POSITNN: TRAINING DEEP NEURAL NETWORKS WITH MIXED LOW-PRECISION POSIT

Gonçalo Raposo *Pedro Tomás* *Nuno Roma*

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal

ABSTRACT

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. Under this premise, the posit numerical format appears to be 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 research aims to evaluate the feasibility to train deep convolutional neural networks using posits. For such purpose, a software framework was developed to use simulated posits and quires in end-to-end training and inference. This implementation allows using any bit size, configuration, and even mixed precision, suitable for different precision requirements in various stages. The obtained results suggest that 8-bit posits can substitute 32-bit floats during training with no negative impact on the resulting loss and accuracy.

***Index Terms*—** 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.

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. 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.

However, most published studies regarding the application of the posit format to Deep Neural Networks (DNNs) rely on the inference stage [6–12]. 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 [13], by training a Fully Connected Neural Network (FCNN) for a binary classification problem using $\{8, 10, 12, 16, 32\}$-bit posits. Later, in [14, 15], a FCNN was trained for MNIST and Fashion MNIST using $\{16, 32\}$-bit posits. In [16, 17], 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 [18], a CNN was trained for CIFAR-10 but using only $\{16, 32\}$-bit posits.

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

- **open-source framework**[1] 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 multithread support;
- adaptation of the framework to support **mixed-precision**, with different stages (forward, backward, gradient, optimizer, and loss) operating under different posit formats;
- training CNNs with only **8 to 12-bit posits** without impacting on the achieved model accuracy.

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

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):

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

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 is characterized by a 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) [20]. It is decoded as in Eq. (2).

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

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, and perhaps the most interesting aspect for DNN applications, refers to the distribution of its values, resembling a log-normal distribution (see Fig. 1), which is similar to the normal distribution of the values commonly found in DNNs. Another interesting point is the definition of the quire, a Kulisch-like large accumulator [21] designed to contain exact sums of products of posits. Table 1 shows the recommended posit and quire configurations.

3. DEEP LEARNING POSIT FRAMEWORK

Current DNN frameworks (such as PyTorch and TensorFlow/Keras) 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. PositNN 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.

Table 1. Main properties of posit formats according to [20].

nbits	8	16	32	64
es	0	1	2	3
dynamic range	$2^{\pm 6}$	$2^{\pm 28}$	$2^{\pm 120}$	$2^{\pm 496}$
quire bits	32	128	512	2048
dot product limit	127	32767	$2^{31}-1$	$2^{63}-1$

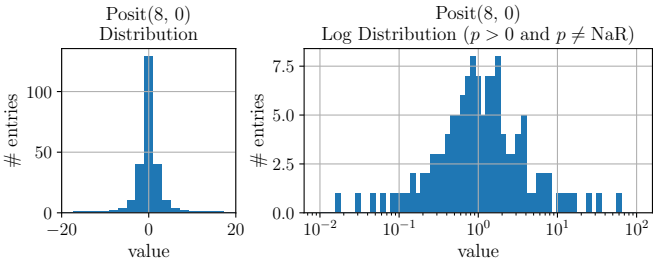

Fig. 1. Distribution of posit(8, 0) in linear and log scale.

As an example, a comparison between PyTorch and the proposed framework regarding the declaration of a 1-layer model is shown in Fig. 2 (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 shown in Fig. 2 (right), which allow implementing all the stages illustrated in Fig. 3. 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).

3.2. Posit variables

Among the several libraries already available to implement posit operators in software [22], the Universal[2] 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 posits. 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 [20] are fully supported and implemented. Furthermore, the proposed framework adopts bitwise operations whenever possible, thus avoiding operating with intermediate float representations, since this could introduce errors regarding a native implementation.

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

```cpp
#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);
```

```cpp
#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:**
  Cross Entropy,
  Mean Squared Error (MSE)
- **Optimizer:**
  Stochastic Gradient Descent (SGD)
- **Utilities:**
  StdTensor, convert PyTorch tensors, mixed precision tensor, save and load model, scaled gradients

Fig. 2. Comparison of PyTorch (left) and the proposed framework (center). Implemented functionalities of PositNN (right).

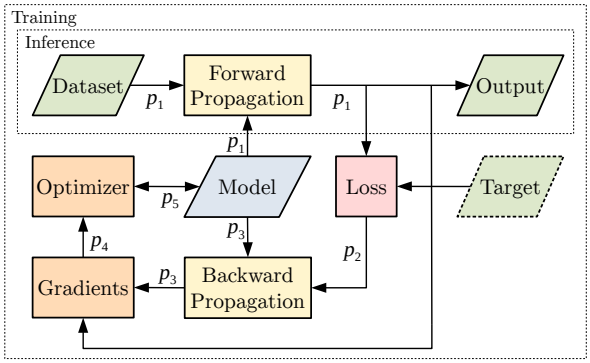

Fig. 3. DNN training diagram, starting at the dataset. The various p_i, with $i = \{1..5\}$, represent the different posit precisions that may be used throughout the proposed framework.

3.3. Implementation

Posit tensors are stored as StdTensors, a class 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.

Given that some stages are more sensitive to numerical errors, the proposed framework supports different precisions per stage, as depicted in the arrows of Fig. 3. Although not illustrated, it even allows the model to use different precisions per layer. To accomplish that, the weights are stored in a class where 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. It also has the option to use quires for the accumulations, significantly improving the accuracy of matrix multiplications, convolutions, etc.

In order to take the maximum advantage of the CPU, most functions were conveniently parallelized and implemented with multithread support, thus dividing each mini-batch by different workers. In matrix multiplication, this corresponds to splitting the left operand by rows, performing the computation, and then concatenating the results. The threads were implemented using std::thread.

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.

4. EXPERIMENTAL EVALUATION

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. Then, the best configuration was used to train a deeper model on a more complex dataset. In this evaluation, small accuracy differences ($< 1\%$) were assumed to be caused solely by the randomness of the training process and not exactly by lack of precision of the numerical format.

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. Just as in [15, 18], posit(16, 1) was first used everywhere and decreased until posit(8, 0) (see Table 2).

As expected, posit(16, 1) achieves a float-like accuracy, and narrower precisions, such as posit(12, 1) and posit(10, 1), are also usable for training, the latter incurring in some accuracy loss. However, when trained using posit(8, 0), the model accuracy does not improve and fixes at $10\,\%$ (equivalent to randomly classifying a 10-class dataset), probably due to the narrow dynamic range (as seen in Fig. 1). This hypothesis was subsequently evaluated by using a different exponent

Table 2. Accuracy of LeNet-5 trained on Fashion MNIST using posit and without quire, using float for reference.

Posit	**Float**	$(16, 1)$	$(12, 1)$	$(10, 1)$	$(8, 0)$
Accuracy [%]	**90.42**	90.87	90.15	88.15	10.00

Table 3. Accuracy of LeNet-5 trained on Fashion MNIST using posit and quire. Posit8 is tested with different *es*.

Posit with quire	**Float**	$(10, 1)$	$(8, 0)$	$(8, 1)$	$(8, 2)$
Accuracy [%]	**90.42**	88.40	13.84	12.86	19.39

Table 4. Accuracy of LeNet-5 trained on Fashion MNIST using posit, quire, and mixed precision. Configuration OxLy means Optimizer (O) with posit(x, 2) and Loss (L) with posit(y, 2), and everything else with posit(8, 2).

Configuration	**Float**	O12L8	O12L12	O12L10	O10L10
Accuracy [%]	**90.42**	88.40	90.07	90.25	88.08

size (*es*) and using quires for the accumulations (see Table 3). The obtained results confirmed the hypothesis, showing that the precision of the 8-bit model slightly increases when using quires, especially when the posit exponent size is $es = 2$.

Another common problem that is particularly noted while using 8-bit posit precisions is the vanishing gradient problem – the gradients become smaller and smaller as the model converges. This is particularly problematic when the model weights are updated with low-precision posits, since they do not have enough resolution for small numbers. As suggested in [17], using 16-bit posits for the optimizer and loss is usually enough to allow models to train with low-precision posits. With this observation in mind, this model was trained with a different precision for the optimizer and loss, while using posit(8, 2) everywhere else (see Table 4). The posit exponent size *es* was fixed at 2, since it gave the best results and simplified the conversion between posits with different *nbits*.

The obtained results showcase the feasibility of using 8-bit posits, achieving an accuracy very close to 32-bits IEEE 754. In particular, while solely computing the optimizer with posit(12, 2) is not enough to achieve a float-like accuracy, when the loss precision is also increased, the model is able to train without any accuracy penalization and using, at most, 12-bit posits. Conversely, if posit(10, 2) is used for both the optimizer and loss, the final accuracy slightly decreases. Therefore, the configuration with 12 bits for optimizer and 10 bits for loss (O12L10 in Table 4) offers the best compromise in terms of low-precision and overall model accuracy. This configuration will be referred to as posit(8, 2)*, since the loss function and weight update, both computed with higher precision, only represent about $15\,\%$ of the operations that are performed while training the considered models.

Given the promising results for the Fashion MNIST dataset, the posit(8, 2)* configuration was also used to train LeNet-5 on MNIST and CifarNet on CIFAR-10, validating the proposed mixed configuration. The resulting accuracies are compared against float in Table 5. Moreover, a plot of the training progress of LeNet-5 on Fashion MNIST is shown in Fig. 4, comparing different posit configurations and float.

Table 5. Accuracy of CNNs trained on MNIST, Fashion MNIST, and CIFAR-10 using float and posit(8, 2)*.

Dataset CNN	MNIST LeNet-5	Fashion MNIST LeNet-5	CIFAR-10 CifarNet
Float [%]	99.19	90.42	70.29
Posit(8, 2)* [%]	99.17	90.25	68.65

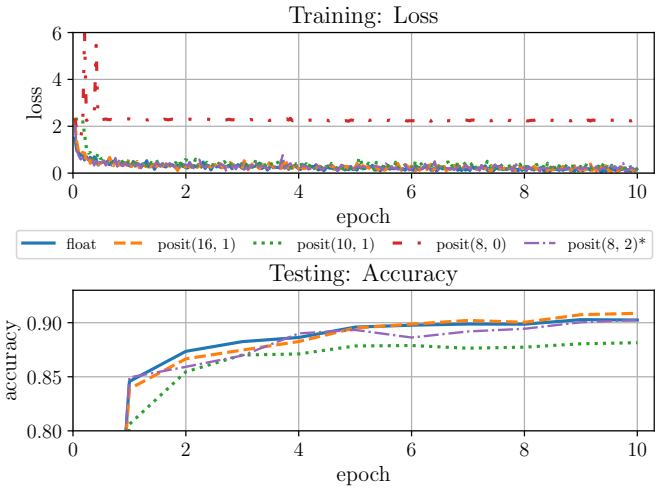

Fig. 4. Training loss and testing accuracy of LeNet-5 trained on Fashion MNIST using float and different posit precisions. Posit(8, 2)* corresponds to configuration O12L10 of Table 4.

5. CONCLUSION

A new DNN framework (PositNN) supporting 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 float. Common CNNs were trained with the majority of the operations performed using posit(8, 2) and showed no significant loss of accuracy on datasets such as MNIST, Fashion MNIST, and CIFAR-10.

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.

Acknowledgments

Work supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), under the projects UIDB/50021/2020 and PTDC/EEI-HAC/30485/2017, and student merit scholarship funded by Fundação Calouste Gulbenkian (FCG).

6. REFERENCES

[1] Neil C. Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F. Manso, "The Computational Limits of Deep Learning," July 2020, arXiv: 2007.05558.

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

[3] Jürgen Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, Jan. 2015, arXiv: 1404.7828.

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

[5] John L. Gustafson and Isaac Yonemoto, "Beating Floating Point at its Own Game: Posit Arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, pp. 71–86, June 2017, doi: 10.14529/jsfi170206.

[6] Marco Cococcioni, Emanuele Ruffaldi, and Sergio Saponara, "Exploiting Posit Arithmetic for Deep Neural Networks in Autonomous Driving Applications," in *2018 International Conference of Electrical and Electronic Technologies for Automotive*. July 2018, number November, IEEE, doi: 10.23919/eeta.2018.8493233.

[7] Jeff Johnson, "Rethinking floating point for deep learning," Nov. 2018, arXiv: 1811.01721.

[8] Seyed Hamed Fatemi Langroudi, Tej Pandit, and Dhireesha 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)*. Mar. 2018, pp. 19–23, IEEE, arXiv: 1805.08624, doi: 10.1109/emc2.2018.00012.

[9] Zachariah Carmichael, Hamed F. Langroudi, Char Khazanov, Jeffrey Lillie, John L. Gustafson, and Dhireesha Kudithipudi, "Deep Positron: A Deep Neural Network Using the Posit Number System," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Mar. 2019, pp. 1421–1426, IEEE, arXiv: 1812.01762, doi: 10.23919/date.2019.8715262.

[10] Zachariah Carmichael, Hamed F. Langroudi, Char Khazanov, Jeffrey Lillie, John L. Gustafson, and Dhireesha Kudithipudi, "Performance-Efficiency Trade-off of Low-Precision Numerical Formats in Deep Neural Networks," in *Proceedings of the Conference for Next Generation Arithmetic 2019*, New York, NY, USA, Mar. 2019, vol. Part F1477, pp. 1–9, ACM, arXiv: 1903.10584, doi: 10.1145/3316279.3316282.

[11] Hamed F. Langroudi, Zachariah Carmichael, John L. Gustafson, and Dhireesha Kudithipudi, "PositNN Framework: Tapered Precision Deep Learning Inference for the Edge," *Proceedings - 2019 IEEE Space Computing Conference, SCC 2019*, pp. 53–59, July 2019, doi: 10.1109/spacecomp.2019.00011.

[12] Hamed F. Langroudi, Vedant Karia, John L. Gustafson, and Dhireesha 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)*. June 2020, pp. 726–727, IEEE, doi: 10.1109/cvprw50498.2020.00371.

[13] Raúl Murillo Montero, Alberto A. Del Barrio, and Guillermo Botella, "Template-Based Posit Multiplication for Training and Inferring in Neural Networks," July 2019, arXiv: 1907.04091.

[14] Hamed F. Langroudi, Zachariah Carmichael, and Dhireesha Kudithipudi, "Deep Learning Training on the Edge with Low-Precision Posits," July 2019, arXiv: 1907.13216v1.

[15] Hamed F. Langroudi, Zachariah Carmichael, David Pastuch, and Dhireesha Kudithipudi, "Cheetah: Mixed Low-Precision Hardware & Software Co-Design Framework for DNNs on the Edge," pp. 1–13, Aug. 2019, arXiv: 1908.02386.

[16] Jinming Lu, Siyuan Lu, Zhisheng Wang, Chao Fang, Jun Lin, Zhongfeng Wang, and Li Du, "Training Deep Neural Networks Using Posit Number System," Sept. 2019, arXiv: 1909.03831.

[17] Jinming Lu, Chao Fang, Mingyang Xu, Jun Lin, and Zhongfeng Wang, "Evaluations on Deep Neural Networks Training Using Posit Number System," *IEEE Transactions on Computers*, vol. 14, no. 8, pp. 1–1, 2020, doi: 10.1109/tc.2020.2985971.

[18] Raul Murillo, Alberto A. Del Barrio, and Guillermo Botella, "Deep PeNSieve: A deep learning framework based on the posit number system," *Digital Signal Processing*, vol. 102, pp. 102762, jul 2020, doi: 10.1016/j.dsp.2020.102762.

[19] Leonel Sousa, "Nonconventional computer arithmetic circuits, systems and applications," *IEEE Circuits and Systems Magazine*, vol. 20, no. 4, pp. 1–26, Oct. 2020.

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

[21] Ulrich Kulisch, *Computer Arithmetic and Validity*, De Gruyter, Berlin, Boston, Jan. 2012, doi: 10.1515/9783110301793.

[22] NGA Team, "Survey of Posit Hardware and Software Development Efforts," Unum & Posit - Next Generation Arithmetic, July 2019, https://posithub.org/docs/PDS/PositEffortsSurvey.html, Accessed on 2020-10-16.