Qtorch+: Next Generation Arithmetic for Pytorch Machine Learning

Nhut-Minh Ho[1], Himeshi De Silva[2], John L. Gustafson[1], and Weng-Fai Wong[1]

[1] National University of Singapore, Singapore
{minhhn,john.gustafson,wongwf}@comp.nus.edu.sg
[2] Agency for Science, Technology and Research (A*STAR), Singapore
{himeshi_de_silva}@i2r.a-star.edu.sg

Abstract. This paper presents Qtorch+, a tool which enables next generation number formats on Pytorch, a widely popular high-level Deep Learning framework. With hand-crafted GPU accelerated kernels for processing novel number formats, Qtorch+ allows developers and researchers to freely experiment with their choice of cutting-edge number formats for Deep Neural Network (DNN) training and inference. Qtorch+ works seamlessly with Pytorch, one of the most versatile DNN frameworks, with little added effort. At the current stage of development, we not only support the novel posit number format, but also any other arbitrary set of points in the real number domain. Training and inference results show that a vanilla 8-bit format would suffice for training, while a format with 6 bits or less would suffice to run accurate inference for various networks ranging from image classification to natural language processing and generative adversarial networks. Furthermore, the support for arbitrary number sets can contribute towards designing more efficient number formats for inference in the near future. Qtorch+ and tutorials are available on GitHub (https://github.com/minhhn2910/QPyTorch).

Keywords: Deep learning · Posit format · Novel number formats · Pytorch framework

1 Introduction

Reducing the bitwidth of number representations employed in Neural Networks to improve their efficiency is a powerful technique that can be used to make Deep Learning more accessible to a wider community. This is especially important when the variety of applications that use Deep Learning and the size and complexity of models have all increased drastically. For example, even with the latest GPU hardware capabilities, the GPT-3 model with 175 billion parameters requires 288 years to train [4]. The reason for the extraordinary training time and computational resources required is primarily due to the fact that the gargantuan amount of parameters cannot fit into the main memory of even the largest GPU [30]. Therefore, lowering the precision to reduce the memory consumption is extremely helpful to improve execution times and enable models to be run on a wider range of general-purpose hardware.

Research into low-precision number representations and their related arithmetic operations for Deep Learning has made many inroads in recent years. Several new low-precision floating-point formats have been proposed, many of them specifically targeted towards this domain. Posit™ arithmetic [13] with its ability to provide tailor-made accuracy to values that are of significance in the application, has seen increasing interest. Due to the arithmetic properties of posits, they naturally lend themselves to low-precision neural network training and inference. In the case of low-precision inference, custom sets of values can also be designed for quantization to achieve high levels of model compression. Therefore, these formats merit comprehensive investigations for the use in DNN training and inference.

Due to the fast-pace and significant interest, a pressing issue the Deep Learning research community has had to grapple with in the recent past is the difficulty for independent groups to reproduce model results that are being published. Though publicly available industry benchmarks [28] have been created to address the problem, even those results cannot practically be reproduced by research groups without access to significant expertise and resources. The fine-tuning and hand-tweaked kernels are almost always proprietary and not publicly available. An open-source Deep Learning framework which enables experimenting with the aforementioned arithmetic formats, will allow researchers to quickly prototype and test newer number representations for Deep Learning.

In this paper we present Qtorch+, a framework for experimenting with posits and arbitrary number sets with flexible rounding for Deep Learning. Qtorch+ is developed upon QPyTorch, a low-precision arithmetic simulation package in PyTorch that supports fixed-point and block floating-point formats [48]. Because our framework operates seamlessly with PyTorch, users are granted all the flexibility that come with it for low-precision experimentation. This includes support for a rich set of models, benchmarks, hardware configurations and extendable APIs. Leveraging the many capabilities of Qtorch+, we evaluate an extensive set of benchmarks for both training and inference with low-precision posits and arbitrary number sets.

The remainder of the paper is organized as follows. Section 2 presents some background into Neural Networks, floating-point and fixed-point formats, posits and arbitrary number sets. It also gives an introduction into integer quantization and discusses work in the area relevant to these topics. In Section 3 we present the design and implementation details of the Qtorch+ framework. Section 4 gives and overview of the practical usage of the framework for training and inference. Section 5 details the results the framework achieved on inference tasks. Some case studies related to training with posits and performing inference with a customized number set are presented in Section 6. Section 7 concludes.

2 Background and Related Work

2.1 Neural Networks

Neural networks have achieved astonishing performance on different complex tasks in recent years. Starting with the introduction of Convolutional Neural Networks (CNN) for image classification, they have branched out to many other diverse tasks today [25]. The initial CNNs were typically trained using the back-propagation method [24] which required intensive computational power. Hardware that could handle such computational demands and the representative datasets required for training more complex tasks remained an obstacle for a long period of time. More recently, with the introduction of GPUs for accelerated training and inference at multiple magnitudes faster than traditional processors, more and more deeper neural network architectures have been designed to tackle more complex datasets and challenges (e.g. Imagenet [10]). Most notably, the introduction of very deep networks such as Resnet [14] have revolutionized the approach to computer vision with Deep Learning increasingly adopted for more difficult tasks. To this day, neural networks have been used for many tasks including vision [14,45,49], language [43,4], audio [37,31], security [36,42], healthcare [39,12], general approximation [47,19], etc.

2.2 Floating-Point and Fixed-Point formats

Floating-point and fixed-point formats have been widely used for general computation since the early days of the computing era. They have different characteristics which make them suitable for different application domains and for different approximations. This led to various works on tuning those formats [1,6,11,18,8,15]. Recently, with the popularity of deep neural networks, hardware vendors and researchers have found that lower bitwidth on these formats can still achieve high accuracy both on inference and training while improving system energy efficiency and performance [29,40]. Thus, there are several works that target the reduced precision of floating point and fixed point format for neural network inference and training [41,5,40,44] [38,3,26,17].

Both arbitrary bitwidth floating-point and fixed-point formats have been supported by the original QPytorch framework. In this paper, we focus on extending the framework to support novel number formats such as posits and, more generally, arbitrary sets of numbers.

2.3 Integer Quantization

Integer quantization in neural networks refers to the mapping FP32 values to 8-bit integer (INT8) values. This process requires selecting the quantization range and defining the mapping function between FP32 values to the closest INT8 value and back (quantize and dequantize). If the selected range is $[\alpha, \beta]$, then uniform quantization takes an FP32 value, $x \in [\alpha, \beta]$ and maps it to an 8-bit value. The most popular mapping function used is $f(x) = s \cdot x$ (scale quantization) where

$s, x, z \in R$; s is the scale factor by which x will be multiplied. The uniform scale quantization is the most popular in hardware [46]. Let s_1 and s_2 be the scales used to quantize weight W and activation A of a dot product operation ($\otimes$). The scale quantized dot product result R' can be dequantized by multiplying with the appropriate factor:

$$R' = W' \otimes A' = \sum_{1}^{K} w_i \times s_1 \times a_i \times s_2 = R \times s_1 \times s_2$$

Integer quantization is already supported by mainstream frameworks and hardware vendors [46,21]. Thus, it is not the primary focus of this paper.

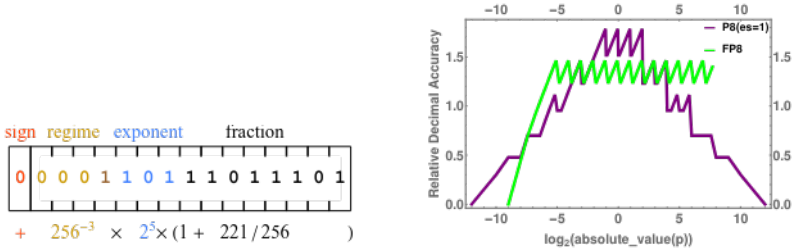
2.4 Posit Format

The posit number format has a distinctive property compared to other formats which results in better numerical stability in many application domains. The distribution of representable values in posits is more concentrated to a central point in the log2 domain (around 2^0) as seen in Figure 1b. This property will benefit certain applications where most of the values are concentrated to a specific range. In contrast, this will overwhelm the number of representable values of both floating-point and fixed-point formats. As seen in the Figure, the floating-point accuracy distribution is uniform when compared to the tapered accuracy of posits. Consequently, many studies [16,27,22,23,7] have shown that DNNs and some specific domain applications [20] are among the beneficiaries of this property of posits.

The above described property is due to the unique representation of posits. Figure 1a shows an example of a posit. A posit environment is defined by the length of the posit, *nsize*, and the size of the exponent field, *es*, which in this case is 16 bits and 3 bits. The first bit is reserved for the sign of the number. What follows after the sign is the regime field which is of variable length. To decode the regime, one can simply count the number of `0`s (or `1`s) after the sign until a `1` (or `0`) is reached. If the first regime bit is `0` the regime is negative and vice-versa. In this case, the regime is therefore -3. The regime value is used as the power to be raised for a value known as *useed* which is computed by using the exponent length: $(2^{2^{es}})$. There is always an implicit hidden bit in the fraction (except for zero). All these fields are read as shown in the Figure to obtain the value the posit is representing. Complete details of the posit format and its related arithmetic can be found in the posit standard [32].

2.5 Arbitrary Number Sets

Apart from the aforementioned formats, we found that allowing arbitrary number sets for inference can help accelerate the research and development of customized hardware for machine learning applications [2,34]. Thus, we also extend



(a) Example Posit (b) Accuracy

Fig. 1: Posit Format

the framework to support any number format which can be customized depending on the application. For this feature, the user will use their own method to craft a highly specialized table set for rounding. The arbitrary number set feature can also directly simulate any number format and other table lookup techniques. In general, any number format can be simulated with this method given the set of all representable values in the format. However, due to the table size, we recommend using this for very low bitwidth representations. As case studies of this feature, we will give some examples of using this to achieve very small sets while maintaining high output quality for selected applications. This feature will support two main research directions :

- Hardware friendly number formats with strict rules on the distribution of representable values. This category consist of number formats that are known to be efficient in multiplication (logarithmic domain, additive of logarithmic numbers).
- Arbitrary number sets which have no rules on the distribution of representable values. To implement this category in hardware, we need a customized table lookup or integer-to-integer mapping combinational logic circuit.

3 Design and implementation of Qtorch+

Because most Deep Learning frameworks and accelerators support extremely fast FP32 inference, we can take advantage of highly optimized FP32 implementations as the intermediate form to simulate our number formats with sufficient rounding. For this to work correctly, we assume that FP32 is the superset of our target format to be simulated. This remains true when the number simulated is low bitwith (e.g. 8-bit and below). For simulating higher bitwith (above 16 bits) arbitrary number formats, we can opt to use FP64 as the intermediate number format to store the rounded values. In the context of this paper, we focus on very low bitwith number formats and using FP32 as the intermediate format. The workflow of a DNN operation simulated in a low bitwidth number

format with correct rounding can be viewed in Figure 2. This method has been widely used to simulate low precision fixed-point and floating-point formats and integer quantization in state-of-the-art techniques [48]. By introducing posits to the framework, the quantizers in Figure 2 will have configuration parameters : *nsize*, *es* for posit format and *scaling* which is used to implementing exponent bias as in Section 3.2. All of the quantizers and their usage will be demonstrated in Section 4.2.

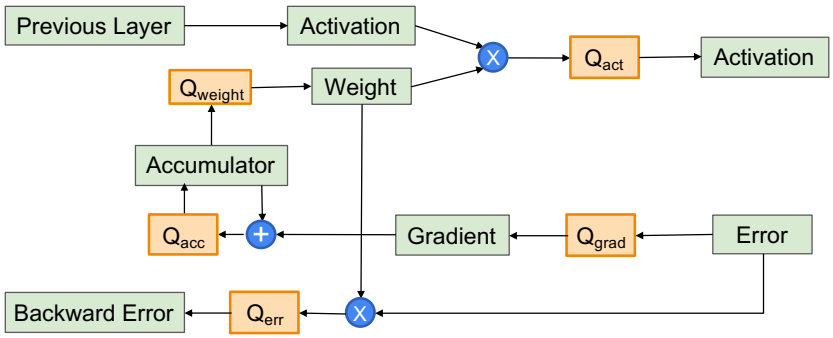


Fig. 2: Qtorch+'s training model and APIs are inherited from the original QPytorch framework with the separated kernel approach to quantize the values to new formats while using FP32 matrix multiplication for fast simulation time. We extend these functionalities to support posits, exponent biases and arbitrary number sets

.

3.1 Floating-point and posit conversion

To simulate posits efficiently, we implement the conversion between the number format and FP32 in Qtorch+ using integer and bit operations as well as built-in hardware instructions. The implementation of the functions are based on efficient encoding and decoding of a 16 bit posit into FP32 [9].

To convert a posit into FP32, the sign bit is first extracted and the two's complement value of the number is obtained if the sign is negative. Thereafter, the regime is decoded as described in Section 2.4. Once these two operations are completed, we can remove these two fields with a right shift operation and directly superimpose the remaining exponent and fraction fields to the corresponding fields of an FP32 value. To get the final exponent, the decoded regime value and the exponent point bias has to be added to the exponent field.

To convert an FP32 value to a posit, first the FP32 value needs to be checked against the maximum and minimum bounds of the posit's representable range. If it can be represented as a posit, then as in the case before the sign can be extracted. The regime and exponent field of the posit can be decoded directly

from the exponent field of the FP32 number. Some post-processing is done to format the regime field afterwards. Once all the fields are known, the posit can be assembled and formatted. There are many tweaks to these algorithms described that are performed to make these two operations very efficient.

3.2 Scaling and using the exponent bias in posit

After studying the the value distribution histograms of many neural networks, we found that both the weights and the activations can be scaled for a more accurate posit representation. For example, in some GANs, the weights are concentrated in the range $[2^{-4}$ to $2^{-5}]$. Therefore, we can shift the peak of the histogram to the range with highest posit accuracy, near 2^0. Note that scaling cannot provide additional accuracy for floating-point formats because their accuracy distribution is flat (see Figure 1b).

Before and after a computation using a posit value, the encoder and decoder are used to achieve scaling. The decoder will decode the binary data in posit format to $\{S, R, E, F\}$ which represent {sign, regime, exponent, fraction}, ready for computation. The definitions of biased encoder and decoder for posit data P and a bias t are as follow:

$$\begin{aligned} &\text{Biased Decoder} : \{P, t\} \rightarrow \{S, R, E - t, F\} \\ &\text{Biased Encoder} : \{S, R, E + t, F\} \rightarrow \{P\} \end{aligned} \tag{1}$$

We scale using the posit encoder and decoder instead of floating-point multiplications for efficiency. If we choose an integer power of 2 for the scale, input scaling and output descaling can be done by simply biasing and un-biasing the exponent value in the encoder and decoder, as shown in Eq. 1. This exponent bias can be easily implemented in hardware by additional integer adder circuit with minimal hardware cost [16].

3.3 Arbitrary number sets

This feature is fully supported by the extended quantizer. To use this, the user will create a full set of all possible representable values of their format and pass it as an input to the quantizer. All the real values will then be rounded to their nearest value in the given set. This feature will be described in detail and demonstrated in Section 6.4.

4 Practical usage of Qtorch+

This section describes the APIs of Qtorch+ and how to use novel number formats in Deep Learning applications.

4.1 Leverage Forward_hook and Forward_pre_hook in Pytorch

To use Qtorch+ in inference seamlessly without any additional effort from the user, we leverage the "hook function" feature in Pytorch [33]. The weights can be quantized with our posit_quant function without the need for modifying the original model. However, for activation, changing the model code to intercept the dataflow of layers is required to apply custom number format simulation. With the recent Pytorch version and the introduction of "hook" functions, there is no need to modify the original model to achieve the same result. The forward_hook function is to process the output of the current layer before going to the next layer. The forward_pre_hook function is used to process the input of the current layer before doing the layer operations. Thus, forward_pre_hook is a universal way to intercept the input of any layers while forward_hook is the convenient way to intercept the output of any layer. For general usage, we can use forward_pre_hook and preprocess activations of the current layer with low bitwidth number formats. Likewise, we use forward_hook for extra simulation of the precision of the accumulator when we do not assume the exact dot product.

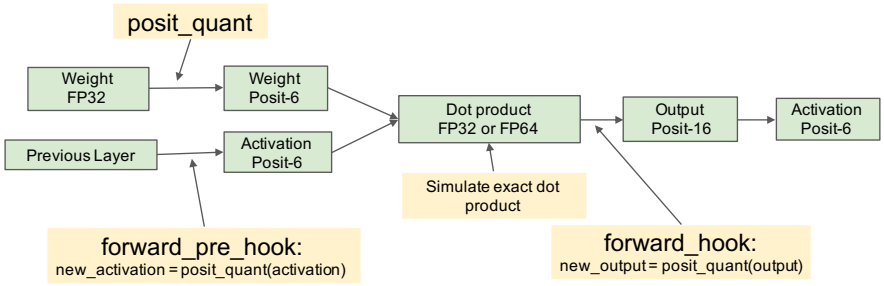


Fig. 3: Using Pytorch's feature to intercept the dataflow and simulate inference.

4.2 Qtorch+ in Training

Listing 1 shows the modification required to prepare the model for training. As we can see, the steps taken are not much different from the standard pytorch models preparation and construction. There are two main steps that we need to perform in order to use posit training:

- Declare all the quantizer used for each component of the optimizer and initialize the new optimizer with these parameters.
- Modify the model source code (MyModelConstructor) to use the argument *act_error_quant* in the forward pass of the model. The quant function must intercept the dataflow between each Convolutional/Linear layer for correct simulation. User can decide their own policy of skipping some layers to use higher precision (posit16, FP16 or FP32) if necessary.

```
from qtorch.quant import Quantizer, quantizer
from qtorch.optim import OptimLP
from qtorch import Posit
# define two different formats for ease of use
bit_8 = posit(nsize=8, es=2)
bit_16 = posit(nsize=16, es=2)

# define quantization function for each component of the neural network
weight_quant = quantizer(bit_8)
grad_quant = quantizer(bit_8)
momentum_quant = quantizer(bit_16)
acc_quant = quantizer(bit_16)

# define a lambda function so that the Quantizer module can be duplicated easily
act_error_quant = lambda : Quantizer(forward_number=bit_8, backward_number=bit_8)

#Step not included here: modify model forward pass to add quant() between layers.
model = MyModelConstrutor(act_error_quant)

#define normal optimizer as usual
optimizer = SGD(model.parameters(), lr=0.05, momentum=0.9, weight_decay=5e-4)
#user the enhanced optimizer with different number formats.
optimizer = OptimLP(optimizer,
                    weight_quant=weight_quant,
                    grad_quant=grad_quant,
                    momentum_quant=momentum_quant,
                    acc_quant=acc_quant,
                    grad_scaling=2**10 ) # do loss scaling if necessary
```

Listing 1: Example of the modification needed to add to prepare the model for training with Qtorch+.

4.3 Qtorch+ in Inference

Listing 2 shows how to utilize posits (or other number formats) in inference. The code in details involve two main steps:

- Decide the number formats for processing convolutional/linear layer. It is implemented as two functions: linear_weight and linear_activation (e.g. posit(6,1) in Listing 2. Decide the number formats for processing other layers (and the layers in excluded list). This number format for other layers needs to be in high precision to prevent accuracy loss. It also needs to be compatible with the low-bitwidth format for efficient hardware design (an accelerator that supports both FP32 and posit6 is likely more expensive than the one that only support posit6 and posit16)
- Given a pretrained model, instead of looking into the model definition, we can prepare and call the prepare_model() function with the logic in Listing 2.

– In general, the simulation of the number format for output with forward_hook as in Figure 3 can be skipped when we assume the dot product is done using the quire and the output format has enough precision to hold the output value (high precision as posit 16-bit or 32-bit).

```
1  from qtorch.quant import posit_quantize
2  def other_weight(input):
3      return posit_quantize(input, nsize=16, es=1)
4  def other_activation(input):
5      return posit_quantize(input, nsize=16, es=1)
6  def linear_weight(input):
7      return posit_quantize(input, nsize=6, es=1, scale=scale_weight)
8  def linear_activation(input):
9      return posit_quantize(input, nsize=6, es=1, scale=scale_act)
10
11 def forward_pre_hook_other(m, input):
12     return (other_activation(input[0]),)
13 def forward_pre_hook_linear(m, input):
14     return (linear_activation(input[0]),)
15
16 layer_count = 0
17 excluded_list = [] # list of all layers to be excluded from using low precision
18 model = torchvision.models.efficientnet_b7(pretrained=True) #load pretrained model
19 for name, module in model.named_modules():
20     if isinstance(module, nn.Conv2d) or isinstance(module, nn.Linear) \
21                                     and layer_count not in excluded_list:
22         module.weight.data = linear_weight(module.weight.data)
23         module.register_forward_pre_hook(forward_pre_hook_linear)
24         layer_count +=1
25     else: #should use fixed-point or posit 16 bits for other layers' weight
26         if hasattr(module, 'weight'):
27             layer_count +=1
28             module.weight.data = other_weight(module.weight.data)
29             module.register_forward_pre_hook(forward_pre_hook_other)
```

Listing 2: Example of the preprocessing code needed to add to prepare the model for inference with Qtorch+. Note that this code is generic to all models which can be loaded at line 18. We do not need to modify the source code of the model definition as other frameworks. For user convenience, we can hide this whole procedure into a single function *prepare_model* which does exactly the same task.

5 Inference Results of Posit

Table 1 shows the inference results of low bitwidth posit formats on different tasks. Because our framework is fully compatible with Pytorch, we can choose a diverse set of models for difficult tasks, especially the recent state-of-the-art models [45,49,4]. Any model that has a script which can be run using Pytorch can leverage our framework. Our models include the state-of-the-art image classification model EfficientNet B7 which reaches 84.3% top 1 accuracy on Imagenet. We also include the released GPT-2 model of OpenAI which achieved state-of-the-art performance in Language Modeling when it was introduced. For performance metric, we follow the guideline of other benchmark suites which set the threshold 99% of FP32 quality when using lower precision. The vanilla posit8 (without scaling) can achieve beyond 99% accuracy of FP32 in half of the models. The accuracy of image classification models when using posit8 conforms with the 99% standard (except GoogleNet which achieves 98.9% FP32 Accuracy). The pretrained models are retrieved from the official Pytorch [3], hugging face framework [4] and the respective authors. The inference models and scripts to run with posits are accessible online [5]. For image classification task, the test dataset is Imagenet. For Object detection, the test set is COCO 2017. For style transfer and super resolution models, we use custom datasets provided by the authors [49,45]. Question answering and language modelling task uses the SQuaD v1.1 and WikiText-103 dataset respectively.

When hardware modification is not allowed, the rest of the model can achieve 99% FP32 standard by dropping the first and the last layer of the models and apply higher precision to them (posit(16,1)). With little modification to the hardware to include an exponent bias, we can increase the accuracy of the model vastly as can be observed in column **P6+DS** in Table 1. The effect of scaling can increase the accuracy up to 7.8% in ResNEXT101. In GANs (Style Transfer and Super resolution tasks), the effect of skipping the first and the last few layers are more important than scaling posit format. Thus, we can see the P6+D can surpass posit8 in most cases. We will provide the results of posit8 when applying scaling and skipping to reach 99% FP32 standard.

6 Case Studies

6.1 Training with posit8

Previous works have shown that posit8 is enough for training neural network to reach near FP32 accuracy both in conventional image classification application [27] and GANs [16]. In the context of this paper, we do not enhance previous results. Instead, we try to show the completeness of the framework which supports several training tasks. Because many pretrained neural networks

[3] https://pytorch.org/vision/stable/models.html
[4] https://huggingface.co
[5] https://github.com/minhhn2910/conga2022

Task	Model	**FP32**	P(8,1)	P(8,2)	P(6,1)	P(6,2)	P6+D	P6+DS
Image Classification	Resnet50	76.1	**75.7**	75.3	66.3	54.9	69.1	74.4
Image Classification	ResNEXT101	79.3	**78.8**	78.4	66.3	65.8	69.8	77.6
Image Classification	GoogleNet	69.8	69.0	68.8	55.8	34.9	59.4	65.5
Image Classification	EfficientNetB7	84.3	**84.0**	**83.7**	79.8	75.3	80.2	82.7
Object Detection	FasterRCNN	36.9	36.4	36.2	25.5	24.0	28.2	35.5
Object Detection	MaskRCNN	37.9	**37.5**	37.2	36.9	25.3	28.8	36.5
Object Detection	SSD	25.1	21.3	24.1	1.6	10.8	15.3	22.6
Style Transfer	Horse-to-Zebra	100	96.4	93.7	84.8	79.6	98.0	**98.4**
Style Transfer	VanGogh Style	100	95.0	90.6	80.7	72.3	96.2	**96.7**
Super Resolution	ESRGAN	100	95.1	89.7	72.9	61.1	**99.2**	**99.6**
Question Answering	BERT-large	93.2	**93.2**	**93.2**	**92.8**	**92.9**	**92.9**	**92.9**
Language Modeling	GPT2-large ↓	19.1	**19.1**	**19.2**	21.4	22.0	20.8	19.5

Table 1: Inference results of 12 models on different tasks. For Image classification applications, the values are accuracy %. For Object Detection application, the values are box average precision (boxAP %). For GAN (style transfer and super resolution), the values are structural similarity index measure (SSIM %). For Question Answering, the values are F1 scores. For Language Modelling task, the value are Perplexity(lower better). P(6,1) means posit format with 6-bit nsize and 1-bit es. P6+D means applying the best posit 6-bit configuration while dropping (excluding) ≈ 2 layers in the original models for use in higher precision. P6+DS mean applying both dropping layers and weight/activation scale (exponent bias). The cells in **bold** font are where the configurations reach 99% FP32 quality as specified by MLPerf benchmark [35].

have been fine tuned for weeks or even months, we also do not replicate the training results of these tasks. Instead, we will show a diverse training tasks on neural networks which converge in less than a day due to time constraints. For other high time-consuming tasks when training with posit, please refer to the related work which used our extension to train Generative Adversarial Networks which typically takes days to weeks to complete one experiment [16]. The results can be seen in Table 3. This training results can be reproduced with our sample code and gradient scaling configuration for **P8+** available at [6]. From the table we can see that, in contrast with inference, **P(8,2)** has dominant performance in training compared to **P(8,1)**. With correct scaling, the **P8+** can reach FP32 accuracy. This agrees with previous works [29] that gradient scaling (also known as loss scaling) in low precision is advantageous and should be applied as a standard procedure in other training framework. For network like VGG11, we saw that correct gradient scaling can recover the training accuracy from 22.9% to 88.6%. In this experiment, we manually set the gradient scales based on experi-

[6] Same link as footnote 5

Models	GoogleNet	FasterRCNN	SSD	Horse-to-Zebra	VanGoghStyle	ESRGAN
P8+	69.5	36.6	24.8	99.8	99.7	99.9

Table 2: Enhancing the benchmarks in Table 1 to reach the 99% FP32 standard with posit 8-bit and with layer skipping and scaling. We pick the best accuracy among P(8,1) and P(8,2) for each model to present the result

menting all the power-of-2 scales possible (from 2^{-10} to 2^{10}) and choose the best scale which results in the best training output. The scale is static and be used for the entire training without changes. We set the number of training epochs to be 10 epochs for Lenet and Transformer [7] and 20 epochs for other networks.

Model	Task (metric)	Dataset	FP32	P(8,1)	P(8,2)	P8+
Lenet	Classification (Top1 %)	MNIST	98.7	9.8	98.6	99.0
Resnet	Classification (Top1 %)	Cifar10	91.0	11.1	89.7	91.6
VGG	Classification (Top1 %)	Cifar10	87.5	10.0	22.9	88.6
Resnet	Classification (Top1 %)	Cifar100	72.9	61.8	72.4	72.7
Transformer	Translation (BLEU %)	30k	35.4	32.9	34.5	35.0

Table 3: Training and inference with Qtorch+ and Posit. P8+ means the posit 8-bit configuration is used with gradient scaling that achieves highest output quality.

6.2 Tips for training with 8-bit posit

After trial and error, we have summarized a few tips on how to successfully train neural networks with posit8, especially with model that is difficult to train in low precision and fail to converge:

- Reduce batch size and use the built-in gradient/loss scaling. The effect of batch size and gradient scaling will be studied in this section.
- If gradient scaling still does not help convergence, the bitwidths need to be increased. Heuristically, we found that increasing the backward error precision is enough for convergence. Because the forward pass of most models is working well with posit8, they generally do not need higher bitwidth in training.
- Adjusting gradient scaling in training is generally more important than adjusting the weight scaling and exponent bias of posits.

[7] For Transformer, we had to use P(16,2) for the backward error propagating instead of P(8,2) to achieve convergence

It is a rule of thumb that using mini-batch will improve training accuracy. However, small batches mean low utilization of GPUs and longer training time. Each model has their own default batch size which is used in our experiment in Table 3. The experiment with different batch sizes are presented in Figure 4. From the figure, we can see that the batch size parameter affects the accuracy of both FP32 and posit training. However, large batch size has stronger adverse effect on the vanilla posit format. We also conclude that, where the vanilla posit format cannot help convergence, gradient scaling must be used. In rare cases, when the gradient scaling on low bitwidth posit format still cannot help convergence, increasing the bitwidth should be considered. In the experiment in Figure 4, we try to further use weight scaling and gradient bias similar to inference but the effect is not significant and weight-scaling/exponent bias alone cannot help posit8 training reach FP32 accuracy as gradient scaling does.

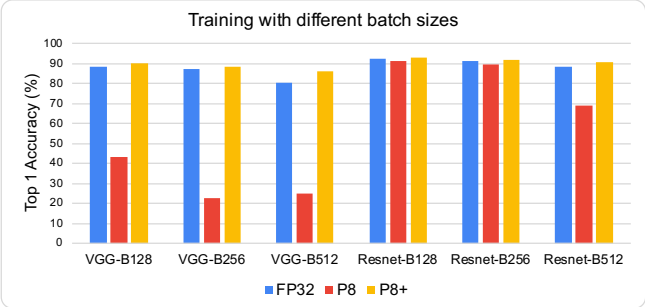


Fig. 4: Training with different batch sizes and the effect of gradient scaling. The variants used are VGG11 and Resnet18. VGG-B128 means VGG11 with 128 images in a batch.

6.3 Inference with lower posit bitwidth

Section 5 shows that posit6 is still good for some inference tasks. In this section we will pick a few tasks which have high posit6 quality and further reduce the precision down to 3 bits to observe the output quality. For each bitwidth, we only select the format with the best accuracy and perform scaling and layer skipping similar to Section 5. The results can be seen in Figure 5.

6.4 Inference with arbitrary number set

To demonstrate the ability of the framework to support designing custom number formats, we conduct experiments with a logarithmic number format. The format is a series of power-of-two values, with the exponent represented with the fixed-point format with N bit. Let $I.F$ be a signed fixed point format with I bits signed

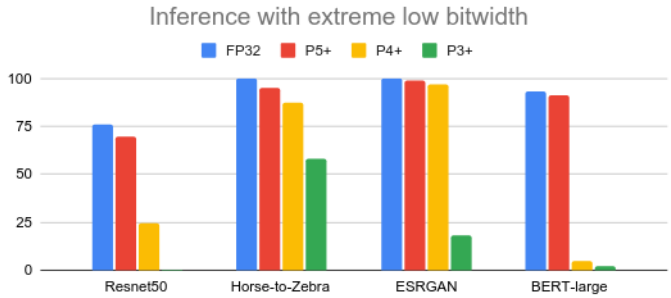


Fig. 5: Inference with bitwidth lower than 6.

integer and F bit fraction part. We can construct a logarithmic format based on the equation : $\pm 2^{I.F}$. The total number of bits required to represent the format is: I+F+1(sign bit). For this type of format, the multiplication is simple because it can be performed by adding the I.F fixed point numbers. To use this feature, the user will generate all possible representable values of the format and supply to our quantizer (*configurable_table_quantize*). Our new quantizer will take all the representable values as an array and round real values to their nearest entry in the given numbers set. Figure 6 shows the inference results of multiple networks on the aforementioned formats. Figure 6, the $L2(7-bit)$ means the $2^{I.F}$ format with I+F+1 = 7 bit. As we can see, a customized format can perform reasonable well on different neural networks with enough bitwidth. However, it cannot reach posit accuracy when using extreme low bitwidth (3-4 bits)

User can easily create their own format, or even a random number set without generating rules and optimized the values in the set to improve accuracy. For optimize number sets with only 4-8 distinct values but achieve good output quality on other networks (ESRGAN, GPT-2), we will have an online demonstration on our GitHub repository. Describing and implementing optimizing method for arbitrary numbers set is beyond the scope of this paper. In this section we only present the features and demonstrations.

6.5 Overhead of the framework

Simulating number formats without hardware support will incur certain overhead on converting the format from and to the primitive FP32 format in the hardware. Our conversions are implemented both in CPU and GPU to support the variety of systems. In the end-to-end pipeline, especially in training, the overhead of simulating novel number formats is overshadowed by other time consuming tasks (data fetching, optimizer, weight update). The overhead of inference and training varies vastly between models. Measuring the computation time to complete one epoch, we got 29% slowdown when training Resnet and 81% slowdown when training VGG. However, when considering the whole end-to-end

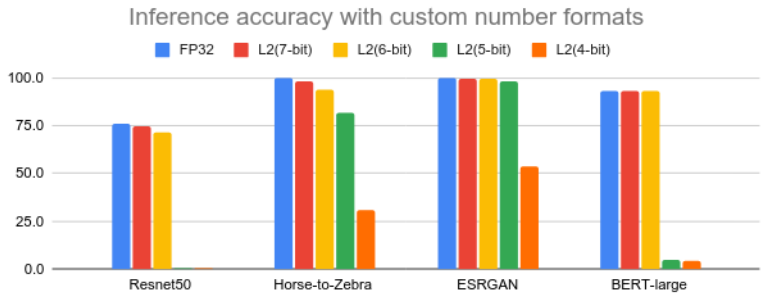


Fig. 6: Inference with a custom $2^{I.F}$ number format

training of 20 epochs, Resnet and VGG got 21% and 70% slowdown respectively. The measured time for inference the whole Imagenet test dataset showed insignificant overhead (<10% in the models we tested). For generic models, our overhead is in range with the original QPytorch framework [48] $\approx$ 30%

7 Conclusion

We have presented the design, implementation and usage of Qtorch+, an extension to Pytorch framework to enable effortless novel number formats inference and training of neural networks. The extension is fully compatible with recent Pytorch version and therefore can be applied to many state-of-the-art models. As shown in our experiment, 8-bit posit arithmetic with scaling and kipping layers are sufficient to pass the 99% FP32 quality standard set by the community. With further extension to remove the restriction on representable number distribution, we support and arbitrary number set for use in Pytorch. This can lead to further development and research on novel low-bitwidth number formats and hardware accelerators in the near future. The tool is available open source and can also be install with *pip* package manager. At the time of writing this paper, our first version of Qtorch+ has received more than 2,000 Python package installations from around the world.

References

1. Abdelfattah, A., Anzt, H., Boman, E.G., Carson, E., Cojean, T., Dongarra, J., Fox, A., Gates, M., Higham, N.J., Li, X.S., et al.: A survey of numerical linear algebra methods utilizing mixed-precision arithmetic. The International Journal of High Performance Computing Applications **35**(4), 344–369 (2021)
2. Bagherinezhad, H., Rastegari, M., Farhadi, A.: Lcnn: Lookup-based convolutional neural network. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 7120–7129 (2017)

3. Boo, Y., Sung, W.: Fixed-point optimization of transformer neural network. In: ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). pp. 1753–1757. IEEE (2020)
4. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language models are few-shot learners. arXiv preprint arXiv:2005.14165 (2020)
5. Carmichael, Z., Langroudi, H.F., Khazanov, C., Lillie, J., Gustafson, J.L., Kudithipudi, D.: Performance-efficiency trade-off of low-precision numerical formats in deep neural networks. In: Proceedings of the Conference for Next Generation Arithmetic 2019. pp. 1–9 (2019)
6. Chiang, W.F., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., Rakamarić, Z.: Rigorous floating-point mixed-precision tuning. ACM SIGPLAN Notices **52**(1), 300–315 (2017)
7. Cococcioni, M., Ruffaldi, E., Saponara, S.: Exploiting posit arithmetic for deep neural networks in autonomous driving applications. In: 2018 International Conference of Electrical and Electronic Technologies for Automotive. pp. 1–6. IEEE (2018)
8. De Silva, H., Santosa, A.E., Ho, N.M., Wong, W.F.: Approxsymate: Path sensitive program approximation using symbolic execution. In: Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems. pp. 148–162 (2019)
9. DE SILVA, H.P.: Software techniques for the measurement, management and reduction of numerica (2020)
10. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: A large-scale hierarchical image database. In: 2009 IEEE conference on computer vision and pattern recognition. pp. 248–255. Ieee (2009)
11. Gaffar, A.A., Mencer, O., Luk, W.: Unifying bit-width optimisation for fixed-point and floating-point designs. In: 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. pp. 79–88. IEEE (2004)
12. Gawehn, E., Hiss, J.A., Schneider, G.: Deep learning in drug discovery. Molecular informatics **35**(1), 3–14 (2016)
13. Gustafson, J.L., Yonemoto, I.T.: Beating floating point at its own game: Posit arithmetic. Supercomputing frontiers and innovations **4**(2), 71–86 (2017)
14. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 770–778 (2016)
15. Ho, N.M., Manogaran, E., Wong, W.F., Anoosheh, A.: Efficient floating point precision tuning for approximate computing. In: 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC). pp. 63–68. IEEE (2017)
16. Ho, N.M., Nguyen, D.T., Silva, H.D., Gustafson, J.L., Wong, W.F., Chang, I.J.: Posit arithmetic for the training and deployment of generative adversarial networks. In: 2021 Design, Automation Test in Europe Conference Exhibition (DATE). pp. 1350–1355 (2021). https://doi.org/10.23919/DATE51398.2021.9473933
17. Ho, N.M., Vaddi, R., Wong, W.F.: Multi-objective precision optimization of deep neural networks for edge devices. In: 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1100–1105. IEEE (2019)
18. Ho, N.M., Wong, W.F.: Exploiting half precision arithmetic in nvidia gpus. In: 2017 IEEE High Performance Extreme Computing Conference (HPEC). pp. 1–7. IEEE (2017)

19. Ho, N.M., Wong, W.F.: Tensorox: Accelerating gpu applications via neural approximation on unused tensor cores. IEEE Transactions on Parallel and Distributed Systems **33**(2), 429–443 (2021)
20. Klöwer, M., Düben, P.D., Palmer, T.N.: Posits as an alternative to floats for weather and climate models. In: Proceedings of the Conference for Next Generation Arithmetic 2019. pp. 1–8 (2019)
21. Krishnamoorthi, R., James, R., Min, N., Chris, G., Seth, W.: Introduction to quantization on pytorch (2020), https://pytorch.org/blog/introduction-to-quantization-on-pytorch/
22. Langroudi, H.F., Carmichael, Z., Gustafson, J.L., Kudithipudi, D.: Positnn framework: Tapered precision deep learning inference for the edge. In: 2019 IEEE Space Computing Conference (SCC). pp. 53–59. IEEE (2019)
23. Langroudi, H.F., Karia, V., Gustafson, J.L., Kudithipudi, D.: Adaptive posit: Parameter aware numerical format for deep learning inference on the edge. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops. pp. 726–727 (2020)
24. LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackel, L.D.: Backpropagation applied to handwritten zip code recognition. Neural computation **1**(4), 541–551 (1989)
25. LeCun, Y., et al.: Lenet-5, convolutional neural networks. URL: http://yann. lecun. com/exdb/lenet **20**(5), 14 (2015)
26. Lo, C.Y., Lau, F.C., Sham, C.W.: Fixed-point implementation of convolutional neural networks for image classification. In: 2018 International Conference on Advanced Technologies for Communications (ATC). pp. 105–109. IEEE (2018)
27. Lu, J., Fang, C., Xu, M., Lin, J., Wang, Z.: Evaluations on deep neural networks training using posit number system. IEEE Transactions on Computers **70**(2), 174–187 (2020)
28. Mattson, P., Reddi, V.J., Cheng, C., Coleman, C., Diamos, G., Kanter, D., Micikevicius, P., Patterson, D., Schmuelling, G., Tang, H., et al.: Mlperf: An industry standard benchmark suite for machine learning performance. IEEE Micro **40**(2), 8–16 (2020)
29. Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., et al.: Mixed precision training. arXiv preprint arXiv:1710.03740 (2017)
30. Nvidia: Scaling Language Model Training to a Trillion Parameters Using Megatron. https://developer.nvidia.com/blog/scaling-language-model-training-to-a-trillion-parameters-using-megatron/ (2021), [Online; accessed 03-January-2022]
31. Oord, A.v.d., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., Kavukcuoglu, K.: Wavenet: A generative model for raw audio. arXiv preprint arXiv:1609.03499 (2016)
32. Posithub.org: Posit Standard Documentation Release 3.2-draft. https://posithub.org/docs/posit_standard.pdf (2018), [Online; accessed 03-January-2022]
33. Pytorch: Pytorch module (2021), https://pytorch.org/docs/stable/generated/torch.nn.Module.html#torch.nn.Module.register_forward_pre_hook
34. Ramanathan, A.K., Kalsi, G.S., Srinivasa, S., Chandran, T.M., Pillai, K.R., Omer, O.J., Narayanan, V., Subramoney, S.: Look-up table based energy efficient processing in cache support for neural network acceleration. In: 2020 53rd Annual

IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 88–101. IEEE (2020)
35. Reddi, V.J., Cheng, C., Kanter, D., Mattson, P., Schmuelling, G., Wu, C.J., Anderson, B., Breughe, M., Charlebois, M., Chou, W., et al.: Mlperf inference benchmark. In: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). pp. 446–459. IEEE (2020)
36. Ryan, J., Lin, M.J., Miikkulainen, R.: Intrusion detection with neural networks. Advances in neural information processing systems pp. 943–949 (1998)
37. Sigtia, S., Benetos, E., Dixon, S.: An end-to-end neural network for polyphonic piano music transcription. IEEE/ACM Transactions on Audio, Speech, and Language Processing **24**(5), 927–939 (2016)
38. Solovyev, R., Kustov, A., Telpukhov, D., Rukhlov, V., Kalinin, A.: Fixed-point convolutional neural network for real-time video processing in fpga. In: 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus). pp. 1605–1611. IEEE (2019)
39. Sordo, M.: Introduction to neural networks in healthcare. Open Clinical: Knowledge Management for Medical Care (2002)
40. Sun, X., Choi, J., Chen, C.Y., Wang, N., Venkataramani, S., Cui, X., Zhang, W., Gopalakrishnan, K., et al.: Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks (2019)
41. Sun, X., Wang, N., Chen, C.Y., Ni, J., Agrawal, A., Cui, X., Venkataramani, S., El Maghraoui, K., Srinivasan, V.V., Gopalakrishnan, K.: Ultra-low precision 4-bit training of deep neural networks. Advances in Neural Information Processing Systems **33**, 1796–1807 (2020)
42. Tobiyama, S., Yamaguchi, Y., Shimada, H., Ikuse, T., Yagi, T.: Malware detection with deep neural network using process behavior. In: 2016 IEEE 40th annual computer software and applications conference (COMPSAC). vol. 2, pp. 577–582. IEEE (2016)
43. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. In: Advances in neural information processing systems. pp. 5998–6008 (2017)
44. Wang, N., Choi, J., Brand, D., Chen, C.Y., Gopalakrishnan, K.: Training deep neural networks with 8-bit floating point numbers. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems. pp. 7686–7695 (2018)
45. Wang, X., Yu, K., Wu, S., Gu, J., Liu, Y., Dong, C., Qiao, Y., Change Loy, C.: Esrgan: Enhanced super-resolution generative adversarial networks. In: Proceedings of the European conference on computer vision (ECCV) workshops. pp. 0–0 (2018)
46. Wu, H., Judd, P., Zhang, X., Isaev, M., Micikevicius, P.: Integer quantization for deep learning inference: Principles and empirical evaluation. arXiv preprint arXiv:2004.09602 (2020)
47. Yazdanbakhsh, A., Park, J., Sharma, H., Lotfi-Kamran, P., Esmaeilzadeh, H.: Neural acceleration for gpu throughput processors. In: Proceedings of the 48th international symposium on microarchitecture. pp. 482–493 (2015)
48. Zhang, T., Lin, Z., Yang, G., De Sa, C.: Qpytorch: A low-precision arithmetic simulation framework. arXiv preprint arXiv:1910.04540 (2019)
49. Zhu, J.Y., Park, T., Isola, P., Efros, A.A.: Unpaired image-to-image translation using cycle-consistent adversarial networks. In: Proceedings of the IEEE international conference on computer vision. pp. 2223–2232 (2017)