NITI: Training Integer Neural Networks Using Integer-only Arithmetic

Maolin Wang
The University of Hong Kong
mlwang@eee.hku.hk

Seyedramin Rasoulinezhad
The University of Sydney
seyedramin.rasoulinezhad@sydney.edu.au

Philip H.W. Leong
The University of Sydney
philip.leong@sydney.edu.au

Hayden K.H. So
The University of Hong Kong
hso@eee.hku.hk

Abstract

While integer arithmetic has been widely adopted for improved performance in deep quantized neural network inference, training remains a task primarily executed using floating point arithmetic. This is because both high dynamic range and numerical accuracy are central to the success of most modern training algorithms. However, due to its potential for computational, storage and energy advantages in hardware accelerators, neural network training methods that can be implemented with low precision integer-only arithmetic remains an active research challenge. In this paper, we present NITI, an efficient deep neural network training framework[1] that stores all parameters and intermediate values as integers, and computes exclusively with integer arithmetic. A pseudo stochastic rounding scheme that eliminates the need for external random number generation is proposed to facilitate conversion from wider intermediate results to low precision storage. Furthermore, a cross-entropy loss backpropagation scheme computed with integer-only arithmetic is proposed. A proof-of-concept open-source software implementation of NITI that utilizes native 8-bit integer operations in modern GPUs to achieve end-to-end training is presented. When compared with an equivalent training setup implemented with floating point storage and arithmetic, NITI achieves negligible accuracy degradation on the MNIST and CIFAR10 datasets using 8-bit integer storage and computation. On ImageNet, 16-bit integers are needed for weight accumulation with an 8-bit datapath. This achieves training results comparable to all-floating-point implementations.

1 Introduction

Training of deep neural networks (DNNs) is a lengthy and computationally demanding process that requires a notoriously large number of floating-point and memory operations, forming a substantial barrier for rapid deployment and development of new applications and models. While the use of graphics processing units (GPUs) have been invaluable in addressing this computational need, there has been a renewed interest in accelerating DNN training with hardware accelerators that employ low-precision integer arithmetic due to their promised power and energy efficiency advantages. Compared to computing with single-precision floating-point operations, computing with integer arithmetic, especially when low-precision datatype such as `int8` is used, significantly reduces the time and silicon area requirement of a system, making them attractive for power and energy-constrained

[1] https://github.com/wangmaolin/niti

Preprint. Under review.

edge deployment, as well as in cost-sensitive cloud datacenters. Training DNNs with integer-only arithmetic is particularly challenging because of their lack of dynamic range and precision that are often needed throughout the training process. Even state-of-the-art integer neural network training frameworks [7, 1] still rely on floating-point arithmetic for at least some portion of the process, such as for intermediate parameter storage and update, as well as for gradient, error and activation computation during backpropagation.

In this work, we present NITI, a deep neural network training framework that operates *exclusively* with integers. To address the unique challenges of training DNN with integers, NITI has three main contributions:

1. a novel *discrete* parameter update scheme that allows low precision integer storage of all intermediate variables;
2. a hardware-efficient pseudo stochastic rounding scheme that utilizes the extra precision in intermediate accumulation as an in-situ random number source;
3. an efficient approximation of cross-entropy loss backpropagation using integer-only arithmetic.

While the long-term goal of NITI is to accelerate DNN training with dedicated hardware accelerators, the focus of this work is to study the underlying numerical properties of DNN training and to develop a family of integer-only DNN training algorithms that can achieve comparable accuracy to their floating-point counterparts. For that, an open-source proof-of-concept software implementation of NITI is presented here. Particular attention has been paid in this software implementation to ensure bit-accurate integer operations are performed in CPUs as well as in GPUs that include native integer matrix-multiplication support.

Using our software implementation as demonstrations, we show that NITI can achieve negligible accuracy degradation on the MNIST dataset by using only 8-bit integer (`int8`) operations. On the CIFAR-10 dataset using VGG-13, also with `int8`, NITI was able to achieve $88.24\,\%$ best-1 validation accuracy when compared with $91.04\,\%$ achieved by an equivalent floating-point implementation. On the ImageNet dataset using the original AlexNet, by using 16-bit integers for weight accumulation, NITI was able to achieve $43.66\,\%$ top-1 compared to $45.16\,\%$ with an equivalent floating point implementation.

In the next section, we summarize related work in integer training. The design of NITI will be presented in Section 3, followed by experimental results in Section 4. We will conclude and discuss future enhancements to NITI in Section 5.

2 Related Works

The study of hardware-efficient neural network training can be traced to research in low-precision neural network inference, which is as a complementary problem. Early studies of binary and ternary neural networks have already demonstrated the feasibility of using hardware-efficient bit-operations to replace complex floating-point multiply-accumulate (MAC) operations during training [3, 14, 18]. Moreover, researchers have also argued that such weight quantization during training could serve as a regularizer that improved the training performance on small datasets [3]. Subsequently, Jacob *et al.* extended quantization to 8 bit weights and activations and demonstrated promising results for deploying DNN models on mobile devices [8]. With a large-scale hardware accelerator, Gupta *et al.* [6] further showed that DNNs could be trained with fixed-point weights using 16-bit precision, and that rounding was crucial to success. They proposed *stochastic rounding* where the probability of rounding x to $\lfloor x \rfloor$ is proportional to their proximity, which has formed the basis of many modern integer training frameworks.

Beyond weight and activations, recent works have begun to address the challenges of quantizing gradient and error computation during the backward pass of training. In the work of DoReFa-Net [17], weight, activation as well as gradients of activations were quantized to allow discretized computation during back propagation. Similarly, Banners *et al.* developed a bifurcation scheme that quantized gradients of activations during training for 8-bit integer operations in mobile processors [1].

Although promising results have been demonstrated in all the above cases, they have all relied on the use of full-precision floating-point computation in at least some parts of the training process. In [18, 14, 3, 8, 17, 1], `fp32` were used as intermediate storage for weight accumulation, while

Table 1: Datatype comparison for various low-precision integer DNN training frameworks

	$\boldsymbol{w}(infer)$	$\boldsymbol{w}(acc)$	$\boldsymbol{a}$	$\boldsymbol{g}$	$\boldsymbol{e}$	softmax
TTQ[18]	2	32	32	32	32	`fp32`
Xnor [14]	1	32	1(32)	32	32	`fp32`
Binaryconnect[3]	1	32	32	32	32	`fp32`
Jacob *et al.* [8]	8	32	8(32)	32	32	`fp32`
Dorefa[17]	1	32	2(32)	32	6(32)	`fp32`
Banner *et al.* [1]	8	32	8(32)	32	8(32)	`fp32`
WAGE[16]	2	8(32)	8(32)	8(32)	8(32)	`fp32`
FxpNet[2]	1	12(32)	1(32)	12(32)	12(32)	`fp32`
NITI (*this work*)	8	8	8	8	8	integer

(32) means the low precision data format is emulated by quantizing `fp32` number.

Algorithm 1: Forward and backward passes in NITI for each batch of data X and label Y

```
/* Forward Pass, with quantized input $\boldsymbol{a}^{(0)}, s_{a^{(0)}}$ from $X$ */
for each layer $l$ do
    $\boldsymbol{a}_{32}^{(l)},\ s_{a^{(l)}} \leftarrow$ INT8MATRIXMULTIPLY$(\boldsymbol{a}^{(l-1)}, \boldsymbol{w}),\ s_{a^{(l-1)}} + s_w$;
    $b \leftarrow$EFFECTIVEBITWIDTH$(\boldsymbol{a}_{32}^{(l)})$;
    $\boldsymbol{a}^{(l)}, s_{a^{(l)}} \leftarrow$ SHIFTANDROUND$(\boldsymbol{a}_{32}^{(l)}, s_{a^{(l)}}, \max(0, b-7))$;
end
/* Backward Pass, with $\boldsymbol{a}, s_a$ being final layer's output, scale */
$\boldsymbol{e} \leftarrow$ INT8LOSSGRADIENT$(\boldsymbol{a}, s_a, Y)$;
for each layer $l$ in reverse order do
    $\boldsymbol{g}_{32}^{(l)} \leftarrow$ INT8MATRXMULTIPLY$(\boldsymbol{a}^{(l-1)}, \boldsymbol{e}^{(l)})$;
    $\boldsymbol{e}_{32}^{(l-1)} \leftarrow$ INT8MATRXMULTIPLY$(\boldsymbol{w}^{(l)}, \boldsymbol{e}^{(l)})$;
    $b \leftarrow$EFFECTIVEBITWIDTH$(\boldsymbol{e}^{(l-1)})$;
    $\boldsymbol{e}^{(l-1)}, \_ \leftarrow$ SHIFTANDROUND$(\boldsymbol{e}_{32}^{(l-1)}, \_, \max(0, b-7))$;
    $b \leftarrow$ EFFECTIVEBITWIDTH$(\boldsymbol{g}_{32}^{(l)})$;
    $\boldsymbol{g}^{(l)}, \_ \leftarrow$ SHIFTANDROUND$(\boldsymbol{g}_{32}^{(l)}, \_, \max(0, b-m_u))$;
    $\boldsymbol{w}^{(l)} \leftarrow \boldsymbol{w}^{(l)} - \boldsymbol{g}^{(l)}$ ;
end
```

in [16, 2] they were used as a proxy for the low-precision datatype for computation. For efficient hardware implementations, a training scheme that employs integer arithmetic exclusively such as this proposed work is highly desirable.

The closest works to NITI that we can identify are the work of FxpNet [2] and WAGE [16]. In FxpNet, 12-bit fixed point arithmetic was employed throughout the training process to produce a binary network for inference. In WAGE, weight, activations, gradients and error values were all quantized to 8-bit integers to train a ternary network for inference. For softmax computation, however, `fp32` were still needed in both cases in order to ensure accurate inference on large dataset. In contrast, NITI employs integer arithmetic exclusively, even when used in training large networks for classifying ImageNet dataset with 1000 classes using softmax and have achieved only moderate accuracy degradation. A summary of the datatype employed in related works is shown in Table 1.

3 An Integer-only Training Framework

NITI operates exclusively on integer arithmetic and is based on the stochastic gradient descent (SGD) with backpropagation (BP) scheme (Algorithm 1). The major differences between the two rest on the use of integer arithmetic throughout, and the discrete weight update scheme that is closely tied to the process of rounding intermediate values back to `int8` throughout the algorithm.

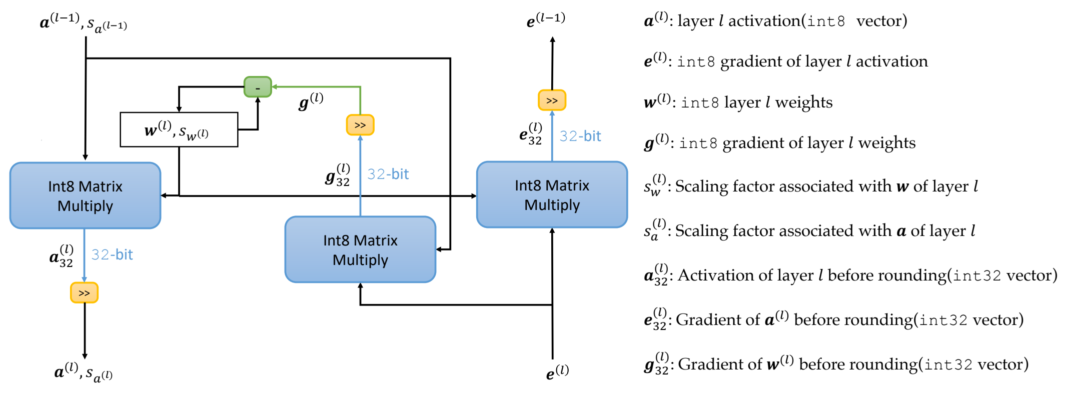

$a^{(l-1)}, s_{a^{(l-1)}}$
$w^{(l)}, s_{w^{(l)}}$
-
$g^{(l)}$
>>
$g_{32}^{(l)}$
32-bit
Int8 Matrix Multiply
Int8 Matrix Multiply
Int8 Matrix Multiply
$a_{32}^{(l)}$
32-bit
>>
$a^{(l)}, s_{a^{(l)}}$
$e^{(l-1)}$
>>
$e_{32}^{(l)}$
32-bit
$e^{(l)}$
$a^{(l)}$: layer l activation(int8 vector)
$e^{(l)}$: int8 gradient of layer l activation
$w^{(l)}$: int8 layer l weights
$g^{(l)}$: int8 gradient of layer l weights
$s_w^{(l)}$: Scaling factor associated with w of layer l
$s_a^{(l)}$: Scaling factor associated with a of layer l
$a_{32}^{(l)}$: Activation of layer l before rounding(int32 vector)
$e_{32}^{(l)}$: Gradient of $a^{(l)}$ before rounding(int32 vector)
$g_{32}^{(l)}$: Gradient of $w^{(l)}$ before rounding(int32 vector)

Figure 1: Integer layer forward and backward pass

Figure 1 shows an overview of the symbols and their datatype employed throughout the algorithm. As shown in the figure, all the model's weight values ($\boldsymbol{w}$), activation ($\boldsymbol{a}$), gradients of activations ($\boldsymbol{e}$) and gradient of weights ($\boldsymbol{g}$), are stored as 8-bit signed integers (`int8`). In addition, similar to the dynamic floating point scheme proposed in [4], weights of the model are paired with a per-layer `int8` exponent (s) such that the actual values are $\boldsymbol{w} \cdot 2^{s_w}$. Apart from quantizing standard floating point (`fp32`) data input during initialization, no further quantization is performed in NITI.

3.1 Forward and Backward Pass

At the core of NITI are the forward pass and backward pass of convolution and fully connected layers performed with integer-only arithmetic. For convolution layers, our framework employed `im2col` to reduce convolution operations into matrix-matrix multiplications [9]. As a result, similar to the case of training fully connected layers, integer matrix multiplies form the dominating operation in our training framework. With our use of `int8` as input, we were able to accelerate this matrix multiply in our current implementation by leveraging the newly introduced integer matrix multiply function unit in latest GPUs that were originally designed for inference acceleration [13].

During the forward pass (left side of Figure 1), activation of previous layer ($\boldsymbol{a}^{(l-1)}$) enters this layer as `int8` with a scaling factor $s_{a^{(l-1)}}$. The results of the matrix multiply are accumulated in `int32` precision and must be rounded back to `int8` before propagating to the next layer. This rounding operation is very important to the success of low precision training and repeatedly appears in forward pass, backward pass and model weights update. They are shown as shift operators in Figure 1 as they are combined with the scaling operation in our scheme.

In theory, the output scaling factor $s_{a^{(l)}}$ is simply a sum of the 2 input scaling factors $s_{a^{(l-1)}}$ and s_w. However to make full use of the signed 8-bit precision to represent integer part of the results, an additional shift operation is performed based on the *effective bitwidth* of the 32-bit matrix multiply output. Here, effective bitwidth of an integer matrix $\boldsymbol{V}$, denoted as $B(\boldsymbol{V})$, is defined as the minimum number of bits required to fully represent the maximum value $v \in \boldsymbol{V}$. If $b = B(\boldsymbol{V}) > 7$, then the 32-bit results are shifted right by $b - 7$ bit, just enough to maximize the use of the `int8` datatype. The fractional part is rounded off using the pseudo stochastic rounding scheme described in Section 3.3 and the scaling factor is adjusted accordingly.

The backward pass involves propagating the error back through the integer network and computing the gradient of weights in each layer for weight update as shown in right hand side of Figure 1. The 32-bit errors are rounded similarly to the forward pass back into 8-bit values before being propagated to the next layer. The 32-bit gradients are rounded with special procedures that fuses with the weight update process as explained in Section 3.2.

3.2 Weight Update

Normally with SGD, the gradient $\boldsymbol{g}^{(l)}$ of layer weights can be computed by a matrix multiplication between its activation input $\boldsymbol{a}^{(l-1)}$ and gradients of its activation output $\boldsymbol{e}^{(l)}$. The model weights could subsequently be updated by some combinations of this gradient $\boldsymbol{g}$, the global learning rate and other heuristics to determine the amount of weight update.

However, due to the limited range of our `int8` weights $\boldsymbol{w}$, the task of maintaining any necessary precision is challenging, notably due to the range discrepancy between $\boldsymbol{w}$ and the desired update value. For example, we observed empirically that even in a medium size network like VGG11, direct application of typical learning rules often resulted in overflow or underflow in the weight update. Consequently, most of the update values are saturated as ± 127 or 0.

Instead, we propose a learning heuristic that combines update with the rounding of gradient for weight computation in `int8`.

In particular, let $b = B\left(\boldsymbol{g}_{32}^{(l)}\right)$, then $\boldsymbol{g}_{32}^{(l)}$ is shifted and rounded by $b - m_u$ bits to obtain an effectively m_u bits weight update value $\boldsymbol{g}^{(l)}$. Recall that the value of s_w for each layer is set during initialization and remain unchanged during training. As a result, the proposed update heuristic essentially operates by updating $\boldsymbol{w}$ with small quantum $\boldsymbol{g}^{(l)} \cdot 2^{s_w}$. We have evaluated different values of m_u and have determined that small values of m_u in the range of 1 to 3 bits performed well in general. See Section 4.2 for experimental results. We show empirically that this learning rule has comparable convergence speed on MNIST and CIFAR10 to SGD with momentum, which is commonly used in floating point training. On ImageNet, this learning rule has comparable convergence speed with SGD without momentum.

3.3 Shifting and Rounding

Mapping `int32` results from matrix-multiply back to `int8` data for downstream computation is a crucial step that has significant influence in the final training accuracy. To propagate activations and errors during training, a special *shift and round* scheme is employed. Using this scheme, the values in concern (i.e. $\boldsymbol{a}$ and $\boldsymbol{e}$) are first logically shifted right by an amount that is determined by their effective bitwidth $B(\boldsymbol{a})$ and $B(\boldsymbol{e})$ respectively (See lines 4 and 11 in Algorithm 1). This shift avoids overflow and maximize the number of useful bits in the final `int8` representation. In addition, the corresponding activation scaling factor s_a is adjusted according to maintain correct magnitude.

Next the `int32` values are rounded to `int8`. In the context of NITI, rounding refers to the task of mapping a 32-bit *fixed point* number $x = \langle q.f \rangle$ to a nearby 8-bit integer $\tilde{x}$. In this notation, q and f are the integer and fraction parts of x respectively, and the binary point is located at position bp, which is equal to the bitwidth of f.

As discussed in [6], the use of stochastic rounding with zero rounding bias is crucial to the success of training neural network with low precision. Stochastic rounding $\text{RS}(\cdot)$ can be defined as:

$$\text{RS}(\langle q.f \rangle) = \begin{cases} q & \text{with probability } 1 - f \cdot 2^{-bp} \\ q+1 & \text{with probability } f \cdot 2^{-bp} \end{cases}$$

A typical implementation of the stochastic rounding function would therefore compute the rounding probability by comparing f with a random number r.

Instead of relying on an external random number source to produce r, we propose a hardware-efficient *pseudo stochastic rounding* algorithm that generates in-situ pseudo random number using the additional bits from the 32-bit input. As shown in Algorithm 2, the proposed scheme operates in ways similar to the original stochastic rounding function, except the stochastic rounding decision is now based on comparing values between different portions of f, the fractional parts of the input. Our current implementation divides the fractional bits into 2 halves for comparison. The more significant (top) half of f is essentially a truncated version of f, and the less significant (bottom) half of f now serves as a pseudo random number. As shown in Section 4.1, the proposed pseudo stochastic rounding scheme is able to support training with comparable accuracy as the original stochastic rounding scheme.

Algorithm 2: Pseudo Stochastic Rounding

```
Input  : $q_{32}$: 32-bit fixed point number, $bp$: Binary point
Output : Rounded 8-bit integer $q$
/* Extract bits for integer ($q$) and fractional ($f$) parts from $|q_{32}|$   */
$q,\ f \leftarrow |q_{32}|[bp+7, bp],\ |q_{32}|[bp-1, 0]$;
if $bp$ is odd then
    $f \leftarrow f \gg 1$ /* $\gg$ 1 represents right shift by 1 bit   */
    $bp \leftarrow bp - 1$;
end
if $f[bp-1, \frac{bp}{2}] > f[\frac{bp}{2} - 1, 0]$ then
    $q \leftarrow q + 1$
end
$q \leftarrow q \cdot \mathrm{sgn}(q_{32})$
```

3.4 Integer Cross Entropy Loss

After computing the final layer activations, $\boldsymbol{a} \cdot 2^{s_a}$, NITI computes the predicted probability of each class for the input image using a softmax layer. Assume we have N classes with $i \in \{1 \dots N\}$, then $\hat{y_i} = \frac{e^{a_i \cdot 2^{s_a}}}{C}$ is the predicted probability of class i, where $C = \sum_{i=1}^{N} e^{a_i \cdot 2^{s_a}}$.

With the predicted probability $\hat{\boldsymbol{y}}$ and the target one hot probability $\boldsymbol{y}$ obtained from the labels, the backpropagation process can be initiated by computing the gradient $\boldsymbol{e}$ of cross entropy loss, E. Cross-entropy loss is defined as $E = -\sum_{i=1}^{N} y_i \ln(\hat{y_i})$, and its partial derivative can simply be computed as:

$$e_i = \frac{\partial E}{\partial(a_i \cdot 2^{s_a})} = \hat{y_i} - y_i = \frac{e^{a_i \cdot 2^{s_a}} - y_i C}{C} \qquad i \in \{1 \dots N\} \tag{1}$$

The main challenge of implementing Equation (1) with integer arithmetic is therefore to approximate the term $t_i = e^{a_i \cdot 2^{s_a}}$ accurately with limited precision. We address this challenge by considering 2 cases.

When $s_a \leq -7$, we know $|a_i \cdot 2^{s_a}| < 1$ because, as an 8-bit integer, $a_i \leq 127$. We therefore approximate t_i via its Taylor expansion:

$$t_i = e^{a_i \cdot 2^{s_a}} \approx 1 + a_i \cdot 2^{s_a} + \frac{1}{2} a_i^2 \cdot 2^{2s^a} \tag{2}$$

When $s_a > -7$, we rearrange the term with a base-2 approximation as follows:

$$t_i = e^{a_i \cdot 2^{s_a}} = 2^{(\log_2 e) \cdot a_i \cdot 2^{s_a}} \approx 2^{(47274 \cdot 2^{-15}) \cdot a_i \cdot 2^{s_a}} \tag{3}$$

Denote the exponent as $x_i = 47274 \cdot 2^{-15} \cdot a_i \cdot 2^{s_a}$. Note that the number 47 274 = `0xB8A8` requires 16 bits to represent, while a_i is a 8-bit integer. We observe empirically that $s_a < 0$ in most cases. Therefore, x_i can be computed by shifting the value $47274 a_i$ by $s_a - 15$ followed by truncation of any resulting fractional bits. Subsequently, for all x_i, let p be the smallest value of x_i larger than $\max(\boldsymbol{x}) - 10$. Then define $\hat{x}_i = \max(0, x_i - p)$. Finally, $t_i = 2^{x_i}$ is approximated as $2^{\hat{x}_i}$, which can be computed by $1 \ll \hat{x}_i$. The error tensor $\boldsymbol{e}$ in (1) is computed using these effectively 12-bit values and eventually rounded stochastically back to 8 bits before being used in back propagation. For dataset with 1000 classes like ImageNet, stochastic rounding is necessary to avoid round bias in e_i. Despite the crude approximation in some cases, we find the accuracy of the resulting network competitive to related works that depend on floating-point implementations.

4 Experimental Results

In this section, we first evaluate the key factors which influence the performance of trained networks by NITI framework including the different rounding schemes for $\boldsymbol{a}, \boldsymbol{e}, \boldsymbol{g}$ and the bit width of $\boldsymbol{g}$

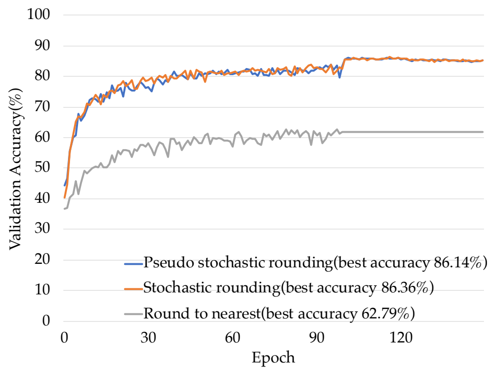


Figure 2: Validation accuracy for different rounding schemes

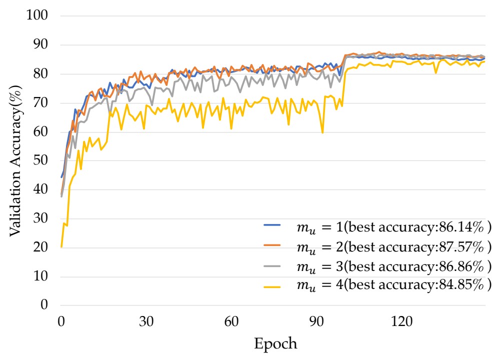
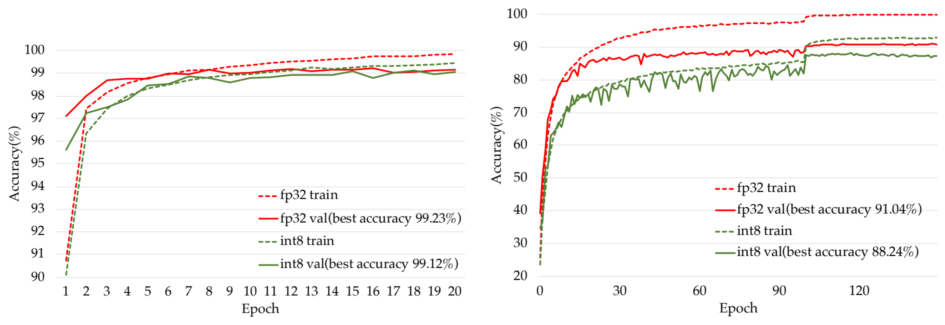


Figure 3: Validation Accuracy for different values of m_u



(a) MNIST (LeNet) (b) CIFAR10 (VGG13)

Figure 4: Training performance comparing NITI (`int8`) and baseline `fp32` implementation.

used for parameter updating. To do so, we used CIFAR10 [10] datasets with the VGG family of network architectures without batch normalization layer [15]. Learning from the above evaluation, we optimized our NITI framework using the proposed rounding schemes and the best configuration for m_u parameter. Then, we compare the performance of NITI framework with its floating point counterpart trained by SGD optimizer over MNIST [11], CIFAR10, and ImageNet [5] datasets.

4.1 Rounding Scheme

As stochastic rounding is adopted by most recent low precision training implementations [1, 16, 2], we use it as the baseline for evaluating our pseudo stochastic rounding scheme. We also include results from straight-forward round-to-nearest for comparison. m_u is fixed as 1 for all three different rounding schemes. Figure 2 shows the validation accuracy for training `int8` VGG11 on CIFAR10 dataset with different rounding schemes. Our pseudo stochastic rounding technique works as well as baseline stochastic rounding while both significantly outperform round-to-nearest method in terms of final best validation accuracy.

4.2 Learning Heuristic with Different Values of m_u

To find the best practice of m_u in our learning heuristic, we have explored different m_u to obtain the best validation accuracy. The best validation accuracy for different m_u while training VGG11 network over CIFAR10 dataset are shown in Figure 3. Generally speaking, small values of m_u work better than larger values. This is as expected because the weight update process updates with multiple of 2^{s_w}. Large values of m_u will thus more likely to diverge from the originally computed update values using `fp32` than smaller values.

Table 2: Validation accuracy for different models on CIFAR10

Model	VGG11 int8	VGG11 fp32	VGG13 int8	VGG13 fp32	VGG16 int8	VGG16 fp32
Accuracy	87.57%	89.58%	88.24%	91.04%	87.94%	91.41%

Table 3: Comparison to related work in training AlexNet with ImageNet dataset using integers.

	$\boldsymbol{w}(infer)$	$\boldsymbol{w}(acc)$	$\boldsymbol{a}$	$\boldsymbol{g}$	$\boldsymbol{e}$	loss	Accuracy(%)
`fp32` baseline	32	32	32	32	32	`fp32`	45.16
Jacob *et al.* [8]	8(32)	32	8(32)	32	32	`fp32`	45.09
Banner *et al.* [1]	8	32	8	32	8	`fp32`	45.36
WAGE [16]	2	8(32)	8(32)	8(32)	8	`fp32`	48.4
NITI (*this work*)	8	16	8	8	8	8	43.66
NITI (*this work*)	8	12	8	4	8	8	33.50
NITI (*this work*)	8	8	8	1	8	8	22.21

(32) means the low precision data format is emulated by quantizing `fp32` number.

4.3 NITI Performance Results on MNIST and CIFAR10 datasets

As shown in Figure 4, we trained LeNet [12] and VGG [15] networks respectively over MNIST and CIFAR10 datasets, using both optimized NITI framework and the baseline `fp32` method. The `fp32` baseline technique uses SGD with momentum 0.9 and weight decay 5×10^{-4}. For the case of LeNet model, we fixed the learning rate to 0.01 and stopped the training at the end of 20^{th} epoch. In the case of VGG, we initiated the learning rate by 0.01 and dropped it to 0.001 at 100^{th} epoch.

As demonstrated, NITI trains LeNet model, which is a relatively small network, with almost identical performance comparing to the `fp32` baseline. In the case of training VGG13 model over CIFAR10, which is a tougher task, NITI was able to successfully train the network with a reasonable descend, achieving a top-1 validation accuracy of $88.24\,\%$. Table 2 summarizes the validation accuracy of our framework on CIFAR10 using few other networks. As observed, the moderate size VGG13 produced the best accuracy by NITI, even when compared to the larger and more capable VGG16 network.

4.4 NITI Performance Results on ImageNet dataset

Table 3 tabulates the results of several related integer-training works on ImageNet dataset using AlexNet. The `fp32` baseline AlexNet were trained using SGD with weight decay 5×10^{-4} but without momentum and batch normalization layer to match with the algorithm currently employed in NITI. Initial learning rate was 0.01 and dropped to 10^{-3} and 10^{-4} at epoch 30 and 60 respectively. Total training epochs were 78. Since the original publication of [1, 8] did not include AlexNet results, we obtained the comparison results by using their released code.

Table 3 shows that given the large data set and the need to classify 1000 categories, additional range and precision were needed to successfully train AlexNet on ImageNet dataset with NITI. We explored different additional bitwidth for weight accumulation and showed that comparable results with floating point baseline can be achieved by using 16 bits for weight accumulation.

As mentioned in Section 2, both [1] and [8] have maintained some degrees of computation using floating point arithmetic during their training process. We believe the added dynamic range and precision have contributed to the generally better accuracy of the resulting network than those produced by NITI. Result of [16] further excluded cross entropy loss and last layer from quantization to avoid performance degradation.

Being the only work that performed network training exclusively with integer arithmetic, we expect further fine-tuning to the NITI algorithm will be able to achieve comparable results to other quantized networks by using the same bitwidth.

5 Conclusion

In this work, we demonstrated the feasibility of integer-only end-to-end training of deep neural networks. Our implementation does not emulate integer training using floating point; rather, it entirely uses native integer operations. The innovations of this work include: our update scheme that utilizes low precision for all intermediate values; a new stochastic rounding scheme that uses discarded bits as a random number source, obviating the need for an additional random number generator; and an efficient cross-entropy loss backpropagation scheme.

Our work lays the foundation for an integer-only accelerator which could greatly reduce cost, chip area and energy consumption required for DNN training. We will also be optimizing our implementation on GPUs and believe that improved performance over floating point on existing GPUs using integer-only techniques is possible.

References

[1] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 5145–5153. Curran Associates, Inc., 2018. 2, 3, 7, 8

[2] Xi Chen, Xiaolin Hu, Hucheng Zhou, and Ningyi Xu. FxpNet: Training a deep convolutional neural network in fixed-point representation. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2494–2501. IEEE, May 2017. 3, 7

[3] Matthieu Courbariaux and Yoshua Bengio. BinaryNet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016. 2, 3

[4] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014. 4

[5] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. 7

[6] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015. 2, 5

[7] 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. 2

[8] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, June 2018. 2, 3, 8

[9] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014. 4

[10] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009. 7

[11] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 7

[12] Yann Lecun, L.D. Jackel, Leon Bottou, Corinna Cortes, J. S. Denker, Harris Drucker, I. Guyon, U.A. Muller, Eduard Sackinger, Patrice Simard, and V. Vapnik. *Learning algorithms for classification: A comparison on handwritten digit recognition*, pages 261–276. World Scientific, 1995. 8

[13] NVIDIA. Turing Architecture. `https://www.nvidia.com/en-us/geforce/turing`, 2019. 4

[14] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016. 2, 3

[15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 7, 8

[16] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks. *arXiv preprint arXiv:1802.04680*, 2018. 3, 7, 8

[17] 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. 2, 3

[18] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016. 2, 3