HyperBlock Floating Point: Generalised Quantization Scheme for Gradient and Inference Computation

Marcelo Gennari do Nascimento
University of Oxford - Active Vision Lab
Parks Road, Oxford OX1 3PJ
marcelo@robots.ox.ac.uk

Victor Adrian Prisacariu
University of Oxford - Active Vision Lab
Parks Road, Oxford OX1 3PJ
victor@robots.ox.ac.uk

Roger Fawcett
Intel Corporation
Eclipse, Globe Park, Marlow SL7 1YL
roger.fawcett@intel.com

Martin Langhammer
Intel Corporation
Eclipse, Globe Park, Marlow SL7 1YL
martin.langhammer@intel.com

Abstract

Prior quantization methods focus on producing networks for fast and lightweight inference. However, the cost of unquantised training is overlooked, despite requiring significantly more time and energy than inference. We present a method for quantizing convolutional neural networks for efficient training. Quantizing gradients is challenging because it requires higher granularity and their values span a wider range than the weight and feature maps. We propose an extension of the Channel-wise Block Floating Point format that allows for quick gradient computation, using a minimal amount of quantization time. This is achieved through sharing an exponent across both depth and batch dimensions in order to quantize tensors once and reuse them during backpropagation. We test our method using standard models such as AlexNet, VGG, and ResNet, on the CIFAR10, SVHN and ImageNet datasets. We show no loss of accuracy when quantizing AlexNet weights, activations and gradients to only 4 bits training ImageNet.

1. Introduction

Convolutional Neural Networks have become ubiquitous in a variety of applications, from Image Recognition [18] to Face Identification [34]. One drawback of these models is the amount of computational resources required. It has been estimated that in order to achieve the current 11.5% error benchmark on ImageNet, around 10^6 $USD has been spent on computation cost alone [36]. Therefore, significant amounts of research has been conducted in order to make models run quicker, and using less memory and energy.

Quantization is now an important way of addressing this problem. This method has been successfully used by the industry to deploy to embedded devices [14] and servers [8]. However, most of the effort has been put into quantizing models for inference, focusing only on the forward pass. There is significantly less research on gradient quantization.

A prohibitive amount of time is used in the development cycle to train networks: the backward pass requires twice as many convolution calculations as the forward pass. Therefore, it is paramount to accelerate the backward pass.

Many factors contribute to this, including the requirement for higher precision for gradients, and the focus on deploying to the end user rather than optimizing for production. Moreover, many methods use different quantization schemes (hardware) for the forward and backward pass. For example, Habana Labs uses a specific hardware for training (Gaudi [19]), and another for inference (Goya [20]); previous papers like DoReFa-Net [49] use different quantization schemes for gradients and weights/ activations.

Building on this motivation, we propose a method to address the challenges in calculating gradients using low precision, whilst using the same quantization building block both at training and inference. Our method quantizes both passes of a convolutional neural network, such that the bulk of the multiplications and additions are done in narrow precision. Our contributions are as follows:

- A method for quantizing forward and backward passes using the same hardware and quantization scheme.
- An extension of block floating point which allows for less quantization conversion operations.
- Robust results in a variety of datasets and different families of architectures, including no loss of accuracy when quantizing AlexNet down to 4 bits.

Our method employs the Block Floating Point (BFP) method to backpropagation, which can achieve high accuracy using narrow bitwidths by using simple heuristics for exponent sharing. Furthermore, we introduce a variant of BFP that we call HyperBFP, which spares redundant quantization passes to tensors and makes the system simpler to design, requiring lower memory bandwidth.

Using these methods, we were able to achieve high quality results even on the most challenging datasets such as ImageNet. We show that when small block sizes are used, such as 8 or 16, we can get little loss of accuracy when quantizing to as low as 2 bits. When we use slightly bigger block sizes such as 32, we achieve no loss of accuracy even when quantizing weights, activations and gradients to 4 bits.

2. Related Work

Many approaches have been tried to accelerate neural network computation. The most natural way is to select better models. The goal is to find architectures that run quicker than traditional methods on retail hardware, such as CPUs and GPUs. This resulted on finding efficient architectures, both human-designed such as MobileNetV2 [31], ShuffleNet [47], and automated methods using Neural Architecture Search (NAS) such as MNasNet [35] and FBNets [40]. Quantization methods can be used in conjunction with better model selection to get additional speed and energy benefits. This has been shown to work even on the highly efficient MobileNet [15]. There is an extensive literature in quantization, which can be divided by two tasks: Quantized Inference, and Quantized Training.

2.1. Quantized Inference

The goal is to quantize both weights and activations, allowing for quick inference algorithms, since using compatible hardware operations in lower precision are quicker and less spacious than regular FP32 format. Most of these methods assume that the training can be performed using FP32 format, but the inference is restricted to low bitwidths. An example of this type of method include QNN [11], which was one of the first methods to include binary weights. Other examples include Halfwave-Gaussian Quantization (HWGQ [2]), ABC-Net [24], LQ-Nets [44], and PSGD [16]. The work of [22] substitutes the ubiquitously used Straight-Through Estimator (STE) with an adaptive gradient scaling function that shrinks the error given when using STE. Another variation of quantized inference is to quantize models without the need for (unlabelled) data, or using synthetic data, which is referred to as post-training quantization. For example [23,25,26,46] use only a small amount of (unlabelled) data (or synthetic data) to quantize a network. The advantage of this method is that no further training is required, often utilizing only a calibration step.

2.2. Quantized Training

The absolute majority of the quantization methods focus on quantizing the forward pass (inference) part of the model only. There are a few methods though that are able to also quantize the gradients to low bitwidth [41,43]. This also means that the backward pass can be done quicker in specialized hardware. Some of these methods include [45], which uses a fixed-point only computation for forward and backward-pass, employing different bitwidths for different layers (from int8 for weights and activations, to a combination of int8, int16 and int24 for gradients); [6] uses an end-to-end block floating point inspired quantization technique to train CNNs with forward and backward passes quantized; [1] also shows good results when quantizing using 8 bits. DoReFa-Net [49], which aims to achieve binary weights, activations and gradient quantization by using hand-designed transformation functions.

The papers that are most related to our work include SWALP [42], and [48], both of which are able to successfully quantize both the backward and forward pass of convolutional neural networks. Other methods such as ALQ/AMQ [7] and TRN [39] quantize only the stochastic gradient for communication efficiency.

Aside from the academic papers on quantization, the industry has adopted different floating point standards in order to accelerate computation using narrow precision. NVidia uses the TensorFloat32 format on the A100 GPUs [29] which uses 8 bit exponents and 10 bit mantissa; and Google uses the BFloat16 format on the TPU v3 [38], which uses 8 bits exponents and 7 bits mantissa. 8-bit floating point numbers has also being proposed for training [37].

This paper will focus on quantized training, and we will expand the literature by showing a new method of quantizing gradients based on the BFP format.

3. Preamble

The Single Floating Point (FP) format, defined by IEEE-754 [12], consists of representing a number using 32 bits (FP32): 1 for the sign, 8 for the exponent, and 23 for the mantissa. Its decimal value can be calculated by the formula $a = (-1)^{b_0} \times 2^{((b_1 b_2 ...)_2 - 127)} \times 1.b_9 b_{11} ... b_{31}$, where $()_2$ indicates binary format and b_n is the n^{th} binary value of this representation: b_0 fixes the sign, $b_{1...8}$ specifies the exponent, and $b_{9...31}$ specifies its mantissa.

Different formats balance range and precision by changing the number of bits used for the exponent and the mantissa (see BFloat16 [38], TensorFloat32 [29], and FP8 [37]). Fixed point formats are slightly more limited as they lack the possibility of a large range of values, but also require smaller area for operations in the hardware.

The BFP format, rather than being a representation for a single value, consists of a format for a group of values.

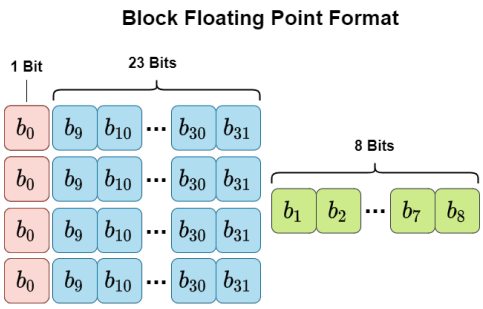


Figure 1: Diagram depicting Block Floating Point format for a block size of 4. Note that in the BFP format, the sign + mantissa bits are fixed-point integer, and the shared exponent acts as a scale value across the entire block.

The representation is similar to FP32, but instead of having one exponent for each number, it shares the same exponent across a certain number of values. Figure 1 shows how it works. Instead of each number being represented by an individual exponent and mantissa values, BFP *shares* a single exponent across in this case 4 values, which are represented then in *fixed-point*. BFP then balances the range of floating point with the efficient computation of fixed point.

This strategy has been applied to CNNs and tensors before. Previous papers have used different strategies for sharing exponents: for inference only [5,27,28], and for training as well [4,6,32,42]. Most of the methods share the exponent across entire tensors, or batches. SWALP [42] uses both *"Big-block"* design (which shares one exponent per batched tensor) and *"Small-block"* design, which also shares exponents across batch normalization layers, and fully connected (FC) layers. We believe that these options are unnecessarily restrictive, and more accuracy can be obtained by using smaller blocks. We follow the strategy of [27], in which exponents are shared *channel-wise*. This allows for a fine-grained trade-off between accuracy and computation, whilst taking advantage of fixed point operations in hardware. When converting from FP32 to BFP format, across the channel dimension, for each vector of size blk, the maximum exponent is chosen and it is shared among all other values. This is done by calculating the shift of the mantissas needed for each value to enforce the same exponent.

During inference, when this format is used, the convolution can be performed by using only integer Multiply and Accumulate (MAC) operations per block (channel-wise), plus a small number of FP summations of the resulting scalars, which was shown to be effective for inference.

4. Method

We propose a method for quantizing weights, activations, and gradients for computation in low-precision based on the BFP format. BFP is not invariant to transposition in the input / output channel dimensions (see Figure 2), which complicates convolution operations in the backward pass.

Given a kernel $\mathbf{W}^{(o,i,k,k)}$, input $\mathbf{A}^{(b,i,w_a,h_a)}$, and feature map $\mathbf{F}^{(b,o,w_a,h_a)}$, where i is the input channels, o is the output channels, w_a is the width, h_a is the height, b is the batch size, and k is the kernel width and height, the forward pass of the convolution is defined as:

$$f_{b,o,w_a,h_a} = \sum_{k}^{3}\sum_{k}^{3}\underbrace{\sum_{i_d}^{i} w_{o,i_d,k,k} \cdot a_{b,i_d,w_a+k,h_a+k}}_{\text{Inner MAC}}$$

assuming kernel size equals to three, padding and stride adjusted to preserve the same width and height.

The convolution can be seen as transforming the input features from having i channels, to the output features having o channels. For BFP, the weight and activation share the exponent in blocks, allowing for low-precision computation. Therefore, the Inner MAC in the right hand side of the equation above becomes:

$$\sum_{n}^{\#blk}\underbrace{\sum_{i_b}^{blk} w_{o,n\cdot blk+i_b,k,k} \cdot a_{b,n\cdot blk+i_b,w_a+k,h_a+k}}_{\text{Done in low-precision}}$$

where $\#blk$ indicates the number of blocks depthwise in the convolution (for example, if $blk = 32$ and $i = 128$, then the number of blocks is 4).

For backpropagation to work, both the gradient of the loss with respect to the kernel $\nabla l_{\mathbf{W}} \in \mathbb{R}^{(o,i,k,k)}$, and with respect to the input features $\nabla l_{\mathbf{A}} \in \mathbb{R}^{(b,i,w_a,h_a)}$, given the gradient with respect to the feature map $\nabla l_{\mathbf{F}} \in \mathbb{R}^{(b,o,w_a,h_a)}$, need to be calculated. Both can be obtained by using transposed convolutions:

$$\nabla l_{\mathbf{A_i}} = \text{tconv}(\mathbf{W}, \nabla l_{\mathbf{A_o}})$$

$$\nabla l_{\mathbf{W}} = \text{pconv}(\mathbf{A_i}, \nabla l_{\mathbf{A_o}})$$

where tconv indicates the transpose convolution operator, and pconv indicates the permute convolution operator. Both of them can be defined in terms of a normal convolution (ignoring stride, padding, grouping, etc).

$$\text{tconv}(\mathbf{W}, \nabla l_{\mathbf{A_o}}) \coloneqq \text{conv}({\mathbf{W_{180}}}^T, \nabla l_{\mathbf{A_o}})$$
$$\text{pconv}(\mathbf{A_i}, \nabla l_{\mathbf{A_o}}) \coloneqq \text{conv}({\mathbf{A_i}}^T, \nabla l_{\mathbf{A_o}}^T)^T$$

where $_{180}$ indicates a rotation of 180 degree across the third and fourth dimensions (w, h), and ^{T} means transposing in the first and second dimensions (i, b).

Note that for both tconv and pconv, the first two dimensions need to be transposed, which causes problems when applying BFP operations, since the exponent that was previously shared across depth is now shared across the batch /

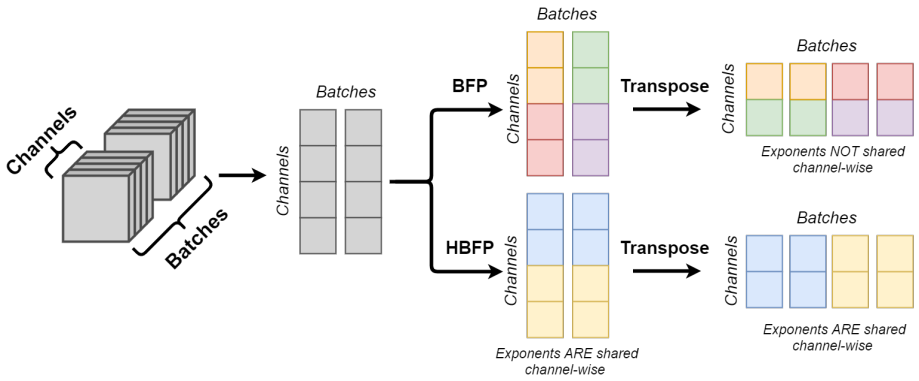


Figure 2: Difference between BFP and HyperBlock. In this example, the tensor has shape $(2, 4, 1, 1)$, such that batch$= 2$, channel$= 4$, width=height$= 1$, $blk = 2$. Same coloured cells correspond to the same shared exponent. In BFP, before transposition, cells were aligned such that MACs could be performed within blocks. After transposition, BFP does not hold this property, as numbers with shared exponent are in different batches. HBFP keeps shared exponents in the same batches after transposition, since they are shared across a 2D block of 2 by 2.

outer channel dimension, which means that MACs using integer only operations can no longer be exploited, since they do not share an exponent anymore. Figure 2 shows the problem for BFP. After transposition of channels and batches / output channels dimensions, exponents are not shared anymore, which makes it impossible to take advantage of low bitwidth MAC operations in hardware.

There are two ways to avoid this "permutation problem": Direct Gradient BFP, and HyperBlock Floating Point. We compare both in the next sections.

4.1. Direct Gradient Block Floating Point

Naively applying BFP to gradients means adding a BFP quantizer just after transposition, but before convolution. This means that just before a tensor is convolved, BFP quantizer is applied across the dimension that is going to be performed the dot-product. In the forward pass, this means applying BFP depthwise on both the weight and activation. In the backward pass, this means applying BFP across batches in the activation and gradient in order to convolve them resulting in the gradient with respect to the weight; and across output channel in the weight and across channel in the gradient to perform the convolution, resulting in the gradient with respect to activation. Figure 3a shows how this is done.

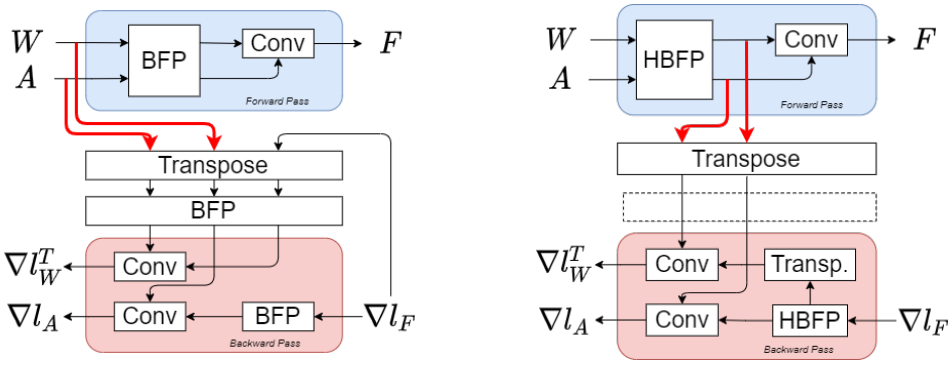
Note that because BFP is not invariant to permutation, the quantization of the tensors in the normal and in the permuted dimension need to be applied every time they are convolved. This has two implications: First, in total there is a need to quantize tensors 6 times for each convolutional layer - two in the forward pass for the weight and activation, and 4 in the backward pass for the transposed version of weight, activation, and gradient with respect to feature map; Second, since different dimensions are being quantized on the fly, all tensors need to be stored in FP32, which consequently requires considerably higher bandwidth.

Algorithm 1: Block Floating Point Quantizer

Input: Input[N, C, W, H], BIT, BLK
Result: IntT[N, C, W, H], ExpT[N, blk, W, H];

```
blk = ceil(C/BLK);
// Part 1:  Assemble ExpT
for n=0:N; w=0:W; h=0:H; b=0:blk do
    max_e ← -127;
    for c_blk=0:BLK do
        c ← c_blk + b * BLK;
        e ← get_exp(Input[n, c, w, h]);
        if e > max_e then
            max_e ← e;
        end
    end
    ExpT[n, b, w, h] ← max_e;
end
// Part 2:  Assemble IntT
for n=0:N; w=0:W; h=0:H;c=0:C; do
    e ← get_exp(Input[n, c, w, h]);
    m ← mantissa(Input[n, c, w, h]);
    s ← sign(Input[n, c, w, h]);
    shift ← (ExpT[n, floor(c/BLK), w, h] - e);
    m = (m >> shift) + U(0, 1);
    m = floor(m, BIT);
    IntT[n, c, w, h] = s && m;
end
```

Algorithm 1 details this implementation. First, for each block in the tensor, the maximum exponent is extracted. For a block size of BLK, this results in an exponent ten-



(a) Convolution Layer using BFP

(b) Convolution Layer using HBFP

Figure 3: Forward and backward passes of Direct Block Floating Point (DBFP) (a), and HyperBlock Floating Point (HBFP) (b). In DBFP, there are 6 BFP quantization operations, whereas 3 are needed in HBFP since they are invariant to transposition. As highlighted by the red arrows, in DBFP the FP32 tensors are used twice, whereas in HBFP the already quantized tensors are reused for backprop. Following previous papers [1,45], only $\nabla l_{\mathbf{A}}$ is quantized, whereas $\nabla l_{\mathbf{W}}$ is kept in FP32.

sor with sizes $(N, \text{ceil}(C/BLK), W, H)$. Therefore, there is approximately $NCWH$ multiplies in low-bit fixed point, and $NWH\text{ceil}(C/BLK)$ adds in FP32 for each tensor multiplication. Note that this can be parallelized in the outer loop. The same implementation works for transposition by changing the index of the first two dimensions. Once the exponent tensor is assembled, a shift operator can be applied to each value in the input tensor, based on which block they belong to, such that they all share the exponent dictated by the exponent tensor. Stochastic rounding [3] is then used (by adding a Uniformly sampled value as explained in Section 4.3), and the final value is stored in the integer tensor. Note that this second part is parallelized in all dimensions. This results in an Integer Tensor that performs all the multiplications, and the bulk of additions during any convolution.

4.2. HyperBlock

We propose a simpler way of avoiding having to quantize the same tensors multiple times. Instead of quantizing using blocks depthwise, 2 dimensional blocks can be used across the two dimensions that are transposed. Figure 2 shows how this is done. First, a block value is selected (say $blk = 2$). This means that for a 2 dimensional block of 2x2, an exponent is selected and shared across the entire 2D block, resulting in an exponent tensor with sizes $(\text{ceil}(N/BLK), \text{ceil}(C/BLK), W, H)$. Therefore, there are $NCWH$ fixed-point multiplies and $WH\text{ceil}(NC/BLK^2)$ adds in FP32 for each tensor multiplication. Whenever the tensor needs to transpose before convolutions, the right blocks still share the same exponents, and there is no need to quantize further.

This has many implications for the entire process, which can be seen in Figure 3b. For the forward pass, the weight and activation are quantized using HyperBFP and convolved. Note that for the backward pass, the already quan-

Algorithm 2: HyperBlock Quantizer

Input: Input[N, C, W, H], BIT, BLK
Result: IntT[N, C, W, H], ExpT[blkn, blkc, W, H];

```
blkc = ceil(C/BLK);
blkn = ceil(N/BLK);
// Part 1:  Assemble ExpT
for w=0:W; h=0:H; b_c=0:blkc; b_n=0:blkn do
    max_e ← -127;
    for c_blk=0:BLK; n_blk=0:BLK do
        c ← c_blk + b_c * BLK;
        n ← n_blk + b_n * BLK;
        e ← get_exp(Input[n, c, w, h]);
        if e > max_e then
            max_e ← e;
        end
    end
    ExpT[b_n, b_c, w, h] ← max_e;
end
// Part 2:  Assemble IntT
for n=0:N; w=0:W; h=0:H;c=0:C; do
    e ← get_exp(Input[n, c, w, h]);
    m ← mantissa(Input[n, c, w, h]);
    s ← sign(Input[n, c, w, h]);
    shift ← (ExpT[floor(n/BLK), floor(c/BLK), w,
      h] - e);
    m = (m >> shift) + U(0, 1);
    m = floor(m, BIT);
    IntT[n, c, w, h] = s && m;
end
```

tized version of the weight and the activations are used, which means that they do not need to be stored in FP32. For the backward pass, the now quantized weight and activation are transposed, together with the gradient with respect to the output map, and they are convolved as expected. This means that the number of quantization operations needed is 3 for each layer, and that once quantized, their floating point equivalent do not need to be stored anymore, considerably reducing the memory bandwidth requirement.

Algorithm 2 shows how the HyperBlock quantizer works. Similary to the BFP Quantizer, the Exponent Tensor is assembled by finding the maximum exponent across these 2 dimensional blocks. Then the exponents are use to shift each value of the input tensor, and then it is stochastically rounded to the desired bitwidth. All of these operations are massively paralellizeable, and since they only involve reading the mantissa and exponent values, and shifting, they can be efficiently implemented in custom hardware.

4.3. Exponent Selection and Stochastic Rounding

In previous papers, the exponent selection is made by sharing the largest exponent across the entire block [42]. This guarantees that the important (large) values are unaltered. We have also adopted this strategy in this paper.

Note that stochastic rounding is used in both methods, which means rounding a value x up with probability $p = \frac{x-\lfloor x \rfloor}{\lceil x \rceil - \lfloor x \rfloor}$, and down with probability $1-p$. In practice, we simply sample from a uniform distribution, sum to the value, and floor the result $x = \lfloor x + U(0,1) \rfloor$. The proof of this equivalence is provided in the supplementary material.

5. Results

We compare the two different approaches for forward and backward pass calculations: We apply DirectBFP and HyperBlock method on Cifar10, SVHN and ImageNet.

CIFAR10 We tested our method on the CIFAR10 dataset [17], which consists of 60k images in total, each with 3 colour channels, in a 32x32 pixels format. The dataset is split between 10k images for testing and 50k images for training, each labelled with 10 different classes. We trained all networks for 200 epochs, using a Stochastic Gradient Descent optimiser, with initial learning rate of 0.01, momentum of 0.9, weight decay of 5e-4, and reducing the learning rate by a factor of 10 every 60 epochs.

The results are shown in Table 1 for ResNet18, and in Table 2 for the VGG16. In both cases the bit value shown represents the bit-length that the weights, activations and gradients of all but the first convolutional layer were quantized. In both tables we can see the influence that bitlength, block size, and method choice affects the results. As expected, the lower the bitlength, the harder it is to train the models and the lower is the accuracy. It is important to note that a lower bitlength corresponds to more efficient computation, so there is a natural trade-off between computational complexity and accuracy. The block size affects the accuracy as well. The lower the block size, the better the accuracy is, but less efficient is the computation. Note however that the block size affects differently depending on the method chosen. The results show that for extremely low bitwidths (2 and 3), the Direct method loses accuracy quicker than HyperBlock as the block size increases. For example, in Table 2, there was a loss of 27% when increasing the block size

ResNet18 HyperBlock

		BLOCK 8	BLOCK 16	BLOCK 32
BIT	4	94.4% (-0.1)	94.5% (-0.0)	94.3% (-0.2)
	3	94.1% (-0.4)	93.6% (-0.9)	92.7% (-1.8)
	2	93.0% (-1.5)	89.2% (-5.3)	78.0% (-16.5)

ResNet18 DirectBFP

		BLOCK 8	BLOCK 16	BLOCK 32
BIT	4	94.1% (-0.4)	93.4% (-1.1)	93.7% (-0.8)
	3	94.0% (-0.5)	93.5% (-1.0)	80.0% (-14.5)
	2	93.3% (-1.2)	69.5% (-25.0)	66.0% (-28.5)

Table 1: Results of the HyperBlock and Direct methods applied to ResNet18 using the CIFAR10 Dataset, which achieves an accuracy of 94.5% using FP32. The number in parenthesis indicate difference in accuracy to FP32 model.

VGG16 HyperBlock

		BLOCK 8	BLOCK 16	BLOCK 32
BIT	4	93.4% (0.8)	93.2% (0.6)	93.4% (0.8)
	3	92.9% (0.3)	93.2% (0.6)	92.6% (-0.0)
	2	92.8% (0.2)	88.9% (-3.7)	83.3% (-9.3)

VGG16 DirectBFP

		BLOCK 8	BLOCK 16	BLOCK 32
BIT	4	92.9% (0.3)	92.9% (0.3)	92.5% (-0.1)
	3	92.5% (-0.1)	93.0% (0.4)	79.2% (-13.4)
	2	92.3% (-0.3)	76.2% (-16.4)	65.4% (-27.2)

Table 2: Results of the HyperBlock and Direct methods applied to VGG16 using the CIFAR10 Dataset, which achieves an accuracy of 92.6% using FP32. The number in parenthesis indicate difference in accuracy to FP32 model.

Model	W	A	G	Block	Method	Top-1
Model A_{dorefa}	2	2	2	-	-	90.0%
Model A_{ours}	**2**	**2**	**2**	**8**	**D**	**97.4%**
Model A_{ours}	2	2	2	16	D	85.7%
Model A_{ours}	4	4	4	16	H	96.9%
Model A_{ours}	4	4	4	32	H	95.8%
Model B_{dorefa}	2	2	2	-	-	91.9%
Model B_{ours}	**2**	**2**	**2**	**8**	**D**	**96.6%**
Model B_{ours}	2	2	2	16	D	86.7%
Model B_{ours}	4	4	4	16	H	96.6%
Model B_{ours}	4	4	4	32	H	96.0%
Model C_{dorefa}	2	2	2	-	-	85.6%
Model C_{ours}	**2**	**2**	**2**	**8**	**D**	**91.4%**
Model C_{ours}	2	2	2	16	D	75.7%
Model C_{ours}	4	4	4	16	H	96.3%
Model C_{ours}	4	4	4	32	H	94.4%

Table 3: Comparison between our method and the one of DoReFa-Net [49] on the SVHN dataset. The "Direct" in method indicates usage of the DirectBFP method, and the rows with "Hyper" indicates HyperBFP.

from 8 to 32 in the case of using 2 bits in the Direct method, whereas there was a loss of only 9% when using the HyperBlock method. The same happens in Table 1, where there was a loss of 15% when using HyperBlock, whereas there was a loss of 27% when the Direct method is used.

We believe this is because the weight tensor used for the forward pass is different than the weight tensor used in the backward pass in the Direct method. Therefore, it is likely that the gradient computation would be impacted. In the Hyperblock method, the same quantized weight used for the forward pass is also used in the backward pass, which means that the gradient computed is likely more representative of the right direction for the update step.

SVHN In order to compare with other methods, we also tested our algorithm on the SVHN dataset, which is similar to CIFAR10. This consists of approximately 73k digits for the regular training set, and 26k digits for the testing set. Just like DoReFa-Net [49], we also used an extra training set consisting of 531k digits. All images are labelled with one of the 10 digits, and each are cropped to 32x32 pixels with 3 channels for colours, and resized to 40x40 pixels.

We implemented DoReFa-Net's model architecture for fair comparison. Model A consists of 8 layers: seven convolutional layers with sizes 48, 64, 64, 128, 128, 128, 512, and an FC layer at the end. Models B, C and D are achieved by multiplying the channels by 0.5, 0.25, and 0.125 respectively. Table 3 shows the results. Although our aim is not to get extremely low bitwidth, we have competitive results even when using only 2 bits. For low block sizes (value of 8), our network outperforms DoReFa-Net in many instances. We achieve within 1% of accuracy using the HyperBlock method using only 4 bits and blocks of 16 to 32.

ImageNet We also trained our model on ImageNet [30]. We used the ILSVRC2012 training set, which consists of 1.2 million images, and the validation set, consisting of 50k images. They are all normalized and cropped to 224x224 pixels, with 3 colour channels before being fed to the network. We use batch of 128 and SGD with learning rate of 0.01, just as indicated in the AlexNet paper [18]. For the ResNet18 training, we used learning rate of 0.1 multiplying it by 0.1 every 30 epochs following the ResNet paper [9].

Based on the results from both SVHN and CIFAR10, we tested using block size of 8 when using bitwidth of 2, and between 16 to 32 for bitwidth of 4 or more. The results are shown in Table 4. There is no loss of accuracy when using only 4 bits for weights, activations and gradients using AlexNet, for block size of 16, and a small loss when using block size of 32. Both DirectBFP and HyperBFP achieved good results when using block size of 32 using only 4 bits. There is an accuracy loss of 8% when using only 2 bits with block size of 8. This is encouraging as even though our method is not meant to provide good accuracy in extreme low bitwidth, we are still competitive with DoReFa-Net.

ResNet18 is a more challenging network to train. We get competitive results by achieving less than 1% accuracy loss for 8 and 5 bits. In comparison to SWALP, this is achieved without the need to perform averaging of the weights. It also avoids the need for tanh computations needed in DoReFa-Net. The accuracy degrades when going to lower bits, and lower block sizes need to be used. We also compared to ALQ/AMQ [7] and TRN [39], even though they quantize only the gradients. We include these results here since they provide an estimate of an "upper bound" on accuracy for quantized gradients. We can see that a drop in accuracy is expected when using low precision gradients regardless of the precision used in weights and activations.

Role of Stochastic Rounding We use stochastic rounding for both BFP and HBFP. Without stochastic rounding, our method's accuracy drops on 5-bit ResNet18 from 68.3% to 40.7%, a loss of 27.6%. We believe that this is so effective because it avoids the problem of *stagnation*, where gradients are rounded down to zero, which stagnates the weight values in suboptimal values. More details about our conclusion is provided in the supplementary material.

5.1. Hardware Comparisons

To show the advantages of being able to perform training and inference by using our method, we created (gate

Algorithm	W | A | G	Block	Method	Model	Top1
DoReFa [49]	1 | 2 | 6	-	-	AlexNet	46.1%
Ours	2 | 2 | 2	8	H	AlexNet	48.8%
Ours	**4 | 4 | 4**	**16**	**H**	**AlexNet**	**56.2%**
Ours	4 | 4 | 4	32	H	AlexNet	55.6%
Ours	4 | 4 | 4	32	D	AlexNet	55.4%
DoReFa [49]	8 | 8 | 8	-	-	AlexNet	53.0%
FP32	32	-	-	AlexNet	55.9%
Ours	**8 | 8 | 8**	**32**	**D**	**ResNet18**	**69.2%**
Ours	5 | 5 | 5	32	D	ResNet18	68.3%
ALQ [7]	32 | 32 | 3	-	-	ResNet18	67.7%
SWALP [42]	8 | 8 | 8	-	-	ResNet18	65.1%
AMQ [7]	32 | 32 | 3	-	-	ResNet18	64.8%
TRN [39]	32 | 32 | 2	-	-	ResNet18	62.8%
Ours	8 | 8 | 8	64	D	ResNet18	61.5%
Ours	4 | 4 | 4	32	D	ResNet18	57.1%
FP32	32	-	-	ResNet18	69.7%

Table 4: Results of running AlexNet and ResNet18 on ImageNet using different methods. We can achieve FP32 results using only 4 bits for weights, activations and gradients.

level VHDL [13]) a wide range of BFP operations (used in both of our methods) and floating point arithmetic operators, and synthesized these to a current production 10nm FinFET library with a target frequency of 800MHz. We also compared a subset of results to Synopsys DesignWare [33] components synthesized with the same libraries and scripts; both methods were always within 10% area. We report area in Table 5: the "Mantissa Multiplier" corresponds to the area cost of multiplying the mantissa portion of the formats; the "FP Multiplier" includes also the operations needed to include the exponent part of the floating point formats; the "FP ALU" is the cost of accumulating the result of the multiplications in the block; the "Cost" is the sum of the "FP Multiplier" and the "FP ALU". 'Relative Cost" is the normalized "Cost" to the smallest operation. To more accurately model the area of these operators, we built a system level construct for each, with pipeline registers before and after the floating point operators. Additionally, FP ALUs also had wrapped multiplexers to model the cost of the dynamic selection of addition and accumulation operations. The BFP functions were implemented as tensors of 10 element dot products, again bounded by registers [21]; the cost of the fixed to floating point conversions is included in the table, amortised over the tensor per multiplier.

From Table 5, it can be seen that the BFP formats are by far the smallest. For INT8, we use 10x less area than FP32,

	Mantissa Multiplier	FP Multiplier	FP ALU	Cost	Relative Cost
FP32	156[1]	186[1]	150 [1]	336	37
TF32 [29]	32 [1]	52[2]	90	142	16
Bfloat16 [38]	19 [1]	40 [2]	150	190	21
FP16 [12]	32 [1]	40 [2]	70	110	12
FP8 [37]	6 [1]	20 [2]	30[2]	50	5.6
INT8 BFP	15 [1]	15	10	25	2.8
INT4 BFP	4 [1]	4	5	9	1

[1] Run on Synopsys DC [2] Estimated

Table 5: Comparison of cost in different number formats.

and we are also at least 2x more efficient than the common formats used for training, such as TF32 [29], BFloat16 [38], and FP16 [12]. In terms of relative cost, we are also 2x more efficient than the emerging FP8 format [37]. The difference is still larger when using INT4. Saving half of the area means that we can pack twice the number of operations, which corresponds to approximately twice the throughput.

The importance of area goes beyond saving silicon, power, and increased throughput. Memory accesses are very expensive [10], with 1 or 2 orders of magnitude for local embedded SRAM (based on analysis of cache memory at 45nm), and 3 or 4 orders of magnitude for DRAM access. HyperBFP has the potential of getting further benefits from memory accesses when comparing to DirectBFP for example, since the same quantized tensor can be used in both forward pass and backward passes, without the need to retrieve expensive FP32 values as is the case when using DirectBFP. The more processing we can perform in the arithmetic datapath, the more efficient the system will be.

6. Conclusion

We have demonstrated that DirectBFP and its HyperBlock variant can be used for quantizing gradients during training of convolutional neural networks. We have achieved virtually no loss of accuracy when training AlexNet on the challenging ImageNet dataset, by using multiplications in fixed-point, and a smaller number of addition using floating point format. We believe that this opens up space for efficient implementation in custom hardware of low bit training and inference.

7. Acknowledgements

This research was partially supported by Intel (previously Omnitek). We thank particularly the Intel FPGA design services team for their contribution to this research.

References

[1] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. *arXiv preprint arXiv:1805.11046*, 2018.

[2] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. Deep learning with low precision by half-wave gaussian quantization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5918–5926, 2017.

[3] Matteo Croci, Massimiliano Fasi, Nicholas J Higham, Théo Mary, and Mantas Mikaitis. Stochastic rounding: Implementation, error analysis, and applications. *Royal Society Open Science*, March 2022.

[4] Bita Darvish Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, et al. Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point. *Advances in Neural Information Processing Systems*, 33, 2020.

[5] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, et al. Mixed precision training of convolutional neural networks using integer operations. *arXiv preprint arXiv:1802.00930*, 2018.

[6] Mario Drumond, Tao Lin, Martin Jaggi, and Babak Falsafi. Training dnns with hybrid block floating point. *arXiv preprint arXiv:1804.01526*, 2018.

[7] Fartash Faghri, Iman Tabrizian, Ilia Markov, Dan Alistarh, Daniel M Roy, and Ali Ramezani-Kebrya. Adaptive gradient quantization for data-parallel sgd. *Advances in neural information processing systems*, 33:3174–3185, 2020.

[8] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 1–14. IEEE Press, 2018.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

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

[11] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[12] IEEE. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[13] IEEE. Ieee standard vhdl language reference manual - redline. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002) - Redline*, pages 1–620, 2009.

[14] Andrey Ignatov, Radu Timofte, Andrei Kulik, Seungsoo Yang, Ke Wang, Felix Baum, Max Wu, Lirong Xu, and Luc Van Gool. AI benchmark: All about deep learning on smartphones in 2019. *CoRR*, abs/1910.06663, 2019.

[15] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[16] Jangho Kim, KiYoon Yoo, and Nojun Kwak. Position-based scaled gradient for model quantization and pruning. *Advances in Neural Information Processing Systems*, 33:20415–20426, 2020.

[17] Alex Krizhevsky et al. Learning multiple layers of features from tiny images, 2009.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[19] Habana Labs. Gaudi training platform white paper. Technical report, Habana Labs, November 2020. Accessed on 20/02/2022.

[20] Haban Labs. Goya inference platform white paper. Technical report, Habana Labs, November 2020. Accessed on 20/02/2022.

[21] Martin Langhammer, Eriko Nurvitadhi, Bogdan Pasca, and Sergey Gribok. Stratix 10 nx architecture and applications. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 57–67, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Junghyup Lee, Dohyung Kim, and Bumsub Ham. Network quantization with element-wise gradient scaling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6448–6457, 2021.

[23] Yuhang Li, Ruihao Gong, Xu Tan, Yang Yang, Peng Hu, Qi Zhang, Fengwei Yu, Wei Wang, and Shi Gu. Brecq: Pushing the limit of post-training quantization by block reconstruction. *arXiv preprint arXiv:2102.05426*, 2021.

[24] Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[25] Yuang Liu, Wei Zhang, and Jun Wang. Zero-shot adversarial quantization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1512–1521, 2021.

[26] Markus Nagel, Rana Ali Amjad, Mart Van Baalen, Christos Louizos, and Tijmen Blankevoort. Up or down? adaptive rounding for post-training quantization. In *International Conference on Machine Learning*, pages 7197–7206. PMLR, 2020.

[27] Marcelo Gennari do Nascimento, Roger Fawcett, and Victor Adrian Prisacariu. Dsconv: Efficient convolution operator. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[28] M. G. D. Nascimento, V. Prisacariu, and R. Fawcett. Dsconv: Efficient convolution operator. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 5147–5156, 2019.

[29] NVIDIA. Training neural networks with tensor cores NVIDIA. https://nvlabs.github.io/eccv2020-mixed-precision-tutorial/files/dusan_stosic-training-neural-networks-with-tensor-cores.pdf. Accessed: 2020-03-08.

[30] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[31] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[32] Zhourui Song, Zhenyu Liu, and Dongsheng Wang. Computation error analysis of block floating point arithmetic oriented convolution neural network accelerator design. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[33] Synopsis. Synopsis designware. https://www.synopsys.com/simpleware.html. Accessed: 2022-22-08.

[34] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. Deepface: Closing the gap to human-level performance in face verification. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, 2014.

[35] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[36] Neil C Thompson, Kristjan Greenewald, Keeheon Lee, and Gabriel F Manso. The computational limits of deep learning. *arXiv preprint arXiv:2007.05558*, 2020.

[37] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. *Advances in neural information processing systems*, 31, 2018.

[38] Shibo Wang and Pankaj Kanware. Bfloat16: The secret to high performance on cloud tpus. https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus. Accessed: 2020-03-08.

[39] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. *Advances in neural information processing systems*, 30, 2017.

[40] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10734–10742, 2019.

[41] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks. *arXiv preprint arXiv:1802.04680*, 2018.

[42] Guandao Yang, Tianyi Zhang, Polina Kirichenko, Junwen Bai, Andrew Gordon Wilson, and Chris De Sa. Swalp: Stochastic weight averaging in low precision training. In *International Conference on Machine Learning*, pages 7015–7024. PMLR, 2019.

[43] Yukuan Yang, Lei Deng, Shuang Wu, Tianyi Yan, Yuan Xie, and Guoqi Li. Training high-performance and large-scale deep neural networks with full 8-bit integers. *Neural Networks*, 125:70–82, 2020.

[44] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[45] Xishan Zhang, Shaoli Liu, Rui Zhang, Chang Liu, Di Huang, Shiyi Zhou, Jiaming Guo, Qi Guo, Zidong Du, Tian Zhi, et al. Fixed-point back-propagation training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2330–2338, 2020.

[46] Xiangguo Zhang, Haotong Qin, Yifu Ding, Ruihao Gong, Qinghua Yan, Renshuai Tao, Yuhang Li, Fengwei Yu, and Xianglong Liu. Diversifying sample generation for accurate data-free quantization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15658–15667, 2021.

[47] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856, 2018.

[48] Kang Zhao, Sida Huang, Pan Pan, Yinghan Li, Yingya Zhang, Zhenyu Gu, and Yinghui Xu. Distribution adaptive int8 quantization for training cnns. In *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence*, 2021.

[49] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.