Deep Learning with approximate computing: an energy efficient approach

Gonçalo Eduardo Cascalho Raposo

Instituto Superior Técnico, Universidade de Lisboa, Portugal

January 2021

Abstract

Faster and energy-efficient deep learning implementations will certainly benefit various application domains, especially those deployed in systems with energy and payload limitations, such as aerial and space devices. Under this premise, low-precision formats have proven to be an efficient way to reduce not only the memory footprint but also the hardware resources and power consumption of deep learning computations. For this purpose, the posit numerical format stands out as a highly viable substitute for the IEEE floating-point, but its application to neural networks training still requires further research. Some preliminary results have shown that 8-bit (and even smaller) posits may be used for inference and 16-bit for training while maintaining the model accuracy. The presented work aims to evaluate the feasibility to train deep convolutional neural networks using posits, focusing on precisions less than or equal to 16 bits. For such purpose, a software framework was developed to use simulated posits and quires in end-to-end deep learning training and inference. This implementation allowed to train and test deep learning models using any bit size, configuration, and even mixed-precision, suitable for different precision requirements in various stages. Additionally, a variation of the posit format able to underflow was also evaluated for low-precision posits. The obtained results suggest that 8-bit posits can replace 32-bit floats in a mixed low-precision posit configuration for the training phase, with no negative impact on the resulting accuracy. Moreover, enabling posits to underflow increased their testing accuracy for very low precisions.

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

1. Introduction

Deep Learning (DL) is, nowadays, one of the hottest topics in signal processing research, spanning across multiple applications. This is a highly demanding computational field, since, in many cases, better performance and generality result in increased complexity and deeper models [1]. For example, the recently published language model GPT-3, the largest ever trained network with 175 billion parameters, would require 355 years and $4.6M to train on a Tesla V100 cloud instance [2]. Therefore, it is increasingly important to optimize the energy consumption required by the training process. Although algorithmic approaches may contribute to these goals, computing architectures advances are also fundamental [3].

The computations involved in DL mostly use the IEEE 754 single-precision (SP) floating-point (FP) format [4], with 32 bits. However, recent research has achieved comparable precision with smaller numerical formats. The novel posit format [5], designed as a direct drop-in replacement for float (i.e. IEEE SP FP), provides a wider dynamic range, higher accuracy, and simpler hardware. Moreover, each posit format has a corresponding exact accumulator, named quire, which is particularly useful for the frequent dot products in DL calculations.

Contrasting with the IEEE 754 FP, the posit numerical format may be used with any size and has been shown to be able to provide more accurate operations than floats, while using fewer bits and, consequently, less energy [6]. Posits may even use sizes that are not multiples of 8, which could be exploited in Field Programmable Gate Arrays (FPGA) or Application Specific Integrated Circuits (ASIC) to obtain optimal efficiency and performance. Moreover, the fast and energy-efficient characteristics of the posit arithmetic make it a good approach for efficient real-time deep learning applications on embedded devices [7].

However, most published studies regarding the application of the posit format to Deep Neural Networks (DNNs) rely on the inference stage [7–11]. The models are trained using floats and are later quantized to posits to be used for inference. Nevertheless, the inference phase tends to be less sensitive to errors than the training phase, making it easier to achieve good performance using {5..8}-bit posits.

In contrast, exploiting the use of posits during the training phase is a more compelling topic since this is the most computationally demanding stage. The first time posits were used in this context was in [12], by training a Fully Connected Neural Network (FCNN) for a binary classification problem using {8, 10, 12, 16, 32}-bit posits. Later, in [13], a FCNN was trained for MNIST and Fashion MNIST using {16, 32}-bit posits. In [14], Convolutional Neural Networks (CNNs) were

trained using a mix of {8, 16}-bit posits, but still relying on floats for the first epoch and layer computations. More recently, in [15], a CNN was trained for CIFAR-10 but using only {16, 32}-bit posits.

Nonetheless, the existing studies fail to train DNN with posits precisions smaller than 16 bits without affecting the achieved model accuracy. Moreover, they are all in accordance with the Posit Standard [16], however, there is a more recent version [17] that has not yet been officially released[1] nor properly evaluated.

Under the premise of these previous works and the identified knowledge gap, the research that is now presented goes a step further by extending the implementation of DNNs with posits in a more general and feature-rich approach. Hence, the original contributions of this paper are:

- Proposal of new techniques to improve the performance achieved by DNN using low-precision posits (e.g. mixed-precision, underflow, etc.);
- open-source framework[2] to natively perform inference and training with posits of any precision (number of bits and exponent size) and quires; it was developed in C++ and adopts a similar API as PyTorch, with multithreading support;
- Results of conventional CNNs trained and tested with ≤8-bit posits replacing 32-bit floats;
- First known analysis of the most recent Posit Standard and its usage for DNNs.

The rest of this paper is organized as follows. Section 2 presents a brief introduction about DL and describes the conventional IEEE 754 floating-point format and the novel posit format. Then, in Section 3, a DL framework cable of end-to-end training and testing with posits is proposed, along with some of its most important details. Using this framework, Section 4 presents a thorough evaluation of the accuracy of the posit format for DNN training, as well as a few techniques to improve the accuracy achieved with low-precision formats. Section 5 applies that acquired knowledge to train and test CNNs with ≤8-bit posits on various image classification datasets. Finally, Section 6 concludes with some final remarks about this work and future studies.

2. Background

To better comprehend deep learning with approximate computing, it is important to first understand its two main topics: deep learning and the posit numbering system, which will be used with low-precision formats. This section will briefly explain these topics.

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

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

2.1. Deep Learning

Deep learning is a specific subfield of machine learning – models that learn useful representations from input data. In DL, the models are structured in what is called a Neural Network (NN), generally represented as in Figure 1. A NN consists in groups of nodes that are connected and relate through Equation (1), where z_i^l is an intermediate output of the i-th neuron of the layer l, w_{ij}^l are the weights that relate the output of previous neurons a_j^{l-1} to the current one a_i^l, b_i^l is a bias, and g is a non-linear activation function.

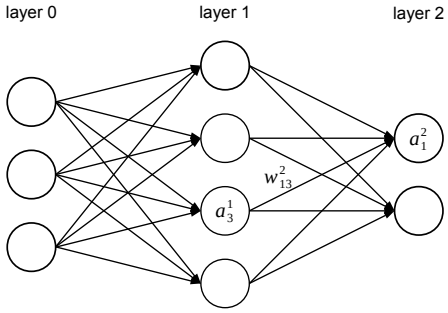


Figure 1: Example of a 2-layer neural network of arbitrary dimensions.

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

$$a_i^l = g\left(z_i^l\right), \tag{1b}$$

The process of applying Equation (1) for the entirety of the network (left to right) to obtain an output is referred to as *inference*. Conversely, the process of updating the network parameters to achieve better performance is referred to as *training* and it is no more than a minimization of a loss/cost function. This minimization is achieved by updating the network weights in the opposite direction of the corresponding gradients – gradient descent algorithm. The gradients with respect to each weight are calculated by propagating the loss function gradient in the backward direction (right to left), using the chain rule – backpropagation. Training is, therefore, more computationally demanding compared to inference, since it also performs an inferring step followed by the optimization of the parameters.

The mathematical theory is relatively simple, which in conjunction with the ability to use many successive layers (DNN), allows the model to learn quite complex relationships and achieve state-of-the-art performance in various domains. A common and useful abstraction is to think of the NN as a block diagram, where each layer is represented by a block that receives a multi-dimensional input, applies a function, and passes its output to the next block(s). Figure 2 illustrates the block diagram representation of the same 2-layer NN.

Nowadays, there are several DL frameworks available that offer all the necessary tools to design, train, and evaluate DNN models. The two most popular frameworks are: PyTorch and TensorFlow. While the latter

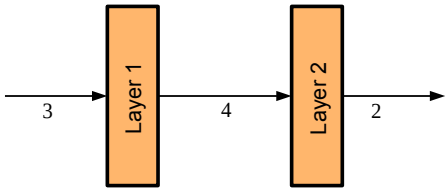


Figure 2: Block diagram of the 2-layer NN example of Figure 1, displaying the input and output dimensions.

framework emerged earlier, which gave it more time to mature and to build a good reputation, PyTorch is becoming the most popular framework for research [18]. Moreover, PyTorch uses a dynamic graph definition, which gives it a more imperative coding style and provides the user greater control of the execution flow.

2.2. Posit Numbering System

Among the several different numbering formats that have been proposed to represent real numbers [19], the IEEE 754 single-precision floating-point (float) is the most widely adopted. It decomposes a number into a sign (1 bit), exponent (8 bits) and mantissa (23 bits):

$$f = (-1)^{\text{sign}} \times 2^{\text{exponent}-127} \times \text{mantissa}. \quad (2)$$

However, it has also been observed that many application domains do not need nor make use of the total accuracy and wide dynamic range that is made available by IEEE 754, often compromising the resulting system optimization in terms of hardware resources, performance, and energy efficiency. One of such domains is DNN training, where most of the computations are zero-centered.

To overcome these issues, the Posit numbering system [5] was recently proposed as a new alternative to IEEE 754. Posit($nbits$, es) is characterized by an arbitrary fixed size/number of bits ($nbits$) and an exponent size (es), being composed by the following fields: sign (1 bit), regime (variable bits), exponent (0..es bits), and fraction (remaining bits) [16]. It is decoded as in Equation (3).

$$p = (-1)^{\text{sign}} \times 2^{2^{es}\times k} \times 2^{\text{exponent}} \times (1 + \text{fraction}). \quad (3)$$

When the number is negative, the two's complement has to be applied before decoding the other fields. The regime bits are decoded by measuring k, determined by their run-length.

A particular characteristic of Posit is that numbers near to 1 (in magnitude) have more accuracy than extremely large or extremely small numbers – tapered accuracy (see Figure 3). Another interesting point is the definition of the quire, a Kulisch-like large accumulator [20] designed to contain exact sums of products of posits. Table 1 shows the recommended posit and quire configurations and their characteristics [16].

More recently, a draft version [17] of an update to the Posit Standard introduced some changes to the posit format, namely, the hyperparameter for the maximum exponent size was fixed to $es = 2$. This es will provide

Table 1: Main properties of recommended posit formats according to the current standard [16].

nbits	8	16	32	n
es	0	1	2	e
dynamic range	$2^{\pm 6}$	$2^{\pm 28}$	$2^{\pm 120}$	$2^{\pm 2^e \times (n-2)}$
quire bits	32	128	512	$n^2/2$
dot prod limit	127	32767	$2^{31}-1$	$2^{n-1}-1$

Table 2: Main properties of posit formats according to the more recent standard [17].

nbits	8	16	32	n
es	⟵		2	⟶
dynamic range	$2^{\pm 24}$	$2^{\pm 56}$	$2^{\pm 120}$	$2^{\pm(4n-8)}$
quire bits	128	256	512	$16n$
dot product limit	⟵		$2^{31}-1$	⟶

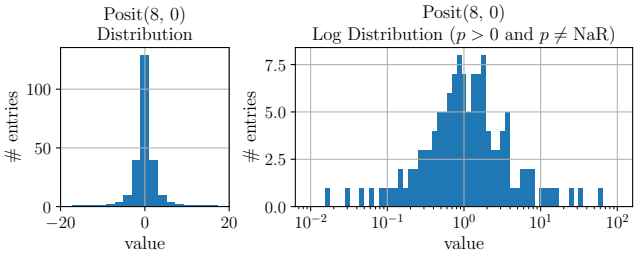


Figure 3: Linear and log distribution of posit(8, 0).

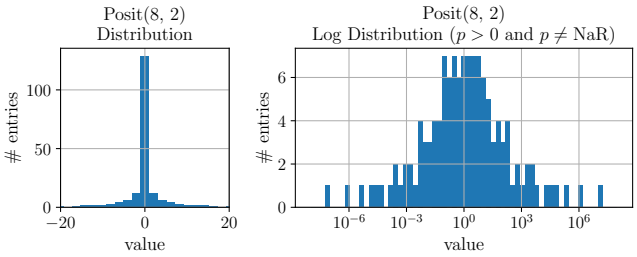


Figure 4: Linear and log distribution of posit(8, 2).

low-precision posits a much wider dynamic range (see Figure 4). Moreover, it greatly simplifies the conversion between posits of different precisions: pad it with 0s to make it larger, and round-off the least significant bit to make it smaller, without having to decode the posit fields. This simple conversion is particularly convenient for implementations with uneven posit precisions. Furthermore, the size of the corresponding quire format was also modified so that it can now accumulate exact sums of at least $2^{31}-1$ products of posits. Table 2 shows the properties of the posit and quire formats according to this version of the standard.

Preliminary studies indicate that the area and energy of a posit compliant unit are comparable to its IEEE 754 compliant counterpart [6]. Moreover, low-precision posit formats may be used to achieve comparable accuracy and dynamic range as larger IEEE 754 formats [21], hence being more energy-efficient. Recently, the reconfigurable tensor unit leveraging the posit format proposed in [22] evidenced how this format can be used to outperform state-of-the-art DL focused units.

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


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

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









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

```
#include <positnn/positnn>

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

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

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

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

- **Activation functions:**
  ReLU, Sigmoid, TanH
- **Layers:**
  Linear, Convolutional, Average and Maximum Pooling, Batch Normalization, Dropout
- **Loss functions:**
  Mean Squared Error (MSE), Cross Entropy
- **Optimizer:**
  Stochastic Gradient Descent (SGD)
- **Utilities:**
  StdTensor, convert PyTorch tensors, mixed precision tensor, save and load model, scaled gradients, quires and underflow

Figure 5: Comparison of PyTorch (left) and PositNN (center). Implemented functionalities of PositNN (right).

3. Deep Learning Posit Framework

Current DNN frameworks do not natively support the posit data type. As a result, the whole set of functions and operators would need to be reimplemented, in order to take advantage of this new numbering system. As such, it was decided to develop an entirely new framework, from scratch, in order to ensure better control of its inner operations and exploit them for the posit data format.

3.1. Posit Neural Network Framework

The developed framework, named PositNN, was based on the PyTorch API for C++ (LibTorch), thus inheriting its program functions and data flow. As a result, any user familiar with PyTorch may easily port their networks and models to PositNN. As an example, a comparison between PyTorch and the proposed framework regarding the declaration of a 1-layer model is shown in Figure 5 (left and center). The overall structure and functions are very similar, the only difference being the declaration of the backward function, since the proposed framework does not currently support automatic differentiation.

Despite being compared against a full-fledged framework, like PyTorch, the proposed framework is also capable of performing DNN inference and training with the most common models and functions. A complete list of the supported functionalities is also shown in Figure 5 (right). Thus, common CNNs, such as LeNet-5, CifarNet, AlexNet, and others, are fully supported. Moreover, the framework allows the user to extend it with custom functions or combine it with existing ones (e.g. from PyTorch). The proposed framework could also be adapted to support other data types, since most functions are independent of the posit format, except those that use the quire to accumulate.

3.2. Posit Tensor

Among the several libraries already available to implement posit operators in software [23], the Universal[3] library was selected, thanks to its comprehensive support for any posit configuration and quires. Furthermore, C++ classes and function templates are generically used to implement different posit precisions. Therefore, declaring a posit(8, 0) variable `p` equal to 1 is as simple as:

```
#include <universal/posit/posit>
sw::unum::posit<8, 0> p = 1;
```

Moreover, all the main operations specified in the Posit Standard [16] are fully supported and implemented. Furthermore, the proposed library adopts bitwise operations whenever possible, thus avoiding operating with intermediate float representations, since this could introduce errors regarding a native implementation.

Since posit variables are represented with a custom class, there was no available libraries that supported declaring multidimensional posit arrays and to operate with them, particularly, that exploited quires for accumulations. Therefore, posit tensors are stored with a custom class called StdTensor, which was implemented using only the C++ Standard Library. Data is internally stored in a one-dimensional dynamic vector and the multidimensional strides are automatically accounted for. For example, declaring a tensor of posit(8, 0) variables with shape $\{2, 2, 2\}$ and setting its first element to 1 can be achieved with:

```
#include <positnn/positnn>
StdTensor<posit<8, 0>> tensor3d({2, 2, 2});
tensor3d[{0, 0, 0}] = 1;
```

In order to manipulate and operate with StdTensor as a multidimensional array, some common functions were implemented. The basic arithmetic operators ($+$, $-$, $*$, $/$) were implemented and can be used with two tensors, by performing an element-wise operation, or with a tensor and a scalar value. In addition, more complex operations such as matrix multiplication and 2D convolution were implemented, which were essential to support some of the most usual DL layers. The latter functions support accumulating with quires, which improves the accuracy of their outputs.

[3] Available at: https://github.com/stillwater-sc/universal

3.3. Deep Neural Networks

In order to implement DNNs that support the posit format, PositNN offers some of the most common DL layers and functions, focusing on CNNs models, which are widely used for image classification tasks. Once again, the layers and functions follow an object-oriented implementation, each having methods for the forward and backward propagations. These can then be combined in sequence to form a more complex model.

To train DL models, some functions to compute the loss and to perform the optimization step are additionally provided. The adopted training and inference procedures are illustrated in Figure 6, which can be summarized as:

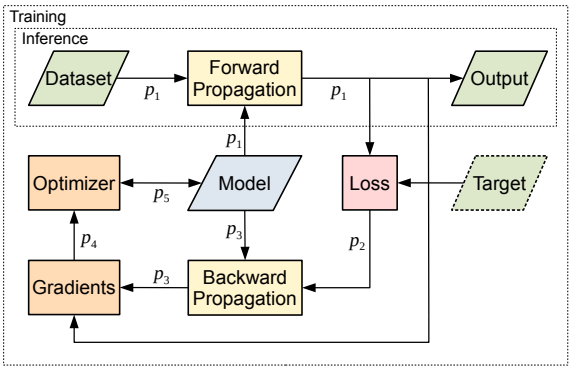


Figure 6: Block diagram of DNN training and inference. The various p_i, with $i = \{1..5\}$, represent the different posit precisions that may be used.

1. The dataset is loaded and converted to StdTensors of posits, which will be the input of the model.
2. The input samples are used for the forward propagation, resulting in the output/prediction. The inference procedure ends with this step.
3. Given the forward propagation output and the target values from the dataset, the loss value is computed, which measures the difference between the predicted and desired values. Then, the loss gradient, with respect to the last layer, is computed.
4. The loss gradient is then propagated through the entire network for the backward propagation. This is similar to the forward propagation, but in the reverse order and with the loss gradient as input. Each layer computes its output gradients, which will be used to calculate the gradient of its weights.
5. From the output gradients, the trainable layers then calculate the gradients with respect to their weights/parameters.
6. At last, having the gradients of each weight, the optimizer is responsible for updating the model parameters in the direction of decreasing loss. The model then assumes these updated weights, thus finishing a training iteration.

Given that some stages are more sensitive to numerical errors, the proposed framework supports different precisions per stage, as depicted in the arrows of Figure 6. Although not illustrated, it even allows the model to use different precisions per layer. To accomplish that, the weights are stored in a class whose members are copies with different posit configurations. Hence, each layer and stage converts its posit tensors to the appropriate precisions and seamlessly updates the copies after every change.

3.4. Parallelization of the proposed framework

In order to take the maximum advantage of the CPU, most functions were conveniently parallelized and implemented with multithreading support. Observing that most layers tend to operate on a mini-batch formed by independent samples, then the mini-batch can be divided by different threads/workers that jointly calculate the layer output. In matrix multiplication, this corresponds to assigning different rows of the output to be calculated in different threads (see Figure 7). The threads were implemented using std::thread, available since C++11, and the input and output data are passed by reference to avoid unnecessary copies.

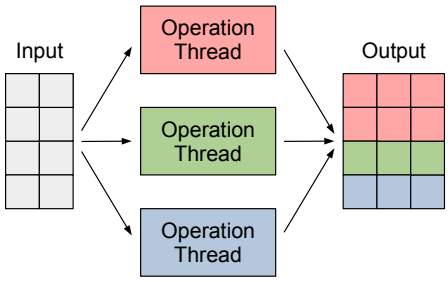


Figure 7: Load balancing for each thread/worker.

4. Training With Low-Precision Posits

By making use of the developed framework, the presented research started by studying how much can the posit precision be decreased without penalizing the DNN model accuracy. These results were compared to those obtained with 32-bit floats using PyTorch. In this evaluation, small accuracy differences ($< 1\%$) were assumed to be caused solely by the randomness of the training process and not exactly by the lack of precision of the numerical format.

4.1. Minimum Posit precision

For the initial evaluation, the 5-layer CNN LeNet-5 was trained on Fashion MNIST (a more complex dataset than the ordinary MNIST) during 10 epochs using the same posit precision everywhere. The 16-bit posit was the first evaluated format, since it was shown to seamlessly replace 32-bit float for DL [13, 15]. Then, the posit precision was decreased until 8-bit posit (see Table 3). A plot of the training progress is shown in Figure 8, comparing different posit configurations.

As expected, the 16-bit posits were able to achieve an accuracy similar to 32-bit floats. Moreover, although the accuracy decreased with the posit precision, 9-bit

Table 3: Evaluation of how different posit precisions compare to 32-bit float for DNN training.

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

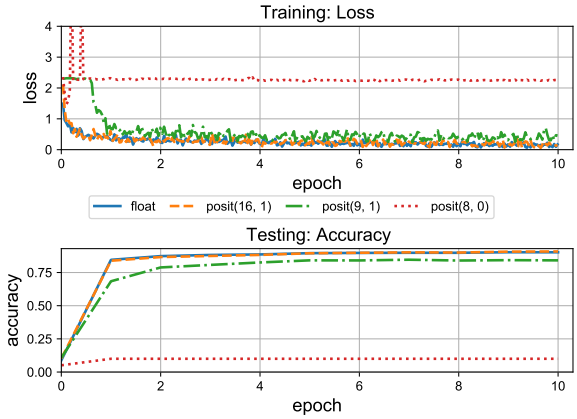


Figure 8: Training progress of LeNet-5 trained on Fashion MNIST with various posit precisions and float.

posits were still capable of achieving an acceptable accuracy (only $\sim 6\%$ less than 32-bit floats). However, it is noticeable that when the posit had an exponent size of $es = 0$, every model achieved a smaller accuracy or was even unable to converge, probably due to the narrow dynamic range (as seen in Figure 3). As for 8-bit posits, independently of the es, the model accuracy did not improve and stayed at $\sim 10\,\%$ (equivalent to randomly classifying a 10-class dataset).

4.2. Posit quires for intermediate accumulation

The numerical error produced in a simple operation performed with low-precision posits might not be that significant, however, when accumulated in long calculations such as matrix multiplications or convolutions, it might severely undermine the accuracy of the result. The Posit Standard defines the quire format, a large accumulator designed to accumulate exact dot products by avoiding intermediate posit quantization errors. To better grasp how useful it is, suppose the following accumulation performed using posit(8, 0):

$$2 \times 10 + 2 \times 10 + 2 \times 10 + 2 \times 2 \tag{4a}$$
$$\rightarrow 16 + 16 + 16 + 4 \tag{4b}$$
$$\rightarrow 32 + 16 + 4 \tag{4c}$$
$$\rightarrow 32 + 4 \tag{4d}$$
$$\rightarrow 32. \tag{4e}$$

All the initial values (2 and 10) and the expected result (64) could be represented without error. However, some intermediate results did not have an exact posit(8, 0) representation, which led the final result to have an error of 50% with respect to the expected value. If the accumulation were to be performed using a quire, the result would exactly correspond to 64. Although this is a specific example to demonstrate how low-precision posits may fail, it clearly shows how better quires are for large accumulations.

Quires were used for the intermediate accumulations of the same training experiment of Section 4.1. Evaluated for 8-bit posits, it can be observed a perceptible increment on the achieved accuracy (see Table 4).

Table 4: Evaluation of the effect of accumulating with quires on DNN training using 8-bit posits.

Format	Accuracy	
	Without quire	With quire
Float (FP32)	**90.28%**	
Posit(8, 0)	10.00%	15.12%
Posit(8, 1)	12.54%	15.41%
Posit(8, 2)	12.55%	19.39%

4.3. Mixed Precision Configurations

A common problem that is particularly noted when using low-precision formats is the vanishing gradient problem – the gradients become smaller and smaller as the model converges. This is problematic because the gradients can decrease so much that the weight update (gradient multiplied by the learning rate) can become too small to be represented with a low-precision format. Moreover, the ratio between the weight value and the weight update is usually large so, even if they are representable, the employed format might not have enough resolution to represent the optimizer step result.

In [14, 24], low-precision floating-point and posit were used when training DNNs and, to prevent model accuracy loss, a higher precision primary copy of the weights was kept and used for the optimizer step. Moreover, it has been noticed that the last layer of the model is very sensible to quantization when using a low-precision floating-point [25]. When implemented with posits, the last layer required 16 bits [14].

Under these observations, the DNN of Section 4.1 was once more trained, but with a different posit precision for the optimizer and loss, and using posit(8, 2) everywhere else. The posit exponent size es was fixed at 2, since it gave good results and to be consistent with the most recent version of the Posit Standard.

The obtained results (see Table 5) showcase the feasibility of training DNNs using 8-bit posits, achieving an accuracy very close to 32-bits floats. In particular, while solely computing the optimizer with posit(12, 2) is not enough to achieve a float-like accuracy (see left side of Table 5), when the loss precision is also increased (right side), the model is able to train without any accuracy penalization and using, at most, 12-bit posits. Moreover, most of the computations performed

Table 5: Evaluation of mixed low-precision posit configurations for DNN training (using quires). Configuration Ox-Ly means Optimizer (O) with posit(x, 2) and Loss (L) with posit(y, 2), and everything else with posit(8, 2). Compared against 32-bit float.

Mixed Precision Configuration			
Optimizer (O)	Accuracy	Loss (L)	Accuracy
Float(FP32)	**90.28%**	**Float(FP32)**	**90.28%**
O16-L8$_q$	88.14%	O12-L16$_q$	90.03%
O12-L8$_q$	88.06%	O12-L12$_q$	90.07%
O10-L8$_q$	86.07%	O12-L10$_q$	90.13%
O9-L8$_q$	84.80%	O12-L9$_q$	89.35%
O8-L8$_q$	19.39%	O12-L8$_q$	88.0%

while training used 8-bit posits, which, for this particular model, corresponded to $\sim$95 % of the operations.

To validate these results, the proposed mixed-precision configuration shall be evaluated for DNN training with more complex datasets and models. From now on, the nomenclature used for the various configurations represents the posit precisions used in different stages: Optimizer (O), Loss (L), Forward propagation (F), Backward propagation (B), and Gradients calculation (G). The number next to these letters represent the number of bits n of the posit(n, 2) used, and the subscript q indicates that quires were used.

4.4. Training with less than 8-bits and Underflow

Being able to train DNNs with 8-bit posits, it is compelling to decrease the posit precision even further until the model is no longer able to train. In [12], the authors observed that the lack of underflow was undermining the model convergence for low-precision posits. The underflow topic was once again addressed in [14], where posits that could underflow were used for DL (for quantization), inspired by the fact that small values can be set to zero without hurting the model performance.

To evaluate if the lack of underflow was in fact harming the model convergence, the posit library Universal was modified so that posits did underflow instead of saturating to the minimum positive value (*minpos*). In [14], posits would underflow below a threshold defined as half the value of *minpos*. However, the present work implements the underflow as a natural extension of the rounding rule defined in the Posit Standard: "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". From this, the threshold for posit(4, 1) (encoded to infinite precision) would be:

$$\begin{aligned} 0 : 0000\,000\ldots &\rightarrow 0, \\ (\text{threshold} : 0000\,100\ldots) &\rightarrow 2^{2^1\times(-3)}, \\ \mathit{minpos} : 0001\,000\ldots &\rightarrow 2^{2^1\times(-2)}, \end{aligned} \tag{5}$$

which can be generalized for any posit *nbits* and *es* configuration as:

$$\text{threshold} = 2^{-2^{es}\times(nbits-1)} = \mathit{minpos}/2^{2^{es}}. \tag{6}$$

The DNN model was once more trained to evaluate how the underflow condition affected the achieved model accuracy. The experiments of Section 4.1 were repeated with underflow using a mixed precision configuration of mostly {5..7}-posits with 12-bit posits for the optimizer and loss (see Table 6).

Table 6: Evaluation of how underflow affects DNN training using a mixed low-precision posit configuration. Float for comparison.

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

The obtained results indicate that, while the proposed mixed-precision configuration already allows to train a DNN model with {6, 7}-bit posits, the ability to underflow significantly increases the accuracy achieved with 5-bit posits. These suggest that underflowing should benefit models trained with particularly small posit precisions.

5. Experimental Evaluation

PositNN allowed evaluating various methods of training DNN with posits, however, these should be validated with more datasets and models. Moreover, the proposed framework was also used to evaluate pre-trained models for DL inference using low-precision posits (post-training quantization), addressing, particularly, the effect of underflow. Every experiment performed with posits, using PositNN, was compared to the same performed with 32-bit floats, using PyTorch.

5.1. DNN Training Evaluation

Given the promising results for the Fashion MNIST dataset, the mixed low-precision posit configuration was also used to train LeNet-5 on MNIST and a variation of CifarNet (with $\sim$ 0.5 million parameters) on CIFAR-10 and CIFAR-100. The resulting accuracies are compared against 32-bit float in Table 7.

From the results previously shown in Table 5, training a DNN with 8-bit posits for everything except the optimizer (O) and loss (L), which used 12-bit posits, was enough to achieve an accuracy equivalent to 32-bit floats. However, this mixed-precision configuration was not sufficient with the more complex CIFAR-10 and CIFAR-100 datasets, which suffered a significant accuracy loss when using this configuration. To overcome this problem, the precision of the optimizer and the loss were increased to 16-bit posits and the models were then able to achieve the 32-bit float accuracy. This increment in precision allows the model weights to use a larger range of values and to be updated more

Table 7: Performance of posits for DNN training with mixed precision for various datasets. Compared against the same models trained with 32-bit floats.

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

accurately, which seems to be necessary for more complex datasets and deeper models.

It is important to stress that being able to replace 32-bit floats with 8-bit posits immediately corresponds to a 4× smaller memory footprint. Moreover, if the power required by a posit unit is comparable to its IEEE 754 compliant counterpart [6], then it will also use much less energy. Even considering the computations that require more precision, 12 and 16-bit posits seem to be enough, never requiring a 32-bit format.

Comparing to low-precision floats, Langroudi et al. [13] trained a FCNN on MNIST and Fashion MNIST using 32 and 16-bit floats. While 32-bit floats achieved an accuracy of 98.09% and 89.11%, 16-bit floats only achieved 90.65% and 81.73%, respectively. Thus, posits can achieve a much better performance than 16-bit floats while using half the memory.

5.2. DNN Inference Evaluation

The datasets considered were the same as in the training experiments and the models were pre-trained with floats and then quantized to the corresponding low-precision posit. These experiments consisted in performing the forward propagation using posits with {3-8}-bit precisions and accumulating with quires. Figure 10 shows the accuracies obtained using posits with $es = \{0, 1, 2\}$ compared against a model using floats and a random/untrained model. Additionally, they are compared to the same tests but with underflow.

The results obtained are similar to those presented in related works [13, 15], but these evaluate various low-precision posits and accumulations performed with quires. For every experiment performed with 8-bit posits, they were capable of performing as well as 32-bit floats, with model accuracy differences lower than 1%. Posit precisions with less than 8 bits were also able to obtain acceptable accuracies, but performed much worse for more complex datasets, particularly for $es = 0$.

When the underflow was introduced, the achieved accuracy increased significantly, especially for very low precision posits with $es = 0$. This was expected, since their not so insignificant *minpos* values were harming the DL computations [12]. In particular, for CIFAR-10 and CIFAR-100, when underflow was enabled, posit(5, 0) achieved better accuracy than posit(5, 2). For comparison, Langroudi et al. [13] tested a 5-bit floating-point format for CIFAR-10 and it did not work at all, since it got an accuracy of, approximately, 13%.

5.3. Comparison of Posit Standards

As explained in Section 2.2, the most recent version of the Posit Standard [17] introduced two changes: posits no longer have an arbitrary exponent size (fixed to $es = 2$), and the quire format was reformulated so that the limit of exact dot products is the same for any posit precision ($2^{31} - 1$ accumulations).

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

As for the new quire, the updated capacity does not impair the presented results, rather on the contrary, it should improve the accuracy obtained in DL. PositNN was used to test the new quire format, but it showed no relevant improvements, which should only be more noticeable for wider models using low-precision posits, since they will have larger accumulations.

5.4. Parallelization Speedup

To evaluate the performance of PositNN, in what concerns the scalability when executed in platforms with parallel processing capabilities, a small throughput analysis was also performed (see Figure 9). The results were obtained in an Intel i7-5930K CPU (6 cores with 2 threads per core), hence the sudden performance degradation after 6 threads. Nonetheless, considering the overhead of spawning the workers and that not all functions were parallelized, the quite proportional speedup observed was satisfactory.

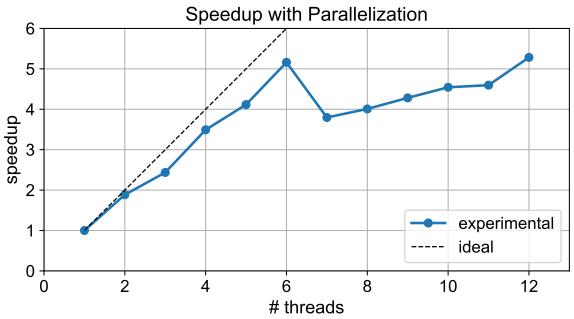
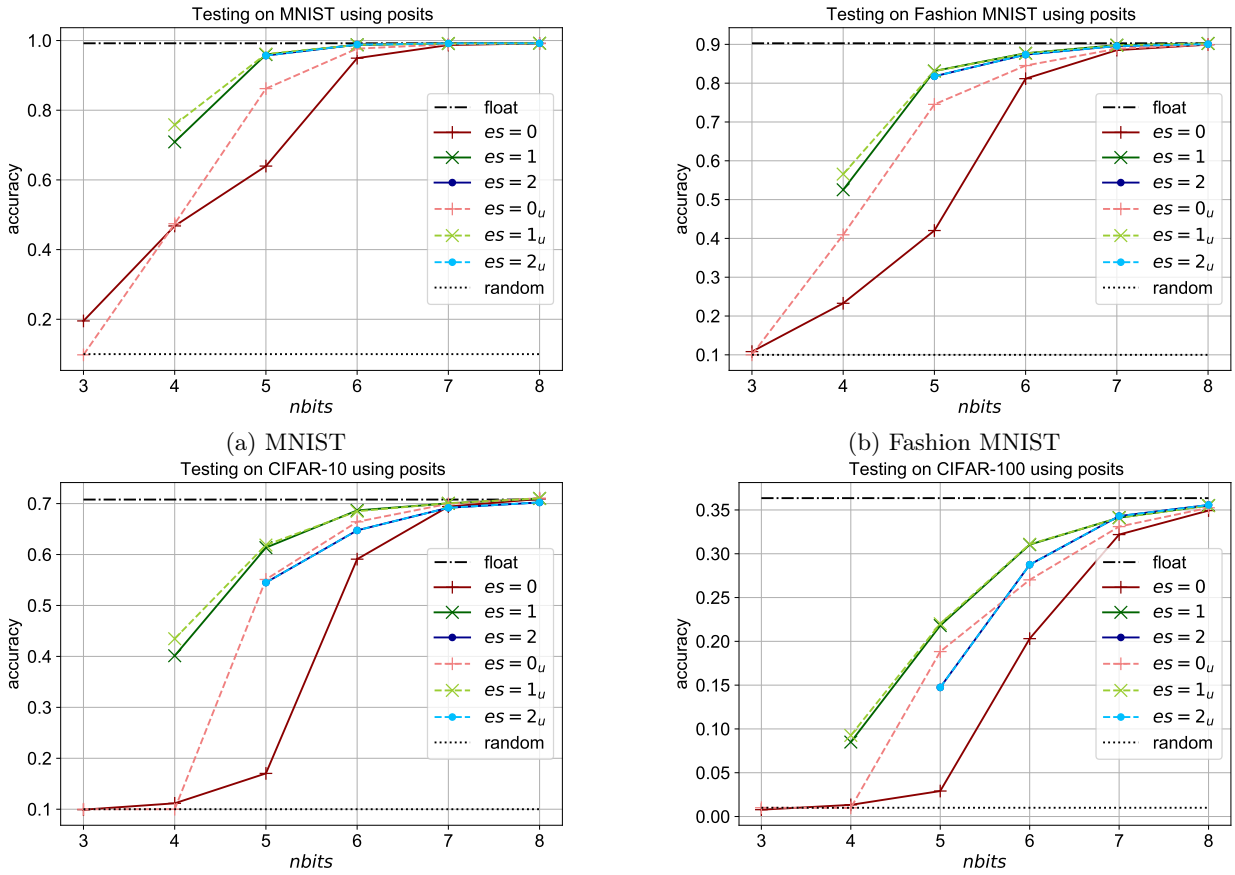


Figure 9: Plot of the speedup gained in function of the number of used threads. Tested in a 6-core CPU (2 threads per core) training a CNN with 8192 samples.



(a) MNIST

(b) Fashion MNIST

(c) CIFAR-10

(d) CIFAR-100

Figure 10: Plots of the accuracies obtained when testing pre-trained models with various datasets and posit configurations (using quires). Evaluating the effect of underflow (subscript u) on DL inference. Compared against the accuracy obtained with 32-bit floats and when the model is randomly initialized (untrained).

6. Conclusions

A new DNN framework, named PositNN, that supports both training and inference using any posit precision is proposed. The mixed precision feature allows adjusting the posit precision used in each stage of the training network, thus achieving results similar to 32-bit float. Common CNNs were trained with the majority of the operations performed using 8-bit posits and showed no significant loss of accuracy on datasets such as MNIST, Fashion MNIST, CIFAR-10, and CIFAR-100. DL inference was performed with even less precision and 6-bit posit exhibited an accuracy degradation of about 5 % when compared against 32-bit floats. Enabling underflow for the posit format showed potential to increase their performance for DL, especially with very low precision posits and with $es = 0$. Regarding other exponent sizes, $es = 1$ exhibited significantly better results than $es = 2$, therefore, the newer version of the Posit Standard should be further evaluated.

Future work shall make use of this knowledge and framework to devise adaptable hardware implementations of posit units that may exploit this feasibility to implement low-resource and low-power DNN implementations while keeping the same model accuracy. Moreover, adopting a progressive reduction of the posit precision during the training process might result in a network whose weights are better distributed and that performs better with very low-precision posits. Finally, posits should be validated for even deeper DL models and more complex datasets.

Acknowledgements

Special thanks to Fundação Calouste Gulbenkian (FCG) for the merit scholarship that supported the entirety of the student degree. This work was carried out under the projects UIDB/50021/2020 and PTDC/EEI-HAC/30485/2017 of Fundação para a Ciência e a Tecnologia (FCT).

References

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

[2] C. Li. OpenAI's GPT-3 Language Model: A Technical Overview, June 2020. URL `https://lambdalabs.com/blog/demystifying-gpt-3/`. Accessed on 2020-10-13.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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