Simão Pedro das Neves Gonçalves Dias

Neural Networks, DeepFloat & TensorFlow Lite
Post-Training Quantization Case Study

Dissertação no âmbito do Mestrado Integrado em Engenharia Electrotécnica e de Computadores Área de Especialização em Computadores orientada pelo Professor Doutor Gabriel Falcão Paiva Fernandes e apresentada à Faculdade de Ciências e Tecnologia/Departamento de Engenharia Electrotécnica e de Computadores.

Fevereiro de 2020

1 2 9 0

UNIVERSIDADE Ð
COIMBRA

Simão Pedro das Neves Gonçalves Dias

Neural Networks, DeepFloat & TensorFlow Lite
Post-Training Quantization
Case Study

Dissertação no âmbito do Mestrado Integrado em Engenharia Electrotécnica e de Computadores Área de Especialização em Computadores orientada pelo Professor Doutor Gabriel Falcão Paiva Fernandes e apresentada à Faculdade de Ciências e Tecnologia/Departamento de Engenharia Electrotécnica e de Computadores.

Fevereiro de 2020

Abstract

In recent years, Machine Learning (ML) went through a renascence due to improvements in computing systems and computer memories. The internet also played an important role, by providing access to and aggregating large amounts of data. As this technology evolves, optimizations to its processes are receiving more attention. Traditionally, machine learning models are intense in both memory and computations during training and inference.

An optimization technique used in ML is focused on the inference phase. Models are typically trained in 32-bits, but instead of performing inference in 32-bits (operations and storage), it can be quantized to a format that uses fewer bits - this is called Post-training quantization.

Usually, the fewer bits being stored and moved around in a computing system, the less energy is consumed, thus faster computations are performed, resulting in a more efficient system, given equivalent tasks.

The goal of this study is to compare two 8-bit Post-training Quantization techniques by using two different basic models and exploit both their potentials and caveats. Both models are trained to classify handwritten numbers, the first one is focused on Fully Connected layers while the second focuses on Convolutional Layers.

One of the techniques examined adopts a novel numeric representation system and this work also explores a model to understand how the system accumulates error. In short, it is an attempt at understanding which method provides a more efficient and practical solution. In both case studies, TensorFlow Lite is concluded to be the best Post-training Quantization solution.

Keywords Machine Learning, Post-Training Quantization, DeepFloat, Systolic-array, Neural Networks

Resumo

Recentemente, Machine Learning (ML) passou por um período de renascimento devido à melhoria dos sistemas de computação e memórias dos computadores. A internet também teve um papel fundamental, permitindo o acesso e agregando enormes quantidades de dados. À medida que a tecnologia evolui, as optimizações feitas aos seus processos têm vindo a obter destaque. Tradicionalmente, os modelos de machine learning são bastante pesados em termos de memória e computações durante as fases de inferência e treino.

Uma técnica de otimização utilizada em ML é focada na fase de inferência. Os modelos são tipicamente treinados em 32-bits, mas em vez de se realizar a inferência em 32-bits (operações e gravação), esta pode ser quantizada para um formato que utiliza menos bits - um processo designado por Quantização pós-treino.

Tipicamente, quanto menos bits forem guardados e movimentados num sistema, menor será a energia consumida e mais rápidas serão as computações implementadas, resultando num sistema mais eficiente, dado o mesmo tipo de tarefas.

O objetivo deste estudo é comparar duas técnicas de quantização pós-treino de 8 bits utilizando dois modelos básicos diferentes, explorando os seus potenciais e as suas ressalvas. Ambos os modelos foram treinados para classificar algarismos escritos manualmente, em que o primeiro modelo é focado em camadas Fully Connected e o segundo é focado em camadas Convolutional.

Uma das técnicas estudadas utiliza um sistema de representação numérica novo e este trabalho também explora um modelo para compreender como este sistema acumula erro. Em suma, é uma tentativa para perceber qual dos métodos fornece uma solução mais eficaz e prática. Em ambos os casos de estudo, conclui-se que o método do TensorFlow Lite é a melhor para realizar Quantização Pós-treino.

Palavras-chave Machine Learning, Quantização Pós-Treino, DeepFloat, Array Sistólico, Redes Neuronais

Acknowledgements

This work was supported by Instituto de Telecomunicações, INESC-ID and Fundação para a Ciência e a Tecnologia, under the Projects UIDB/EEA/50008/2020, PTDC/EEI-HAC/30485/2017 and Future TPM – H2020-DS-06-2017-779391.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Objectives 1

1.3 Outline 2

2 Machine Learning Pipeline **3**

2.1 Deep Learning 3

2.1.1 Neural Networks 3

2.1.2 Layers 4

2.1.3 Training & Inference 5

2.1.4 Datasets 5

2.1.5 Accuracy 6

2.1.6 Parameters 6

2.2 Development Frameworks 6

2.2.1 TensorFlow 6

2.3 Machine Learning Computing Systems 7

2.3.1 CPU+GPU 7

2.3.2 FPGA 7

2.3.3 ASICs (TPU and other AI accelerators) 8

2.4 Machine Learning Optimizations 8

2.5 Post-Training Quantization 8

2.5.1 TensorFlow Lite 9

2.5.2 Posit & DeepFloat 9

3 DeepFloat vs TensorFlow Lite **12**

3.1 Neural Networks Models: A Case Study 13

3.1.1 Model A 14

3.1.2 Model B 15

3.2 TensorFlow Lite Implementation 17

3.3 DeepFloat Implementation 17

3.3.1 Posit & DeepFloat 17

3.3.2 Systolic Array Layer Decomposition 19

3.3.3 HDL Simulation 21

3.3.4 Simulation Setup 21

3.3.5 HDL Adaptation Process 22

3.3.6 DeepFloat Architecture 22

3.4 Inspection Tools 23

3.5 Quantization analysis 25

3.5.1 Numerical Error 25

3.5.2 Classification Error 26

3.6 Error analysis - Theoretic Model 26

3.7 Computing systems 33

4 Post-Training Model Quantization Results **34**

4.1 Model A 34

4.1.1 Classification Analysis - Accuracy Degradation Using TensorFlow & DeepFloat Simulations 34

4.1.2 Context Classification Analysis 34

4.1.3 Classification Analysis - Accuracy Degradation Using TensorFlow Lite 36
4.1.4 Context Classification Analysis 36
4.2 Model B 38
4.2.1 Classification Analysis - Accuracy Degradation Using TensorFlow & DeepFloat Simulations 38
4.2.2 Context Classification Analysis 42
4.2.3 Classification Analysis - Accuracy Degradation Using TensorFlow Lite 43
4.2.4 Context Classification Analysis 44

5 Conclusion **45**
5.1 Future Work 45

Annex A **49**

Annex B **58**

List of Figures

1 Neural Network Diagram 3
2 16-bit floating-point Decoding example 9
3 DeepFloat Decoding example 10
4 ELMA diagram 11
5 Diagram to explain this work's DeepFloat development process . 13
6 Model A Diagram 15
7 Model B Diagram 16
8 TensorFlow Lite Implementation Options 17
9 DeepFloat range distribution 18
10 DeepFloat Multiplication underflow analysis using scalars 18
11 Model A DeepFloat implementation 19
12 CONV Layer as a product 20
13 Tool GUI designed for matrix inspection 23
14 Heatmap generated from the matrix inspection tool 25
15 Multiplication Matrix on 32-bit float and 8-bit DeepFloat 27
16 Linear-to-Log Theoretic Error 29
17 Infered Multiplication Error: 3D Visualization 30
18 Application Example for Model B L1 31
19 DeepFloat Theoretic Model Multiplication Example 32
20 Model B Input and L1 Outputs (Example 1) 39
21 Model B L2 Outputs (Example 1) 40
22 Model B L3 Outputs (Example 1) 40
23 Model B Input and L1 Outputs (Example 2) 41
24 Model B L2 Outputs (Example 2) 41
25 Model B L3 Outputs (Example 2) 42

List of Tables

1 Activation Function Examples 5
2 Computing Systems Setup 33
3 Model A TensorFlow 32-bit float & 8-bit DeepFloat 34
4 Model A Context Classification TensorFlow 32-bit Float (MNIST) 34
5 Model A Context Classification 8-bit DeepFloat (MNIST) 35
6 Model A TensorFlow Lite 32-bit Converted & Optimized 8-bit Model 36
7 Model A Context Classification TensorFlow Lite 32-bit Float (MNIST) 37
8 Model A Context Classification TensorFlow Lite 8-bit int (MNIST) 37
9 Model B TensorFlow 32-bit float 38
10 Model B Context Classification TensorFlow Lite 32-bit Float (MNIST) 42
11 Model B TensorFlow Lite 32-bit Converted & Optimized 8-bit Model 43
12 Model B Context Classification TensorFlow Lite 32-bit Float (MNIST) 44
13 Model B Context Classification TensorFlow Lite 8-bit int (MNIST) 44

Abbreviations and Acronyms

AI	Artificial Intelligence. 3, 8, 10, 12, 46
ASIC	Application Specific Integrated Circuits. 8, 45
CNN	Convolutional Neural Network. 4, 10
CONV	Convolutional. 4, 13, 14, 20, 23, 38, 45
CPU	Central Process Unit. 6–8, 21
DL	Deep Learning. 3
DNN	Deep Neural Networks. 13
ELMA	Exact Log-linear Multiply-Add. 10, 12, 26–28
FC	Fully Connected. 3, 4, 6, 13, 14, 16, 20, 38, 45
FPGA	Field Programmable Gate Array. 7, 12, 46
GPU	Graphics Processing Unit. 6–8, 46
GUI	Graphical User Interface. 23
HDL	Hardware Description Language. 6, 21, 22, 46
IoT	Internet of Things. 7, 46
ML	Machine Learning. 1, 3
MNIST	Modified National Institute of Standards and Technology; database. 6, 14–17, 36–38
NN	Neural Network. i, 3, 13
ReLu	Rectified Linear Unit. 38
SoC	System-on-a-chip. 8
TPU	Tensor Processing Unit. 8

1 Introduction

Machine Learning models typically perform training and inference in 32-bits Single-precision floating-point. Recently, new number representation schemes have been proposed that can be employed to implement computing operations and storage for inference by using fewer bits - this is called Post-training Quantization [1].

In [6], a Post-training Quantization technique inspired by Posits (or Universal Numbers Type III) is proposed. It promised a method without retraining with small accuracy drops (from 32-bit float to 8-bit "DeepFloat"). This approach has also been heavily focused on hardware optimizations for machine learning and neural networks.

Other post-training quantization solutions do exist, for example Google's TensorFlow Lite. By implementing this method, the model operations and storage can also be represented using 8-bits.

Excluding the computing system architectures, the optimizations provided by the reduction of data representation size are virtually the same for both methods. This is the main argument to justify the importance of understanding which one of the techniques offered is a better solution, in terms of achieving a lower accuracy degradation, or in which situation each method has its best performance.

1.1 Motivation

Machine Learning (ML) is not new. In recent years the advances in this field have been considerable due to improvements in computer memories and computing systems [18].

As technology matures, faster and better solutions are required to continue further innovation. Currently, breakthroughs in this field are achieved by using huge data-centers and time-consuming processes that only a few have access to [18]. Recently, the paradigm has undergone continuous evolution due to open-source initiatives and competitive cloud services, that provide reduced prices [18].

Nevertheless, optimizing the machine learning pipeline is of great importance as it saves energy, development costs and time. Improvements can be made regarding development tools, and computing engines, as well as in the hardware infrastructure.

In essence, more optimized machine learning models and pipelines provide equivalent output results using more efficient processes.

1.2 Objectives

The aim of this study is to compare two post-training quantization techniques: DeepFloat and TensorFlow Lite, by using two models and the same dataset. The goal is to understand which method can provide a more efficient and practical solution, based on the obtained results.

1.3 Outline

The starting point of this work were the materials provided alongside [6], that describe the computing system used by DeepFloat in a conceptual and hardware dimension (See Annex A). The implementation resources were available on a GitHub repository.

Chapter 2 (Machine Learning Pipeline) shall introduce the topics and concepts required to understand the experimental setup and results obtained.

In order to develop a competitive and attractive deep learning solution, the hardware could become a bottleneck, so optimized hardware solutions were preferred. Because the previous work [6] provided a hardware-focused process, this thesis shall use the systolic array provided to implement part of its solution.

The experimental setup will be described in Chapter 3 (DeepFloat vs TensorFlow Lite). Two neural networks were chosen to be trained and to be quantized using DeepFloat and TensorFlow Lite. The quantization process and tools will be described in this section.

The results of the experiments shall be shown and analyzed in Chapter 4 (Post-Training Model Quantization Results). The main metric used to gauge results was model accuracy. Measurements were taken before and after the quantization process and then compared. The accuracy degradation of the DeepFloat method can be described as an error accumulation and this analysis will be further discussed in this section.

Chapter 5 (Conclusion) will reflect on the obtained the experimental results: DeepFloat quantization and TensorFlow Lite.

The last Section 5.1 (Future Work) states future experiments and studies that could be performed following this study, in order to improve the results displayed here.

2 Machine Learning Pipeline

2.1 Deep Learning

Deep Learning (DL), a sub-field of Machine Learning, has gained considerable attention in the past few years [19]. This can be attributed to the publication of several papers, achieving state-of-the-art results on ImageNet [7], an object recognition challenge.

Achievements like this have been taking place in areas such as image recognition, speech recognition, and language translation as well as other Artificial Intelligence (AI) applications [19].

2.1.1 Neural Networks

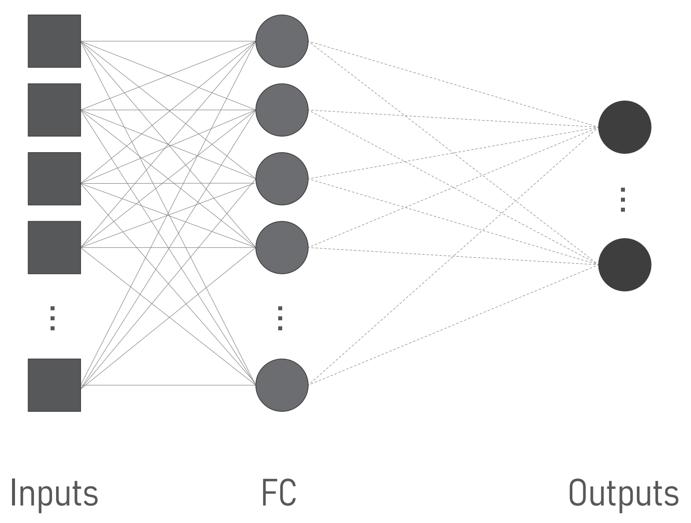


Figure 1: Neural Network Diagram
Diagram representing a Neural Network, depicting a Fully Connected layer and how it connects the network's inputs to the outputs.

Neural Networks comprise an entire class of Machine Learning architectures [14] (See Figure 1). The networks were inspired by the human brain and how each neuron is connected to the next [14]. They are usually designed using layers and activations. A layer provides a specific operation to its inputs and produces an output, becoming the next layer's input. A layer output can sometimes go through an activation function where some manipulation occurs, as it will be shown in section 2.1.2 (Activation Functions). Non-linear properties are frequently introduced to the network by this function [13]. The following are examples of activation functions: *sigmoid*, *tanh*, *softmax* and *ReLU* (Please see Table 1 for more details).

2.1.2 Layers

Neural Networks are best described by their layers. Each layer can be defined by an operation or group of operations. These are usually stacked and grouped differently, depending on the network structure. Each type of layer follows a specific strategy. On occasion, the same type of layers will be shared by different networks, but these are used in different configurations, therefore achieving different results [8].

Two types of layers are Fully Connected (FC) layers and Convolutional (CONV) layers. When a network uses Convolutional (CONV) layers it is commonly referred to as Convolutional Neural Network (CNN) and these are typically employed in image classification problems [14].

FC Layer

Fully Connected layers connect the inputs from one layer to every activation unit of the next one. These layers can be describe as performing $O = I \times W + B$ (O - Outputs; I - Inputs; W - Weights; B - Bias). The Weights and Bias are the trainable parameters. This equation can be manipulated to be performed as single product. A row of ones is added to the Input matrix and the Weight matrix is concatenated with the Bias matrix ($O = I_{Ones} \times WB$).

CONV Layer

Convolution Layers differ from the FC layers because they can only connect part of their inputs to the activation units, rather than all the inputs. Instead of Weights, CONV layers have filters or kernels. The way that the Filter slides over the inputs is defined by the Stride. The CONV layer is computed by matching the Filter with a region of the inputs, performing element-wise matrix multiplication and then adding the Bias factor [F - Filter, S - Stride, B - Bias]. After repeating this process for the entirety of the input matrix, an output is computed and it is sometimes referred to as a feature map.

Using a process called *im2col*, the CONV layer can resemble a FC layer, and the same process of converting it to a single matrix multiplication can be applied. During a CONV layer computation, pairs between the layer inputs and the convolution filter are created. *im2col* focus on these pairs, instead of computing them, the inputs can be re-arranged in a new input matrix that allows the computation to be performed as a single matrix multiplication, instead of sequential element-wise matrix multiplications by the same filter (see Algorithm 1 in Section 3.3.2).

In CONV Layers, the Filter and the Bias matrices hold the trainable parameters.

Activation Functions

Activation functions provide an operation to a layer output [13]. It adds a non--linear transformation to the layer. They are sometimes used to "turn some

neurons off", i.e. to transform the output of a layer in a way that highlights some information and reduces the impact of another.
The models studied by this work have implemented *ReLU* and *softmax* activations. In Table 1 some activation functions examples are provided.

sigmoid	$\frac{1}{1+e^{-x}}$
softmax	$\frac{e^{x_i}}{\sum_j e^{x_j}}$
tanh	$tanh(x)$
ReLU	$max(0, x)$
Leaky ReLU	$max(0.1x, x)$

Table 1: Activation Function Examples

2.1.3 Training & Inference

In machine learning, there is usually a model that represents a set of operations and their parameters, and data, that constitutes the model inputs. The model is trained to fit that data, that is, to apply a process to data that produces desired outputs, for instance, label a group of images, predict the weather, denoise an image, etc. [17]. The process can be divided into two steps, the training phase, and the inference phase [17].

During the training phase, most of the times, there is a group of data, that is labeled, and divided into subgroups - training and testing. A given model is tweaked iteration by iteration using the training data so that the output produced by the model predicts the label of the testing data. The underlying concept of such process is to produce a model by showing it new data within a perceivable similar context, so that the accuracy is the same [17]. There are models capable of dealing with unlabeled data.

The inference phase occurs when a model is shown data that was not used to perform training, and it corresponds to computations on the testing subgroup or on new data.

Inference performed using the test subgroup is sometimes referred to as the testing phase. For example, by training a model using handwritten digits, the rate at which a model correctly predicts a digit should be identical during the testing phase and inference phase.

The training methodologies can vary. Some of them include observing the performance of a new generation, that is a similar model with its parameters modified, one at a time, while others keep a record of several iterations of the models and compare them to see which one is performing best [17]. Other methodologies may employ subgroups of the training data one at the time, while the entire training data is used by others, in order to modify the model [17].

2.1.4 Datasets

Datasets are a useful resource that machine learning engineers have at their disposal. They correspond to large groups of labeled data that can be used to train and test models. For example, datasets can be labeled images, weather logs, audio files and its transcription, etc [17]. They allow the standardization of the training and testing processes when applied to the development of different models. This allows for model accuracy of different models to be compared using the same dataset [17].

The test subgroup is a subset of the dataset that is not utilized during the training process of a model. It differs from the training group because it is used to measure the performance of the model. By dividing the dataset into subgroups, problems such as overfit can be mitigated [17]. Overfit occurs when a model becomes too specialized in the training samples, thus unable to maintain its performance when new similar data is given for inference [17].

An example of a dataset is MNIST [9]. It is a collection of labeled handwritten digits, has a training set of 60.000 examples, and a test set of 10.000 samples. This dataset is usually used to train models that solve classification problems and it is used for this study.

2.1.5 Accuracy

Accuracy rate or accuracy is a very important metric for classification problems Since it measures the relation between correct predictions that a trained or in-training model has, and the labeled inputs regarding a specific dataset. It reflects the rate of correct precitions a model can produce in relation to its wrong ones, when referring to classification models and problems.

For the same dataset and classification problem, a model with higher accuracy is preferred because it may have a chance to produce a correct prediction more often.
When applying optimization processes to models so to improve their computation performance, accuracy degradation may occur, e.g. Post-training Quantization, pruning, drop-out [15]. It is best to choose an optimization process that has lower accuracy drops, in order to preserve the original model capabilities.

2.1.6 Parameters

Some model layers perform operations that require parameters such as weights and biases when referring to FC layers, and filters when referring to CONV layers. These layers' parameters are decided or adjusted during the training phase. However, they need to be assigned at the definition stage of the model. Usually, they are initialized randomly and changed during the training process [17].

2.2 Development Frameworks

Machine Learning frameworks take advantage of the fact that most of the setup and development processes between machine learning problems are the same. Some of them provide different development platforms and technologies, while others provide special inspection tools [15].

2.2.1 TensorFlow

TensorFlow, a development framework for machine learning, enables developers to create, study and reuse machine learning models. It has a strong community and a vast library of intuitive examples. Python, a programming language, is supported by TensorFlow, lowering the implementation complexity and allows the developer to integrate other tools available within the Python ecosystem. Numpy and Matplotlib are both Python libraries and together they provide a Matlab-like experience.

TensorFlow also offers tools that are designed for specific hardware setups. They translate the model into new operations either to run or to be trained in distributed systems or in other platforms.

2.3 Machine Learning Computing Systems

Machine Learning, as its name implies, needs a computation system to operate. Depending on the application, several solutions are available, which combine software and hardware. Most of the times, the software implementation needs to be optimized based on the available hardware [12].

Regarding hardware, like most computer science challenges, a Central Process Unit (CPU) system is the most accessible option, but it is also the most inefficient at performing this task. A CPU+GPU hybrid system shows significant improvement and it is the industry's standard for these types of problems [12]. Better systems can be developed using more specific hardware, such as FPGAs, that use tailored fitted implementations [3]. The most efficient solutions are implemented using ASICs, and they can be found mainly in cloud applications, IoT devices, and mobile phones [12].

2.3.1 CPU+GPU

CPU+GPU setup is the most common hardware implementation [12]. This approach can be found in conventional personal machines, servers, mobile phones and other types of applications. By using a GPU, better and faster results can be achieved, compared to a conventional CPU because they exploit the parallelization possibilities of problems of this nature.

2.3.2 FPGA

Some of the most efficient systems for accelerating machine learning computations are implemented using FPGAs [3]. Some important points about the FPGAs solutions are the following:

- FPGAs provide design flexibility and reconfigurability which cannot be achieved with ASICs [3].
- FPGA implementations can be highly specialized to a certain model's needs, providing an advantage compared to the CPU+GPU.
- FPGA implementations imply lower implementation prices upfront when compared to ASICs but higher than GPU.
- There are cloud services available that offer competitive options using this technology.

2.3.3 ASICs (TPU and other AI accelerators)

Tensor Processing Unit or TPU is developed by Google. It is an ASIC designed to run Machine Learning models on the cloud [15].

Some mobile phones and specific applications already use a custom System on a Chip (SoC) for AI, instead of the traditional CPU+GPU configuration, in order to save energy and optimize the pipeline.

2.4 Machine Learning Optimizations

Due to the complexity regarding computation and inefficiency of the methods used in machine learning, a priority at this moment is to optimize them. Several stages of development can be tackled to improve efficiency, namely:

- **Training**: Since the training algorithms affect model performance, some methods can achieve better results for the same model and dataset.
- **Model**: The model architecture affects its potential. By choosing better models or redefining them, the same machine learning problem can be solved more efficiently.
- **Inference**: The inference phase can be optimized by reducing the model's size or redefining the model. Alternatively, optimization can be achieved with the employment of more efficient and faster hardware.

2.5 Post-Training Quantization

An optimized computing system strives to get the same output as before (or an equivalent one) requiring less power while being faster.

This achievement can be obtained with better fabrication technologies, and better design techniques.

By using more bits to represent a number in a digital system, the numerical precision is increased. However, this can affect drastically the performance (power, throughput) of the system when compared to an equivalent one that uses fewer bits. There is a trade-off between the number of bits that a system uses and its performance.

Machine Learning Models are typically trained using a 32-bit single-precision floating-point format to represent numbers. It has been shown that by using higher floating-point representation, some training methods and algorithms converge better [15].

However, after a model has been trained, the higher precision provided by that representation may be unnecessary, since a lower bit model can achieve similar results to the 32-bit, using fewer bits [15].

The process of taking a model and changing the number of bits or its codification after it has been trained is called post-training quantization [15].

A valid way to optimize machine learning inference is by quantizing the model to a lower-bit representation. The model's size is by definition reduced and the computations during inference are also optimized because fewer bits are being moved/stored. With fewer bits per number, smaller memories can be used, and so can smaller data buses, etc. By using smaller-bit models, the inference hardware can also be optimized.

Several post-training techniques have been proposed and some machine learning frameworks also provide solutions.

2.5.1 TensorFlow Lite

TensorFlow Lite is an addition to the TensorFlow ecosystem (See 2.2.1 TensorFlow), that allows a model to be transposed to a lower-bit representation, therefore enabling it to be easily used in low-power devices and devices that use smaller data types. Some computing systems only support integer operations and it is also used to port a model to those cases. TensorFlow can aid during training to optimize the parameters and models for those devices. However, it can also be used for post-training quantization.

2.5.2 Posit & DeepFloat

Unums or Universal Numbers are a binary representation system for real numbers that also encompass arithmetic operations. They were proposed by John Gustafson as an alternative to IEEE 754 system [5] (See Figure 2). There are different types of Unums. Type III includes Posits, they are hardware optimized, being able, for instance, to maintain the same bit size.

Posits have 4 main parts: a sign bit, a regime, an exponent and a fraction. All parts have a variable number of bits excluding the sign bit. Their variation is defined by $< n, es >$, n being the number of bits and es the maximum number of bits dedicated to the exponent (See Annex B for its specification).

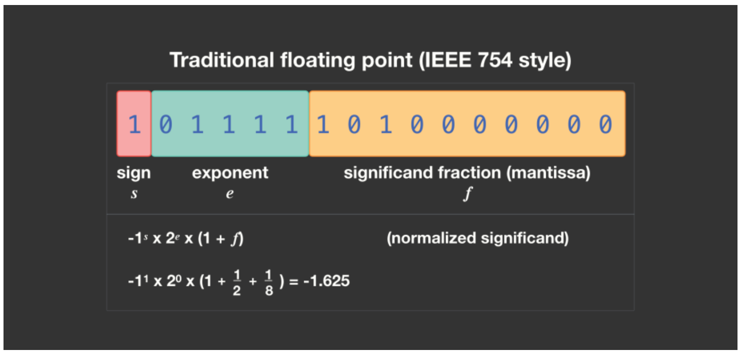

Traditional floating point (IEEE 754 style)
1 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0
sign
s
exponent
e
significand fraction (mantissa)
f
-1^s x 2^e x (1 + f)
(normalized significand)
-1^1 x 2^0 x (1 + 1/2 + 1/8) = -1.625

Figure 2: 16-bit floating-point Decoding example
Source: Making floating point math highly efficient for AI hardware [5]

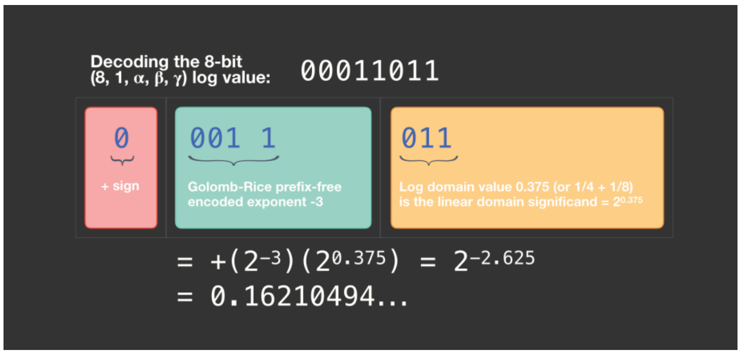

Decoding the 8-bit
(8, 1, α, β, γ) log value:
00011011
0
+ sign
001 1
Golomb-Rice prefix-free encoded exponent -3
011
Log domain value 0.375 (or 1/4 + 1/8) is the linear domain significand = 2^0.375
= +(2^-3)(2^0.375) = 2^-2.625
= 0.16210494...

Figure 3: DeepFloat Decoding example
Source: Making floating-point math highly efficient for AI hardware [5]

8-bit (8, 1, 5, 5, 7)log or DeepFloat is a number format and computing method proposed by Jeff Johnson in [6]. In this paper, it is suggested that this representation can be a "drop-in replacement for IEEE 754 binary32 single-precision floating-point via round to nearest even for CNN inference on ResNet-50 on ImageNet" [6].

This format uses a posit-like encoding. The main difference between posit (8,1) and DeepFloat is that the faction part in the DeepFloat representation corresponds to the $log_2(x)$ of the equivalent Posit number (See Figure 3). By following the techniques described in this paper, it is suggested that one can perform inference using a model, thus achieving similar accuracy results to the original one, while using less power and area. To do so, the computing engine or AI accelerator implements a special processing element that performs the multiply-accumulate operation using a sequence of special techniques - ELMA.

ELMA takes advantage of the complexity of multiplications versus additions and performs the multiplication as an addition because the fraction part of the number is a *log* number. The additions and operations of accumulation are stored using 16 bits so to preserve precision. The mapping between linear-to-log and log-to-linear is performed using LUTs and they have a fixed error (See Figure 4).

By transposing a model from a Float representation system to (8, 1, 5, 5, 7)log (DeepFloat), one is post-training quantizing the model. In the afore mentioned paper, this representation was not used to train the model.

In order to quantize a model using DeepFloat, following this method, one needs to take a pre-trained model and perform direct translation of the original 32-bit parameters to DeepFloat. For inference, the same layer operations must be computed, using the ELMA system instead.

Despite being similar representation systems, DeepFloat does not follow all Posit specifications. It is noteworthy, that DeepFloat can be perceived as a Posit version because it uses a similar codification structure that is computed differently with an expected similar behavior. Both versions have a higher concentration of represented values surrounding zero and most machine learning models work in this region of the spectrum.

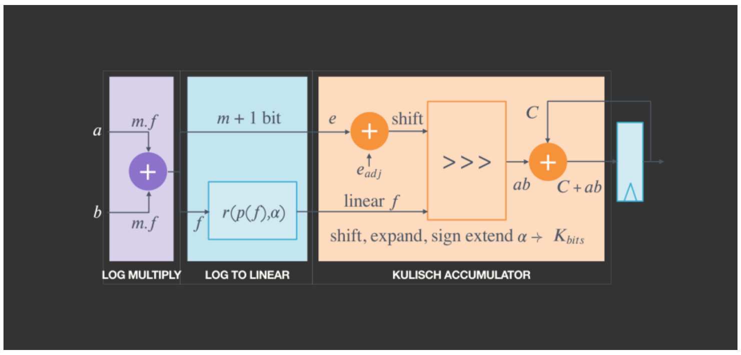


Figure 4: ELMA diagram

DeepFloat Processing Element, Perform the operation of Multiply–accumulate. The multiplication is performed in the log domain. The Addition is performed in the linear domain. *Source: Making floating-point math highly efficient for AI hardware [5]*

3 DeepFloat vs TensorFlow Lite

A novel way of performing Post-training quantization of Machine Learning models is proposed in [6]. The number system itself is inspired by Unums and it is referred to as DeepFloat or (8,1,5,5,7)log. It also proposes this new arithmetic architecture denominated Exact Log-linear Multiply-Add (ELMA).

The author in [6] indicates, that both strategies in conjunction offer a solution that 1) provides low accuracy drops, 2) has a seamless integration with the original float 32-bit model 3) offers a low-power and low-area or equivalent ones to a 8-bit integer architecture. Given that DeepFloat does not require a quantization process, the numerical system's transposition is itself the quantization process.

The previously mentioned study performs the power and area analysis using a systolic array architecture that uses the ELMA processing elements and implements the accuracy study using the addition and multiplication systems inside ELMA. These elements are programmed in an FPGA and used both as a kernel and a backend to a Machine Learning Framework.

To explore this technology and improve upon the obtained results, an AI accelerator that uses this architecture would facilitate the distribution and implementation of these systems. However, the performance of this technique needs to be compared against other solutions in order to justify its implementation.

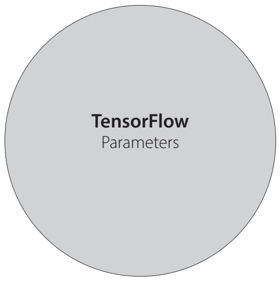
One of the objectives of the current work is to understand whether or not DeepFloat is a good representation system to be used in an AI accelerator for inference. The said AI accelerator should also provide a better power performance compared to traditional methods, as to justify its development. This type of optimizations is hinted in previous research when using the DeepFloat method [6]. Nonetheless, the accuracy degradation needs to be analyzed to justify the efficiency gains.

Following the original research [6], DeepFloat shall be used as a drop-in replacement for the 32-bit model, what constitutes a Post-Quantization process.

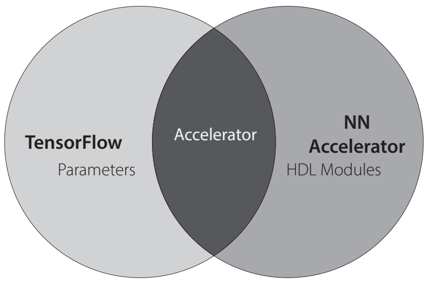
Two quantization processes shall be analyzed and compared: TensorFlow Lite's 8-bit integer quantization and DeepFloat's 8-bit (8, 1, 5, 5, 7)log. If the DeepFloat method provides worse results than the integer method, one can consider that the integer method still holds good results, therefore the DeepFloat approach may not bring forth a significant improvement that justifies its development. If the opposite is verified, the accelerator's development can be justified and further models can be used with the accelerator, to further validate the method. However, in order to implement the accelerator, a compiler and a driver need to be developed as well, which may imply that undertaking such efforts might not be justified.

The starting point of the evaluation method has been a model trained in 32-bits floating-point using Tensorflow. A compromise for not implementing a full function accelerator to study this system is to use hardware simulations for the main compute system. This allows a more efficient validation process, since it proved to be faster and easier to iterate using software than hardware designs. To do so, and to validate the hardware design process as well, hardware simulations have aided the validation process.

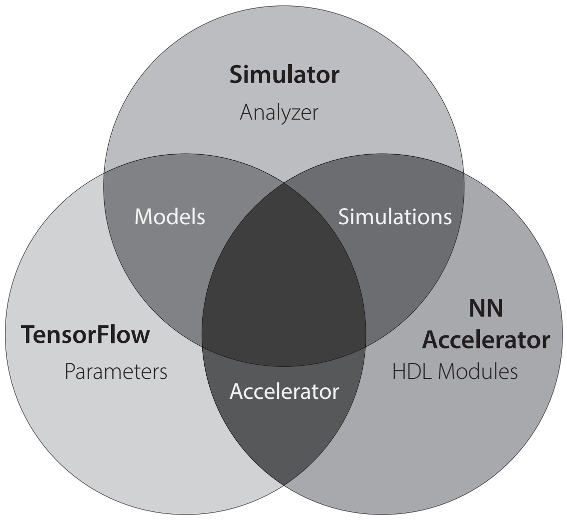
The starting model shall be considered the original model and each quantized model shall be regarded as derived from it, given each method applied.



The starting point where there is only a model in the TensorFlow ecosystem



The goal is to validate an accelerator for performing inference instead of using the traditional CPU+GPU platforms



The development environment is composed by Tensorflow, the accelerator and supports simulations that aid the validation process

Figure 5: Diagram to explain this work's DeepFloat development process

The original model has been provided with a measurable accuracy and each quantization process will affect it differently. All things consided, the accuracy of a model is key and ideally the quantization method should strive to provide the same accuracy as the original model. However, this is not always the case and such degradation can enlighten on how to determine if a method is better than other (See Figure 5).

3.1 Neural Networks Models: A Case Study

Neural Network (NN) and Deep Neural Networks (DNN) are layered and several models can be designed using the same type of layers, although using different sizes and placed differently.

The most common layers applied are Fully Connected (FC) and Convolutional (CONV) layers. Almost every neural network model utilizes these layers, or the principles of these layers, and applies them in a specific configuration.

The models chosen to evaluate the post-training technique focus on these types of layers. The results of a model are more dependent on to the way it was trained and the dataset used, than the type of layers it is composed. However, the type of operations will be same if the same layer is used by them. The underlying belief backing this approach is that if these layers integrated into a model degrade with a quantization method, they will also degrade when applied to another model using the same method. Also, if a model uses some of these layers, at least that section of the model will tend to degrade according to a specific method. This approach regarding the given models is speculative but one can contemplate if they represent the problem.

Bearing this in mind, the chosen models must present the following characteristics:

- They must be shallow
- They must be from the TensorFlow documentation and community
- They must use a simple dataset

The two selected models have been denominated Model A and Model B, so to ease their identification during this study. These were example models provided by the tutorial documentation from TensorFlow [11]. They have been trained using MNIST and this process has also been provided by example source code [11]. The focus of this study has been the output of the training process, which corresponds to two 32-bit models trained in 32-bit single precision floating point (IEEE 754 standard).

The Model A is focused on the FC layers and Model B focus on CONV layers to cover the more common types of layers used in NNs.

3.1.1 Model A

Model A is a linear-inference based model. It has been trained using the MNIST dataset[9] and it is composed of one fully-connected layer (W [784, 10] b[10] softmax). This model is an example model provided by the TensorFlow official documentation [11] and it is trained for 2.000 iterations following the example code (See Figure 6).

To implement the FC layers using the DeepFloat Post-training Quantization technique, it has been subjected to some adaptation. The first step of this process has been to extract both the 32-bit floating-point parameters from Model A and 32-bit integer test-set from MNIST.

The (8,1,5,5,7)log model has been obtained by taking the original 32-bit float model and mapping its parameters to (8,1,5,5,7)log, using an HDL module that has been adapted from the source code provided from the DeepFloat GitHub repository [6].

This module takes 32-bit Float inputs and maps them to a (8,1,5,5,7)log representation. This operation has been performed by simulating the HDL module using Vivado® Simulator - xsim. The inputs have also been mapped to a (8,1,5,5,7)log representation.

The (8,1,5,5,7)log inference is performed using an adapted HDL systolic array module that is available in the DeepFloat repository[6]. To do so, all of the original model operations were decomposed in matrix multiplications.

For the numerical precision analysis, the model has been run in traditional TensorFlow fashion, 32-bit float, and 8-bit (8,1,5,5,7)log. The inputs, the parameters, and the outputs were saved in both situations and then compared against each-other.

For the classification analysis, the outputs from both models' representations were compared against the original labeled data and to each-other. The results of the mentioned comparison are stated in Chapter 4.

The outputs, on the original model use *softmax*, and the model was trained using this activation. During the quantization process, this activation has not been used, however, the classification results have not been affected (See Section 2.1.2 for more information about *softmax*).

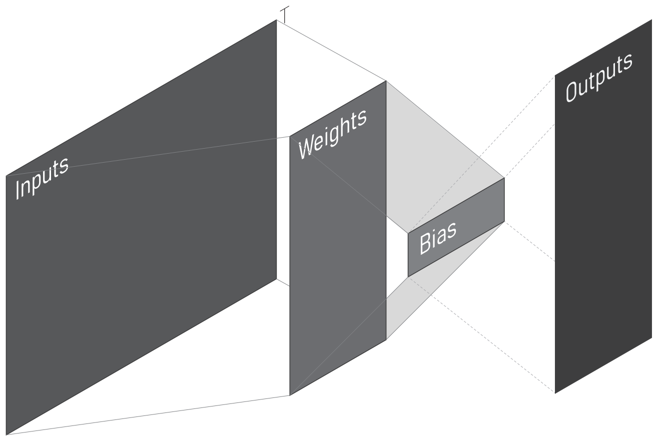


Figure 6: Model A Diagram
(See Section 2.1.2 Layers - FC Layer)
Inputs [batch, 28 × 28 = 784]
Weights [784, 10] Bias[10]
Outputs [batch, 10]

3.1.2 Model B

The Model B has five layers, with the first tree being convolution layers (L1: X [batch, 28, 28, 1], stride 1, W1 [5, 5, 1, 4], B1 [4] *ReLu*; L2: Y1 [batch, 28, 28, 4], stride 2, W2 [5, 5, 4, 8], B2 [8] *ReLu*; L3: Y2 [batch, 14, 14, 8], stride 2, W3 [4, 4, 8, 12], Y3 [batch, 7, 7, 12] $\Rightarrow$ reshaped to YY [batch, 7×7×12] *ReLu*). The final two are fully connected layers (L4: W4 [7×7×12, 200], B4 [200]; L5: Y4 [batch, 200] *ReLu*; L5: W5 [200, 10], B5 [10], Y [batch, 10] *softmax*). It has also been trained using the MNIST dataset[9]. This model is also an example model provided by the TensorFlow official documentation[11] and it was trained for 2.000 iterations following the example code (See Figure 7).

Similarly to what succeeded with Model A, the first step of this process has been to extract both the 32-bit floating-point parameters from Model B and 32-bit integer test-set from MNIST.

The adaptation process for the DeepFloat quantization for the FC layers is also very similar to Model A but it uses a different approach for the CONV layers.

Layers 1, 2, 3, 4 have *ReLu* activations and the last one applies *softmax*. In the DeepFloat version of Model B, the *ReLu* activations are implemented but the *softmax* is excluded, as it occurred with Model A.

The numerical precision analysis and classification analysis are the same as in Model A (See results on Chapter 4 Post-Training Model Quantization Results).

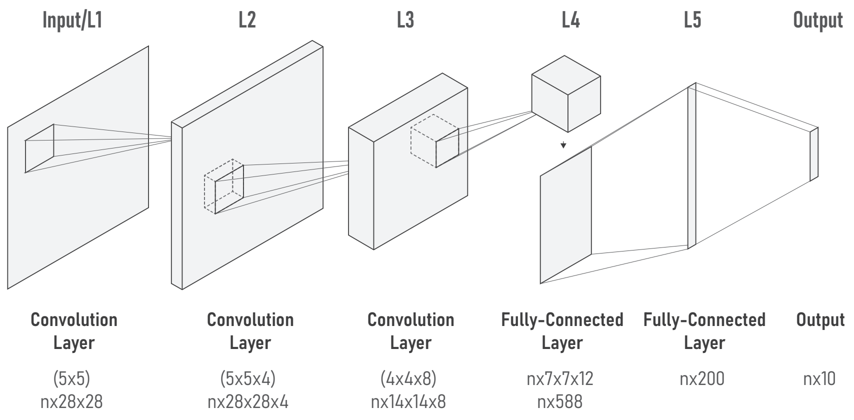


Figure 7: Model B Diagram
Inspired by AlexNet Diagram

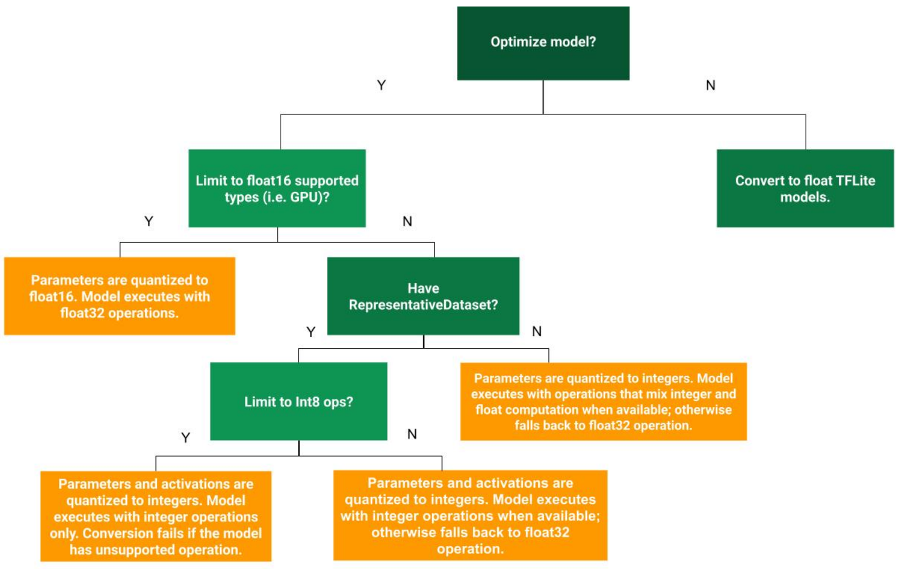
Both models can be considered small networks, and these types of networks tend to be more affected by Post-training Quantization than deeper ones, since smaller networks have higher accuracy degradation. By choosing these type of models, one can avoid model resilience due to model plasticity and noise reduction capabilities, masking the accuracy degradation. By choosing a shallower model, the idea is that the results are more focused on each quantization technique rather than on each model's properties, allowing for a better comparison study.

In both quantization processes, the same 32-bit floating-point model has been quantized, i.e., the starting point for both methods takes on after the same model and parameters. Although the implementation of both solutions is not equivalent, the size of each model and activations before and after quantization is the same in both situations, being the accuracy degradation the aim of this study - one can achieve the same "model compression". Any gain obtained by reducing the size of the model and feature maps is virtually the same in both approaches, although in reality, this does not occur due to differences in implementation architectures. The computation latency, resource utilization, and computational architectures are not to be considered at the moment.

3.2 TensorFlow Lite Implementation

The official description says that *TensorFlow Lite is an open-source deep learning framework for on-device inference* [15]. It integrates with TensorFlow very easily and provides a post-training quantization feature.

As previously stated, this study aims to compare the optimization results using 8-bit integers for inference and DeepFloat.



Optimize model?
Y
N
Limit to float16 supported types (i.e. GPU)?
Convert to float TFLite models.
Y
N
Parameters are quantized to float16. Model executes with float32 operations.
Have RepresentativeDataset?
Y
N
Limit to Int8 ops?
Parameters are quantized to integers. Model executes with operations that mix integer and float computation when available; otherwise falls back to float32 operation.
Y
N
Parameters and activations are quantized to integers. Model executes with integer operations only. Conversion fails if the model has unsupported operation.
Parameters and activations are quantized to integers. Model executes with integer operations when available; otherwise falls back to float32 operation.

Figure 8: TensorFlow Lite Implementation Options

Source [15]

In order to implement the TensorFlow Lite post-quantizing 8-bit int version, the tool has been configured so to optimize the model, using a representative dataset, thus limiting its operation to 8-bit integers (See Figure 8). The TensorFlow lite 8-bit quantization approximates floating point values using $real\_value = (int8\_value - zero\_point) \times scale$ [15].

The official documentation provides an example source code for this process. That script has been adapted to use the MNIST dataset, Model A and Model B. After the optimization process, the inference has been performed using the test set and the results compared against the original 32-bit model.

3.3 DeepFloat Implementation

3.3.1 Posit & DeepFloat

Posit has a higher distribution concentration around zero. In neural networks, given a set of parameters and activations, there is often a higher concentration of values near zero. It can be perceived as a sign of a well-trained model. These qualities reinforce the idea that DeepFloat should be able to provide more resolution to computations in these areas, theoretically meaning, it has the potential of having more accuracy than traditional representation schemes, given the same number of bits. These observations have been made for Posit. The goal with DeepFloat is to have identical precision as in the 32-bit Float model.

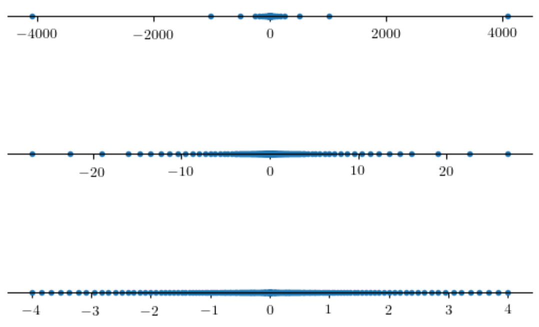


Figure 9: DeepFloat range distribution

It has been shown that some problems that arise during the Post-training Quantization using Posit have cheap solutions.

One of them, as in other codification schemes, is underflow. For Posit and DeepFloat this is common because most of the computations are being placed around zero. To a lesser degree, overflow is also a problem. To solve such problems, a scalar can be defined to expand or compress a set of numbers along the DeepFloat range, minimizing the effects of overflow or underflow. In a given layer, should the difference between the lowest and highest number be higher than the range of the Posit number-set, there is a scalar that can be defined to scale them to a representable size.

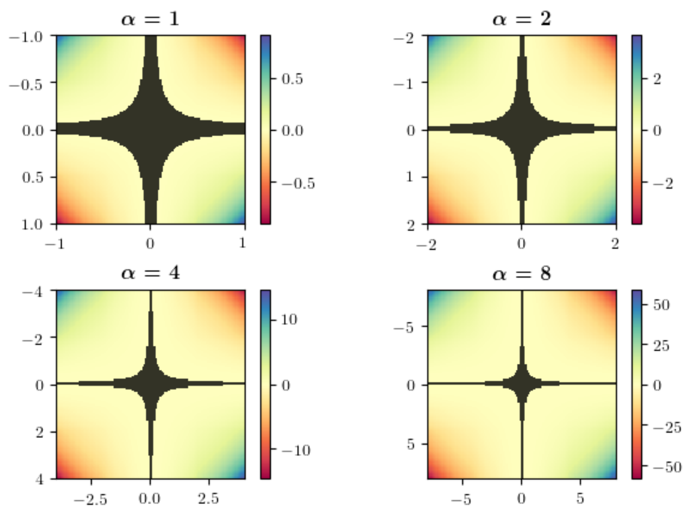


Figure 10: DeepFloat Multiplication underflow analysis using scalars $\alpha a \times \alpha b = \alpha^2 c$

If the activations of a layer are being targeted by underflow, and overflow is nonexistent, that layer can also be scaled up.

The main problem arises when there is overflow and underflow at the same time. For inference purposes, overflow is less present because the operations are being performed more often around zero. When using scalars, the original result can be obtained by multiplying the output by the inverse of the scalar.

Moreover, to implement Posit/DeepFloat to quantize operations, the original model and the quantized model need to operare quite equally to minimize errors. When a quantized model has either underflow or overflow problems, differences are introduced. By using scalars, the underflow/overflow is overcome. Nonetheless, complexity is being added to the inference process at the expense of time and energy, what should be noteworthy.

Using Posit/DeepFloat for training, these problems would exist, though becoming less relevant because they are intrinsic to the model, due to the training process.

3.3.2 Systolic Array Layer Decomposition

Figure 11 shows how Model A has been fed into the systolic array. Since the systolic array performs matrix multiplications, the FC layers were converted into matrix multiplications, instead of being performed as a matrix multiplication and an addiction. In order to perform inference on the testing group, the inputs have been grouped in a matrix with 32 columns, reducing to 313 the number of multiplications, instead of 10.000 - $ceil(\frac{test-set=10.000}{systolic-size=32}) = 313$.

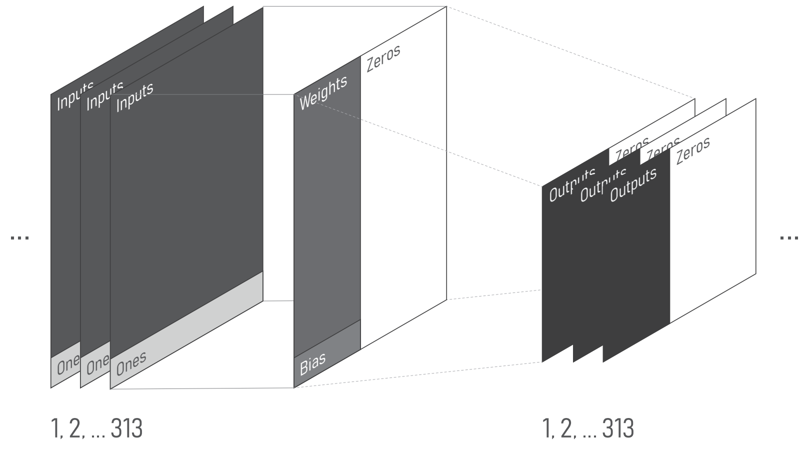


Figure 11: Model A DeepFloat implementation
FC Layer as a product

Model B is composed by CONV layers and to optimize its computations for the systolic array, they are converted into products. Figure 12 shows how the *im2col* transformation has been performed. The sliding window is overlaid and captures a subsection so to create an input matrix, thus allowing the CONV layer to be computed as a matrix multiplication. For this model, some output matrices are bigger than 32 by 32, so the input matrices have been divided into several smaller matrices and reconstructed at the end.

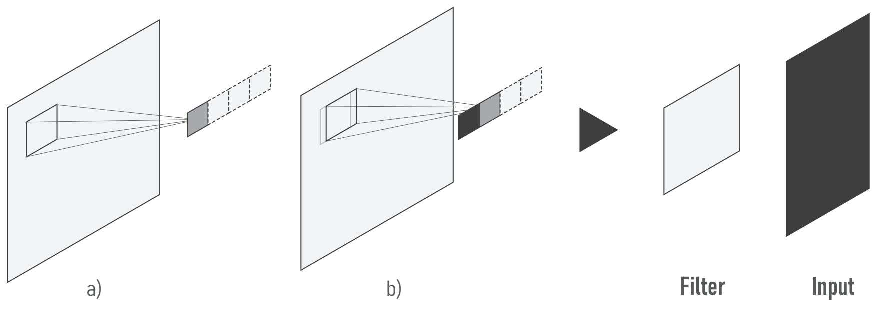


Figure 12: CONV Layer as a product
a) A window with the filter size is placed over the original layer input and the overlapping section is stored in the new input matrix.
b) The sliding windows moves to the next section using stride

Algorithm 1: Im2Col Adaptation (CONV to FC)

```
Input: $sizei$, $depth$, $stride$, $filtersize$, $layer$
Result: Translated Input

$pad = \frac{filterSize - stride}{2}$
$finalSize = \frac{ceil((sizei - filterSize + pad + floor(pad))}{stride + 1}$
$result = ones((filterSize \times filterSize, k, finalSize, finalSize))$
$halfFilter = floor(\frac{filterSize}{2})$
$padDim = ceil(sizei + pad + floor(pad))$
for $k \leftarrow 0$ to $depth$ do
    for $i_c \leftarrow stride$ to $floor(padDim - pad)$ by $stride$ do
        for $j_c \leftarrow stride$ to $floor(padDim - pad)$ by $stride$ do
            for $i \leftarrow -halfFilter$ to $filterSize - halfFilter$ do
                for $j \leftarrow -halfFilter$ to $filterSize - halfFilter$ do
                    result[ (j+halfFilter)*filterSize+(i+halfFilter), k,
                      $floor(\frac{j_c}{2-1})$, $floor(\frac{i_c}{2-1})$]
                    = layer[(j$_c + j) * padDim + (i_c + i), k$]
                end
            end
        end
    end
end
```

3.3.3 HDL Simulation

While developing HDL, one has tools to aid the implementation process. This technology is not compiled to be used in a CPU but its digital behavior can be simulated using a testbench. A testbench is a special module that is designed to produce stimuli for an HDL module, working both as a debug and testing tool. It gives one insight about module behavior. The results from the use of a testbench are known as simulations. After a module has been tested and validated, a similar technique can be used but instead of validating and testing the module, it aids in the development process of the rest of the system.

The original source-code from DeepFloat provides a systolic array that performs matrix multiplication. For this work, the simulations have been used to validate the model performance when using DeepFloat [6]. The systolic array is simulated using a testbench - a special module that can feed and read outputs from the systolic array. The inputs and outputs can be text files that function as memories. By changing the text files, different matrix multiplications can be defined. The systolic array is configured as 32 by 32 grid, in order to allow the multiplication of matrices than are 32 by m and n by 32. This configuration has been selected to allow a faster implementation of Model A, and it can be considered to be a good trade-off regarding size and parallelization.

Model A and Model B have been simulated using the systolic array to perform all computations for the DeepFloat version of the quantized model, allowing the inference accuracy to be measured without the development of an accelerator architecture.

3.3.4 Simulation Setup

To simulate the HDL modules, it has been employed *xsim*, a tool provided along Vivado by Xilinx. The HDL modules undergo an elaboration and compilation phase and the input files and outputs can be provided by the tool. Vivado generates a simulation script that allows for the toolchain to be called from another program. This has proven itself to be a very powerful option. To automate the simulation process the logic is programmed using Python, a technology traditionally used in machine learning. Doing so, allows both the hardware element and the Machine Learning development environment to work together in a more streamlined way. The underlying concept is that a Python script can be used to generate the text files that the simulations require, call the simulation process and in the end analyze the results. This setup has modularity but also abstracts the hardware modules after they have been validated. Furthermore, the Python script can be used to control the pipeline, asides the fact that by using software, one can iterate faster than by designing/re-designing a pipeline in HDL. This may be of great advantage because the systolic array demands data to be fed in a very specific manner.

The text files can be perceived as memories and the Python script as the pipeline, implemented in software. The computations have been performed by simulating the hardware that represents the systolic array. In essence, the system operates using software but represents hardware.

The testbench, in this case, has been used as a computation method rather than as a validation one.

3.3.5 HDL Adaptation Process

The Deepfloat GitHub repository has provided the source code to reproduce the results of the article. There is a folder with the HDL modules that describes a Deepfloat systolic array. This systolic array has also a testbench validating it. The systolic array has been used in the original research only for the area and energy analysis. However, in this study it has been adopted for computation purposes.

There are also modules that translate 32-bit floating-point to the format used by DeepFloat (a Posit-like format) and back again by implementing them as testbenches. These modules have also been adapted for this study by using text files as memories.

The adaptation process has consisted of taking the testbenches, but instead of generating data to validate the modules by simulation, they have been fed from text files and have saved the produced outputs to another text file, computed through simulation. As for the translation modules, they have taken the numbers to be translated from a file and a file has been created with the translation, for example. They have taken the parameters of a module in 32-bit float and produced a file with the parameters in DeepFloat. For the systolic adaptation, the same concept has been employed, but the inputs can be considered as two matrices that shall be multiplied, since they both have been stored in files instead of having been generated so to validate the systolic array. The output is the product of both matrices, stored in a text file.

For proper systolic array use, the matrices need to be fed in a specific manner because of the way it was designed (See Feeding - Algorithm 3).

By using text-files, inspection tools can be easily developed, allowing faster debugging processes.

3.3.6 DeepFloat Architecture

Not all matrices are 32 by 32, so when they are smaller than that, they must be zero-padded. An algorithm is responsible for padding (See Algorithm 2).

Algorithm 2: Padding

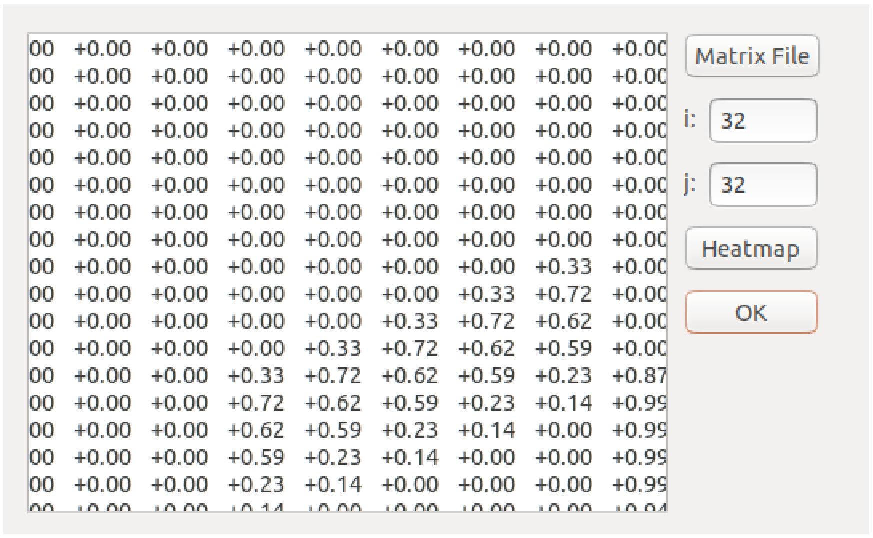
```
Input: $inputfile$, $outputfile$, $\mathrm{in}_n$, $in_m$, $tile = 32$
Result: Padded Matrix File (32 by 32)

$paddedmatrix$ = table of zeros with size tile by tile
$inputmatrix$ = load($inputfile$)

for $i \leftarrow 0$ to $in_n$ do
    for $j \leftarrow 0$ to $in_m$ do
        $paddedmatrix = inputmatrix[j \times tile + i]$
    end
end
write($outputfile$) = $paddedmatrix$
```

The output matrix from the systolic array cannot be larger than 32 by 32. Considering this, to perform multiplications with larger outputs, the input matrices need to be separated into smaller input matrices.

When feeding the matrices to the systolic array, for the first and last 32 cycles of clock, padding (or placement of zeros) is required to ensure that the multiplication is being performed correctly (See Algorithm 3).

3.4 Inspection Tools

During the implementation stage, it has been of utmost importance to understand if all elements are functioning properly. The debugging process usually consists of examining the inputs and outputs of each layer while the model is being fed. Considering that all the inputs and outputs from the simulations are stored in text files, this task can be accomplished by several procedures (See Figure 13). The inspection can be implemented automatically in a script, but since this solution may vary according to each situation, it has been deemed appropriate to use a generic GUI to run an inspection script on the background, able to achieve the same results, thus providing flexibility. This inspection tool has been implemented using wxpython and allows one to select a text file and choose its corresponding matrix dimensions. It allows one to inspect both the size and heatmap of a matrix. The heatmap can provide very useful debug information, namely in outputs with visually important information, such as with CONV layers.



```
00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.0
00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.0
00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.0
00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.0
00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.0
00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.0
00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.0
00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.0
00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.33  +0.0
00  +0.00  +0.00  +0.00  +0.00  +0.00  +0.33  +0.72  +0.0
00  +0.00  +0.00  +0.00  +0.00  +0.33  +0.72  +0.62  +0.0
00  +0.00  +0.00  +0.00  +0.33  +0.72  +0.62  +0.59  +0.0
00  +0.00  +0.00  +0.33  +0.72  +0.62  +0.59  +0.23  +0.87
00  +0.00  +0.00  +0.72  +0.62  +0.59  +0.23  +0.14  +0.9
00  +0.00  +0.00  +0.62  +0.59  +0.23  +0.14  +0.00  +0.9
00  +0.00  +0.00  +0.59  +0.23  +0.14  +0.00  +0.00  +0.9
00  +0.00  +0.00  +0.23  +0.14  +0.00  +0.00  +0.00  +0.9
```

Figure 13: Tool GUI designed for matrix inspection

Algorithm 3: Feeding

```
Input: $inputfile$, $outputfile$, $in_n$, $in_m$
Result: Feed Matrix File ($in_n$ by $in_m$)
------------------------------------------------------------
$inputmatrix = \text{load}(inputfile)$

for $n_{out} \leftarrow 0$ to $in_m + in_n - 1$ do
    for $m \leftarrow 0$ to $in_m$ do
        // Innit
        if $n_{out} \leq m$ then
            if $m \leq n_{out}$ then
                $outputfile.\text{Add}(inputmatrix[\text{in}_m \times (\text{n}_{out} - m) + m])$
            else
                $outputfile.\text{AddZero}$
            end
        end
        // Middle
        else if $(n_{out} > m)$ and $(n_{out} < in_n)$ then
            $outputfile.\text{Add}(inputmatrix[\text{in}_m \times (\text{n}_{out} - m) + m])$
        end
        // End
        else if $n_{out} \geq in_m$ then
            if $m > (n_{out} - in_n)$ then
                $outputfile.\text{Add}(inputmatrix[\text{in}_m \times (\text{n}_{out} - m) + m])$
            else
                $outputfile.\text{AddZero}$
            end
        end
    end
end
```

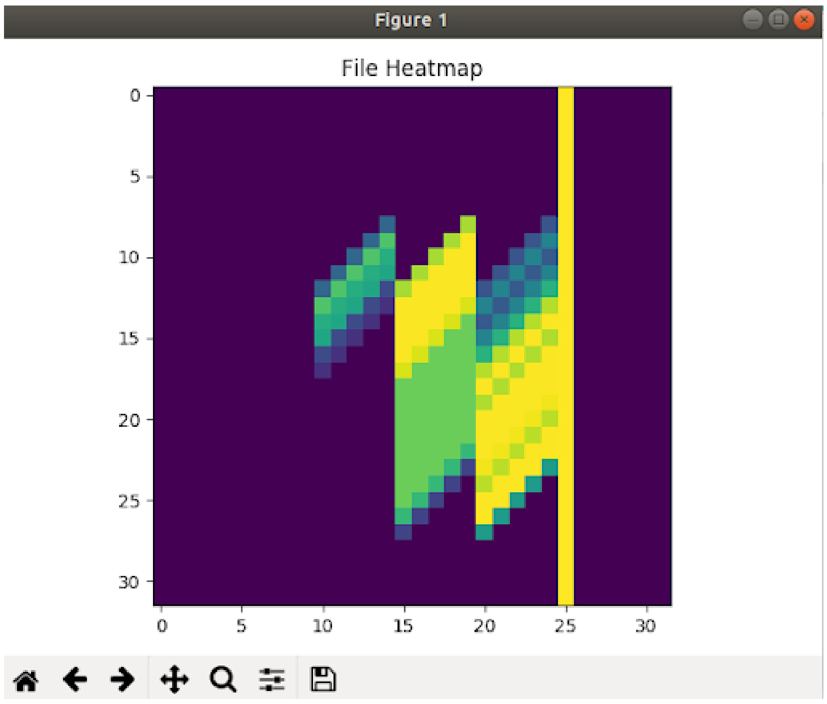


Figure 14: Heatmap generated from the matrix inspection tool
In this example, a column of ones can be easily identified around index 25.

3.5 Quantization analysis

For simplification, classification models have been used as examples for the definition of these concepts.

Due to the quantization process, one can define a numerical error associated with the process of mapping the parameters of a model from one domain to another. During the inference phase, there have been results expected from each layer's operations that diverge from those obtained when using the original model. Also, the classification of an input can suffer alterations, depending on the quantization method, thus resulting in different model accuracy.

Considering this, error metrics can be defined as associated with the quantization process: Numerical Error and Classification Error.

3.5.1 Numerical Error

The numerical error can be defined as the difference between the computations from the original model and the quantized model. The DeepFloat quantization method has strived to represent and map the original information, having used an 8-bit format. In this case, it is suitable to use a numerical error. The TensorFlow Lite method does not attempt to perform a direct translation of the original range, so a numerical error is not appliable.

3.5.2 Classification Error

The classification error can be defined as the difference between the original model classification and the quantized model classification given an input. For example, if the classification for a given input is identical in the original model and in the quantizatized model, the classification error is 0. The classification error has been shown to be different from the accuracy, because the accuracy concerns the performance of all labeled groups, regarding classification problems. The classification error can either be used to examine at each test input or it can be used to observe groups of labeled inputs.

The error analysis, other than the accuracy, has provided an insight into the degradation of the model and can denote potential problems or validate the processes used.

3.6 Error analysis - Theoretic Model

During this study, there has been the need to predict the DeepFloat quantization outcome, isolated from its direct implementation. This allows: 1) The validation of each model's simulations and implementations, as there is a prediction to confront besides the original model results; 2) Understanding how DeepFloat performs in other simulations without its empiric implementation. This understanding can aid the interpretation of the results, besides looking at them as facts. This can be viewed as a theoretic prediction.

From the numerical error (See Chapter 3.5.1 Numerical Error) a theoretic error concept can be derived from specific situations. The DeepFloat quantization method, for example, is deterministic and aims to emulate the original operations and computations. To optimize its performance this method trades numerical precision for model size reduction and power efficiency.

Given its structure and behavior, the error provided by this method can be both defined and theorized - Theoretic Error. By modeling this error, one can predict the outcome of a DeepFloat, by using the original model together with the Theoretic Error, by accumulation.

This Theoretic Error is a powerful tool, because it has provided insight on the degradation of a model by understanding its limitations without empiric accuracy measurements.

DeepFloat's operation core is ELMA (See Section 2.5.2 Posit DeepFloat Figure 4). Thus, by studying its structure a theoretic error can be proposed. ELMA has two main sections and two numerical basis for each one. The multiplications have been performed in the log domain and the multiplication output range has been limited to 256 possible outcomes. Hence, a multiplication error can be defined. The addition inside ELMA has been performed in the linear domain using 16 bits, reducing virtually the accumulation error to none, as stated in the original paper [6]. Considering such particularity, a linear-to-log and log-to-linear error can be defined.

Multiplication Error

The multiplication error was modeled using the DeepFloat range and the systolic array. A row vector with the DeepFloat range can be multiplied by itself transposed using the systolic array. The product corresponds to all multiplication possibilities that exist for DeepFloat. By performing the same with the DeepFloat range in floating-point and subtracting it to the DeepFloat multiplication matrix, a multiplication error matrix can be defined $Emul_{i,j}$. By using a row, the error from the multiplication has been isolated.

The systolic array is configured in order to compute a product with size 32 by 32. However, DeepFloat has 256 values, generating a multiplication matrix table with the size 256 by 256 (256 because DeepFloat is represented using 8 bits). To create this multiplication matrix, the range vector can be divided in 8 parts, generating 64 parts than can be put together to create the full multiplication matrix (See Figure 15).

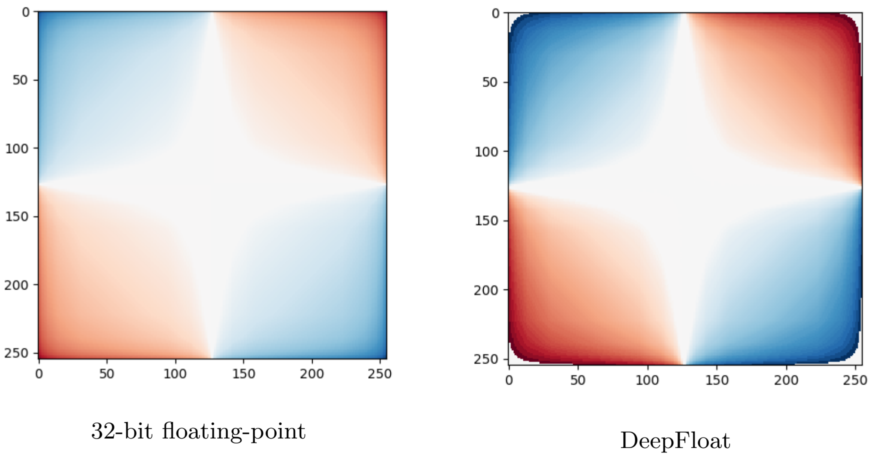


Figure 15: Multiplication Matrix on 32-bit float and 8-bit DeepFloat

Both images depict the distribution of the multiplication table. In blue are the positive regions and in red the negative.
$Emul_{i,j}$ difference between these two matrices
Original DeepFloat (without infinity): $[-4.096; +4.096]$
Multiplication range DeepFloat (without infinity): $[-4.096; +4.096]$
Multiplication range from DeepFloat using floating-point:
$[-16.777.216; +16.777.216]$

Linear-to-Log Error

The linear to log theoretic error ($Elin(x)$) is approximated by taking into account the DeepFloat range distribution (See Figure 9) and floating-point. Inside ELMA the accumulators have 16-bits, but because its error is virtually zero, it is considered the same as the former. The DeepFloat range has a Float representation and it can be used to compute the error for a number outside of this range. For this model, the error is the difference between a value and its nearest correspondent in the DeepFloat range, as shown in Figure 16. This linear-to-log and log-to-linear error model may not provide a perfect representation of the reality, though coming very close to do so.

Systolic Error Model

A product between two matrices can be defined as follows:

A is an m × n matrix and B is an n × p

i = 1, ..., m and j = 1, ..., p

$$\sum_{k=1}^{n} a_{ik} b_{kj} = c_{ij}$$

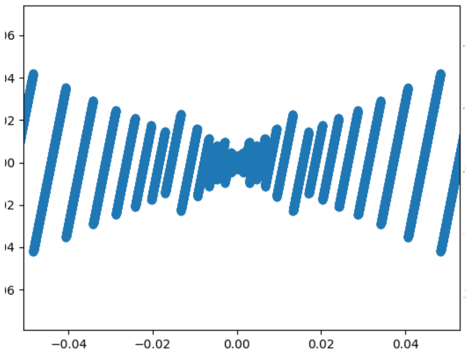
Taking the previous considerations into account regarding the DeepFloat and the ELMA structure, the previous definition can be adjusted, so to incorporate the linear-to-log error and the multiplication errors. This model can be used for the systolic array error. In order to apply the error model to an entire neural network, one must consider the initial transposition of the inputs and parameters from floating-point to DeepFloat - linear-to-log error.

i = 1, ..., m and j = 1, ..., p

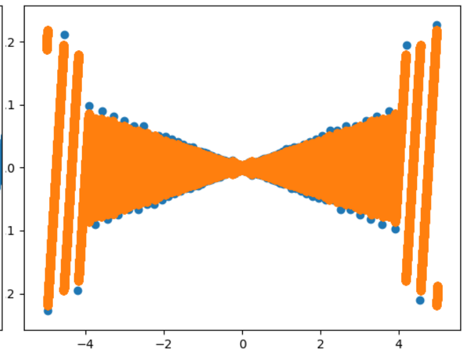
$$\sum_{k=1}^{n} (a_{ik} b_{kj} + Emul(ik, kj)) = cmul_{ij}$$

$$c_{ij} = Elin(cmul_{ij})$$

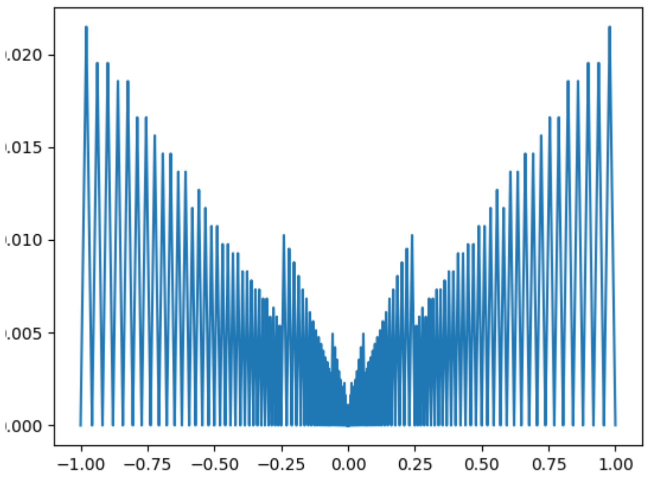
As illustrated by Figure 19, the error model is not perfect, but it is an attempt at modeling the DeepFloat error. By understanding how $Emul_{i,j}$ and $Elin(x)$ affect matrix multiplication, the quantization error can be minimized. This model also demonstrates that the usage of the systolic array has affected the error, because of $Elin(x)$. As illustrated by Figure 20, $Emul_{i,j}$ does not follow a perfectly smooth curve, therefore meaning that two adjacent multiplication results can have a positive or negative error component. A slight shift on the input matrices may result in very different outputs due to the power of compounding. Further reasearch would be beneficial for a better understanding of this. The performance of the Theoretic Model has been illustrated by Figure 19, where an example can be found with its visual representation.



a) Linear-to-Log
/Log-to-linear
Theoretic Error
Horizontal axis: DeepFloat Range
Vertical axis: Error



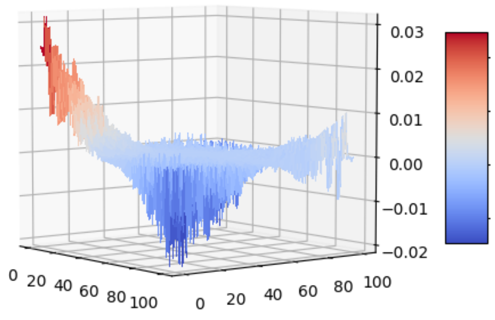
b) Real Error (Orange) on top of the
Linear-to-Log/Log-to-linear Theoretic
Error (Blue)
Horizontal axis: DeepFloat Range
Vertical axis: Error



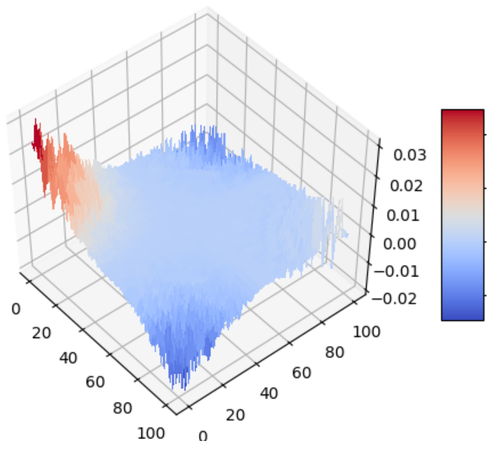
Linear-to-Log/Log-to-linear Theoretic Error (Module)
Horizontal axis: DeepFloat Range
Vertical axis: Error

Figure 16: Linear-to-Log Theoretic Error

(Units: Error FP32)



10000 inner samples



10000 inner samples

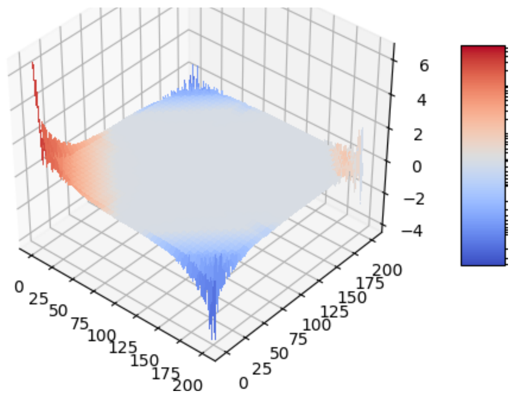
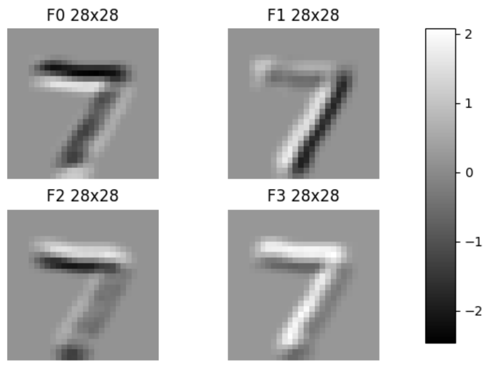
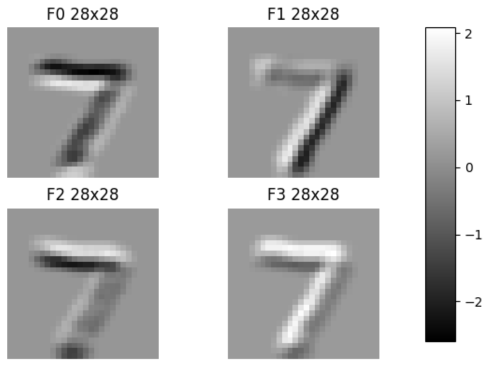


Figure 17: Infered Multiplication Error: 3D Visualization

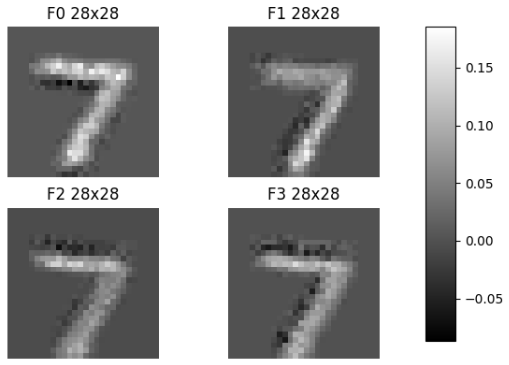
Difference between the two multiplication Matrices (See Figure 15)
Vertical axis: Error



Model A Output L1 Float 32-bit



Model A Output L1 Theoretic DeepFloat



Theoretic Error (Floating-point - DeepFloat Theoretic)

Figure 18: Application Example for Model B L1

Using the Error Model, a DeepFloat predicted output can be generated (Theoretic DeepFloat)

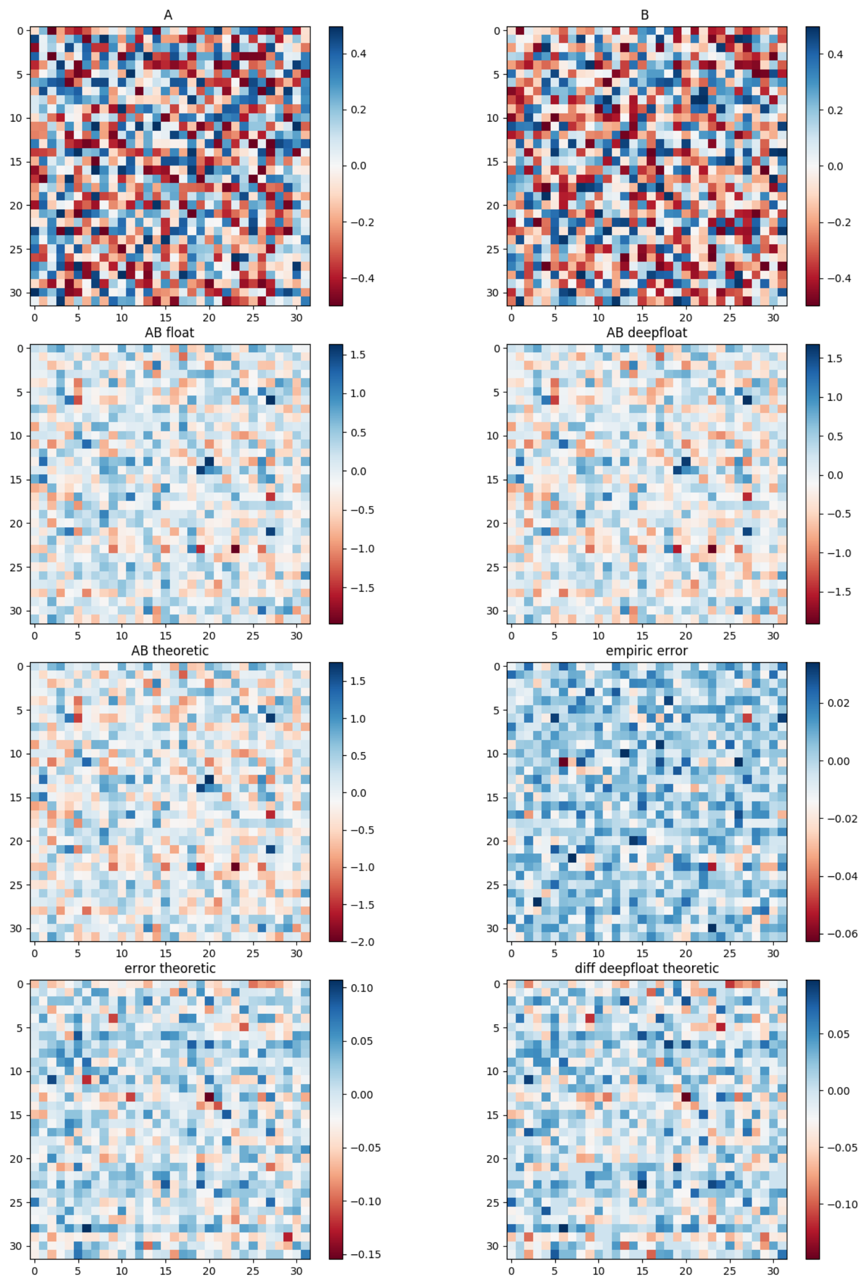


Figure 19: DeepFloat Theoretic Model Multiplication Example

Average Error for the Theoretic DeepFloat Model and Deepfloat is 0.98 for this example (A and B are two 32 by 32 matrices initialize with random uniform numbers between -0.5 and 0.5)

3.7 Computing systems

The study has been developed using two machines. The DeepFloat inference results were obtained using Machine 2, due to the simulations. All other data was obtained using Machine 1.

Machine 1	**Hardware**	**OS**: Ubuntu 18.04.3 LTS **CPU**: Intel(R) Core(TM) i5 CPU M 450 @ 2.40GHz **GPU**: NVIDIA Corporation GT216M [GeForce GT 325M] (rev a2) **RAM**: $\sim 4GB$
	Software	tensorflow 1.5.0 tensorflow-estimator 1.13.0 Python 3.6.7 Vivado v2019.1 (64-bit)
Machine 2	**Hardware**	**OS**: Linux release 7.6.1810 (Core) **CPU**: Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz **GPU**: NVIDIA Corporation GK110 [GeForce GTX TITAN] (rev a1) **RAM**: $\sim 32GB$
	Software	tensorflow-estimator 1.14.0 tensorflow-gpu 1.14.0 Python 3.6.8 Vivado v2019.1 (64-bit)

Table 2: Computing Systems Setup

4 Post-Training Model Quantization Results

4.1 Model A

4.1.1 Classification Analysis - Accuracy Degradation Using TensorFlow & DeepFloat Simulations

	Counts	Total	
32-bit float model - correct predictions	9,216	10,000	92.16%
8-bit (8,1,5,5,7)log model - correct predictions	9,200	10,000	92.00%
32-bit float model - incorrect predictions	784	10,000	7.84%
8-bit (8,1,5,5,7)log model - incorrect predictions	800	10,000	8.00%
Same prediction before and after quantization	9,902	10,000	99.02%

Table 3: Model A TensorFlow 32-bit float & 8-bit DeepFloat

Table 3 shows that the trained Model A only has 0.16% of accuracy drop after its quantization when using DeepFloat. It also demonstrates that 99.02% of the classifications are identical before and after quantization, resulting in 0.98% of different classifications. This information can mean that not all the different classifications after the quantization correspond to accuracy loss (0.98% > 0.16%). In this case, accuracy has only dropped by 0.16%. The accuracy is a key metric for a model performance. However, because it is a quantization problem, the percentage of different classifications holds a very important metric as well. In order to better understand these results, one can perform a classification analysis.

Ideally, the percentage of accuracy drop and the percentage of different classifications should be low and very close. Because the percentages are so low, and it is a shallow model, these are top performing results for this quantization process. The initial accuracy is high and after the quantization is still very high.

4.1.2 Context Classification Analysis

TensorFlow 32-bit float

Correct answer (Label)	**Correct classifications**	**Incorrect classifications**	**Total**
0	962	12	980
1	1,106	29	1,135
2	907	125	1,032
3	907	103	1,010
4	910	72	982
5	776	116	892
6	904	54	958
7	954	74	1,028
8	901	73	974
9	889	120	1,009

Table 4: Model A Context Classification TensorFlow 32-bit Float (MNIST)

Table 4 and Table 5 give better insight into the distinctions between the classification differences and why its percentage differs from the accuracy degradation.

DeepFloat 8-bit (8,1,5,5,7)log

Correct answer (Label)	Correct classifications	Incorrect classifications	Total
0	967	13	980
1	1,108	27	1,135
2	914	118	1,032
3	912	98	1,010
4	916	66	982
5	773	119	892
6	897	61	958
7	950	78	1,028
8	892	82	974
9	871	138	1,009

Table 5: Model A Context Classification 8-bit DeepFloat (MNIST)

For example, the images labeled with the number 9, 120 times out of 1,009 (11.89%) are incorrectly classified in the original model but only 38 times of 1,009 (3.77%) in the DeepFloat version of the same model. The opposite phenomena also can be verified. For example, the images labeled with the number 8, in the original model display incorrect classification times out of 974 (7.49%). And incorrectly classified 82 times out of 974 (8.42%) in the DeepFloat version of the model. In none of the classification possibilities, do both versions of the model have the same behavior.

When using DeepFloat, since it is a quantization process, the accuracy degradation is expected, meaning that some of the images that have the correct classification in the original model will most likely obtain incorrect classifications in the quantized model. This would lead one to assume that the number of incorrect classifications is bigger in the quantization version, which has been verified. However, when all of the classification groups are separated and individually analyzed, this assumption does not provide any true meaning.

Both the parameters and the input images in the DeepFloat version have been quantized and the computations have been performed using the systolic array. There is still a vast array of possibilities to explore, in order to explain such behavior. Nevertheless, no pattern nor explanation has been found. A speculative answer would be that given the nature of the problem - to classify handwritten, with hard edges, a quantization process as drastic as this, can have a significant effect on important edges, changing the input meaning.

Another possible answer could be that, since the classification is given by the biggest output, if other outputs are closer to the right output answer in the original model, in the quantization version, due to accumulative error, such as the outputs shift, incorrect classifications or right classification can be triggered. This is mainly due to positive and negative shifts in the bigger outputs, which can explain both situations.

These are some explanations that would benefit from further investigation. However, even without definitive answers, it is still possible to compare this quantization technique with others, because the ultimate performance metric is the accuracy. These answers would, however, help to generalize the viability of DeepFloat without comparative testing against other quantization techniques.

4.1.3 Classification Analysis - Accuracy Degradation Using TensorFlow Lite

	Counts	Total	
32-bit float model number of correct predictions	9,236	10,000	92.36%
8-bit int model number of correct predictions	9,235	10,000	92.35%
32 bit float model number of incorrect predictions	764	10,000	7.64%
8-bit int model number of incorrect predictions	765	10,000	7.65%
Same prediction before and after quantization	9,965	10,000	99.65%

Table 6: Model A TensorFlow Lite 32-bit Converted & Optimized 8-bit Model

When using TensorFlow Lite, the trained model must be converted. The converted model is then tested, displaying differences in its accuracy. The original Model A has 92.16% accuracy after training and the converted model has 92.36% (0.20% increase). To implement this conversion, the code from the official documentation page has been adapted [11] [15]. This difference has not been expected and no reason has been so far found, though, there is an accuracy increase. No retraining has been performed during conversion. Looking at the TensorFlow Lite implementation options (Figure 8), they correspond to a model not optimized.

To implement the 8-bit quantization version of the model, some optimization steps have been implemented. This implementation follows the official example code. The representative dataset used is the training data and the operations are converted to int8. (See Figure 8). The accuracy of the quantized version is 92.35% corresponding to a 0.01% accuracy drop, compared to the converted model and 0.19% increase regarding the accuracy of the original model.

The obtained data (Tables 6 and 3) shows that using TensorFlow Lite, for the Model A trained with MNIST, results in a lower accuracy degradation from the converted model to the optimized (quantized) version than the degradation from the original model to the DeepFloat version. They also show that the TensorFlow Lite quantized version has a bigger accuracy than the original model.

4.1.4 Context Classification Analysis

TensorFlow Lite 32-bit float

As in the quantization from the original model to DeepFloat, in the quantization from the converted model to the optimized model, higher counts of incorrect classifications have been expected in the optimized version. In Table 6, the difference is 1 count. However, by grouping the classification counts per label, the majority of the classification groups show more incorrect classifications that the original converted model. In some cases like label 4, the opposite has been verified (See Tables 7 and 8).

Another noteworthy key result, is that the converted model has a lower incorrect count compared to the original model for the labels 1, 2, 4, 5, 6, 9 (See Tables 4 and 7). These results cascade to the optimized (quantized) version of

Correct answer (Label)	Correct classifications	Incorrect classifications	Total
0	959	21	980
1	1,114	21	1,135
2	932	100	1,032
3	901	109	1,010
4	922	60	982
5	785	107	892
6	916	42	958
7	947	81	1,028
8	852	122	974
9	908	101	1,009

Table 7: Model A Context Classification TensorFlow Lite 32-bit Float (MNIST)

TensorFlow Lite 8-bit int

Correct answer (Label)	Correct classifications	Incorrect classifications	Total
0	961	19	980
1	1,114	21	1,135
2	934	98	1,032
3	906	104	1,010
4	923	59	982
5	786	106	892
6	915	43	958
7	945	83	1,028
8	851	123	974
9	900	109	1,009

Table 8: Model A Context Classification TensorFlow Lite 8-bit int (MNIST)

the model, its incorrect predictions are still higher than the original TensorFlow 32-bit model, and 8-bit DeepFloat.

Understanding why the converted model has higher accuracy than the original model is yet to be explained, but it may give insight into why the TensorFlow Lite quantized version performs better than the 8-bit DeepFloat version in terms of accuracy.

The accuracy degradation from the original model is 0.16% for DeepFloat and -0.19% for TensorFlow Lite. When comparing the quantized version between the original and the converted models, the accuracy degradation from TensorFlow Lite is still proven to be lower. The accuracy degradation from the optimized model for TensorFlow Lite is 0.01% (<0.16%).

The conclusion that can be obtained from this case study (Model A trained using MNIST) is that quantizing the model using TensorFlow Lite offers the best results when compared to DeepFloat.

By analyzing more complex models or models with different layers these results may change because the concepts/behaviors than lead to these results are not fully explored and this evidence is thus empiric. However, this case study has made some valid observations, and highlights some important analysis strategies such as grouping the labels when studying the accuracy degradation. Comparing accuracy degradation in both methods is also important, even when using an experimental result.

4.2 Model B

4.2.1 Classification Analysis - Accuracy Degradation Using TensorFlow & DeepFloat Simulations

	Counts	Total	
32 bit float model number of correct predictions	9,886	10,000	98.86%
32 bit float model number of incorrect predictions	114	10,000	1.14%

Table 9: Model B TensorFlow 32-bit float

Table 9 shows that the accuracy of Model B trained with MNIST using the original script is 98.86%, which is higher than Model A (92.16% Table 3). As previously stated, Model B has 5 layers, 3 CONV and 2 FC and Model A has 1 FC layer. The differences in accuracy can be attributed to each model's structure. Model B uses CONV layers meaning it is deeper, thus tending to provide higher accuracy in this specific situation.

To quantize Model B to DeepFloat, the same methods used for Model A have been applied. The parameters and inputs have been converted to DeepFloat. To validate its implementation, some inputs have been fed to Model B. During this validation, none of the two inputs studied (Figures 20 and 23) have been correctly classified, meaning that the output with the higher value is not the one that pairs with each input label. These inputs have been correctly classified when using the original TensorFlow 32-bit float model. The DeepFloat version of this model has also proven to take a significantly greater amount of time to simulate, because the layers are more complex. Understanding its behavior before the accuracy evaluation is necessary to validate its implementation.

Because the first 3 layers are CONV, the outputs from each one of these, especially the first ones, hold meaningful visual information than can be used for debugging and to understand what is happening within each layer by using visual inspection.

For instance, Figure 20 depicts the number 7. When this image is fed to the original model and to the DeepFloat model, it is evident the divergence that occurs layer after layer (Figures 21 and 22). A hypothetic solution for this divergence, would be that after each layer there is a *ReLu* activation that "turns-off" the negative outputs of each layer. Studying this example, one can observe that by using the Model B DeepFloat quantization, some output values are "pushed" into the negative direction and are turned to 0 after the activation, if they are negative. This effect accumulates layer after layer as depicted in Figure 22. The output of the third layer is very different in the original model and the quantized version. Should the final classification be correct, this difference would be irrelevant, however the opposite is true. Another example using a different image shows the same behavior, making this hypothesis a stronger explanation (See Figures 23, 24 and 25).

There are 10 outputs on the last layer, corresponding to the 10 Arabic numerals. Since there are only 10 outputs to this model, and because there is this classification problem with the input samples studied with the quantized model, it has been very difficult to understand if a correct classification can be either attributed to the model performance or random events, e.g., accumulation of noise provided by the activation and quantization. Also, each classification using the DeepFloat version of the model has had a duration of 40 minutes. This number can go down with an accelerator or the parallelization of the simulations, but it is a steep cost for all the 10,000 test images to measure its accuracy. For these reasons Table 9 only has the accuracy results for the original TensorFlow 32-bit Float model. Other methods can be used to compare the DeepFloat quantization to the TensorFlow Lite (See Classification Analysis - Accuracy DegradationUsing TensorFlow Lite).

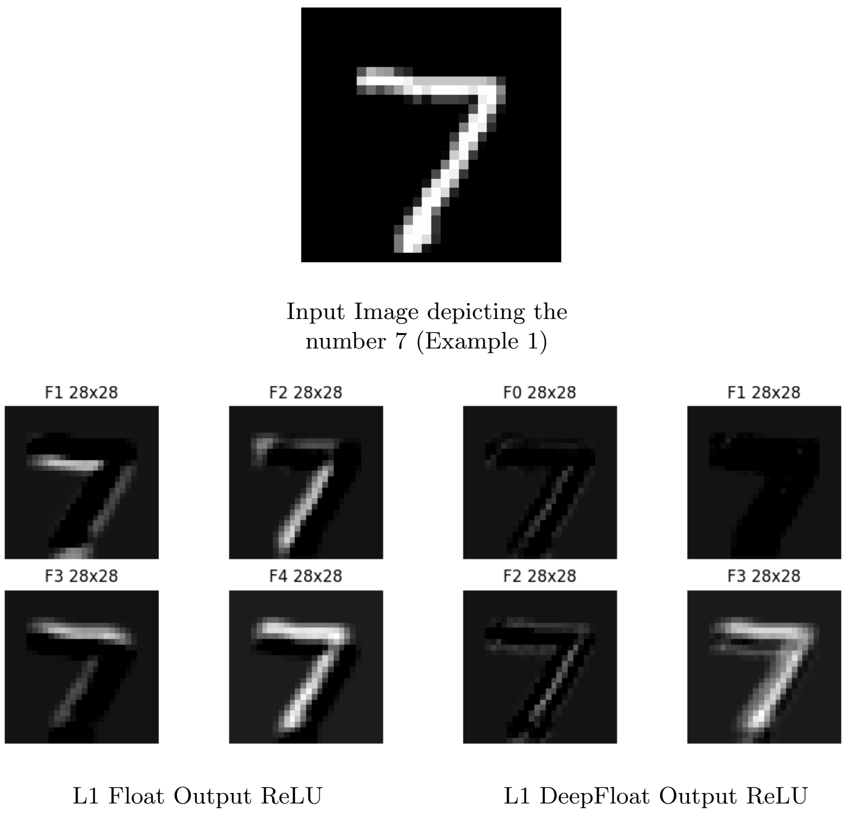
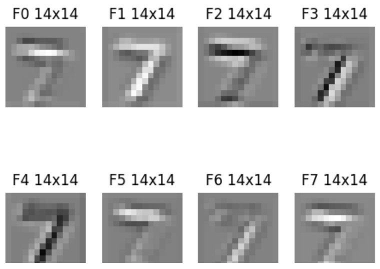
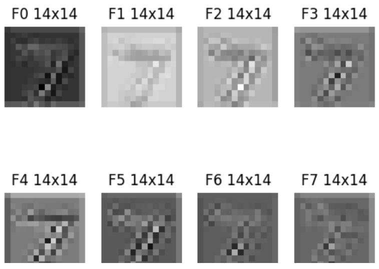


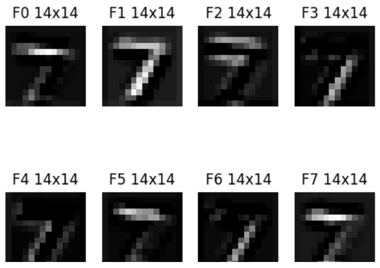
Figure 20: Model B Input and L1 Outputs (Example 1)



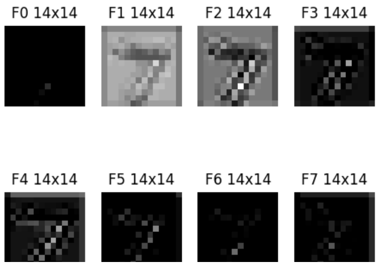
L2 Float Output



L2 DeepFloat Output

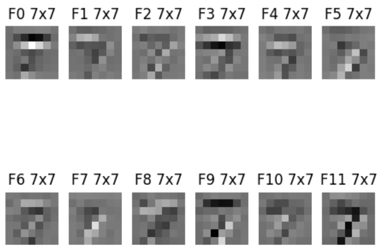


L2 Float Output ReLU

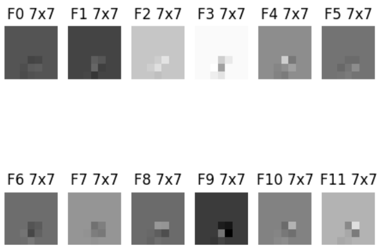


L2 DeepFloat Output ReLU

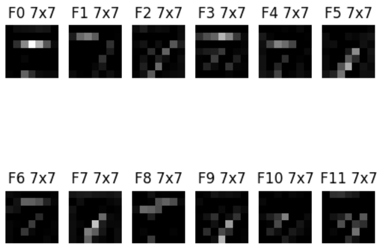
Figure 21: Model B L2 Outputs (Example 1)



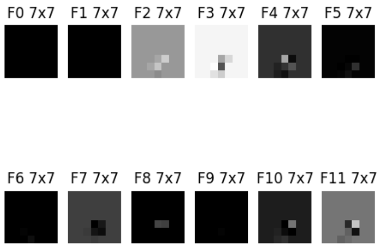
L3 Float Output



L3 DeepFloat Output



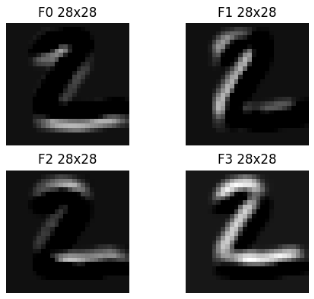
L3 Float Output ReLU



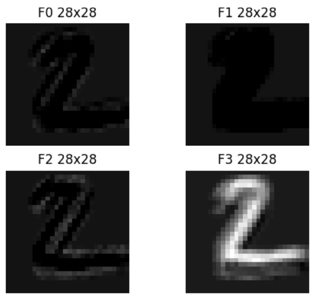
L3 DeepFloat Output ReLU

Figure 22: Model B L3 Outputs (Example 1)

Input Image depicting the number 2 (Example 2)

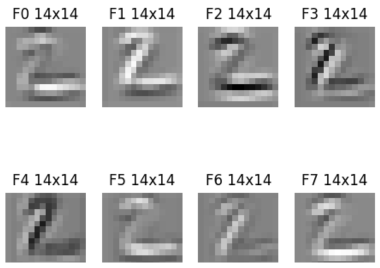


L1 Float Output ReLU

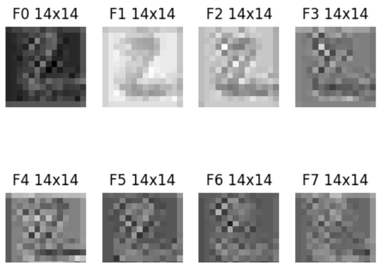


L1 DeepFloat Output ReLU

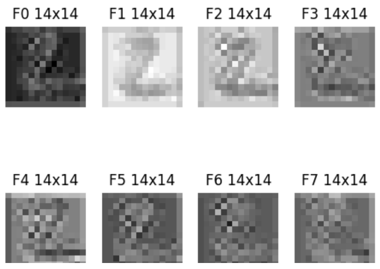
Figure 23: Model B Input and L1 Outputs (Example 2)



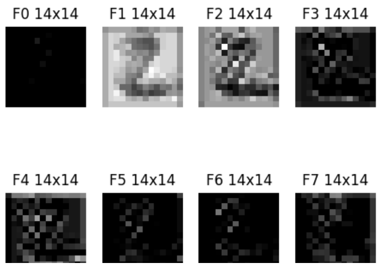
L2 Float Output



L2 DeepFloat Output

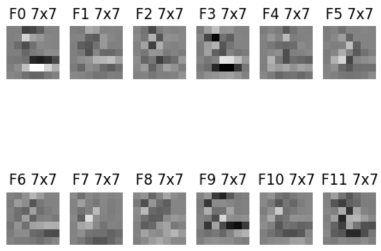


L2 Float Output ReLU

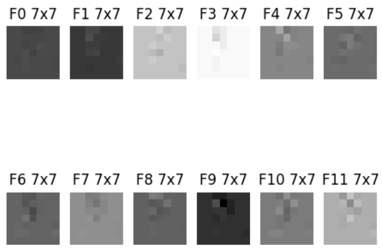


L2 DeepFloat Output ReLU

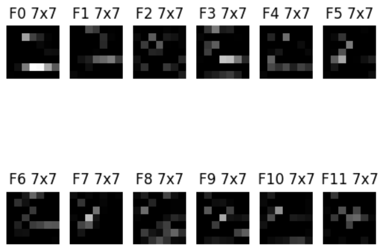
Figure 24: Model B L2 Outputs (Example 2)



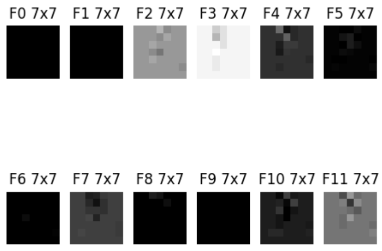
L3 Float Output



L3 DeepFloat Output



L3 Float Output ReLU



L3 DeepFloat Output ReLU

Figure 25: Model B L3 Outputs (Example 2)

4.2.2 Context Classification Analysis

32-bit float

Correct answer (Label)	Correct classifications	Incorrect classifications	Total
0	974	6	980
1	1,127	8	1,135
2	1,021	11	1,032
3	1,003	7	1,010
4	969	13	982
5	880	12	892
6	947	11	958
7	1,015	13	1,028
8	961	13	974
9	989	20	1,009

Table 10: Model B Context Classification TensorFlow Lite 32-bit Float (MNIST)

Table 10 depicts the classification results from the original Model B grouped by label. The accuracy per label is higher than the original model (See Table 4)

4.2.3 Classification Analysis - Accuracy Degradation Using TensorFlow Lite

	Counts	Total	
32 bit float model number of correct predictions	9,884	10,000	98.84%
8-bit int model number of correct predictions	9,890	10,000	98.90%
32 bit float model number of incorrect predictions	116	10,000	1.16%
8-bit int model number of incorrect predictions	110	10,000	1.10%
Same prediction on both models	9,992	10,000	99.92%
Same prediction on both models (correct)	9,884	10,000	98.84%
Same prediction on both models (incorrect)	108	10,000	1.08%

Table 11: Model B TensorFlow Lite 32-bit Converted & Optimized 8-bit Model

Like the Model A, Model B is quantized using TensorFlow Lite. Table 11 shows that the original model converted has 98.84% accuracy as the original model has 98.86% (a 0.02% accuracy drop). This accuracy drop has yet to be fully understood, however, it is very low. Regarding Model A, the converted version has had a higher accuracy, representing the opposite behavior.

The TensorFlow Lite quantized version has an accuracy of 98.90% and it represents an accuracy increase of 0.06% from the converted version and 0.04% from the original model. These differences, though not significative, may present a different situation than Model A. The quantized version has higher accuracy than the converted and original models, hinting that the accuracy for the quantized model provided by TensorFlow Lite is going to be similar to the original model, if one assumes a similar structure.

This accuracy study can be used to place constraints to the accuracy results that the DeepFloat quantized model should achieve, in order to be comparable. The TensorFlow Lite method has had an accuracy increase of 0.04% from the original 32-bit TensorFlow Model, meaning that the DeepFloat version, to be further competitive, must have at least the same accuracy performance.

Taking into consideration the two examples given to study the behavior per layer when using DeepFloat, their output classifications correspond to discrepancies between the labels and model classification. By only considering these two inputs' images, the DeepFloat version has at least a 0.02% accuracy drop. It has also been hinted by the analysis that the noise provided by DeepFloat and the *ReLu* activations have been present in all classification outputs. These results mean that the TensorFlow Lite method, in order to quantize the Model B, has provided better results. The DeepFloat method has at least a 0.02% accuracy drop as the TensorFlow Lite method has had an accuracy increase, hence preserving at least the same accuracy as the original model.

4.2.4 Context Classification Analysis

32-bit float

Correct answer (Label)	Correct classifications	Incorrect classifications	Total
0	976	4	980
1	1,130	5	1,135
2	1,021	11	1,032
3	1,002	8	1,010
4	972	10	982
5	879	13	892
6	942	16	958
7	1,015	13	1,028
8	958	16	974
9	989	20	1,009

Table 12: Model B Context Classification TensorFlow Lite 32-bit Float (MNIST)

8-bit int

Correct answer (Label)	Correct classifications	Incorrect classifications	Total
0	976	4	980
1	1,130	5	1,135
2	1,022	10	1,032
3	1,002	8	1,010
4	974	8	982
5	880	12	892
6	942	16	958
7	1,017	11	1,028
8	958	16	974
9	989	20	1,009

Table 13: Model B Context Classification TensorFlow Lite 8-bit int (MNIST)

Table 11 and Table 13 show that the TensorFlow Lite model has had an accuracy increase for the classification groups corresponding to the numbers 2, 4, 5 and 7, when compared to the converted model. Despite this increase not being fully understood, like Model A, due to the nature of the problem and the simplicity of the dataset, a quantization process can act as an edge definer to the input images, increasing in some cases its performance. This conclusion is speculative and requires further research. Nevertheless, the empiric results allows a comparative discussion about the accuracy results between the two quantization processes.

Comparing Table 11 and Table 10, differences between the classification accuracy from the original Model B and the converted model appear. They are not reflected in the accuracy drop (0.02%). Some output groups see accuracy improvements(3, 5, 6 and 8) and others see accuracy drops (0, 1 and 4). Just like with Model A, this analysis does not provide useful information other than there is a small shift on accuracy when converting the models.

Taking into consideration the classification accuracy of the original Model B and both quantization methods, the TensorFlow Lite method has shown better results.

5 Conclusion

This study takes on after [6]. In [6], the systolic array was not used to measure the accuracy of any model and the parallelization aspect of the implementation was not explored. The systolic array was used to study how an ASIC setup performs in terms of power and area, when compared to a traditional 8/32 bit system.

Although the parallelization techniques are independent of the format that is being used, this study shows options on how to implement and parallelize two of the most used types of layers - FC and CONV. This work also uses hardware simulations alongside a collection of Python scripts, in order to implement the DeepFloat solutions. Some inspection tools for debugging were also proposed.

The theoretic model proposed for the error also shows how the operations are implemented in the systolic array matters, due to the accumulation of error. Two operations can be mathematical equivalent but its implementations will affect the result and they will not achieve the same computed result at the end if implemented differently.

By analyzing these two models and their accuracy in the original format (32-bit Float), in the TensorFlow Lite int8 and DeepFloat 8-bit quantization versions, one can verify that the TensorFlow Lite solution presents similar, if not better, results, taking into consideration that it uses already existing computation systems, that are both well distributed and well established.

DeepFloat presents a new method to perform post-training quantization. However, in both case studies, Model A and Model B, the results provided by TensorFlow Lite for the same amount of bit reduction are better (lower accuracy degradation). This means that the DeepFloat Post-training Quantization technique may not be the best option for the case studies of Model A and Model B. These two cases are isolated and these results cannot be directly translated to other models, but they can highlight the importance of defining an error instead of focusing on the accuracy degradation of each method. This study also tries to define a theoretic error for DeepFloat, in order to find a solution for this problem, and it could benefit from further development.

5.1 Future Work

This study does not define a generic test or theoretic model that allows both DeepFloat and TensorFlow to be analyzed without an experiment that uses a real working model, but it highlights its importance.

Ideally, a robust model would allow engineers to understand both methods strengths and fragilities and in which context they perform the best, but said model is yet to be defined. For the area to evolve, the speculation and the empiric observations need to be replaced by the knowledge provided by a global understanding of the Post-training Quantization technique. This belief can be applied to other emerging techniques such as new machine learning architectures and optimization processes.

At a first glance, accuracy is an important metric, since it gives insight on a very specific Machine Learning problem. Nonetheless, it does not provide much value to the global pool of knowledge on how an architecture can be implemented to solve a specific problem.

An accelerator architecture for DeepFloat should be implemented. Since the implementation is heavily dependent on hardware, a software infrastructure can lead to faster iterations, without relying on the systolic implementation or kernels - distributed alongside the original paper. On a first stage, the implementation of a library allows for an easier access of basic arithmetic building blocks would be a a great improvement for the distribution of DeepFloat.

On a second stage, should the DeepFloat technology be proved to be useful in other use cases, implementing a hardware accelerator on GPU or FPGA would improve the inference performance that software is unable to provide.

Furthermore, if deemed fitting, a reusable HDL accelerator can be developed to complement other AI cores, given that the DeepFloat quantization seamlessly integrates with the already supported 32-bit single precision. Another possibility is to develop a core that only uses this architecture for IoT devices.

Lastly, this study only focus on the inference phase. Sould TensorFlow Lite be proven to be a better alternative for a Post-training Quantization problem, it does not mean that DeepFloat is not better suited for training, since it uses fewer bits and power, when compared to traditional approaches. Another interesting path for further research concerns the implementation of this format and architecture in other contexts.

References

[1] Ron Banner, Yury Nahshan, and Daniel Soudry. "Post training 4-bit quantization of convolutional networks for rapid-deployment". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 7948–7956. URL: http://papers.nips.cc/paper/9008-post-training-4-bit-quantization-of-convolutional-networks-for-rapid-deployment.pdf.

[2] Jared Duke. *Inside TensorFlow: TensorFlow Lite*. TensorFlow. 2019. URL: https://www.youtube.com/watch?v=gHN0jDbJz8E.

[3] Kaiyuan Guo et al. "A Survey of FPGA Based Neural Network Accelerator". In: *CoRR* abs/1712.08934 (2017). arXiv: 1712.08934. URL: http://arxiv.org/abs/1712.08934.

[4] Song Han et al. "Learning both Weights and Connections for Efficient Neural Network". In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes et al. Curran Associates, Inc., 2015, pp. 1135–1143. URL: http://papers.nips.cc/paper/5784-learning-both-weights-and-connections-for-efficient-neural-network.pdf.

[5] Jeff Johnson. *Making floating point math highly efficient for AI hardware*. 2019. URL: https://engineering.fb.com/ai-research/floating-point-math/.

[6] Jeff Johnson. "Rethinking floating point for deep learning". In: *ArXiv* abs/1811.01721 (2018).

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Neural Information Processing Systems* 25 (Jan. 2012). DOI: 10.1145/3065386.

[8] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* 521.7553 (2015), pp. 436–444. DOI: 10.1038/nature14539. URL: https://doi.org/10.1038/nature14539.

[9] Yann LeCun and Corinna Cortes. "MNIST handwritten digit database". In: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[10] *Making floating point math highly efficient for AI hardwareA history of machine learning*. 2019. URL: https://cloud.withgoogle.com/build/data-analytics/explore-history-machine-learning/.

[11] Google Cloud Developer Advocate Martin Görner. *Learn TensorFlow and deep learning, without a Ph.D.* January 19, 2017. URL: https://cloud.google.com/blog/products/gcp/learn-tensorflow-and-deep-learning-without-a-phd.

[12] Eriko Nurvitadhi et al. "Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC". In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Aug. 2016. DOI: 10.1109/fpl.2016.7577314.

[13] Chigozie Nwankpa et al. "Activation Functions: Comparison of trends in Practice and Research for Deep Learning". In: *CoRR* abs/1811.03378 (2018). arXiv: 1811.03378. URL: http://arxiv.org/abs/1811.03378.

[14] Keiron O'Shea and Ryan Nash. "An Introduction to Convolutional Neural Networks". In: *ArXiv e-prints* (Nov. 2015).

[15] *Post-training quantization.* 2019. URL: `https://www.tensorflow.org/lite/performance/post_training_quantization`.

[16] Suharsh Sivakumar. *Inside TensorFlow: TF Model Optimization Toolkit (Quantization and Pruning).* TensorFlow. 2020. URL: `https://www.youtube.com/watch?v=4iq-d2AmfRU`.

[17] *Stanford CS class CS231n: Convolutional Neural Networks for Visual Recognition.* URL: `http://cs231n.github.io/`.

[18] Peter Stone et al. *Artificial Intelligence and Life in 2030.* Sept. 2016. URL: `http://ai100.stanford.edu/2016-report`.

[19] *The AI Index 2019 Annual Report.* Dec. 2019.

Annex A

Rethinking floating point for deep learning

Jeff Johnson
Facebook AI Research
New York, NY
`jhj@fb.com`

Abstract

Reducing hardware overhead of neural networks for faster or lower power inference and training is an active area of research. Uniform quantization using integer multiply-add has been thoroughly investigated, which requires learning many quantization parameters, fine-tuning training or other prerequisites. Little effort is made to improve floating point relative to this baseline; it remains energy inefficient, and word size reduction yields drastic loss in needed dynamic range. We improve floating point to be more energy efficient than equivalent bit width integer hardware on a 28 nm ASIC process while retaining accuracy in 8 bits with a novel hybrid log multiply/linear add, Kulisch accumulation and tapered encodings from Gustafson's posit format. With no network retraining, and drop-in replacement of all math and float32 parameters via round-to-nearest-even only, this open-sourced 8-bit log float is within 0.9% top-1 and 0.2% top-5 accuracy of the original float32 ResNet-50 CNN model on ImageNet. Unlike int8 quantization, it is still a general purpose floating point arithmetic, interpretable out-of-the-box. Our 8/38-bit log float multiply-add is synthesized and power profiled at 28 nm at $0.96\times$ the power and $1.12\times$ the area of 8/32-bit integer multiply-add. In 16 bits, our log float multiply-add is $0.59\times$ the power and $0.68\times$ the area of IEEE 754 float16 fused multiply-add, maintaining the same signficand precision and dynamic range, proving useful for training ASICs as well.

1 Introduction

Reducing the computational complexity of neural networks (NNs) while maintaining accuracy encompasses a long line of research in NN design, training and inference. Different computer arithmetic primitives have been considered, including fixed-point [21], uniform quantization via 8 bit integer [15], ternary [20] and binary/low-bit representations [29, 3, 1]. Some implementations are efficiently implemented on CPU/GPU ISAs [35, 33], while others demand custom hardware [10]. Instead of developing quantization techniques increasingly divorced from the original implementation, we seek to improve floating point itself, and let word size reduction yield efficiency for us. It is historically known to be up to $10\times$ less energy efficient in hardware implementations than integer math [14]. Typical implementation is encumbered with IEEE 754 standard compliance [37], demanding specific forms such as fused multiply-add (FMA) that we will show as being inefficient and imprecise. Memory movement (SRAM/DRAM/flip-flops) dominates power consumption; word bit length reduction thus provides obvious advantages beyond just reducing adder and multiplier area.

We explore encodings to better capture dynamic range with acceptable precision in smaller word sizes, and more efficient summation and multiplication (Sections 3-5), for a reduction in chip power and area. Significant inspiration for our work is found in logarithmic number systems (LNS) [2] and the work of Miyashita et al. [24] that finds logarithmic quantizers better suited to data distributions in NNs, and alternative visions of floating point from Gustafson [11, 12] and Kulisch [19]. We sidestep prior LNS design issues with numerical approximation and repurpose ideas from Gustafson and

Preprint. Work in progress.

Table 1: Dynamic range and significand fractional precision of math types considered

Word bits	**Encoding type**	**Range in decibels** $20 \log_{10}(f_{max}/f_{min})$	**Fraction bits (max)**
8	symmetric integer $[-2^7+1, 2^7-1]$	42.1	—
8	(8, 0) posit or (8, 0, α, β, γ) log	72.2	5
8	(4, 3) float (w/o denormals)	83.7	3
16	symmetric integer $[-2^{15}+1, 2^{15}-1]$	90.3	—
8	(4, 3) float (w/ denormals)	101.8	3
8	(8, 1) posit or (8, 1, α, β, γ) log	144.5	4
16	(5, 10) float16 (w/o denormals)	180.6	10
16	(5, 10) float16 (w/ denormals)	240.8	10
12	(12, 1) posit or (12, 1, α, β, γ) log	240.8	8
8	(8, 2) posit or (8, 2, α, β, γ) log	289.0	3
16	(16, 1) posit or (16, 1, α, β, γ) log	337.2	12

Kulisch, producing a general-purpose arithmetic that is effective on CNNs [13] without quantization tinkering or re-training (Section 7), and can be as efficient as integer math in hardware (Section 8).

2 Floating point variants for NNs

There are few studies on NNs for floating point variants beyond those provided for in CPU/GPU ISAs. [4] shows a kind of 8 bit floating point for communicating gradients, but this is not used for general computation. Flexpoint [17] and the Brainwave NPU [6] use variants of *block floating point* [36], representing data as a collection of significands with a shared exponent. This requires controlled dynamic range variation and increased management cost, but saves on data movement and hardware resources. For going to 8 bits in our work, we seek to improve the encoding and hardware for a reasonable tradeoff between dynamic range and precision, with less machinery needed in software.

For different precisions, [5] shows reduced-precision floating point for training smaller networks on MNIST and CIFAR-10, with (6, 5)[1] floating point without denormal significands being comparable to float32 on these examples. (8, 7) *bfloat16* is available on Google's TPUv2 [9]. This form maintains the same normalized exponent range as float32, except with reduced precision and smaller multipliers. However, the forms of encoding and computation for many of these variants are not substantially different than implementations available with common ISAs, hardened FPGA IP, and the like. We will seek to improve the encoding, precision and computation efficiency of floating point to find a solution that is quite different in practice than standard (e, s) floating point.

3 Space-efficient encodings

IEEE 754-style fixed width field encodings are not optimal for most data distributions seen in practice; float32 maintains the same significand precision at 10^{-10} as at 10^{10}. Straightforward implementation of this design in 8 bits will result in sizable space encoding NaNs, $\sim 6\%$ for (4, 3) float. Denormals use similar space and are expensive in hardware [26]; not implementing them restricts the dynamic range of the type (Table 1). *Tapered floating point* can solve this problem: within a fixed-sized word, exponent and significand field size varies, with a third field indicating relative size. To quote Morris (1971): "users of floating-point numbers are seldom, if ever, concerned *simultaneously* with loss of accuracy and with overflow. If this is so, then the range of possible representation can be extended [with tapering] to an extreme degree and the slight loss of accuracy will be unnoticed." [25]

A more efficient representation for tapered floating point is the recent *posit* format by Gustafson [12]. It has no explicit size field; the exponent is encoded using a Golomb-Rice prefix-free code [8, 22], with the exponent e encoded as a Golomb-Rice quotient and remainder (q, r) with q in unary and r in binary (in posit terminology, q is the *regime*). Remainder encoding size is defined by the *exponent*

[1]Throughout, (e, s)-float refers to IEEE 754-style floating point, with sign bit, e-bit biased exponent and s-bit 0.s or 1.s fixed point significand; float16/float32 are shorthand for IEEE 754 binary16/binary32.

scale s, where 2^s is the Golomb-Rice divisor. Any space not used by the exponent encoding is used by the significand, which unlike IEEE 754 always has a leading 1; gradual underflow (and overflow) is handled by tapering. A posit number system is characterized by (N, s), where N is the word length in bits and s is the exponent scale. The minimum and maximum positive finite numbers in (N, s) are $f_{min} = 2^{-(N-2)2^s}$ and $f_{max} = 2^{(N-2)2^s}$. The number line is represented much as the projective reals, with a single point at $\pm\infty$ bounding $-f_{max}$ and f_{max}. $\pm\infty$ and 0 have special encodings; there is no NaN. The number system allows any choice of $N \geq 3$ and $0 \leq s \leq N - 3$.

s controls the dynamic range achievable; e.g., 8-bit $(8, 5)$-posit $f_{max} = 2^{192}$ is larger than f_{max} in float32. $(8, 0)$ and $(8, 1)$ are more reasonable values to choose for 8-bit floating point representations, with f_{max} of 64 and 4096 accordingly. Precision is maximized in the range $\pm[2^{-(s+1)}, 2^{s+1})$ with $N - 3 - s$ significand fraction bits, tapering to no fraction bits at $\pm f_{max}$.

4 Accumulator efficiency and precision

A sum of scalar products $\sum_i a_i b_i$ is a frequent operation in linear algebra. For CNNs like ResNet-50 [13], we accumulate up to 4,608 (2d convolution with $k = 3 \times 3, c_{in} = 512$) such products.

Integer addition is associative (excepting overflow); the order of operations does not matter and thus it allows for error-free parallelization. In typical accelerator use, the accumulation type is 32 bits. Typical floating point addition is notorious for its lack of associativity; this presents problems with reproducibility, parallelization and rounding error [26]. Facilities such as *fused multiply-add* (FMA) that perform a sum and product $c + a_i b_i$ with a single rounding can reduce error and further pipeline operations when computing sums of products. Such machinery cannot avoid rounding error involved with tiny (8-bit) floating point types; the accumulator can become larger in magnitude than the product being accumulated into it, and the significand words no longer overlap as needed even with rounding (yielding $c + ab = c$); increasing accumulator size a bit only defers this problem.

There is a more efficient and precise method than FMA available. A *Kulisch accumulator* [19] is a fixed point register that is wide enough to contain both the largest and smallest possible scalar product of floating point values $\pm(f_{max}^2 + f_{min}^2)$. It provides associative, error-free calculation (excepting a single, final rounding) of a sum of scalar floating point products; a float significand to be accumulated is shifted based on exponent to align with the accumulator for the sum. Final rounding to floating point is performed after all sums are made. A similar operation known as *Auflaufenlassen* was available in Konrad Zuse's Z3 as early as 1941 [18], though it is not found in modern computers.

We will term this operation of summing scalar products in a Kulisch accumulator *exact multiply add* (EMA). For an inner product, given a rounding function[2] $r(\cdot)$ with the argument evaluated at infinite precision, EMA calculates $r(\sum_i a_i b_i)$, whereas FMA calculates $r(a_n b_n + r(a_{n-1} b_{n-1} + r(\cdots + r(a_1 b_1 + 0) \cdots)))$. Both EMA and FMA can be implemented for any floating point type. Gustafson proposed Kulisch accumulators to be standard for posits, terming them *quires*.

Depending upon float dynamic range, EMA can be considerably more efficient than FMA in hardware. FMA must mutually align the addends c and the product ab, including renormalization logic for subtraction cancellation, and the proper alignment cannot be computed until fairly late in the process. Extra machinery to reduce latency such as the *leading zero (LZ) anticipator* or *three path architectures* have been invented [28]. If multiply-add needs to be pipelined for timing closure, EMA knows upfront the location of the floating point of c needed in alignment (as it is fixed), and can thus accumulate a new product into it every clock cycle, while a FMA must hold onto the starting value of the accumulator c until later in the process, increasing the pipeline non-combinational area and often requiring greater use of an external register file (for multiple accumulators c_i in concurrent use) and effective "loop unrolling" at software level to fill all pipeline slots. The rounding performed every FMA requires additional logic, and rounding error can still compound greatly across repeated sums.

[2] $r(\cdot, b)$ is a rounding function that produces b fractional bits, and $r_i(\cdot, b)$ is the i-th fractional bit returned. We assume IEEE 754-style round-to-nearest-even (with sticky bit OR-reduction) for $r(\cdot)$.

5 Multiplier efficiency

Floating point with EMA is still expensive, as there is added shifter, LZ counter, rounding, etc. logic. Integer MAC and float FMA/EMA both involve multiplication of fixed-point values; for int8/32 MAC this multiply is 63.4% of the combinational power in our analysis at 28 nm (Section 8).

A logarithmic number system (LNS) [16] avoids hardware multipliers entirely, where we round and encode $\log_B(x)$ for some base B to represent a number $x \in \mathbb{R}$. Hitherto we have considered *linear domain* representations, where $x \in \mathbb{R}$ is rounded and encoded as x in integer, fixed or floating point representation (note that floating point is itself a combination of linear and log encodings). Log domain operations on linear $x > 0, y > 0$ represented as $i = \log_2(x), j = \log_2(y)$ are:

$$\begin{aligned}\log_2(x \pm y) &= i + \sigma_\pm(j - i)\\ \log_2(xy) &= i + j \qquad\qquad [2]\\ \log_2(x/y) &= i - j\end{aligned}$$

As values $x \le 0$ are outside the log domain, sign and zero are handled separately [31], as is $\pm\infty$. We encode $B = 2$ log numbers with a sign bit and a signed fixed-point number of the form $m.f$, which represents the linear domain value $\pm 2^{(m + \sum_i f_i/2^i)}$. For add/sub, without loss of generality, order $j \le i$, and $\sigma_\pm(x) = \log_2(1 \pm 2^x)$; this is the historical weak point of a LNS, as implementations use costly LUTs or piecewise linear approximation of $\sigma_\pm(x)$. This can be more expensive than hardware multipliers. The approximation $\log_2(1 + x) \approx x$ for $x \in [0, 1]$ could also be used [24], but this adds significant error, especially with repeated sums.

$\sigma_\pm(x)$ need only be evaluated if one wishes to keep the partial sum in the log domain. As with Kulisch accumulation versus FMA, we accumulate in a different representation than the scalar product for efficiency. For $\sum_i a_i b_i$, we multiply $a_i b_i$ in the log domain, and then approximate as a linear domain floating point value for accumulation. Translating log domain $m.f$ to linear is easier than $\sigma_\pm(x)$, as we can just consider the fractional portion f; m is linear domain multiplication by 2^m (floating point exponent addition or fixed point bit shift). A LUT maps $f \in [0, 1)$ to $p(f) = 2^f - 1$. $p(f)$ is the linear representation of the log number fractional part; the LUT maps all bits of f to a desired number of bits α of $p(f)$ or $r(p(f), \alpha)$, for a $(2^{f_{bits}} \times \alpha)$-bit LUT. Linear approximation of $m.f$ is the floating point value $\pm 2^m(1 + \sum_{i=1}^{\alpha} 2^{-i} r_i(p(f), \alpha))$. This is expanded in the usual way for Kulisch accumulation. Just as Kulisch accumulation is efficient for linear domain values up to a reasonably wide dynamic range, it proves quite efficient for our linear approximations of log values.

To convert a linear domain value back to log domain, we map $g \in [0, 1)$ to $q(g) = \log_2(1 + g)$. g is a linear domain fixed-point fraction; to control the size of the LUT we only consider β bits via rounding of g. $q(r(g, \beta))$ is similarly rounded to a desired γ bits; note that this latter rounding is log domain. $r(q(r(g, \beta)), \gamma)$ is then a $(2^\beta \times \gamma)$-bit LUT. We also choose $\alpha \ge f_{bits} + 1, \beta \ge \alpha, \gamma = f_{bits}$ to ensure that log-to-linear-to-log conversion of f is the identity, or $f = r(q(r(r(p(f), \alpha), \beta)), \gamma)$.

We will name this (somewhat inaccurately) *exact log-linear multiply-add* (ELMA). The log product and linear sum are each exact, but the log product is not represented exactly by $r(p(f))$ as this requires infinite precision, unlike EMA which is exact except for a final rounding. The intermediate log product avoids overflow or underflow with an extra bit for the product's m. If a linear-to-log mapping is desired (returning a log number after summation), there is also loss via $r(q(g))$.

Combining log-to-linear mapping with Kulisch accumulation makes log domain multiply-add efficient and reasonably accurate. Small p and q LUTs reduce well in combinational logic. They are practical for 16-bit types too, as compression can be used to reduce the size. For larger types they are impractical, as α, β, γ need to scale with $2^{f_{bits}}$, at which point $\sigma_\pm$ is a better strategy. As with FMA, repeated summation via $\sigma_\pm$ is subject to magnitude difference error (*e.g.*, the $c + ab = c$ case). Our approximation introduces error with $r(p(f))$ and $r(q(g))$, but mitigates repeated summation error and is immune to magnitude differences. This tradeoff seems acceptable in practice (Section 7).

An 8-bit log number by default suffers from the same problem as 8-bit IEEE-style floating point; the dynamic range is limited by the fixed point encoding. We can use the same tapering as used in (N, s) posit for $m.f$ log numbers. m is encoded as an exponent, and f as a floating point significand. f_{min} and f_{max} are then exactly the same for posit-tapered base-2 log or linear domain values. Setting $\gamma = f_{bits}$ (which is at maximum $(N - 3 - s)$ for posits) introduces additional tapering rounding error, as subsequent rounding in encoding is performed outside regimes of maximum

Table 2: ResNet-50 ImageNet validation set accuracy per math type

Math type	Multiply-add type	top-1 acc (%)	top-5 acc (%)
float32	FMA	76.130	92.862
(8, 1, 5, 5, 7) log	**ELMA**	**-0.90**	**-0.20**
(7, 1) posit	EMA	-4.63	-2.28
(8, 0) posit	EMA	-76.03	-92.36
(8, 1) posit	EMA	-0.87	-0.19
(8, 2) posit	EMA	-2.20	-0.85
(9, 1) posit	EMA	-0.30	-0.09
Jacob et al. [15]:			
float32	FMA	76.400	n/a
int8/32	MAC	-1.50	n/a
Migacz [23]:			
float32	FMA	73.230	91.180
int8/32	MAC	-0.20	-0.03

precision. γ is increased up to 3 bits (guard, round and sticky bits in typical round-to-nearest-even) to improve accuracy here. This encoding we will refer to as $(N, s, \alpha, \beta, \gamma)$ log (posit tapered). We can similarly choose to encode log numbers using an IEEE 754 format (with biased exponents, NaN representations etc.); we use this for our ELMA comparison against float16 FMA in Section 8.

6 Additional hardware details

To make EMA/ELMA more energy efficient, we restrict accumulator range to $[f_{min}^2, f_{max}]$; handling temporary underflow rather than overflow is more important in our experience. Kulisch accumulator conversion back to log or linear N-bit types uses a LZ counter and shifter but can be substantially amortized in two ways. First, many sums are performed, with final conversion done only once per inner product. Energy for the majority of work is thus lower than MAC/FMA (Section 8); increased area for increased energy efficiency is generally useful in the era of "dark silicon" [32], or conversion module instances can be rationed (limiting throughput) and/or clock gated. Second, structures with local operand reuse (*e.g.*, systolic arrays, fixed-function convolvers) naturally require fewer converter instances, reducing area (discussion in Section 8 as well). EMA and FMA accuracy are the same for a single sum $c + ab$; our power advantage would disappear in this domain, but the vast majority of flops/ops in NNs require repeated rather than singular sums. Note that int8/32 usage itself requires some conversion back to int8 in the end that we do not evaluate.

7 FPGA experiments

Our implementation is in SystemVerilog for ASIC evaluation, built into an FPGA design with Intel FPGA OpenCL RTL integration support, with rudimentary PyTorch [27] integration. Source code is available at `github.com/facebookresearch/deepfloat`. We evaluate (N, s) posit and $(N, s, \alpha, \beta, \gamma)$ log arithmetic on the ResNet-50 CNN [13] with the ImageNet ILSVRC12 validation set [30]. We use float32 trained parameters from the PyTorch model zoo, with batch normalization fused into preceding affine layers [15]. float32 parameters and network input are converted to our formats via round-to-nearest-even; no other adjustment of these values is performed. When converting into or out of a Kulisch accumulator, we can add a small exponent bias factor, adjusting the input exponent by m, or the output exponent by n. This is effectively free (a small adder). No changes are made to any activations except for such a bias of $n = -4$ at the last (fully connected) layer to recenter unnormalized log probabilities from around 16.0 to 1.0. Without this we have an additional loss in top-1 of around 0.5-1%, with little change to top-5. If the Kulisch accumulator itself can be directly considered for top-k comparison, this avoids the need as well. All math is replaced with the corresponding posit or log versions; average pooling is via division of the Kulisch accumulator.

Our results are in Table 2, along with two int8/32 quantization comparisons. (8, 0) linear posit has insufficient dynamic range to work; activations are quickly rounded to zero. Our (8, 1, 5, 5, 7)

Table 3: Chip area and power for 28 nm, 1-cycle multiply-add at 500 MHz

Component	**Area** μm^2	**Power** μW
int8/32 MAC PE	**336.672**	**283**
multiply	121.212	108.0
add	117.810	62.3
non-combinational	96.768	112.7
(8, 1, 5, 5, 7) log ELMA PE	**376.110**	**272**
log multiply (9 bit adder)	32.760	17.1
$r(p(f))$ (16x5 bit LUT)	8.946	5.4
Kulisch shift (6 $\rightarrow$ 38 bit)	81.774	71.0
Kulisch add (38 bit)	123.732	54.2
non-combinational	126.756	124.3
float16 (w/o denormals) FMA PE	**1545.012**	**1358**
(5, 10) (11, 11, 10) log ELMA PE	**1043.154**	**805**
(this log is (5, 10) float16-style encoding, same dynamic range; denormals for log and float16 here are unhandled and flush to zero)		
32x32 systolic w/ int8/32 MAC PEs	**348231**	**226000**
32x32 systolic w/ (8, 1, 5, 5, 7) log ELMA PEs	**457738**	**195500**

log result remains very close to (8, 1) linear posit. The int8/32 results listed do not start from the same float32 parameters as our trained network, so they are not directly comparable. They use training with simulated quantization [15] and KL-divergence calibration with sampled activations [23], whereas we perform math in the usual way in our log or linear domain arithmetic after rounding input and parameters. We obtain reasonably similar precision without retraining, sampling activations or learning quantization parameters, while retaining general floating point representations in 8 bits.

8 ASIC evaluation

We use Synopsys Design Compiler and PrimeTime PX with a commercially available 28 nm library, target clock 500 MHz. Process corners are SS@-40°C synthesis, TT@25°C power analysis at 0.81V. Table 3 investigates multiply-add PEs, and as a proxy for an accelerator design, a 32x32 matrix multiplication systolic array with these PEs. The float16 FMA is Synopsys DesignWare dw_fp_mac. We accumulate to the C matrix in place (*stationary C*), shifting out values upon completion. The int8/32 array outputs unprocessed int32; for ELMA, Kulisch accumulators are shifted across the PEs for C output and converted to 8 bit log at the boundary via 32 conversion/encoder modules. The 1024 PEs within do not include these (as discussed in Section 6). 64 posit taper decoders are included for where A and B are passed as input. Power analysis uses testbench waves for 128-d vectors with elements drawn from $N(0, 1)$; int8 quantization has a max of 2σ. PEs evaluate a variety of these inner products, and the systolic arrays a variety of GEMMs with these vectors.

ELMA saves 90.9 μW over int8/32 on multiplication, but loses 68.3 μW on the add. ELMA non-combinational demands are higher with additional state required (Kulisch and decoded log numbers), but could be reduced by not handling underflow all the way to f_{min}^2. Despite the larger Kulisch adder, effectively only 6 bits are summed (with carry) each cycle versus up to 16 with int8/32; strategies for 500+ bit Kulisch accumulators [34] might work in this small regime to further take advantage of this. Our 16-bit ELMA $\alpha = 11$ $p(f)$ combinational LUT is 386 μm^2 despite compression, now a significant portion of the design. Larger α likely needs a compiled ROM or explicit compute of $p(f)$.

A more in-depth analysis for our work would need to determine a Pareto frontier between frequency/latency, per-operation energy, area, pipeline depth, math implementation and accuracy similar to the Galal et al. FPU generator work [7], to see precisely in what regimes ELMA is advantageous. We provide our limited analysis here, however rough, to help motivate future investigation.

9 Conclusions

DNNs are resilient to many forms of numerical tinkering; they allow re-evaluation of design decisions made long ago at the bottom of the hardware stack with reduced fear of failure. The design space of hardware real number representations is indeed quite large and underexplored [22], as is as the opportunity to improve hardware efficiency and software simplicity with alternative designs and judicious use of numerical approximation. Log domain representations, posits, Kulisch accumulation and combinations such as ELMA show that floating point efficiency and applicability can be substantially improved upon. We plan on continuing investigation of this arithmetic design space at the hardware level with DNN training, and on general numerical algorithms in the future.

Acknowledgments

We thank Synopsys for their permission to publish baseline and comparative results obtained by using their tools and DesignWare components, so we could present realistic numbers on our research using a popular 28 nm semiconductor technology node.

References

[1] Z. Cai, X. He, J. Sun, and N. Vasconcelos. Deep learning with low precision by half-wave gaussian quantization.

[2] J. N. Coleman, E. Chester, C. I. Softley, and J. Kadlec. Arithmetic on the european logarithmic microprocessor. *IEEE Transactions on Computers*, 49(7):702–715, 2000.

[3] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[4] T. Dettmers. 8-bit approximations for parallelism in deep learning. *arXiv preprint arXiv:1511.04561*, 2015.

[5] R. DiCecco, L. Sun, and P. Chow. Fpga-based training of convolutional neural networks with a reduced precision floating-point library. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 239–242, Dec 2017.

[6] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, et al. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 1–14. IEEE Press, 2018.

[7] S. Galal, O. Shacham, J. S. Brunhaver II, J. Pu, A. Vassiliev, and M. Horowitz. Fpu generator for design space exploration. In *Computer Arithmetic (ARITH), 2013 21st IEEE Symposium on*, pages 25–34. IEEE, 2013.

[8] S. Golomb. Run-length encodings (corresp.). *IEEE transactions on information theory*, 12(3):399–401, 1966.

[9] Google. *TPU TensorFlow ops*. `https://cloud.google.com/tpu/docs/tensorflow-ops`.

[10] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015.

[11] J. Gustafson. *The End of Error: Unum Computing*. Chapman & Hall/CRC Computational Science. Taylor & Francis, 2015.

[12] J. L. Gustafson and I. T. Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations*, 4(2):71–86, 2017.

[13] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[14] M. Horowitz. 1.1 computing's energy problem (and what we can do about it). In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pages 10–14. IEEE, 2014.

[15] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[16] N. G. Kingsbury and P. J. Rayner. Digital filtering using logarithmic arithmetic. *Electronics Letters*, 7(2):56–58, 1971.

[17] U. Köster, T. Webb, X. Wang, M. Nassar, A. K. Bansal, W. Constable, O. Elibol, S. Gray, S. Hall, L. Hornof, et al. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In *Advances in Neural Information Processing Systems*, pages 1742–1752, 2017.

[18] U. Kulisch. *Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units*. Springer mathematics. Springer Vienna, 2002.

[19] U. Kulisch. *Computer Arithmetic and Validity: Theory, Implementation, and Applications*. De Gruyter Studies in Mathematics. De Gruyter, 2012.

[20] F. Li and B. Liu. Ternary weight networks. *CoRR*, abs/1605.04711, 2016.

[21] D. Lin, S. Talathi, and S. Annapureddy. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning*, pages 2849–2858, 2016.

[22] P. Lindstrom, S. Lloyd, and J. Hittinger. Universal coding of the reals: alternatives to ieee floating point. In *Proceedings of the Conference for Next Generation Arithmetic*, page 5. ACM, 2018.

[23] S. Migacz. 8-bit inference with tensorrt. *Nvidia GTC*, 2017.

[24] D. Miyashita, E. H. Lee, and B. Murmann. Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025*, 2016.

[25] R. Morris. Tapered floating point: A new floating-point representation. *IEEE Transactions on Computers*, 100(12):1578–1579, 1971.

[26] J.-M. Muller, F. De Dinechin, C.-P. Jeannerod, S. Torres, et al. *Handbook of floating-point arithmetic*. Springer, 2010.

[27] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[28] E. Quinnell, E. E. Swartzlander, and C. Lemonds. Floating-point fused multiply-add architectures. In *Signals, Systems and Computers, 2007. ACSSC 2007. Conference Record of the Forty-First Asilomar Conference on*, pages 331–337. IEEE, 2007.

[29] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[30] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[31] E. E. Swartzlander and A. G. Alexopoulos. The sign/logarithm number system. *IEEE Transactions on Computers*, 100(12):1238–1242, 1975.

[32] M. B. Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1131–1136. IEEE, 2012.

[33] A. Tulloch and Y. Jia. High performance ultra-low-precision convolutions on mobile devices. *CoRR*, abs/1712.02427, 2017.

[34] Y. Uguen and F. De Dinechin. Design-space exploration for the kulisch accumulator. 2017.

[35] V. Vanhoucke, A. Senior, and M. Z. Mao. Improving the speed of neural networks on cpus. Citeseer.

[36] J. H. Wilkinson. *Rounding errors in algebraic processes*. Prentice-Hall, 1963.

[37] D. Zuras, M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo, et al. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

Annex B

Posit Standard Documentation
Release 3.2-draft

Posit Working Group

Jun 23, 2018

Contents

1 Overview **3**
1.1 Scope . . . 3
1.2 Purpose . . . 3
1.3 Inclusions . . . 3
1.4 Requirements vs. Recommendations . . . 3
1.5 Programming Environment . . . 3

2 Definitions, abbreviations, and acronyms **4**
2.1 Definitions . . . 4

3 Posit and quire formats **6**
3.1 Overview . . . 6
3.1.1 Formats . . . 6
3.1.2 Compliance . . . 6
3.1.3 Represented data . . . 6
3.2 Binary interchange format encoding . . . 7
3.2.1 Posit format encoding . . . 7
3.2.2 Quire format encoding . . . 8

4 Rounding **9**
4.1 Definition and Method . . . 9
4.2 Fused Operations . . . 9

5 Operations **10**
5.1 Guiding principles . . . 10
5.2 Mathematical functions . . . 10
5.2.1 Elementary functions of one argument . . . 10
5.2.2 Elementary functions of two arguments . . . 10
5.2.3 Comparison functions of two arguments . . . 10
5.2.4 Functions of one argument . . . 10
5.2.5 Functions of two posit arguments . . . 11
5.2.6 Functions of a posit argument and an integer argument . . . 11

6 Conversion to and from character format **12**
6.1 Guiding principles . . . 12

7 Language support **13**
7.1 Guiding principles . . . 13

Standard for Posit Arithmetic

Sponsor

Agency for Science, Technology and Research (A*STAR)

Abstract

This standard specifies the storage format, operation behavior, and required mathematical functions for posit arithmetic in computing environments. It describes the binary storage used by the computer and the human-readable character input and output for posit representation. They may be realized in software or hardware or any combination of the two. A system that meets this standard is said to be *posit compliant* and will produce results that are identical to any other posit compliant system.

Keywords

Arithmetic, binary, exponent, format, fraction, NaR, number rounding, quire, regime

Participants

The following people in the Posit Working Group contributed to the development of this standard:
John Gustafson, *Chair*
Gerd Bohlender
Vassil Dimitrov
Siew Hoon Leong (Cerlane)
Peter Lindstrom
Theodore Omtzigt
Andrew Shewmaker
Isaac Yonemoto
Other valued contributors include:
Shin Yee Chung
Geoff Jones

1 Overview

1.1 Scope

This standard specifies the storage format and mathematical behavior of posit numbers, and the set of functions a posit arithmetic system must support, including basic arithmetic operations. It includes a description of how results are to be rounded, what situations generate an exception, and whether those exceptions are handled by the implementation or by the user.

1.2 Purpose

This standard provides a system for computing with real numbers represented in a computer using fixed-size binary values. Deviations from mathematical behavior (including loss of accuracy) are kept to a minimum while preserving the ability to represent a wide dynamic range of values. All features are accessible by programming languages; the source program and input data are sufficient to specify the output exactly on any computer system, similar to the way 2's complement integer arithmetic produces bitwise-identical results.

1.3 Inclusions

This standard specifies:

- Formats for binary data, for computation and data interchange
- Addition, subtraction, multiplication, division, dot product, compare, and other operations
- Mathematical functions such as logarithm, exponential, trigonometric, and hyperbolic functions
- Conversions of other number representations to posit format
- Conversions between different posit formats
- Exception handling when a result is not a real number (NaR).

1.4 Requirements vs. Recommendations

All descriptions herein are requirements of the behavior of the system. The decision of how to satisfy the requirements (using any combination of hardware and software) is up to the implementer of this standard, but all functionality must be provided and behave as described for a system to be posit-compliant.

1.5 Programming Environment

A programming environment may claim to be compliant with this standard if it supports at least one of the four precisions (8, 16, 32, 64) completely. If it includes more than one precision, then it must also provide the ability to convert between those precisions.

2 Definitions, abbreviations, and acronyms

2.1 Definitions

correct rounding This standard's method of converting an infinitely precise value to a posit. A posit so obtained is said to be correctly rounded.

es exponent size. The maximum number of bits 0, 1, 2, 3, . . . that are available for expressing the exponent.

exponent The part of the power-of-two scaling determined by the exponent bits.

exponent bits A field of bits within a posit that, in combination with the regime bits, determines the power-of-two scaling of the fraction.

exponent size es

format A set of bits and the definition of their meaning.

fraction The component of a posit containing its significant binary digits after the binary point; $0 \leq fraction < 1$.

hidden bit An assumed 1 bit before the MSB of the fraction.

LSB least significant bit

lg logarithm base 2

MSB most significant bit

maxpos The largest real value expressible as a posit.

minpos The smallest nonzero value expressible as a posit.

NaR Not a real. A value that has infinite magnitude, is indeterminate, is multi-valued, or requires an imaginary component to express (like $\sqrt{-1}$) is represented as `NaR` .

nbits number of bits. The precision of a posit format, the total number of bits (8, 16, 32, or 64).

not a real number `NaR`

number of bits `nbits`

pintmax posit integer maximum. The largest consecutive integer expressible as a posit.

posit A real number that is exactly representable using a fixed number of bits in the format described in this standard, or a `NaR` .

precision The number of bits available for expressing a quantity.

quire A fixed-point format capable of storing sums of products of posits without rounding, up to some large number of such products.

regime A subfield of a posit consisting of some number of identical bits terminated by the opposite bit or the end of the number, that contributes to the specification of the power-of-two scaling of the fraction.

sign The value +1 for positive numbers, -1 for negative numbers. Exception values 0 and `NaR` have no sign.

sign bit The MSB of a posit or quire, 0 or 1.

significand The implicit 1 bit followed by the fraction bits; $1 \leq significand < 2$.

universal number unum

unum universal numbers express real numbers (posits) and ranges of real numbers (valids).

unum seed useed

useed unum seed. A value obtained by starting with 2 and squaring repeatedly *es* times: 2, 4, 16, 256, . . . influencing the way the projective real circle of unums gets populated.

property	**posit8**	**posit16**	**posit32**	**posit64**
Max significand bits	6	13	28	59
Max exponent bits, `es`	0	1	2	3
`minpos`	$2^{-6} \approx 1.5 \times 10^{-2}$	$2^{-28} \approx 3.7 \times 10^{-9}$	$2^{-120} \approx 7.5 \times 10^{-37}$	$2^{-496} \approx 4.9 \times 10^{-150}$
`maxpos`	$2^{6} \approx 6.4 \times 10^{1}$	$2^{28} \approx 2.7 \times 10^{8}$	$2^{120} \approx 1.3 \times 10^{36}$	$2^{496} \approx 2.0 \times 10^{149}$
`pintmax`	8	256	2^{22}	2^{52}
quire bits	32	128	512	2048
Exact sum quire limit	32767	$2^{43} - 1$	$2^{151} - 1$	$2^{559} - 1$
Exact dot product quire limit	127	32767	$2^{31} - 1$	$2^{63} - 1$

Table 1: Properties of posit formats

3 Posit and quire formats

3.1 Overview

3.1.1 Formats

This clause defines posit formats, which are used to represent a finite set of real numbers. Posit formats are specified by their precision, *nbits*. For each posit format, there is also a format of size $nbits^2/2$ that is used to contain exact sums of products of posits. All properties such as dynamic range, accuracy, quire size and format, are determined solely by the precision.

There are four precisions described in this standard: 8, 16, 32, and 64. We sometimes refer to the four corresponding formats as posit8, posit16, posit32, and posit64.

3.1.2 Compliance

An implementation is compliant with this standard if it supports full functionality of at least one precision (8, 16, 32, or 64). If the implementation supports more than one precision, then it must support conversions between the precisions that it supports.

Note: If hardware supports posit multiplication, addition, subtraction, and the quire, all remaining functionality can be supported with software.

3.1.3 Represented data

Within each format, a posit represents either NaR, or a number of the form $m \times 2^n$, where m and n are integers limited to a range symmetrical about and including zero. The maximum m range is $-2^p < m < 2^p$ where $p = nbits - lg(nbits) + 1$ is the maximum number of significant digits (bits).

The smallest positive posit, *minpos*, is $2^{\frac{1}{8}\text{nbits}(2-nbits)}$ and the largest positive posit, *maxpos*, is the reciprocal of *minpos*, or $2^{\frac{1}{8}\text{nbits}(nbits-2)}$. Every posit is an integer multiple of *minpos*. Every real number maps to a unique posit representation; there are no redundant representations.

The quire represents either NaR or an integer multiple of $minpos^2$, represented as a 2's complement binary number with $2^{\text{nbits}^2/2}$ bits. This enables it to add a list of posits or a list of exact products of posits without rounding error and thereby satisfy the associative and distributive laws of algebra up to some minimum length. Sums of lists longer than that minimum are capable of integer overflow.

Posits can exactly express all integers i in a range $-pintmax \leq i \leq pintmax$; outside that range, integers exist that cannot be expressed as a posit without rounding to a different integer.

The values for the posit formats are summarized in properties-table.

The exact sum quire limit and exact dot product quire limit are the number of additions or multiplication-additions up to which the quire cannot overflow. Up to these limits, the quire obeys the associative law of addition and the distributive law.

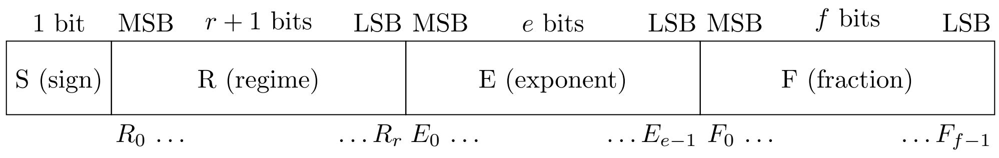


Figure 1: General binary posit format

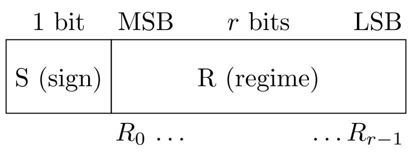


Figure 2: Binary posit format with zero-length exponent and fraction

3.2 Binary interchange format encoding

3.2.1 Posit format encoding

All posits have just one encoding in a binary interchange format shown in posit_format_encoding-general and posit_format_encoding-signedregime. The four fields are:

1. Sign bit S
2. Regime R consisting of r bits identical to R_0, terminated by $1 - R_0$ ($r+1$ bits total length) or the end of the posit (r bits total length).
3. Exponent E represented by e exponent bits, terminated by a maximum of es or the end of the posit
4. Fraction F represented by f fraction bits, terminated by the end of the posit

The meaning of each field is as follows:

1. S is its literal value, 0 or 1.
2. R is $-r$ if R_0 is 0, and $r-1$ if R_0 is 1.
3. E is an es-bit unsigned integer, with 0 bit padding in the least significant bits if the exponent field has fewer than es bits because of the regime length.
4. F represents an unsigned integer divided by 2^f.

Note

The exponent field size e and fraction field size f can each be 0, in which case they represent 0; $0 \leq e \leq es$ and $0 \leq f \leq nbits - lg(nbits)$. The hidden bit is 1 even if f is 0.

The representation (S, R, E, F) of the posit and value v of the datum represented are inferred from the fields as follows:

1. If $S = 0$ and all other fields contain only 0 bits, then $v = 0$.
2. If $S = 1$ and all other fields contain only 0 bits, then v is NaR and undefined.
3. If any bits in the (R, E, F) are 1, then $(1 - 3S + F) \times 2^{(-1)^S (R \times 2^{es} + E + S)}$.

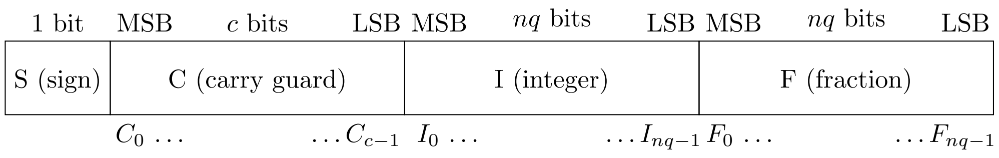


Figure 3: Binary quire format

3.2.2 Quire format encoding

The quire is a fixed-point 2's complement value of length $nbits^2/2$ which is 32, 128, 512, or 2048 bits for the posit sizes 8, 16, 32, and 64 respectively.

The number of bits for the fraction is $nq = 1/4nbits^2 - 1/2nbits$. The integer part also has nq bits. The carry guard has $c = nbits - 1$ bits to guarantee that sums of products cannot overflow, up to $2^{nbits-1} - 1$ products.

The representation (S, C, I, F) of the quire and value v of the datum represented are inferred from the fields as follows:

1. If $S = 1$ and all other fields contain only 0 bits, then v is NaR and undefined.
2. For all other cases, the value v is the 2's complement signed integer represented by all bits, divided by 2^{nq}.

4 Rounding

4.1 Definition and Method

Rounding is the substitution of an expressible posit for any exact real number that is not expressible as a posit. The results of all operations are regarded as mathematically exact prior to rounding.

The method for rounding a real value x is as follows:

1. If x is exactly expressible as a posit, it is unchanged.
2. If $|x| > maxpos$, x is rounded to `sign(x) * maxpos`.
3. If $0 < |x| < minpos$, x is rounded to `sign(x) * minpos`.
4. For all other values, the value is rounded to the nearest binary value if the posit were encoded to infinite precision beyond the *nbits* length; if two posits are equally near, the one with binary encoding ending in 0 is selected.

Note: Rule (4) has the effect of rounding to the posit with the nearest logarithm when the dropped bit is an exponent bit, and to the nearest posit by absolute difference in other cases.

4.2 Fused Operations

A fused operation is an expression with two or more operations that is not rounded until the entire expression is evaluated exactly. Fused operations are distinct from non-fused operations and must be explicitly requested in a posit-compliant programming environment. Fused operations are those expressible as sums and differences of the exact product of two posits; no other fused operations are allowed.

All fused operations can be performed as accumulations in a . A particular posit environment may perform fused operations without using a quire, but may not fuse any operations that cannot be performed as exact dot products of vectors with posit components. Exact sums are dot products where one vector consists of all 1 values. The fused multiply-add operation $ab + c$ is a dot product of vectors $(a, 1)$ and (b, c). A complex product $(a + bi) \times (c + di)$ can be performed as two fused operations, $ac - bd$ and $ad + bc$. An expression such as abc or ab/d is not in the form of the sum or difference of products and may not be fused.

5 Operations

5.1 Guiding principles

If `NaR` is the input to an operation, the result is also `NaR` . For a function $f(p)$ of a posit p, the mathematical value y is the limit $f(x)$ as x approaches p within the domain of the function, from any direction. If that limit is a determinate real number, the operation should return the closest posit to y using the rounding rules of clause 4. Similarly, for functions of more than one argument; the limits for each argument are taken without correlation to one another. If the limit is not a real number, the result is `NaR` .

Many functions are identical to standard 2's complement integer operations. The internal processor flags for those posit operations behave identically.

5.2 Mathematical functions

The following functions shall be supported, with correct rounding for all input arguments per clause 4. If cases can produce `NaR` from non-NaR inputs, the function description notes those cases.

5.2.1 Elementary functions of one argument

negate(*posit*) is identical to 2's complement integer negation.

abs(*posit*) is identical to 2's complement integer absolute value.

round(*posit*) converts *posit* to the nearest posit with integer value, and the nearest even integer if two integers are equally far from *posit*.

sign(*posit*) returns 1 if the value of *posit* is positive, and -1 if the value of *posit* is 1. If *posit* is zero or NaR, sign(*posit*) returns 0.

5.2.2 Elementary functions of two arguments

addition(posit1, posit2) returns `posit1` + `posit2`, rounded

subtraction(posit1, posit2) returns `posit1 - posit2`, rounded

multiplication(posit1, posit2) returns `posit1 * posit2`, rounded

division(posit1, posit2) returns `NaR` if `posit2` is 0, else `posit1 / posit2`, rounded

5.2.3 Comparison functions of two arguments

All comparison functions are identical to the comparisons of the posit bit strings regarded as 2's complement integers, so there is no need for separate machine-level instructions. The value `NaR` has the bit string of the most negative integer, so `NaR < posit` returns True if `posit` is not `NaR` . The posit environment shall support:

```
boolean compareEqual(posit1, posit2)
boolean compareNotEqual(posit1, posit2)
boolean compareGreater(posit1, posit2)
boolean compareGreaterEqual(posit1, posit2)
boolean compareLess(posit1, posit2)
boolean compareLessEqual(posit1, posit2)
```

Note: Testing if a posit is `NaR` is not an exception: **compareEqual**(NaR, *posit*).

5.2.4 Functions of one argument

sqrt(*posit*) returns `NaR` if $posit < 0$, else the square root of *posit*, rounded.
rSqrt(*posit*) returns `NaR` if $posit < 0$, else 1/sqrt(*posit*), rounded.
exp(*posit*) returns e^{posit}, rounded.
expm1(*posit*) returns $e^{posit} - 1$, rounded.

exp2(*posit*) returns 2^{posit}, rounded.
exp2m1(*posit*) returns $2^{posit} - 1$, rounded.
exp10(*posit*) returns 10^{posit}, rounded.
exp10m1(*posit*) returns $10^{posit} - 1$, rounded.
log(*posit*) returns $log_e(posit)$, rounded. Returns NaR if $posit \leq 0$.
logp1(*posit*) returns $log_e(posit + 1)$, rounded. Returns NaR if $posit \leq -1$.
log2(*posit*) returns $log_2(posit)$, rounded. Returns NaR if $posit \leq 0$.
log2p1(*posit*) returns $log_2(posit + 1)$, rounded. Returns `NaR` if $posit \leq -1$.
log10(*posit*) returns $log_{10}(posit)$, rounded. Returns NaR if $posit \leq 0$.
log10p1(*posit*) returns $log_{10}(posit + 1)$, rounded. Returns `NaR` if $posit \leq -1$.
sin(*posit*) returns sin(*posit*), rounded.
sinPi(*posit*) returns $\sin(\pi \times posit)$, rounded.
cos(*posit*) returns cos(*posit*), rounded.
cosPi(*posit*) returns $\cos(\pi \times posit)$, rounded.
tan(*posit*) returns tan(*posit*), rounded.
tanPi(*posit*) returns $\tan(\pi \times posit)$, rounded.
asin(*posit*) returns `NaR` if $|posit| > 1$, else arcsin(*posit*), rounded.
asinPi(*posit*) returns `NaR` if $|posit| > 1$, else arcsin(*posit*) / π, rounded.
acos(*posit*) returns `NaR` if $|posit| > 1$, else arccos(*posit*), rounded.
acosPi(*posit*) returns `NaR` if $|posit| > 1$, else arccos(*posit*) / π, rounded.
atan(*posit*) returns arctan(*posit*), rounded.
atanPi(*posit*) returns arctan(*posit*) / π, rounded.
sinh(*posit*) returns sin(*posit*), rounded.
cosh(*posit*) returns cos(*posit*), rounded.
tanh(*posit*) returns tan(*posit*), rounded.
asinh(*posit*) returns `NaR` if $|posit| > 1$, else arcsin(*posit*), rounded.
acosh(*posit*) returns `NaR` if $|posit| > 1$, else arccos(*posit*), rounded.
atanh(*posit*) returns arctan(*posit*), rounded.

5.2.5 Functions of two posit arguments

hypot(*posit*1, *posit*2) returns the square root of $posit1^2 + posit2^2$, rounded.
pow(*posit*1, *posit*2) returns $posit1^{posit2}$, rounded. (exceptions...)
atan2(*posit*1, *posit*2) atan2Pi**(*posit*1, *posit*2)

5.2.6 Functions of a posit argument and an integer argument

compound(*posit*, *n*) returns `NaR` if $x \leq -1$, else $(1 + x)^n$, rounded
pown(*posit*, *n*) returns `NaR` if ...,
rootn(*posit*, *n*)

6 Conversion to and from character format

6.1 Guiding principles

Complete section on conversion to and from character format

7 Language support

7.1 Guiding principles

Complete section on language support