Posit Arithmetic for the Training and Deployment of Generative Adversarial Networks

Nhut-Minh Ho*, Duy-Thanh Nguyen†, Himeshi De Silva*, John L. Gustafson*, Weng-Fai Wong*, Ik Joon Chang†
* National University of Singapore † Kyung Hee University

***Abstract*—This paper proposes a set of methods that enables low precision posit™ arithmetic to be successfully used for the training of generative adversarial networks (GANs) with minimal quality loss. We show that ultra low precision posits, as small as 6 bits, can achieve high quality output for the generation phase after training. We also evaluate floating-point (float) formats and compare them to 8-bit posits in the context of GAN training. Our scaling and adaptive calibration techniques are capable of producing superior training quality for 8-bit posits that surpasses 8-bit floats and matches the results of 16-bit floats. Hardware simulation results indicate that our methods have higher energy efficiency compared to both 16- and 8-bit float training systems.**

***Index Terms*—Posit Arithmetic, GAN, Neural Networks**

I. Introduction

Deep neural networks (DNN) have achieved state-of-the-art performance in many visual and language related tasks. However, the cost to train and deploy them is significant. This has led to recent efforts to develop efficient training methods and low-energy training hardware. Initial work in this field explored methods to improve the efficiency of training *convolutional neural networks* (CNN), a popular type of DNN mainly used for image-related tasks. CNNs typically consider image classification with their network structure predominantly consisting of convolutional layers. The output of a majority of CNNs is the discrete classification, that is, an integer. Moreover, their training and inference usually involves only one network.

More recently, neural networks have been adopted for tasks with more complex architectures and interactions. One of the most notable and interesting ideas is the *generative adversarial network* (GAN) [1]. The training of GANs involves at least two distinct neural networks whose structures are different from the traditional ones. The main components of one of those networks are LeakyRELU and $\tanh(x)$ activations, as well as transposed convolutions, which are not used in traditional neural networks. More details about GANs are in Sect. II-A. The output of a GAN can be thousands, or millions of color image pixels, which are sensitive to numerical error. Due to these important differences, previous methods developed to speedup the training of CNNs cannot be easily adopted to train GANs. As we will show in our experiments, the only current method that trains GANs reliably is a half precision framework from Nvidia [2].

Recently, several non-standard real number formats with small bitwidths have been proposed to support the training of DNNs. Notable examples are an 8-bit hybrid floating-point format and posits [3]. In this work, we implement low-precision GAN training with these candidates. Posits have the potential to lower the numerical error for low-precision execution because they concentrate representable values where they are most needed. Details about this property are given in Sect. II-C.

* {minhhn,himeshi,john.gustafson,wongwf}@comp.nus.edu.sg
† {dtnguyen,ichang}@khu.ac.kr

In this paper, we study the application of low-precision arithmetic for the training of GANs. More specifically, our contributions are:

- The first (to the best of our knowledge) successful use of an 8-bit nonstandard (non-IEEE 754™) format for GAN training and 6-bit for GAN deployment.
- A fast approximation of the $\tanh(x)$ function in posit format. The approximation was successfully used in the deployment phase with minimal quality loss.
- Evaluation of software and hardware components of posits and other floating-point alternatives for use in GANs.

In Sect. II-A we will introduce GANs, their training, as well as posits. This is followed by a study of the numerical properties of GAN training in Sect. III, followed by our proposed posit training scheme in Sect. IV. Sect. V describes how we benchmark our proposal against the state of the art as well as the results. This is followed by a discussion on the hardware projections of performance and energy in Sect. VI-B and a conclusion.

II. Background and related work

A. GANs and their application

Fig. 1 shows the overall structure of a GAN during the training and deployment phases. There are several different GANs involving more networks and performing different tasks. Here we show the basic GAN when it was first introduced [1]. The task of a basic GAN is to generate fake data that looks indistinguishable from real data. The basic GAN consists of two networks, the Generator ($\mathcal{G}$) and Discriminator ($\mathcal{D}$). The $\mathcal{G}$ network is used for data generation and in the deployment phase. In the basic task, $\mathcal{G}$'s input is a random tensor which forms the latent space. Each element in the input vector is drawn randomly from a specific range (typically from $N(0,1)$). $\mathcal{G}$'s output is the image with predetermined size. During training, the $\mathcal{D}$ network is used to tell whether the output of $\mathcal{G}$ is good or bad. $\mathcal{D}$ has a task similar to traditional CNNs. It classifies images into two categories: "fake" or "real". Both $\mathcal{D}$ and $\mathcal{G}$ use highly customized activation and architecture, optimized for training each specific task. The training is the iteration between back propagation to optimize both $\mathcal{G}$ and $\mathcal{D}$. The two networks are *adversarial*; $\mathcal{G}$ generates more realistic images, while $\mathcal{D}$ becomes better at distinguishing $\mathcal{G}$'s fake output from real data.

978-3-9819263-5-4/DATE21/©2021 EDAA

Eventually, the two networks reach a state where the output of $\mathcal{G}$ defeats $\mathcal{D}$. Then, the $\mathcal{D}$ network is discarded and the $\mathcal{G}$ network is deployed. Recently, more complicated training method and networks architecture can perform sophisticated tasks such as style transfer, image coloring, and drawing a realistic scene. One example is CycleGAN [4] which has two $\mathcal{G}$s and one $\mathcal{D}$.

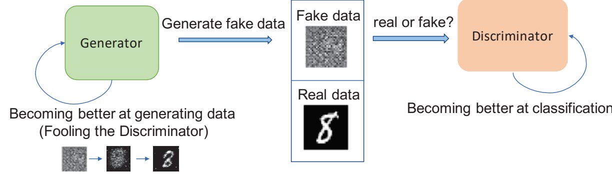


Fig. 1: Generative Adversarial Networks and their application

B. Low Precision CNN and GAN Training

Due to the recent emergence of CNNs, techniques for improving their training and inference efficiency have received widespread interest. Here, we use FP16 and FP32 to refer to IEEE 754 binary16 and binary32 floats. One noteworthy technique is to reduce the bitwidth of real-valued parameters in FP32. For inference, low bitwidth fixedpoint formats and quantizations have been proposed [5], [6]. Besides, several 16-bit float formats such as FP16, bfloat16, and DLfloat have been proposed for training [2], [7], [8]. More recently, 8-bit format has been successfully utilized for weights, activations, gradients or a combination of these [9], [10]. In using posit for CNN training, 8-bit Posit has been used for training [11], [12]. These works focus on CNN training and provide some insight into the possible saving on posit MAC units. Of particular interest is *HFP8*, a hybrid scheme that uses different 8-bit formats for forward and backward propagation; the accumulator, batch norms and optimizer operations use 16-bit floats with a 6-bit exponent [13]. Because HFP8 is the state-of-the-art work in float-based training, we refer to their 8-bit format as FP8 in this paper.

Different from CNN, GAN output is a high resolution image where each pixel value affects the output quality. Also, GAN training is quite complex and susceptible to numerical instability [14]. Thus, GANs are trained using FP32 for safety. Currently, the most reliable method to use precision less than FP32 for GAN training is Nvidia's mixed precision training framework [2]. This framework has three modes of optimization dependent on which operations will be performed in FP16. The framework only recommends "O1" mode, which uses FP16 GEMM for convolution-alike layers and leaves other operations in FP32. To the best of our knowledge, there are no proposals to use a bitwidth as low as 8 for the training of GANs. It is well known that during inference or deployment phase, neural networks can use lower bitwidth than during training; for GAN deployment, we have not seen any proposals to use low bitwidth apart from using FP16.

C. Posit Format

Posits are a unique number representation that is rapidly gaining traction due to their ability to vary the significant digits (length of fraction) within the dynamic range, thereby allowing the more important parameters of an application to have more accuracy. A posit environment is defined by the length of the posits, n, and the exponent size, *es*. The default *es* value is 2. The format of a posit is shown in Fig. 2. The regime field is of variable length and consists of r bits identical to R_0 terminated by $1 - R_0$ ($r + 1$ bits total length) or by reaching the end of the posit (r bits total length). The value of R is $-r$ if R_0 is 0, and $r - 1$ if R_0 is 1. If the posit is terminated by the exponent field, zeros are padded to end until the field is *es* bits wide. The hidden bit for the fraction is always 1. Posits also can use a structure known as the *quire* to accumulate the exact dot product without rounding error. More details about their rounding, special values, operation and arithmetic can be found in the Posit Standard [15].

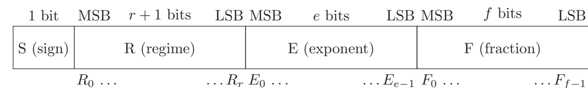


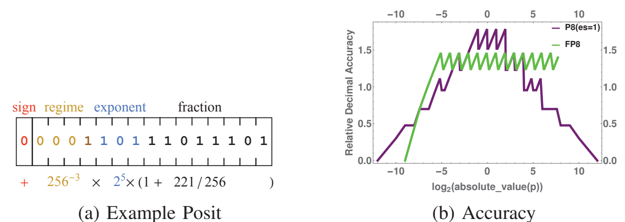
Fig. 2: Posit Format

An example posit decoding is shown in Fig. 3a. Here $n = 16$ and *es* $= 3$; $2^{2^{es}}$ is raised to the power of the regime value, -3. The remaining fields are decoded as shown. Fig. 3b shows the accuracy distribution of every consecutive pair of posits when *es* $= 2$ and $n = 8$, that is, P8. (The definition for relative decimal accuracy can be found in [16]). It shows the higher concentration of high accuracy values about magnitude zero on the x-axis. Scaling can move this range to where accuracy is most needed in the application.

To better label different training results in this paper, we use P(n, *es*) to indicate an n-bit posit format with *es* exponent bits. P6, P8 and P16 refer to 6-bit, 8-bit and 16-bit posits with the *es* size defined in the corresponding section in this paper.

III. The Numerical Properties of GAN Training

To find the best training method, we first histogram all values that occurred during GAN training. The histograms of weights and activations of DCGAN over different training epochs are presented in Fig. 4. For readability, we only present the histograms when 0% (first epoch), 50%(halfway) and 100% (fully-trained) of the epochs are completed. In Fig. 4, the z-axis is the frequency of $\log_2(|values|)$ bins. $W_$ and $A_$ indicate weights or activations; $G_$ and $D_$ indicate Generator $\mathcal{G}$ or Discriminator $\mathcal{D}$. 0%, 50%, and 100% refer to which epoch



(a) Example Posit (b) Accuracy

Fig. 3: Posit - Example and Accuracy Distribution

the histogram is recorded (at the first epoch, half training time and the last epoch, respectively).

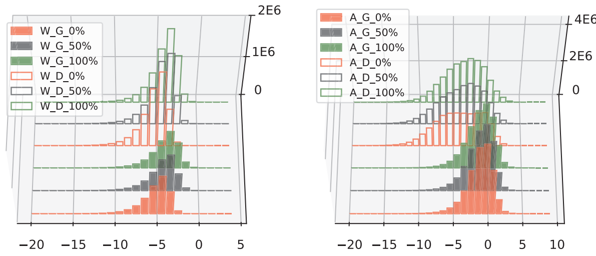


Fig. 4: Histogram of non-zero weights and activation values during training of DCGAN

IV. Our proposed training scheme

Based on the above distribution of values during GAN training, we propose different methods to correct the weight, activation and gradient of each generator and discriminator.

A. Training and deployment system

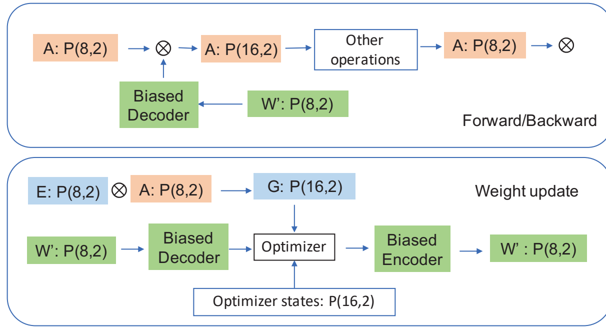


Fig. 5: Training using different posit formats

Fig. 5 shows the data type of each component of the networks during training. The $\otimes$ means dot product and operations that can be implemented by it. For simplicity, we only show the forward pass when training. In the backward pass, the weights (W) and Error (E) are used in backpropagation algorithm. The number format for W and E during the backward pass is the same as for the forward pass, which is $\mathrm{P}(8,2)$. Note that, W′ in Fig. 5 is the scaled weights using our method in the next Section. Before and after the computation of each posit value, the encoder and decoder are used. 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:

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

The standard posit decoding and encoding algorithm can be found in several works and the original posit introduction [3], [11], [15]. The small change in our Biased Encoder and Decoder is to add an exponent bias t when encoding weights, and subtract it when decoding weights. In Fig. 5, ‘Biased Encoder’ and ‘Biased Decoder’ are used whenever an operation involving scaled weight is performed. For normal operation, the normal decoder and encoder are used. For other operations and weight updates, because the change in weights is small and the optimizer is very sensitive to quantization error, we keep other optimizer parameters and gradient accumulator at $\mathrm{P}(16, 2)$. This is the standard practice in low precision CNN training [13].

A key feature of this training system is to keep the exponent *es* the same for both the 16-bit and 8-bit formats and enough to not hurt the output quality. Unlike floats, posits can change precisions n simply by appending 0 bits to make them larger and rounding the lowest-significance bits to make them smaller, when the *es* value does not change. [15], [17].

Although it is conceivable that some GANs may allow lower bitwidth than $\mathrm{P}(8, 2)$, we try to set it as a standard format for all GANs training without additional analysis from end users. Exponent size determination is rather simple; we try *es* $= 1$, 2, or 3, and *es* $= 2$ gives the best output quality across different GANs. *es* ≤ 1 fails to converge in most GANs due to underflow and overflow, while *es* ≥ 3 gives lower output quality due to reduced fraction accuracy. For inference system, only the forward pass of network $\mathcal{G}$ is involved. We will show the output quality for different formats in Sect. V-C.

B. Parameters scaling and exponent bias

The histograms in Fig. 4 show that the weights can be scaled for more accurate posit representation. Because the weights are concentrated in the range $[2^{-4}$ to $2^{-5}]$, we can shift the peak of the histogram to the range with highest posit accuracy, near 2^0. The activations histogram is concentrated at around $[2^{-2}$ to $2^0]$; thus, scaling activations is unnecessary. Note that weight scaling is not beneficial in float-based training schemes, because their accuracy distribution is flat (see Sect. II-C).

The weights are scaled to be better represented by limited precision only when they are in the encoded posit format. When weights are decoded or in an intermediate form in the data bus, they are unpacked as described in Sect. II-C. The decoded form can hold the weights without scaling.

We scale using the posit encoder and decoder, not by multiplication. 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. We only need to store the scaled weights in 8-bit encoded form. Weights are decoded to the unpacked form $\{S, R, E, F\}$ just before sending to the MAC unit; we restore the weights to their original value using the Biased Decoder. We keep weights in the scaled form by adding the exponent bias t when encoding them during weight update.

For the choice of t, the histogram of each layer varies slightly. Except for the first and the last layer, other layers have similar histograms during training as shown in Fig. 4. Thus, we propose a method to calculate the scale given a set of samples

from weights (W). The method to find t is summarized below, where argmax is the index of the maximum array element:

$$\{\text{bins, frequencies}\} = \text{histogram}(\log_2(|\text{W}|)) \quad (2)$$
$$t = \left\lfloor 0 - \text{bins}[\text{argmax}(\text{frequencies})] \right\rfloor$$

By enforcing that t is an integer, we also observe that the value t remain stable across different layers. Thus, we can set a single t value for all layers in a network. Across different GANs, the value of t varies slightly from 3 to 5. To find t, we implement a module to record the histogram of the first training iteration, and set t for usage afterwards. In our experiments, further calibration of t does not significantly improve the training results. The additional analysis overhead during training does not seem to pay off.

C. Loss Scaling

To scale the gradient during training to the representable region, loss scaling is the standard approach for low precision number formats. Loss scaling scales the loss only once by multiplying by the scale s. The scale will then propagate and scale all intermediate gradient values accordingly. Finally, weight updating will be corrected by multiplying the weight gradient by $1/s$. It is more efficient than scaling every gradient value and scaling back the gradient output at each layer during backpropagation. For float formats, the goal is to try to avoid underflow to 0. The automatic scaling module in Nvidia Apex [2] increases s until an overflow occurs; then the scale will be decreased. This scaling method is illustrated by the blue histogram in Fig. 6. The maximum values occurring during runtime determine the actual scale that shifts the histogram to near the maximum representable value of FP16 (65504).

For posit formats, the goal is to shift the center of the distribution towards the range of posits that have the highest accuracy (2^{-4} to 2^4 for $es = 2$). The scale value s is determined by the method in Eq. 2. Because the scale does not change much during training time and to reduce overhead of recording and analyzing the histogram, we set the scale based on the first iteration. As will be shown in the experiments, the fixed scaling factor in $\text{P}(8, 2)$ gives results comparable to FP16 with automatic loss scaling.

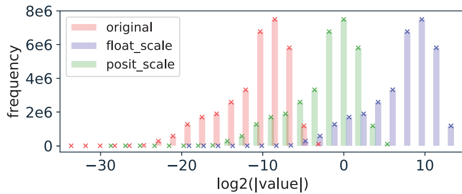


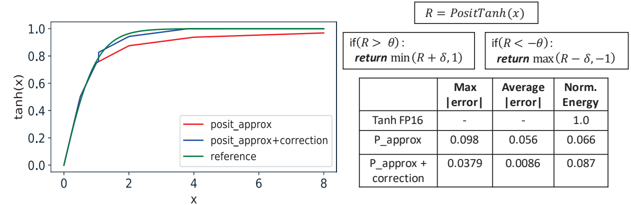
Fig. 6: Conventional loss scaling in floating point format and our proposed loss scaling based on gradient histogram

D. Fast Approximation of $\tanh(x)$ with Posits

The posit format allows a highly efficient trick to approximate sigmoid functions, summarized in Eq. 3, where x is a posit format with $es = 0$ [3].

$$\text{Sigmoid}(x) = (x \oplus 8000_{16}) >> 2 \quad (3)$$
$$\text{PositTanh}(x) = 2 \cdot \text{Sigmoid}(2x) - 1 \quad (4)$$

Most GANs use $\tanh$ as the output layer in the Generator. We can replace the exact $\tanh$ with our approximation in Eq. 4 in the deployment phase. We also propose a correction scheme to limit the maximum error caused by the approximation. Simply, we add a small quantity, δ, to the result if it exceeds a threshold θ. In our experiments, we use $\delta = 0.068$ and $\theta = 0.76$; both quantities were found through exhaustive search. We use P(16,0) for this approximation because it can retains the information from the output of the previous layer in $\text{P}(16, 2)$. The approximation, its error (in region[0,8]) and normalized energy consumption(using configuration in Sect. VI) can be seen in Fig. 7. Because $\tanh$ is an odd function and the approximation also has that property, we only plot the region $[0, 8]$ for clarity. We apply the correction scheme as default in all of our experiments because it has much lower error compared to normal $\text{PositTanh}()$.



	Max |error|	Average |error|	Norm. Energy
Tanh FP16	-	-	1.0
P_approx	0.098	0.056	0.066
P_approx + correction	0.0379	0.0086	0.087

Fig. 7: $\tanh$ approximation and correction scheme

V. Accuracy measurement on Pytorch

A. Software module for training with posits

We developed a module to integrate posits into the Pytorch framework. This helps us quickly measure the accuracy of the training, and implements all scaling and loss scaling methods described in Sect. IV. We also implemented the two variants of the $\tanh$ activation function mentioned in Sect. IV-D. We extended QPytorch [18] to support posits with arbitrary bitwidth. The module enables automatic gradient quantization and can compute exact matrix multiplication using the quire, as implemented in Sect. VI-B. Note that quire dot products gives higher accuracy than the standard FP multiply-adds that accumulate rounding error. However, implementing a true quire in software would slow down GAN training because quire precision (256-bit fixed point) exceeds even FP64. Thus, for the accuracy measurement, we use an FP32 accumulator as default for the fastest training time. Note that this provides *lower bound* for output quality, because using the quire would result in at least this level of output quality or higher. Based on our experiments, increasing the accumulator up to FP64 (as it is supported by our GPU's hardware) does not change the output quality much because the output is then quantized to P16 for subsequent computation. The module has source code and tutorial available at [19].

B. Training output quality

We use the above framework to validate the accuracy of GANs trained by our proposed method. In addition, we also trained the selected GANs using popular float types introduced recently. More specifically, half precision proposed by Nvidia in Apex [2] and FP8 with the re-implementation in [18]. Our observation confirmed that the Apex framework works well in the default *O1* mode recommended by Nvidia as described in Sect. II-B. The other training modes did not converge in some of our experiments. Thus, we show the accuracy of Apex *O1* results for FP16 in this section as the reference output quality.

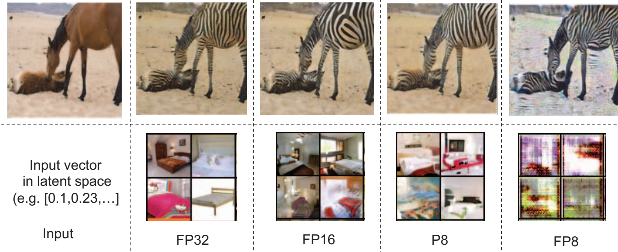
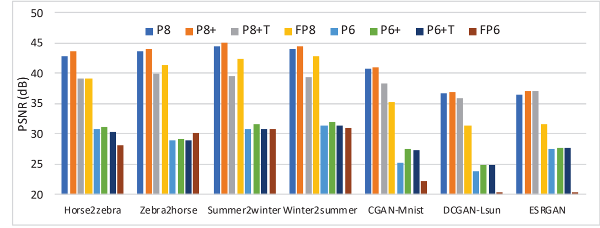
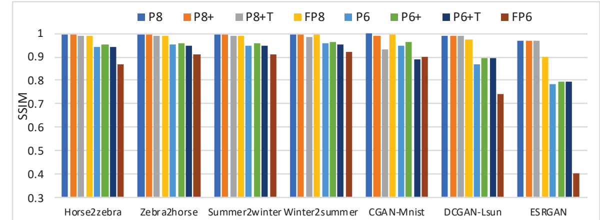


Fig. 8: Sample GAN outputs, trained with different formats

GANs trained using different number formats will give different output even given the same input and trained with the same random number generator state. The behavior can be observed in Fig. 8. The first row is the task to transform an image of a horse to that of a zebra. GANs trained with different formats generate different stripe patterns on the zebra body. This behaviour makes it difficult to justify the approximated training result because there is no ground truth image to compare with. Recently, to subjectively qualify the output images, Inception Score (IS) and Frechet Inception Distance (FID) [14] were introduced as the metrics to measure the quality of GAN-generated output. Table I shows the training output quality by using FP32, FP16, FP8, P8 and P8 with scaling techniques (P8+). Usually, a lower FID score means higher output quality. ESRGAN is one exception in Tab. I. The ESRGAN task is to upscale an input image to higher quality; the inputs and reference outputs are provided and the required metric is to compare PSNR (dB) with the validation dataset. A higher PSNR metric indicates better result. As we can see in the table, P8 outperforms FP8. With scaling techniques, the P8 training system matches the quality of the current FP16 training standard in most of the GANs. We used the same configuration (random number generator state, default optimizer hyper parameters, and the number of epochs set by the original work) when training each GAN using different formats. For the validation set, Horse2zebra, Zebra2horse, Summer2winter, and Winter2summer use the corresponding test set provided in [4]; ESRGAN uses *Val14* set; DCGAN (trained on LSUN Bedroom dataset) [20] and CGAN (trained on MNIST) [21] results are measured by generating 10000 images. The largest experiment is ESRGAN with more than 500 layers.



(a) PSNR results.



(b) SSIM results.

Fig. 9: Quality of deployment using different formats.

	FP32	FP16	P8+	P8	FP8
CGAN-Mnist	67.31	**64.58**	70	75.41	423.28
DCGAN-Lsun	**43.12**	49	44.82	49.54	318.07
Horse2zebra	77.2	64.3	**61.1**	70.1	75.63
Summer2winter	77.6	78.52	77.12	**75.67**	84.01
Zebra2horse	**134.2**	134.5	138.36	148.49	138.13
Winter2summer	75	73.46	**72.79**	74.75	77.16
ESRGAN (PSNR ↑)	24.67	24.69	**24.74**	24.53	4.62

TABLE I: Output quality in FID score of popular GANs trained by different formats

C. Post-training deployment output quality

Unlike training which does not have ground truth images for comparison, the deployment of GAN can be viewed as the quantization of pretrained models. Consider a pretrained $\mathcal{G}$ network that was supposed to run on FP32, we can use the output of GAN in FP32 as our ground truth to compare with (ESRGAN also uses FP32 output for consistency). Thus, for the deployment, we compare output image pixels using conventional error metrics used in image processing applications. For completeness, we measured both (Peak Signal-to-Noise Ratio) PSNR and (Structural Similarity Index Measure) SSIM as they are the two most popular metrics for image quality. The GAN output quality with different number formats can be seen in Fig. 9. We further added 6-bit floating point (FP6) for comparison. Posit schemes P8 and P6 use $es = 1$, which gives the best output quality in deployment. P8+ and P6+ use our scaling techniques presented in Sect. IV-B. P8+T and P6+T use both the scaling technique and $\tanh$ approximation in Sect. IV-D. We can see that most of P6+T and P6+ experiments have SSIM > 0.9, which indicates a high output quality.

VI. Hardware simulation

A. Methodology

To compare hardware costs, we used the design with posit quire dot products in [22]. For floats, we use the Synopsys

Designware library. We use the integrated power estimation in Synopsys' design compiler with LP65nm CMOS standard cells. The synthesis latency of each component can be extracted with a 1 GHz clock. We used the GEM5 simulator with MLPACK to implement DC_GAN. The simulator configuration is presented in Table II. The simulator outputs the performance improvement of each network and the statistics of instructions issued during training and deployment. We then multiply the number of instructions and their respective energy consumption for the final energy consumption.

TABLE II: Evaluation Setup

CPU	X86 DerivO3CPU 1Ghz
Floating-Point Unit	Designware & Deepfloat [22] @ CMOS LP65nm
System Configuration	128 kB 2-way L1, 1 MB 2-way L2 64-byte cache line, 1 GHz system clock
DRAM Configuration	16 GB DDR4_2400_16×4, scheduling mode: FRFCFS

B. Experimental results

Fig. 10-a shows the normalized energy consumption and runtime when training DCGAN with different schemes (FP16, FP8 and P8+). We can compare FP8 with P8 since they have the same bitwidth. The posit format consumes less energy and less computing time than the float format of the same size, similar to results reported for CNNs in the literature.

Fig. 10-b shows the results in the deployment phase of three schemes FP16, FP8 and P6+T. The advantage of reducing the bitwidth is superlinear because power is reduced at least linearly, but execution time is also reduced, compounding the effect. The cost of multiplication also grows approximately as the square of the bitwidth. In deployment, the effect on using low bitwidth is amplified. It is because the proportion of costly division operations (used to compute tanh in the Generator) will be increased when the Discriminator is discarded.

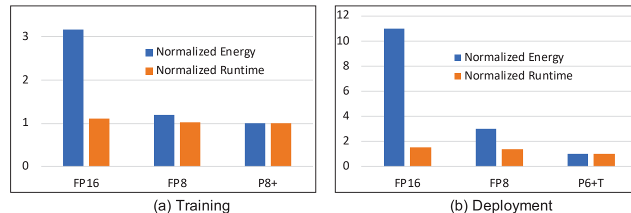


Fig. 10: Normalized Performance and Energy.

VII. Conclusion

We have presented methods to enable high-quality GAN training using posits with only 8 bits. For deployment, we can even use 6-bit posit format and retain acceptable output quality. The methods were realized in the modern training framework Pytorch which allows user to freely experiment with more sophisticated training schemes. On the hardware side, we simulated the delay and energy based on a realistic architecture to estimate the benefit of using posits in a full system instead of only simulating the arithmetic unit as was done in previous works. The experimental results show a promising use case for low bitwidth non-IEEE number formats in special application such as GANs. We believe this work will accelerate further research and deployment of posits and other low precision formats in the emerging GAN domain. We believe low precision accelerators will become more popular in the near future as their use cases become more appealing.

Acknowledgment

This research was supported in parts by Singapore Ministry of Education Academic Research Fund T1-251RES1818 and MOE2016-T2-2-150. This was also supported by National Research Foundation of Korea(NRF) funded by Ministry of Science and ICT(2020M3F3A2A0108575).

References

[1] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[2] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[3] J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, pp. 71–86, 2017.

[4] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," in *Proceedings of the IEEE Int. Conf. on Comp. Vision.*, 2017, pp. 2223–2232.

[5] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *arXiv preprint arXiv:1806.08342*, 2018.

[6] N.-M. Ho, R. Vaddi, and W.-F. Wong, "Multi-objective precision optimization of deep neural networks for edge devices," in *DATE 2019*. IEEE, 2019, pp. 1100–1105.

[7] N. Burgess, J. Milanovic, N. Stephens, K. Monachopoulos, and D. Mansell, "Bfloat16 processing for neural networks," in *2019 IEEE 26th Symp. on Computer Arithmetic*. IEEE, 2019, pp. 88–91.

[8] A. Agrawal, B. Fleischer, S. Mueller, X. Sun, N. Wang, J. Choi, and K. Gopalakrishnan, "Dlfloat: a 16-bit floating point format designed for deep learning training and inference," in *ARITH 2019*. IEEE, pp. 92–95.

[9] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, "Scalable methods for 8-bit training of neural networks," in *Advances in neural information processing systems*, 2018, pp. 5145–5153.

[10] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *NeurIPS 2018*, 2018, pp. 7675–7684.

[11] 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 *DATE 2019*. IEEE, 2019, pp. 1421–1426.

[12] J. Lu, C. Fang, M. Xu, J. Lin, and Z. Wang, "Evaluations on deep neural networks training using posit number system," *IEEE Trans. on Computers*, 2020.

[13] X. Sun, J. Choi, C.-Y. Chen, N. Wang, S. Venkataramani, V. V. Srinivasan, X. Cui, W. Zhang, and K. Gopalakrishnan, "Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks," in *NeurIPS 2019*, 2019, pp. 4900–4909.

[14] M. Lucic, K. Kurach, M. Michalski, S. Gelly, and O. Bousquet, "Are gans created equal? a large-scale study," in *NeurIPS*, 2018, pp. 700–709.

[15] https://posithub.org/docs/posit_standard.pdf, accessed: 2020-01-07.

[16] H. De Silva, J. L. Gustafson, and W.-F. Wong, "Making strassen matrix multiplication safe," in *HiPC*. IEEE, 2018, pp. 173–182.

[17] H. De Silva, "Software techniques for the measurement, management and reduction of numerical error in programs," 2020, PhD thesis, NUS.

[18] T. Zhang, Z. Lin, G. Yang, and C. De Sa, "Qpytorch: A low-precision arithmetic simulation framework," *NeurIPS 2019 EMC2 Workshop*, 2019.

[19] https://github.com/minhhn2910/QPyTorch.

[20] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[21] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *arXiv preprint arXiv:1411.1784*, 2014.

[22] J. Johnson, "Rethinking floating point for deep learning," *arXiv preprint arXiv:1811.01721*, 2018.