 32-bit floats, the energy efficient nature of the posit would certainly benefit several real-time applications. One of the earliest evaluations of posits for DL focused in autonomous driving [23], but other energy efficient applications may include computer vision tasks performed by small satellites or drones, whose period of operation is frequently limited by the size of the battery on-board. Another interesting characteristic of posits is that they do not overflow nor underflow, so they may prove very useful for calculations prone to numerical instabilities, such as attitude estimation and non-linear control.

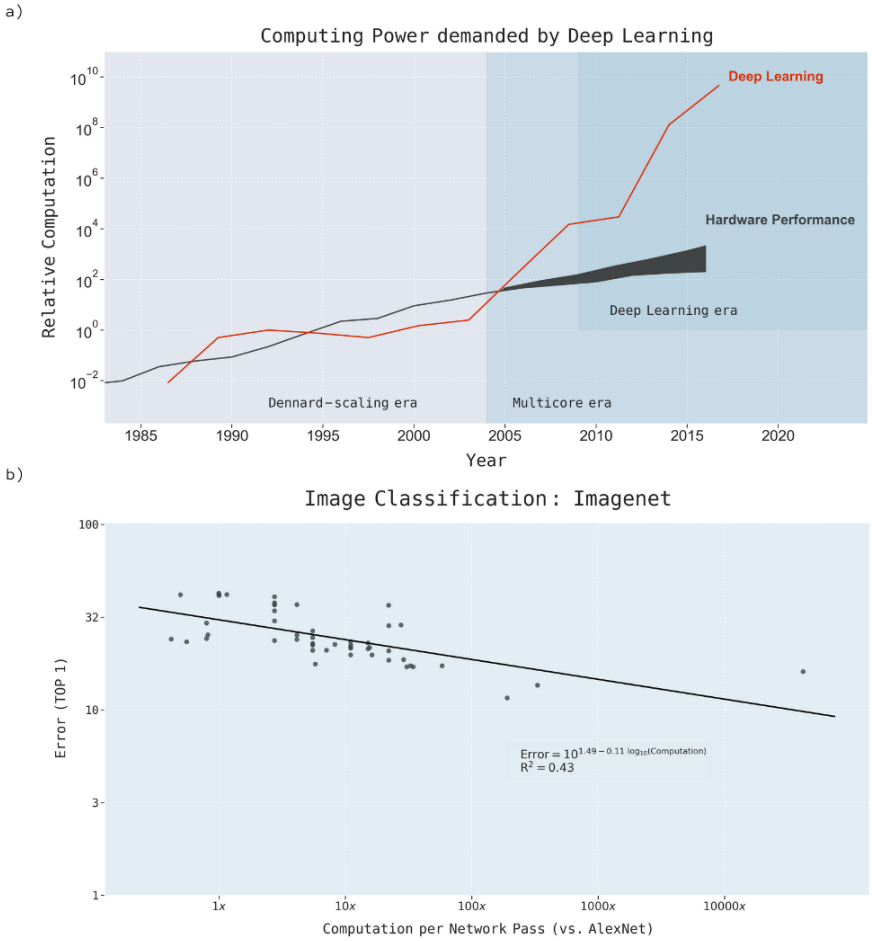


Figure 1.1: Analysis of the growth of the computing power demanded by DL against the hardware performance in (a), and the performance of various image classification models against the number of computations (normalized to the CNN AlexNet) in (b), as presented by Thompson et al. in [5] (2020).

1.2 Objectives

The main objective of this dissertation is to evaluate the use of posits in conventional DNNs, both for inference and training. Posit is a data format that was introduced only in 2017 [20] and there is still much ongoing research regarding its applications and performance. Therefore, the starting point of this work was to get familiar with this novel data format.

Since there was no hardware implementation of Posit Arithmetic for DNNs during most of the time of this research, a software implementation was used to study its application in this field. In fact, some preliminary research about using posits for DNN inference and some incipient studies about training were available, however, deeper NNs such as conventional CNNs remained unexplored at the beginning of this thesis. Moreover, while many studies show that 16-bit posits can replace 32-bits for DNN training, smaller precisions fail to converge.

Thus, the main objectives of this thesis can be summarized as:

- Train DNNs with posit precisions smaller than 16-bits;
- Evaluate the performance of CNN models with the posit format;
- Devise techniques to improve the accuracy achieved by models trained with low-precision posits (e.g. exploit uneven precision requirements in different layers and/or stages of DL);
- Discuss how the most recent version of the Posit Standard [24] compares to the available one [25].

It is worth noting that the results obtained with this work will show how posits compare to floats or other formats in terms of accuracy in DNNs applications. Some takeaways about their performance and energy consumption may be estimated when combined with some existing studies about posits and that topic, but solid conclusions will require a hardware implementation capable of replacing the software simulation of posit arithmetic employed in this work.

1.3 Original Contributions

The original contributions made during this dissertation can be presented as follows:

- Proposal of new techniques to improve the performance achieved by DNN using posits (e.g. mixed precision, underflow, etc.);
- An open source framework[1] for DL with NN using the posit format:
  - The posits are simulated with an existing library that supports any posit configuration;
  - Supports mixed precision and the accumulation of exact sums of products using quires;
  - Developed in C++ and using C++11 parallel threads to increase performance;
  - Designed with an API similar to the popular framework PyTorch.

[1] Available at: `https://github.com/hpc-ulisboa/posit-neuralnet`

- A comprehensive analysis of training and inference implementations with a low-precision posit format:
  - How the chosen posit configuration affects the training of DNNs;
  - Post training quantization inference with {3..8}-bit posits;
  - Results of conventional CNNs implemented with 8-bit posits.
- First known analysis of the most recent Posit Standard and its usage for DNNs.

The contributions of this thesis have been partially submitted for publication in:

[26] G. Raposo, P. Tomás, and N. Roma. PositNN: Training Deep Neural Networks with Mixed Low-Precision Posit. In *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Oct. 2020. (submited and under review)

1.4 Thesis Outline

This Master of Science (MSc) dissertation is structured in multiple chapters, each addressing a relevant topic for this research work:

- Chapter 2 aims to give the reader a brief introduction to DL, numerical formats, and the use of low-precision arithmetic in DNNs. In what concerns DL, NNs are explained as well as the main layers used. Then, some important DNNs architectures, datasets, and frameworks are presented to later evaluate this work. It is followed by an overview of existing numerical formats, namely: floating-point, fixed-point, posit, and variations of posit. Regarding the posit format, a brief inspection of various implementations is presented. At last, the most relevant literature about DL with low-precision and, specifically, with posits is summarized.
- Chapter 3 covers the implementation of a DL framework that is able to use posits. The main decisions and characteristics of its development are presented, such as: how posit variables are manipulated with multidimensional arrays, the implementation of the forward and backward propagation of the most relevant layers, and how it was optimized with multithreading.
- Chapter 4 gives a better insight on how using posit arithmetic affects DNN training. To do so, various results of a CNN trained using the proposed DL framework for posits are presented. Moreover, some techniques to maintain the model accuracy while using low-precision posits are described and evaluated, such as: accumulating with quires, using mixed precision, improving the operations accuracy, and allowing posits to underflow.
- Chapter 5 uses the acquired knowledge regarding DL with posits and showcases the best results achieved with the proposed DNN framework on some conventional CNNs. This results are divided into training (fully implemented with posits) and inference (training done with floats and quantization to posits). The considered datasets were MNIST, Fashion MNIST, CIFAR-10, and CIFAR-100,

which were trained with the models LeNet-5 and a variation of CifarNet. From those results, a brief discussion about the most recent Posit Standard is presented. Furthermore, the framework performance is evaluated in terms of the speedup provided by parallelization.

- Chapter 6 summarizes the developed investigation by recalling the implemented framework, its features, and achievements. At last, considerations about the future work of deep learning with low-precision posits and some relevant remarks are presented.

Chapter 2

Background

Contents

To define the background behind the implementation of deep learning with approximate computing, it is important to first understand its two main topics: deep learning and numerical formats. For that purpose, this chapter will start with a brief overview about DL and how it can be used to train and evaluate different models. Then, some important DNNs, datasets, and frameworks will be presented, allowing for a better evaluation of this work. Most data used in DL is mainly composed of real numbers so it is important to understand how they are stored digitally. In this sense, various numerical formats will be exposed, as well as the novel posit format [20]. This chapter will finish with a brief survey of the research that has been done regarding DL using low-precision formats and, more specifically, posits.

2.1 Deep Learning

Deep learning is a specific subfield of machine learning – models that learn useful representations from input data. In DL, the models are structured in what is called an Artificial Neural Network (or simply, Neural Network), which is a biologically inspired structure that received its name due to the way it relates a given input to a generated output. The unit element of a NN is the neuron – a node that is connected to other nodes through certain operations, resembling biological neurons connected through synapses. A layer consists of a group of neurons that are connected in a distinctive way, which may also be characterized by a set of parameters used for the operations performed between them. In ANNs, multiple layers are stacked in sequence to perform a series of operations to the input.

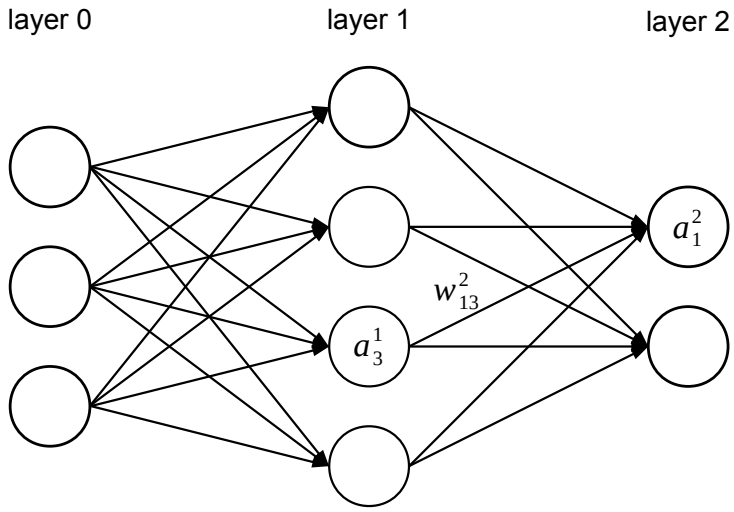


Figure 2.1: Example of a 2-layer neural network of arbitrary dimensions.

Figure 2.1 shows an example of a neural network. Each circle represents a neuron and the arrows between them are their connections. Layer 0 is the input layer, layer 1 is an intermediate layer, and layer 2 is the output layer. One relevant characteristic of a model is its depth ℓ, which corresponds to the number of layers that form that NN. Typically, the input layer does not count for the number of layers of the network, therefore, the depth of this model is $\ell = 2$.

The general mathematical formulation of a neuron i in a layer $l > 0$ is

$$z_i^l = b_i^l + \sum_{j \in \text{previous layer}} w_{ij}^l a_j^{l-1}, \tag{2.1a}$$

$$a_i^l = g\left(z_i^l\right), \tag{2.1b}$$

where z_i^l is an intermediate output, w_{ij}^l are the weights that relate the output of previous neurons a_j^{l-1} to the current one a_i^l, b_i^l is a bias, and g is a non-linear activation function. The mathematical theory is relatively simple, which, in conjunction to the ability of using many successive layers, allows the model to learn quite complex relationships and achieve state of the art performance in multiple tasks.

Deep learning refers to deeper NN architectures, with lots of parameters that are usually trained on large datasets. The large flow of data and relatively simple formulation makes it "a hands-on discipline in which ideas are proven empirically more often than theoretically" [27].

The present work focus on a branch of machine learning algorithms called supervised learning, which is also the most common one. This branch consists of algorithms where the model learns how to map input data to known targets (labeled data) and it is mostly used in classification and regression problems. Nonetheless, there are other branches such as: unsupervised learning, self-supervised learning, and reinforcement learning [27].

2.1.1 Inference and Training

The ultimate objective of DL is to obtain a meaningful output after applying a model to a given input. This process is referred to as *inference*. Usually, the output of the model ($\hat{y}$) corresponds to the activations of the last layer (a^ℓ), but it may also be the result of an additional calculation after the last layer (e.g. in image classification problems, the activations of the last layer resemble probabilities and $\hat{y}$ will correspond to the index of the neuron with the highest probability).

Conversely, the process of updating the network parameters to achieve a better performance is referred to as *training* and it is no more than a minimization of a loss/cost function. Training is, therefore, more computationally demanding when compared to inference, since it also performs an inferring step, followed by the optimization of the parameters, in function of the obtained results. It is also important to mention that, while training aims to minimize a chosen loss function (e.g. cross entropy), the model performance (e.g. accuracy) might not improve correspondingly, since the two might not necessarily measure the same thing [28]. Moreover, this optimization occurs for the data used during the training phase, which may differ from that used in other evaluations and/or result in a model that fits the training data too well at the expense of generalization – overfitting problem.

Considered a labeled training set with a total of N_t samples, $\mathcal{T} = \{(x^{(n)}, y^{(n)}),\ n = 1, \ldots, N_t\}$, where $x^{(n)}$ are the inputs and $y^{(n)}$ the target outputs. The most popular algorithm to train NNs is the gradient descent [29–31]. This algorithm is a way to minimize the empirical risk, defined as

$$\mathcal{R} = \frac{1}{N} \sum_{n=1}^{N} L\left(y^{(n)}, \hat{y}^{(n)}\right), \tag{2.2}$$

where N is the number of considered samples, L is a loss function that measures how different the output values are from the target ones, and $\hat{y}^{(n)}$ is the network output for the input $x^{(n)}$. This is achieved by updating the model weights in the opposite direction of the gradient of $\mathcal{R}$:

$$w(t+1) = w(t) - \eta \left.\frac{\partial \mathcal{R}}{\partial w}\right|_{w(t)}, \tag{2.3}$$

with $w(t)$ being the model parameters at step t and η a factor referred to as learning rate/step.

There are multiple variants of the gradient descent, differing in the number of samples used to calculate the accumulated gradient[2]. Batch gradient descent is computed based on the entire training set at once ($N = N_t$), which can be very slow for large datasets. On the other hand, in an online gradient descent approach the parameters are updated based on only one sample ($N = 1$), resulting in high fluctuations of the empirical risk function. Mini-batch gradient descent is a mix of the previous two and it is the most common on deep learning frameworks. For this variant, the training set is divided into various subsets that are used in each parameter update ($N =$ batch size), such that the gradient may be calculated as

$$\frac{\partial \mathcal{R}}{\partial w} = \frac{1}{N} \sum_{n=1}^{N} \frac{\partial L^{(n)}}{\partial w}. \tag{2.4}$$

The last term of Equation (2.4) may be calculated using the chain rule, as explained in [31, 32]. For a given input sample (omitting the superscript (n)) and a weight w_{ij}^l, it can be generally calculated as

$$\frac{\partial L}{\partial w_{ij}^l} = \frac{\partial L}{\partial z_i^l} \frac{\partial z_i^l}{\partial w_{ij}^l} = a_j^{l-1} \frac{\partial L}{\partial z_i^l}. \tag{2.5}$$

The term $\frac{\partial L}{\partial z_i^l}$ may be expanded using the chain rule and the derivatives from Equations (2.1a) and (2.1b):

$$\begin{cases} \dfrac{\partial L}{\partial z_i^l} = \dfrac{\partial L}{\partial a_i^l} \dfrac{\partial a_i^l}{\partial z_i^l} \\ \dfrac{\partial L}{\partial a_i^l} = \displaystyle\sum_{m \in \text{next layer}} \dfrac{\partial L}{\partial z_m^{l+1}} \dfrac{\partial z_m^{l+1}}{\partial a_i^l} \end{cases} \Rightarrow \begin{cases} \dfrac{\partial L}{\partial z_i^l} = \dfrac{\partial L}{\partial a_i^l} g'\left(z_i^l\right) & \text{for all layers,} \\ \dfrac{\partial L}{\partial a_i^l} = \displaystyle\sum_{m \in \text{next layer}} \dfrac{\partial L}{\partial z_m^{l+1}} w_{mi}^{l+1} & \text{for all layers except last.} \end{cases} \tag{2.6}$$

Note that, if the activation function is the identity function, then $\frac{\partial L}{\partial z_i^l} = \frac{\partial L}{\partial a_i^l}$. The sum used to calculate this last term refers to the next layer, so it is usual to implement the computation of Equation (2.6) backward, from the last layer to the first. In practice, each layer receives $\frac{\partial L}{\partial a_i^l}$, referred to as error or output gradient, and computes the propagated error $\frac{\partial L}{\partial a_j^{l-1}}$. Hence, the errors are propagated through the network in the backward direction, making up an algorithm known as backpropagation. The presented formulation was based on a network with generalized layers. Naturally, the implementations will specialize differently depending on the layer type.

[2]The number of samples per iteration, or batch size, is not to be confused with the number of epochs, which is the number of passes through the entire training dataset.

2.1.2 Main Deep Learning Layers and Functions

Instead of the representation of a neural network illustrated in Figure 2.1, a more common approach is to think of it as block diagram, where each block will represent the function performed by each layer. Therefore, for these diagrams, the boxes represent the layers and the arrows correspond to their inputs and outputs, whose dimensions will depend on the number of neurons that the layers connect (numbers bellow the arrows). Figure 2.2 illustrates the block diagram representation of the same 2-layer NN.

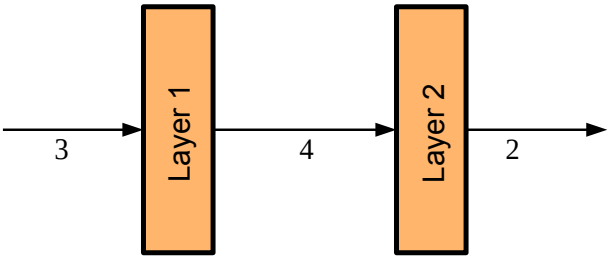


Figure 2.2: Block diagram of the 2-layer neural network example of Figure 2.1.

When structuring a neural network, the choice of the layers and functions depends a great deal on the type of problem to solve. Next, it will be presented the most common layers and functions used in DL classification problems.

Linear Layer

The linear layer, also referred to as Fully Connected (FC) or Dense layer, is equivalent to the generalized layer used for the NN in Figure 2.1. It takes an input of size p and transforms it into an output of size q through a matrix multiplication. The connections between input and output neurons are organized in a weight matrix of size $q \times p$ plus an optional bias vector of size q,

$$w_{ij} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1p} \\ w_{21} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ w_{q1} & \dots & \dots & w_{qp} \end{bmatrix} \quad \text{and} \quad b_i = \begin{bmatrix} w_{10} \\ w_{20} \\ \vdots \\ w_{q0} \end{bmatrix}. \tag{2.7}$$

By considering this formulation, the forward propagation, weight gradient, and backward propagation may be computed as in Equations (2.1a), (2.5) and (2.6), respectively, where the sums of products may be computed as matrix multiplications.

Convolutional Layer

The convolutional layer is an important building block of CNNs, characterized by a convolution operation applied over an input data, corresponding to the implementation of a filter. This convolution may be performed in any dimension, but since this thesis focus on the image classification task, the 2D and 3-dimensional (3D) implementations will be presented. One benefit of the convolutional layer over a fully connected layer is the ability to capture spatial and temporal dependencies, since the input does

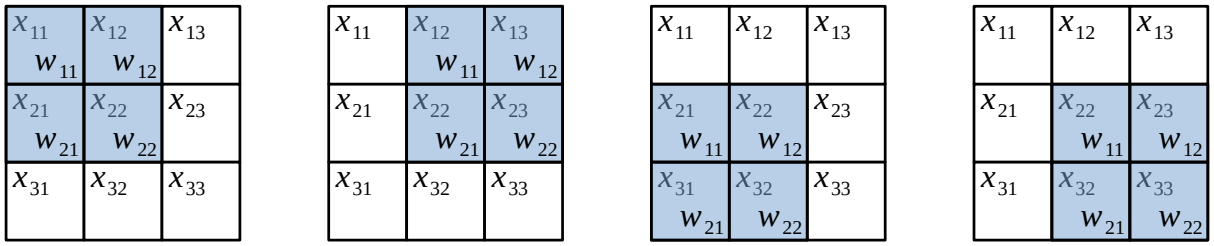
not need to be flattened. Multiple convolutional layers may be applied to an image, each extracting a different level of features that result in a meaningful representation of the data. This is inspired by the organization of the visual cortex, where different cells fire to different properties of the visual input [3].

By definition, the convolution is the result of sliding a filter/kernel over an input sample, computing the product of the input and the filter in the overlapped region (window), and summing for each filter position. However, most DL frameworks actually compute the cross-correlation but call it convolution, since the latter first flips the kernel before sliding it over the input sample. Nonetheless, the cross-correlation avoids that additional step of flipping the kernel and the results are equivalent (as long as the backpropagation is also consistent). Therefore the convolution (actual cross-correlation) is computed as:

$$z_{i_2 i_1} = \sum_{\substack{k_2=1 \\ j_1=i_1+k_1-1 \\ j_2=i_2+k_2-1}}^{H_w} \sum_{k_1=1}^{W_w} x_{j_2 j_1} w_{k_2 k_1} = x * w, \tag{2.8}$$

where $z_{i_2 i_1}$ is the convolution output, H_w is the height of the window/kernel, W_w is the width, $x_{j_2 j_1}$ is the input, $w_{k_2 k_1}$ is the filter, and $*$ represents the convolution operator.

As an example, assume an input sample x of size 3×3 and a filter w of size 2×2. The filter slides over the image as depicted in Figure 2.3. Hence, the result of the convolution, may be computed as in Equation (2.9a), which can be equivalently calculated by a matrix multiplication as in Equation (2.9b).



(a) z_{11} (b) z_{12} (c) z_{21} (d) z_{22}

Figure 2.3: Overlaps of image and kernel (windows) that originate the convolution output z.

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} * \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} = \begin{bmatrix} x_{11}w_{11} + x_{12}w_{12} + & x_{12}w_{11} + x_{13}w_{12} + \\ +x_{21}w_{21} + x_{22}w_{22} & +x_{22}w_{21} + x_{23}w_{22} \\ x_{21}w_{11} + x_{22}w_{12} + & x_{22}w_{11} + x_{23}w_{12} + \\ +x_{31}w_{21} + x_{32}w_{22} & +x_{32}w_{21} + x_{33}w_{22} \end{bmatrix}, \tag{2.9a}$$

$$\underbrace{\begin{bmatrix} w_{11} & w_{12} & 0 & w_{21} & w_{22} & 0 & 0 & 0 & 0 \\ 0 & w_{11} & w_{12} & 0 & w_{21} & w_{22} & 0 & 0 & 0 \\ 0 & 0 & 0 & w_{11} & w_{12} & 0 & w_{21} & w_{22} & 0 \\ 0 & 0 & 0 & 0 & w_{11} & w_{12} & 0 & w_{21} & w_{22} \end{bmatrix}}_{A} \cdot \begin{bmatrix} x_{11} \\ x_{12} \\ x_{13} \\ x_{21} \\ x_{22} \\ x_{23} \\ x_{31} \\ x_{32} \\ x_{33} \end{bmatrix} = \begin{bmatrix} z_{11} \\ z_{12} \\ z_{21} \\ z_{22} \end{bmatrix}. \tag{2.9b}$$

When applying a convolution layer to image data, the input a^{l-1} may have more than 1 channel (e.g. an ordinary colored image has 3 channels), thus, the filter shall also have such additional dimension and the same size. This layer may also generate an output with more than 1 channel, so the number of filters and biases is increased proportionally. Therefore, the convolutional layer is characterized by a weight tensor w of shape $\{C_{\text{out}}, C_{\text{in}}, H_w, W_w\}$ and a bias b of size C_{out}, where C_{out} is the number of output channels and C_{in} is the number of input channels. The calculation of the output $z^l_{i_1 i_2 i_3}$ of this layer is only slightly different than the formulated in Equation (2.8), due to the extra dimension and bias:

$$z^l_{i_3 i_2 i_1} = b^l_{i_3} + \sum_{k_3=1}^{C_{\text{in}}} \sum_{\substack{k_2=1 \\ j_1=i_1+k_1-1 \\ j_2=i_2+k_2-1}}^{H_w} \sum_{k_1=1}^{W_w} a^{l-1}_{k_3 j_2 j_1} w^l_{i_3 k_3 k_2 k_1} = b^l_{i_3} + \sum_{k_3=1}^{C_{\text{in}}} a^{l-1}_{k_3} * w^l_{i_3 k_3}. \tag{2.10}$$

By extending the previous example with an input sample x, a filter w, and a bias b, with shapes $\{3,3,3\}$, $\{1,3,2,2\}$, and $\{1\}$, the first entry of the output of this layer would be calculated as represented in Figure 2.4.

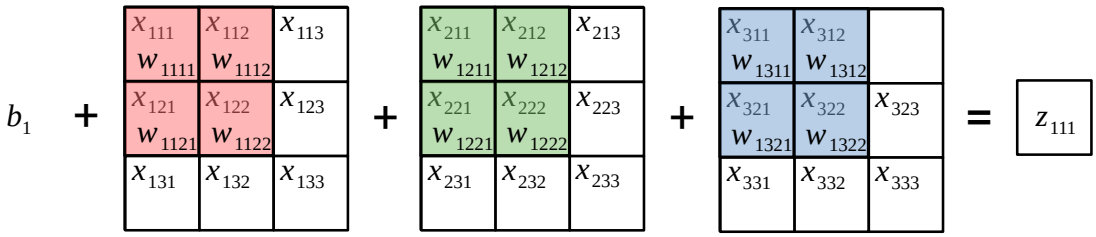


Figure 2.4: Computation of the first element of a convolutional layer output z, given an input sample x, a filter w, and a bias b, with shapes $\{3,3,3\}$, $\{1,3,2,2\}$, and $\{1\}$, respectively. colored regions represent sums of the products of the overlapped elements of each input channel.

The convolution layer is characterized by 3 more options: stride, padding, and dilation. The stride of the convolution is the number of steps the kernel takes when sliding over the image. Figure 2.3 is an example of a convolution with the default stride of 1, since the kernel moves one place at a time. The padding of the convolution is the number of elements added to the borders of the images – the default value is 0. The dilation controls the spacing between the elements of the kernel, with the normal spacing being 1. Equation (2.11) shows an example of a matrix that is dilated by 2 and then padded by 1. Lastly, the size of the output of the convolution layer will correspond to Equation (2.12).

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \xrightarrow[=2]{\text{dilation}} \begin{bmatrix} x_{11} & 0 & x_{12} & 0 & x_{13} \\ 0 & 0 & 0 & 0 & 0 \\ x_{21} & 0 & x_{22} & 0 & x_{23} \\ 0 & 0 & 0 & 0 & 0 \\ x_{31} & 0 & x_{32} & 0 & x_{33} \end{bmatrix} \xrightarrow[=1]{\text{padding}} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & x_{11} & 0 & x_{12} & 0 & x_{13} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & x_{21} & 0 & x_{22} & 0 & x_{23} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & x_{31} & 0 & x_{32} & 0 & x_{33} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}. \tag{2.11}$$

$$\text{output size} = \frac{\text{input size} + 2 \times \text{padding} - \text{dilation} \times (\text{kernel size} - 1) - 1}{\text{stride}} + 1 \tag{2.12}$$

In the particular case of the computation of the backward propagation of the error (computing $\frac{\partial L}{\partial a^{l-1}}$), if one represents the convolution in a similar matrix multiplication as the one in Equation (2.9b), then it immediately takes the form of Equation (2.6). This would correspond to multiplying the transpose of A with the propagated error $\frac{\partial L}{\partial z^l}$ [33]. Nonetheless, this operation also has its dual – transposed convolution. To avoid formulating the transposed convolution, the same result may be achieved, as derived in [34, 35], by a convolution of the propagated error $\frac{\partial L}{\partial z^l}$ and a 180° rotated version of the filter/weight tensor w^l with the C_{out} and C_{in} dimensions swapped and:

- stride $= 1$
- padding $=$ (kernel size $- 1$) $\times$ forward dilation $-$ forward padding
- dilation $=$ forward dilation
- error tensor dilation $=$ forward stride

The weight gradient may be computed, as demonstrated in [36, 37], by a convolution of the layer input a^{l-1} and the propagated error $\frac{\partial L}{\partial z^l}$. However, the convolution is slightly different, since the input channels are considered individually and convolved with each error channel, originating the $C_{\text{out}} \times C_{\text{in}}$ kernels of the weight tensor. The convolution parameters are:

- stride $=$ forward dilation
- padding $=$ forward padding
- dilation $=$ forward stride

The bias gradient will simply correspond to the sum of the propagated error along its rows and columns.

Pooling Layer

This layer applies a pooling operation to an input and has no associated weights nor activation function. It is mostly used in CNNs, since it allows to downsample feature maps, improve robustness against local changes, and decrease the computational complexity [27]. Similar to a convolution, there is a window that slides over an "image" and performs an operation for each overlap. This layer is characterized by the same options as a convolutional layer: stride, padding, and dilation – however, the default stride is the kernel size, so that the windows do not overlap.

The most popular pooling layers perform a 2D average pool or a maximum pool. Average pooling consists of calculating the average of the elements of each window, while in maximum pooling the maximum value of each window is selected. Equation (2.13a) shows the result of average pooling a 4×4 matrix x by a kernel of size 2×2 and Equation (2.13b) is its general formulation. Equations (2.14a) and (2.14b) are the equivalent but for maximum pooling.

When applying this layer in the backward propagation formulation, the error may be propagated using the chain rule and the derivatives of Equations (2.13b) or (2.14b). Consequently, in the average pooling layer, each node j from the window that originated the node i is assigned the value of $\frac{\partial L}{\partial z_i^l}$ divided by the number of elements of the window. In the maximum pooling layer, the node j that was the argmax of

the window is assigned the value of $\frac{\partial L}{\partial z_i^l}$, while the others are assigned 0. Equations (2.13c) and (2.14c) formalize the backpropagation for each type of pooling, respectively.

$$\left[\begin{array}{cc|cc} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ \hline x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{array}\right] \xrightarrow{\text{average pooling}} \left[\begin{array}{c|c} \dfrac{x_{11}+x_{12}+x_{21}+x_{22}}{4} & \dfrac{x_{13}+x_{14}+x_{23}+x_{24}}{4} \\ \hline \dfrac{x_{31}+x_{32}+x_{41}+x_{42}}{4} & \dfrac{x_{33}+x_{34}+x_{43}+x_{44}}{4} \end{array}\right], \tag{2.13a}$$

$$z_i^l = \frac{1}{H_w W_w} \sum_{j \in \text{window}} \sum a_j^{l-1}, \tag{2.13b}$$

$$\left.\frac{\partial L}{\partial a_j^{l-1}}\right|_{j \in \text{window}} = \frac{\partial L}{\partial z_i^l} \frac{\partial z_i^l}{\partial a_j^{l-1}} = \frac{\partial L}{\partial z_i^l} \frac{1}{H_w W_w}; \tag{2.13c}$$

$$\left[\begin{array}{cc|cc} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ \hline x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{array}\right] \xrightarrow{\text{maximum pooling}} \left[\begin{array}{c|c} \max\{x_{11}, x_{12}, x_{21}, x_{22}\} & \max\{x_{13}, x_{14}, x_{23}, x_{24}\} \\ \hline \max\{x_{31}, x_{32}, x_{41}, x_{42}\} & \max\{x_{33}, x_{34}, x_{43}, x_{44}\} \end{array}\right], \tag{2.14a}$$

$$z_i^l = \max_{j \in \text{window}} \left\{a_j^{l-1}\right\}, \tag{2.14b}$$

$$\frac{\partial L}{\partial a_j^{l-1}} = \frac{\partial L}{\partial z_i^l} \frac{\partial z_i^l}{\partial a_j^{l-1}} = \begin{cases} \dfrac{\partial L}{\partial z_i^l} & \text{if } j \text{ is argmax of window,} \\ 0 & \text{otherwise.} \end{cases} \tag{2.14c}$$

Dropout

Dropout is one of the most common and effective regularization techniques applied to NNs [27, 38]. The main aim of a regularization step is to promote the generalization of the network, improving its performance with unseen data. During training, the dropout layer receives an input tensor, randomly fills it with zeroes (with probability p), and outputs it scaled by $\frac{1}{1-p}$ [39]. This technique improves the classification performance because it forces different subsets of the network to train, improving their independence and redundancy. During inference, no elements are zeroed nor scaled, it simply computes the identity function.

The backward propagation formulation may be derived from the chain rule and ensues the same principle: nodes that were zeroed out during the forward propagation are zeroed out again, otherwise, they are scaled by the same factor.

Activation Functions

The activation function is usually a non-linear operation that is applied after all the computations of a given layer (such as Linear or Convolutional), as defined in Equation (2.1b). Without such non-linear activation function, a layer would only be able to learn linear transformations, restricting the modeling ability of the network, even with multiple layers. By introducing such non-linearity, it provides the model with the capability of approximating non-linear functions and of benefiting from deeper structures [27]. Some common non-linear activation functions are: Sigmoid, TanH, and ReLU [39] – their choice depending on the problem at hand.

Sigmoid is an activation function with a characteristic "S"-shape curve, mapping its arguments between 0 and 1. Because of its range, This function is especially used in problems where the output corresponds to a predicted probability. However, as the input values are farther from the center, their derivatives tend to become more flat, which correspond to gradients that are small or have vanished, which may lead to the training getting stuck. Equations (2.15a) and (2.15b) define the formulations that are used during forward and backward propagation, respectively.

$$g\left(z_i^l\right) = \text{sigmoid}\left(z_i^l\right) = \sigma\left(z_i^l\right) = \frac{1}{1+e^{-z_i^l}}, \tag{2.15a}$$

$$g'\left(z_i^l\right) = -\frac{\left(-e^{-z_i^l}\right)}{\left(1+e^{-z_i^l}\right)^2} = \frac{1}{1+e^{-z_i^l}} \cdot \left(1 - \frac{1}{1+e^{-z_i^l}}\right) = \sigma\left(z_i^l\right) \cdot \left(1 - \sigma\left(z_i^l\right)\right). \tag{2.15b}$$

TanH is an activation function very similar to sigmoid, since it is also "S"-shaped, but its arguments are mapped between -1 and 1. It has stronger gradients, but it still may have vanishing gradients. One advantage of this mapping is that the sign of the input is not lost. Equations (2.16a) and (2.16b) define the formulations that are used during forward and backward propagation, respectively.

$$g\left(z_i^l\right) = \tanh\left(z_i^l\right) = \frac{e^{z_i^l} - e^{-z_i^l}}{e^{z_i^l} + e^{-z_i^l}} = 2 \cdot \sigma\left(2 \cdot z_i^l\right) - 1, \tag{2.16a}$$

$$g'\left(z_i^l\right) = \frac{\left(e^{z_i^l} + e^{-z_i^l}\right)^2 - \left(e^{z_i^l} - e^{-z_i^l}\right)^2}{\left(e^{z_i^l} + e^{-z_i^l}\right)^2} = 1 - \frac{\left(e^{z_i^l} - e^{-z_i^l}\right)^2}{\left(e^{z_i^l} + e^{-z_i^l}\right)^2} = 1 - \tanh^2\left(z_i^l\right). \tag{2.16b}$$

ReLU is the most popular activation function in deep learning [27]. Its implementation consists of setting negative values to 0 and leaving the others unchanged. Some advantages of this function are the non-saturation of its gradient [7], sparser networks [40], and less expensive operations compared to sigmoid or TanH. Moreover, ReLU does not have the vanishing gradient problem for values far from zero, although it still has a similar one (called dying ReLU problem), caused by the derivative of negative values being zero. Nonetheless, this function is usually preferred due to the advantages above. Equations (2.17a) and (2.17b) define the formulations that are used during forward and backward prop-

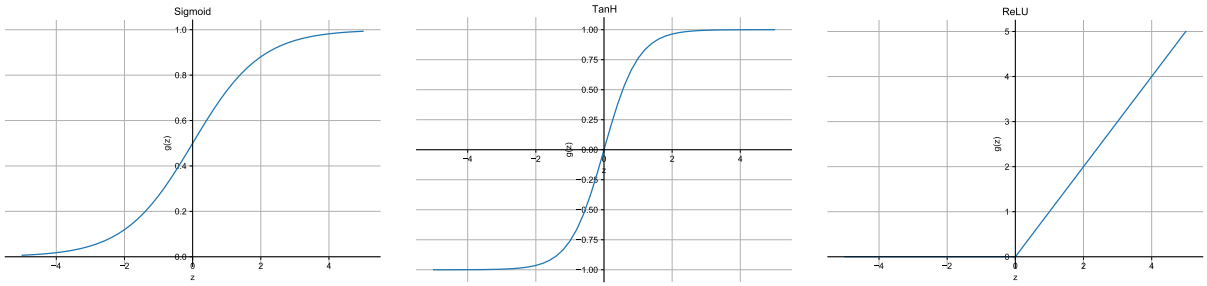
agation, respectively.

$$g\left(z_i^l\right) = \text{relu}\left(z_i^l\right) = \max\left(0, z_i^l\right), \tag{2.17a}$$

$$g'\left(z_i^l\right) = \begin{cases} 0 & \text{if } z_i^l < 0, \\ 1 & \text{if } z_i^l > 0. \end{cases} \tag{2.17b}$$

The derivative is not defined for $z_i^l = 0$. Usually, one extends $g'(0) = 0$ to obtain a sparser result.

Figure 2.5 illustrates the sigmoid, TanH, and ReLU activation functions, respectively.



(a) Sigmoid (b) TanH (c) ReLU

Figure 2.5: Activation functions: Sigmoid, TanH, and ReLU, respectively.

2.1.3 Loss Functions

The role of the loss function (L) was presented when explaining the training procedure of neural networks, where the objective was to minimize it. This function is used as a sort of distance between the network output and the corresponding target. As one would expect, the choice of the loss function will greatly impact the training process, in terms not only of the achieved performance, but also of the required time to do it. The empirical risk $\mathcal{R}$, presented in Equation (2.2), is often also referred to as loss function.

Mean Squared Error (MSE)

For regression problems, the most common loss function is the Squared Error [41], also known as Mean Squared Error (MSE) [39] (if one averages the squared errors along the output nodes $a^\ell = y$). It is computed as:

$$L\left(y, \hat{y}\right) = \sum_{i \in \text{last layer}} \left(y_i - \hat{y}_i\right)^2. \tag{2.18}$$

Since it calculates the square of the error, this function is more sensitive to outliers. Its gradient is proportional to the error:

$$\frac{\partial L}{\partial y_i} = 2 \cdot \left(y_i - \hat{y}_i\right). \tag{2.19}$$

Although unusual, this loss can also be used for classification problems, by assigning a probability of 1 to the target class and 0 to the others.

Cross Entropy

For classification problems, the Cross Entropy loss is a better choice. It is mostly useful for multi-class problems, by combining a softmax and a Negative Log Likelihood (NLL) function [39, 41]. For this type of problems, the target is usually specified as the index of the target class ($\hat{y}$). Thus, its values are calculated for the last layer ℓ as:

$$S\left(a^{\ell}\right)_i = \frac{\exp\left(a_i^{\ell}\right)}{\sum\limits_{j \in \text{last layer}} \exp\left(a_j^{\ell}\right)}, \tag{2.20a}$$

$$L\left(a^{\ell}, \hat{y}\right) = -\log\left(S\left(a^{\ell}\right)_{\hat{y}}\right). \tag{2.20b}$$

Hence, the softmax function S value can be interpreted as the predicted probability for a given class, since it will assign values between 0 and 1 and the sum of the softmax values of all the different classes is 1. Thus, the minimization of the cross entropy loss is analogous to obtaining the maximum likelihood estimation for the network weights with the softmax as the probability distribution. Its gradient can be computed as:

$$\frac{\partial L}{\partial a_i^{\ell}} = -\frac{1}{S\left(a^{\ell}\right)_{\hat{y}}} \frac{\partial S\left(a^{\ell}\right)_{\hat{y}}}{\partial a_i^{\ell}} = \begin{cases} S\left(a^{\ell}\right)_{\hat{y}} - 1 & \text{for } i = \hat{y}, \\ S\left(a^{\ell}\right)_i & \text{otherwise.} \end{cases} \tag{2.21}$$

2.1.4 Optimizers

The optimizer is the procedure that is executed to update the network parameters. The most popular one is the gradient descent algorithm, presented in Section 2.1.1. Depending on the number of samples used in each update, some other variants of this algorithm may be used, with mini-batch being the most common choice. In addition to varying the size of the mini-batch, there are more variations of the gradient descent algorithm, where the update step, formulated in Equation (2.3), is slightly different. The most important optimizer algorithms will be briefly presented in the following paragraphs [30].

Stochastic Gradient Descent (SGD)

SGD [42] is no more than a basic implementation of the mini-batch or online gradient descent class of algorithms. It approximates the actual gradient of the entire dataset by the gradient of a small random/shuffled mini-batch. The weight update is characterized by the learning rate/update step η, which is the same for all the parameters. However, this learning rate may be constant or be adjusted throughout the training process with a certain learning rate schedule. Although this algorithm may have a slow convergence, it can be accelerated by implementing a momentum term v (velocity) in the weight update. A common analogy to this term is to imagine a ball accelerating down a hill that keeps moving in the

same direction, even if it finds a small valley (local minima). This algorithm may be formulated as:

$$v(t+1) = \gamma v(t) + \eta \left.\frac{\partial \mathcal{R}}{\partial w}\right|_{w(t)}, \tag{2.22a}$$

$$w(t+1) = w(t) - v(t+1), \tag{2.22b}$$

where γ is a momentum rate usually set to 0.9 (or similar).

Although the momentum term is good to speed up the convergence of SGD, if set too large, it has the potential to overshoot. Nesterov Accelerated Gradient (NAG) [43] implements a "smarter" momentum, where the gradient is calculated at an estimate of the next set of weights, as indicated in Equation (2.23). This look-ahead technique not only makes the algorithm to not overshoot too much, but it also makes it more responsive. Its implementation is slightly different from the one previously presented:

$$v(t+1) = \gamma v(t) + \eta \left.\frac{\partial \mathcal{R}}{\partial w}\right|_{w(t)-\gamma v(t)}. \tag{2.23}$$

Root Mean Square Propagation (RMSprop)

One disadvantage of the SGD algorithm is that its hyperparameters η and γ are the same for all the model parameters. To improve this, adaptive optimization algorithms have also been proposed, where those hyperparameters are adapted for each and every weight. This will result in smaller updates for parameters that are changing frequently and larger updates for parameters that are not changing as much.

RMSprop is an adaptive optimization algorithm proposed in [44]. In this algorithm, the learning rate is normalized by the square root of a decaying average of the gradient squared, hence its name (root mean square). Thus, the weights may be updated as

$$E(t+1) = \gamma E(t) + (1-\gamma) \left(\left.\frac{\partial \mathcal{R}}{\partial w}\right|_{w(t)} \right)^2, \tag{2.24a}$$

$$w(t+1) = w(t) - \frac{\eta}{\sqrt{E(t+1)+\epsilon}} \left.\frac{\partial \mathcal{R}}{\partial w}\right|_{w(t)}, \tag{2.24b}$$

where ϵ is a small smoothing term that avoids dividing by 0. The authors suggest using $\gamma = 0.9$ and a good choice for the learning rate would be $\eta = 0.001$. RMSprop algorithm is more resilient than SGD to poorly initialized networks and converges faster.

Adaptive Moment Estimation (Adam)

Adam is also an adaptive algorithm [45] that differs from RMSprop by the introduction of a momentum term used in the updating step. For its implementation, the first moment m (mean) and the second moment v (uncentered variance) are computed as in Equation (2.25a). Since these values are initialized as 0, they are biased towards 0, specially in the first iterations. Therefore, they are bias-corrected into

$\hat{m}$ and $\hat{v}$ as in Equation (2.25b). At last, each weight is updated with Equation (2.25c).

$$m(t+1) = \beta_1 m(t) + (1-\beta_1)\left.\frac{\partial \mathcal{R}}{\partial w}\right|_{w(t)}, \qquad v(t+1) = \beta_2 v(t) + (1-\beta_2)\left(\left.\frac{\partial \mathcal{R}}{\partial w}\right|_{w(t)}\right)^2, \tag{2.25a}$$

$$\hat{m}(t+1) = \frac{m(t+1)}{1-(\beta_1)^{t+1}}, \qquad \hat{v}(t+1) = \frac{v(t+1)}{1-(\beta_2)^{t+1}}, \tag{2.25b}$$

$$w(t+1) = w(t) - \frac{\eta}{\sqrt{\hat{v}(t+1)+\epsilon}}\hat{m}(t+1), \tag{2.25c}$$

These hyperparameters were proposed with the default values of $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$.

From the subset of optimizers that were presented, Adam is the one with more features and it usually presents the best overall performance [30] for DNNs. It has a fast convergence (just like RMSprop) but outperforms it as the gradients become sparser. Nonetheless, recent papers still use a simple SGD optimizer. Despite its slower convergence, it tends to outperform faster optimizers in later stages of the training [46].

2.1.5 Convolutional Neural Networks

Throughout the years, several DNN architectures have been proposed that achieve state of the art performance in various application domains. As a result, it is usually better to use one of these reference architectures to solve a given problem instead of designing a new one, because these were already widely studied and benchmarked, so they provide more reliable results.

The reference models that will be presented in the following paragraphs are some of the most important that were designed for image classification problems [47]. These are characterized by the use of convolutional layers and are referred to as CNNs. They will be presented by **order of increasing complexity**.

Some networks are illustrated as block diagrams, where the colored blocks refer to trainable layers and the white blocks refer to other non-trainable layers. Each block will indicate the main characteristics of the associated layer. The arrows correspond to the flow of data through the network. The first input is an image, whose dimensions are displayed above the arrow and the number of channels displayed below. When this tensor is flattened, the arrows display the size of the resulting vector.

LeNet-5

LeNet-5, proposed in 1998 [48], was historically important because it was the first CNN to achieve state of the art accuracy (slightly above $99\,\%$) in handwritten digit recognition. This network is even able to classify digits without being affected by small distortions on the images. It is also a good starting point to understand CNNs, since it is only composed of 5 (trainable) layers: 3 convolutional and 2 fully connected layers, as shown in Figure 2.6. This model has about 0.06 million parameters.

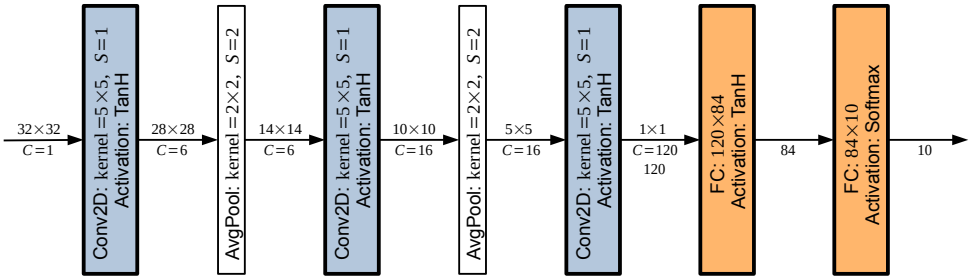


Figure 2.6: Architecture of the CNN LeNet-5 [48].

CifarNet

CifarNet was presented in 2015 [49]. It is a small network that was designed to classify the CIFAR-10 dataset [50], which consists of various small colored images. Its number of layers is not specifically defined, but the main characteristic is the use of 3 convolutional layers and 1 or 2 fully connected layers. The number of parameters depends on the chosen structure, but it is of the order of 0.1 million. Figure 2.7 shows the structure of CifarNet, as presented in [49].

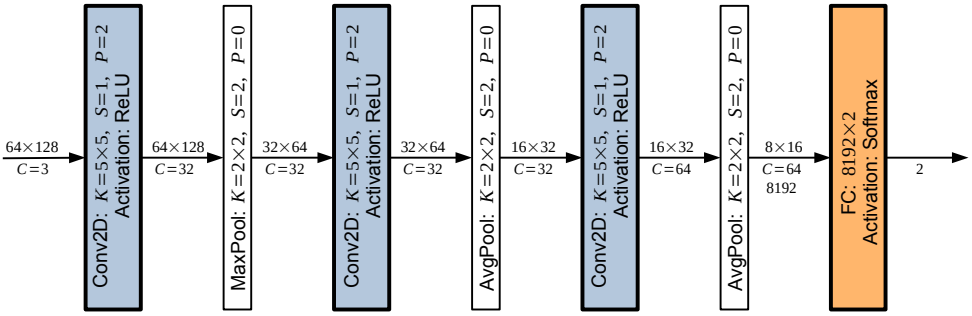


Figure 2.7: Architecture of the CNN CifarNet [49].

AlexNet

AlexNet, proposed in 2012 [7], broadened the applicability of CNNs for image classification, which was practically limited to hand digit recognition. This network was one of the largest to that date to be used in image classification and object recognition tasks, achieving almost $85\,\%$ Top-5 accuracy (correct class is in the 5 most probable classes predicted) in ImageNet, winning the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012. It is composed of 8 layers: 5 convolutional and 3 fully connected layers, as shown in Figure 2.8. In addition to having more layers than LeNet-5, it also has a lot more

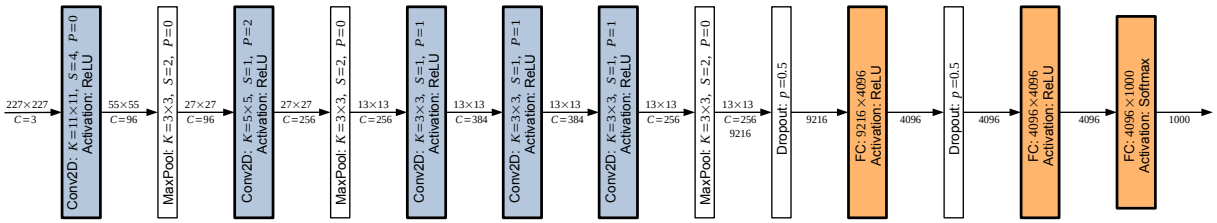


Figure 2.8: Architecture of the CNN AlexNet. The originally proposed [7] was slightly different because it was mean to be run in 2 parallel GPUs.

parameters, approximately 60 million. It also introduced dropout layers, ReLU activations, and overlap pooling (stride $<$ kernel size).

VGGNet

The CNN models VGG-16 and VGG-19 were proposed in 2014 by the Visual Geometry Group (VGG) [51]. Following the steps from AlexNet, these models became even deeper, being formed by 16 and 19 layers, respectively. The initial layers are convolutional and the last 3 are fully connected. The number of parameters consequently increased to about 138 and 144 million. Some of the introduced innovations were the use of small size kernels and a homogeneous topology (the kernels have all the same size). These allowed to improve the Top-5 accuracy to about $93\,\%$ for image recognition (ImageNet). However, it takes very long to train. It achieved 2nd place in the ILSVRC 2014.

ResNet

The ResNet architecture was proposed in 2015 [52] and its main networks have 50, 110, and 152 layers. These revolutionized CNNs by introducing the concept of residual learning – shortcuts that allow layers to skip some connections. Furthermore, the use of multiple fully connected layers at the end was replaced by a global average pooling layer, which decreased the number of parameters and the overfitting caused by FC layers [53]. With this change, the proposed architecture improved the performance over VGG with reduced computation complexity and using about 26, 45, and 60 million parameters, respectively [54]. It won the ILSVRC 2015 competition, with a Top-5 accuracy on ImageNet dataset around $96\,\%$.

2.1.6 Reference Datasets

To evaluate the performance of a CNN model in image-related tasks, there are several common datasets to use and to benchmark with. Usually, these benchmarks are large sets of labeled images that allow comparing the performance of different models for the same task and input. In the following paragraphs, it will be presented some popular image datasets in the order of increasing complexity.

MNIST

MNIST is a handwritten digits dataset [55]. It is known for its simplicity and it is widely tried as the first dataset in various machine learning problems. It is frequently said that "if it doesn't work on MNIST, it won't work at all" [56]. It has a training set of $60\,000$ samples and a test set of $10\,000$ samples. Each sample corresponds to a 28×28 grayscale image (1 channel) of a handwritten digit between 0 and 9 (10 classes).

However, due to its simplicity, if an algorithm "does work on MNIST, it may still fail on others". Therefore, it is recommended to use datasets more complex and more representative of modern CV tasks, as mentioned in [56].

Figure 2.9: Illustration of few samples of the handwritten digits MNIST dataset. Different classes per row. Image obtained from Wikimedia Commons at [57].

Fashion MNIST

Fashion MNIST is a dataset composed of images from 10 different types of clothing pieces (10 classes) [56]. It is designed as a direct drop-in replacement for the MNIST dataset, therefore, it also has $60\,000$ training samples and $10\,000$ test samples, being each one a 28×28 grayscale image (1 channel). Its 10 classes are: t-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot. This dataset was introduced in response to some of the setbacks of MNIST, presenting itself as more complex and diverse, resulting in more advanced features.

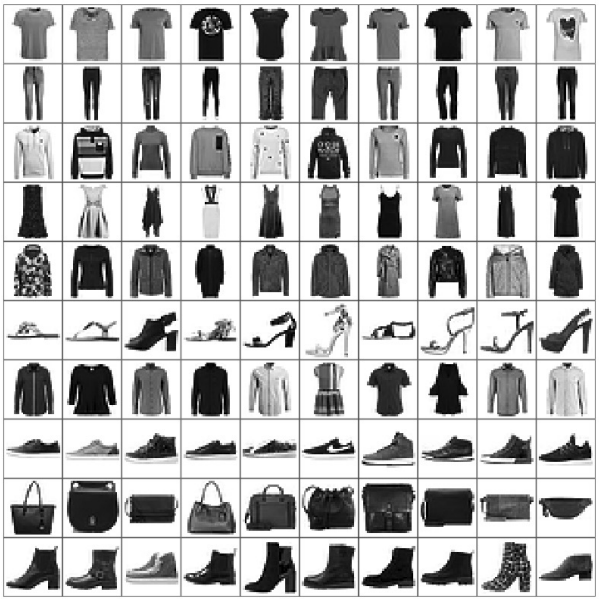


Figure 2.10: Illustration of a few samples of the Fashion MNIST dataset. Different classes per row. Image obtained from Markus Thill at [58].

CIFAR-10

CIFAR-10 is a dataset of small colored images from 10 different classes [50]. It consists of $50\,000$ training images and $10\,000$ test images, resulting in 6000 images per class. Each sample is a 32×32 colored image (3 channels). The 10 classes of images are: airplane, automobile, bird, cat, deer, dog, frog,

horse, ship, and truck. Not only because the images are colored and not grayscale, but also due to the complexity of each class, this dataset is much harder to classify, when compared to the previously presented ones.

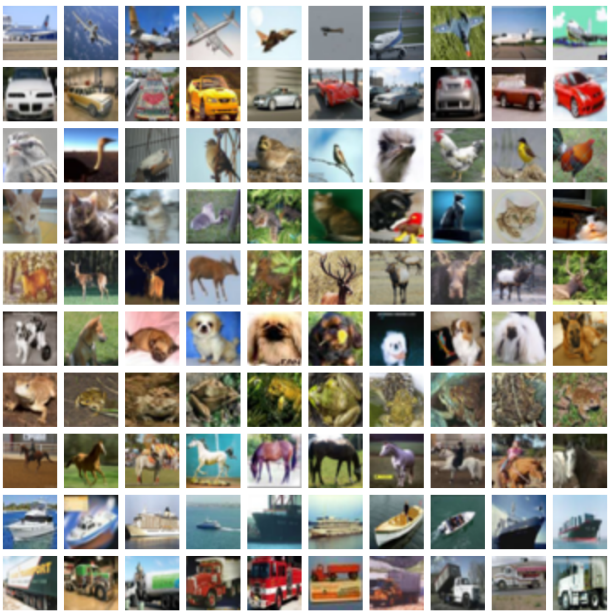


Figure 2.11: Illustration of a few samples of the CIFAR-10 dataset. Different classes per row. Image obtained from Alex Krizhevsky at [50].

CIFAR-100

CIFAR-100 is a dataset very similar to CIFAR-10, one difference being that it is divided into 100 classes instead of 10 [50]. It also consists of 50 000 training images and 10 000 test images, but using only 600

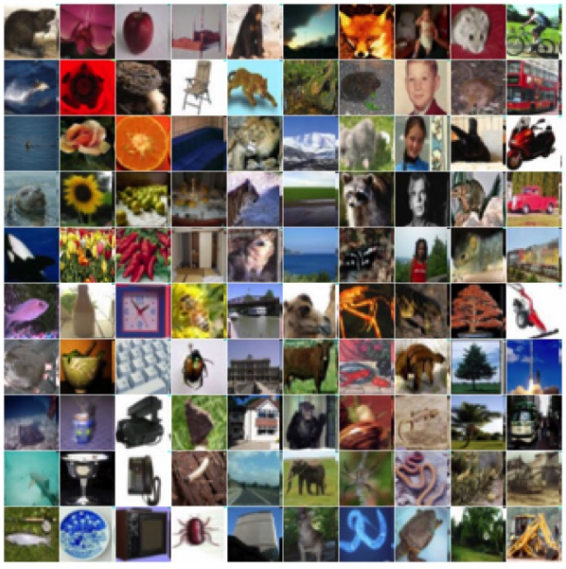


Figure 2.12: Illustration of a few samples of the CIFAR-100 dataset. Each sample is from a different class. Image obtained from [59].

images per class. Each image is 32×32 and colored (3 channels). The 100 classes are grouped into 20 superclasses, resulting in "fine" and "coarse" labels, respectively. This dataset also differs from the previous one by providing many more classes, presenting itself as a harder classification problem.

ImageNet

ImageNet is a still ongoing research project that provides one of the largest labelled images dataset. It consists of about 14 million images, classified into 27 high-level categories and using about $20\,000$ subcategories. On average, there are about 500 images per subcategory, but it varies. The images are colored (3 channels) and not all have the same dimensions, although they are usually resized to 256×256.

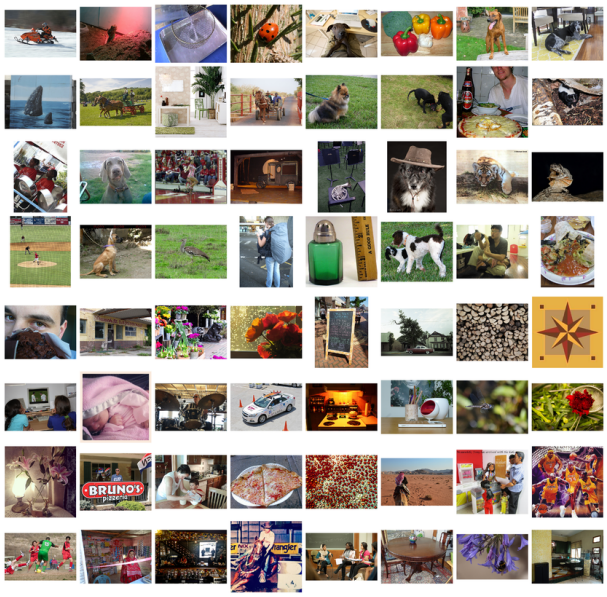


Figure 2.13: Illustration of a few samples of the ImageNet dataset, obtained from ImageNet Large Scale Visual Recognition Challenge, 2015 [60].

2.1.7 Deep Learning Frameworks

Deep learning frameworks offer all the necessary tools to design, train, and evaluate DNN models. Nowadays, there are several DL frameworks available, to be used both for research and production. Most of them are also based on high-performance GPU accelerated implementations [61]. Some of the most popular are: PyTorch [37], TensorFlow [62], Keras [63], Caffe [64], Apache MXNet [65], Microsoft Cognitive Toolkit (CNTK) [66], etc.

The first two mentioned frameworks, **PyTorch** [37] and **TensorFlow** [62], were both considered for this work, since they are open source and offer similar functionalities. Both of them are being developed by powerful enterprises: Facebook for PyTorch, and Google for TensorFlow. The latter framework emerged earlier, which gave it more time to mature and to build a good reputation, being the most frequently used for production. Nonetheless, due to the smaller learning curve and dynamic computation

of PyTorch, Facebook's framework picked up the pace and it is becoming the most popular framework for research, as depicted in Figure 2.14, obtained from [67].

These frameworks allow implementing arbitrary DNNs and their associated functions. More importantly, the training of a model requires no additional effort due to automatic differentiation functionality, which essentially automatically computes the backpropagation when the available functions are used. However, there is a relevant difference between these two: PyTorch uses a dynamic graph definition, while TensorFlow uses a static definition. This results in PyTorch having a more imperative coding style, giving the user greater control of the execution flow.

Still on the easiness of use, both frameworks are commonly used with the Python programming language, but they also offer APIs for C++. Although Python tends to offer an easier and more flexible syntax, C++ usually provides better performance. Moreover, these frameworks may be extended with custom user-designed functions and modules, even supporting the integration of C++ functions within Python. In terms of data types, the two frameworks support more or less the same formats but, provide no straightforward and "high level" method to extend them.

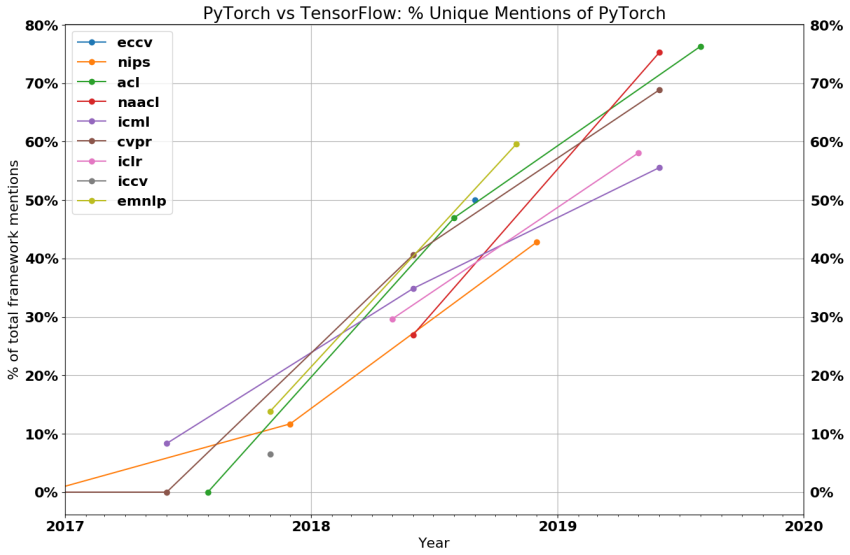


Figure 2.14: Ratio of unique mentions of PyTorch compared against TensorFlow in various top research conferences over time, obtained from [67].

2.2 Computer Number Formats

The emergence of digital computing devices, in the second half of the twentieth century, raised the need to formalize the internal representations of the involved numerical values, since these devices use binary encoding. Although representing integer values in binary is trivial (usually using the two's-complement notation), most of the real-life computations require the representation of real numbers. For this purpose, there are several formats available [68], each providing different compromises between the complexity of its manipulation and the involved rounding error.

2.2.1 Fixed-Point

A real number encoded with a fixed-point format is similar to an integer value scaled by an implicit factor, usually a power of 2. This is identical to assigning a fixed number of bits for the integer part (m) and for the fraction part (f). An additional bit is used to account for the sign of the number. Thus, its layout is:

Sign (1 bit)	Integer (m bits)	Fraction (f bits)

Figure 2.15: Format encoding of a generic fixed-point format.

Moreover, decoding a fixed-point format is as simple as decoding an integer value, composed by the integer and fraction fields, and then scaling it by the power of 2 obtained with the fraction size:

$$x = \text{decoded integer} \times 2^{-f}. \tag{2.26}$$

Since a fixed-point value may be treated as an integer value, its operations may be computed with the same processing units. However, a downside of this format is the limited dynamic range, caused by the fixed radix point position.

2.2.2 Floating-Point

If an application requires a larger dynamic range, the use of a Floating-Point (FP) format may be a more appropriate choice. The floating-point representation is similar to scientific notation, in the sense that a number is encoded with a significand and an exponent, thus, the radix point has no fixed position. The most popular format for floating-point arithmetic is the IEEE Standard for Floating-Point Arithmetic (IEEE 754) [13], a standard established in 1985 that suffered only some slight changes throughout the years.

IEEE 754 Standard

Nowadays, most computers implement the IEEE 754 standard, more specifically, the binary32 and binary64 formats, also known as single-precision floating-point (float) and double-precision floating-point (double). Although not as frequent, the standard also defines the binary16 and binary128 formats, also known as half-precision floating-point (half) and quadruple-precision floating-point. These formats are characterized by 3 fields: a sign, a biased exponent, and a trailing significand field (fraction). The layout of these formats is depicted in Figure 2.16.

Quadruple-precision floating-point (128 bits):	Sign (1 bit)	Exponent (15 bits)	Fraction (112 bits)
Double-precision floating-point (64 bits):	Sign (1 bit)	Exponent (11 bits)	Fraction (52 bits)
Single-precision floating-point (32 bits):	Sign (1 bit)	Exponent (8 bits)	Fraction (23 bits)
Half-precision floating-point (16 bits):	Sign (1 bit)	Exponent (5 bits)	Fraction (10 bits)

Figure 2.16: Format encodings of quadruple-precision, double-precision, single-precision, and half-precision floating-points according to [13].

From the extracted fields of these formats, it is obtained a triplet representing the number. Moreover, the exponent field is offset with a bias (equal to $2^{\text{exponent size}-1} - 1$); and the leading bit of the significand is implicitly encoded in the biased exponent (1 for normalized and 0 for subnormal). At last, the number may be generally decoded as:

$$x = (-1)^{\text{sign}} \times 2^{\text{exponent}-\text{bias}} \times \text{mantissa}. \tag{2.27}$$

In addition to defining arithmetic formats, the IEEE 754 standard also specifies the rounding rules, operations, and exception handling. This latter emphasises one disadvantage of the IEEE 754 floating-point, involving various repeated patterns to represent Not a Number (NaN) values, $\pm\infty$ values, and even the representation of ± 0. Some other disadvantages pointed for this data format in [69] are: lack of reproducibility guarantees across systems, possibility of overflow/underflow, the added complexity of using normalized/subnormal numbers, and misused exponent size.

Variations

Despite the widely established formats defined in the IEEE 754 standard, there are a few variations. One especially useful variation for deep learning applications is the Brain Floating Point (bfloat16) [70], which is a 16-bit truncated version of the 32-bit float that aims to accelerate machine learning computations. When compared to the IEEE 16-bit floating-point, the main difference is the exponent size, which is greater in the bfloat16 format. Moreover, there is also an 8-bit floating-point format [71], sometimes referred to as minifloat, although its use is more challenging.

Bfloat16 (16 bits):	Sign (1 bit)	Exponent (8 bits)	Fraction (7 bits)
Minifloat (8 bits):	Sign (1 bit)	Exponent (4 bits)	Fraction (3 bits)

Figure 2.17: Format encoding of bfloat16 [70] and minifloat [71].

This 8-bit variation of IEEE 754 floating-point stresses some of its limitations for low-precision floating-points. Namely, it will have 14 representations for NaN, both $\pm\infty$, and also the redundant ± 0, which make up $\sim 6.6\%$ of useless values. An important characteristic of any numbering format is the numeric range from the minimum positive number (*minpos*) to the maximum positive number (*maxpos*) – dynamic range – which, for this small exponent size, will be narrow (see Figure 2.18).

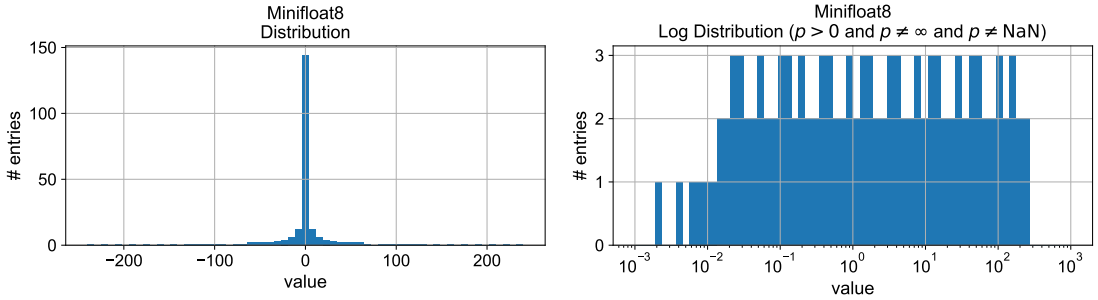


Figure 2.18: Distribution of minifloat8 values in linear (left) and logarithmic (right) scales.

Table 2.1 summarizes some characteristics of the mentioned floating-point formats, emphasizing how the exponent and fraction sizes can drastically change the range of the formats.

Table 2.1: Characteristics of some floating-point formats, namely, their size, the significand length/precision, and the minimum (subnormal) and maximum positive values.

Format	Size	Significand precision	*minpos*	*maxpos*
Quadruple (FP128)	128 bits	113 bits	6.5×10^{-4966}	1.2×10^{4932}
Double (FP64)	64 bits	53 bits	4.9×10^{-324}	1.8×10^{308}
Float (FP32)	32 bits	24 bits	1.4×10^{-45}	3.4×10^{38}
Half (FP16)	16 bits	11 bits	6.0×10^{-8}	6.6×10^{4}
Bfloat16	16 bits	8 bits	9.2×10^{-41}	3.4×10^{38}
Minifloat8	8 bits	4 bits	2.0×10^{-3}	2.4×10^{2}

2.2.3 Posit Format

Aiming to improve and replace the IEEE 754 floating-point, the Posit format (Type III Unum) was introduced by Gustafson in 2017 [20]. The proposition of this novel numeric format is that it delivers a larger dynamic range for smaller precisions, higher accuracy, better reproducibility, and simpler hardware and exception handling.

This alternative format [25] is characterized by a fixed size/number of bits (*nbits*) and an exponent size (*es*), which give the user the freedom to use the posit with the most appropriate properties for the problem at hand. The chosen posit configuration is usually specified as posit(*nbits*, *es*). Furthermore, each posit number is composed by the fields: sign, regime, exponent, and fraction. The layout of this format is as follows:

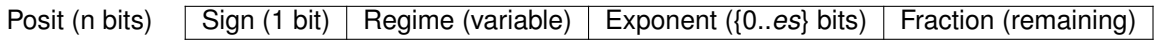


Figure 2.19: Format encoding according to the Posit Standard [25].

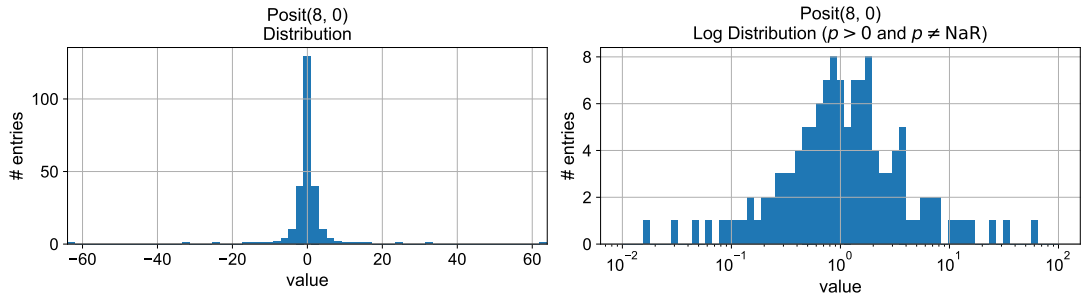
When the number is negative (sign bit $= 1$), one must take the 2's complement before decoding the other fields. Then, the regime field encodes a run-length value k, measured by the number of successive 0s or 1s until the opposite is found, as demonstrated in Table 2.2. Notice that the size of the regime is variable. Then, it follows the exponent, which encodes an unsigned integer and may occupy 0 to *es* bits, depending on the available bits. At last, if any bits remain, they are populated by a fraction similar to the IEEE 754, but with an implicit bit that is always 1. Thus, a number (p) encoded as a posit may be decoded as in Equation (2.28).

Table 2.2: Example of the regime field and corresponding run-length k for a maximum of 4 regime bits. If the first bit is a 0, negate the count, and if the first bit is a 1, then decrement the count by 1.

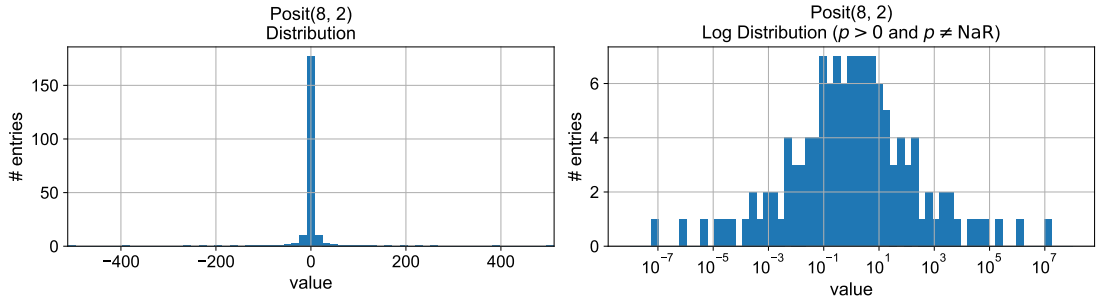
Binary	0000	0001	001x	01xx	10xx	110x	1110	1111
Numerical meaning, k	-4	-3	-2	-1	0	1	2	3

$$x = \begin{cases} 0 & p = 000\ldots0, \\ \pm\infty = \text{NaR} & p = 100\ldots0, \\ (-1)^{\text{sign}} \times 2^{2^{es}\times k} \times 2^{\text{exponent}} \times (1 + \text{fraction}) & \text{all other } p. \end{cases} \tag{2.28}$$

When comparing the structure of the posit format against the IEEE 754 floating-point, the main difference corresponds to the presence of the regime field. Its decoded value (k) will produce an additional exponent, as seen in Equation (2.28). The variable length of the regime allows numbers near to 1 (in magnitude) to have more accuracy than extremely large or extremely small numbers – tapered accuracy. Furthermore, the exponent can be heavily affected by the chosen value of *es*, which will, in turn, establish the largest and smallest numbers representable as posits. Therefore, posits offer a large dynamic range, that is adaptable through the value of *es*, and with a great accuracy around 1, as illustrated in Figure 2.20 for two 8-bit posit formats with different exponent sizes.



(a) Posit(8, 0)



(b) Posit(8, 2)

Figure 2.20: Distribution of posit(8, 0) and posit(8, 2) values in linear (left) and logarithmic (right) scales.

Despite the different fields, decoding a posit is still similar to decoding a float. However, and unlike floats, posits do not overflow nor underflow, but saturate to $\pm$maxpos or $\pm$minpos, respectively. Moreover, and as seen in Equation (2.28), the only 2 special values correspond to 0 and $\pm\infty$ = Not a Real (NaR). There are no subnormal values, hence the implicit bit of the fraction is always 1, which eases decoding. The NaR representation gives rise to the only exception handling required by posits, which considerably simplifies their hardware implementation when compared to IEEE 754 floating-points, which have various NaN representations and exceptions.

Quire

For each posit configuration, there is an associated quire format, similar to a large Kulisch accumulator [72]. The quire is a fixed-point format of size $nbits^2/2$ designed to accumulate exact sums of at least $2^{nbits-1} - 1$ products of posits without rounding or overflow. This quire format is particularly useful to implement the frequent dot products present in DNN computations, such as matrix multiplications, convolutions, etc. Figure 2.21 shows the format encoding of the quire, as specified in [25].

Quire ($nbits^2/2$ bits = total)	Sign (1 bit)	Carry Guard ($nbits - 1$ bits)	Integer ($(\text{total} - nbits)/2$ bits)	Fraction ($(\text{total} - nbits)/2$ bits)

Figure 2.21: Format encoding of the quire according to the Posit Standard [25].

Table 2.3 presents some key characteristics of example and general posit formats. Comparing to Table 2.1, the interesting aspects of this format is the possibility to declare a posit of arbitrary size and associated dynamic range (*nbits* and *es*).

Table 2.3: Characteristics of the recommended and general posit formats and the associated quires, according to the Posit Standard [25].

Posit		Significand precision	*minpos*	*maxpos*	Quire size	Quire exact dot product limit
nbits	*es*					
64 bits	3 bits	1 to 59 bits	4.9×10^{-150}	2.0×10^{149}	2048 bits	9.2×10^{18}
32 bits	2 bits	1 to 28 bits	7.5×10^{-37}	1.3×10^{36}	512 bits	2.1×10^{9}
16 bits	1 bits	1 to 13 bits	3.7×10^{-9}	2.7×10^{8}	128 bits	3.3×10^{4}
8 bits	0 bits	1 to 6 bits	1.5×10^{-2}	64	32 bits	127
n bits	e bits	1 to $(n - e - 2)$ bits	$2^{-2^e \times (n-2)}$	$2^{2^e \times (n-2)}$	$n^2/2$ bits	$2^{n-1} - 1$

Most Recent version of the Posit Standard

The posit format just described is according to the specifications of the publicly available Posit Standard [25]. However, there is a draft version[3] of an updated Posit Standard [24], formalized on August 2020.

This newer version introduced some changes compared to the prior version, namely, the hyperparameter for the maximum exponent size was fixed to $es = 2$ (see Figure 2.22). This greatly simplifies the conversion between posits of different sizes: pad it with 0s to make it larger, and round-off the least significant bit to make it smaller. It should be noted that there is no need to decode the posit fields in order to vary its size. This convenience may be exploited in implementations with uneven precision requirements.

Moreover, the size of the corresponding quire format was also modified to $16 \times nbits$, and it can now accumulate exact sums of at least $2^{31}-1$ products of posits. This a very positive change for its application for DL, since smaller posit formats can now accumulate more times and without error. The format encoding for the new quire format is shown in Figure 2.22. Table 2.4 shows the main characteristics of the posit and quire formats according to this standard.

[3] It was kindly shared via e-mail by Dr. John L. Gustafson on September 8, 2020.

Quire ($16n$ bits = total)	Sign (1 bit)	Carry Guard (31 bits)	Integer ($(\text{total} - 32)/2$ bits)	Fraction ($(\text{total} - 32)/2$ bits)

Figure 2.22: Format encoding of the quire according to the most recent Posit Standard [24].

Table 2.4: Characteristics of example and general posit formats, according to the most recent version of the Posit Standard [24].

Posit		Significand precision	*minpos*	*maxpos*	Quire size	Quire exact dot product limit
nbits	*es*					
64 bits	↑	1 to 60 bits	2.2×10^{-75}	4.5×10^{74}	1024 bits	↑
32 bits	|	1 to 28 bits	7.5×10^{-37}	1.3×10^{36}	512 bits	$2^{31} - 1 \approx$
16 bits	2 bits	1 to 12 bits	1.4×10^{-17}	7.2×10^{16}	256 bits	2.1×10^{9}
8 bits	|	1 to 4 bits	6.0×10^{-8}	1.7×10^{7}	128 bits	
n bits	↓	1 to $n - 4$ bits	2^{-4n+8}	2^{4n-8}	$16n$ bits	↓

Variations of Posit

Meanwhile, Lu et al. [73] have recently evaluated the use of posit representations in DNN applications and proposed and an interesting variation of the posit format, specifically, posits are allowed to underflow. This decision was justified by the fact that small values may be set to zero without penalizing the model performance. Although not formalized, this topic had already been addressed in [74].

More recently, in [75], another format was proposed, called Adaptive Posit. To better control the dynamic range, this alternative format introduced an additional hyperparameter, which can be a regime bias or a maximum regime length. Selecting the appropriate configuration, adaptive posits can represent floats, posits, or other tapered accuracy formats in between, whichever is more appropriate.

2.2.4 Posit Arithmetic Libraries

There has been a lot of research about designing parameterized posit arithmetic units to be synthesized for Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) [16, 21, 22, 74, 76–79]. More recently, [80] proposed a tensor unit with variable-precision posit arithmetic that can be directly used for DL applications with posits. However, at the time of development of this thesis, none of the mentioned hardware implementations is mature and flexible enough to build a complete DL framework with posits, as also noted by [81]. To overcome those obstacles, it arose the need to develop an integrated posit computing framework, where the posit arithmetic was simulated via software.

In [82], it was presented an extensive survey of many posit implementations, up until mid-2019. From those, the most notable software implementations available are: SoftPosit [83], Universal [84], and PySigmoid [85] – all of these made available as programming libraries.

SoftPosit [83] is a C library endorsed by the Next Generation Arithmetic (NGA) team [82], of which John L. Gustafson (proposer of the posit format) is a member. It offers fast and comprehensive functions to operate with posits and quires and it was already exhaustively tested. However, it does not support an arbitrary posit configuration, namely, it only supports posit(8, 0), posit(16, 1), posit(32, 2), and posit(n, 2) with $n = \{2, \ldots, 32\}$. Although the latter format would allow testing the new standard, it is slower than

the others since it uses 32-bits in the background to store all sizes.

Universal [84] is a C++ template library that supports any arbitrary precision posit and quires, with programmable capacity. This flexibility even allows to easily switch from the older standard and the newer one. Moreover, it comes with a complete validation suite that assures that the posit arithmetic is simulated correctly and its repository is still frequently maintained.

PySigmoid [85] is a Python library that also supports any arbitrary posit precision. However, while it has a very easy to use interface, it lacks performance when compared to the other libraries implemented in C and C++. Nonetheless, a particular advantage of using Python is that it can integrate very well with the current most popular deep learning frameworks.

Each of the detailed software implementations for the posit format has its advantages and disadvantages. Nonetheless, at the start of this work, the **Universal** library [84] was the chosen one due to its flexibility for any posit and quire configuration, as well as the provided comprehensive support for the main operators and functions specified by the Posit Standard. Although it was not anticipated, this flexibility would also allow to very easily implement the posit arithmetic according to the most recent standard [24] and, later on, evaluate this version. Furthermore, its implementation in C++ allows for faster executions and provides an easier integration with DL frameworks available for C++.

The Universal library implements posits and quires as C++ templated classes, whose arguments correspond to the *nbits* and *es* parameters for posits and *capacity* for quires. Internally, the fields of these formats are stored as bit vectors (`std::bitset`) and, whenever possible, everything is implemented without any intermediate float representation. As an example, declaring a 8-bit posit ($es = 0$) variable (`p`) equal to 1 is straightforwardly done using templates (see Listing 2.1).

Listing 2.1: Example of the declaration a posit(8, 0) variable (`p`) equal to 1 using the Universal library [84].

```
1 #include <universal/posit/posit>
2 using namespace sw::unum;
3
4 posit<8, 0> p = 1;
```

2.2.5 Energy Efficiency

According to the literature [78], it has been shown that posits can offer the same accuracy as IEEE 754 floats while using fewer bits. As a result, the simplest application of posits could be its use as a storage format only [86]. An example would be to perform the training of a DNN model using IEEE 754 floating-point and then store the model weights using posits, taking advantage of the more efficient usage of memory bits of this format. In fact, given the large dynamic range of low-precision posits, floats may be replaceable by posit16 or even posit8, greatly reducing the memory footprint.

However, a more interesting approach is to completely replace IEEE 754 floating-point, not only as a storage format but also for the computations. Preliminary studies indicate that the area and energy consumption of a posit compliant unit are comparable to its IEEE 754 compliant counterpart [22]. However,

since low-precision posit formats may be used to achieve comparable accuracy and dynamic range as larger IEEE 754 formats [78], they promote the opportunity of a greater exploitation for energy-efficiency. At last, [80] proposed a reconfigurable tensor unit leveraging the posit format and showed that it outperforms state-of-the-art tensor and Single Instruction, Multiple Data (SIMD) units.

2.3 Related Work

Low-precision arithmetic has been gaining a lot of attraction in machine learning. As a result, many implementations are already available that exploit 16-bit floating-point formats. For example, recent NVIDIA's Graphics Processing Units (GPUs) and Google's Tensor Processing Units (TPUs) both support the FP16 and bfloat16 formats. The IEEE 754 16-bit format is prone to overflowing, due to its limited numerical range, so it is usually implemented with special techniques, such as loss scaling [15, 87]. The bfloat16 format improved that problem by having an exponent of the same size as a FP32, at the cost of precision. However, some DL operations do not need as much precision, so its interesting to evaluate even smaller precisions, such as low-precision posits.

Since the posit format is fairly recent, most studies regarding its application to DNNs only address the inference stage [16, 23, 75, 78, 79, 88–90]. The models are first trained using floats and are later quantized to posits to be used for inference. Nevertheless, this stage tends to be less sensitive to errors than the training phase, making it easier to achieve good performance using {5..8}-bit posits. Conversely, exploiting the use of posits for DNNs training is a more compelling topic, given that not only is this the most computational demanding stage, but it usually involves more energy consumption.

The very first time posits were used for NN training was in [74]. In this work, a Fully Connected Neural Network (FCNN) was trained on a simple binary classification problem, using different posit configurations. The authors observed that {32, 16, 14, 12}-bit posits were able to train the model without penalizing the achieved model accuracy. Smaller formats such as posit(10, 0) and posit(8, 0) were also evaluated but showed an irregular convergence, which the authors suggested to be caused by the lack of underflow.

Later, in [21, 91], a FCNN was trained for a multiclass classification problem on the MNIST and Fashion MNIST datasets. This time, only 32 and 16-bit posits were evaluated, and, as expected, achieved equivalent results as to when 32-bit floats were used.

In [73, 92], CNNs were trained for the first time using the posit format. In these works, the networks were trained using an 8-bit posit everywhere except for the optimizer and the last layer, which used 16-bit posit. These works gave a better and favourable insight on the errors associated to using low-precision posits for DNNs. However, they still relied on floats for the entire first epoch and for the intermediate calculations. Even the implemented posit format was also slightly different, as it was able to underflow.

Most recently, a DL framework based on the posit format was proposed [81]. With this framework, named Deep PeNSieve, 32 and 16-bit posits were once more evaluated for end-to-end CNN training. Yet again, both posit precisions demonstrated an equivalent performance to 32-bit float. Nonetheless, the experiments that were executed with posit(8, 0) exhibited again its inability to converge when used

for DNN training. This framework was made available in [93] and was implemented in Python with TensorFlow [62], by emulating posits via software with the SoftPosit library [83].

However, the existing studies fail to train DNN with posits precisions smaller than 16-bits without affecting the achieved model accuracy. Moreover, given the novelty of the most recent version of the Posit Standard [24], all the research just mentioned refers to the older version of the standard [25]. Hence, as far as the author knows, this is the first available work that addresses the most recent version of the standard.

2.4 Summary

This chapter provides all the fundamentals to understand NNs and to train them. A brief explanation of the main layers and functions is given, followed by an overview of common CNNs models and datasets for object classification. Then, there is a discussion about computer number formats, focused on real number representations. The most popular format is the IEEE 754 floating-point, but it has some disadvantages that the novel posit format aims to improve. This novel format appears to be a good candidate for deep learning applications, particularly given the precision and range that small posits can offer.

Finally, follows an overview of the research work about deep learning training with low-precision formats. Preliminary studies show that DNNs may be trained using 16-bit posits, but fall short for smaller formats. In conclusion, there is still a need to experiment training DNN, end-to-end, with low-precision posits (< 16 bits), taking advantage of quires for accumulations, and preferably, according to the most recent Posit Standard [24].

Chapter 3

Proposed Deep Learning Posit Framework

Contents

Chapter 2 introduced all the theoretical fundamentals to train DNNs, focusing on CNNs for object classification. Furthermore, the posit format was introduced as a drop-in replacement for the IEEE 754 floating-point and the presented review of recent literature showed that posits are able to achieve the same model accuracy with a smaller memory footprint and energy consumption.

However, all the published results about end-to-end CNN training with posits refer to experiments where 16-bit posits were used. Given the accuracy requirements of NNs and the offered precision of small posits, it is appealing to also consider and study the training of DNNs with those low-precision representations. Moreover, given the novelty of the updated Posit Standard [24], there is not yet any results comparing its performance in DL with the previous one.

To respond to these matters, this chapter proposes a new framework that exploits posits for end-to-end DNN training and inference, describing the implementation of its main components. Notably, it aims to be flexible enough to support any posit configuration, either according to the most recent standard or the previous one. Finally, this framework should be as fast as possible to allow working with deeper models. Therefore, various techniques to improve the performance shall be analyzed.

3.1 Posit Neural Network Framework

As it was referred before, there is no framework available for DL training and inference with arbitrary precision posits. Hence, the first step is to evaluate the feasibility of extending an existing framework, so as to support the posit format. Conversely, building a new framework from scratch might be a better option, since it allows for better control of its inner operations.

3.1.1 First option: Extending an Existing Framework

PyTorch [37] and TensorFlow [62] are the two most popular frameworks for DL. However, the PyTorch's dynamic graph computation provides a more flexible implementation of custom functionalities, in addition to an easier debugging. Furthermore, the shallower learning curve and the fact that there has been a growing interest in this framework by the research community [67], has led to the decision to select **PyTorch** as the base framework.

C++ and Python integration

Although PyTorch's main interface is Python, an API for C++ is also available (**LibTorch**). This C++ API is particularly useful, since posits are to be simulated with the Universal [84] C++ library. With this combination, Python and PyTorch may be extended with C/C++ functions [94], promoting the combination of Python's easiness and simplicity with the fast and comprehensive C++ library for posits.

When extending Python with C++ code (as described in PyTorch's documentation [39]) the library pybind11 [95] is used, which is responsible for the mapping between the two languages. A simple test was performed and confirmed that it was possible to copy posit objects from C++ to Python and vice

versa. However, it was noted that while a posit(32, 2) in C++ occupied 8 bytes in memory[4], its copy in Python occupied 56 bytes, which could, eventually, be much slower when working with tensors with many more posits. Moreover, the posit configuration is chosen by C++ templates, which are only evaluated at compile-time and could prove problematic when binding with Python.

On the other hand, PyTorch provides a C++ API that offers the majority of its functionalities. Hence, instead of trying to extend Python with multiple C++ functions and variables, which would result in a more complicated implementation and, probably, incur on a performance overhead, it was decided to develop the proposed framework completely in C++. Nonetheless, there was still the problem of integrating the posit library with this DL framework.

Converting to Posit

The first step was to implement the conversion layers that would receive floating-point variables and convert them to posit variables. After that, all the computations could be performed according to the posit arithmetic. However, to declare the weights of a neural network, PyTorch requires them to be initialized as tensors, which only support the following predefined data types: floating-point, complex, integer, and boolean. Moreover, there is no easy method to extend the supported data types without modifying the low-level implementation of the framework.

One non-ideal alternative could be to cast the posit raw bit data to an integer value. For example, a posit(8, 0) equal to 1 would correspond to an 8-bit integer equal to 64, as seen in Equation (3.1). Note, however, that the computations wouldn't be performed with integer arithmetic, since this format would only be useful to "bypass" the supported data types and store posits in PyTorch tensors.

$$\text{posit}(8, 0) = (1)_{10} = (01000000)_2 \Rightarrow \text{int8} = 64. \tag{3.1}$$

Computing with Posit

The functions provided by PyTorch are not implemented to operate with posit arithmetic, thus it would be necessary to reimplement (from scratch) every function that would use this custom data type. That almost corresponds to reimplementing every necessary function to train and test NNs, which were presented in Section 2.1. At this stage, one question arose: if the majority of the functions necessary for NNs need to be reimplemented to support the posit data type, why not develop an entirely new DL framework?

3.1.2 Second option: Implementing a Framework From Scratch

Extending PyTorch to support the posit data format for NN training and inference either requires to change the low-level implementation of the tensor object, or to circumvent the supported data types by casting the posit variables to different types (like a memory copy, without rounding). Moreover, most of the functions associated with NNs need to be reimplemented to support the posit format. Therefore,

[4]Recall that the Universal library implements posits with a C++ class, hence the 8 bytes instead of only 4 bytes (32 bits).

it was decided to develop a new framework from scratch, since it would require practically the same amount of work. By not relying on PyTorch, this new framework can be more flexible when supporting this custom data type. Moreover, the implemented functions shall be able to fully exploit all the functionalities provided by the posit format, for example, the use of quires for accumulations. Furthermore, the posit variables shall be passed by reference instead of by value, thus decreasing the load associated with constructing posit objects repeatedly.

Hence, the developed open-source framework, named **PositNN** [96], aims to provide all the necessary functionalities to train and test DNNs using posits and quires, based on a header-only library. Different posit precisions are supported throughout the framework, which may be used to take advantage of uneven accuracy requirements. The implemented operators are those covered in the theoretical background presented in Section 2.1, whose functions and program flow are very similar to PyTorch's C++ API. In Listing 3.1, a simple example depicting the declaration of a 1-layer model is presented, to compare PyTorch and PositNN. As it can be seen, the overall structure and functions are very similar, the only difference being the declaration of the backward function, since the proposed framework does not currently support automatic differentiation.

Listing 3.1: Comparison of Pytorch C++ API (left) and the proposed framework PositNN (right) for the declaration of an 1-layer model. The `typename P` is used to represent the posit data type.

```
#include <torch/torch.h>


struct FloatNetImpl : torch::nn::Module{
    FloatNetImpl() : linear(10, 2){
        register_module("linear", linear);
    }

    torch::Tensor forward(torch::Tensor x){
        x = linear(x);
        return torch::sigmoid(x);
    }









    torch::nn::Linear linear;
};
TORCH_MODULE(FloatNet);
```

```
#include <positnn/positnn>

template <typename P>
struct PositNet : Layer<P>{
    PositNet() : linear(10, 2){
        this->register_module(linear);
    }

    StdTensor<P> forward(StdTensor<P> x){
        x = linear.forward(x);
        return sigmoid.forward(x);
    }

    StdTensor<P> backward(StdTensor<P> x){
        x = sigmoid.backward(x);
        return linear.backward(x);
    }

    Linear<P> linear;
    Sigmoid<P> sigmoid;
};
```

The overall procedure for DNN training and inference is shown in Figure 3.1. The proposed framework supports every illustrated stages, implemented with posit arithmetic. A comprehensive list of its functionalities is presented in Appendix A.1. The procedure of each stage illustrated in Figure 3.1 may be summarized as:

1. The **dataset** is loaded and converted to the posit format. Since PyTorch already had functions to load some common datasets, those were used before converting to posit.

2. The input samples are used for the **forward propagation**, resulting in the output/prediction. The

model is implemented using posits and each layer is able to use a different precision. The inference procedure ends with the output calculation.

3. Given the output of the forward propagation and the target values from the dataset, the **loss** value is computed, which measures how different the predicted values are from the desired ones. Next, the gradient of the loss, with respect to the last layer, is computed.

4. The loss gradient is then propagated through the entire network for the **backward propagation**, which can also use different precisions. This is slightly similar to the forward propagation, but in the reverse order and with the loss gradient as input. Each layer computes its output gradients, which will be used to calculate the gradient of its weights.

5. From the output gradients, the trainable layers then calculate the **gradients** with respect to their parameters. Once more, it is possible to use different posit precisions in this phase.

6. At last, having the gradients of each weight, the **optimizer** is responsible for updating the model parameters. The used algorithm will modify the weights in the direction of decreasing loss. The model then assumes these updated weights, thus finishing a training iteration.

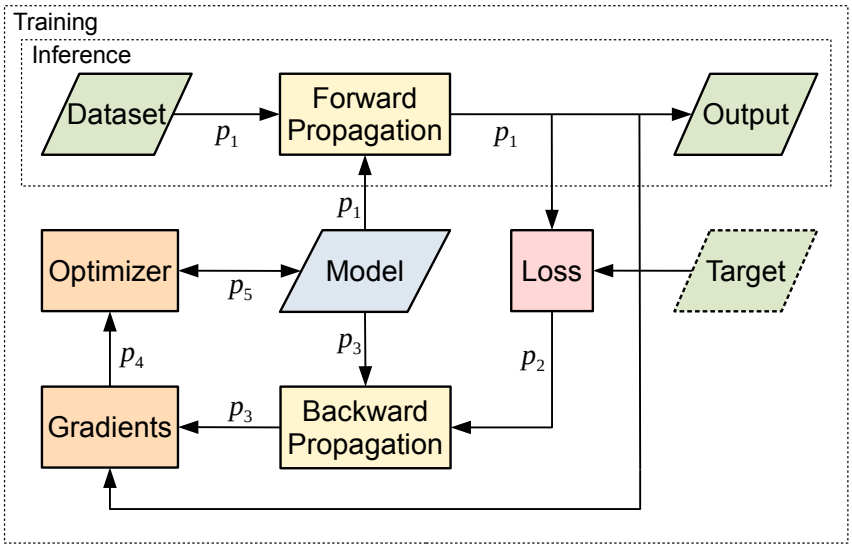


Figure 3.1: Block diagram of DNN training and inference procedures, starting at dataset. Parallelograms represent input or output of data. Rectangles represent functions/processes. The various arrows represent flow of data and each p_i, with $i = \{1..5\}$, represent some of the different posit precisions that may be used throughout the proposed framework. The colors denote blocks with similar or related calculations.

3.2 Posit Tensor

In NNs, data is usually manipulated as tensors, defined as multi-dimensional arrays. For example, a grayscaled image can be represented by a 2-dimensional array, mapping the height and width with rows and columns. If the images are colored, then there is an additional dimension, representing the color channel. Grouping various images (like in a mini-batch) originates an additional dimension, whose index

selects a particular image from the lot. In this example, the tensor would have a total of 4 dimensions. As one can see, representing data in multidimensional arrays is a very simple and extensible method to store information for NN computations.

As already mentioned in Section 3.1.1, PyTorch tensors do not natively support posit variables nor have a straightforward method to extend their supported data types. Therefore, it was necessary to either employ an existing library that implements tensors/multidimensional arrays composed of custom data types or to completely develop a new but equivalent data structure.

For this purpose, two linear algebra libraries were considered: Eigen [97] and Matrix Template Library (MTL) [98]. Both libraries offer vector and matrix structures with custom data types and offer a wide range of functions, all thoroughly optimized. Particularly, MTL was tested, verifying that it was very easy to declare a matrix of posits and to operate with it. However, MTL fails to support multidimensional arrays. Regarding Eigen, it supports tensors but, currently, it is not fully validated yet. It would be possible to construct tensors by implementing vectors of vectors, and so on, but it would defeat the purpose of using an optimized library. Moreover, to implement operations that accumulate with quires would require to rewrite those functions.

3.2.1 Custom Tensor Class

Given the limitations of existing libraries for tensor support, a custom tensor class was implemented. The problem at hands consists in designing a flexible structure able to store an arbitrary number of values and perform some basic manipulations. It should behave as a tensor and support posits variables.

Multidimensional Array

One possible approach could be to use C/C++ multidimensional arrays, as exemplified in Listing 3.2. However, this is not flexible enough, since the number of dimensions and their sizes need to be declared at compile time (constant expressions).

Listing 3.2: Declarating a 3-dimensional array to store posit(8, 0) variables. It could be used to store a colored image.

```
1 posit<8, 0> array3d[NCHANNELS][NROWS][NCOLS];
```

Nested Vectors

A similar approach would be to dynamically allocate those arrays, allowing to initialize their dimensions using any variable value as size. C++ Standard Library provides a useful container for dynamic arrays named std::vector<T> (T is the variable type), which automatically and safely handles the initialization and resizing of the array. This templated class could be used in cascade (nested) to declare a multidimensional vector, as shown in Listing 3.3. However, it still has some limitations, namely, the number of dimensions does not scale well and the memory is not guaranteed to be contiguous, which could slow down the computations.

Listing 3.3: Declarating a 3-dimensional vector to store posit(8, 0) variables. It could be used to store a colored image.

```
std::vector<std::vector<std::vector<posit<8, 0>>>> vector3d;
```

Vector with Row-Major Order

A more common approach for implementing a multidimensional structure in linear storage is to allocate its entire memory in one single array or vector and then compute the correct indices when accessing its elements, as shown in Listing 3.4. This method is more efficient, since the elements of the tensor are stored contiguously in memory. It is also more flexible in terms of multiple dimensions, since it is abstracted by the index function.

Listing 3.4: Declaration of a 1-dimensional vector to store posit(8, 0) variables as a 3D tensor and assigning 1 to the element at (i, j, k).

```
std::vector<posit<8, 0>> tensor3d(nchannels * nrows * ncols);
tensor3d[index({i, j, k})] = 1;
```

The function `index` is responsible for corresponding a set of multidimensional coordinates to a specific position of the stored vector. The order in which the elements are stored usually follows two possible conventions: row-major order and column-major order. In row-major order, consecutive elements of a row are stored next to each other in memory, which corresponds to increasing the indices of the tensor from the last dimension (right) to the first (left). In column-major order, the elements of a column are stored next to each other and it corresponds to increasing the indices of the tensor from the first dimension (left) to the last (right). An example of a $4 \times 3 \times 2$ tensor stored in a vector with row-major order is shown in Equation (3.2).

$$X_{ijk} = \left(\begin{bmatrix} x_{000} & x_{001} \\ x_{010} & x_{011} \\ x_{020} & x_{021} \end{bmatrix}, \begin{bmatrix} x_{100} & x_{101} \\ x_{110} & x_{111} \\ x_{120} & x_{121} \end{bmatrix}, \begin{bmatrix} x_{200} & x_{201} \\ x_{210} & x_{211} \\ x_{220} & x_{221} \end{bmatrix}, \begin{bmatrix} x_{300} & x_{301} \\ x_{310} & x_{311} \\ x_{320} & x_{321} \end{bmatrix} \right)_{4\times3\times2}, \tag{3.2}$$

$$\Rightarrow \{x_{000}, x_{001}, x_{010}, x_{011}, x_{020}, x_{021}, x_{100}, \ldots, x_{321}\}_{24}.$$

Proposed StdTensor Class

From the presented approaches, the best one for this specific application was the last presented – vector with row-major order – so it was the one chosen to implement a tensor container. The designed templated class, inspired by [99], was named StdTensor, since it only depends on functions from the C++ Standard Library. Its template argument defines the data type of the elements to store. Besides storing the vector that will be used to contain the multidimensional tensor, this class also provides some useful methods to declare new objects, to access them, and to operate with them.

When declaring a StdTensor object, one may do it with a vector whose size is obtained from the sizes of its dimensions, instead of having to specify the total size of the corresponding vector. The class automatically computes the size of the vector and the necessary strides to access each of its dimensions.

When accessing a tensor element, the user may also input its coordinates and the flattened index is then calculated "under the hood" from the strides of each dimension. Moreover, this class redefines the operators for the basic arithmetic operations: $+$, $-$, $*$, $/$. They can be used with two tensors, by performing an element-wise operation, or with a tensor and a scalar value. Listing 3.5 shows an example of a declaration of a StdTensor. For more details, its implementation is available in [96].

Listing 3.5: Declaration of a 3-dimensional StdTensor to store posit(8, 0) variables and assigning 1 to the element at (i, j, k).

```
#include <positnn/positnn>

StdTensor<posit<8, 0>> tensor3d({nchannels, nrows, ncols});
tensor3d[{i, j, k}] = 1;
```

3.2.2 Data structures conversion from PyTorch

Since PyTorch functions may still be used with PositNN, for example, to load a specific dataset, it is useful to have a function to convert a PyTorch tensor to a StdTensor data structure. PositNN offers such function, which given a PyTorch tensor, declares a StdTensor with the same shape, copies the original elements to the new StdTensor (converting them to the corresponding type), and returns the created object. The reverse function is also available, that is, converting from a StdTensor to a PyTorch tensor.

3.3 DNN Model Implementation

The main components of a DNN model are the layers that constitute it. In the proposed PositNN framework, layers are implemented as classes, which can be grouped in sequence to make up a model. Each model can also be considered itself as a layer, to be encompassed in another more complex model.

These layers are implemented as classes with two main methods: forward and backward – which are used during the respective propagation stages. In practice, each layer is responsible for taking an input, applying the forward or backward method, and passing the output to the next layer, so that, in the end, the model executed the forward propagation or the backward propagation.

Layers with trainable weights allow a model to be trained and to improve its performance. Particularly, this type of layers has a base class (Layer) to be derived from, which allows registering the associated weights and gradients, along with some other useful features. During training, the backward methods also call the functions that calculate the gradient of each weight. The trainable layers currently supported by this framework are: linear, convolutional, and batch normalization.

Layers without trainable weights are simpler to implement because they do not compute any gradients. Likewise, the activation functions are implemented similarly to non-trainable layers, since they do not have any weights and only need forward and backward methods. PositNN supports the following non-trainable layers: average pooling, maximum pooling, and dropout – and the activation functions: sigmoid, TanH, and ReLU. A complete list of functionalities is presented in Appendix A.1 (in appendix).

Bellow, some implementation details of the main layers are discussed. Each layer implements the corresponding calculations presented in the theoretical background in Section 2.1.2.

3.3.1 Linear Layer

To declare a linear layer, two arguments should be provided: input size and output size. From those, the associated weight matrix and bias vector are initialized and stored as tensors. The calculations performed by this layer are equivalent to matrix operations, hence these functions needed to be redefined by using the proposed StdTensor class.

Matrix Operations

Matrix operations, such as addition, subtraction, and operations between a matrix and a scalar value are implemented the same way as multidimensional tensors, that is, as element-wise operations.

More interesting is the implementation of the matrix multiplication, which is usually implemented by multiplying rows of the left operand by columns of the right operand. However, a more efficient version was implemented and later generalized. This version multiplies rows of the left operand by rows of the right operand, which ultimately corresponds to a matrix multiplication with the right operand transposed (AB^T). Accessing the tensors along the rows improves performance because there is a better locality of their elements of the cache memories during the dot products and the tensor indexing is simpler. To perform the normal multiplication, the right operand is first transposed and then the described function is used. The same rationale is applied to implement the matrix multiplication along the columns (A^TB), which will be useful to calculate the gradient of the Linear layer.

Another common optimization to the matrix multiplication algorithm is to use a technique called loop tiling/blocking [100], which consists in addressing each matrix as a group of submatrices/blocks. Although this should speedup the algorithm, because accesses to the cache memory become more efficient, it was not observed any significant improvements. Either the compiler was already performing a similar optimization or the simulation of the posit arithmetic made it ineffective.

Implementation Details

The implementation of this layer is slightly different from what would be expected from its theoretical formulation. The reason is that the input data is not a column vector, but, actually, a matrix composed by multiple row vectors, each corresponding to a different sample of the mini-batch. The same shape applies to the propagated error. These leads to some transposes in the matrices multiplications, which can be easily achieved with the variations described above. Listing 3.6 shows a simplified version of the operations performed by the Linear layer.

Listing 3.6: Simplified example of the main operations performed in a Linear layer.

```
1 // Forward propagation
2 output = matmul_row(input, weight) + bias;  // A * B^T + C
3
```

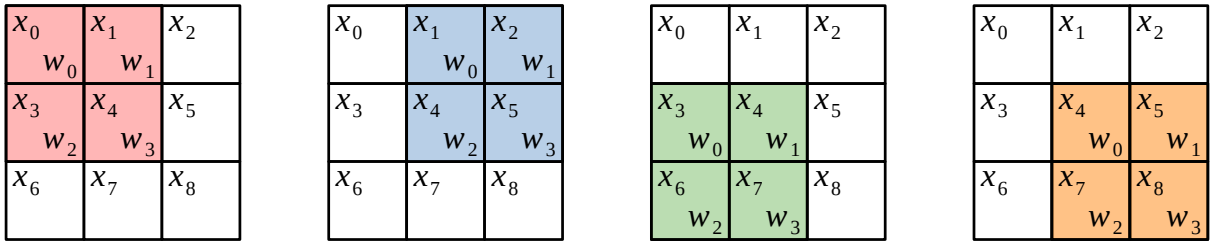
```
4 // Backward propagation
5 error_1 = matmul(error, weight);            // A * B
6
7 // Gradient
8 weight_gradient = matmul_col(error, input); // A^T * B
9 bias_gradient = error;
```

3.3.2 Convolutional Layer

The implemented convolutional layer performs a 2D convolution over an input composed of several planes (input channels). To declare a Conv2d layer, the following arguments shall be provided: number of input and output channels, kernel size, stride, padding, and dilation. From those, the trainable weight and bias tensors are initialized with the appropriate dimensions. The calculations performed by this layer are computed with convolution operations, thus, this operation was also implemented by using the proposed StdTensor objects.

Convolution Operation

The formulation of a 2D convolution was presented in Equation (2.8). Recall that many DL frameworks implement cross-correlation (the kernel is not flipped) but call it convolution. When both the input and the kernel are illustrated in a matrix form, the convolution operation is easily understood by the kernel sliding over the image. While it is simple to visualize such operation in a 2D space, when a tensor is stored in memory as a 1-dimensional array, most of its spatial continuity is lost, which makes it more difficult to compute the indices of the involved elements. In fact, although the StdTensor may be accessed as if it were a multidimensional array, the recurrent implicit calculation of the flattened indices would still affect the resulting computational performance, since it involves numerous integer multiplications. Figure 3.2 illustrates a convolution example, where the elements are represented with their flattened indices (each tensor is stored as a 1-dimensional vector).



(a) z_0 (b) z_1 (c) z_2 (d) z_3

Figure 3.2: Overlaps of the input and the kernel (windows) during a 2D convolution (flattened indices).

One common approach to implement the convolution operation is to use a technique called im2col. This consists in rearranging the input and kernel matrices, such that the convolution becomes equivalent to a matrix multiplication [33]. In particular, the input and output are flattened, and the input is multiplied by a sparse matrix A derived from the kernel. This approach makes the convolution as efficient as a

matrix multiplication. However, a downside is that the matrix A grows very quickly, since its dimensions correspond to $(H_{\text{out}}W_{\text{out}} \times H_{\text{in}}W_{\text{in}})$. Moreover, while this matrix improves the elements locality, it is expensive to construct it, since it needs to be updated every time the kernel changes and becomes more complicated for more complex convolutions. An example of a simple convolution of a 3×3 image with a 2×2 kernel using the im2col method is depicted in Equation (3.3).

$$x_{3\times3} * w_{2\times2} = z_{2\times2} \Leftrightarrow \underbrace{\begin{bmatrix} w_0 & w_1 & 0 & w_2 & w_3 & 0 & 0 & 0 & 0 \\ 0 & w_0 & w_1 & 0 & w_2 & w_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & w_0 & w_1 & 0 & w_2 & w_3 & 0 \\ 0 & 0 & 0 & 0 & w_0 & w_1 & 0 & w_2 & w_3 \end{bmatrix}}_{A_{4\times9}} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix} = \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix}. \tag{3.3}$$

To implement the convolution operation in a more efficient way (without computing and storing the expanded matrix A), a different approach was taken, consisting in creating a sort of "**recipe**" indices for each element of the output, that indicates which elements of the input and the kernel they will use.

Consider the convolution example illustrated in Figure 3.2. In this figure, the indices of the elements are in their flattened forms, which correspond to their original positions in the tensors. With this method, for each overlap, the corresponding input and kernel indices are orderly stored in arrays, as shown in Table 3.1. From these arrays, each pair of input and kernel blocks (represented with the same color) will originate one element of the convolution output. Thus, to compute the convolution, the function simply needs to loop through the indices arrays, multiply the input elements by the corresponding kernel elements, and sum the products of each group, which will result in the output elements. The indices intervals array serves to delimit the colored blocks that correspond to each output element. This method is properly defined in Algorithm 1.

The arrays that make up the "recipe" can be obtained using the ordinary convolution algorithm, but instead of computing it, the indices of the involved elements are stored. Since the convolution parameters and corresponding indices do not change, this only needs to be executed once at the start, so the performance of the "recipe" creation is not that important.

Table 3.1: Input and kernel "recipe" indices to perform the convolution illustrated in Figure 3.2.

input indices =	0	1	3	4	1	2	4	5	3	4	6	7	4	5	7	8
kernel indices =	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
indices intervals =	0				4				8				12			16
	z_0				z_1				z_2				z_3			

Algorithm 1: Calculating the convolution output using the window indices method ("recipe").

Input: input indices, kernel indices, indices intervals
Data: input, kernel
Result: convolution output
for $i \leftarrow 0$ **to** *output size*-1 **do**
  begin $\leftarrow$ indices intervals$[i]$;
  end $\leftarrow$ indices intervals$[i+1]$;
  for $j \leftarrow$ *begin* **to** *end* **do**
    input idx $\leftarrow$ input indices$[j]$;
    kernel idx $\leftarrow$ kernel indices$[j]$;
    output$[i] \leftarrow$ output$[i]$ + input[input idx] $\times$ kernel[kernel idx] ;
  end
end

At its core, this method is similar to im2col while using much less memory, since the total size of the arrays will be at most $(H_{\text{out}}W_{\text{out}} \times (2 \times H_w W_w + 1) + 1)$, which, for most cases, is smaller than matrix A. It is also easier to implement for convolutions with a non default stride, padding, or dilation.

Implementation Details

With the convolution operation defined for the proposed StdTensor objects, the convolutional layer can be easily implemented. As stated in Section 2.1.2, the backpropagation can be computed with a transposed convolution or a convolution with padding and dilation. The latter convolution might seem more expensive, but the "recipe" method avoids unnecessary calculations (e.g. padded regions, indices calculations, etc.), therefore, the two operations are equivalent in terms of computations. Recall that the convolution performed for the weight gradient is slightly different because it considers each channel of the input and error tensors separately. The simplified implementation of this layer is shown in Listing 3.7.

Listing 3.7: Simplified example of the main operations performed in a Convolutional layer (Conv2d).

```
// Forward propagation
output = convolution2d(input, weight, bias,
            stride, padding, 1, dilation);

// Backward propagation
rotated = rotate_weight(weight);
empty = StdTensor<T>();
error_1 = convolution2d(error, rotated, empty,
            1, (kernel_size-1)*dilation-padding, stride, dilation);

// Gradient
weight_gradient = convolution2d_gradient(input, error,
                    stride, padding, dilation);
bias_gradient = sum_last2(error);   // sum elements of last 2 dimensions
```

3.3.3 Pooling Layers

The pooling layers perform a 2D operation over an input composed of several input planes. Similarly to the convolution operation, the "recipe" method was also used to avoid repeatedly computing the kernel windows and respective indices. This way, each output element knows exactly which input elements it depends on.

Regarding the average pooling layer, its backward propagation is simple to implement by using the same method but in the backward direction, that is, relating each input element to the corresponding output element. Since each input element knows which output elements it affects, it simply has to sum the errors at those positions and divide by the window size.

Conversely, the maximum pooling layer requires some additional information. The windows indices can be calculated in a similar manner, but, for the maximum pooling operation, each output element is affected by only one input element (the maximum element). For the backward propagation, it is necessary to know which input elements were used, thus, the indices of the maximum values are registered during the forward propagation. With those indices, it is trivial to propagate the error, by assigning those values to the elements that were maximums (the rest is assigned to 0).

However, when the maximum pool windows overlap (stride $<$ kernel size), there might be some values that are the maximums for more than one window, thus, in the backward propagation, the corresponding error values need to be added together. Although the indices of the maximum values are registered, they are not stored in any particular order that facilitates the addition of the error values in case of overlap, which would be desirable for a more efficient implementation. To gather the common indices of the elements to be added, they are organized in a hash table using the std::unordered_map container, which has an average cost for search, insert and delete of O(1). This data structure is then used to backward propagate the error in cases of overlap.

3.3.4 Activation Functions

The activation functions were implemented similarly to layers and not as stateless functions, since they needed to keep some information during the forward propagation to be later used for the backward propagation. For the forward propagation, they take as input the previous layer output, they apply the respective non-linear function, and output the result. For the backward propagation, they first calculate the derivative of the non-linear function evaluated at the input (the intermediate output z^l, stored in the forward propagation), and then return the element-wise product of that derivative with the propagated error (see Equation (2.6)).

3.3.5 Save and Load

After training a model, it is useful to be able to save it and load it for later use. The proposed framework provides functions to achieve that. When a model is saved, the weights of the trainable layers are stored in a binary file, along with the corresponding posit configuration and tensors dimensions. A model may

later be loaded from this file. Hence, thanks to the adopted numbering format representation, a model saved using posit(8, 0) will occupy about $4\times$ less memory than it would if it used a 32-bit format.

Nevertheless, although posits may have a size that is not a multiple of 8, 0s are appended to each posit value when storing to a file, in order to store them using a byte granularity. Naturally, for posits whose size is not a multiple of 8, a more storage efficient approach would be to pack more than one posit per byte and/or split them between various bytes. However, that was not yet implemented[5].

3.3.6 Loading a PyTorch model

Despite the ability to load a model obtained using PositNN, the proposed framework can also load or use a PyTorch model. If that model was saved in a file using PyTorch, then this framework is used to first load that model. Then, having that model loaded with 32-bit float, the presented functions to convert from PyTorch tensors to StdTensors are used to copy and convert the model weights to the equivalent PositNN model, which can use posits of arbitrary precision. This functionality is particularly useful to train a model using 32-bit float and then evaluate the already trained model using posit.

3.4 Loss Functions

Although this model definition is already enough to test a DNN for inference, it is still not enough to implement a training procedure nor even evaluate how far is the output from the target. For that purpose, loss functions need to be implemented. To achieve that, the PositNN framework defines a base class (Loss) that is meant to be derived from when defining a specific loss function. To calculate its value, the user only has to initialize the object with the output and target values. To perform the backward propagation, the backward method is called with the DNN model as an argument, which then takes care of calculating the loss gradient and propagating it. The supported loss functions are: MSE and Cross Entropy – by implementing the equations presented in Section 2.1.3. Since the Cross Entropy loss is more complex and it is also the one that is most used (given that this work focused on image classification), its implementation details are presented below.

3.4.1 Cross Entropy loss function

As it was referred before, the Cross Entropy loss function may be calculated with Equation (2.20). However, the softmax calculation is prone to overflow or underflow in the exponential functions (prone to saturate when using posits). Fortunately, the softmax function has an identity property that can be used

[5]For the performed tests, the model weights were stored using 16-bit posits, so it would not make any difference.

to shift the input and avoid these issues [101]:

$$
\begin{aligned}
S\left(a^{\ell}\right)_i &= \frac{\exp\left(a_i^{\ell}\right)}{\sum\limits_{j \in \text{last layer}} \exp\left(a_j^{\ell}\right)} = \\
&= \frac{\exp\left(a_i^{\ell}\right)}{\sum\limits_{j \in \text{last layer}} \exp\left(a_j^{\ell}\right)} \cdot \frac{\exp(-c)}{\exp(-c)} \\
&= \frac{\exp\left(a_i^{\ell} - c\right)}{\sum\limits_{j \in \text{last layer}} \exp\left(a_j^{\ell} - c\right)} = S\left(a^{\ell} - c\right)_i ,
\end{aligned}
\tag{3.4}
$$

where c is an arbitrary constant. If c is chosen to be the maximum value of a^{ℓ}, then the arguments of the exponential are all less than or equal to 0 and the value of the exponential will never be greater than 1.

As for the calculation of the loss value, it is still necessary to calculate the negative log likelihood. Using this softmax identity, the equation becomes

$$
\begin{aligned}
L\left(a^{\ell}, \hat{y}\right) &= -\log\left(S\left(a^{\ell}\right)_{\hat{y}}\right) = -\log\left(S\left(a^{\ell} - a_{\max}^{\ell}\right)_{\hat{y}}\right) = \\
&= -\log\left(\frac{\exp\left(a_{\hat{y}}^{\ell} - a_{\max}^{\ell}\right)}{\sum\limits_{j \in \text{last layer}} \exp\left(a_j^{\ell} - a_{\max}^{\ell}\right)}\right) \\
&= -\left(a_{\hat{y}}^{\ell} - a_{\max}^{\ell}\right) + \log\left(\sum_{j \in \text{last layer}} \exp\left(a_j^{\ell} - a_{\max}^{\ell}\right)\right).
\end{aligned}
\tag{3.5}
$$

Its implementation consists in first finding the maximum value of a^{ℓ}, calculating the exponential of every element of a^{ℓ} subtracted by the maximum value, summing them, calculating the logarithm of the sum, and, finally, subtracting $\left(a_{\hat{y}}^{\ell} - a_{\max}^{\ell}\right)$.

Regarding the backward propagation of the loss, the softmax identity is still valid, thus, its calculation is equivalent to

$$
\frac{\partial L}{\partial a_i^{\ell}} = \begin{cases}
\dfrac{\exp\left(a_{\hat{y}}^{\ell} - a_{\max}^{\ell}\right)}{\sum\limits_{j \in \text{last layer}} \exp\left(a_j^{\ell} - a_{\max}^{\ell}\right)} - 1 & \text{for } i = \hat{y}, \\
\dfrac{\exp\left(a_i^{\ell} - a_{\max}^{\ell}\right)}{\sum\limits_{j \in \text{last layer}} \exp\left(a_j^{\ell} - a_{\max}^{\ell}\right)} & \text{otherwise.}
\end{cases}
\tag{3.6}
$$

Basically, its implementation consists in calculating the fraction above for every element and subtracting 1 if the element corresponds to the target class $\hat{y}$. To avoid repeating most of the computations, the exponentials of $\left(a_j^{\ell} - a_{\max}^{\ell}\right)$ and their sum are stored during the computation of the loss value, so that, in the end, only a division and a subtraction are performed.

3.5 Optimizer

The final and essential step to train a NN is the optimizer, which is responsible for updating the weights of the model in order to improve its performance. The developed PositNN framework provides a base class

(Optimizer) to be derived from when implementing a specific optimization algorithm. When initialized, this class will receive and keep a list of the model parameters, that will contain the registered weights and corresponding gradients, passed by reference. Then, the class will have a method (called step) responsible for applying the optimization algorithm for each model parameter.

Currently, the proposed framework only supports the SGD optimizer, although it can be easily extended with other optimizers. When selecting which optimizer to implement, the following alternatives were tested with PyTorch: SGD, RMSprop, and Adam. For LeNet-5 trained with MNIST during 10 epochs, the SGD was the one that achieved the highest accuracy ($\sim 99\%$). Surprisingly, RMSprop only achieved an accuracy of $\sim 90\%$ for that number of epochs. When trained with Adam, the accuracy increased faster, but it stayed at $\sim 96\%$. In the end, SGD seemed to be a safer choice, with better results and simpler implementation. Notwithstanding, the other algorithms could possibly achieve a higher accuracy if trained during more epochs.

3.5.1 Stochastic Gradient Descent (SGD) implementation

SGD is one of the algorithms that were briefly presented in Section 2.1.4 to perform a gradient descent optimization. In the proposed framework, it is implemented as a class derived from the Optimizer class and overrides a method named 'step', which will perform the update of the weights.

This optimizer was implemented similarly to PyTorch [37, 39], that is, using a set of formulas slightly different from those shown in Equation (2.22). One difference is the learning rate that is used for the entire momentum term (velocity) and not only for the gradient. This implementation also includes a weight decay factor, λ_1, also known as L2 regularization, which is an additional term added to the gradient before the weight update that will cause the weights to exponentially decay to 0. Additionally, it also includes a dampening value, λ_2, which is used for the momentum term. The resulting equations are the following:

$$G(t) = \left.\frac{\partial \mathcal{R}}{\partial w}\right|_{w(t)} + \lambda_1 w(t), \tag{3.7a}$$

$$v(t+1) = \gamma v(t) + (1-\lambda_2) G(t), \tag{3.7b}$$

$$w(t+1) = w(t) - \eta v(t+1), \tag{3.7c}$$

where $G(t)$ is an auxiliary term equal to the gradient added to the weight decay term. Moreover, this optimizer may also receive a boolean to enable the NAG technique, which changes Equation (3.7c) to

$$w(t+1) = w(t) - \eta \left(G(t) + \gamma v(t+1)\right). \tag{3.8}$$

One known issue of the SGD is its constant learning rate. If too small, the optimizer will take a long time to converge and, if too large, might not be able to improve the model accuracy above a certain level (the model needs a sort of fine-tuning). Thus, a common technique is to implement a learning rate scheduler, which will adapt the learning rate throughout the training process. By using a large learning

rate for the initial epochs (to speed up convergence) and decreasing it as the model accuracy converges (to fine-tune), it usually provides good results. The options received by this optimizer are stored as public class attributes, thus, the user can change them anytime, particularly, during the training process.

3.6 Parallelization of the proposed framework

Most deep learning calculations consist of basic linear algebra operations. However, they usually involve lots of data, making them very computationally expensive. To overcome this problem, many DL frameworks allow their users to accelerate those calculations in GPUs [61] and to perform them in parallel, thus greatly increasing the computational performance. However, the Universal software library used to simulate posit arithmetic does not support GPU acceleration. Nevertheless, although PositNN will strongly rely on the CPU, there is still space for improvement. Besides compiling it with full optimization (gcc option -O3), parallel computing will also greatly improve the framework performance. The following subsections describe the adopted methodology to parallelize the implemented framework.

3.6.1 Profiling

Before trying to improve the framework performance, it is important to know which functions take the longest to execute. To do so, a profiling tool named GNU Profiler (gprof) [102] was used to determine the slowest and most frequent parts of the program. In particular, this tool generates two tables: a flat profile that shows the total amount of time the program spent executing each function, and a call graph that shows how much time was spent in each function and its children.

This profiler was applied to evaluate the training of a 5-layer CNN, with 1024 samples and using posit representations. The top 5 entries of the flat profile and their corresponding percentage of execution time were: quire_mul ($19.5\,\%$), scale ($16.5\,\%$), add_value ($9.5\,\%$), subtract_value ($8.3\,\%$), extract_fields ($8.2\,\%$). All these functions are quire or posit related, which was expected given that they are being simulated via software. Hence, since the main objective is to improve the proposed PositNN framework and not exactly the posit library, it is necessary to know where these functions are being called. The top 4 are quire related, thus, the slowest parts of the program will probably correspond to the linear and convolutional layers, which is where the quire is mainly used. This is backed up by the call graph, where it can be seen that the most time-consuming functions are in the training loop, specifically, in the backpropagation.

3.6.2 Implementation details

In order to improve the program performance, the most time-consuming functions were implemented with multithreading support, that is, their load was divided by multiple threads. These threads were implemented with std::thread, a class provided by the C++ Standard Library, which ensures that it is portable and compatible with any system that supports C++11. To select the maximum number of threads to use, the user only needs to specify that option during the compilation.

The first functions to be implemented with multithreading support were the matrix operations. By doing so, a user will benefit from the parallelization of these functions in all parts of the program that use matrix operations. Considering, for example, a matrix multiplication, each output element can be calculated independently of the others, therefore, the output matrix can be divided into different regions to be calculated in parallel in separate threads (see Figure 3.3b).

A more general approach comes from observing that each layer tends to operate on a mini-batch, instead of one sample at a time. Usually, the operations performed on a sample are independent of the other samples, thus, each mini-batch can be divided into multiple subgroups of samples that are processed in parallel in different threads (see Figure 3.3c). This load balancing is usually simpler, since, dividing by samples can be easily achieved by indexing the first dimension of the tensor; while dividing by output elements (like in the previous approach) will frequently require tinkering with more dimensions. This approach was employed to parallelize most layers.

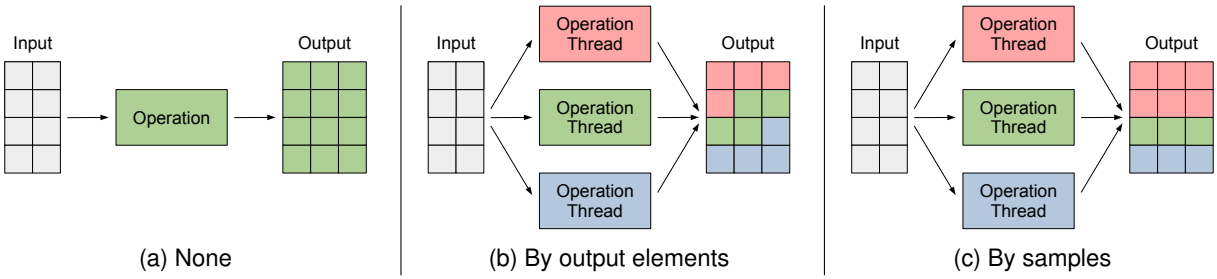


Figure 3.3: Different ways to divide the load of a layer in multithreading. Each row will correspond to a different sample and each column to a different output index (neuron).

The considered optimizer was also parallelized since the model weights can be updated independently. With this implementation, the majority of the functions are executed in multiple threads and the program can take full advantage of the several cores that integrate the CPU.

Finally, in what concerns concurrency problems, it should be recalled that the adopted parallelization is applied over independent calculations, thus avoiding any race condition between threads. Moreover, since the parallelization is implemented per operation, layer, or stage, then the main thread only has to wait for each worker to finish its assigned computations and synchronizes with a barrier.

3.7 Summary

In this chapter, PositNN – an entire framework for DNN training and inference using posits – was proposed. First, it was discussed how extending an existing DNN framework to support posit arithmetic would require the reimplementation of a large portion of its functions, possibly having to use some additional tricks to circumvent its limitations. Thus, it was comparable to implementing a separate framework from scratch, in terms of the amount of work. It was therefore decided to develop a new framework for DNNs and the first implemented component was the multidimensional tensor class that would store posit values. Based on this class, common NN layers and functions were implemented, whose main details were discussed. The chapter ends with a discussion about the framework performance, particularly, how the most time-consuming functions were parallelized with multithreading support.

Chapter 4

Training with Low-Precision Posits

Contents

The previous chapter proposed the PositNN framework, aiming the evaluation of DNNs when implemented with the posit numbering system. In particular, despite some related work already reporting that 16-bit posits can directly replace 32-bit floats for DNN training, the usage of even smaller posit precisions has yet to be thoroughly studied. This chapter aims to give a better insight on DNN training using posits, specifically, how much can the posit precision be decreased without penalizing the achieved model accuracy. Moreover, some particularly useful techniques to improve the numerical errors when using low-precision formats are presented, along with some implementation details. Those techniques are: accumulation with quires, usage of different/mixed precisions, and usage of posits underflow.

In accordance, the structure of this chapter will be similar to an iterative/optimization process, where the goal is to decrease the used posit precision while maintaining an accuracy similar to 32-bit float. **The performed experiments consist in training the CNN LeNet-5 on the Fashion MNIST dataset for 10 epochs, followed by a thorough evaluation of the model performance**[6]. PositNN will be used to train with the posit format, while PyTorch will be used to train with floats and used as the reference implementation. More details of the training process are shown in Table 5.2 (in appendix). Due to the randomness of this process (e.g. network initialization, shuffled dataset), whenever relevant, the experiments were repeated 2-3 times and averaged.

The implementation of a complete training example of LeNet-5 on Fashion MNIST using PositNN is presented in Appendix A.2.

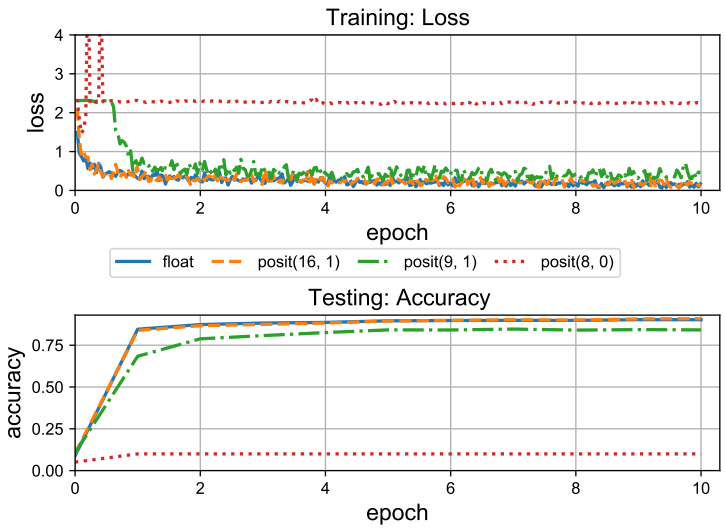
4.1 Minimum Posit precision

As referred in Section 2.2.3, posits are defined by two parameters: *nbits* and *es*. However, despite the recommended configurations presented in Table 2.3, it is interesting to further investigate how the DNN performance is affected by different combinations of these two parameters. PositNN is the appropriate tool for such study, since it allows training DNN with posits of any arbitrary precision. In this first experiment, a model was trained using the same posit precision in all stages and without using quires in the accumulations. The obtained results are presented in Figure 4.1 for different configurations.

As described in similar studies [21, 81], the recommended posit(16, 1) format can seamlessly substitute the 32-bit float. However, as the number of bits decreases, the situation becomes more intricate. In particular, posits with the exponent size $es = 0$ start to fail, even when 16 bits are used. The same is observed for smaller posits, particularly, the 10 and 9-bit posits are unable to train when $es = 0$ (a 10% accuracy is equivalent to randomly classifying a 10-class dataset). Regarding the 8-bit posit, despite the configuration with $es = 0$ being the recommended [25], the model is unable to train with any *es*. In fact, considering that most data in DNNs have normal distributions with small variances [73], narrow posits with $es = 0$ do not have a wide enough dynamic range to represent those smaller values.

Hence, despite the promising gains in terms of the used hardware and energy resources, the adoption of such a numbering system that used up to one quarter (8-bits) of the number of bits used by a

[6] Small accuracy differences (< 1%) were assumed to be caused solely by the randomness of the training process and not exactly by lack of precision of the numerical format.



Format	Accuracy		
	$es = 0$	$es = 1$	$es = 2$
Float (FP32)		$\mathbf{90.28\%}$	
Posit16	88.23%	90.87%	90.55%
Posit12	66.66%	90.15%	90.26%
Posit10	19.86%	88.15%	88.52%
Posit9	11.65%	84.65%	82.50%
Posit8	10.00%	12.54%	12.55%

Figure 4.1: Evaluation of how different posit precisions compare to 32-bit float for DNN training. On the left, it is presented a plot of the training loss and testing accuracy of a model trained with various posits. The table on the right shows the achieved accuracies when using a with different number of bits and exponent sizes. The float implementation was used as reference.

IEEE 754 float (32-bits) is a challenging target that still requires some further investigation in order to identify some complementary techniques that have also to be considered.

4.2 Posit quires for intermediate accumulation

Although the numerical error produced in a simple operation performed with low-precision posits might not be that significant, when accumulated in successive operations (such as in matrix multiplications or convolutions), it might severely undermine the accuracy of the result. To mitigate the error introduced in such large accumulations, the Posit Standard defines the use of a quire format, designed to accumulate exact dot products. This mechanism is particularly useful in this specific application, since it avoids the many quantization errors that would occur during the accumulation, by ensuring that the rounding to posit is only applied in the end. To better grasp how useful this mechanism is, suppose the following hypothetical accumulation:

$$2 \times 10 + 2 \times 10 + 2 \times 10 + 2 \times 2 = 64. \tag{4.1}$$

If it were to be performed using posit(8, 0), all those values (2, 10, and 64) could be represented without error. However, each term (2×10) would be rounded to 16 and both ($32 + 16$) and ($32 + 4$) would also be wrongly rounded to 32, as shown in Equation (4.2):

$$2 \times 10 + 2 \times 10 + 2 \times 10 + 2 \times 2 \tag{4.2a}$$

$$\rightarrow 16 + 16 + 16 + 4 \tag{4.2b}$$

$$\rightarrow 32 + 16 + 4 \tag{4.2c}$$

$$\rightarrow 32 + 4 \tag{4.2d}$$

$$\rightarrow 32 \tag{4.2e}$$

which corresponds to an error of 50% with respect to the expected value. However, if the accumulation were to be performed using a quire, the result would exactly correspond to the expected value. Although this is a specific example to demonstrate how low-precision posits may fail, it clearly shows how better quires are for large accumulations.

To accommodate this useful mechanism, PositNN framework provides support to perform accumulations with or without quires, specified through a compilation option. Moreover, the posit library also allows choosing how large is the carry guard field (capacity) used for the quire format. To be compliant with the Posit Standard [25], this field should have a size[7] of *nbits* $- 1$.

To evaluate the effect of accumulating with/without quires, the same model was once more trained with 8-bit posits. The results of training with and without quires are shown in Table 4.1. Although the model still does not achieve an acceptable accuracy, it shows a slight improvement when quires are used. However, when evaluating a previously trained model (using floats) only in terms of the subsequent inference phase computed with 8-bit posits, all the tests presented in Table 4.2 show a better accuracy when quires are used. This also shows how less sensible to numerical error the inference process is when compared to training, since it was already able to achieve good results with 8-bit posits.

Table 4.1: Training and testing a model using 8-bit posits while accumulating with and without quires.

Format	Accuracy	
	Without quire	With quire
Float (FP32)	$\mathbf{90.28\%}$	
Posit(8, 0)	10.00%	15.12%
Posit(8, 1)	12.54%	15.41%
Posit(8, 2)	12.55%	19.39%

Table 4.2: Testing a model using 8-bit posits while accumulating with and without quires. Model pre-trained with floats.

Format	Accuracy	
	Without quire	With quire
Float (FP32)	$\mathbf{90.28\%}$	
Posit(8, 0)	88.80%	89.93%
Posit(8, 1)	89.14%	90.23%
Posit(8, 2)	88.71%	90.05%

4.3 Mixed Precision Configurations

Since different stages of DL may have different accuracy requirements, it is interesting to analyze models that use different posit precisions throughout their computations, which may allow training with smaller posit precisions in some stages or network layers. To satisfy this feature, the designed tensor class, StdTensor, allows to declare tensors of arbitrary precisions and to convert them to other data types. With such capability, computations with mixed precision can conveniently be implemented.

From now on, the **used nomenclature for the various configurations** represents the posit precisions used in different stages: Optimizer (O), Loss (L), Forward propagation (F), Backward propagation (B), and Gradients calculation (G). The number next to these letters represents the number of bits of the posit used, and the subscript q indicates that quires were used. Both exponent sizes of 1 and 2 provided good results for DNN training and since the most recent Posit Standard [24] fixes *es* $= 2$, the next experiments will also use that setting (unless specified otherwise)[8].

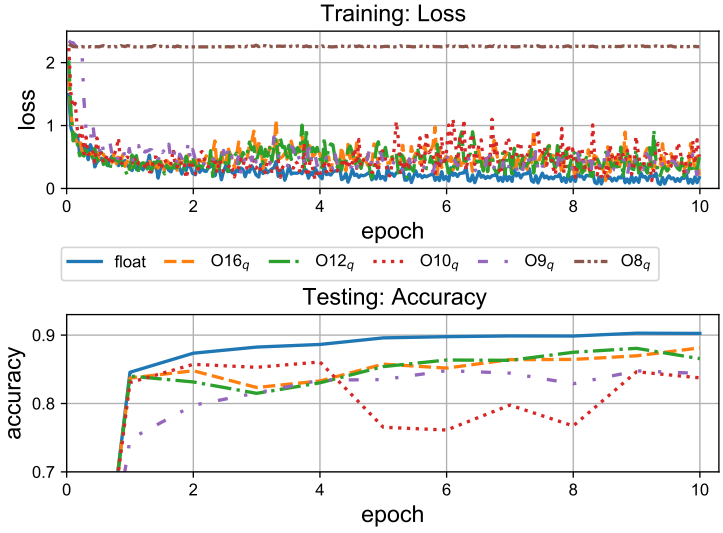
[7] In order to implement the standard quire with the Universal library, the size of the capacity field should actually be set to *nbits* $- 2$, since it implicitly implements the integer field with an extra bit.

[8] For example, the configuration O16-L12-FBG8$_q$ means: optimizer (O) with posit(16, 2), loss (L) with posit(12, 2), forward (F),

4.3.1 Optimizer Precision

This step is very sensitive to numerical precision, which makes this technique particularly important. In fact, as the model converges and the gradients decrease, the weight update (gradient multiplied by the learning rate) might become too small to be represented with a low precision format. Moreover, the ratio between the weight value and the weight update is usually large so, even if the weight update is representable, the utilized format might not have enough resolution to represent the optimizer step result. This representation problem, with the gradients becoming too small, is commonly referred to as the vanishing gradient problem. However, the opposite problem, known as the exploding gradient problem, may also occur. As a consequence, the error introduced by a low-precision format might unstabilize the network and cause its weights to "explode"/diverge to an unrecoverable state. In [15, 73], low-precision floating-point and posit representations are used in the DNN training phase. In particular, in order to prevent any model accuracy loss, a higher precision primary copy of the weights is kept and used in the optimizer step.

To evaluate the consequences of this problem, a CNN model was trained using 8-bit posits everywhere except for the optimizer, which used a higher precision. The results of using an optimizer with {16, 12, 10, 9, 8}-bit posits are presented in Figure 4.2. Impressively, by making only the optimizer to use a higher precision than the rest of the stages, which use 8-bit posits, it showed to be enough to allow the model to train and achieve a proper accuracy. Although it is not at the same level as a model trained using floats ($\sim 90\%$), the model accuracy comes very close when only 16 or 12-bit posits are used ($\sim 88\%$).



Format	Accuracy
Float (FP32)	$\mathbf{90.28\%}$
O16-LFBG8$_q$	88.14%
O12-LFBG8$_q$	88.06%
O10-LFBG8$_q$	86.07%
O9-LFBG8$_q$	84.80%
O8-LFBG8$_q$	19.39%

Figure 4.2: Evaluation of how the model accuracy changes when the model is trained using a higher posit precision for the optimizer and 8-bit posits everywhere else. Table with the accuracies achieved using various precisions for the optimizer. Using the results obtained with 32-bit float for reference.

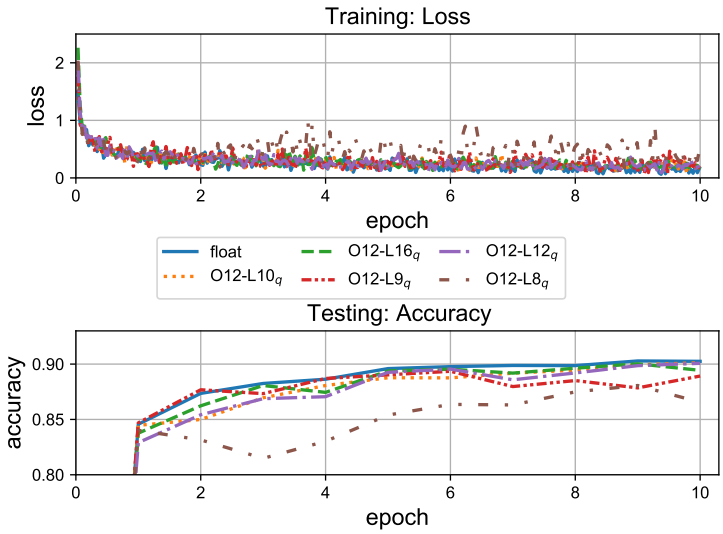
backward (B), and gradient (G) with posit(8, 2), and accumulating with quires (q).

4.3.2 Loss Precision

The experiments presented in the previous subsection have shown that training a DNN model using almost exclusively 8-bit posits (except for the optimizer, which uses a higher precision), was enough to boost the model accuracy to a value very near to the obtained when 32-bit floats were used. In [103] and [104], which both train DNNs with low-precision floating-point numbers, the authors observed that the last layer of the model was very sensitive to quantization. More precisely, when compared with the forward, backward, and gradient calculations, the softmax function requires more precision. For posit numbers, Lu et al. [73] also found that the model accuracy was very sensitive to the precision of the last layer.

Having the previous observations in mind, the next experiments consisted in training the model using a higher precision for both the optimizer and the loss layers. PositNN implements the softmax operation within the Cross Entropy loss function, therefore, the precision of the loss function was increased instead of the last layer. At this respect, it should be noted that increasing the precision of the loss function will not directly affect the model accuracy because it is not used for inference. However, it will greatly affect the training process because it is where the backpropagation starts.

The results of these experiments are shown in Figure 4.3. The first experiment, where the loss was computed with 16-bit posits, immediately showed that using the optimizer and loss functions with higher precisions is enough to train a model and to achieve an accuracy equivalent to when 32-bit floats are used everywhere. Moreover, it can be observed that the accuracy stays practically unaffected when the model uses as few as 12-bit posits for the optimizer, 9-bit posits for the loss, and 8-bit posits everywhere else.



Format	Accuracy
Float (FP32)	$\mathbf{90.28\%}$
O12-L16-FBG8$_q$	90.03%
O12-L12-FBG8$_q$	90.07%
O12-L10-FBG8$_q$	90.13%
O12-L9-FBG8$_q$	89.35%
O12-L8-FBG8$_q$	88.06%

Figure 4.3: Evaluation of the accuracy achieved by a model trained using higher precision posits for the optimizer and loss and 8-bit posits everywhere else. The optimizer uses 12-bit posits while the loss is tested with various precisions. Table with the achieved accuracies and using the results obtained with 32-bit float for reference.

Naturally, the overhead of using a higher precision for the optimizer and loss will naturally depend on the model and functions used. However, for this model in particular, these stages represent only about

5 % of the total computations. The majority of the computations correspond to the forward propagation, backward propagation, and gradients calculations, which can be computed with 8-bit posits. Moreover, even when higher precision is required, 16-bit posits are enough to replace 32-bit floats. Note also that increasing the precision of the loss function is not as computationally expensive as increasing the precision of the last layer would be, since the latter usually involves many more operations.

4.3.3 Implementation Details

There are some situations where the use of mixed precision is not straightforward, given the framework implementation. One such example is a model whose weights use different posit precisions per layer. This is problematic because the optimizer takes a list of the model weights as an input argument, which will influence the precision used to perform the computations. Ultimately, each posit configuration would correspond to a different data type, and, consequently, this list would be a heterogeneous container. Although C++ provides some data types that could be used to contain different data types (e.g. std::any, std::pair, std::tuple, std::variant, etc), they fail to be flexible and general enough for the framework requirements.

To overcome this problem, a new container, named MixedTensor, was developed. The idea behind this container is to provide a tensor of a specific data type, which also provides copies in other arbitrary types. Hence, in scenarios where a homogeneous vector of objects is necessary, as just explained for the optimizer, declaring the weights with this MixedTensor allows to then construct that vector with the main versions of the tensors (which were all declared with the same data type). Then, for the individual computations of each layer, the corresponding copies are used, which can have an arbitrary data type. Every time the main tensor changes, its copies are seamlessly and automatically updated.

However, one might question if a heterogeneous container would still be necessary to store all the different MixedTensors[9]. This was answered with C++ polymorphism. By deriving the MixedTensor class from an auxiliary non-template base class, it is then possible to populate the container with pointers to objects of the base class, thus, their data types are the same. The address of a MixedTensor is cast to the address of the base class, but it can still call methods overridden by the derived class (MixedTensor).

This approach allows layers to use different precisions and avoids unnecessary conversions. The only limitation is that all model weights have to use the same posit precision in the optimizer. It is a small price to pay, given that the other stages are free to use any precision.

4.4 Operations Accuracy

In the previous sections, it was observed that as the posit precision decreased, so did the achieved model accuracy. Consequently, it is specially important to reduce the sources of numerical errors when

[9] MixedTensor is a templated class, so when two objects are declared with different data types associated, the class is different.

using low-precision posits. One particularly useful metric is the decimal accuracy, defined as [20]:

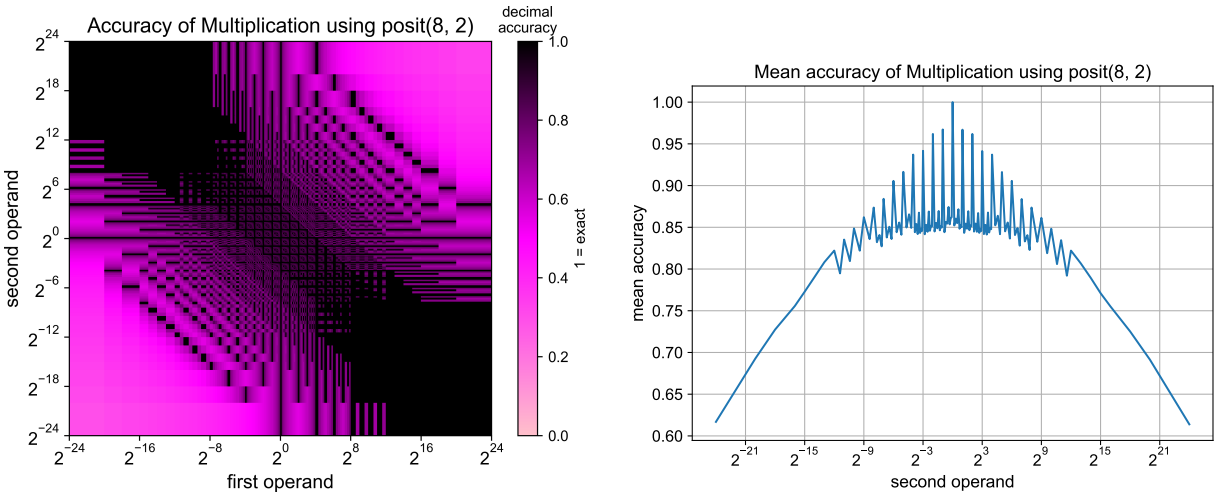
$$\text{decimal accuracy} = -\log_{10}\left(\left|\log_{10}\left(\frac{x}{y}\right)\right|\right), \tag{4.3}$$

where x and y are either the correct value or the computed value when using a certain numeric format. Accurate values (x and y are close) will correspond to large decimal accuracies, while inaccurate values (x and y are very different) will correspond to smaller decimal accuracies.

4.4.1 Powers of 2

Whenever one of the operands of an IEEE 754 multiplication or division is a power of 2, the result can be calculated without error, since it will only affect the exponent field (as long as it does not overflow or underflow). For posits, this is not always the case, as seen in Equation (4.2) with the 2×10 term. Since the exponent will also have a contribution from the variable-length regime field, some bits of the fraction might be lost, thus altering the information of the significand.

Figure 4.4 illustrates the decimal accuracy obtained with the multiplication of two posit(8, 2) values. Since the decimal accuracy can vary between $\pm\infty$, it was normalized with the sigmoid function, so that it is bounded between 0 and 1 (exact values will correspond to 1). On the left plot, the horizontal and vertical black lines will mostly correspond to operands that are powers of 2, which is more evident through the peaks on the right plot, obtained by averaging the decimal accuracy along the first operand. Thus, multiplying by powers of 2 is generally more accurate and should be preferred. One practical use of this observation in the scope of DNN training is to adopt powers of 2 for the learning rate of the optimizer.



(a) In function of the first and second operands

(b) Averaged along the first operand

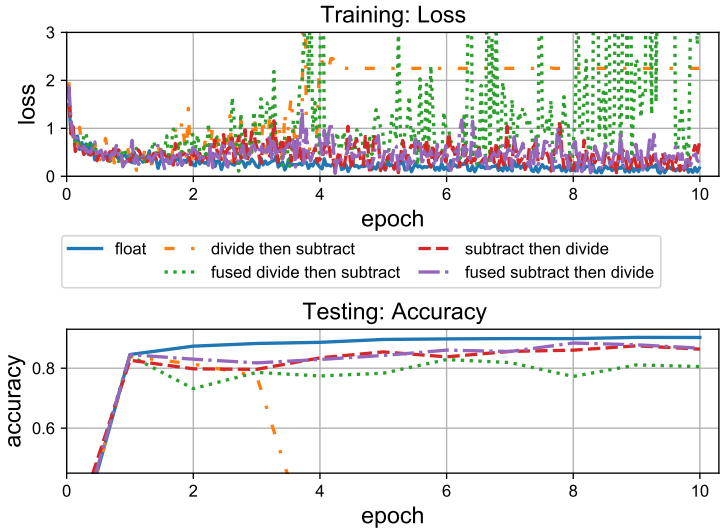
Figure 4.4: Decimal accuracy obtained with the multiplication of two posit(8, 2) values, normalized with the sigmoid function. Note there is not any result that is extremely inexact (decimal accuracy $= 0$), which would occur if the product could overflow or underflow as it does with floating-point formats.

4.4.2 Operations Order

Looking once more at the computation of the Cross Entropy loss, it was observed that the way this function is implemented greatly affects the model accuracy. Considering the backward propagation of the target class, it is equivalent to performing a division followed by a subtraction (see Equation (3.6)), but it can be equivalently calculated by first performing a subtraction and then a division:

$$\frac{A}{B} - 1 = \frac{A - B}{B}. \tag{4.4}$$

The first implementation is the most common one, since the division part will correspond to the softmax function. However, from the observation of the conducted experiments, performed by training a model with both implementations, it was concluded that the first one was actually unstable, while the second formulation not only showed to be more stable but also achieved a better accuracy. At this respect, it shall be recalled that the Posit Standard defines some fused operations, which the used posit library (Universal) also supports. The usage of these operations should, in principle, reduce the associated rounding errors. Once more, subtracting first and then dividing showed to provide better results, as presented in Figure 4.5 (training with configuration O12-FBGL8$_q$).

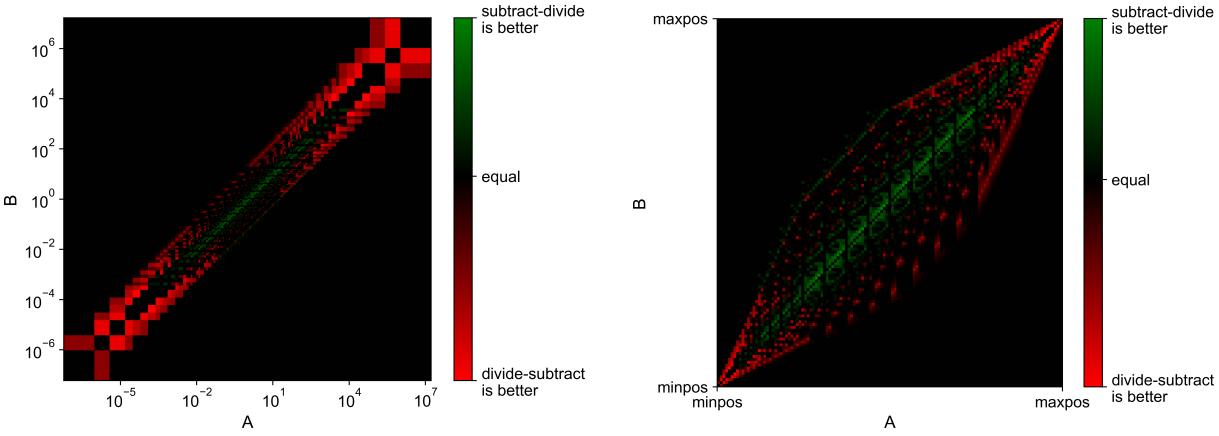


Implementation	Accuracy	
	Normal	Fused
Float (FP32)	**90.42%**	
Divide then Subtract	83.93%	84.18%
Subtract then Divide	87.47%	88.40%

Figure 4.5: Evaluation of different implementations of the Cross Entropy loss operation when training with posits. Table with the achieved model accuracies. Compared against the accuracy of a model trained with 32-bit float.

To get a better insight on why subtracting first showed better results than dividing first, the decimal accuracies obtained with both operations orders were calculated and compared in Figure 4.6. In these plots, black regions correspond to operations where the result is the same for both implementations and colored ones correspond to operations where a certain implementation is more accurate than the other.

By observing the presented plot drawn with logarithmic scale, dividing first seems to be more accurate than subtracting first, since the red regions appear to be larger. However, this does not account for the distribution of the posit format, which is denser near 1. If the plot is shown with no scale (the posits are evenly distributed/spaced), the green region in the center is much larger, which corresponds



(a) Logarithmic scale

(b) No scale (posits evenly spaced)

Figure 4.6: Comparison of the normalized decimal accuracy of divide first then subtract vs subtract first then divide (see Equation (4.4)), using posit(8, 2). If green, subtract first is better, otherwise, if red, divide first is better.

to where subtracting first is more accurate. Recalling the softmax analogy to probabilities, A/B will correspond to the probability of the target class, which should tend to 1 as the model trains. Therefore, the value of A will be similar to B (with $A < B$) and it will exactly correspond to the region where subtract first is more accurate.

4.5 Training with less than 8-bits and Underflow

Being able to train DNNs with 8-bit posits, one might wonder how much more can the precision be decreased until the model is no longer able to train. In [74], the authors observed that the absence of the underflow condition was undermining the posit convergence for low-precision posits. The underflow topic was once more addressed in [73], where specially adapted posit formats that could underflow were used for DL (for quantization), inspired by the fact that small values can be set to zero without hurting the model performance.

To evaluate if the lack of underflow is undermining the posit convergence, the considered posit library (Universal) was modified to support underflow, enabled through a user-defined option during compilation. Lu et al. [73] set the underflow threshold to *minpos*$/2$, which means that all values smaller than such threshold will underflow to 0 when quantized to posit.

However, the present work proposes a different and more natural approach, by implementing the underflow condition as it would occur if a posit could round to 0 (the same way it rounds to any other value) instead of saturating to *minpos*. According to the Posit Standard [25], the rounding rules state that *the value is rounded to the nearest binary value if the posit were encoded to infinite precision beyond the nbits length; if two posits are equally near, the one with binary encoding ending in 0 is selected*. Thus, the threshold value will correspond to the infinite precision posit representation whose binary value is equally near to 0 and *minpos*. As an example, for a posit(4, 1) configuration, the 0 and *minpos* values

are:

$$\begin{aligned} 0 : 0000 \quad &\rightarrow 0, \\ \textit{minpos} : 0001 \quad &\rightarrow 2^{2^1 \times (-2)}. \end{aligned} \tag{4.5}$$

Since the underflow condition applies the rounding rule, the threshold value will correspond to the binary encoding that is equally near to 0 and *minpos* (represented with infinite precision):

$$\begin{aligned} 0 : 0000\,000\ldots \quad &\rightarrow 0, \\ (\text{threshold} : 0000\,100\ldots) \quad &\rightarrow 2^{2^1 \times (-3)}, \\ \textit{minpos} : 0001\,000\ldots \quad &\rightarrow 2^{2^1 \times (-2)}. \end{aligned} \tag{4.6}$$

This threshold value is specific to the posit(4, 1) configuration, but it generalizes for any *nbits* and *es* configuration as

$$\text{threshold} = 2^{-2^{es} \times (nbits-1)} = \textit{minpos}/2^{2^{es}}. \tag{4.7}$$

Hence, if a value is less than or equal to[10] the threshold value, it will round to 0. This threshold condition is more flexible than a fixed one when working with arbitrary posit configurations. For example, for posits with $es = 2$, the threshold will be $\textit{minpos}/16$, which is more appropriate for the corresponding larger dynamic range than the $\textit{minpos}/2$ adopted in [73]. To implement it, the posit software library was modified such that, instead of always saturating for values smaller than *minpos*, it only saturates if the value is smaller than *minpos* but larger than the threshold, otherwise, when less than or equal to the threshold it underflows.

To test and evaluate this modification, the model was once more trained with 12-bit posits for the optimizer and loss calculations, but this time using {7, 6, 5}-bit posits everywhere else, with and without underflow, as presented in Figure 4.7. When 5-bit posits were used, the model was clearly unstable. However, when the posits could underflow to zero, the model was capable of achieving a much higher accuracy and took longer to diverge.

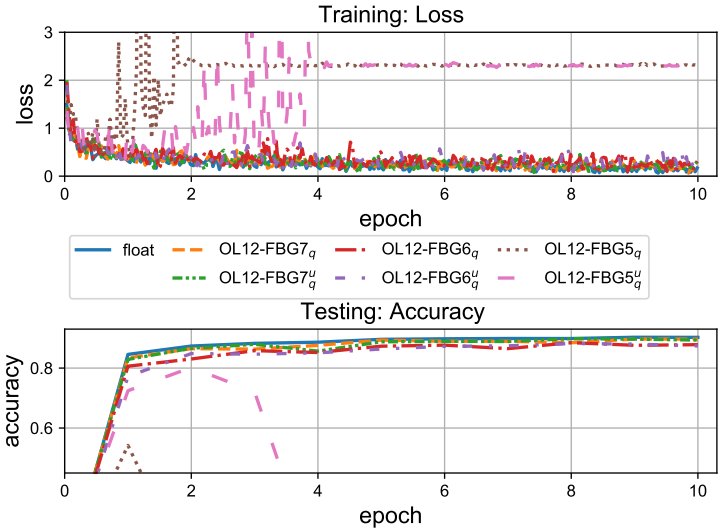
Hence, in what regards the implementation of DL networks, it is concluded that the problem of saturating to *minpos* (instead of underflowing to 0) is that, as the values decrease, they will continue affecting the involved computations (e.g. damaging the directions of the gradients).

Loss Scaling

Similar studies where NNs were trained with 16-bit floats [15, 87] observed that the gradients tend to have small magnitudes values and often underflow. One solution to this problem was to scale up the gradients, shifting the logarithmic distribution of their values in order to occupy more of the representable range of this format. This can be achieved with a technique called loss scaling, which consists in scaling the loss value before the backward propagation. By applying the chain rule, all the gradients end up scaled by the same amount.

However, considering that the backpropagation stage has already been shown to be feasible with only

[10] Since the binary encoding ending in 0 takes precedence, the threshold value underflows to 0.



Format	Accuracy	
	Normal	Underflow
Float (FP32)	$\mathbf{90.28\%}$	
OL12-FBG7$_q$	89.69%	89.81%
OL12-FBG6$_q$	88.47%	88.18%
OL12-FBG5$_q$	54.19%	80.20%

Figure 4.7: Evaluation of how underflow affects a model trained with low-precision posits. Plot of the training loss and testing accuracy and a table summarizing the achieved accuracies.

8-bit posits or even less, it does not seem necessary to use such an additional technique. Nonetheless, this technique was evaluated for lower posit precisions but no accuracy improvements were observed. Moreover, more complex scaling techniques, such as adaptive loss scaling, when applied to models with posits of reduced precision, introduced more rounding errors than actual accuracy benefits.

4.6 Summary

This chapter evaluated the model accuracy performance when applying the posit format in DNN implementations. Initially, it was analyzed how the posit configuration (*nbits* and *es*) affected the training process, without any other special technique to improve the performance. It was observed that 16-bit posits could directly replace 32-bit floats, but lower precision posits or any with es $= 0$ had trouble training or did not even converge (8-bit posits). Then, the usage of quires for accumulations was evaluated, which corresponded to a small increase in the model accuracy. Moreover, it was observed that there are some computations in DNN training that require more precision than others. Therefore, the PositNN framework was extended to support computing with mixed precision. This modification allowed to train networks using as few as 8-bit posits, relying only on 12-bit posits for the optimizer and the loss calculations. Since low-precision posits are more prone to numerical errors, a few methods to take advantage of their most accurate regions were presented. Furthermore, the posit precision was decreased even below 8 bits, which still gave acceptable results when 5-bit posits were implemented with underflow.

Chapter 5

Experimental Evaluation

Contents

The previous chapter presented multiple results concerning the adoption of posit formats to implement DNNs, namely, it showed that a mixed-precision configuration using mostly 8-bit posits can be used for DL training. However, such preliminary analysis only addressed the Fashion MNIST dataset and the LeNet-5 model. Therefore, in order to validate some of the preliminary conclusions that were already presented, the same configuration will be evaluated when training more complex datasets and models. Besides training, posits will also be evaluated for DNN inference when using the same datasets and models. This step is expected to be more resilient to numerical errors and to allow even lower precisions. The obtained results will be used to discuss the most recent version of the Posit Standard. At last, the proposed framework PositNN is evaluated in terms of gained speedup with parallelization.

5.1 Experimental Setup

To validate the use of posits for DNN training and inference, a few more datasets and models were implemented in PositNN and PyTorch (to compare with floats). Once again, and similarly to what is observed in related works, the experiments focused on image classification. In addition to Fashion MNIST, the simpler MNIST dataset was evaluated, also with the CNN LeNet-5. Nonetheless, more complex datasets were evaluated, such as CIFAR-10 and CIFAR-100 (detailed in Section 2.1.6). Besides the LeNet-5 model, the presented evaluation also considered a variation of the CifarNet model. In particular, this model is similar to the one used in [81], being composed of ~ 0.5 million parameters and it is illustrated in Figure 5.1. Table 5.1 summarizes the covered datasets and models. The training hyperparameters are shown in Table 5.2, which were the same for all models and datasets, since they showed the best results.

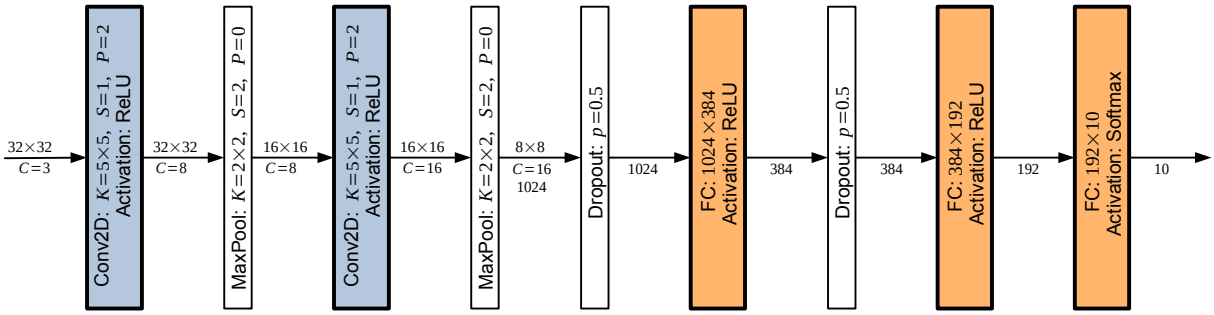


Figure 5.1: Architecture of the evaluated variation of CifarNet based from [81] with $\sim 5 \times 10^5$ parameters.

Table 5.1: Considered datasets, models, and number of epochs used for training.

Dataset	Model	Epochs
MNIST	LeNet-5	10
Fashion MNIST	LeNet-5	10
CIFAR-10	CifarNet	20
CIFAR-100	CifarNet	20

Regarding the considered datasets, they were preprocessed by applying a normalization step that moved their mean value to 0 and the standard deviation to 1. The training datasets were conveniently

Table 5.2: Configurations used for the training of the various CNNs. LR is for Learning Rate.

Loss	Optimizer	Initial LR	LR Scheduler	Momentum	Batch Size
Cross Entropy	SGD	$1/16$	Divide by 2 after every 4 epochs	0.5	64

shuffled, since such randomness helps with the model convergence and achieved generality. All the experiments were repeated 2-3 times and then averaged, to account for the randomness caused by the network initialization, shuffled training data, and dropout layers.

To evaluate the models, the top-k performance metric was used, which measures the percentage of tests where the target class was in the k most probable classes of the predicted output. When $k = 1$, top-1 is exactly equivalent to the model accuracy, but greater k values are particularly useful to evaluate more complex datasets or those with many classes.

5.2 DNN Training Evaluation

The results obtained for DNN training using posits are shown in Table 5.3. They were obtained using the developed PositNN framework and with mixed-precision training, where the majority of the computations were performed with 8-bit posits and only the optimizer and loss functions used posits with higher precision. One of the chosen mixed configurations is illustrated in Figure 5.2 (similar to Figure 3.1 previously shown), which clearly specifies the various posit formats used throughout the training procedure and the employed optimizer and loss functions. Appendix A.2 exemplifies how the experiments were implemented with PositNN. For comparison, the same models were trained with 32-bit floats using PyTorch.

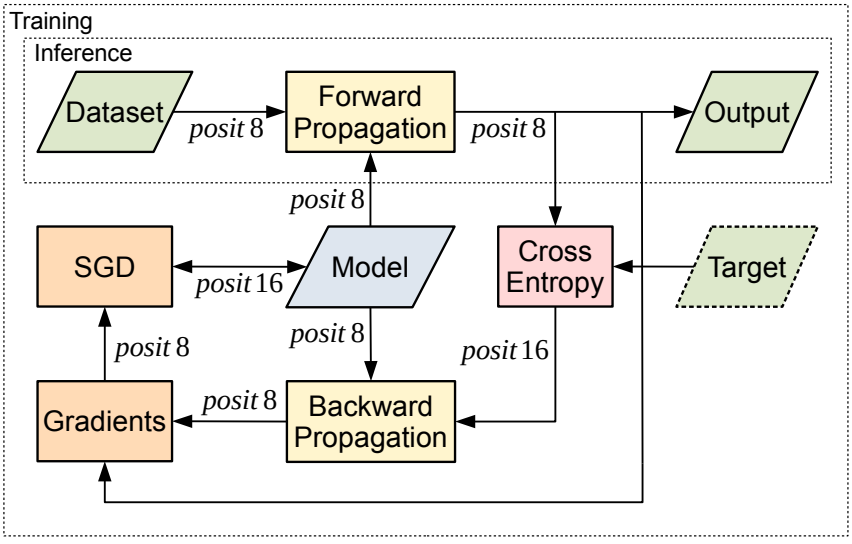


Figure 5.2: Block diagram of the mixed low-precision posit configuration used to train and test various CNN models. It uses the SGD optimizer and the Cross Entropy loss, both calculated with 16-bit posits. Everything else is computed with 8-bit posits. The colors denote blocks with similar or related calculations.

According to the obtained results, the mixed-precision configuration that was proposed in the previous chapter – consisting in using 8-bit posits for everything except the optimizer (O) and loss (L),

Table 5.3: Accuracy evaluation of using posits for DNN training with mixed precision and various datasets and models. The obtained results were compared against the same models trained with 32-bit floats with PyTorch.

Format	MNIST (LeNet-5)	Fashion MNIST (LeNet-5)	CIFAR-10 (CifarNet)		CIFAR-100 (CifarNet)	
	Accuracy	Accuracy	Top-1	Top-3	Top-1	Top-5
Float (FP32)	$\mathbf{99.21\%}$	$\mathbf{90.28\%}$	$\mathbf{70.79\%}$	$\mathbf{92.64\%}$	$\mathbf{36.35\%}$	$\mathbf{66.92\%}$
Posit8 and O16-L16$_q$	99.19%	90.46%	71.30%	92.65%	35.41%	67.00%
Posit8 and O16-L12$_q$	99.17%	90.14%	71.09%	92.83%	35.27%	66.57%
Posit8 and O12-L12$_q$	99.20%	90.07%	68.28%	91.22%	25.85%	57.77%

that use 12-bit posits – was enough to achieve an accuracy equivalent to 32-bit floats when evaluated with the Fashion MNIST dataset. As expected for MNIST, which is a simpler dataset, the same was observed. However, with this mixed-precision configuration, CIFAR-10 suffered a small accuracy loss and CIFAR-100 an even larger loss when compared to floats. To overcome this problem, the precision of the optimizer and the loss functions were increased to 16-bit posits and the models were then able to achieve the 32-bit float performance. This precision increase allowed the model weights to use a larger dynamic range and to be updated more accurately, which seems to be necessary for more complex datasets. For CIFAR-100, the top-1 accuracy appears to be slightly worse, but the top-5 accuracy reassures that the performance achieved with posits is equivalent to 32-bit floats.

It is important to recall that being able to replace 32-bit floats with 8-bit posits immediately corresponds to a $4\times$ smaller memory footprint. Moreover, if the power required by a posit unit is comparable to its IEEE 754 compliant counterpart (as suggested by [22]), then it will also use much less energy. Even for the computations that require more precision, 12 and 16-bit posits seem to be enough, never requiring a 32-bit format.

The obtained results can be also compared with other studies that used low-precision floats, such as in Langroudi et al. [21], where a FCNN was trained with MNIST and Fashion MNIST using 32 and 16-bit floats. While 32-bit floats achieved an accuracy of 98.09% and 89.11% for these two datasets, 16-bit floats only achieved 90.65% and 81.73%, respectively. Thus, posits can achieve a much better performance than 16-bit floats while using half the memory.

5.3 DNN Inference Evaluation

In addition to DNN training, it is also relevant to evaluate the performance of posits for DNN inference. To do so, all the considered models were trained in advance with floats (but could also be trained with posits, as long as a good model accuracy was achieved) and then the model weights were quantized to a low-precision posit to test the forward propagation. For each dataset, {3-8}-bit posits were evaluated with different values of the exponent size (*es*) and using quires for the accumulations – see Tables 5.4 to 5.7. For {3, 4}-bit posits, some *es* values are not represented because the posit library does not support those configurations[11].

[11] For example, a 3-bit posit uses 1 bit for the sign and 2 bits for the regime, therefore, the exponent size is 0.

Table 5.4: Accuracy of LeNet-5 inference on MNIST using various posit configurations.

Format	MNIST: Accuracy		
	$es = 0$	$es = 1$	$es = 2$
Float (FP32)	**99.21%**		
Posit8$_q$	99.18%	99.21%	99.20%
Posit7$_q$	98.70%	99.13%	99.17%
Posit6$_q$	94.94%	98.77%	98.83%
Posit5$_q$	63.97%	95.97%	95.68%
Posit4$_q$	46.79%	70.90%	-
Posit3$_q$	19.53%	-	-

Table 5.5: Accuracy of LeNet-5 inference on Fashion MNIST using various posit configurations.

Format	Fashion MNIST: Accuracy		
	$es = 0$	$es = 1$	$es = 2$
Float (FP32)	**90.28%**		
Posit8$_q$	89.93%	90.23%	90.05%
Posit7$_q$	88.51%	89.80%	89.58%
Posit6$_q$	81.16%	87.71%	87.36%
Posit5$_q$	42.03%	83.14%	81.77%
Posit4$_q$	23.28%	52.53%	-
Posit3$_q$	10.83%	-	-

Table 5.6: Top-1 and Top-3 accuracies of CifarNet inference on CIFAR-10 using various posit configurations.

Format	CIFAR-10: Top-1 / Top-3		
	$es = 0$	$es = 1$	$es = 2$
Float (FP32)	**70.79% / 92.64%**		
Posit8$_q$	70.91% / 92.89%	71.01% / 92.84%	70.23% / 92.58%
Posit7$_q$	69.41% / 92.54%	70.04% / 92.46%	69.18% / 92.19%
Posit6$_q$	59.09% / 91.08%	68.66% / 92.21%	64.74% / 91.14%
Posit5$_q$	17.03% / 72.60%	61.35% / 89.67%	54.50% / 86.82%
Posit4$_q$	11.16% / 41.11%	40.13% / 79.97%	-
Posit3$_q$	9.91% / 30.41%	-	-

Table 5.7: Top-1 and Top-5 accuracies of CifarNet inference on CIFAR-100 using various posit configurations.

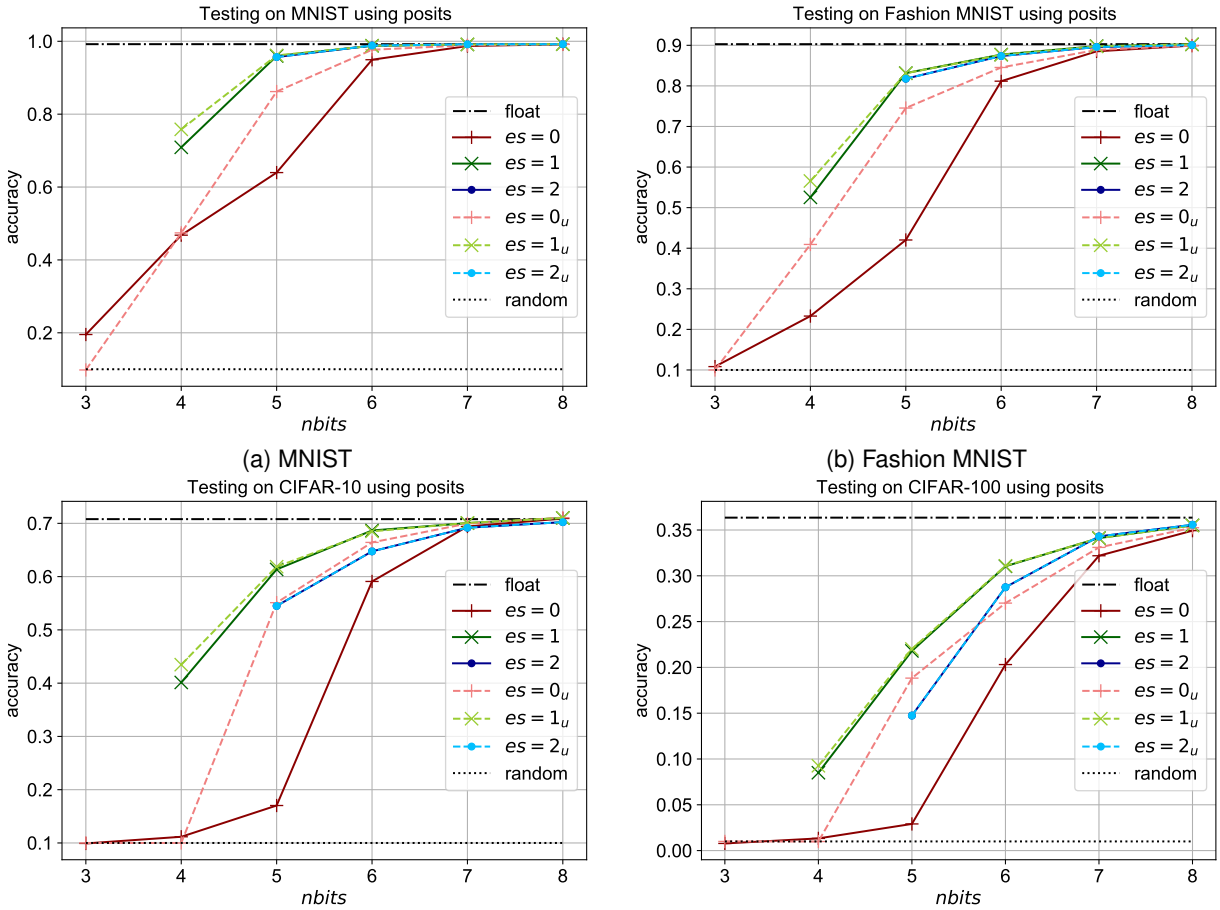
Format	CIFAR-100: Top-1 / Top-5		
	$es = 0$	$es = 1$	$es = 2$
Float (FP32)	**36.35% / 66.92%**		
Posit8$_q$	34.93% / 66.33%	35.51% / 66.70%	35.57% / 66.67%
Posit7$_q$	32.19% / 64.86%	34.12% / 66.25%	34.30% / 65.68%
Posit6$_q$	20.31% / 55.41%	31.04% / 64.62%	28.75% / 61.08%
Posit5$_q$	2.91% / 13.96%	21.80% / 56.90%	14.74% / 44.07%
Posit4$_q$	1.33% / 7.20%	8.50% / 28.25%	-
Posit3$_q$	0.78% / 4.55%	-	-

The obtained results are similar to those presented in related works [21, 81] considering only the inference phase. However, the conducted experiments evaluate even lower precision posits and use quires for intermediate accumulation. In more detail, for every experiment performed with 8-bit posits, the implemented networks showed to be capable of performing as well as 32-bit floats, with model accuracy differences lower than 1% (except when $es = 0$). Posit precisions with less than 8 bits were also able to obtain good accuracies, although they start struggling with more complex datasets. In particular, the 6-bit posit was able to achieve acceptable accuracy for CIFAR-100, but the 5-bit posit presents an accuracy loss higher than 10%. The {3, 4}-bit posits, which are in the borderline of the possible posit formats, have much worse accuracies. However, posit(4, 1) can still be used to correctly

classify some samples.

Nonetheless, being able to get a meaningful accuracy with 5-bit posits (instead of 32-bit floats) is still very impressive, since it is replacing the same exact operations that it would otherwise perform with 32-bit floats, without any reformulation of the calculations. Moreover, in [21], a 5-bit floating-point format was evaluated for CIFAR-10 and it did not work at all, since it got an accuracy of only around 13%[12].

To evaluate how the underflow of the posit format affected the model accuracy, the same experiments were executed with support for underflow. In Figure 5.3, the accuracies obtained are compared against those presented in Tables 5.4 to 5.7. These results suggest that enabling underflow is particularly beneficial for very low-precision posits with *es* $= 0$. This was already expected because their not so insignificant *minpos* values were harming their computations [74]. For example, when posit(5, 0) was tested with underflow for CIFAR-10 and CIFAR-100, it achieved better accuracy than posit(5, 2). Nonetheless, this accuracy gain is not enough to make posits with *es* $= 0$ generally as good as posits with *es* $= 1$ or 2.



(a) MNIST (b) Fashion MNIST

(c) CIFAR-10 (d) CIFAR-100

Figure 5.3: Obtained accuracies when testing pre-trained models with various datasets and posit configurations (using quires) and when considering the effect of underflow (subscript u). The results were compared against the accuracy obtained with 32-bit floats and when the model is randomly initialized (untrained).

[12] A random model should obtain an accuracy of $\sim 10\%$ for a dataset with 10 classes.

5.4 Comparison of Posit Standards

As explained in Section 2.2.3, the most recent version of the Posit Standard [24] (still unpublished) introduced two changes: posits no longer have an arbitrary exponent size (fixed to $es = 2$), and the quire format was reformulated so that the limit of exact dot products is the same for any posit precision ($2^{31} - 1$ accumulations).

To account for this change of the exponent size, the performed experiments already focused on posits with $es = 2$. Therefore, the obtained results apply to both versions of the standard. However, the conducted inference experiments, particularly those with more complex datasets and models, obtained significantly better results for posits with $es = 1$ than for posits with $es = 2$. Therefore, a more thorough analysis should be performed to completely evaluate this modification of the Posit Standard.

As for the new quire format, the main practical consequence is that it can accumulate many more products for posit precisions smaller than 32 bits. Implementing it was also straightforward with the proposed PositNN framework, since the quire capacity is defined through the template of the quire class. To switch between the two quire formats, PositNN provides a compilation option. Once again, this change does not impair the results above, rather on the contrary, it should improve the accuracy obtained in DL. In the particular case of the performed tests, the new quire format showed no relevant improvements, but it should be more noticeable for even wider models using low-precision posits, since they will have larger accumulations.

5.5 Parallelization Speedup

To evaluate the performance of PositNN, in what concerns the scalability when executed in platforms with parallel processing capabilities, a small throughput analysis was also performed. Ideally, the offered speedup would be directly proportional to the number of used threads. However, it is expected some overhead caused by the spawning of the different workers and synchronization. To conduct this evaluation, a simple CNN was trained on 8192 samples multiple times and the execution times were measured for a different number of used threads. These tests were executed in a system with an Intel i7-5930K CPU (6 cores with 2 threads per core) operating at $3.5\,\text{GHz}$ and with $32\,\text{GB}$ of RAM, whose results are shown in Figure 5.4. As expected, the speedup increases quite proportionally with the number of threads. Above 6 threads, the speedup worsens, since the CPU only had 6 cores. Hence, although the total number of threads was 12, each core was not able to completely parallelize 2 threads for the given task.

Considering the overhead associated with the spawning of the threads/workers and that the parallelization was only implemented for the most computational demanding layers, the results are quite satisfactory. For such reason, it is expected that the speedup should become even closer to the ideal for deeper and more complex models.

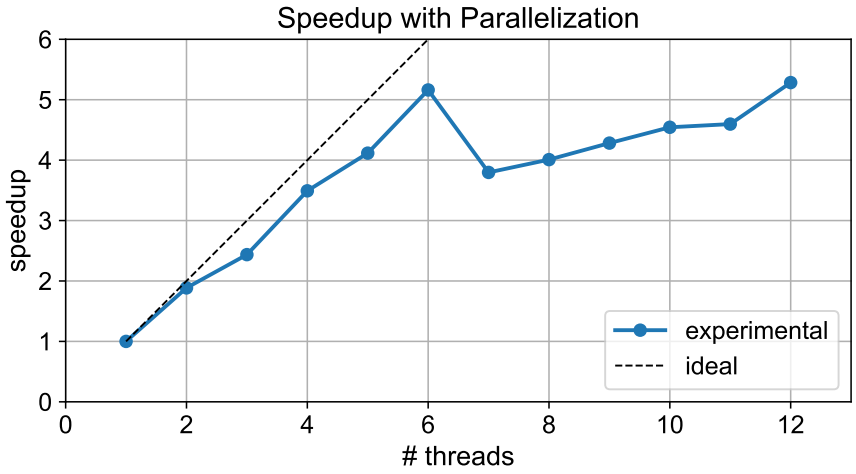


Figure 5.4: Obtained speedup of the program execution in function of the number of used threads. Tested in a 6-core CPU (2 threads per core) when training a simple CNN on 8192 samples.

5.6 Summary

In this chapter, several mixed low-precision posit configurations were evaluated for various datasets and models. These configurations consist in using 8-bit posits for most of the computations and {12, 16}-bit posits for only the loss and the optimizer functions. The obtained results support that these posit configurations can substitute 32-bit floats for DNN training and achieve equivalent model accuracies.

As for experiments regarding only DNN inference, that is, performing the forward propagation with a previously trained model, it was possible to use even lower posit precisions, instead of 32-bit floats, incurring little to no harm in the model accuracies.

From the obtained results about the accuracy of posits for DL training and inference, a brief discussion was presented regarding the most recent Posit Standard. Considering that posits with $es = 1$ performed generally better than posits with $es = 2$, the modification of the exponent size proposed in the newer standard should be further evaluated.

The chapter is finalized with an evaluation of the performance scalability of the proposed PositNN framework, more precisely, the speed up gained with its parallelization. For the executed tests, the speedup was quite proportional to the number of threads (up to the number of cores of the CPU). Nonetheless, one shall recall that the main objective of the PositNN framework is to allow studying the performance of the posit format applied to DNN in terms of accuracy and not of computational speed (since posits are simulated by software).

Chapter 6

Conclusions and Future Work

Contents

In the last years, it has been observed a growing interest about using Deep Learning (DL) techniques by several application research domains. However, such interest is often accompanied by the need to use significant computational demands to implement not only the required training procedures, but also the inference phase. Consequently, when the posit format was introduced as a direct drop-in replacement for the IEEE 754 floating-point format, offering greater accuracy and better energy performance for fewer bits, its application to DL gained an immediate and considerable interest. Most works on this topic focused on the evaluation of posits for DNN inference and were able to achieve a performance comparable to 32-bit floats using as few as {5..8}-bit posits. However, most of these studies adopted models that were pre-trained with 32-bit floats and were then quantized to the posit format to be used at the inference phase. A more compelling topic is to also benefit from the advantages of the posit format for DNN training, which was still very incipient at the beginning of this thesis. More recently, a few studies that addressed this same topic have appeared. However, they are limited in terms of posit precisions that were evaluated, since they were only able to achieve good accuracy for DNN training with 16-bit posits. The present work aimed to fill that gap, by developing an entire DL framework that allowed to thoroughly study and exploit DNN training and inference using posits of any precision on even more challenging models and datasets.

6.1 Main Contributions

The proposed open-source software framework, named PositNN [26], was the main contribution of this work, because it not only allowed to obtain the presented results regarding the evaluated performance of posits for DL, but also facilitates further work on this topic. PositNN was developed with an API that is very similar to PyTorch C++ API, to facilitate its use by any user who is familiar with PyTorch. It is capable of performing end-to-end DNN training and inference with any posit precision and allows the use of the corresponding quires, instead of being limited to the recommended posit configurations (see Table 2.3). Furthermore, it supports the most common layers and operators usually used by CNNs, which allows implementing and evaluating the most typical models with posits (see Table A.1). Moreover, it offers a set of base classes that can be used to easily extend the framework with custom modules. The development of PositNN in C++ with a multithreaded parallelization guarantees a fast performance, which is particularly useful given that posits are simulated via software.

Regarding DNN training with posits, the first presented experiments consisted in training a CNN using various posit configurations, without using any additional techniques to improve the accuracy. As expected, 16-bit posits achieve the same level of accuracy as 32-bit floats. However, as the precision decreases bellow 16-bits, the model accuracy is penalized and the 8-bit posit implementation is difficult to converge. It is also noted that low-precision posits with $es = 0$ perform worse than with $es = 1$ or 2. Nevertheless, associated to the posit format, there is a large accumulator (named quire), which improves the accuracy obtained with low-precision posits, but it is not enough to solve the lack of convergence problems.

Hence, using low-precision posits for every computation of the DL training procedure proved to be

a naive approach, because some calculations require more precision than others. Fortunately, the PositNN framework was implemented in such a way that each layer or stage can use any arbitrary posit precision, which was subsequently exploited to implement mixed-precision configurations. The initial experiments showed that it is indeed possible to train DNNs and to obtain a model accuracy equivalent to 32-bit floats by using mostly 8-bit posit formats combined with higher precision operations used only in the optimizer and loss functions. When tested with more complex datasets and using 16-bit posits for the optimizer and the loss functions, the models achieved accuracies on the same level as 32-bit floats (differences $< 1\%$). For the considered models, the subset of operations that were performed with higher posit precisions only represent about $5 - 15\%$ of the total computations and do not even need a 32-bit format. It is important to highlight that this was the first work where 8-bit posits were able to replace 32-bit floats for DNN training and achieve an equivalent accuracy for typical datasets and models.

Regarding inference, the models were tested with {3..8}-bit posits and the obtained results reinforce how well low-precision posits perform for DL. For all datasets, the tests with 8-bit posits had accuracies similar to 32-bit floats and those with 6-bit posits present a difference not higher than $\sim 5\,\%$. The presented research also considered the possibility to implement an underflow condition, which significantly benefited the achieved accuracy, especially for very low precision posits with $es = 0$. In particular, posit(5, 0) with underflow even achieved a better accuracy than posit(5, 2) for CIFAR-10 and CIFAR-100.

Furthermore, and as far as the author knows, this work was the first to address a more recent version of the Posit Standard. The main modifications involve the exponent size of the posit format, which was fixed to 2, and the capacity of the quire that, consequently, was increased for posit precisions lower than 32 bits. Although the quire increment will certainly benefit DL training with low-precision posits, fixing the exponent size to 2 might not be the best choice for DL, since posit(8, 1) often performed better than posit(8, 2) in the executed experiments. Nonetheless, it is important to keep in mind that the posit format is useful not only for DL but also for other applications that might benefit from that wider dynamic range.

6.2 Future Work

The conclusions yielded by the present work show how promising the posit format is for DL. The ability to perform most of the computations involved in DNN training with only 8-bit posits (or maybe even less) will result in significant improvements in terms of the memory footprint, energy efficiency, and speed, since they are currently performed with 32-bit floats, allowing to train even deeper and more complex models. Moreover, this energy-efficient approach could be particularly useful for DL in portable devices (such as smartphones) or environments with higher restrictions in terms of computations, such as in satellites, which would also greatly benefit from the reduced memory footprint and efficient energy consumption.

Despite its relevancy and usefulness, by evaluating and demonstrating the performance of posits for DL applications, it shall be noted that the posit arithmetic is simulated via software. This is the main limitation of the proposed framework, because, although it allows drawing conclusions about how posits perform in terms of accuracy, the speed and power consumption could only be properly compared to IEEE 754 floating-point formats with a hardware implementation. The posit tensor unit proposed in

[80] by the same research group where this thesis was developed would be an excellent starting point, since it was implemented for an ASIC and showed a great performance in various DL training related operations. However, the hardware implementation to be evaluated should support mixed-precision configurations, given the promising results obtained in this work.

Nonetheless, there are still many research topics for which PositNN would be useful. In particular, it could be used to test different posit precisions per layer of the model, since certain layers might not require as much precision. Moreover, for DNN training, it would also be compelling to study different precisions for the forward propagation, backward propagation, and gradients calculation, since the range of the involved values can be very different. To further evaluate DL with posits, deeper models and more complex datasets should be tested, preferably by also covering other topics besides image classification. Furthermore, ensemble methods for both training and inference might improve the obtained accuracies and should be particularly interesting when combining models that use very small posit precisions.

In another perspective, when a model that was trained with 32-bit floats is converted to be used with a low-precision posit, the conducted quantization might not be ideal, because the original weights had available a larger set of values that they could assume during training. On the other hand, if the model were to be trained with posits and the precision of the forward propagation dynamically reduced during the training process (the other stages could still use 16-bit posits to guarantee an accurate process), the final model weights would probably adapt much better to very small posit precisions. PositNN may be used to perform this study.

At last, the PositNN framework could be easily adapted to study other data types, since the template-based implementation should allow for a more or less simple adaption. For example, it could be adapted to support low-precision floating-point formats in order to evaluate them against the presented results obtained for posits, under the same circumstances.

Finally, and regarding the Posit Standard, the results shown in this work should help in the formulation of this novel format. Posit with $es = 2$ have shown good results, but the same precision with $es = 1$ showed similar and even better results for DL. Furthermore, the posit format is characterized by the fact that it saturates instead of overflowing or underflowing. Nevertheless, for low-precision posits, the inability to underflow might undermine their application in DL, as shown by the presented training and inference procedures using very low precision posits. Ultimately, a domain-specific posit unit could relax the posit format and allow them to underflow. However, this modification to the posit format should be more thoroughly evaluated.

Bibliography

[1] A. Oliveira. *The Digital Mind: How Science is Redefining Humanity*. MIT Press, Cambridge, Massachusetts, 2017. ISBN 9780262036030. doi: 10.7551/mitpress/9780262036030.001.0001.

[2] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. doi: 10.1037/h0042519. URL https://doi.org/10.1037/h0042519.

[3] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, Jan. 2015. doi: 10.1016/j.neunet.2014.09.003. URL https://arxiv.org/abs/1404.7828.

[4] L. Fridman. Deep Learning State of the Art (2020) | MIT Deep Learning Series. MIT Deep Learning and Artificial Intelligence Lectures, Jan. 2020. URL https://deeplearning.mit.edu/. Accessed on 2020-09-24.

[5] N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso. The Computational Limits of Deep Learning. *arXiv: 2007.05558*, July 2020. URL https://arxiv.org/abs/2007.05558.

[6] M. Minsky and S. A. Papert. *Perceptrons; an introduction to computational geometry*. MIT Press, Cambridge, Mass, 1969. ISBN 9780262630221.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[8] C. Szegedy, A. Toshev, and D. Erhan. Deep Neural Networks for Object Detection. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 2553–2561. Curran Associates, Inc., 2013. URL http://papers.nips.cc/paper/5207-deep-neural-networks-for-object-detection.pdf.

[9] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to Sequence Learning with Neural Networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3104–3112. Curran Associates, Inc., 2014. URL https://arxiv.org/abs/1409.3215.

[10] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language Models are Few-Shot Learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, 2020. URL https://arxiv.org/abs/2005.14165.

[11] R. Brooks. Predictions Scorecard, 2019 January 01. Rodney Brooks - Robots, AI, and other stuff, Jan. 2019. URL https://rodneybrooks.com/predictions-scorecard-2019-january-01/. Accessed on 2020-09-24.

[12] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber. Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition. *Neural Computation*, 22(12):3207–3220, dec 2010. doi: 10.1162/neco_a_00052. URL https://arxiv.org/abs/1003.0358.

[13] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. doi: 10.1109/ieeestd.2019.8766229.

[14] T. Dettmers. 8-Bit Approximations for Parallelism in Deep Learning. *arXiv: 1511.04561*, 2015. URL https://arxiv.org/abs/1511.04561.

[15] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. Mixed Precision Training. *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, pages 1–12, Oct. 2018. URL https://arxiv.org/abs/1710.03740.

[16] J. Johnson. Rethinking floating point for deep learning. In *NeurIPS Systems for ML Workshop, 2019*, Nov. 2018. URL https://arxiv.org/abs/1811.01721.

[17] D. Lin, S. Talathi, and S. Annapureddy. Fixed point quantization of deep convolutional networks. In M. F. Balcan and K. Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2849–2858, New York, New York, USA, 20–22 Jun 2016. PMLR. URL http://proceedings.mlr.press/v48/linb16.html.

[18] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[19] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In B. Leibe, J. Matas, N. Sebe, and M. Welling, editors, *Computer Vision – ECCV 2016*, pages 525–542, Cham, 2016. Springer International Publishing. ISBN 978-3-319-46493-0.

[20] J. L. Gustafson and I. Yonemoto. Beating Floating Point at its Own Game: Posit Arithmetic. *Supercomputing Frontiers and Innovations*, 4(2):71–86, June 2017. ISSN 2313-8734. doi: 10.14529/jsfi170206.

[21] H. F. Langroudi, Z. Carmichael, D. Pastuch, and D. Kudithipudi. Cheetah: Mixed Low-Precision Hardware & Software Co-Design Framework for DNNs on the Edge. *arXiv: 1908.02386*, pages 1–13, Aug. 2019. URL https://arxiv.org/abs/1908.02386.

[22] R. Chaurasiya, J. Gustafson, R. Shrestha, J. Neudorfer, S. Nambiar, K. Niyogi, F. Merchant, and R. Leupers. Parameterized Posit Arithmetic Hardware Generator. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 334–341. IEEE, Oct. 2018. ISBN 9781538684771. doi: 10.1109/iccd.2018.00057. URL https://ieeexplore.ieee.org/document/8615707/.

[23] M. Cococcioni, E. Ruffaldi, and S. Saponara. Exploiting Posit Arithmetic for Deep Neural Networks in Autonomous Driving Applications. In *2018 International Conference of Electrical and Electronic Technologies for Automotive*. IEEE, 2018. ISBN 9788887237382. doi: 10.23919/eeta.2018.8493233. URL https://ieeexplore.ieee.org/document/8493233/.

[24] Posit Working Group. Posit™ Standard Documentation, Release 4.3-draft, Aug. 2020. E-mailed by Professor John Gustafson on 2020-09-08.

[25] Posit Working Group. Posit Standard Documentation, Release 3.2-draft, 2018. URL https://posithub.org/docs/posit_standard.pdf. Accessed on 2020-09-24.

[26] G. Raposo, P. Tomás, and N. Roma. PositNN: Training Deep Neural Networks with Mixed Low-Precision Posit. In *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Oct. 2020. (submited and under review).

[27] F. Chollet. *Deep Learning with Python*. Manning, 2017. ISBN 9781617294433. URL https://isbnsearch.org/isbn/9781617294433.

[28] H. Kumar. Loss vs Accuracy, Dec. 2018. URL https://kharshit.github.io/blog/2018/12/07/loss-vs-accuracy. Accessed on 2020-11-27.

[29] L. Bottou and Y. LeCun. Large scale online learning. In *Advances in neural information processing systems*, pages 217–224, 2004. URL http://yann.lecun.com/exdb/publis/pdf/bottou-lecun-04b.pdf.

[30] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv: 1609.04747*, 2016. URL https://arxiv.org/abs/1609.04747.

[31] J. S. Marques. Machine Learning Slides, Lecture Notes, IST, 2017. URL https://fenix.tecnico.ulisboa.pt/disciplinas/AAut137/2019-2020/1-semestre. Accessed on 2020-09-24.

[32] M. Nielsen. How the backpropagation algorithm works - neural networks and deep learning, Dec. 2019. URL http://neuralnetworksanddeeplearning.com/chap2.html. Accessed on 2020-09-24.

[33] V. Dumoulin and F. Visin. A guide to convolution arithmetic for deep learning. *arXiv: 1603.07285*, 2016. URL https://arxiv.org/abs/1603.07285.

[34] A. Agrawal. Back propagation in dilated convolution layer, Jan. 2018. URL https://www.adityaagrawal.net/blog/deep_learning/bprop_dilated_conv. Accessed on 2020-09-24.

[35] M. Kaushik. Part 1: Backpropagation for convolution with strides, May 2019. URL https://medium.com/@mayank.utexas/backpropagation-for-convolution-with-strides-8137e4fc2710. Accessed on 2020-09-24.

[36] M. Kaushik. Part 2: Backpropagation for convolution with strides, May 2019. URL https://medium.com/@mayank.utexas/backpropagation-for-convolution-with-strides-fb2f2efc4faa. Accessed on 2020-09-24.

[37] PyTorch. pytorch/pytorch: Tensors and Dynamic neural networks in Python with strong GPU acceleration - GitHub, Apr. 2020. URL https://github.com/pytorch/pytorch. Accessed on 2020-09-24.

[38] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv: 1207.0580*, 2012. URL https://arxiv.org/abs/1207.0580.

[39] PyTorch. PyTorch Documentation, 2019. URL https://pytorch.org/docs/master/index.html. Accessed on 2020-09-27.

[40] X. Glorot, A. Bordes, and Y. Bengio. Deep Sparse Rectifier Neural Networks. In G. Gordon, D. Dunson, and M. Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. JMLR Workshop and Conference Proceedings. URL http://proceedings.mlr.press/v15/glorot11a.html.

[41] Q. Wang, Y. Ma, K. Zhao, and Y. Tian. A Comprehensive Survey of Loss Functions in Machine Learning. *Annals of Data Science*, apr 2020. doi: 10.1007/s40745-020-00253-5.

[42] L. Bottou. Online Algorithms and Stochastic Approximations. In D. Saad, editor, *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK, 1998. URL http://leon.bottou.org/papers/bottou-98x.

[43] Y. Nesterov. A method for solving the convex programming problem with convergence rate O(1/k$\hat{2}$). *Dokl. Akad. Nauk SSSR*, 269:543–547, 1983. URL https://ci.nii.ac.jp/naid/10029946121/en/.

[44] G. Hinton, N. Srivastava, and K. Swersky. Neural Networks for Machine Learning: Lecture 6e, Feb. 2014. URL http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. Accessed on 2020-10-06.

[45] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pages 1–15, Dec. 2015. URL https://arxiv.org/abs/1412.6980.

[46] N. S. Keskar and R. Socher. Improving Generalization Performance by Switching from Adam to SGD. *arXiv: 1712.07628*, 2017. URL https://arxiv.org/abs/1712.07628.

[47] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi. A Survey of the Recent Architectures of Deep Convolutional Neural Networks. *Artificial Intelligence Review*, 53(8):5455–5516, apr 2020. doi: 10.1007/s10462-020-09825-6. URL https://arxiv.org/abs/1901.06032.

[48] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. ISSN 0018-9219. doi: 10.1109/5.726791. URL http://ieeexplore.ieee.org/document/726791/.

[49] J. Hosang, M. Omran, R. Benenson, and B. Schiele. Taking a Deeper Look at Pedestrians. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015. URL https://arxiv.org/abs/1501.05790.

[50] A. Krizhevsky. CIFAR-10 and CIFAR-100 datasets, 2009. URL https://www.cs.toronto.edu/~kriz/cifar.html. Accessed on 2020-10-29.

[51] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations (ICLR) 2015*, 2015. URL https://arxiv.org/abs/1409.1556.

[52] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90. URL https://arxiv.org/abs/1512.03385.

[53] M. Lin, Q. Chen, and S. Yan. Network In Network. In Y. Bengio and Y. LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014. URL http://arxiv.org/abs/1312.4400.

[54] S. Zagoruyko and N. Komodakis. Wide Residual Networks. In E. R. H. Richard C. Wilson and W. A. P. Smith, editors, *Proceedings of the British Machine Vision Conference (BMVC)*, pages 87.1–87.12. BMVA Press, September 2016. ISBN 1-901725-59-6. doi: 10.5244/C.30.87. URL https://dx.doi.org/10.5244/C.30.87.

[55] Y. LeCun, C. Cortes, and C. J. C. Burges. The MNIST database of handwritten digits, 2002. URL http://yann.lecun.com/exdb/mnist/. Accessed on 2020-10-28.

[56] Zalando Research. zalandoresearch/fashion-mnist: A mnist-like fashion product database. benchmark - github, 2017. URL https://github.com/zalandoresearch/fashion-mnist. Accessed on 2020-10-28.

[57] J. Steppan. File:MnistExamples.png - Wikimedia Commons, 2017. URL https://commons.wikimedia.org/wiki/File:MnistExamples.png. Accessed on 2020-10-28.

[58] M. Thill. Zalando's Fashion-MNIST Dataset - ML & Stats, Oct. 2017. URL https://markusthill.github.io/ml/programming/zalandos-fashion-mnist-dataset/. Accessed on 2020-10-28.

[59] B. Efron and T. Hastie. *Computer Age Statistical Inference*. Cambridge University Pr., 2016. ISBN 1107149894. URL http://www.cambridge.org/9781107149892.

[60] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, Dec. 2015. ISSN 0920-5691. doi: 10.1007/s11263-015-0816-y. URL https://doi.org/10.1007/s11263-015-0816-y.

[61] NVIDIA. Deep Learning Frameworks | NVIDIA Developer, 2020. URL https://developer.nvidia.com/deep-learning-frameworks. Accessed on 2020-10-29.

[62] TensorFlow. tensorflow/tensorflow: An Open Source Machine Learning Framework for Everyone - GitHub, 2020. URL https://github.com/tensorflow/tensorflow. Accessed on 2020-10-29.

[63] Keras. keras-team/keras: Deep Learning for humans - GitHub, 2020. URL https://github.com/keras-team/keras. Accessed on 2020-12-09.

[64] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv: 1408.5093*, 2014. URL https://arxiv.org/abs/1408.5093.

[65] The Apache Software Foundation. apache/incubator-mxnet: Lightweight, Portable, Flexible Distributed/Mobile Deep Learning with Dynamic, Mutation-aware Dataflow Dep Scheduler - GitHub, 2020. URL https://github.com/apache/incubator-mxnet. Accessed on 2020-12-09.

[66] Microsoft. microsoft/CNTK: Microsoft Cognitive Toolkit (CNTK), an open source deep-learning toolkit - GitHub, 2020. URL https://github.com/microsoft/CNTK. Accessed on 2020-12-09.

[67] H. He. The state of machine learning frameworks in 2019. *The Gradient*, 2019. URL https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/.

[68] L. Sousa. Nonconventional computer arithmetic circuits, systems and applications. *IEEE Circuits and Systems Magazine*, 20(4):1–26, Oct. 2020.

[69] J. L. Gustafson. Beyond Floating Point: Next-Generation Computer Arithmetic, 2017. URL http://web.stanford.edu/class/ee380/Abstracts/170201-slides.pdf. Accessed on 2020-09-24.

[70] S. Wang. BFloat16: The secret to high performance on Cloud TPUs | Google Cloud Blog, 2019. URL https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus. Accessed on 2020-11-04.

[71] R. Munafo. Survey of Floating-Point Formats at MROB, 2008. URL http://www.mrob.com/pub/math/floatformats.html. Accessed on 2020-11-04.

[72] U. Kulisch. *Computer Arithmetic and Validity*. De Gruyter, Berlin, Boston, Jan. 2012. ISBN 978-3-11-030179-3. doi: https://doi.org/10.1515/9783110301793. URL https://www.degruyter.com/view/title/126024.

[73] J. Lu, C. Fang, M. Xu, J. Lin, and Z. Wang. Evaluations on Deep Neural Networks Training Using Posit Number System. *IEEE Transactions on Computers*, 14(8):1–1, 2020. ISSN 0018-9340. doi: 10.1109/tc.2020.2985971. URL https://ieeexplore.ieee.org/document/9066876/.

[74] R. M. Montero, A. A. D. Barrio, and G. Botella. Template-Based Posit Multiplication for Training and Inferring in Neural Networks. *arXiv: 1907.04091*, July 2019. URL https://arxiv.org/abs/1907.04091.

[75] H. F. Langroudi, V. Karia, J. L. Gustafson, and D. Kudithipudi. Adaptive Posit: Parameter aware numerical format for deep learning inference on the edge. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 726–727. IEEE, June 2020. doi: 10.1109/cvprw50498.2020.00371. URL https://ieeexplore.ieee.org/document/9151086/.

[76] H. Zhang, J. He, and S.-B. Ko. Efficient Posit Multiply-Accumulate Unit Generator for Deep Learning Applications. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, May 2019. ISBN 9781728103976. doi: 10.1109/iscas.2019.8702349. URL https://ieeexplore.ieee.org/document/8702349/.

[77] Y. Uguen, L. Forget, and F. de Dinechin. Evaluating the Hardware Cost of the Posit Number System. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 106–113. IEEE, Sept. 2019. ISBN 9781728148847. doi: 10.1109/fpl.2019.00026. URL https://ieeexplore.ieee.org/document/8892116/.

[78] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi. Deep Positron: A Deep Neural Network Using the Posit Number System. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1421–1426. IEEE, Mar. 2019. ISBN 9783981926323. doi: 10.23919/date.2019.8715262. URL https://ieeexplore.ieee.org/document/8715262/.

[79] S. Nambi, S. Ullah, A. Lohana, S. S. Sahoo, F. Merchant, and A. Kumar. ExPAN(N)D: Exploring Posits for Efficient Artificial Neural Network Design in FPGA-based Systems. *arXiv: 2010.12869*, 2020. URL https://arxiv.org/abs/2010.12869.

[80] N. Neves, P. Tomás, and N. Roma. Reconfigurable Stream-based Tensor Unit with Variable-Precision Posit Arithmetic. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 149–156. IEEE, jul 2020. doi: 10.1109/asap49362.2020.00033. URL https://ieeexplore.ieee.org/document/9153231.

[81] R. Murillo, A. A. D. Barrio, and G. Botella. Deep PeNSieve: A deep learning framework based on the posit number system. *Digital Signal Processing*, 102:102762, jul 2020. doi: 10.1016/j.dsp.2020.102762. URL https://www.sciencedirect.com/science/article/pii/S105120042030107X.

[82] NGA Team. Unum & Posit- Next Generation Arithmetic. Unum & Posit - Next Generation Arithmetic, July 2019. URL https://posithub.org/. Accessed on 2020-10-16.

[83] A*STAR. Cerlane Leong / SoftPosit · GitLab, Aug. 2018. URL https://gitlab.com/cerlane/SoftPosit. Accessed on 2020-11-01.

[84] Stillwater Supercomputing, Inc. stillwater-sc/universal: Universal Number Arithmetic - GitHub, 2020. URL https://github.com/stillwater-sc/universal. Accessed on 2020-11-02.

[85] K. Mercado. mightymercado/PySigmoid: A Python Implementation of Posits and Quires (Drop-in replacement for IEEE Floats) - GitHub, 2020. URL https://github.com/mightymercado/PySigmoid. Accessed on 2020-11-02.

[86] F. de Dinechin, L. Forget, J.-M. Muller, and Y. Uguen. Posits: the good, the bad and the ugly. In *Proceedings of the Conference for Next Generation Arithmetic 2019*. ACM, Mar. 2019. doi: 10.1145/3316279.3316285. URL https://hal.inria.fr/hal-01959581v3/document.

[87] R. Zhao, B. Vogel, and T. Ahmed. Adaptive Loss Scaling for Mixed Precision Training. *arXiv: 1910.12385*, 2019. URL https://arxiv.org/abs/1910.12385.

[88] S. H. F. Langroudi, T. Pandit, and D. Kudithipudi. Deep Learning Inference on Embedded Devices: Fixed-Point vs Posit. In *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*, pages 19–23. IEEE, Mar. 2018. ISBN 9781538673676. doi: 10.1109/emc2.2018.00012. URL https://ieeexplore.ieee.org/document/8524018/.

[89] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi. Performance-Efficiency Trade-off of Low-Precision Numerical Formats in Deep Neural Networks. In *Proceedings of the Conference for Next Generation Arithmetic 2019*, pages 1–9, New York, NY, USA, Mar. 2019. ACM. ISBN 9781450371391. doi: 10.1145/3316279.3316282. URL https://arxiv.org/abs/1903.10584.

[90] H. F. Langroudi, Z. Carmichael, J. L. Gustafson, and D. Kudithipudi. PositNN Framework: Tapered Precision Deep Learning Inference for the Edge. *Proceedings - 2019 IEEE Space Computing Conference, SCC 2019*, pages 53–59, July 2019. doi: 10.1109/spacecomp.2019.00011. URL https://ieeexplore.ieee.org/document/8853677/.

[91] H. F. Langroudi, Z. Carmichael, and D. Kudithipudi. Deep Learning Training on the Edge with Low-Precision Posits. *arXiv: 1907.13216*, July 2019. URL https://arxiv.org/abs/1907.13216.

[92] J. Lu, S. Lu, Z. Wang, C. Fang, J. Lin, Z. Wang, and L. Du. Training Deep Neural Networks Using Posit Number System. *arXiv: 1909.03831*, Sept. 2019. URL https://arxiv.org/abs/1909.03831.

[93] R. Murillo. RaulMurillo/deep-pensieve: A Deep Learning Framework for the Posit Number System - GitHub, 2020. URL https://github.com/RaulMurillo/deep-pensieve. Accessed on 2020-11-01.

[94] Python Software Foundation. Python Documentation contents - Python 3.9.0 documentation, 2020. URL https://docs.python.org/3/contents.html. Accessed on 2020-11-06.

[95] W. Jakob. pybind/pybind11: Seamless operability between C++11 and Python - GitHub, 2020. URL https://github.com/pybind/pybind11. Accessed on 2020-11-06.

[96] G. Raposo. hpc-ulisboa/posit-neuralnet: PositNN - Framework for training and inference with neural nets usings posits - GitHub, 2020. URL https://github.com/hpc-ulisboa/posit-neuralnet. Accessed on 2020-11-06.

[97] B. Jacob. Eigen (C++ library), 2020. URL http://eigen.tuxfamily.org/. Accessed on 2020-11-08.

[98] SimuNova. MTL 4 C++ library, 2020. URL https://www.simunova.com/en/mtl4/. Accessed on 2020-11-08.

[99] K. Rudolph. Create a multidimensional array dynamically in C++ - Stack Overflow, 2017. URL https://stackoverflow.com/a/47664858/13518635. Accessed on 2020-11-08.

[100] M. J. Wolfe, C. Shanklin, and L. Ortega. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995. ISBN 0805327304.

[101] T. Vieira. Exp-normalize trick - Graduate Descent, Feb. 2014. URL https://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/. Accessed on 2020-11-14.

[102] J. Osier and B. Baccala. GNU gprof, Sept. 1997. URL https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html. Accessed on 2020-11-15.

[103] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *arXiv: 1606.06160*, 2016. URL https://arxiv.org/abs/1606.06160.

[104] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan. Training Deep Neural Networks with 8-bit Floating Point Numbers. *Advances in Neural Information Processing Systems*, 2018-Decem(NeurIPS):7675–7684, Dec. 2018. ISSN 1049-5258. URL https://arxiv.org/abs/1812.08011.

[105] M. Cococcioni, F. Rossi, E. Ruffaldi, and S. Saponara. Fast Approximations of Activation Functions in Deep Neural Networks when using Posit Arithmetic. *Sensors*, 20(5):1–18, Mar. 2020. ISSN 1424-8220. doi: 10.3390/s20051515. URL https://www.mdpi.com/1424-8220/20/5/1515.

[106] R. Banner, I. Hubara, E. Hoffer, and D. Soudry. Scalable Methods for 8-bit Training of Neural Networks. *Advances in Neural Information Processing Systems*, 2018-Decem(NeurIPS):5145–5153, May 2018. ISSN 1049-5258. URL https://arxiv.org/abs/1805.11046.

Appendix A

PositNN

A.1 Functionalities

Table A.1: List of functionalities, organized by category, implemented in PositNN and supporting any posit precision.

Category	Functionalities	Details
Activation Functions	ReLU	Applies the ReLU function.
	Sigmoid	Supports approximation for posits with $es = 0$ [105].
	TanH	Supports approximation for posits with $es = 0$ [105].
Layers	Linear	The same as a Fully Connected or a Dense layer.
	Convolutional	Applies a convolution over a 2D input.
	Average Pooling	Applies a 2D average pooling over an input.
	Maximum Pooling	Applies a 2D maximum pooling over an input.
	Batch Normalization	Applies over a 2D input (mini-batch of 1D samples).
	Range Batch Normalization	Approximation of Batch Normalization [106].
	Dropout	Only affects training.
Loss Functions	Mean Squared Error (MSE)	Corresponds to Squared Error loss.
	Cross Entropy	Equivalent to combining softmax with NLL loss.
Optimizer	Stochastic Gradient Descent (SGD)	Supports momentum, LR scheduler, L2 penalty, etc.
Utilities	StdTensor	Multidimensional array for posits or other data types.
	Convert PyTorch tensors	Convert between PyTorch tensors and StdTensor.
	Copy PyTorch model to PositNN	To convert a model from float to posit.
	Mixed precision tensor	To have the model weights with different precisions.
	Save and load model	Posits are stored in a file in binary form.
	Loss scaling	Scales the gradients as in [15, 87].
	Quire mode	An option to switch between quire capacities [24, 25].
	Underflow	An option to enable underflow for posits.

A.2 PositNN: Training Example

A.2.1 Source files

main.c

Listing A.1: This will execute the training loop. It trains LeNet-5 on Fashion MNIST using posits.

```
// General headers
#include <iostream>
#include <stdio.h>
#include <torch/torch.h>
#include <universal/posit/posit>
#include <positnn/positnn>
using namespace sw::unum;

// Custom headers
#include "LeNet5_float.hpp"
#include "LeNet5_posit.hpp"

// Custom functions
#include "train_posit.hpp"
#include "test_posit.hpp"

// Posit configuration
struct Type{
    typedef posit<16, 2> Optimizer;
    typedef posit<8, 2> Forward;
    typedef posit<8, 2> Backward;
    typedef posit<8, 2> Gradient;
    typedef posit<16, 2> Loss;
    typedef Optimizer SaveFile;
};

// Options
#define DATASET_PATH            "../../../datasets/fashion_mnist"
#define SAVE_FILENAME_POSIT     "model.dat"

int main() {
    std::cout << "Training␣LeNet-5␣on␣Fashion␣MNIST" << std::endl;

    // Training and Testing settings
    size_t const kTrainBatchSize = 64;
    size_t const kTestBatchSize = 1024;
    size_t const num_epochs = 10;
    size_t const kLogInterval = 32;

    // Optimizer parameters
    float learning_rate = 1./16;
    float const momentum = 0.5;
    size_t const adaptive_lr = 4;

    // Float and Posit models
    LeNet5_float model_float;
    LeNet5_posit<Type> model_posit;

    // Copy the float model to initialize the posit model
    copy_parameters(model_float->parameters(), model_posit.parameters());

    // Load Fashion MNIST training dataset and normalize
    auto train_dataset = torch::data::datasets::MNIST(DATASET_PATH)
                            .map(torch::data::transforms::Normalize<>(0.2860, 0.3300))
                            .map(torch::data::transforms::Stack<>());
    const size_t train_dataset_size = train_dataset.size().value();

    // Create data loader from training dataset
    auto train_loader = torch::data::make_data_loader(
                            std::move(train_dataset),
                            torch::data::DataLoaderOptions().batch_size(kTrainBatchSize));

    // Load Fashion MNIST testing dataset and normalize
    auto test_dataset = torch::data::datasets::MNIST(DATASET_PATH,
                            torch::data::datasets::MNIST::Mode::kTest)
```

```
66                          .map(torch::data::transforms::Normalize<>(0.2860, 0.3300))
67                          .map(torch::data::transforms::Stack<>());
68      const size_t test_dataset_size = test_dataset.size().value();
69
70      // Create data loader from testing dataset
71      auto test_loader = torch::data::make_data_loader(
72                          std::move(test_dataset),
73                          torch::data::DataLoaderOptions().batch_size(kTestBatchSize));
74
75      // Optimizer
76      SGD<Type::Optimizer> optimizer_posit(model_posit.parameters(),
77                          SGDOptions<Type::Optimizer>(learning_rate, momentum));
78
79      // Test with untrained model
80      test_posit(model_posit, *test_loader, test_dataset_size);
81
82      // Train the model
83      std::cout << std::endl << "Running..." << std::endl;
84      for (size_t epoch = 1; epoch<=num_epochs; ++epoch) {
85          // Perform a training iteration (1 epoch) and test the model
86          train_posit(epoch, num_epochs,
87                          model_posit, *train_loader, optimizer_posit,
88                          kLogInterval, train_dataset_size);
89          test_posit(model_posit, *test_loader, test_dataset_size);
90
91          // Update learning rate every adaptive_lr epochs
92          if(adaptive_lr>0 && epoch%adaptive_lr==0) {
93              learning_rate /= 2.;
94              optimizer_posit.options().learning_rate = learning_rate;
95          }
96      }
97
98      // Save the trained model
99      save<Type::SaveFile>(model_posit, SAVE_FILENAME_POSIT);
100
101     std::cout << "Finished!\n";
102     return 0;
103 }
```

A.2.2 Header files

LeNet_float.hpp

Listing A.2: Declaration of LeNet-5 with PyTorch.

```
1  // General headers
2  #include <torch/torch.h>
3
4  struct LeNet5_floatImpl : torch::nn::Module {
5      LeNet5_floatImpl() :
6          conv1(torch::nn::Conv2dOptions(1, 6, 5).padding(2)),
7          conv2(torch::nn::Conv2dOptions(6, 16, 5)),
8          conv3(torch::nn::Conv2dOptions(16, 120, 5)),
9          fc1(120, 84),
10         fc2(84, 10)
11     {
12         register_module("conv1", conv1);
13         register_module("conv2", conv2);
14         register_module("conv3", conv3);
15         register_module("fc1", fc1);
16         register_module("fc2", fc2);
17     }
18
19     torch::Tensor forward(torch::Tensor x) {
20         x = conv1->forward(x);
21         x = torch::max_pool2d(x, 2, 2);
22         x = torch::relu(x);
23
24         x = conv2->forward(x);
25         x = torch::max_pool2d(x, 2, 2);
26         x = torch::relu(x);
27
```

```
28          x = conv3->forward(x);
29          x = torch::relu(x);
30
31          x = x.view({-1, 120});
32
33          x = fc1->forward(x);
34          x = torch::relu(x);
35
36          x = fc2->forward(x);
37          return torch::log_softmax(x, 1);
38      }
39
40      torch::nn::Conv2d conv1, conv2, conv3;
41      torch::nn::Linear fc1, fc2;
42  };
43
44  TORCH_MODULE(LeNet5_float);
```

LeNet_posit.hpp

Listing A.3: Declaration of LeNet-5 with PositNN.

```
1   // General headers
2   #include <positnn/positnn>
3
4   template <typename T>
5   class LeNet5_posit : public Layer<typename T::Optimizer>{
6   public:
7       LeNet5_posit() :
8           conv1(1, 6, 5, 1, 2),
9           conv2(6, 16, 5),
10          conv3(16, 120, 5),
11          fc1(120, 84),
12          fc2(84, 10),
13          max_pool1(2, 2),
14          max_pool2(2, 2)
15      {
16          this->register_module(conv1);
17          this->register_module(conv2);
18          this->register_module(conv3);
19          this->register_module(fc1);
20          this->register_module(fc2);
21      }
22
23      // Posit precisions
24      using O = typename T::Optimizer;
25      using F = typename T::Forward;
26      using B = typename T::Backward;
27      using G = typename T::Gradient;
28
29      StdTensor<F> forward(StdTensor<F> x) {
30          x = conv1.forward(x);
31          x = max_pool1.forward(x);
32          x = relu1.forward(x);
33
34          x = conv2.forward(x);
35          x = max_pool2.forward(x);
36          x = relu2.forward(x);
37
38          x = conv3.forward(x);
39          x = relu3.forward(x);
40
41          x.reshape({x.shape()[0], 120});
42
43          x = fc1.forward(x);
44          x = relu4.forward(x);
45
46          x = fc2.forward(x);
47          return x;
48      }
49
50      StdTensor<B> backward(StdTensor<B> x) {
51          x = fc2.backward(x);
```

```
52
53          x = relu4.backward(x);
54          x = fc1.backward(x);
55
56          x.reshape({x.shape()[0], 120, 1 ,1});
57
58          x = relu3.backward(x);
59          x = conv3.backward(x);
60
61          x = relu2.backward(x);
62          x = max_pool2.backward(x);
63          x = conv2.backward(x);
64
65          x = relu1.backward(x);
66          x = max_pool1.backward(x);
67          x = conv1.backward(x);
68          return x;
69      }
70
71  private:
72      Conv2d<O, F, B, G> conv1, conv2, conv3;
73      Linear<O, F, B, G> fc1, fc2;
74      MaxPool2d<F, B> max_pool1, max_pool2;
75      ReLU relu1, relu2, relu3, relu4;
76  };
```

train_posit.hpp

Listing A.4: Training loop of a model during 1 epoch of the training dataset.

```
1   // General headers
2   #include <cstdint>
3   #include <iostream>
4   #include <torch/torch.h>
5   #include <positnn/positnn>
6
7   template <typename Type, template<typename> class Model,
8             typename DataLoader, typename Optimizer>
9   void train_posit(size_t epoch, size_t const num_epochs,
10                   Model<Type>& model, DataLoader& data_loader, Optimizer& optimizer,
11                   size_t const kLogInterval, size_t const dataset_size) {
12      // Setup training
13      model.train();
14      size_t batch_idx = 0;
15      size_t total_batch_size = 0;
16
17      // Setup data types
18      using F = typename Type::Forward;
19      using L = typename Type::Loss;
20      using T = unsigned short int;
21
22      for(auto const& batch : data_loader) {
23          // Update number of trained samples
24          size_t const batch_size = batch.target.size(0);
25          total_batch_size += batch_size;
26
27          // Get data and target
28          auto data_float = batch.data;
29          auto target_float = batch.target;
30
31          // Convert data and target to float32 and uint8
32          data_float = data_float.to(torch::kF32);
33          target_float = target_float.to(torch::kUInt8);
34
35          // Convert data and target from PyTorch Tensor to StdTensor
36          auto data = Tensor_to_StdTensor<float, F>(data_float);
37          auto target = Tensor_to_StdTensor<uint8_t, T>(target_float);
38
39          // Forward pass
40          auto output = model.forward(data);
41          cross_entropy_loss<L> loss(output, target);
42
43          // Backward pass and optimize
```

```
        optimizer.zero_grad();
        loss.backward(model);
        optimizer.step();

        // Print progress
        if(++batch_idx % kLogInterval == 0) {
            float loss_value = loss.template item<float>();

            std::printf("Train␣Epoch:␣%.3f/%2ld␣Data:␣%5ld/%5ld␣Loss:␣%.4f\n",
                        epoch-1+static_cast<float>(total_batch_size)/dataset_size,
                        num_epochs, total_batch_size, dataset_size, loss_value);
        }
    }
}
```

test_posit.hpp

Listing A.5: Tests a model with the entire testing dataset.

```
// General headers
#include <cstdint>
#include <iostream>
#include <torch/torch.h>
#include <positnn/positnn>

template <typename Type, template<typename> class Model, typename DataLoader>
void test_posit(Model<Type>& model, DataLoader& data_loader, size_t dataset_size) {
    // Setup inference
    model.eval();
    float test_loss = 0;
    size_t correct = 0;

    // Setup data types
    using F = typename Type::Forward;
    using L = typename Type::Loss;
    using T = unsigned short int;

    // Loop the entire testing dataset
    for(auto const& batch : data_loader) {
        // Get data and target
        auto data_float = batch.data;
        auto target_float = batch.target;

        // Convert data and target to float32 and uint8
        data_float = data_float.to(torch::kF32);
        target_float = target_float.to(torch::kUInt8);

        // Convert data and target from PyTorch Tensor to StdTensor
        auto data = Tensor_to_StdTensor<float, F>(data_float);
        auto target = Tensor_to_StdTensor<uint8_t, T>(target_float);

        // Forward pass
        auto output = model.forward(data);

        // Calculate loss
        test_loss += cross_entropy_loss<L>(output, target,
                        Reduction::Sum).template item<float>();

        // Get prediction from output
        auto pred = output.template argmax<T>(1);
        correct += pred.eq(target).template sum<size_t>();
    }

    // Get average loss
    test_loss /= dataset_size;

    // Print results
    std::printf("Test␣set:␣Loss:␣%.4f␣|␣Accuracy:␣[%5ld/%5ld]␣%.4f\n",
                        test_loss, correct, dataset_size,
                        static_cast<float>(correct)/dataset_size);
}
```