Supporting Posits in Deep Learning Frameworks: A PyTorch Implementation

Afonso Vieira da Silva Luz

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisors: Doctor Nuno Filipe Valentim Roma
Doctor Pedro Filipe Zeferino Aidos Tomás

Examination Committee

Chairperson: Doctor José Fernando Alves da Silva
Supervisor: Doctor Nuno Filipe Valentim Roma
Member of the Committee: Doctor Gabriel Falcão Paiva Fernandes

November 2021

Acknowledgments

First, I would like to thank my supervisors, Doctor Nuno Roma and Doctor Pedro Tomás, who were available to provide guidance and support throughout the whole duration of the development of this work. Their commitment to leading and helping whenever necessary was invaluable to successfully complete this MSc thesis. I would also like to thank João Vieira for his help in quickly and successfully addressing any technical issues that arose throughout these months.

I would like to express my gratitude towards my family, who have always cultivated in me the drive towards knowledge acquisition and mastery. During these months, their support was the safe haven I needed when difficulties were presented.

To my friends, who have shared experiences, trips and stories during these months, their presence helped a lot in this endeavour.

Finally, I would like to thank INESC-id for having given me access to their infrastructure, namely their computing platforms, which helped in the successful testing of this work.

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under projects UIDB/CEC/50021/2020, PTDC/EEI-HAC/30485/2017 and PCIF/MPG/0051/2018.

Resumo

Reduzir o consumo de energia de implementações de aprendizagem profunda tem vindo a atrair um interesse crescente nos últimos anos. Isto é particularmente relevante em aplicações onde existem limitações de energia, como dispositivos espaciais e aéreos. Com esta finalidade, o formato numérico *posit* tem mostrado resultados promissores como uma alternativa ao sistema de vírgula flutuante IEEE para cálculos de aprendizagem profunda. Investigação recente sugere que *posits* de 16 *bits* atingem resultados semelhantes a números de vírgula flutuante IEEE-754 de 32 *bits* e que mesmo *posits* com menos *bits* podem ser usados para treinar e avaliar modelos de aprendizagem profunda. No entanto, para estudar a utilização de *posits* em aprendizagem profunda, é necessário desenvolver funções específicas, uma vez que as ferramentas mais populares ainda não suportam este formato numérico. Este trabalho pretende colmatar essa lacuna, integrando *posits* de forma nativa no PyTorch, a ferramenta mais popular para investigação em aprendizagem profunda. A implementação proposta torna os *posits* um tipo de dados base da ferramenta, o que significa que podem ser usados da mesma forma que qualquer outro tipo de dados já suportado. Para validar a implementação, a rede neuronal convolucional LeNet-5 foi treinada e testada utilizando posits nas bases de dados MNIST e FashionMNIST. Os resultados obtidos com *posits* de 16 *bits* foram semelhantes aos de números de vírgula flutuante IEEE-754 de 32 *bits*, sugerindo que a implementação dos operadores considerados para *posits* está correta. Para difundir esta contribuição, o código e documentação produzidos foram disponibilizados num repositório de *GitHub* público.

Palavras-chave: Formato numérico *posit*, aprendizagem profunda, redes neuronais, PyTorch

Abstract

Reducing the energy consumption of computationally intensive deep learning implementations has received a growing interest in the last years. This is particularly relevant in applications where there are strict energy restrictions, such as space and aerial devices. To this end, the posit number format has shown promising results as a more energy efficient replacement to the standard IEEE-754 floating-point for deep learning computations. Recent research suggests that 16-bit posits achieve similar results as 32-bit floating-point and even smaller posits can be used to train and evaluate deep learning models. However, to study the use of posits for deep learning, researchers have to develop customized functions, since the most popular deep learning frameworks do not yet support posits. This work aims at bridging this gap, by integrating posits natively in PyTorch, the most popular framework for deep learning research. The proposed implementation makes posits a built-in data type in the framework, which means that they can be used in the same way as any other data type that the framework already supports. To validate the implementation, the convolutional neural network LeNet-5 was trained and tested using posits on the MNIST and FashionMNIST datasets. The obtained results with 16-bit posits were similar to those with 32-bit floating-point, suggesting that the implementation of the considered posit operators is correct. To disseminate this contribution, the produced code and documentation was made available on a public GitHub repository.

Keywords: Posit numerical format, Deep Learning (DL), Neural Networks (NNs), PyTorch

Contents

List of Tables

List of Figures

Listings

Nomenclature

Greek symbols

β_1, β_2	Exponential decay rates for the moments estimates.
ϵ	Smoothing term that prevents division by 0.
η	Learning rate.

Roman symbols

a	Neuron output after activation function.
b	Bias.
C	Number of channels.
H	Height.
K	Kernel size.
L	Loss function.
l	Layer index.
m	First moment (mean).
$\hat{m}$	Bias-corrected first moment.
N	Batch size.
P	Padding.
S	Stride.
v	Velocity/momentum term or second moment (uncentered variance).
$\hat{v}$	Bias-corrected second moment.
W	Width.
w	Weight tensor.
x	Input sample.

$\hat{y}$	Model output.
z	Neuron output before activation function.

Subscripts

i, j, k, m	Computational indexes.

Superscripts

l	layer index.

Abbreviations

1-D 1-dimensional

2-D 2-dimensional

3-D 3-dimensional

4-D 4-dimensional

Adam Adaptive Moment Estimation

AI Artificial Intelligence

NN Neural Network

ASIC Application-Specific Integrated Circuit

CNN Convolutional Neural Network

CPU Central Processing Unit

DL Deep Learning

es exponent size

FC Fully Connected

FCNN Fully Connected Neural Network

FPGA Field-Programmable Gate Array

GPU Graphics Processing Unit

IEEE 754 IEEE Standard for Floating-Point Arithmetic

ILSVRC ImageNet Large Scale Visual Recognition Challenge

MSc Master of Science

MSE Mean Squared Error

NaN Not a Number

NaR Not a Real

$nbits$ number of bits

NGA Next Generation Arithmetic

NLL Negative Log Likelihood

ReLU Rectified Linear Unit

SGD Stochastic Gradient Descent

TanH Hyperbolic Tangent

TPU Tensor Processing Unit

Chapter 1

Introduction

Contents

1.1 Motivation

The fascination with building machines capable of mimicking human behaviour has long been one of the human race, from scientists to science fiction writers. At first, physical machines to perform physical tasks was the main focus, but the interest quickly turned to the search for synthesizing human-like intelligence - or even greater.

Interest in forms of machine intelligence was, thus, interdisciplinary, up to 1950, when the Turing test was introduced [1] and modern Artificial Intelligence (AI) research is considered to have started [2], leading to its development as an independent field. However, unrealistic hopes were placed on the field, and, in the 1970s, the unmet expectations led to what is known as the AI Winter [3], staling the field for many decades.

However, with the increase in the available computing power, some practical successes revived interest in the field [4]. In particular, Deep Learning (DL) was one of the subfields that most grew, currently benchmarking results in visual recognition, natural language processing, fraud detection, among others [5].

Deep Learning models are defined as being those with several layers, to represent multiple levels of abstraction, potentiating the ability to learn features from data and make future predictions [5]. The most popular model types are Deep Artificial Neural Networks (NNs) [6]. These are comprised of a variable number of layers, each consisting of neurons holding weights that get updated with the labeled input data. In complex networks, the number of weights can reach hundreds of millions [7], representing very high computational demands.

To design and train NNs, there are several available frameworks that abstract the implementation details from the end user. The most popular are PyTorch [8], developed by Facebook, and TensorFlow [9], developed by Google. The first is currently the most popular among researchers, while the latter is the most used in production [10].

Given its computationally intensive nature, there is significant research interest in reducing the memory footprint and energy consumption of the operations performed by DL models [11]. This is particularly relevant in domains where there are strict constraints on processing capacity and energy consumption, such as space applications [12].

One possible approach to reduce this footprint is by searching for alternative formats to the traditional IEEE-754 32-bit floating-point numbers for computations, such as integers [13] and fixed-point numbers [14]. The introduction of the posit number format in 2017, with claims that it offers a higher accuracy and requires simpler hardware and exception handling structures than IEEE-754 floating-point numbers [15], has led to an increasing interest in using it within DL applications [16–18]. Recent research as shown that the same accuracy can be achieved with 16-bit posits as with 32-bit floats, and even smaller posit configurations have achieved state-of-the-art results [19, 20]. This means that, by using less bits, there is the potential of reducing hardware resources, energy consumption and memory footprint.

The reduction of energy consumption is of great interest for the aforementioned space applications, where energy requirements tend to be the bottleneck. As an example, Neuraspace [21] is using Machine

Learning models to detect and prevent collisions between satellites. Maintaining accuracy while reducing resources would be of great interest to extend this application.

However, in order to train and test DL models with Posits, researchers have to develop all the functions and operators used for DL computations from scratch [19, 20]. This constitutes a significant overhead in the research process, which could only be mitigated if the most common DL frameworks supported the posit number format.

This premise motivated the work presented in this thesis: to introduce support of Posits in the most widely used DL framework for research, PyTorch. This contribution will not only have the goal of providing a tool to train NNs with Posits, but also of documenting the necessary steps to add further functionalities related to Posits. At the time of development of this work, hardware units specific for Posits have already been proposed [11, 16, 22], but these are not complete enough to be integrated with PyTorch. This way, this work also aims at providing a tool to test future developments of hardware for Posits with PyTorch.

1.2 Objectives

The main goal of this dissertation is to introduce native support of the posit data type and related operations in PyTorch. To this end, the fundamental objectives can be summarized as:

- Provide posit support to the main layers and functions of NNs in PyTorch;
- Expose these functions in the Python frontend in the same way as other native PyTorch data types;
- Test the conducted implementation through an end-to-end design and training of a Neural Network;
- Document the process of extending posit support for further operators.

1.3 Thesis Outline

This dissertation is organized in multiple chapters, each addressing a relevant topic of this work.

- Chapter 2 presents the background for this work. It starts by presenting the characteristics of different computer number formats, namely integers, fixed-point, floating-point and Posit. It also discusses some of the available arithmetic libraries for Posits. It follows with the presentation of the main layers and functions of Neural Networks, as well as reference datasets and models, and the most popular DL frameworks. It finishes with a summary of the most relevant literature and related work on DL with Posits.
- Chapter 3 aims at describing the frontend API of PyTorch. The tensor data structure is presented, since it constitutes the core of all data representation within the framework. It follows by presenting how the concepts introduced in the previous chapter are available in the framework. It finishes with an end-to-end example of designing and training a Neural Network in PyTorch.

- Chapter 4 explores the internal structure of PyTorch and the process of supporting Posits in it. It starts with a brief description of how to contribute to the codebase of this framework. It follows with a description of the main concepts that compose the internal structure of the framework. Finally, the contribution of this work through the creation of a new data type for Posits, the support for the main NN operators and how Posits are exposed in the frontend is presented.
- Chapter 5 presents the experimental evaluation that was conducted to validate the implementation of Posits in the framework. It starts by exposing the used experimental setup, explaining the need to reduce the size of the original dataset. Finally, the results of training the LeNet-5 network with Posits are presented.
- Chapter 6 summarizes the conclusions of the present work and draws considerations on future work.

Chapter 2

Background

Contents

Given that the purpose of this work is to integrate a novel numerical format into an established Deep Learning framework, this chapter provides context on its two main topics: computer number formats and Deep Learning. It starts by exposing different numerical formats used in computers, with emphasis on the characteristics of the posit number format. It is followed by an overview of the main aspects of Deep Learning (DL), some of its reference models along with frameworks used for DL applications. It finishes with a brief survey of previous research into the use of Posits in DL applications, as well as existing frameworks to build DL models supporting computation with Posits.

2.1 Computer Number Formats

Given the digital nature of modern computers, information is represented in binary form, numbers in particular. Encoding numbers - be them integer, real, or complex - into a limited amount of bits can be achieved in different ways, each with advantages and disadvantages regarding not only the range and granularity, but also the complexity of their handling for calculations.

Because of their vital importance for computer science and, in particular, in the field of Deep Learning [13, 14, 23], this section exposes some of the most common binary encodings for real numbers.

2.1.1 Integers

Representing unsigned integer numbers in binary is trivial, following the general rule of representing numbers in any base: each digit multiplies the base to the power of the digit's index. As an example for base 2, the number 10110_2 represents, in base 10, the number $1\times2^4+0\times2^3+1\times2^2+1\times2^1+0\times2^0 = 22$. As for signed numbers, the initial approach was to use one bit for the signal and the remaining bits for its absolute value. This had several inconveniences, such as the double representation of zero, and the need to perform operations differently depending on the magnitude and sign of the numbers involved [24].

The introduction of two's complement encoding [25], where a negative integer number x is encoded with n bits as $2^n + x$, solved both of the above problems, becoming the standard after IBM's System/360 adopted it [26]. As an example, the two's complement representation of -2 with 3 bits is 110_2, since $2^3 - 2 = 6$. However, codifying real numbers in binary is a greater challenge, and different possible representations have been put forward.

2.1.2 Fixed Point

A given interval $[n, m]$ has finite cardinality for integer numbers, but for real numbers that is no longer the case. This means that, when encoding real numbers in a finite number of bits, there will need to be a compromise between range (the amplitude of the interval that can be represented) and precision (the rounding error when encoding a number).

Fixed point encoding is the simplest representation for non-integer numbers, characterized by having l bits to the left of the radix point and r bits to the right. The bits to the right of the radix point are given negative weights. Two's complement is also used to represent signed numbers, as in integer encoding.

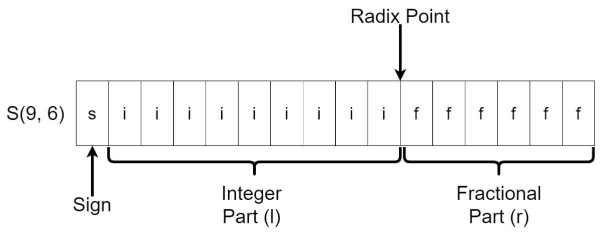


Figure 2.1: Example of a fixed point number layout: the radix point is 9 bits to the right of the sign bit and it has 6 fractional bits, obtained from [27].

This way, to decode a fixed point number, the whole bitset can be regarded as representing an integer in two's complement, which, after decoding, is scaled by the power of 2 determined by the number of bits to the right of the radix point, r:

$$x = \text{decoded integer} \times 2^{-r} \tag{2.1}$$

This resemblance with integers means that operations with fixed point numbers are very similar to those with integers, hence avoiding the need for specific processing units.

Another way to decode a fixed point number is to consider the sign bit negative, making use of equation (2.2), thus avoiding taking the two's complement to decode it [28].

$$x = -x_{l-1}2^{l-1} + \sum_{i=-r}^{l-2} x_i 2^i \tag{2.2}$$

From this representation, it can be seen that the difference between two consecutive fixed point numbers is always the same, 2^{-r}, leading to a rounding error of at most 2^{-r-1}. The range is given by the bits to the left of the radix point, from -2^{l-1} to $2^{l-1} - 1$.

For a bitset of size n, moving the radix point to the left (increasing the number of bits to the right of it) will result in more precision and less range, whereas moving it to the right will have the opposite effect. The choice will, therefore, depend on the application's requirements.

2.1.3 Floating Point

The numerical range offered by fixed point encoding is insufficient for many computer applications, consequently, its use is limited to applications where performance is very important and range and/or precision can be sacrificed [29].

To increase the obtainable range, floating point formats are regarded as the most appropriate. These formats are characterized by having the radix point "float" within the number, that is, there is no fixed number of bits to the right nor to the left of the radix point. This is achieved by using the same principle as in scientif notation: a *mantissa* is multiplied by 2 raised to the power of the *exponent*.

Several formats could be used to represent a floating point number, but the one used in most computer systems is the IEEE 754 Standard for Floating-Point Arithmetic [30], introduced in 1985. To decode a floating point number encoded through this standard, equation (2.3) should be used.

$$x = (-1)^S \times M \times 2^{E-b} \tag{2.3}$$

S represents the value of the sign bit (the most significant bit); M is the mantissa (where the leading bit is implicit, 1 for normalized representation and 0 for subnormal); E is the value of the exponent; $b = 2^{exponentsize-1} - 1$ is a bias applied to the exponent in order to have both negative and positive exponents represented with unsigned integers only, avoiding 2's complement all together.

From IEEE's standard, the most commonly used floating-point formats are single precision (32 bits) and double precision (64 bits). Figure 2.2 shows the bit usage in single precision floats: bits 0 to 22 form the mantissa, bits 23 to 30 form the exponent, and bit 31 is the sign bit.

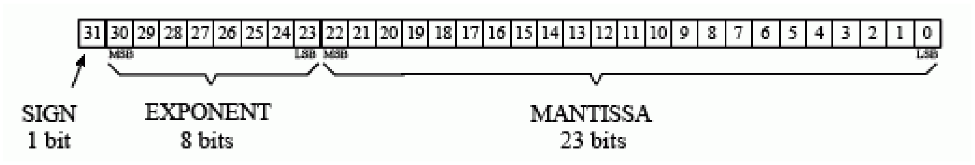


Figure 2.2: Single precision floats bit encoding, obtained from [31].

An example of the decoding is provided in Figure 2.3, where the normalized form is used (the implicit mantissa bit is 1).

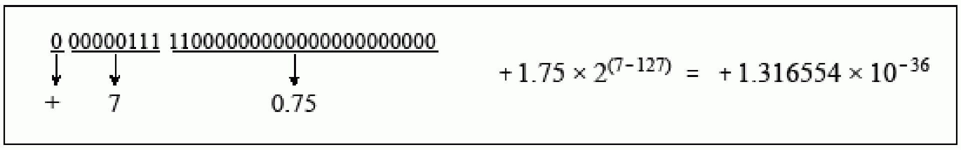


Figure 2.3: Example of the decoding of a single precision float, obtained from [31].

IEEE's standard reserves some bit patterns to represent special numbers, namely:

- ± 0: all exponent and mantissa bits set to 0
- $\pm\infty$: all exponent bits set to 1 and all mantissa bits set to 0
- NaN (Not a Number): all exponent bits set to 1 and all combinations of mantissa bits (except all 0s)

NaN values are used for undefined or unrepresentable real numbers - the square root of a negative number, for instance. For single precision floats, there are 2^{24} NaN values, which means that approximately 0.4% of encodings are unusable. For double precision floats this percentage is approximately 0.05%.

The representation range of floating point numbers is determined by the magnitude of the exponent: $[2^{-2^{\text{exponent size}-1}+2}, 2^{2^{\text{exponent size}-1}}]$ as opposed to fixed point, where it is the number of bits to the left of the radix point. For single precision floats (32 bits), the range is approximately $[10^{-38}, 10^{38}]$. In order to achieve the same range with fixed point numbers, 128 bits would need to be at the left side of the radix point, which illustrates the power of floating point numbers to store both very small and very large numbers.

However, this large dynamic range comes at the cost of having unequally spaced numbers: the gap between two consecutive numbers n and m is approximately $m - n = n/10^7$ [31]. This means that for smaller numbers the gap between consecutive numbers is smaller than for big numbers, which is desirable since precision is more important when dealing with small quantities.

There are other formats of floating point encodings. Among others, Half floating point and Brain Floating Point (BFloat16) [32]. Half floating point is defined in IEEE's standard as having 16 bits, of which 1 is for the sign, 5 for the exponent and 10 for the mantissa. With less exponent and mantissa bits, both precision and range are diminished when compared with single precision floats.

BFloat16, developed by Google, is a truncated version of single precision floats, thus having the same 8 exponent bits but only 7 mantissa bits. This means that it has the same range as single precision floats, with only the precision being affected by the use of a the smaller mantissa. This encoding was developed with Deep Learning applications in mind, since Neural Networks are more sensitive to exponent size than that of the mantissa [32].

2.1.4 Posit

Despite the current ubiquity of IEEE's floating point format in computer systems, it has some problems and limitations, such as breaking linear algebra laws (e.g. due to the rounding process, $(a + b) + c \neq a + (b + c)$), overflowing to infinity and underflowing to 0, or the complexity of its manipulation, especially due to the multiple NaN values, leading to complex hardware to support it [33].

From the desire to overcome some of these problems, John L. Gustafson has recently proposed a set of alternative formats, denoted by Universal numbers (Unums). Firstly, Type I unum [34] introduced an extra bit to IEEE's floating point to assert whether the represented number was exact or whether it represented the lower end of an interval containing that number. With the introduction of Type II unum, compatibility with IEEE's standard was broken, in order to have more freedom to make it more hardware friendly [35].

In 2017, Type III unum - also known as Posit - was introduced, having in mind the complete replacement of IEEE's floating point. Gustafson states that this new format offers a larger dynamic range, a higher accuracy and requires simpler hardware and exception handling structures than traditional IEEE-754 standard floating-point numbers [15].

A posit number is defined by its total number of bits ($nbits$) and by the size of the exponent field (es). This way, a posit configuration is usually represented in the form posit($nbits$, es). The layout of a posit number (p) is represented in Figure 2.4, and its decoding is shown in equation (2.4).

$$x = \begin{cases} 0, & p = 000...0, \\ \pm\infty = \text{NaR}, & p = 100...0, \\ (-1)^s \times useed^k \times 2^e \times f, & \text{all other } p. \end{cases} \tag{2.4}$$

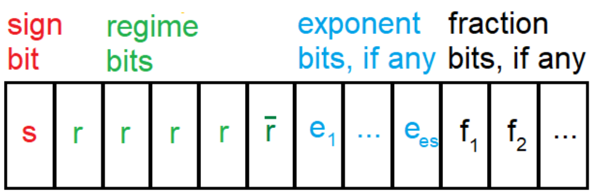


Figure 2.4: Generic representation of an $nbits$-bit posit with es exponent bits.

The meaning of the sign bit is similar to other formats: 0 for positive numbers and 1 for negative numbers. If the number is negative, the 2's complement of the other fields must be taken before decoding. The regime field is characterized by a run of identical bits (r) that is either terminated by an opposite bit ($\bar{r}$) or by using up all the $nbits - 1$ bits. The numerical value (k) of the regime is given by the count of identical bits in this run. For a run of m bits, all equal to 0, $k = -m$; if they are 1s then $k = m - 1$.

As an example, for a posit with $nbits = 5$ and $es = 2$ the possible decoding options of the regime are represented in Table 2.1, after taking the 2's complement in case of a negative sign bit.

Table 2.1: Example of regime field decoding for a posit with $nbits = 5$ and $es = 2$.

Binary	s0000	s0001	s001e_1	s01e_1e_2	s10e_1e_2	s110e_1	s1110	s1111
Numerical meaning, k	-4	-3	-2	-1	0	1	2	3

Hence, this simple example illustrates how the number of exponent bits varies depending on the length of the regime. In particular, while for such posits fraction bits might not even exist, for larger posits their number is also dependent on the length of the regime.

As represented in equation (2.4), the numerical value of the regime, k, represents an additional exponent, applied over $useed = 2^{2^{es}}$. Hence, this scale factor is given by $useed^k$. The value encoded by the exponent bits (e) represents a scaling by 2^e. Finally, just as in normalized IEEE's floats, the value encoded in the fraction bits (f) has a hidden bit at 1. Therefore, the scale factor is $1.f$, as shown in equation (2.4).

There are two special encodings that represent 0 (all bits at 0) and Not a Real (NaR) or $\pm\infty$ (first bit at 1 followed by all bits at 0). Having only two exception values with simple bit patterns makes their manipulation at the hardware level easier than the several NaN values of floats [15].

Contrary to IEEE floats, there is no bias for the exponent field, since negative values of k already lead to negative exponents. The variable size of the fraction field - determined by the size of the regime - leads to a tapered precision, that is, numbers with magnitude near 1 have more fraction bits therefore greater precision, whereas very small and very large numbers have smaller precision. This is illustrated in Figure 2.5 where the decimal accuracy (i.e. how many digits to the right of the decimal point are correctly represented) is compared for 8-bit fixed point, 8-bit floatint-point and different configurations of 8-bit Posits.

The dynamic range of a positive ($nbits$, es) posit number is [$useed^{nbits-2}$, $useed^{2-nbits}$], which means that, for a given posit size (nbits), the range increases only with the increase of the number of exponent bits, since $useed = 2^{2^{es}}$. Just as in floats, increasing the size of the exponent field comes at

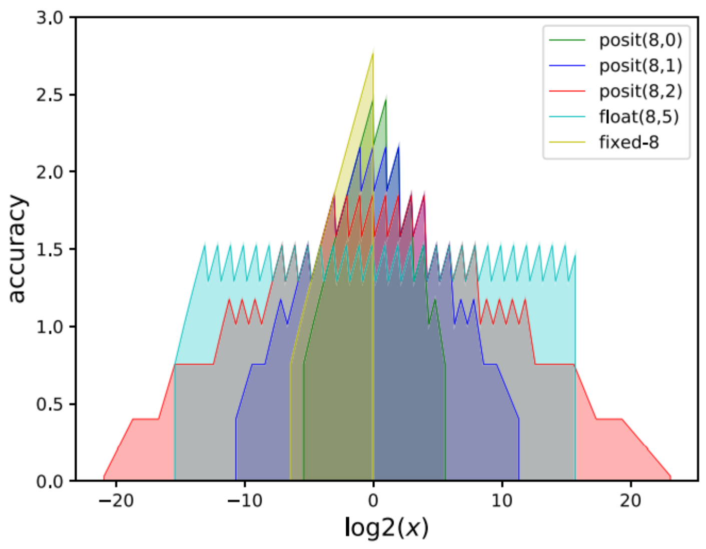


Figure 2.5: Decimal accuracy of 8-bit fixed-point, 8-bit floating-point and several 8-bit Posits, obtained from [18].

the cost of decreasing the size of the fraction, hence reducing precision.

Posit Standard and Quire

The previously described posit format is in accordance with the Posit Standard published in 2018, publicly available at the Posit working group's web page [36]. Nevertheless, there have been major updates to this Standard, with the most recent version, as of June 2021, kindly emailed by Dr. Gustafson [37]. This new version of the Standard establishes a fixed size of the exponent (as 2 bits) for all posit sizes, with the goal of making casting between posits with different sizes much easier: padding with 0s will make it larger and removing the least significant bits will make it smaller, thus avoiding the need to decode before casting.

One of the intended uses of posit numbers is low precision applications (16 or less bits) [15]. Under this premise, a particular mechanism to decrease the rounding error when doing chained operations, such as sum of products (in matrix multiplications, for example), was introduced upon the proposal of posit: the **quire**. The quire is the posit equivalent of a Kulisch accumulator [38], with each posit configuration having an associated quire configuration. Upon its introduction, the size of the quire for a $(nbits, es)$ posit number was $nbits^2$. However, with the new version of the standard, it is now fixed at $16 \times nbits$. Its layout is presented in Figure 2.6.

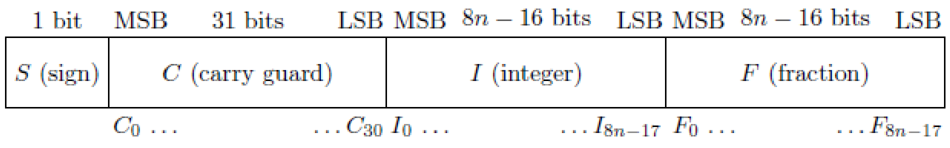


Figure 2.6: Quire layout according to the most recent Posit Standard, obtained from [37].

Being represented using a fixed-point format, decoding the quire is straightforward: the 2's complement of the signed integer represented by all bits concatenated is taken and multiplied by $2^{16-8nbits}$: $q =$ decoded integer $\times$ $2^{16-8nbits}$. The only exception value is the representation where the sign bit is equal to 1 and all other fields are set to 0, which represents a NaR.

The quire is particularly useful with small posits, in applications where sums of products are ubiquitous, such as Deep Learning [19], given its potential to minimize rounding errors and thus compensate the reduced accuracy of small posits.

2.1.5 Posit Arithmetic Libraries

Developing hardware units for posit computations, synthesized either for Field-Programmable Gate Arrays (FPGAs) or Application-Specific Integrated Circuits (ASICs), has been an active topic of research [11, 16, 17, 39]. Many of these frameworks were developed with Deep Learning applications in mind [11, 16, 17]. In [22], a tensor unit supporting posit arithmetic to be used in DL applications with Posits was proposed. However, at the time of the development of this thesis, these hardware implementations are not complete and flexible enough to be used as the backbone of support for posits in DL frameworks, such as PyTorch or TensorFlow, as also noted by [19] and [20].

A possible alternative is to use a library that simulates operations with Posits via software. A survey of the existing libraries, up to mid-2019, was conducted by the Next Generation Arithmetic team (NGA) [40], of which John L. Gustafson (proposer of the posit format) is a member. In it, the more complete libraries with posit operations emulated via software are: PySigmoid [41], SoftPosit [42] and Universal [43].

PySigmoid is a Python library supporting any arbitrary posit configuration, together with math operations (basic linear algebra, square root, trigonometric functions, etc.) and with an easy to use interface through a class that represents a posit number with overloaded operators. Nonetheless, it is fully implemented in Python, which makes it difficult to integrate with Pytorch's C++ backend, and it is no longer mantained.

SoftPosit is a C library that offers a set of functions to operate on Posits and quire, tested upon its development. It supports any posit configuration of the form posit($nbits$, 2), with $nbits = \{2, \cdots, 32\}$ along with posit(8, 0) and posit(16, 1), which makes it compliant with the new standard, even if not all configurations are supported.

Universal is a header only C++ library, supporting any arbitrary configuration of Posits and quire. It also supports the two other Unum formats (Type-I and Type-II). It has a comprehensive test suite for the mathematical operations it supports and is frequently maintained, which makes it very reliable.

From the libraries considered at the start of this work, the **Universal** library was the one chosen, mainly for its reliability and its easier integration with the C++ backend of PyTorch, where most of the work was to be developed. It has a comprehensive support for all the main functions and operators needed to implement DL operations, along with active contributors available to clarify doubts on the library.

Furthermore, Universal is a header-only library, so there is no need to build it when building PyTorch's source code, making integration seamless. Moreover, Universal implements Posits and the quire as templated classes, whose arguments correspond to the $nbits$ and es parameters of posits and $capacity$ for quires, meaning that adopting it with the different posit configurations is straightforward and involves minimal alterations to the code.

2.2 Deep Learning - Neural Networks

Deep Learning (DL) is a subfield of Artificial Intelligence characterized by models with multiple layers, where each layer builds knowledge on top of the previous one [44]. These models are fed with input data and, by iterating over this data, they update their parameters, effectively learning from experience. The advantage of DL models over simpler machine learning techniques is their ability to capture complex non-linear behaviours by decomposing them into simpler tasks addressed by each layer [45].

Currently, the most widespread DL models are Deep Artificial Neural Networks (NNs), inspired by the connections between neurons in the brain. This section briefly describes how these models are structured, how they learn from the input and output data, along with some benchmark examples for image recognition, which is the topic of this work.

2.2.1 Overview of Neural Networks

Drawing inspiration from the synapses that occur between neurons in the human brain, NNs have weights connecting neurons in consecutive layers. These weights represent the transmission of a signal between neurons, which, together with an associated bias, pass through a non-linear activation function to determine if the neuron is ”fired” or not.

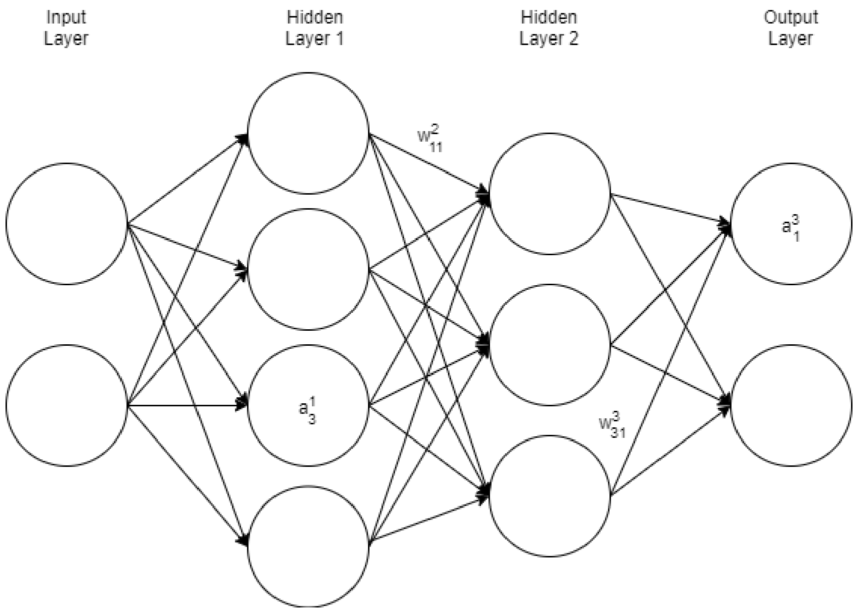


Figure 2.7: Example of a Neural Network with 2 hidden layers

Figure 2.7 shows an example NN with 3 layers (the input layer is not counted). In order for a NN to be considered deep, it should have at least 2 hidden layers. The input layer is simply the input fed to the network: the pixels that constitute an image, for example. The output layer is responsible for connecting back to the real world problem at hand. If, for example, the problem is that of determining the class of an input image, each output neuron will represent the probability of the image belonging to a given class, hence there will be as many output neurons as there are classes of images.

Each circle in Figure 2.7 represents a neuron, and the arrows between them are the connections, each of them with an associated weight. Each neuron is connected to all the neurons of the previous layer and to all the neurons of the next layer.

The output value of neuron i in layer $l > 0$, a_i^l, is given by equation (2.5).

$$a_i^l = f(z_i^l), \text{ with } \qquad z_i^l = b_i^l + \sum_{j \epsilon \text{previous layer}} w_{ij}^l a_j^{l-1} \tag{2.5}$$

z_i^l is the intermediate output, b_i^l the bias, w_{ij}^l the weight connecting neuron j of the previous layer with this neuron and f is a non-linear activation function. For the input layer, a_i^0 is simply equal to the the content at index i of the input vector.

The ultimate goal of a NN is to receive an input (an image, for example) and give a meaningful output (predict if the image is of a dog, a person, etc., in an image classification problem). This process is called **inference**. However, in order for the network to be able to make correct predictions, it must first learn from experience, just as humans do, in a process called **training**.

2.2.2 NN Training Procedure

Training a NN consists of updating its parameters (weights) in order to obtain a model that can address as accurately as possible the problem at hand. During the training procedure, the measure that is used to evaluate whether the model is accurate is a **loss function** (L). This function compares the predicted value, $\hat{y}$, and the correct label, y_{true}, penalizing wrong predictions, $L = f(\hat{y}, y_{true})$.

Since the predicted value is a function of all the weights of the network, the loss will also be a function of these weights: $L = g(w_{11}^1, w_{12}^1, ..., w_{ij-1}^l, w_{ij}^l, y_{true})$. Therefore, the goal of the training procedure is to find the weights that minimize this loss function.

Among the several techniques for NN training, the **gradient descent** algorithm is the most popular [46]. The concept of gradient descent is illustrated in Figure 2.8 for the simple case of a single variable function where, for each step, the weight is updated in the direction opposite to that of the gradient.

Mathematically, this can be formulated as

$$w(t+1) = w(t) - \eta \frac{\partial L}{\partial w}\bigg|_{w(t)}, \tag{2.6}$$

where t is the step being considered and η is a hyperparameter called the **learning rate**, used to adjust the magnitude of the weight updates.

Gradient descent can be applied in three different ways, corresponding to three different variants of the algorithm [48]:

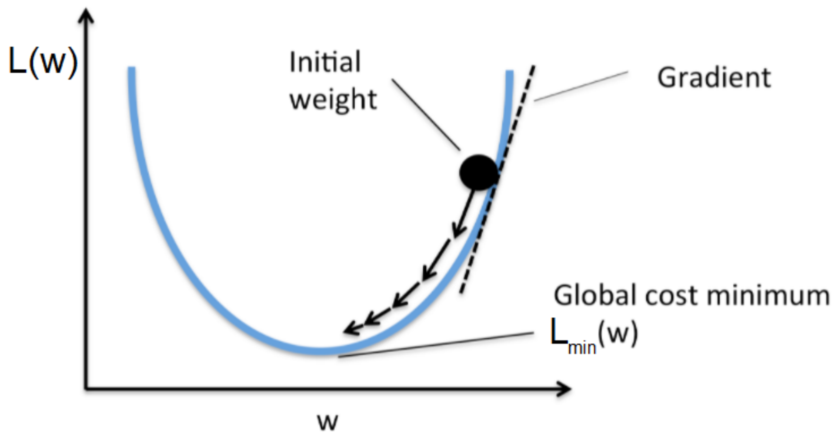


Figure 2.8: Gradient Descent illustrated for a single variable function, obtained from [47]

- Batch gradient descent: The loss is averaged over the entire training dataset before performing any update to the weights.

- Stochastic (or online) gradient descent (SGD): weights get updated after calculating the loss for each sample.

- Mini-batch gradient descent: The loss is averaged over a subset of the training dataset, with the batch size being a tunable hyperparameter.

Batch gradient descent has the disadvantage of needing the whole dataset to be evaluated before performing an update, which constitutes a problem if the dataset is too big to fit in memory and does not allow an *online* update, that is, update the model with the arrival of a new sample. Stochastic gradient descent circumvents these problems, but since updates are done with high frequency the variance is high, which can lead to unstable convergence. A compromise solution, mini-batch gradient descent, combines the advantages of both and is usually the adopted technique [48].

In order to calculate the derivative of the loss with respect to a certain weight, the chain rule is usually used, following equation (2.5) for the activation of each neuron. For the weights connecting to last layer neurons, it is computed as:

$$\frac{\partial L}{\partial w_{ij}^l} = \frac{\partial L}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{ij}^l} = \frac{\partial L}{\partial a_j^l} f'(z_j^l) a_i^{l-1}. \tag{2.7}$$

For weights in the middle layers, the influence on the loss is felt through different paths in the network. Taking as an example Figure 2.7, the weight w_{11}^2 influences the first neuron of the second hidden layer, which in turn affects both output neurons, effectively affecting the loss in two ways. This way, the derivative of the loss with respect to this weight will be the sum of these two paths. The general formulation is, then:

$$\begin{cases} \frac{\partial L}{\partial w_{ij}^l} = \frac{\partial L}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{ij}^l} \\ \frac{\partial L}{\partial a_j^l} = \sum\limits_{m \in \text{next layer}} \frac{\partial L}{\partial a_m^{l+1}}\frac{\partial a_m^{l+1}}{\partial z_m^{l+1}}\frac{\partial z_m^{l+1}}{\partial a_j^l} \end{cases} \Rightarrow \begin{cases} \frac{\partial L}{\partial w_{ij}^l} = \frac{\partial L}{\partial a_j^l} f'(z_j^l) a_i^{l-1} \\ \frac{\partial L}{\partial a_j^l} = \sum\limits_{m \in \text{next layer}} \frac{\partial L}{\partial a_m^{l+1}} f'(z_m^{l+1}) w_{mj}^{l+1}. \end{cases} \tag{2.8}$$

From equation (2.8), the sum used to calculate $\frac{\partial L}{\partial a_j^l}$ refers to the next layer, which means that to calculate the gradients it is more practical to do the computation in reverse: start from the loss in the last layer, going backwards up to the input layer. As a result, this algorithm is know as **backpropagation**, the most commonly used to train NN [49].

In summary, the training of a neural network has 4 main phases:

- **forward pass**: feeding an input to the network and propagating forward to obtain an output;
- **loss calculation**: with the predicted output and the true value, calculating the loss according to the chosen loss function;
- **backward pass**: calculating the gradients with respect to the weights, starting from the last layer up to the beginning;
- **weight update**: following the principle of gradient descent, update the weights to minimize the loss function.

2.2.3 Activation Functions

As mentioned in section 2.2.1 and expressed by function f in equation (2.5), each neuron goes through a non-linear activation function. Without this function, the equation for each neuron would only be the last part of equation (2.5), which is a linear equation, meaning that the network would not be able to model non-linear data.

Drawing inspiration from the human brain, the first intuition was to use a step function that is zero up to a certain threshold and 1 after it, representing the ”firing” of the biological neurons. However, given the non-continuous and non-differentiable nature of this function, it would make it unfit for backpropagation, since its derivative is needed in the first part of equation (2.8) [50]. Nevertheless, the idea of ”firing” of neurons was not discarded, and the most common activation functions approximate the behavior of the step function.

Even though many activation functions exist and current research is still very active in putting forward alternatives [51], some of the most commonly used include **Sigmoid**, **TanH** and **ReLU** [52].

Sigmoid is a monotonic S-shaped curve that maps any input into the range $[0, 1]$. This is particularly useful in cases where the output should be a probability. One of its downsides is the smooth curve, which implies small derivatives when values are far from 0. This leads to the vanishing gradients problem, which consists on gradients being too small when applying the chain rule described in equation (2.8), leading to small or even no weight updates and the training getting stuck. Equation (2.9) defines the sigmoid function and its derivative. One interesting property is the fact that the derivative can be expressed in terms of the function evaluated at the same point, which can be useful to reduce calculations during the backward pass. Figure 2.9 is a graphic representation of the sigmoid function.

$$\begin{cases} \text{sigmoid}(x) = \sigma(x) = \frac{1}{1+e^{-x}} \\ \sigma'(x) = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}}(1 - \frac{1}{1+e^{-x}}) = \sigma(x)(1-\sigma(x)) \end{cases} \tag{2.9}$$

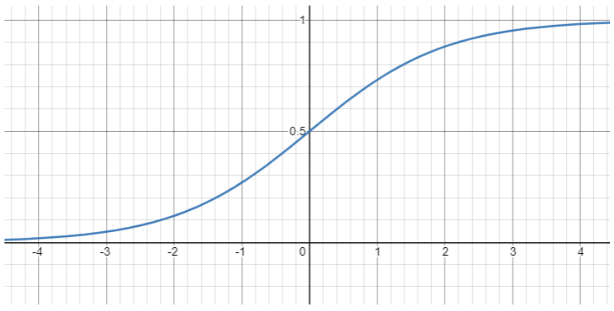


Figure 2.9: Graphic representation of the sigmoid function.

TanH is the hyperbolic tangent function, similar to sigmoid in shape but with output range $[-1, 1]$. One advantage when compared to sigmoid is that negative inputs are mapped to negative outputs, so the sign of the input is not lost in the activation. Equation (2.10) defines the tanh function and its derivative. Figure 2.10 is a graphic representation of the tanh function.

$$\begin{cases} \text{tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1 \\ \text{tanh}'(x) = \frac{(e^x+e^{-x})^2 - (e^x - e^{-x})^2}{(e^x+e^{-x})^2} = 1 - \frac{(e^x-e^{-x})^2}{(e^x+e^{-x})^2} = (1 - \text{tanh}^2(x)) \end{cases} \tag{2.10}$$

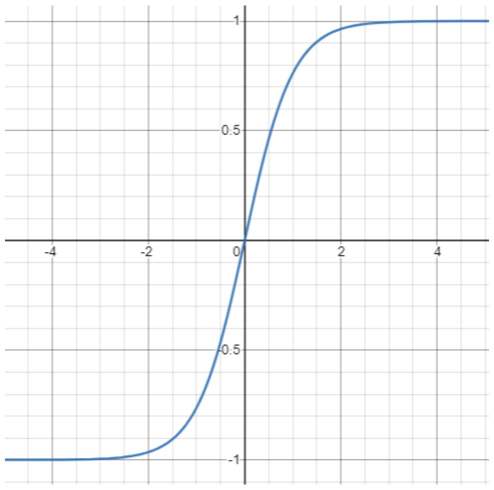
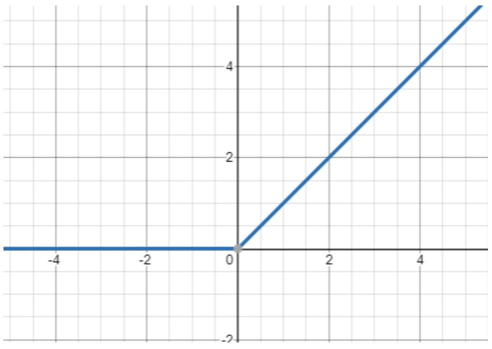


Figure 2.10: Graphic representation of the TanH function.

ReLU (Rectified Linear Unit) is one of the most popular activation functions, albeit its simplicity. It consists in setting negative values to 0 and leaving positive ones unchanged. Even though this function is non-differentiable at 0, its derivate is usually extended as 0 when evaluated at 0, which is trivial to implement in software. This function is much less expensive to compute than both sigmoid and TanH, and does not have the vanishing gradient problem for positive input values. For negative inputs, however, the gradient is always 0, which prevents the weights associated with these neurons from being updated. Some variants, such as Leaky ReLU, address these problem by having a slight slope for negative values. Equation (2.11) defines the ReLU function and its derivative. Figure 2.11 is a graphic representation of the ReLU function.

$$\begin{cases} \text{ReLU}(x) = \max(0, x) \\ \text{ReLU}'(x) = \begin{cases} 0, & \text{if } x \leq 0 \\ 1, & \text{if } x > 0 \end{cases} \end{cases} \tag{2.11}$$


(a) ReLU

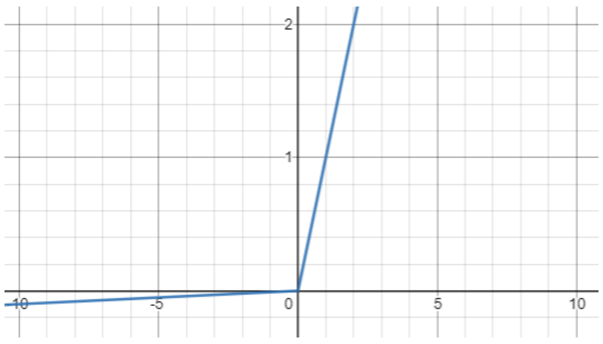
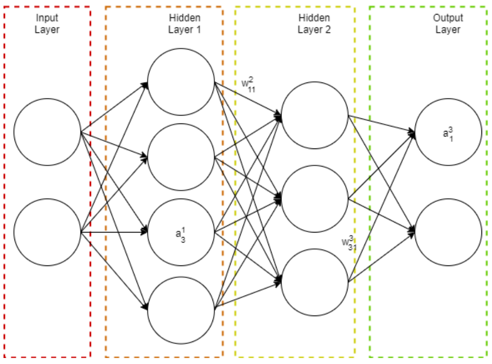

(b) Leaky ReLU

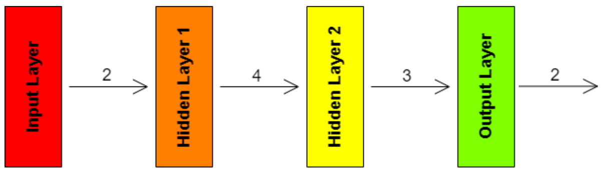
Figure 2.11: Graphic representation of the ReLU function and one possible variant

2.2.4 Types of Layers

The representation of a Neural Network presented in Figure 2.7 is somewhat misleading, in that the layers are associated to the neurons, whereas in fact layers are comprised of the weights and the neurons together. This way, a more accurate representation is presented in Figure 2.12, alongside a corresponding block diagram used since, for more complex networks, connections between neurons quickly make the image two dense.



(a) Neural Network example



(b) Neural Network block diagram

Figure 2.12: Neural Network example along with its corresponding block diagram

It is also important to note that the input layer represents the input data fed to the network, which means that in the case of image classification, if the input is not flattened, instead of having a single dimension it could be 2-D or even 3-D, in the case of a coloured image.

There are many different types of layers that can be used when building a neural network, each with a different purpose to better address the problem at hand. For image classification problems, **Linear**, **Convolutional** and **Pooling** layers are the most commonly used.

Linear Layer

The linear layer is the one generally represented in Figure 2.7, where all neurons of the previous layer are connected with the neurons of the next layer, hence its alternative name of **fully connected** layer. Saying that the output of this layer is of size m, means that each neuron of the previous layer has m weights associated with it. This way, if the number of neurons of the previous layer is n, a linear layer can be mathematically represented with a matrix of size $m \times n$ that will multiply the input, and a bias vector of size m to sum, generalizing equation (2.5) to compute all neurons at once:

$$\text{output}_{m\times 1} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & \ddots & & w_{2n} \\ \vdots & & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{bmatrix} \begin{bmatrix} in_1 \\ in_2 \\ \vdots \\ in_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \tag{2.12}$$

For backpropagation, equation (2.8) can be extended to compute all weight gradients at once with matrix multiplication, in a similar fashion to what is done in equation (2.12) for the forward pass.

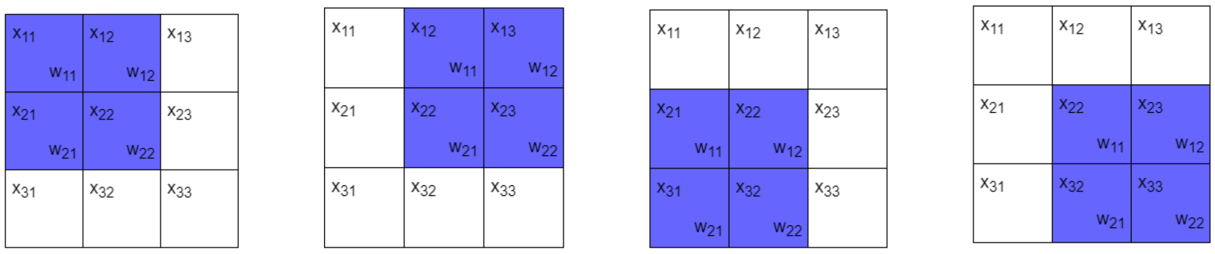
Convolutional Layer

Convolutional layers are responsible for the name of Convolutional Neural Networks (CNNs), the state-of-the-art networks in image classification problems [53]. These layers perform a convolution operation on the 2-D (or 3-D) data, that consists in sliding a filter (called **kernel**) through the image in order to outline certain features. One of the main advantages over a linear layer is that the images don't need to be flattened, which means that spatial features can be detected [54].

Mathematically, a convolution operation would require the kernel to be flipped before it is slid over the image. However, since weights are randomly initialized, sliding the kernel directly produces the desired results. This way, what in the DL community is called convolution does not correspond to the mathematical operation of convolution, but to cross-correlation. In this frame, convolution is defined as:

$$y_{i_2 i_1} = \sum_{\substack{k_2=1 \\ j_2=k_2+i_2-1}}^{H_w} \sum_{\substack{k_1=1 \\ j_1=k_1+i_1-1}}^{W_w} x_{j_2 j_1} w_{k_2 k_1}, \tag{2.13}$$

where $y_{i_2 i_1}$ is the convolution output at index (i_2, i_1), H_w is the height of the kernel, W_w is its width, $x_{j_2 j_1}$ is the input at index (j_2, j_1) and $w_{k_2 k_1}$ is the kernel weight at index (k_2, k_1). Graphically, the convolution operation can be visualized in Figure 2.13, for an example where the input is of size 3×3 and the kernel is of size 2×2.



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

Figure 2.13: Graphic representation of kernel and input overlaps to compute convolution.

The representation presented in Figure 2.13 would correspond to 4 matrix multiplications, one for each position of the kernel over the input matrix. Nonetheless, by flattening the input and considering a sparse matrix of the weights, the operation can be computed with only one matrix multiplication, as represented in equation (2.14).

$$\begin{bmatrix} w_{11} & w_{12} & 0 & w_{21} & w_{22} & 0 & 0 & 0 & 0 \\ 0 & w_{11} & w_{12} & 0 & w_{21} & w_{22} & 0 & 0 & 0 \\ 0 & 0 & 0 & w_{11} & w_{12} & 0 & w_{21} & w_{22} & 0 \\ 0 & 0 & 0 & 0 & w_{11} & w_{12} & 0 & w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} x_{11} \\ x_{12} \\ x_{13} \\ x_{21} \\ x_{22} \\ x_{23} \\ x_{31} \\ x_{32} \\ x_{33} \end{bmatrix} = \begin{bmatrix} x_{11}w_{11} + x_{12}w_{12} + x_{21}w_{21} + x_{22}w_{22} \\ x_{12}w_{11} + x_{13}w_{12} + x_{22}w_{21} + x_{23}w_{22} \\ x_{21}w_{11} + x_{22}w_{12} + x_{31}w_{21} + x_{32}w_{22} \\ x_{22}w_{11} + x_{23}w_{12} + x_{32}w_{21} + x_{32}w_{22} \end{bmatrix} \tag{2.14}$$

Another advantage of having the weights in a single matrix is that the forward propagation is exactly the same operation as for the linear layer, which means that the weight gradient for backward propagation can also be computed through equation (2.8).

Besides the kernel size, there are 3 more hyperparameters that characterize a convolutional layer: **stride**, **padding** and **dilation**. Stride represents the size of the step when the kernel moves over the input. In Figure 2.13, the kernel is moving with the default stride of 1. Increasing stride represents a downsampling, which can be useful to decrease computation if the features to be detected allow for smaller resolution [55].

Without any padding, the borders of the image have less interaction with the kernel than the rest of the image, since only the left part of the filter is ever at the edge of the image. By starting the kernel slide outside the image borders, the borders have the same interaction with the kernel as any other part of the image. Padding represents the number of elements added to the border of an image, which represents starting the kernel outside the image. The default padding value is 0. Equation (2.15) represents an input that is padded by 1.

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \xrightarrow{\text{padding = 1}} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & x_{11} & x_{12} & x_{13} & 0 \\ 0 & x_{21} & x_{22} & x_{23} & 0 \\ 0 & x_{31} & x_{32} & x_{33} & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.15}$$

Dilating a kernel is the incorporation of 0s between elements of the kernel. This reduces the size of the output of the convolution operation, but depending on the application might come at the cost of loss of accuracy [56]. The value of the dilation corresponds to the spacing between elements of the kernel, which means that the default value is 1. Graphically, a dilation of 2 applied to a 2×2 kernel is represented in Equation (2.16).

$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \xrightarrow{\text{dilation = 2}} \begin{bmatrix} w_{11} & 0 & w_{12} \\ 0 & 0 & 0 \\ w_{21} & 0 & w_{22} \end{bmatrix} \tag{2.16}$$

Taking all the presented hyperparameters into account, the size of the output after a convolution operation is calculated with equation (2.17). This size is computed for each dimension, since all hyperparameters might have different values for the horizontal and vertical dimensions.

$$\text{output size} = \frac{\text{input size} + 2 \times \text{padding} - \text{dilation} \times (\text{kernel size} - 1) - 1}{\text{stride}} + 1 \tag{2.17}$$

Pooling Layer

The output of a convolutional layer is a feature map, highlighting features of the input image. Downsampling this output leads to the network being more robust to changes in the position of the features in the input and consequently enables more general predictions [57]. This downsampling can be achieved by increasing the stride and/or dilation in the convolutional layer, but it is generally more effective to have a pooling layer after the convolutional layer in the network architecture [57].

The pooling layer does not have weights associated with it. It consists of a kernel that slides over the input image, performing a pooling operation for each overlap. The pooling layer has the same hyperparameters as the convolutional: kernel size, stride, padding and dilation; the only difference is that the default value for the stride is the kernel size. The most common operations are:

- **average pooling**: the average value of the inputs within each overlap is calculated;
- **maximum pooling**: the maximum value of the inputs within each overlap is selected.

An average pooling layer can be mathematically expressed as:

$$y_{i_2 i_1} = \frac{1}{H_w W_w} \sum_{j_2 = i_2 \times \text{stride}}^{H_w + i_2 \times \text{stride}} \sum_{j_1 = i_1 \times \text{stride}}^{W_w + i_1 \times \text{stride}} x_{j_2 j_1}. \tag{2.18}$$

Since there are neither weights nor activation function, the derivative for backpropagation in an average pooling layer simplifies to:

$$\frac{\partial L}{\partial a_j^{l-1}} = \frac{\partial L}{\partial a_i^l}\frac{\partial a_i^l}{\partial z_i^l}\frac{\partial z_i^l}{\partial a_j^{l-1}} = \frac{\partial L}{\partial a_i^l}\frac{1}{H_w W_w} \tag{2.19}$$

Graphically, an example of a 4×4 sized input with a kernel of size 2×2 is shown in equation (2.23).

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{bmatrix} \xrightarrow{\text{average pooling}} \begin{bmatrix} \frac{x_{11}+x_{12}+x_{21}+x_{22}}{4} & \frac{x_{13}+x_{14}+x_{23}+x_{24}}{4} \\ \frac{x_{31}+x_{32}+x_{41}+x_{42}}{4} & \frac{x_{33}+x_{34}+x_{43}+x_{44}}{4} \end{bmatrix} \tag{2.20}$$

As for the maximum pooling layer, it can be mathematically expressed as:

$$y_{i_2 i_1} = \max_{j_2, j_1 \in \text{kernel overlap}} \{x_{j_2 j_1}\}. \tag{2.21}$$

The derivative of a maximum pooling layer is 0 whenever the input was not the maximum and 1 if it was, so for backpropagation it can be expressed as:

$$\frac{\partial L}{\partial a_j^{l-1}} = \frac{\partial L}{\partial a_i^l}\frac{\partial a_i^l}{\partial z_i^l}\frac{\partial z_i^l}{\partial a_j^{l-1}} = \begin{cases} \frac{\partial L}{\partial a_i^l} & \text{if } j \text{ is max of overlap,} \\ 0 & \text{otherwise.} \end{cases} \tag{2.22}$$

Graphically, an example of a 4×4 sized input with a kernel of size 2×2 is shown in equation (2.23).

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{bmatrix} \xrightarrow{\text{maximum pooling}} \begin{bmatrix} \max\{x_{11}, x_{12}, x_{21}, x_{22}\} & \max\{x_{13}, x_{14}, x_{23}, x_{24}\} \\ \max\{x_{31}, x_{32}, x_{41}, x_{42}\} & \max\{x_{33}, x_{34}, x_{43}, x_{44}\} \end{bmatrix} \tag{2.23}$$

Droupout

One of the main concerns when training a network is that it generalizes well for unseen data. The **overfitting** problem - when a network fits too well the training data and hence performs poorly on new data - is important to take into account, especially with deep networks. From the notion that combining the predictions of different models yields better results in testing, dropout layers aim at increasing redundancy within a network, simulating the effect of having several networks making a prediction [58].

A dropout layer is characterized by a single parameter, p, that represents the probability of a given node being dropped. This layer receives a tensor and outputs the same tensor with some values set to 0 and the remaining scaled by $\frac{1}{1-p}$ [59]. It can be applied after the input layer and/or in any hidden layer. During inference, this layer is simply the identity function, as the network has already been trained and the goal is to make a prediction using the whole network.

The derivative for backpropagation follows the same principle: 0 for nodes zeroed out during the forward pass and the scale factor $\frac{1}{1-p}$ otherwise. In each forward pass the nodes to be zeroed out change, forcing different subsets of the network to train.

2.2.5 Loss functions

As mentioned in section 2.2.2, the metric used during the training of a neural network is a loss function, that measures the distance between a prediction and the expected target. Several loss functions have been proposed, their choice depends on the type of problem being handled. For regression problems, the most popular function is the **Mean Squared Error**, whereas for multi class classification problems the most commonly used is **Cross Entropy Loss** [60].

Mean Squared Error (MSE) consists in calculating the mean of the squared difference between the output and the target, greatly penalizing outliers, as the error is squared. Formally, it can be expressed by equation:

$$L(\hat{y}, y_{true}) = \frac{\sum\limits_{i \in \text{last layer}} (\hat{y}_i - y_{true})^2}{\text{\# output neurons}}, \tag{2.24}$$

where $\hat{y_i}$ is the predicted value at neuron i and y_{true} is the expected value at neuron i. Its gradient, used for backpropagation, is proportional to the error:

$$\frac{\partial L}{\partial y_i} = 2(\hat{y} - y_{true}). \tag{2.25}$$

Cross Entropy Loss, used for classification problems, combines the softmax function with the Negative Log Likelihood function (NLL). The softmax function can be regarded as the predicted probability for a given class, since its output is between 0 and 1 and the sum for all classes is equal to 1. This way, NLL can be seen as the log probability of the target class. Mathematically, Cross Entropy Loss can be expressed as:

$$\begin{cases} S(\hat{y_i}) = \frac{\exp(y_i)}{\sum\limits_{j \in \text{last layer}} \exp(y_j)} \\ L(\hat{y}, \text{target}) = -\log(S(\hat{y}_{\text{target}})), \end{cases} \tag{2.26}$$

where target represents the index of the expected class. Its gradient can be expressed solely through the value of the function, which reduces the amount of computation during the backward pass:

$$\frac{\partial L}{\partial y_i} = -\frac{1}{S(\hat{y}_{\text{target}})} \frac{\partial S(\hat{y}_{\text{target}})}{\partial y_i} = \begin{cases} S(\hat{y}_{\text{target}}) - 1 & \text{for } i = \text{target}, \\ S(\hat{y_i}) & \text{otherwise}. \end{cases} \tag{2.27}$$

2.2.6 Optimizers

After the forward pass, the loss calculation and the backward pass, the weights are updated with the goal of minimizing the loss function in the next pass. Gradient descent is the algorithm usually used to perform the weight updates. Equation (2.6) represents the most basic optimizer, that depending on the amount of samples considered will be batch, mini-batch or stochastic gradient descent (SGD), as exposed in section 2.2.2. This algorithm has some shortcomings, namely [48]:

- The learning rate is the same for all model weights;
- Slowness of conversion;
- The algorithm might get trapped in a local minimum, leading to no more weight updates and a suboptimal solution.

To improve performance regarding the last two challenges, a momentum term (v) can be added to the update. It is based on an exponentially weighted average, where the current gradient only constitutes a fraction of the total term to be added to the weight. This is formally expressed as:

$$\begin{cases} v(t+1) = \gamma v(t) + \eta \frac{\partial L}{\partial w}\big|_{w(t)} \\ w(t+1) = w(t) - v(t+1), \end{cases} \tag{2.28}$$

where γ is the momentum rate, η is the learning rate and $w(t)$ is the weight at time step t.

Nonetheless, even with the introduction of the momentum term, the first shortcoming was not addressed: not only is the learning rate still the same for all model weights but the momentum rate is also unique. Adaptive algorithms address this issue, by having hyperparameters that are adaptable to each weight. The most commonly used is Adam (Adaptive Moment Estimation) [48]. It introduces a second momentum term, proportional to the gradient squared, that will be used to have an adaptive learning rate. The two momentum terms, mean (m) and uncentered variance (v), are computed by:

$$\begin{cases} m(t+1) = \beta_1 m(t) + (1-\beta_1)\frac{\partial L}{\partial w}\big|_{w(t)} \\ v(t+1) = \beta_2 v(t) + (1-\beta_2)(\frac{\partial L}{\partial w}\big|_{w(t)})^2 \end{cases} \tag{2.29}$$

Since these are exponentially weighted averages (with β_1 and β_2 determining over how many points the average is taken), there is a bias towards 0 in the first iterations, so a bias correction term is added:

$$\begin{cases} \hat{m}(t+1) = \frac{m(t+1)}{1-(\beta_1)^{t+1}} \\ \hat{v}(t+1) = \frac{v(t+1)}{1-(\beta_2)^{t+1}}. \end{cases} \tag{2.30}$$

Finally, the weight updates are performed with the adaptive learning rate through:

$$w(t+1) = w(t) - \frac{\eta}{\sqrt{\hat{v}(t+1)+\epsilon}}\hat{m}(t+1), \tag{2.31}$$

where ϵ prevents division by 0. Upon the introduction of Adam, the proposed values for the hyperparameters were $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$ [61].

Given that it can be customized through the hyperparameters and that it has two momentum terms that induce faster convergence, Adam is used in most state-of-the-art NNs [48]. However, despite its slower convergence, SGD has been found to generalize better, so techniques of switching from Adam to SGD at later stages of training have been explored [62].

2.2.7 Reference Datasets

The previous sections described the main building blocks to construct a neural network, in particular the layers that constitute Convolutional Neural Networks (CNNs). The possible architectures are endless, by choosing which layers to use, their sizes, the tuning of hyperparameters and the depth of the network. There are competitions for a given DL problem (classification of coloured images into different classes and subclasses, for instance [63]) where novel architectures are proposed, and winner networks establish themselves as reference networks.

This section presents some of the reference datasets used in these competitions, which are composed by large sets of labeled images so that different network architectures can be compared on the same task and input.

MNIST

MNIST is a dataset that consists of 28×28 pixel images of handwritten digits in grayscale, divided into 10 classes (a class per digit) [64]. It is comprised of 60000 images used for training and 10000 images for validation. It was assembled in 1998 from the combination of two existing datasets of handwritten digits. It is usually the first dataset on which a model is tested, given its simplicity.

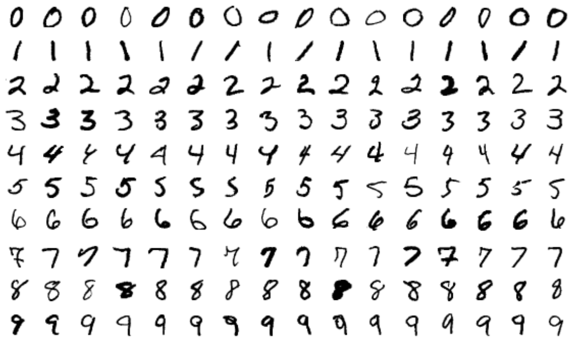


Figure 2.14: A few samples from the MNIST dataset, with different classes on different rows, obtained from [65].

Cifar-10 and Cifar-100

Cifar-10 is a dataset that consists of 32×32 pixel coloured images, which means that each image has 3 channels (red, green and blue). It is divided into 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. It is comprised of 50000 images for training and 10000 for validation. Cifar-100 has the same types of images and the same cardinality, but images are classified into 100 classes that are grouped into 20 superclasses, meaning that the labeling of an image involves

2 classifications. Given that classes are more complex than digits and the images are coloured, these datasets are harder to classify than MNIST.

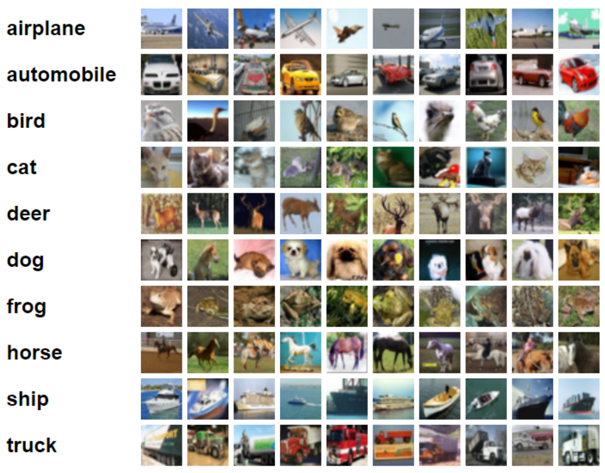


Figure 2.15: A few samples from the Cifar-10 dataset, with different classes on different rows, obtained from [66]

ImageNet

ImageNet is a dataset being built with the aim of being the most comprehensible set of labeled images. Currently it consists of over 14 million images divided into 21841 classes, but the goal is to expand further to tens of millions of images divided into one hundred thousand classes [63]. The images are coloured (3 channels) and have varying dimensions. This dataset is the base for the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), held every year from 2010 to 2017 [63], where different network architectures would compete to classify images from the dataset with the greatest possible accuracy.

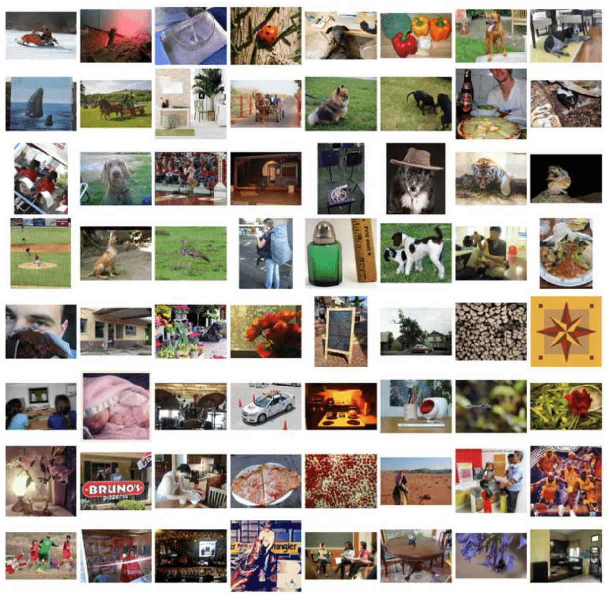


Figure 2.16: A few samples from the ImageNet dataset, obtained from [63]

Several networks have been proposed to address the datasets presented, namely during the 8 years of the ILSVRC. Some of the most important are presented next, with a description of their architecture along with the reasons that made them become benchmark models. These are presented in increasing order of complexity.

2.2.8 Benchmark Models

This section presents some of the CNNs that have won competitions on the datasets presented in the previous section. These models have become benchmarks for image classification tasks, being used to compare and validate results on these tasks.

LeNet-5

LeNet-5 is a CNN proposed in 1998, the first to achieve an accuracy above 99% in handwritten digit recognition [67]. It is a common entry point for CNNs, given its relatively simple architecture and the many tutorials available for implementation in different DL platforms. It is comprised of 7 layers: 3 convolutional, 2 average pooling and 2 fully connected, adding to a total of about 60 thousand trainable parameters. A block diagram representing its architecture is presented in Figure 2.17, where the input is a 32×32 image (one from the MNIST dataset, for instance), and the output is the probabilities of each of 10 classes.

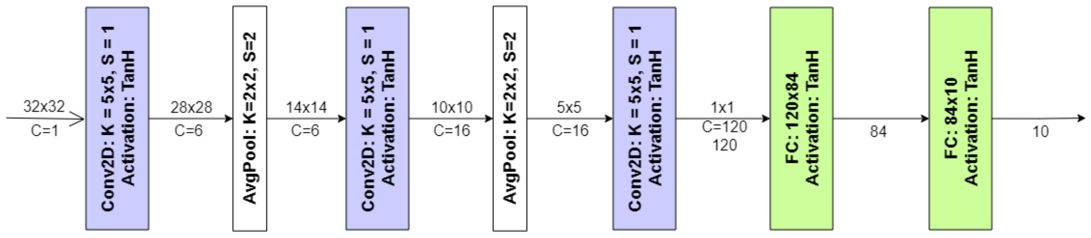


Figure 2.17: Block Diagram representing the architecture of the LeNet-5. K stands for kernel, S for stride and FC for fully connected.

CifarNet

CifarNet was designed to classify the previously presented Cifar-10 dataset with a relatively small network structure. Its architecture is similar to LeNet-5, the main differences being the introduction of padding in the convolutional layers and using ReLU instead of TanH for activations. It does not have a fixed structure, but it is characterized by having 3 convolutional layers and 1 to 3 fully connected layers. Figure 2.18 presents a block diagram of CifarNet, based on the one used in [68].

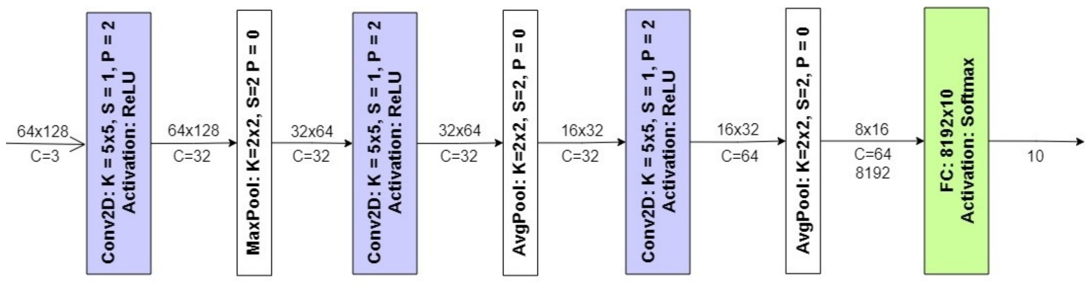

64x128
C=3
Conv2D: K = 5x5, S = 1, P = 2
Activation: ReLU
64x128
C=32
MaxPool: K=2x2, S=2 P = 0
32x64
C=32
Conv2D: K = 5x5, S = 1, P = 2
Activation: ReLU
32x64
C=32
AvgPool: K=2x2, S=2, P = 0
16x32
C=32
Conv2D: K = 5x5, S = 1, P = 2
Activation: ReLU
16x32
C=64
AvgPool: K=2x2, S=2, P = 0
8x16
C=64
8192
FC: 8192x10
Activation: Softmax
10

Figure 2.18: Block Diagram representing the architecture of the CifarNet. K stands for kernel, S for stride, P for padding and FC for fully connected.

AlexNet

AlexNet was proposed in 2012, winning the ILSVRC with a top-5 test error rate of 15.3% (the correct class is one of the 5 most probable in the model's prediction), over 10% better than the second place [58]. It is a deep network of 13 layers, with a total of 650 thousand neurons and 60 million trainable parameters, making it one of the largest at that time. AlexNet was among the first to leverage the use of GPUs to speed-up training, contributing to the generalization of GPUs in DL applications [69]. It introduced dropout layers and overlapped pooling to reduce overfitting and ReLU activation to increase training speed [58].

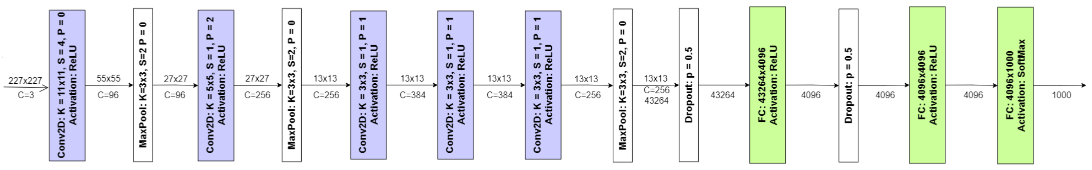


Figure 2.19: Block Diagram representing the architecture of the AlexNet. K stands for kernel, S for stride, P for padding, p for probability and FC for fully connected.

ResNet

Following AlexNet, even deeper architectures started to be explored, and the ResNet class of networks, proposed in 2015, explored up to 1000 layers [70]. This increase in depth led to better performance in more complex task, such as object detection, where ResNet won the 2015 Common Objects in Context Competition (COCO). An architecture with 152 layers also won the ILSVRC in 2015.

Increasing the depth of networks brought other challenges, such as the increase in training time, the problem of vanishing gradients and the degradation of training accuracy [70]. The vanishing gradient problem had already been addressed, but ResNet introduced the concept of residual learning - connections that are skipped, the input is passed directly to the next layer - that not only improved training accuracy but also reduced training time for deeper networks, as some computations are skipped.

2.2.9 Deep Learning Frameworks

In order to build, train and perform inference with NNs, one can either implement all the mathematical operations for the different layers, derivative calculations for backpropagation, etc. from scratch, or leverage existing frameworks that provide higher level APIs that address these operations. Some of the most popular frameworks are PyTorch, TensorFlow, Keras (built on top of TensorFlow), Caffe and CNTK.

Among these, PyTorch [8] and TensorFlow [9] are the most established [10]. The first is developed by Facebook and the second by Google, but both of them are open source with very active communities. Google's TensorFlow was released earlier with a stable API. It provided solutions both for server and mobile and thus became dominant in production. However, given PyTorch's shallower learning curve and the easiness to experiment and debug novel architectures, researchers have been switching to PyTorch, and it has surpassed TensorFlow in popularity, as shown in Figure 2.20.

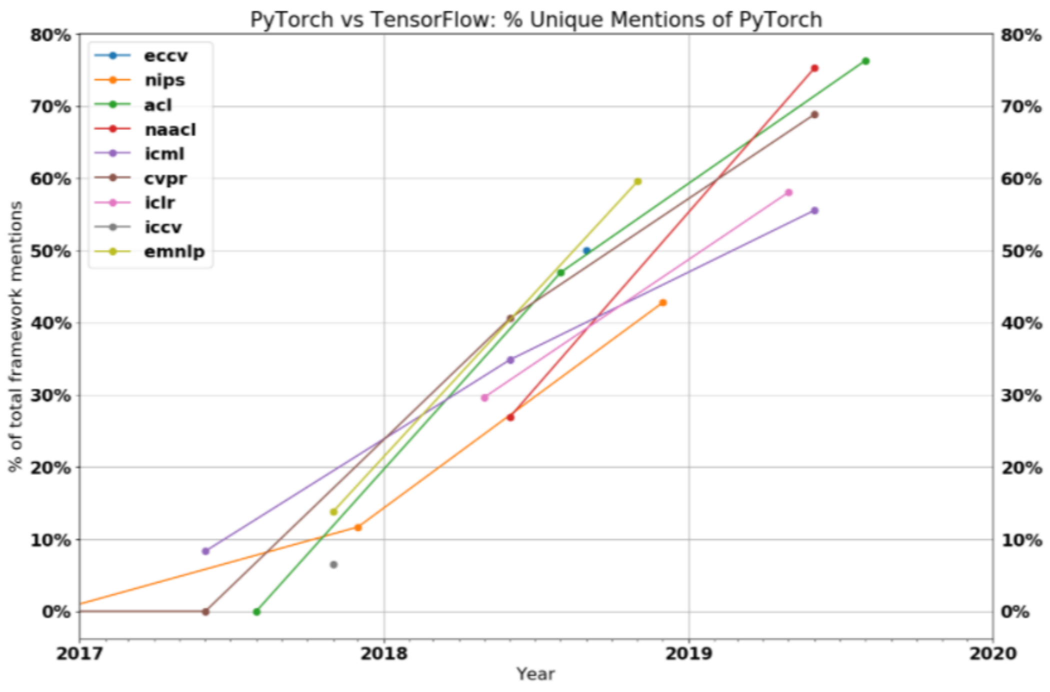


Figure 2.20: Unique mentions of PyTorch in DL articles that either mention PyTorch or TensorFlow, obtained from [10].

Both frameworks offer similar functionalities, namely high level Python APIs for all NN layers and associated functions, along with automatic differentiation, responsible for calculating the gradients needed for backpropagation. Even though the main API is in Python, PyTorch also has a C++ API and TensorFlow includes an API in C. These are used mainly for speed, avoiding the need of the Python interpreter, but they do not offer all the functionalities available in the Python API. Nonetheless, given their open source nature, in both frameworks can be extended with user designed functions, either purely written in Python or through C/C++ extensions that are translated to Python.

2.3 Related Work

The high computational and memory requirements of NN training and inference have led to an increasing interest in using low-precision arithmetic in DL problems. This is observed in the available tools for model design with smaller sized (less than 32 bits) floating point numbers, such as the hardware support for half floating point (16 bits) and BFloat16 in recent NVIDIA's GPUs and Google's Tensor Processing Units.

Upon the introduction of Posits, with the claim that with less bits they could provide similar accuracy to IEEE 754 floating-points in DL applications [15], several experiments with different posit configurations in NN inference and training have been conducted. However, in order to perform these experiments, it is necessary to develop frameworks that support DL operations with Posits, as existing frameworks do not have this support.

The first work to address inference with Posits [14] did not actually support any operations with Posits. It was limited to converting floating-point values to low-precision Posits and back to floats for computation, in order to evaluate if the smaller precision of Posits resulted in a loss of accuracy in the network's predictions. Subsequent works [11, 16, 17] introduced hardware support for multiplication and addition with Posits, since these are the only operations needed for inference.

The first work to address training with Posits [20] used the PySigmoid library [41], mentioned in section 2.1.5 as the basis to simulate operations with Posits. Since it consisted on a simple Fully Convolutional Neural Network (FCNN), the operations were implemented from scratch, that is, without using any of the DL frameworks exposed in section 2.2.9.

Later, in [18, 71], CNN were trained for the first time using the posit format. However, there was a pre-training phase with floats and intermediate calculations were also performed with floats. This removed the need to build a framework to train with Posits, as PyTorch's functions were used on floats.

It was in [72] that a module to enable training with Posits was first introduced. It consisted in an extension of the quantization module QPytorch [73] to support Posits. This module is built on top of PyTorch, allowing for integration with the framework to be seamless. However, this is a quantization tool, meaning that all calculations are still performed with floats and only afterwards are the values quantized to Posit.

In [20], a framework to design and train NN with Posits was proposed. Built on top of Tensorflow, this framework provided all the functions needed to train and evaluate NN with Posits. However, given the out-of-tree nature of this framework with respect to TensorFlow, there is a significant overhead in maintaining it.

More recently, a framework called PositNN was introduced [19]. Based on PyTorch's C++ frontend, this framework implements the most common functions used to train and evaluate NNs. It provides the flexibility to train NNs with Posits of any configuration, and even change configuration for different steps of training.

Up until now, as far as the author knows, every researcher who wanted to study training of NNs with Posits has developed a framework from scratch. This means that models firstly developed in PyTorch have to be completely rewritten to make use of the new frameworks. In contrast, this work aims at providing a tool for researchers to build and test their models in PyTorch without the need to learn a new tool or build one from scratch. Moreover, with the implementation of the NN operators for Posits, this work provides the intermediate layer for hardware implementations of posit arithmetic to be integrated in PyTorch.

2.4 Summary

In this chapter, the background to this work was presented. First, the main computer number formats, with particular attention to posits and the software libraries available to emulate its operations. Then, the main DL concepts were briefly described: the process of training a NN, its main layers and functions, some of the reference datasets and benchmark models as well as the most popular frameworks to build models. The chapter finishes with an overview of the related work of using Posits for DL applications.

Chapter 3

PyTorch Framework API

Contents

As mentioned in section 2.2.9, PyTorch is currently one of the most popular frameworks to build DL models. Its Application Programming Interface (API) provides functions for the most common DL operations, while abstracting their implementation details from the end user. Developed and owned by Facebook, it is open-source, meaning that anyone can contribute. These contributions by the external community are not only accepted and encouraged by the maintainers at Facebook but they have also influenced major design decisions. [8]

This chapter starts by exposing PyTorch Tensors, as these are the core data structures used for all data manipulation. It follows with details on how the functions needed to train and evaluate a NN that were introduced in section 2.2 are exposed in PyTorch's API. Even though PyTorch has APIs based on other languages (C++ and Java), the Python interface is the one with more features and the focus of this work. For these reasons, this chapter is solely focused on the Python API.

3.1 Tensors as the base data structure

From the decription presented in section 2.2, it was observed that most operations either consist of matrix multiplications or can be represented in such a way. Hence, PyTorch uses Tensors (multi-dimensional arrays) to represent all data within NNs. In their words, "PyTorch is an optimized tensor library for deep learning" [59].

As an example, gray-scale images can be represented by a 2-D array, where each row and column position maps a pixel in a certain height and width position on the image. If the image is coloured, a new dimension representing each of the 3 channels is added. If images are grouped (within a batch, for example), an extra dimension is added, where the index denotes the image's position in the batch. A 4-D tensor is, thus, the adequate data structure to store a batch of coloured images.

Another example are the weights of a linear layer. Here, the input is a column tensor corresponding to the neurons of the previous layer with length m and the output is a column tensor with length n. The weights of the linear layer can then be represented by an $m \times n$ tensor. If there is a batch of inputs, an extra dimension is added whose index represents a certain input.

From these examples, one can see that multi-dimensional arrays are a flexible structure to store information concerning the computations executed within NNs. However, PyTorch's Tensor class is more powerful than a simple container to store data. It stores metadata that is relevant for NN design, namely:

- **dtype**: the type of the stored data (32-bit float, 16-bit integer, 64-bit complex, etc);
- **device**: whether the tensor is allocated on the CPU or on the GPU, and in which core;
- **layout**: How the tensor's data is stored in memory (rows concatenated contiguously, a sparse matrix representation, etc);
- **requires_grad**: if, when integrated in a NN, this tensor will need to have its gradient calculated for backpropagation.

From these, the last attribute is particularly important. It constitutes the main difference between Tensors dedicated to NNs and multi-dimensional arrays. If this attribute is set to *True*, PyTorch will automatically store the gradient of the loss with respect to this tensor. Moreover, if other tensors are constructed through mathematical operations on a tensor with *requires_grad=True*, these tensors will also keep track of their gradient. With this, end users do not need to compute gradients, everything is done in the background once the network's design is established.

Listing 3.1 shows an example of creating 2 tensors and performing a matrix multiplication. Since *tensor_1* has its *require_grad* attribute set to *True*, *tensor_3* keeps track of the operation from which it was originated, so that it knows how to calculate its gradient.

```
>>> import torch
>>>
>>> tensor_1 = torch.tensor([[1.2, 3.4], [3.1, 4.2]], dtype=torch.float64, device="cpu",
    requires_grad=True)
>>> print(tensor_1)
tensor([[1.2000, 3.4000],
        [3.1000, 4.2000]], dtype=torch.float64, requires_grad=True)
>>>
>>> tensor_2 = torch.ones((2,2), dtype=torch.float64)
>>> print(tensor_2)
tensor([[1., 1.],
        [1., 1.]], dtype=torch.float64)
>>>
>>> tensor_3 = torch.mm(tensor_1, tensor_2)
>>>
>>> print(tensor_3)
tensor([[4.6000, 4.6000],
        [7.3000, 7.3000]], dtype=torch.float64, grad_fn=<MmBackward0>)
```

Listing 3.1: Tensor creation and matrix multiplication example

The Tensor class also provides a suite of operators to facilitate the development. These include:

- creation operators (from existing data, filled with zeros, from another tensor, etc.);
- indexing, slicing and joining operators (concatenating tensors, splitting into chunks, reshaping, etc.);
- pointwise math operators (trignometric functions, exponential, logarithm, etc.);
- basic linear algebra operators (addition after element-wise multiplication of tensors, chained matrix multiplication, etc.).

3.1.1 Tensor Data Types

As mentioned in the previous section, tensors have a *dtype* property, that stores the data type of the tensor in question. The data types that PyTorch supports are:

- IEEE floating-point: 16-bit (half), 32-bit (float) and 64-bit (double);

- integers: 8-bit (char), 16-bit (short), 32-bit (int), 64-bit (long) and unsigned integer with 8 bits;
- complex: 16-bit (half), 32-bit (float) and 64-bit (double);
- brain floating point with 16 bits (bfloat16).

Some of these are not supported for some operators in the framework, namely complex and brain floating point numbers. Whenever the *dtype* property is not specified, the default data type is 32-bit floating point (float). With this, unless clearly specified otherwise, all computations in PyTorch are performed with floats.

```
>>> tensor_1 = torch.tensor([[1.2, 3.4], [3.1, 4.2]])
>>> print(tensor_1.dtype)
torch.float32
```

Listing 3.2: Declaring a tensor with the default *float* datatype

3.2 Neural Network Design and Training

PyTorch provides functions and operators that allow the concepts presented in section 2.2 to be used through a simple interface that hides the implementation details from the end user. This section provides a high level description of how these concepts are exposed in PyTorch's frontend. All of the mentioned classes and functions are under the *nn* module. This way, the two lines shown in listing 3.3 should come immediately before all the code examples throughout this chapter, to import the modules *torch* and *nn* . Lower level implementation details of these classes and functions are explored later, in Chapter 4.

```
>>> import torch
>>> from torch import nn
```

Listing 3.3: Importing of the *torch* and *nn* modules

3.2.1 Layers

Linear Layer

As it was previously described in section 2.2, a linear layer consists of a multiplication of an input by a matrix of weights and the addition of a bias: $y = Ax + b$. A is the matrix with the weights, x is the input, b is the bias and y is the output.

To declare a linear layer in PyTorch, it suffices to specify the dimension of the input and that of the output. Extra arguments can be passed, such as the device and the data type of the layer. Internally, it stores two variables: the matrix of weights and the vector of bias. These are learnable parameters, randomly initialized from a distribution in the range $[-\sqrt{k}, \sqrt{k}]$, where $k = \frac{1}{\text{dimension of the input}}$ [59].

Listing 3.4 has a minimal example of declaring a linear layer and applying it to an input. In this example, a batch of 32 inputs is passed to the linear layer, which leads the output to also have 32 batches of dimension 10 each.

```
>>> m = nn.Linear(20, 10)
>>> input = torch.randn(32, 20)
>>> output = m(input)
>>> print(output.size())
torch.Size([32, 10])
```

Listing 3.4: Declaring a linear layer in PyTorch

Convolutional Layer

PyTorch's 2-D convolutional layer has more features than the convolution operation described in section 2.2, namely multi-channel output. The main parameters provided upon declaration are:

- in_channels: number of input channels;
- out_channels: number of channels produced by the convolution;
- kernel_size: size of the convolving kernel;
- stride: optional for the stride of the convolution, defaults to 1;
- padding: optional for the padding added to the 4 sides of the input, defaults to 0;
- dilation: optional spacing between kernel elements, defaults to 1;
- bias: optional learnable bias added to the output, defaults to True.

After initialization, the convolutional layer receives a 4-dimensional input, $(N, C_{in}, H_{in}, W_{in})$, where N represents the number of images in the batch, C_{in} the channels of each image and H_{in} and W_{in} the height and width of the image, respectively. The produced output, with shape $(N, C_{out}, H_{out}, W_{out})$ is derived from equation:

$$\text{output}(N_i, C_{out_j}) = \text{bias}(C_{out_j}) + \sum_{k=0}^{C_{in}-1} \text{weight}(C_{out_j}, k) * \text{input}(N_i, k), \tag{3.1}$$

where $*$ represents the cross-correlation operator (as explained in section 2.2.4, most DL frameworks implement cross-correlation but call it convolution). From equation (3.1) it can be seen that the weights have 4 dimensions: 1 for the output channels, 1 for the input channels and 2 that make up the kernel that slides for the convolution. The bias has only 1 dimension for the output channels. Similarly to the linear layer, the weights and biases are initialized with random values from a distribution in the range $[-\sqrt{k}, \sqrt{k}]$, where $k = \frac{1}{C_{in} \times \text{kernel\_size}[0] \times \text{kernel\_size}[1]}$. The height (H_{out}) and width (W_{out}) of the output are determined by equation (3.2).

$$\begin{cases} H_{out} = \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \\ W_{out} = \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \end{cases} \tag{3.2}$$

Listing 3.5 presents a minimal example of declaring a convolutional layer and applying it to an input. In this example, unequal stride, padding and dilation are applied, the first element of the tuple corresponds to the height and the second to the width.

```
>>> m = nn.Conv2d(in_channels=16, out_channels=33, kernel_size=3, stride=(2, 1), padding
    =(4, 2), dilation=(3, 1))
>>> input = torch.randn(20, 16, 50, 100)
>>> output = m(input)
>>> print(output.size())
torch.Size([20, 33, 26, 102])
```

Listing 3.5: Declaring a convolutional layer in PyTorch

Average Pooling Layer

To declare an average pooling layer, the kernel_size must be provided, and optionally the stride (defaults to kernel_size) and padding (defaults to 0). Subsequently, this layer receives a 4-dimensional input, (N, C, H_{in}, W_{in}), where N represents the number of images in the batch, C the channels of each image and H_{in} and W_{in} the height and width of the image, respectively.

The produced output, with shape (N, C, H_{out}, W_{out}) is derived from equation:

$$\text{output}(N_i, C_j, h, w) = \frac{1}{kH \times kW} \sum_{m=0}^{kH-1} \sum_{n=0}^{kW-1} \text{input}(N_i, C_j, \text{stride}[0] \times h + m, \text{stride}[1] \times w + n), \tag{3.3}$$

where kH is the kernel height and kW is the kernel width. The height (H_{out}) and width (W_{out}) of the output are determined by equation (3.4).

$$\begin{cases} H_{out} = \frac{H_{in} + 2 \times \text{padding}[0] - \text{kernel\_size}[0]}{\text{stride}[0]} + 1 \\ W_{out} = \frac{W_{in} + 2 \times \text{padding}[1] - \text{kernel\_size}[1]}{\text{stride}[1]} + 1 \end{cases} \tag{3.4}$$

Listing 3.6 presents a minimal example of declaring an average pooling layer and applying it to an input. This example has a rectangular kernel_size, where the first element of the tuple corresponds to the height of the kernel and the second to the width.

```
>>> m = nn.AvgPool2d((3, 2), stride=(2, 1))
>>> input = torch.randn(20, 16, 50, 32)
>>> output = m(input)
>>> print(output.size())
torch.Size([20, 16, 24, 31])
```

Listing 3.6: Declaring an average pooling layer in PyTorch

Maximum Pooling Layer

To declare a maximum pooling layer, the kernel_size must be provided, and optionally the stride (defaults to kernel_size), padding (defaults to 0) and dilation (defaults to 1). Subsequently, this layer receives a 4-dimensional input, (N, C, H_{in}, W_{in}), where N represents the number of images in the batch, C the channels of each image and H_{in} and W_{in} the height and width of the image, respectively.

The produced output, with shape (N, C, H_{out}, W_{out}) is derived from equation:

$$\text{output}(N_i, C_j, h, w) = \max_{m=0,\ldots,kH-1} \max_{m=0,\ldots,kW-1} \{\text{input}(N_i, C_j, \text{stride}[0] \times h + m, \text{stride}[1] \times w + n)\}, \tag{3.5}$$

where kH is the kernel height and kW is the kernel width. The height (H_{out}) and width (W_{out}) of the output are determined by equation (3.6).

$$\begin{cases} H_{out} = \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \\ W_{out} = \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \end{cases} \tag{3.6}$$

Listing 3.7 presents a minimal example of declaring a maximum pooling layer and applying it to an input. This example has a rectangular kernel_size, the first element of the tuple corresponds to the height of the kernel and the second to the width.

```
>>> m = nn.MaxPool2d((3, 2), stride=(2, 1))
>>> input = torch.randn(20, 16, 50, 32)
>>> output = m(input)
>>> print(output.size())
torch.Size([20, 16, 24, 31])
```

Listing 3.7: Declaring a maximum pooling layer in PyTorch

Dropout Layer

The dropout layer's interface is fairly simple, since it only zeroes out some inputs with probability p while the remaining are scaled by $\frac{1}{1-p}$. This way, upon initialization it receives the optional parameter p, that defaults to zero, and can subsequently be applied to an input of arbitrary shape. Listing 3.8 provides an example of passing a random 4×4 sized input through a dropout layer.

```
>>> m = nn.Dropout(p=0.2)
>>> input = torch.randn(4, 4)
>>> print(input)
tensor([[-1.0306, -0.6194,  0.8334, -0.3314],
        [-0.0365,  1.9288,  1.0851,  0.0906],
        [-0.4671,  1.5090,  0.1446,  0.6689],
        [ 1.1268,  2.2224,  2.6797,  1.3940]])
>>> output = m(input)
>>> print(output)
tensor([[-1.2883, -0.0000,  1.0418, -0.0000],
        [-0.0457,  2.4110,  1.3564,  0.1133],
        [-0.5838,  1.8863,  0.1808,  0.8361],
        [ 1.4085,  2.7780,  0.0000,  0.0000]])
```

Listing 3.8: Declaring a Dropout layer in PyTorch

3.2.2 Activation functions

PyTorch has implementations for a plethora of activation functions, among them all of the referred in 2.2.3 (Sigmoid, TanH and ReLU) since these are the most used in NN design [52].

All have the same structure: upon initialization there are no arguments; subsequently, the input is of shape $(N, *)$, where N is the batch size and $*$ represents an arbitrary number of dimensions. The pointwise function (i.e. to each element of the input) is applied over the input. The output has the same shape as the input. Listing 3.9 provides an example of applying the TanH function to a 2×2 sized input.

```
>>> m = nn.Tanh()
>>> input = torch.tensor([[1.2, 0.4], [4.3, 6]])
>>> output = m(input)
>>> print(output)
tensor([[0.8337, 0.3799],
        [0.9996, 1.0000]])
```

Listing 3.9: Declaring a TanH layer in PyTorch

3.2.3 Loss functions

PyTorch has implementations of the major loss functions, among them the two refered in section 2.2.5: Mean Squared Error and Cross Entropy Loss.

Mean Squared Error (MSE)

The MSE class can compute either the square of the error, the sum of the squares of the error, or the mean of the squares of the error (default). This is dependent on the optional parameter *reduction*, passed upon creation as a string whose default is 'mean'.

The loss without any reduction is given by:

$$l(x, y) = L = \{l_1, ..., l_N\}^T, \qquad l_n = (x_n - y_n)^2, \tag{3.7}$$

where N is the batch size and x_n and y_n are tensors of arbitrary shape, with n elements each. If reduction is not none, the two possibilities are:

$$l(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction is 'mean'} \\ \text{sum}(L), & \text{if reduction is 'sum'} \end{cases} \tag{3.8}$$

When called, it should receive two parameters: the input and the target, both with shapes $(N, *)$, where N represents the batch size and $*$ any additional number of dimensions. Listing 3.10 provides an example of applying the MSE loss function to an input.

```
>>> loss_criterion = nn.MSELoss(reduction='mean')
>>> input = torch.tensor([[1.2, 0.4, 2.3], [4.3, 6, -5.1]])
>>> target = torch.tensor([[1.2, -1.2, 1], [4, 7.2, -5.4]])
>>> loss = loss_criterion(input, target)
>>> print(loss)
tensor(0.9783)
```

Listing 3.10: Applying the MSE loss function to an input

Cross Entropy Loss

Upon initialization, the Cross Entropy Loss class receives two optional arguments: weight and reduction. Weight is an 1-D tensor with the cardinality equal to the number of classes, C. It is useful when the training set is unbalanced, with one class being predominant over the others [59]. Similarly to MSE, the *reduction* parameter determines whether the losses are unaltered, averaged, or summed over a batch.

When called, it receives two arguments: an input tensor of size (N, C), where N is the batch size, and a target of size N, where each element is in the range $[0, C-1]$, representing the index of the target class. As mentioned in section 2.2.5, Cross Entropy loss is mainly used in classification problems. PyTorch's implementation of it is designed with this in mind, hence the target corresponding to class indexes. Equation (3.9) describes the loss for each element of the input, x, where $target$ is the index of the target class.

$$loss(x, target) = -\log\left(\frac{\exp(x[target])}{\sum_j \exp(x[j])}\right) = -x[target] + \log\left(\sum_j \exp(x[j])\right) \tag{3.9}$$

In case that the weight argument is specified, equation (3.9) becomes

$$loss(x, target) = weight[target]\Big(-x[target] + \log\left(\sum_j \exp(x[j])\right)\Big). \tag{3.10}$$

Depending on the value of the *reduction* parameter, losses are either unaltered, summed or average over the batch. For the default, average, if the weight argument is specified it becomes a weighted average, given by equation (3.11).

$$loss = \frac{\sum_{i=1}^{N} loss(i, target[i])}{\sum_{i=1}^{N} weight[target[i]]} \tag{3.11}$$

Listing 3.11, in the next subsection, provides an example of applying the MSE loss function to an input.

Automatic Gradient Computation

As mentioned in section 3.1, one of the main features of PyTorch is the automatic gradient calculation. While the network is being constructed, the trainable weights belonging to any of the layers described in the current section have the attribute *requires_grad* set to *True*. This implies that each intermediary tensor also stores the function that originated it (in the attribute *grad_fn*), as the example in listing 3.1 illustrated.

Once the loss function is applied, the *backward* method can be called to compute the gradients. It goes over the network in the reverse direction, computing the partial derivate with respect to each weight (w) of the network, making use of the chain rule. This gradient is stored in the attribute $grad$ of each weight, through $w.grad \mathrel{+}= \frac{\partial loss}{\partial w}$. The gradients should be zeroed before this step, in order to avoid accumulating with the ones calculated in previous backward passes. Listing 3.11 provides a minimal example of this calculation. In it, the input is passed through a linear layer to produce the output (NN with only 1 hidden layer). The output stores the function that originated it so that the chain rule can be applied in the backward pass. The weights of the linear layer have their attribute *grad* set to *None* at first, and after the call to *loss.backward()* this attribute holds the derivate of the loss with respect to each weight.

```
>>> input = torch.tensor([[1.4, 5.6, 0.5], [-0.4, 4, 3.4]])
>>> m = nn.Linear(3,4)
>>> output = m(input)
>>> print(output)
tensor([[-1.3677, -0.7026,  1.7642,  3.3604],
        [-0.4372, -0.6408,  0.3862,  2.4435]], grad_fn=<AddmmBackward0>)
>>> target = torch.tensor([2, 1])
>>> loss_criterion = nn.CrossEntropyLoss()
>>> loss = loss_criterion(output, target)
>>> print(m.weight)
Parameter containing:
tensor([[-0.4047, -0.2056, -0.0437],
        [ 0.2410, -0.1743,  0.0747],
        [-0.2235,  0.3294, -0.4321],
        [ 0.4850,  0.4107,  0.2115]], requires_grad=True)
>>> print(m.weight.grad)

>>> loss.backward()
>>> print(m.weight.grad)
tensor([[-0.0041,  0.1114,  0.0793],
        [ 0.2024, -1.8864, -1.6332],
        [-0.6053, -2.1303, -0.0321],
        [ 0.4071,  3.9053,  1.5860]])
```

Listing 3.11: Applying the Cross Entropy loss function to an input and computing the gradients

3.2.4 Optimizer

As mentioned in section 2.2.6, after the gradients are computed the weights can be updated through different algorithms, all based on the principle of updating the weights in the opposite direction of their gradient - gradient descent.

PyTorch has a class ($torch.optim.Optimizer$) that serves as the base class for all optimization algorithms implemented. Both optimization algorithms referred in section 2.2.6 inherit from this class. Upon initialization, an iterable of the parameters to optimize should be passed. Additionally, each specialization of the base class has other parameters that can be passed during initialization:

- Stochastic Gradient Descent (SGD): learning rate and optional momentum term, as per equation (2.28).
- Adam: optional learning rate, β_1, β_2 and ϵ, as per equation (2.31). These default to the values proposed in the paper that introduced Adam [61].

As mentioned in the previous section, the gradients should be zeroed before the *loss.backward()* function is called, so that they do not accumulate over different forward passes. This is done through the optimizer's method *zero_grad()*. After the gradients are backpropagated through *loss.backward()*, to perform the update the optimizer's function *step()* is used. Listing 3.12 provides an example of using Adam to update the weights of the simple network proposed in listing 3.11 of the previous section.

```
>>> input = torch.tensor([[1.4, 5.6, 0.5], [-0.4, 4, 3.4]])
>>> m = nn.Linear(3,4)
>>> output = m(input)
>>> target = torch.tensor([2, 1])
>>> optimizer = torch.optim.Adam(m.parameters())
>>> loss_criterion = nn.CrossEntropyLoss()
>>> loss = loss_criterion(output, target)
>>> optimizer.zero_grad()
>>> loss.backward()
>>> print(m.weight)
Parameter containing:
tensor([[-0.2461, -0.2261, -0.2981],
        [-0.0325,  0.4279,  0.3408],
        [ 0.0308, -0.4562,  0.1933],
        [-0.1138, -0.5093, -0.4616]], requires_grad=True)
>>> optimizer.step()
>>> print(m.weight)
Parameter containing:
tensor([[-0.2471, -0.2271, -0.2991],
        [-0.0335,  0.4269,  0.3398],
        [ 0.0318, -0.4552,  0.1943],
        [-0.1148, -0.5103, -0.4626]], requires_grad=True)
```

Listing 3.12: Applying the Adam optimization algorithm to a simple network with 1 hidden layer

3.2.5 Model Training Example

To start the training process, data should be loaded into tensors to serve as input to the network. Pytorch has a package called *torchvision*, dedicated to computer vision, that has an utility to download the most common computer vision datasets. Among others, all of the image datasets mentioned in section 2.2.7 are available through this utility. Another utility called *Dataloader* provides a wrapper around the dataset. With it, iteration in batches and automatic shuffling of the data from one epoch to the next is abstracted from the end user.

Listing 3.13 provides an example of loading the MNIST dataset and wrapping it into a dataloader that provides 32 data samples (batch size) per iteration. In order to transform the image into a tensor, the utility *transforms* from *torchvision* is used. The resulting input tensor, X, is a 4-D tensor where the first dimension represents the batch size (32), the second the number of channels of the image (1 since it is grayscale), and the third and fourth the height and width of the image, respectively.

```
>>> from torchvision import datasets, transforms
>>> from torch.utils.data import DataLoader
>>> train_dataset = datasets.MNIST(root='mnist_data', train=True, transform=transforms.
    ToTensor(), download=True)
>>> train_loader = DataLoader(dataset=train_dataset, batch_size=32, shuffle=True)
>>> for X, y_true in train_loader:
...     print(X.shape)
...     break
...
torch.Size([32, 1, 28, 28])
```

Listing 3.13: Loading the MNIST dataset and wrapping it into a dataloader

PyTorch provides a base class *Module* from which the class representing the network should be derived. In the *__init__* method of the network's class, the design of the network should be specified, through the layers that constitute it. This class should also implement the method *forward*, responsible for passing the input through the network's layers and returning the output.

After the definition of the model, the 4 phases of the training process described in section 2.2.2 (forward pass, loss calculation, backward pass and weight update) are done for each batch of the training data. After going over the entire training data, an **epoch** is said to be completed. For a neural network to achieve a desirable precision on unseen data, the training is repeated for several epochs. However, if the number of epochs is too large, there is the risk of overfitting the training data and thus perform poorly on unseen data.

Combining all these concepts, listing 3.14 provides an end-to-end example of training the LeNet-5 network, presented in section 2.2.8, over the MNIST dataset.

```
class LeNet5(nn.Module):
    def __init__(self, n_classes):
        super(LeNet5, self).__init__()

        self.feature_extractor = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, stride=1),
            nn.Tanh(),
            nn.AvgPool2d(kernel_size=2),
            nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride=1),
            nn.Tanh(),
            nn.AvgPool2d(kernel_size=2),
            nn.Conv2d(in_channels=16, out_channels=120, kernel_size=5, stride=1),
            nn.Tanh()
        )
        self.classifier = nn.Sequential(
            nn.Linear(in_features=120, out_features=84),
            nn.Tanh(),
            nn.Linear(in_features=84, out_features=n_classes),
        )

    def forward(self, x):
        x = self.feature_extractor(x)
        x = torch.flatten(x, 1)
        logits = self.classifier(x)
        return logits

model = LeNet5(n_classes=10)
optimizer = torch.optim.Adam(model.parameters())
loss_criterion = nn.CrossEntropyLoss()

train_dataset = datasets.MNIST(root='mnist_data', train=True, transform=transforms.
    ToTensor(), download=True)
train_loader = DataLoader(dataset=train_dataset, batch_size=32, shuffle=True)

for epoch in range(0, epochs):
    for X, y_true in train_loader:
        optimizer.zero_grad()
        y_hat  = model(X)
        loss = loss_criterion(y_hat, y_true)
        loss.backward()
        optimizer.step()
```

Listing 3.14: Training the LeNet-5 network on the MNIST dataset

3.3 Summary

This chapter provides an overview of the API of the PyTorch framework. Firstly, the concept and utilities of Tensors were presented, given that these are the fundamental data structures of PyTorch. The data types currently supported were also presented, with 32-bit floating point as the default. Then, the PyTorch interface for the NN layers and functions described in the previous chapter was exposed. Finally, an example of end-to-end training of a model was showcased.

Chapter 4

Supporting Posit in PyTorch

Contents

The previous chapter detailed how PyTorch exposes the main DL layers and functions in a simple to use interface. This upper level API is fully in Python and most implementation details are abstracted from the end user. However, the tensor data structure, the CPU and GPU operators and the automatic differentiation are fully implemented in C++ on the backend to achieve high performance [8]. Therefore, in order to support posits in PyTorch's Python frontend, the required modifications must be implemented within the C++ backend.

This chapter describes the necessary steps to incorporate Posits in the PyTorch backend. It starts with an overview on how one can contribute to the project. A description of the organization of PyTorch's source code follows: the abstraction layers, the architecture, the binding between C++ and Python, etc.. After this introduction to the structure, the introduced contributions that were made to support posits are presented. Finally, the interface to use Posits in the Python frontend is exposed.

4.1 Contributing to PyTorch

As mentioned in the previous chapter, PyTorch is an open-source framework managed by Facebook. In practice, this means that developers at Facebook are the maintainers of the code and are the ones responsible for monitoring and accepting contributions of the wider community. This way, people wishing to contribute can either go through the issue list and propose a fix, or put forward their feature suggestion by raising an issue. By the time of writing of this thesis, over 5000 issues were open on GitHub [74].

At the beginning of this work, the author also went through this process, opening an issue stating the intention to add posit support in PyTorch. From here, three possibilities were considered:

- Create a Python class to represent a posit tensor, similar to the tensor class;
- Create a C++ class that represents a posit tensor and bind it to Python;
- Add a new data type representing posit to the backend and extend the existing classes to support it.

The first two options had the advantage of being relatively simple to implement, since the class that would represent the posit tensor would be fully customizable. However, the first option would have the inconvenient of introducing a significant overhead, since this class would need to be supported by an existing tensor structure. The second option would also have the inconvenient of leading to two separate tensor structures, one for the already supported data types of PyTorch and a different one for Posits.

In contrast, the third option implies adding support throughout the whole verticality of PyTorch's structure, which means that a thorough understanding of the organization of the codebase is needed. Nonetheless, this leads to the most desirable user interface, since the network design is exactly the same as for floats, and Posits are incorporated in the same way as other non-native data types that PyTorch already supports. Therefore, this was the chosen approach for this work. As an example, listing 4.1 compares the creation of a float tensor and a posit tensor.

```
>>> float_tensor = torch.tensor([[1.2, 3.4], [3.1, 4.2]], requires_grad=True)
>>> posit_tensor = torch.tensor([[1.2, 3.4], [3.1, 4.2]], dtype=torch.posit16es2,
    requires_grad=True)
```

Listing 4.1: Comparison of the creation of a float tensor with the creation of a posit tensor

4.2 PyTorch's Internal Structure

In order to understand how Posits were supported in PyTorch as one of the native data types, it is first important to understand the internal structure of PyTorch's codebase and the main abstraction layers in it. Even though the codebase is constantly mutating and has undergone a major design change throughout 2019 [74], this section addresses its structure at the time of writing of this thesis.

4.2.1 Codebase Structure

At the time of writing of this thesis, the PyTorch codebase had over 8000 files with over 2 million lines of code. Of this, 60% is C++ code and 35% Python code, the remaining 5% being comprised of C++ CUDA (gpu specific code) and mostly deprecated C code. [74]. The codebase is divided in 3 main directories that encompass all the core functionalities of the framework:

- **c10**: The core library files that are used in every other part of the code, common to both server and mobile;
- **aten**: The C++ tensor library, that is, where all the tensor definitions and operators are implemented;
- **torch**: The translation from C++ code to the Python frontend, with both C++ and Python files.

In order to support Posit, files from all of the 3 directories must be altered: *c10* to define the new data type for Posits; *aten* to define the operators on tensors with posit data and *torch* to expose posit tensors in the Python frontend. From these, aten is the most relevant, since it is where every function actuating on Posits will be defined. This directory is further subdivided in subdirectories comprising deprecated code (previous to the major refactor undergone in 2019), and the more recent code under *aten/src/ATen*. Within *aten/src/ATen* there are several subdirectories (for CPU operators, CUDA operators, operators that use external libraries, etc.) making the code organized in a way that makes it intuitive to navigate.

Besides those 3 core directories, there are others that address the different stages of development of the codebase: a directory with the scripts for the build process; a directory with the unit tests of the Python frontend and another dedicated to the tool used for integration of new code.

4.2.2 Tensor Implementation

A tensor consists of a multidimensional array, whose data is stored in memory, usually with each row contiguous to the previous. Pytorch's most common storage format for tensors is this, also denoted as strided layout [8]. This way, the tensor's metadata should hold not only the size of each dimension, but also the stride associated with it. The stride is what the logical index should be multiplied with in order to get to the physical index where the element is stored. Moreover, there is an offset, representing the position in memory where the count starts. Equation (4.1) provides the translation between the logical indexes and the physical position in memory for a tensor with n dimensions.

$$\text{memory position} = \text{index}[n-1] \times \text{stride}[n-1] + \cdots + \text{index}[0] \times \text{stride}[0] + \text{offset} \qquad (4.1)$$

Figure 4.1 provides an example where a 2×2 sized tensor whose data is stored row after row leads to a stride of 2 for the first dimension and 1 for the second. With these strides, the logical position $[1, 0]$ can be translated to the physical position 2 to fetch the data from memory.

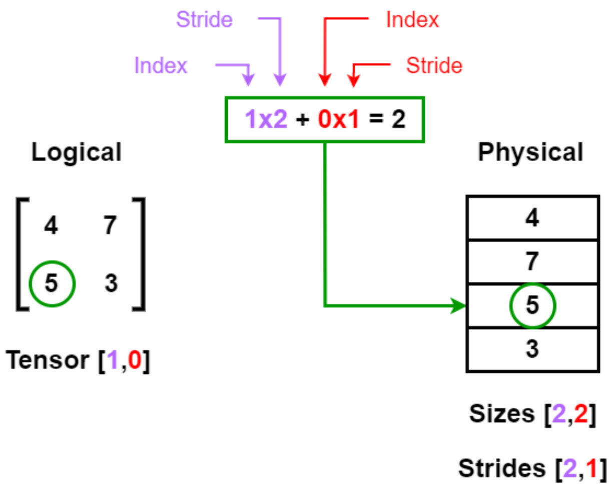


Figure 4.1: Illustration of a tensor with a strided layout.

From here, it follows that for each tensor some metadata needs to be stored, namely the sizes and strides. Besides these, three main parameters are stored as metadata:

- **device**: represents where the tensor's physical memory is stored: CPU, GPU, TPU, etc.;
- **layout**: describes how to logical interpret the physical data. The most common is a strided tensor, but PyTorch supports other types of memory configurations;
- **dtype**: corresponds to the type of the tensor's data: 32-bit floating-point, 64-bit integer, brain floating-point, etc..

However, the storing of this metadata is divided according to whether it is metadata corresponding to the logical interpretation of a tensor or to the physical storage of data. This way, tensors have a level of indirection: the Tensor structure that holds the metadata relative to the logical interpretation (layout, sizes, strides, offset) and the Storage structure that records the device and data type of the tensor as well as a pointer to the raw data.

With this level of indirection, PyTorch effectively supports different views of a Tensor, that is, different logical interpretations of the same underlying data. As an example, a tensor containing the second column of the tensor represented in Figure 4.1 could be based on the same physical data, but with sizes $= [2]$, strides $= [2]$ and offset $= 1$. This way, to access the element at index $[1]$ of this tensor, using equation 4.1, the memory position would be 3.

The number of possible types of tensors comes from the cartesian product of the 3 parameters presented above: *device*, *layout* and *dtype*. In order to extend tensors to support Posit, it is enough to add another *dtype* representing Posit. This is detailed in section 4.3.

Tensor Iterator Utility

A lot of operations involve iterating over all the elements of a tensor (or more than one tensor). This is the case not only in all point-wise operations, where a certain function is applied to all the elements of the tensor individually, but also for operations involving two tensors, (e.g. addition). In order to facilitate these operations, the *TensorIterator* API offers a standardized way to iterate over elements of a tensor, automatically parallelizing operations, while abstracting device and data type details.

A *TensorIterator* should first be built through *TensorIteratorConfig*, where the input and output tensors are specified, along with more information such as whether there should be checks on the data types or dimensions, among other utilities. Once built, it can be iterated over using the *for_each* function, that can loop over 1 or 2 dimensions at once. There are also built-in kernels that take a *TensorIterator* as argument and apply an operation over each element. Listing 4.2 provides an example of building a *TensorIterator* with two inputs and using the built-in *cpu_kernel* to perform addition. This implementation works for all data types (as long as the $+$ operator is overloaded), since the *dtype* is abstracted in the *TensorIterator*.

```
at::TensorIteratorConfig iter_config;
iter_config
  .add_output(c)
  .add_input(a)
  .add_input(b);

auto iter = iter_config.build();
auto datatype = iter.dtype()
at::native::cpu_kernel(iter, [] (datatype a, datatype b) -> datatype {
  return a + b;
});
```

Listing 4.2: Example of usage of the *TensorIterator* utility

4.2.3 Dispatcher

Operators to actuate on tensors need to be different depending on the type of the tensor. The implementation of matrix multiplication, for instance, is different if tensors are stored on the CPU or the GPU, if the data type is integers or complex reals, etc.. For this purpose, PyTorch has an abstraction layer, working similarly to a virtual table, that addresses this issue, called **dispatcher**.

The dispatcher is implemented through a bitset, called *DispatchKeySet*, where each relevant metadata contributes with a key to the set. These keys are ordered in decreasing order of priority. A kernel is called for the first key of the set, adding that key to the exclude set. In the next pass through the dispatcher, the next key is used to identify the kernel to be called. With this logic, the same dispatch key set can be used for multiple calls, which is useful for PyTorch's automatic differentiation (autograd), for example.

Figure 4.2 provides an example where an operation is to be performed on 2 tensors, one on the CPU and the other on the GPU. The key for autograd is excluded from the bitset - since it has already been handled in a previous dispatch - and the backend select key is the one used to call the kernel that will address this operation.

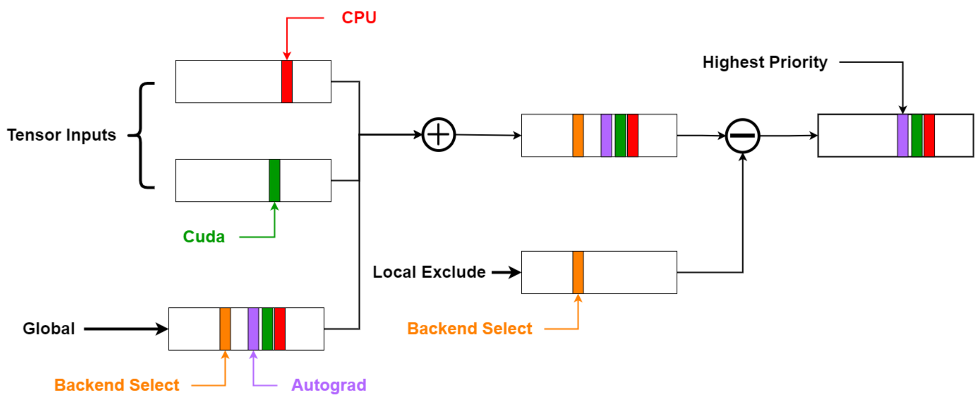


Figure 4.2: Illustration of a dispatcher call.

The main advantage of the dispatcher is its decentralized nature. Once the dispatch keys are defined, all the kernels corresponding to a given operator can be defined independently, without the need for a centralized if statement.

4.2.4 Kernels

For each dispatch key, there should be a kernel that implements the intended operation. In order to register these kernels, PyTorch provides the Registration API, which provides 4 endpoints in the form of C++ macros:

- **DECLARE_DISPATCH**: where the function signature is associated with the name of the dispatch registry;
- **DEFINE_DISPATCH**: defines a function to be the dispatch registry;
- **TORCH_IMPL_FUNC**: calls the dispatch registry function through the function that is exposed to higher-level APIs;
- **REGISTER_DISPATCH**: associates the dispatch registry function with the kernel that implements the operation.

Even though the API is not intuitive at first and implies changing multiple files, it provides flexibility and speed when calling dispatched functions [75]. Listing 4.3 provides an example for the *add* function.

```cpp
// file aten/src/ATen/native/BinaryOps.h
DECLARE_DISPATCH(void(*)(TensorIteratorBase&, const Scalar& alpha), add_stub);

// file aten/src/ATen/native/BinaryOps.cpp
DEFINE_DISPATCH(add_stub);

TORCH_IMPL_FUNC(add_out) (
  const Tensor& self, const Tensor& other, const Scalar& alpha, const Tensor& result
) {
  add_stub(device_type(), *this, alpha);
}

// file aten/src/ATen/native/cpu/BinaryOpsKernel.cpp
void add_kernel(TensorIteratorBase& iter, const Scalar& alpha_scalar) {
    //implementation of the add function on CPU
}

REGISTER_DISPATCH(add_stub, &add_kernel);

// file aten/src/ATen/native/cuda/BinaryAddSubKernel.cu
void add_kernel_cuda(TensorIteratorBase& iter, const Scalar& alpha_scalar) {
    //implementation of the add function on gpu
}

REGISTER_DISPATCH(add_stub, &add_kernel_cuda);
```

Listing 4.3: Registration API to associate function calls to kernel implementations

In this example, *add_stub* is the dispatch registry function, *add_out* is the function that is exposed to higher level API's, *add_kernel* is the CPU implementation of the add operation and *add_kernel_cuda* the gpu implementation.

4.3 Posit Integration in PyTorch

With an understanding of the main structure of the backend as exposed in the previous section, adding support for posits must make use of these tools. This section presents the different stages of this work, starting from adding the posit data type to Pytorch up to the support of the relevant operations on posit tensors and the exposure in the Python frontend.

4.3.1 Posit Data Type

As mentioned in section 4.1, the adopted strategy to integrate Posits was to add them as one of the built-in data types supported by PyTorch. This approach has the advantage of exposing Posits in the frontend as any other data type, avoiding the need for special representations. To emulate the operations with Posits, the Universal library was the one chosen, as mentioned in section 2.1.5. Hence, in order for the posit manipulation functions to be available in the namespaces of PyTorch, two files were used as translation: *c10/util/Posit16es2.h* for the posit declaration and *c10/util/Posit16es2-math.h* for the math functions. A snippet of the code in these two files is presented in listing 4.4.

```
// file c10/util/Posit16es2.h
#include <universal/number/posit/posit.hpp>
namespace c10 {
  using posit16es2 = sw::universal::posit<16,2>;
}

// file c10/util/Posit16es2-math.h
namespace std {
    inline c10::posit16es2 tanh(c10::posit16es2 a) {
      return sw::universal::tanh(a);
    }
}
```

Listing 4.4: Conversions from the Universal library functions to those in PyTorch's namespaces

PyTorch implements all the data types in the *c10/core* directory (since this is the common directory to the whole project) through the macro *AT_FORALL_SCALAR_TYPES_WITH_COMPLEX_AND_QINTS*. In it, a data type is associated with an alias (with a number associated), to be used throughout the codebase for comparison purposes. This file also contains a macro that encompasses all the data types belonging to a given set (integers, floats, complex, etc.), along with a helper function to determine if a type belongs to a given set. These were extended for posits with $nbits = 16$ and $es = 2.$, as listing 4.5 shows.

```
// file c10/core/ScalarType.h
#include <c10/util/Posit16es2.h>

#define AT_FORALL_SCALAR_TYPES_WITH_COMPLEX_AND_QINTS(_) \
  _(uint8_t, Byte) /* 0 */                              \
  _(int8_t, Char) /* 1 */                               \
  _(int16_t, Short) /* 2 */                             \
  _(int, Int) /* 3 */                                   \
  _(int64_t, Long) /* 4 */                              \
  _(at::Half, Half) /* 5 */                             \
  _(float, Float) /* 6 */                               \
  _(double, Double) /* 7 */                             \
  _(c10::complex<c10::Half>, ComplexHalf) /* 8 */       \
  _(c10::complex<float>, ComplexFloat) /* 9 */          \
  _(c10::complex<double>, ComplexDouble) /* 10 */       \
  _(bool, Bool) /* 11 */                                \
  _(c10::qint8, QInt8) /* 12 */                         \
  _(c10::quint8, QUInt8) /* 13 */                       \
  _(c10::qint32, QInt32) /* 14 */                       \
  _(at::BFloat16, BFloat16) /* 15 */                    \
  _(c10::quint4x2, QUInt4x2) /* 16 */       \
  _(c10::posit16es2, Posit16es2) /* 17 */

#define AT_FORALL_POSIT_TYPES(_)        \
  _(c10::posit16es2, Posit16es2)

static inline bool isPositType(ScalarType t) {
  return t == ScalarType::Posit16es2;
}
```

Listing 4.5: Supported data types in PyTorch, including posit(16,2)

Since a macro is a pre-processor directive that is only expanded at compile time, it is not possible to define a generic data type for Posits. Despite the flexibility provided by the templates of the Universal library to implement Posits, these must be instantiated in the macro at compile time.

As a consequence, only a posit datatype with $nbits = 16$ and $es = 2$ was added, since it is the most used to replace 32-bit floating-point numbers for NN training [19, 20]. Nonetheless, in order to support more posit configurations, it is enough to add them to the posit macros exposed throughout this section. After that, all operations will be automatically supported.

4.3.2 Dispatcher for Posit Types

As presented in section 4.2.3, PyTorch implements a cascading dispatcher system for operations on tensors. As such, it was necessary to define a dispatcher for the posit data types.

This dispatcher receives 3 arguments:

- **TYPE**: the type of the tensor;
- **NAME**: the name of the function being called;
- a C++ lambda function for the operation in question.

The dispatcher stores the type of the data, gets its alias and calls the passed function with the correct data type. Listing 4.6 shows the code for the newly created posit(16,2) datatype. To support more posit types, they should be added to the switch statement.

The amount of indirection levels is due to PyTorch's structure, where the usage of macros is maximised to increase the execution speed [76]. This comes at the cost of readability of the code and increase in complexity and build times when altering these core files.

```
#define AT_DISPATCH_POSIT_TYPES(TYPE, NAME, ...)             \
  [&] {                             \
    const auto& the_type = TYPE;                  \

    at::ScalarType _st = ::detail::scalar_type(the_type);          \
    RECORD_KERNEL_FUNCTION_DTYPE(NAME, _st);                \
    switch (_st) {
      \
      AT_PRIVATE_CASE_TYPE(NAME, at::ScalarType::Posit16es2, c10::posit16es2, __VA_ARGS__
    )      \
      default:                          \
          AT_ERROR(#NAME, " not implemented for '", toString(TYPE), "'");        \
    }                           \
  }()
```

Listing 4.6: Supported data types in PyTorch, including posit(16,2)

The next section will present the operations that were extended to support Posit. In all of them, this dispatcher is used, given the structure of the indirection levels of PyTorch.

4.3.3 NN Operators for Posits

PyTorch contains thousands of operators, and supporting all of them for a new data type results in a workload that is above the scope of this thesis. This is evidenced by the two years taken to have full support for complex tensors in PyTorch [74]. As a result, the followed approach consisted of supporting the subset of operators needed to train a NN solely with Posits. To achieve this objective, all the functions and layers mentioned in the previous chapter were to be supported.

The binding from a C++ operator implementation and the corresponding Python frontend functions is mostly done by auto-generated code. Adding this to the size of the codebase, it is not recommended to navigate the stack trace to find the C++ function that needs to be extended for the new data type [76]. The most efficient way to locate the relevant function is to call it, record the error message and search through the codebase for that message. Listing 4.7 provides an example for the MSE loss function, where the error message *"mse_cpu" not implemented for 'Posit16es2'* can be searched through the the codebase to find the kernel that implements this operation.

```
>>> loss = nn.MSELoss()
>>> input = torch.tensor ([[1.2 , 0.4, 2.3], [4.3, 6,  -5.1]], dtype=torch.posit16es2)
>>> target = torch.tensor ([[1.2 ,  -1.2, 1], [4, 7.2,  -5.4]], dtype=torch.posit16es2)
>>> output = loss(input, target)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/afonsoluz/pytorch/torch/nn/modules/module.py", line 1102, in _call_impl
    return forward_call(*input, **kwargs)
  File "/home/afonsoluz/pytorch/torch/nn/modules/loss.py", line 520, in forward
    return F.mse_loss(input, target, reduction=self.reduction)
  File "/home/afonsoluz/pytorch/torch/nn/functional.py", line 3112, in mse_loss
    return torch._C._nn.mse_loss(expanded_input, expanded_target, _Reduction.get_enum(
    reduction))
RuntimeError: "mse_cpu" not implemented for 'Posit16es2'
```

Listing 4.7: Calling the MSE loss function before it was supported for Posits

Once the kernel is located, extending it for Posits can either consist of simply using the code that is being used for other dtypes or writing a specific kernel. The second happens when the implementation for other data types is making use of hardware optimizations not available for Posits, such as vectorization through the AVX2 instruction set [77].

Listing 4.8 provides an example of supporting the average pooling kernel through the same operation that was being done for the other data types. Since the kernel *cpu_avg_pool* only performs operations that are supported by the Universal library (and hence work with the posit datatype), it is enough to add it to the dispatcher that was already being used for the other data types. This is done by adding the alias for the posit(16,2) type *at::ScalarType::Posit16es2* as an argument to the dispatcher.

```
AT_DISPATCH_FLOATING_TYPES_AND2(at::ScalarType::Long, at::ScalarType::Posit16es2, input.
    scalar_type(), "avg_pool2d", [&] {
        cpu_avg_pool<scalar_t>(output, input, kW, kH, dW, dH, padW, padH,
    count_include_pad, divisor_override);
      });
```

Listing 4.8: Extending the average pooling operator for Posits through the same kernel as for other data types

Listing 4.9 provides an example of supporting the addition with scaling operation, where a different kernel should be used. Here, the *TensorIterator* API is explored, by using the *cpu_kernel*, which takes an iterator with two input tensors and applies the lambda function passed to them. This is an example of the case where for the other data types an hardware optimization was being used.

```
if (isPositType(dtype)) {
    AT_DISPATCH_POSIT_TYPES(dtype, "addcmul_cpu_out", [&] {
      scalar_t scalar_val = value.to<scalar_t>();
      cpu_kernel(
        iter,
        [=](scalar_t self_val, scalar_t t1_val, scalar_t t2_val) -> scalar_t {
          return self_val + scalar_val * t1_val * t2_val;
        }
      );
    });
}
```

Listing 4.9: Supporting addition with scaling by calling a separate kernel for Posits

There are also cases where the *TensorIterator* API cannot be used. In these cases, the iterations through the tensor to perform the operation should be written extensively. Until the point of the writing of this thesis, this only happened for the softmax activation function, used for the cross entropy loss function. The code for the kernel is presented in listing 4.10. This code implements equation (4.2), a variation of equation (2.26), where the maximum element of the last dimension (*dim*) is subtracted to each element so that the exponential is more numerically stable. Moreover, it is templated to work for any *posit_type*, which means that if more posit types are added the softmax function will automatically be supported.

$$S(\hat{y_i}) = \frac{\exp(y_i - \max_{k \in \text{last dim}}\{\hat{y_k}\})}{\sum\limits_{j \in \text{last dim}} \exp(y_j - \max_{k \in \text{last dim}}\{\hat{y_k}\})} \tag{4.2}$$

```
template <typename posit_type>
```

```
inline void _softmax_lastdim_posit(
    const Tensor& output,
    const Tensor& input) {

  int64_t outer_size = 1;
  int64_t dim_size = input.size(input.ndimension() - 1);

  for (int64_t i = 0; i < input.ndimension() - 1; ++i)
    outer_size *= input.size(i);

  posit_type* input_data_base = input.data_ptr<posit_type>();
  posit_type* output_data_base = output.data_ptr<posit_type>();

  int64_t grain_size = internal::GRAIN_SIZE / (16 * dim_size);
  if (grain_size < 1)
    grain_size = 1;

  parallel_for(
      0,
      outer_size,
      grain_size,
      [&](int64_t begin, int64_t end) {
        for (int64_t i = begin; i < end; i++) {
    posit_type* input_data = input_data_base + i * dim_size;
    posit_type* output_data = output_data_base + i * dim_size;
    posit_type max_input = 0;

    max_input = std::accumulate(input_data, input_data + dim_size, max_input, [&](
    posit_type max_input, posit_type elem) { return (max_input > elem ? max_input : elem)
    ; });

    std::transform(input_data, input_data + dim_size, output_data, [&](posit_type elem) {
     return std::exp(elem - max_input); } );

    posit_type tmp_sum = 0;

    tmp_sum = std::accumulate(output_data, output_data + dim_size, tmp_sum);
          tmp_sum = 1 / tmp_sum;

    std::transform(output_data, output_data + dim_size, output_data, [&](posit_type elem)
     { return elem * tmp_sum; });

        }
      });
}
```

Listing 4.10: Custom kernel for the softmax activation function for Posits

Table 4.1 lists all the operators whose support for Posits was implemented in the scope of this thesis. It presents the name of the operator, its role and whether the extension was using the existing kernel for other data types or through a custom kernel for Posits. In the latter case, the *TensorIterator* API was used, except for the softmax functions, as previously mentioned. Operators that call others on this table, such as the cross entropy loss function (combination of the log softmax function and the NLL loss function), are automatically supported.

Table 4.1: Operators that were extended to support Posits, along with their role and the type of extension.

Name of the operator	Role	Extension type
scalar_fill	Tensor Implementation	Same Kernel
local_scalar_dense	Tensor Implementation	Same Kernel
copy	Tensor Implementation	Specific Kernel
uniform	Tensor Implementation	Same Kernel
fill	Tensor Implementation	Specific Kernel
add	Addition	Specific Kernel
sub	Subtraction	Specific Kernel
sum	Reduction addition	Specific Kernel
max	Maximum Tensor Value	Specific Kernel
sqrt	Square Root	Specific Kernel
add_mm	Matrix Multiplication	Same Kernel
addcmul	Multiplication and Addition Scaling	Specific Kernel
addcdiv	Division and Addition	Specific Kernel
unfolded2d_acc	Convolution	Same Kernel
avg_pool2d	Average Pooling	Same Kernel
avg_pool2d_backward	Gradient Computation	Same Kernel
tanh	Activation function	Specific Kernel
tanh_backward	Gradient Computation	Specific Kernel
softmax	Activation Function	Specific Kernel
softmax_backward	Gradient Computation	Specific Kernel
log_softmax	Activation Function	Specific Kernel
log_softmax_backward	Gradient Computation	Specific Kernel
nll_loss	Loss Function	Same Kernel
nll_loss_backward	Gradient Computation	Same Kernel
mse	Loss Function	Specific Kernel

4.3.4 Posits in PyTorch's Frontend

As mentioned in the previous section, most of the code that binds the C++ operators to the Python frontend functions is auto-generated. Moreover, as it was referred in section 4.1, the approach followed to support Posits was such that in the frontend it can be used just as any other data type. This way, the only code related to the frontend was the connection of the C++ posit(16,2) to a PyTorch *dtype*. The chosen name was *posit16es2*, since it represents a posit with $nbits = 16$ and $es = 2$. This is shown in Listing 4.11.

```
case at::ScalarType::Posit16es2:
      return std::make_pair("posit16es2", "");
```

Listing 4.11: Posit(16,2) exposure to the Python frontend as *posit16es2*

After this, any tensor can be converted to posit. Furthermore, PyTorch also provides an utility to convert all of a network's parameters to a different data type. Hence, a NN model can be designed in the exact same way as it would be for floats. Upon training, it is also possible to convert all its parameters to Posit. This is shown in Listing 4.12, where the code previously shown in Listing 3.14 to train a LeNet-5 with floats is altered to train it with Posits.

```
model = LeNet5(n_classes=10).type(torch.posit16es2)
optimizer = torch.optim.Adam(model.parameters())
loss_criterion = nn.CrossEntropyLoss()

train_dataset = datasets.MNIST(root='mnist_data', train=True, transform=transforms.
    ToTensor(), download=True)
train_loader = DataLoader(dataset=train_dataset, batch_size=32, shuffle=True)

for epoch in range(0, epochs):
    for X, y_true in train_loader:
        X.type(torch.posit16es2)
        optimizer.zero_grad()
        y_hat  = model(X)
        loss = criterion(y_hat, y_true)
        loss.backward()
        optimizer.step()
```

Listing 4.12: Training a LeNet-5 network with Posits

4.4 Summary

In this chapter, the internals of PyTorch were explored. Firstly, the procedure to contribute to the source code of PyTorch was briefly described, along with the different possibilities considered to incorporate posit in the framework. To this end, it is important to understand the structure of the codebase, along with the main concepts and APIs useful for development of the codebase. With this knowledge in mind, the last section of the chapter details the incorporation of Posits, from the declaration of the new data type to the operator support and frontend exposure, showcasing how to train a model using Posits.

Chapter 5

Experimental Evaluation

Contents

In the previous chapter, the process of integrating of Posit operators in PyTorch was described. It finished with the exposure of the developed interface to design and train NN models using Posits. This chapter starts with a description of the benchmark models that are covered by this work. It is followed with the experimental evaluation conducted to validate the implementation of the operators presented in the previous chapter.

5.1 Coverage Analysis

With the support of the operators presented in Table 4.1, most of the building blocks that constitute the benchmark models presented in section 2.2.8 are supported. This section presents, for each of those models, the layers that are currently supported and whether the model is fully supported as a result of this work.

LeNet-5

LeNet-5 was the first CNN to achieve an accuracy above 99% in handwritten digit recognition, which led to it becoming a benchmark model. It is also a common entry point for CNNs given its relatively simple architecture and the many tutorials available for implementation in different DL platforms. It uses convolutional layers, average pooling layers, fully connected layers and TanH as the activation function. All of these elements are supported by this work, as is summarized in table 5.1.

Table 5.1: Supported layers and functions of LeNet-5.

Operator Name	Operator Type	Support
Convolution	Layer	Supported
Average Pooling	Layer	Supported
Fully Connected	Layer	Supported
TanH	Activation Function	Supported

CifarNet

CifarNet was designed to classify the Cifar-10 dataset, presented in section 2.2.7, with a relatively small network structure. Its architecture is similar to LeNet-5, the main differences being the introduction of padding in the convolutional layers, using ReLU instead of TanH for activations, using maximum pooling layers and using the softmax activation function in the output layer. All of its building blocks are supported by this work, as table 5.2 summarizes.

AlexNet

AlexNet, with 13 layers, was one of the largest networks at the time of its proposal in 2012, winning the ILSVRC. To the elements that constitute the previous models it adds dropout layers and overlapped

Table 5.2: Supported layers and functions of CifarNet.

Operator Name	Operator Type	Support
Padded Convolution	Layer	Supported
Average Pooling	Layer	Supported
Maximum Pooling	Layer	Supported
Fully Connected	Layer	Supported
ReLU	Activation Function	Supported
Softmax	Activation Function	Supported

pooling, which are also supported by the present work. This way, AlexNet is also fully supported, as table 5.3 summarizes.

Table 5.3: Supported layers and functions of AlexNet.

Operator Name	Operator Type	Support
Convolution	Layer	Supported
Overlapped Average Pooling	Layer	Supported
Overlapped Maximum Pooling	Layer	Supported
Fully Connected	Layer	Supported
Dropout	Layer	Supported
ReLU	Activation Function	Supported
Softmax	Activation Function	Supported

ResNet

Following AlexNet, even deeper architectures started to be explored, and the ResNet class of networks, proposed in 2015, explored up to 1000 layers. Of these layers, most are convolutional and pooling layers, but batch normalization layers are also present. These are not supported at the time of writing of this thesis. Table 5.4 summarizes the supported layers and functions.

Table 5.4: Supported layers and functions of ResNet.

Operator Name	Operator Type	Support
Convolution	Layer	Supported
Average Pooling	Layer	Supported
Maximum Pooling	Layer	Supported
Fully Connected	Layer	Supported
Batch Normalization	Layer	Not Supported
ReLU	Activation Function	Supported
Softmax	Activation Function	Supported

Loss Functions and Optimizers

The building blocks of each of the models presented are sufficient to design and infere with those architectures. However, in order to train a NN, a loss function and an optimizer algorithm are needed. Table 5.5 presents the loss functions and optimizers supported at the time of writing of this thesis.

Table 5.5: Supported loss functions and optimizers.

Function Name	Function Type
Mean Squared Error (MSE)	Loss Function
Cross Entropy Loss	Loss Function
Adam	Optimizer

5.2 Experimental Setup

To validate the implementation of the developed operators for Posits within PyTorch, the CNN LeNet-5 was trained with Posits with both the MNIST dataset and the more complex Fashion MNIST dataset. A block diagram representing the layers and parameters of the network is presented in Figure 5.1.

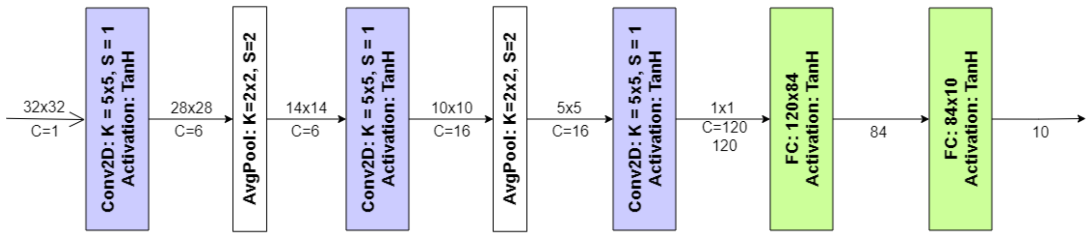


Figure 5.1: Block Diagram representing the architecture of the evaluated LeNet-5. K stands for kernel, S for stride and FC for fully connected.

In addition to the architecture of the network, it was necessary to define the loss function and the optimizer algorithm. The chosen loss function was the Cross Entropy Loss, since it is one of the most commonly used for multi-class classification problems [60]. The same reasoning was followed when chosing the Adam optimizer for the weight updates [48]. The hyperparameters of Adam (learning rate, β_1, β_2 and ϵ), described upon the presentation of the algorithm in section 2.2.6, were set as the default values. Table 5.6 summarizes the configurations of the various hyperparameters used for training of the CNN LeNet-5.

Table 5.6: Hyperparameter configurations for the training of LeNet-5.

Loss Function	Optimizer	Learning Rate	β_1	β_2	ϵ	Batch Size
Cross Entropy	Adam	0.001	0.9	0.999	1×10^{-8}	32

The training was conducted in a system with an Intel Xeon E312xx CPU with 8 cores, operating at 2.4GHz and with 32 GB of RAM.

5.2.1 Dataset reduction

As expected, given the overhead introduced by the emulation of Posits with software, the time taken to train the network significantly increased, reaching weeks per epoch for the whole dataset. This way, there was the need to substantially reduce the training time, in order to adjust it to the available time for the development of this thesis. The goal was to reduce it as much as possible and still draw a meaningful comparison between training with floats and Posits.

Naturally, the training time increases linearly with the number of training samples, since the training process consists of iterating through each batch of data and performing the same operations. This way, reducing the training dataset, whose initial size was 60000 samples, was the first solution adopted to reduce the training time.

To this end, a first evaluation of the model accuracy with decreasing portions of the dataset was done using floats. The results are presented in Figure 5.2, where the accuracy is the percentage of correct predictions.

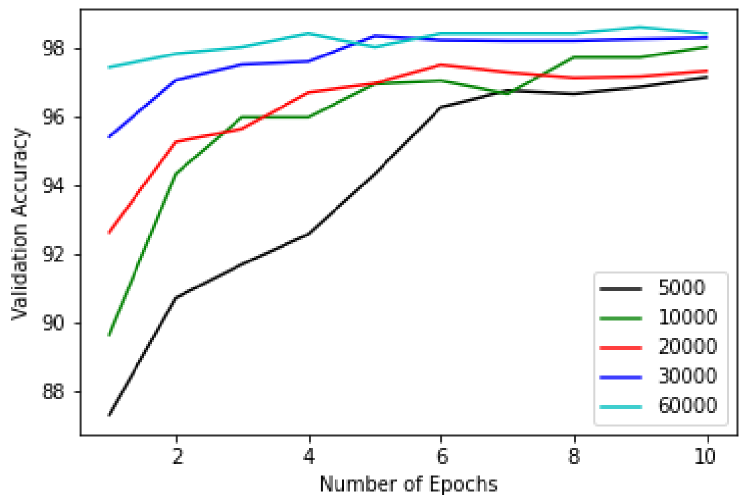


Figure 5.2: Validation accuracy when training LeNet-5 with different portions of the MNIST dataset.

As it can be observed, with the reduction of the number of samples used for training, the accuracy decrease is significant. Nonetheless, the accuracy still improves for each epoch, which means that the training is still being successful, even for the reduced number of samples. Given that the purpose of these experiments is to validate the implementation of Posits in PyTorch, the reduction in accuracy is acceptable, provided that the accuracy achieved with training based on Posits does not differ significantly.

The number of epochs also linearly affects the training time, since each epoch consists of a pass over the training dataset. As it can be observed in Figure 5.2, for the first 7 epochs the increase in accuracy is steady, which illustrates a successful training process. This way, 7 epochs was considered sufficient to evaluate the training with Posits.

Table 5.7 summarizes the covered reduced datasets, the number of samples and the number of epochs used for training.

Table 5.7: Covered datasets, number of samples and number of epochs.

Model	Dataset	Number of Samples	Number of Epochs
LeNet-5	MNIST	5000	7
LeNet-5	FashionMNIST	2500	7

5.3 LeNet-5 Training Evaluation

As mentioned in the previous section, the implemented CNN LeNet-5 model was trained on both the MNIST and FashionMNIST reduced datasets, using Posits with $nbits = 16$ and $es = 2$. The sample images of both datasets were normalized to 32×32 pixels. The training samples were shuffled, since the randomness is beneficial for model convergence and reduction of overfitting. Moreover, each experiment was repeated 2 times and the results averaged, given the randomness of the initialization of the weights of the network and the shuffling of the training data.

Figure 5.3 shows the evolution of the accuracy after each epoch for 32-bit floats and 16-bit posits on the MNIST dataset.

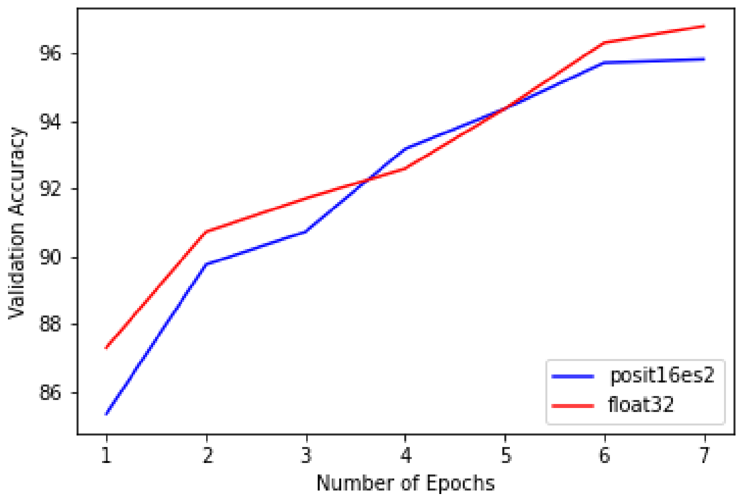


Figure 5.3: Comparison of the obtained accuracy of LeNet-5 training with floats and Posits on the MNIST dataset.

As it can be observed, the obtained accuracy with 16-bit Posits is similar to that of 32-bit floats, with a difference of around 1% after 7 epochs. This is in line with the results obtained in other works that evaluate training with small precision Posits [19, 20].

Figure 5.4 shows the evolution of the obtained accuracy after the execution of the same experiment on the FashionMNIST dataset.

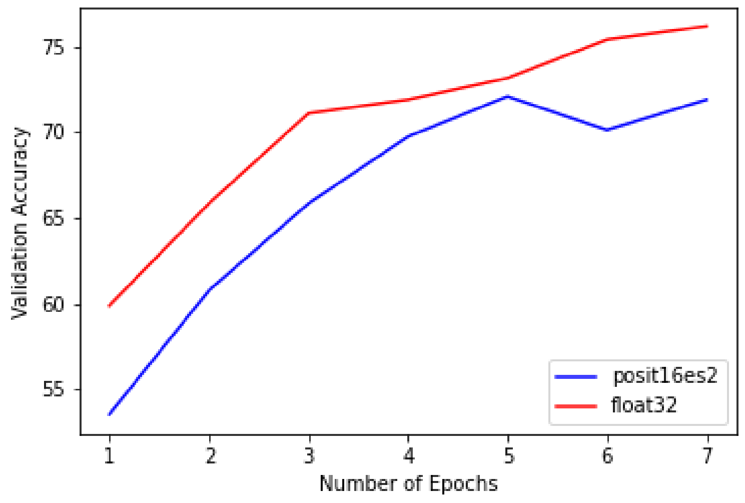


Figure 5.4: Comparison of the accuracy of LeNet-5 training with 32-bit floats and 16-bit Posits on the FashionMNIST dataset.

The FashionMNIST dataset is more complex, as it consists of clothes' items rather than digits. This leads to poorer performances, which is evidenced by the lower accuracy attained with 32-bit floats after 7 epochs of around 76%, contrasting with the 97% obtained when evaluating MNIST. Nonetheless, with Posits with $nbits = 16$ and $es = 2$ the attained accuracy of around 72% is 4% smaller than that of floats. This gap is likely due to the reduced number of samples used for training, which prevents the network from generalizing adequately.

5.4 Summary

This chapter presents the experimental evaluation that was conducted to validate the implementation of operations with Posits in PyTorch proposed in the previous chapter. It starts with exposing the experimental setup used, along with the need to reduce the number of samples used for training given the long execution times that operations with Posits take. Finally, it presents the results of evaluating the CNN LeNet-5 on both the MNIST and FashioMNIST datasets.

Chapter 6

Conclusions

Contents

6.1 Summary

Upon the introduction of the Posit number format (in 2017) as an alternative to traditional IEEE standard floating-point numbers, there has been an increasing interest in exploring its use in DL applications. To this end, researchers have developed their own platforms to design and train NNs using Posits. This has the disadvantage of not leveraging powerful and established DL frameworks, thus introducing an overhead. Of these frameworks, PyTorch is the most popular among researchers, given its shallow learning curve and flexibility.

Under these premises, the objective of this dissertation consisted of supporting Posits in PyTorch, in order to create a tool for further research with Posits in DL applications. The first step was to get acquainted with the framework by exploring its user facing API and understanding its internal structure.

From here, three approaches were considered to extend the framework to support Posits: to create a Python class that would represent a Posit tensor; to create a C++ class that would represent a Posit tensor and bind it to Python; to add Posits as a native type of PyTorch's tensors. The last option was followed, given that it represented the most complete integration and provided a seamless frontend API to design and train NNs with Posits.

Once the approach was established, the contribution was conducted on a bottom-up fashion. It started with the Posit data type and Tensor definition, followed by the support of the main NN operators for Posit Tensors. Finally, these tensors were exposed to the Python frontend. Given the internal structure of PyTorch, data types cannot be templated, which means that each Posit configuration has to be declared as an individual data type. This way, the Posit configuration with $\boldsymbol{nbits = 16}$ and $\boldsymbol{es = 2}$ was the one that was implemented, but other configurations can be seamlessly added by altering the core files.

In order to validate the implementation, the CNN LeNet-5 was trained and evaluated on both the MNIST and FashionMNIST datasets with Posits. There was the need to reduce the size of the datasets, since the software emulated nature of the Posit computations introduced a significant overhead. Training with Posits led to similar accuracies as with floats on MNIST. On FashionMNIST, the accuracies were smaller than floats, likely due to the reduced number of samples used for the network training.

The produced code is publicly available on GitHub [78] as a fork of the PyTorch repository. It contains documentation on the process of extending this work for more operators and more Posit data types, contribution that is also present in this dissertation.

6.2 Future Work

The presented work was naturally limited by the time available to develop a MSc thesis. Therefore, optimizations to the developed work and improvements to provide more complete support of Posits in PyTorch can be considered, namely:

- Including more Posit configurations to perform mixed precision training and evaluation of NN models. The process of extending the present work for more Posit configurations is detailed in this dissertation and publicly available on GitHub.

- Supporting a quire for accumulations of sums of products. Even though its inclusion is not as direct as other extensions, it is a compelling topic given the potential of the quire when performing computations with low-precision Posits.
- Integrating hardware dedicated to Posits in the framework. At the present date, this work is built on top of a software library that emulates Posit computations. However, given its modular nature, the implemented software layer could be replaced by a library based on hardware dedicated to Posits, with the potential of accelerating computations significantly.

Furthermore, the main objective of this work was to build a general tool to design and train DL models with Posits in a familiar and popular framework. Therefore, any research with Posits in the context of DL can be performed following this work, by making use of the publicly available tool.

Bibliography

[1] A. M. TURING. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236):433–460, 10 1950. doi:10.1093/mind/LIX.236.433.

[2] B. G. Buchanan. A (very) brief history of artificial intelligence. *AI Magazine*, 26(4):53, Dec. 2005. doi:10.1609/aimag.v26i4.1848.

[3] J. Hendler. Avoiding another ai winter. *IEEE Intelligent Systems*, 23(02):2–4, 2008.

[4] A. Holzinger, P. Kieseberg, E. Weippl, and A. M. Tjoa. Current advances, trends and challenges of machine learning and knowledge extraction: From machine learning to explainable ai. In A. Holzinger, P. Kieseberg, A. M. Tjoa, and E. Weippl, editors, *Machine Learning and Knowledge Extraction*, pages 1–8, Cham, 2018. Springer International Publishing.

[5] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521:436–444, 2015. doi:10.1038/nature14539.

[6] A. Mosavi, S. Ardabili, and A. Varkonyi-Koczy. List of deep learning models. In *Engineering for Sustainable Future*, pages 202–214. Springer International Publishing, 01 2020. doi:10.1007/978-3-030-36841-8_20.

[7] N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso. The computational limits of deep learning, 2020. arXiv:2007.05558.

[8] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[9] TensorFlow. tensorflow/tensorflow: An open source machine learning framework for everyone - github, v2.4.3, 2021. URL `https://github.com/tensorflow/tensorflow`. Accessed on 06/10/2021.

[10] H. He. The state of machine learning frameworks in 2019, 2019. URL `https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/`. Accessed on 06/10/2021.

[11] J. Johnson. Rethinking floating point for deep learning. *CoRR*, abs/1811.01721, 2018.

[12] A. Mcgovern and K. Wagstaff. Machine learning in space: Extending our reach. *Machine Learning*, 84:335–340, 09 2011. doi: 10.1007/s10994-011-5249-4.

[13] S. Wu, G. Li, F. Chen, and L. Shi. Training and inference with integers in deep neural networks, 2018. arXiv:1802.04680.

[14] S. H. Fatemi 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, 2018. doi:10.1109/EMC2.2018.00012.

[15] J. L. Gustafson and I. Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations*, 4(2), 2017. doi: 10.14529/jsfi170206.

[16] H. F. Langroudi, Z. Carmichael, J. L. Gustafson, and D. Kudithipudi. Positnn framework: Tapered precision deep learning inference for the edge. In *2019 IEEE Space Computing Conference (SCC)*, pages 53–59, 2019. doi: 10.1109/SpaceComp.2019.00011.

[17] Z. Carmichael, S. H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson, and D. Kudithipudi. Deep positron: A deep neural network using the posit number system. *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1421–1426, 2019.

[18] 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*, 70(2):174–187, 2021. doi: 10.1109/TC.2020.2985971.

[19] G. E. C. Raposo, P. Tomás, and N. Roma. Positnn: Training deep neural networks with mixed low-precision posit. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP2021)*. IEEE, June 2021.

[20] R. Murillo, A. A. Del Barrio, and G. Botella. Deep pensieve: A deep learning framework based on the posit number system. *Digital Signal Processing*, 102:102762, 2020. ISSN 1051-2004. doi: https://doi.org/10.1016/j.dsp.2020.102762. URL `https://www.sciencedirect.com/science/article/pii/S105120042030107X`.

[21] Neuraspace, 2021. URL `https://www.neuraspace.com/`. Accessed on 05/12/2021.

[22] 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, 2020. doi: 10.1109/ASAP49362.2020.00033.

[23] A. Y. Romanov, A. L. Stempkovsky, I. V. Lariushkin, G. E. Novoselov, R. A. Solovyev, V. A. Starykh, I. I. Romanova, D. V. Telpukhov, and I. A. Mkrtchan. Analysis of posit and bfloat arithmetic of real numbers for machine learning. *IEEE Access*, 9:82318–82324, 2021. doi:10.1109/ACCESS.2021.3086669.

[24] G. Arroz, J. Monteiro, and A. Oliveira. *Arquitectura de Computadores: Dos Sistemas Digitais aos Microprocessadores*. IST Press, 3 edition, 2014. ISBN: 978-972-8469-54-2.

[25] J. von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15 (4):27–75, 1993. doi:10.1109/85.238389.

[26] A. Padegs. System/360 and beyond. *IBM Journal of Research and Development*, 25(5):377–390, 1981. doi:10.1147/rd.255.0377.

[27] L. Pyeatt and W. Ughetta. *Non-integral mathematics*, pages 239–292. 01 2020. ISBN 9780128192214. doi: 10.1016/B978-0-12-819221-4.00015-8.

[28] B. Parhami. Number representation and computer arithmetic. In H. Bidgoli, editor, *Encyclopedia of Information Systems*, pages 317–333. Elsevier, New York, 2003. ISBN 978-0-12-227240-0. doi: https://doi.org/10.1016/B0-12-227240-4/00122-2.

[29] H. So. Introduction to fixed point number representation. University of Berkeley, CS61c Spring 2006, 2006. URL `https://inst.eecs.berkeley.edu/~cs61c/sp06/handout/fixedpt.html`. Accessed on 14/09/2021.

[30] 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.

[31] S. W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*, chapter 4. California Technical Publishing, USA, 1997. ISBN 0966017633.

[32] S. Wang and P. Kanwar. Bfloat16: The secret to high performance on cloud tpus, August 2019. URL `https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus`. Accessed on 17/09/2021.

[33] J. L. Gustafson. Beyond floating point: Next generation computer arithmetic. stanford seminar, 2016. URL `https://www.youtube.com/watch?v=aP0Y1uAA-2Y&ab_channel=stanfordonline`. Accessed on 20/09/2021.

[34] J. L. Gustafson. *The End of Error, Unum Computing*. Chapman and Hall/CRC, 2 edition, 2015. ISBN: 9781482239867.

[35] W. Tichy. Unums 2.0: An interview with john l. gustafson. *Ubiquity*, September 2016. doi: 10.1145/3001758.

[36] Posit Working Group. Posit standard documentation, release 3.2-draft, 2018. URL `https://posithub.org/docs/posit_standard.pdf`. Accessed on 25/09/2021.

[37] Posit Working Group. Posit standard documentation, release 4.11-draft, May 2018. Emailed by Dr. John Gustafson on 21/06/2021.

[38] U. Kulisch. *Computer Arithmetic and Validity: Theory, Implementation, and Applications*. De Gruyter, 2013. ISBN 9783110301793. doi: doi:10.1515/9783110301793.

[39] 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, 2018. doi: 10.1109/ICCD.2018.00057.

[40] N. Team. Unum & posit- next generation arithmetic., 2019. URL `https://posithub.org/`. Accessed on 08/10/2021.

[41] K. Mercado. mightymercado/pysigmoid: A python implementation of posits and quires (dropin replacement for ieee floats) - github. URL `https://github.com/mightymercado/PySigmoid`. Accessed on 08/10/2021.

[42] S. H. Leong. Softposit, v0.4.2 - gitlab. URL `https://gitlab.com/cerlane/SoftPosit/-/tree/master`. Accessed on 08/10/2021.

[43] Stillwater Supercomputing, Inc. stillwater-sc/universal: Universal number arithmetic - github, v3.41.1. URL `https://github.com/stillwater-sc/universal`. Accessed on 08/10/2021.

[44] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[45] A. Shokry and A. Espuña. The ordinary kriging in multivariate dynamic modelling and multistep-ahead prediction. In *28th European Symposium on Computer Aided Process Engineering*, volume 43 of *Computer Aided Chemical Engineering*, pages 265–270. Elsevier, 2018. doi: https://doi.org/10.1016/B978-0-444-64235-6.50047-4.

[46] L. Bottou and Y. LeCun. Large scale online learning. In *Advances in Neural Information Processing Systems*, volume 16. MIT Press, 2004.

[47] S. Raschka. Fitting a model via closed-form equations vs. gradient descent vs stochastic gradient descent vs mini-batch learning. what is the difference?, 2021. URL `https://sebastianraschka.com/faq/docs/closed-form-vs-gd.html#fitting-a-model-via-closed-form-equations-vs-gradient-descent-vs`. Accessed on 27/09/2021.

[48] S. Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[49] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533 – 536, 1986. doi: 10.1038/323533a0.

[50] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *CoRR*, abs/1811.03378, 2018.

[51] P. Ramachandran, B. Zoph, and Q. V. Le. Searching for activation functions. *CoRR*, abs/1710.05941, 2017.

[52] S. Sharma and S. Sharma. Activation functions in neural networks. *Towards Data Science*, 6(12): 310–316, 2017.

[53] S. Albawi, T. A. Mohammed, and S. Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6, 2017. doi: 10.1109/ICEngTechnol.2017.8308186.

[54] J. Brownlee. How to visualize filters and feature maps in convolutional neural networks, 2019. URL `https://machinelearningmastery.com/how-to-visualize-filters-and-feature-maps-in-convolutional-neural-networks/`. Accessed on 03/10/2021.

[55] J. Brownlee. A gentle introduction to padding and stride for convolutional neural networks, 2019. URL `https://machinelearningmastery.com/padding-and-stride-for-convolutional-neural-networks/`. Accessed on 03/10/2021.

[56] F. Yu and V. Koltun. Multi-scale context aggregation by dilated convolutions. *CoRR*, abs/1511.07122, 2016.

[57] J. Brownlee. A gentle introduction to pooling layers for convolutional neural networks, 2019. URL `https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/`. Accessed on 04/10/2021.

[58] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[59] PyTorch. Pytorch documentation, 2021. URL `https://pytorch.org/docs/master/index.html`. Accessed on 06/10/2021.

[60] Q. Wang, Y. Ma, K. Zhao, and Y. Tian. A comprehensive survey of loss functions in machine learning. *Annals of Data Science*, 2020. doi: 10.1007/s40745-020-00253-5.

[61] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015.

[62] N. S. Keskar and R. Socher. Improving generalization performance by switching from adam to SGD. *CoRR*, abs/1712.07628, 2017.

[63] 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, 2015. doi: 10.1007/s11263-015-0816-y.

[64] Y. LeCun, C. Cortes, and C. J. Burges. The mnist database of handwritten digits. URL `http://yann.lecun.com/exdb/mnist/`. Accessed on 05/10/2021.

[65] J. Steppan. File:mnistexamples.png - wikimedia commons, 2017. URL `https://commons.wikimedia.org/wiki/File:MnistExamples.png`. Accessed on 05/10/2021.

[66] A. Krizhevsky, V. Nair, and G. Hinton. Cifar-10 and cifar-100 datasets, 2009. URL https://www.cs.toronto.edu/~kriz/cifar.html. Accessed on 05/10/2021.

[67] 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. doi: 10.1109/5.726791.

[68] J. H. Hosang, M. Omran, R. Benenson, and B. Schiele. Taking a deeper look at pedestrians. *CoRR*, abs/1501.05790, 2015.

[69] T. Dettmers. Deep learning in a nutshell: History and training, 2015. URL https://developer.nvidia.com/blog/deep-learning-nutshell-history-training/. Accessed on 06/10/2021.

[70] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[71] J. Lu, S. Lu, Z. Wang, C. Fang, J. Lin, Z. Wang, and L. Du. Training deep neural networks using posit number system. *2019 32nd IEEE International System-on-Chip Conference (SOCC)*, pages 62–67, 2019.

[72] N.-M. Ho, D. T. Nguyen, H. D. Silva, J. L. Gustafson, W.-F. Wong, and I. J. Chang. Posit arithmetic for the training and deployment of generative adversarial networks. *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1350–1355, 2021.

[73] T. Zhang, Z. Lin, G. Yang, and C. D. Sa. Qpytorch: A low-precision arithmetic simulation framework. *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS)*, pages 10–13, 2019.

[74] PyTorch. pytorch/pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration - github, v1.9.1, 2021. URL https://github.com/pytorch/pytorch. Accessed on 14/10/2021.

[75] E. Yang. Let's talk about the pytorch dispatcher, 2020. URL http://blog.ezyang.com/2020/09/lets-talk-about-the-pytorch-dispatcher/. Accessed on 19/10/2021.

[76] E. Yang. Pytorch internals, 2019. URL http://blog.ezyang.com/2019/05/pytorch-internals/. Accessed on 18/10/2021.

[77] Intel. Optimizing performance with intel advanced vector extensions - white paper, 2014. URL https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/performance-xeon-e5-v3-advanced-vector-extensions-paper.pdf. Accessed on 21/10/2021.

[78] A. Luz. Afonso-2403/pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration - posit support, 2021. URL https://github.com/Afonso-2403/pytorch. Accessed on 25/10/2021.