Flexpoint: Predictive Numerics for Deep Learning

(Invited Paper)

Valentina Popescu, Marcel Nassar, Xin Wang, Evren Tumer, Tristan Webb

Artificial Intelligence Products Group, Intel Corporation

***Abstract*—Deep learning has been undergoing rapid growth in recent years thanks to its state-of-the-art performance across a wide range of real-world applications. Traditionally neural networks were trained in IEEE-754 `binary64` or `binary32` format, a common practice in general scientific computing. However, the unique computational requirements of deep neural network training workloads allow for much more efficient and inexpensive alternatives, unleashing a new wave of numerical innovations powering specialized computing hardware. We previously presented *Flexpoint*, a blocked fixed-point data type combined with a novel predictive exponent management algorithm designed to support training of deep networks without modifications, aiming at a seamless replacement of the `binary32` widely in practice today. We showed that *Flexpoint* with 16-bit mantissa and 5-bit shared exponent (`flex16+5`) achieved numerical parity to `binary32` in training a number of convolutional neural networks. In the current paper we review the continuing trend of predictive numerics enhancing deep neural network training in specialized computing devices such as the Intel®Nervana™ Neural Network Processor.**

***Index Terms*—Flexpoint, Deep Learning, Neural Networks**

1. Introduction

The rapidly growing computational demand by deep learning has led to an increased interest in specialized hardware architectures optimized for training and inference of deep neural networks with increasing performance at decreasing cost. Today's deep learning workloads are commonly processed on CPU and/or GPU architectures using IEEE-754 `binary32` or `binary16` floating-point precision. However, numerous recent studies suggest that substantial improvements in hardware footprint, power consumption, speed, and memory requirements could be obtained with more efficient data formats.

The data format design depends on the target workload: (i) for *inference* workloads, a network, typically trained in `binary32`, is quantized to closely approximate various metrics produced by the higher-precision network; (ii) *training* workloads, on the other hand, present additional numerical challenges as the network tensors dynamically shift between different operational ranges as training progresses.

To meet these challenges, we introduced in [1] *Flexpoint*, a low-precision numerical format specially designed for deep learning applications, that combines the advantages of fixed-point and floating-point arithmetic. It uses a common exponent for all values in an array, thereby reducing computational and memory requirements in comparison with competing methods. Flexpoint's novelty lies in its predictive exponent management algorithm designed to overcome the dynamic range limitations introduced by the blocked feature of the data format. We showed that Flexpoint faithfully maintains algorithmic parity with binary32 in training a wide range of deep network topologies, while, at the same time, substantially reducing consumption of computational resources. Thus, Flexpoint may be a powerful numerical solution for specialized hardware optimized for field deployment of training already existing deep learning models.

2. Deep Learning

This section provides an overview of deep learning, we refer the interested reader to [2] for a more detailed treatment. Deep learning is a branch of machine learning inspired by the human brain that employs models called neural networks. These are organized in layers, i.e. the bottom layer receives input data, while the top layer produces, after learning, a desired output that realizes a meaningful inference of the input data.

These models typically contain millions of parameters and are usually trained iteratively using stochastic gradient descent optimization techniques over vast amounts of data. The gradient is computed using backpropagation, an efficient algorithm that produces the derivative with respect to each parameter. The data is fed to the network in batches (subsets that fit into device memory), in the form of tensors, i.e. multidimensional arrays that admit multilinear algebraic operations. After each batch is forward propagated through the entire network resulting in a loss function value (predicted output), gradients (Jacobian tensors of the loss) are computed and backpropagated through the network. Finally, parameter updates are applied for each layer to complete the training of the batch.

A full pass over the entire training data set comprises an epoch. Networks can require hundreds of epochs before parameter updates become negligible, which is referred to as "convergence". While a typical model may combine many different types of layers, the use of convolutional layers has been a key innovation which constitutes the major computational workload of modern deep networks.

From a numerical perspective, we identify some common trends and issues unique to deep learning:

i Activations can be normalized to have a narrow range of values across the entire set.

ii Layer parameters change slowly in terms of order of magnitude during the course of training, thereby suggesting that low-precision formats can encode values effectively (see Fig.2).

iii A common problem during training is that the gradients may become very small comparing to the parameter's values, thus their value being discarded during update as rounding error.

iv Many of the tensor operations have a large number of multiplyaccumulate (MAC) operations, which may lead to overflows during accumulation.

With hardware implementations in mind, some studies have proposed alternatives for the underlying arithmetic formats used for deep learning. Data formats designed for inference achieved dramatic reductions in bit width from 32-bit all the way down to one bit (i.e. binary networks [3]), and has already made its way into production hardware such as Googles tensor processing unit (TPU) [4]. In contrast, numerical data formats for training are less mature, divided between mixed-precision and homogeneous data formats.

Mixed data formats typically combine `binary32` with a lower precision format either to accelerate the forward pass [5] and/or to perform higher-precision updates on the parameters [6]. While this approach leads to impressive results such as 6-bit gradients [6], it is typically model-specific and requires hardware to support both the high and low precision formats. Homogeneous data formats, on the other hand, represent all tensors in the same format, making them attractive from the design perspective as it yields specialized, efficient hardware. Dynamic fixed-point (DFXP) is an example of a homogeneous format in which all values in a tensor share an exponent, which is adaptively updated based on levels of observed overflows [7]. However, DFXP's passive reaction to overflows is insufficient to train modern neural nets [7].

3. Flexpoint

We propose an adaptive low precision format called Flexpoint, formally defined in Def. 3.1.

Definition 3.1. *Consider a tensor $\mathcal{T}$ with k elements.* Flexpoint *is its representation as two main components:*

(i) *m_i, with $i \in \{1, \ldots, k\}$, is each element's mantissa, stored as an N-bit integer value in two's complement form;*

(ii) *e is an M-bit unsigned exponent, shared across all elements of $\mathcal{T}$, that is dynamically managed.*

We denote this as `flexN+M` *and each element's value can be computed as: $\mathcal{T}_i = m_i \times \varkappa$, where $\varkappa = 2^{-e}$ is the scale.*

Fig. 1 shows an illustration of a `flex16+5` tensor (i.e., 16-bit mantissa and 5-bit exponent) compared to `binary32` and `binary16` tensors. The floating-point structure of size l linked to the Flexpoint tensor is the statistics buffer corresponding to the exponent management algorithm (see Section 3.1).

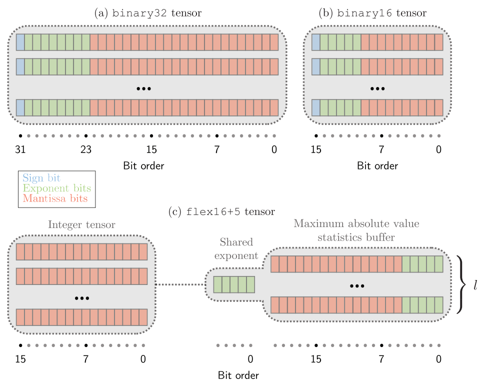

Figure 1. Tensor representations using (a) `binary32` and (b) `binary16` vs. (c) `flex16+5` formats.

In Fig. 2 one can observe the trends in tensor's value range, pointed out in the previous section. It shows that a flex16+5 format is robust enough for encoding tensor variance in its mantissa bits.

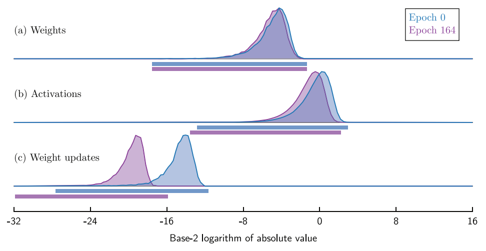

Figure 2. Numerical values for (a) weights, (b) activations and (c) weight updates, during first (blue) and last (purple) epoch of training a residual neural network using `binary32`. The horizontal axis covers the entire range of values that can be represented in `flex16+5`, and the horizontal bars indicate the dynamic range covered by the 16-bit mantissa.

While significantly improving precision compared to binary16, when compared to binary32, flex16+5 reduces not only memory and bandwidth requirements in hardware, but also reduces power and area requirements thanks to simplified fixed-point adders and multipliers. However, these advantages come at the cost of added complexity on the exponent management and dynamic range limitations imposed by sharing one exponent. For hardware-implemented Flexpoint representation, one needs to determine the output exponent before the operation is actually performed, otherwise the intermediate result needs to be stored in high-precision, before reading the new exponent and quantizing the result, which would negate much of the potential savings in hardware.

3.1. *Autoflex* algorithm

Our exponent management algorithm (Autoflex) was specially designed for iterative optimizations where tensor operations are performed repeatedly and outputs are stored in hardware buffers. It aims to predict an optimal exponent for the output of each tensor operation based on tensor-wide statistics gathered from values computed in previous iterations.

Autoflex tracks the maximum absolute value of every tensor using a dequeue to store a bounded history of these values. Based on a statistical model, it is then possible to estimate a trend in the stored values to anticipate an overflow or a decrease in tensor values and adjust the exponent accordingly.

Throughout this section we use the notations introduced in Definition 3.1, to which we add:

- Γ as the maximum absolute mantissa value of $\mathcal{T}$, and
- $\phi = \Gamma\varkappa$ its floating-point representation.

At the beginning of training the statistics queue is empty, so we use a simple trial-and-error scheme described in Algorithm 1 to initialize the exponents. We perform each operation in a loop: we inspect the output value of Γ for overflows or unused mantissa bits, and repeat until the target exponent is found. This mode of operation, called "Init Mode", is employed until the statistics queue is full at which point the tensor is considered to be initialized.

Once a tensor has been initialized, Autoflex is switched to "Adjust Mode" (Algorithm 2) which attempts to predict the optimal exponent based on the collected statistics. We maintain a fixed length dequeue, $\mathbf{f}$, of the maximum floating-point values encountered in the previous l iterations, and predict the expected maximum value for the next iteration based on the maximum and standard deviation of values stored in the dequeue. If an overflow is encountered, the stats buffer is reset and the exponent is increased by one additional bit.

Algorithm 1 *Autoflex Init Mode.* Scales are initialized by repeatedly performing the operation and increasing or decreasing the exponent in the case of overflows or mantissa underutilization, repectively.

```
 1: initialized ← False
 2: $\varkappa = 1$
 3: while not initialized do
 4:     $\Gamma \leftarrow$ returned by kernel call
 5:     if $\Gamma \geq 2^{N-1} - 1$ then
 6:         ▷ overflow: increase scale $\varkappa$
 7:         $\varkappa \leftarrow \varkappa \times 2^{\lfloor \frac{N-1}{2} \rfloor}$
 8:     else if $\Gamma < 2^{N-2}$ then
 9:         ▷ underutilized mantissa
10:         $\varkappa \leftarrow \varkappa \times 2^{\lceil \log_2 \max(\Gamma,1) \rceil - (N-2)}$
11:         ▷ jump directly to target exponent
12:         if $\Gamma > 2^{\lfloor \frac{N-1}{2} \rfloor - 2}$ then
13:             ▷ ensure enough bits for reliable jump
14:             initialized ← True
15:     else
16:         ▷ scale $\varkappa$ is correct
17:         initialized ← True
```

Algorithm 2 *Autoflex Adjust Mode.* The hyperparameters are: $\alpha = 2$ - the multiplicative headroom factor, $\beta = 3$ - number of standard deviations, and $\gamma = 100$ - additive constant. Statistics are computed over a moving window of length l. Returns expected maximum $\varkappa$ for the next iteration.

```
 1: $\mathbf{f} \leftarrow$ stats dequeue of length $l$
 2: $\Gamma \leftarrow$ returned by kernel call
 3: $\varkappa \leftarrow$ previous scale value $\varkappa$
 4: if $\Gamma \geq 2^{N-1} - 1$ then
 5:     ▷ overflow: double scale and clear stats
 6:     clear $\mathbf{f}$
 7:     $\Gamma \leftarrow 2\Gamma$
 8: $\mathbf{f} \leftarrow [\mathbf{f}, \Gamma\varkappa]$ ▷ extend dequeue
 9: $\chi \leftarrow \alpha\,[\max(\mathbf{f}) + \beta \mathrm{std}(\mathbf{f}) + \gamma\varkappa]$
10: ▷ predicted maximum value for next iteration
11: $\varkappa \leftarrow 2^{\lceil \log_2 \chi \rceil - N + 1}$
12: ▷ nearest power of two
```

For memory efficiency, buffers that are used for intermediate results can be reused since statistics buffers are stored separately for each computation. This allows for the allocated memory to be reused without disrupting the exponent management.

3.2. Autoflex Example

The following example is taken from the initial Flexpoint paper [1] and it illustrates the Autoflex algorithm by training a small multilayer perceptron (MLP) with 2-layers for 400 iterations on the classic CIFAR-10 dataset.

During training, $\varkappa$ and Γ values are stored at each iteration, as shown in Fig. 3.2. The weight tensor, illustrated in Fig. 3.2(a), is highly stable as it is only

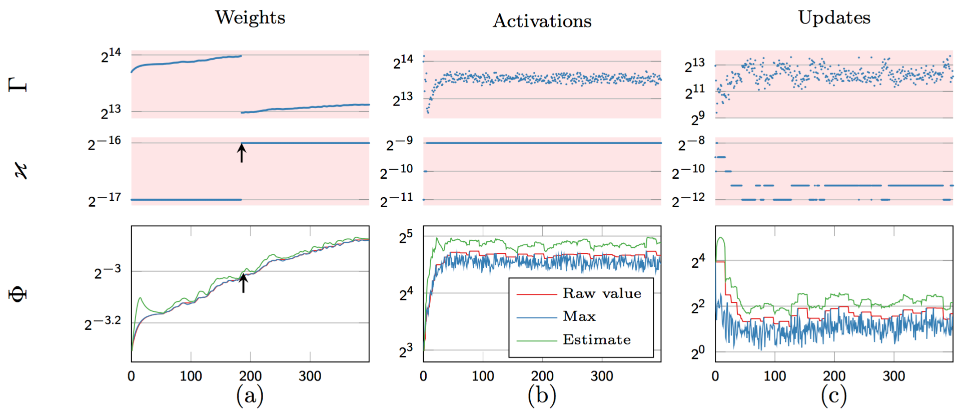

Figure 3. Evolution of different tensors during training with corresponding mantissa and exponent values. The black arrow indicates how scale changes are synchronized with crossings of the exponent boundary. In each case Autoflex estimate (green line) crosses the exponent boundary (gray horizontal line) before the actual data (red) does, which means that exponent changes are predicted before an overflow occurs.

updated with small gradient steps. In the two upper plots we can observe that Γ slowly approaches its maximum value of 2^{14}, at which point the $\varkappa$ value is updated, and Γ drops by one bit. The last plot shows the corresponding floating-point representation of the statistics computed from ϕ, which is used to perform the exponent prediction. Using a sliding window of $\ell = 16$ values, the predicted maximum is computed, and used to set the exponent for the next iteration. In this first case, the prediction crosses the exponent boundary of 2^3 about 20 iterations before the value itself does, safely preventing an overflow.

Tensors with more variation across epochs are shown in Fig. 3.2(b) - activations and Fig. 3.2(c) - updates, respectively. The standard deviation across iterations is higher, therefore the algorithm leaves about half a bit or one bit of headroom. Even as the tensor fluctuates in magnitude by more than a factor of two, the maximum absolute value of the mantissa Γ is safely prevented from overflowing.

Even though the goal of Autoflex is to keep the maximum absolute mantissa values at the top of the dynamic range without overflowing, its flexibility does not guarantee that this is always the case. For instance, when looking at the updates tensor, Γ reaches 3 bits below the cutoff, which means that the 3 leading bits are zero and we use only 13 of the 16 mantissa bits for representing the data.

4. Conclusions

In the previous work [1] we demonstrated that a Flexpoint data format, `flex16+5`, achieved numerical performance on par with `binary32` in training a number of deep convolutional networks without altering model design and topology.

Some of the advantages observed include: no retuning of hyperparameters is necessary; the training procedure remains exactly the same, eliminating the need of intermediate high-precision representations (except for the intermediate higher precision accumulation needed for multipliers and adders); networks trained in floating-point formats can be readily deployed in Flexpoint hardware for inference. Despite the added complexity of exponent management, Autoflex algorithm manages all Flexpoint tensors in the same way, an encapsulated mechanism hidden from the user.

Our discovery supports Flexpoint's potential in realizing gains in efficiency and performance of future hardware architectures specialized for deep neural network training. Our work laid the foundation for future research on remaining questions: (i) potentials and implications of using other data formats in the Flexpoint family, namely `flexN+M` for certain (N, M), (ii) numerical limitations and proper error analysis on the results, (iii) whether/how the Autoflex algorithm could be improved for better performance or less computation, etc.

Future research into the numerical properties of data formats will lead to future efficiency in machine learning, and this can be just as important to the field as new models and algorithms.

References

[1] U. Köster *et al.*, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," *NIPS 2017*, pp. 1742–1752.

[2] I. Goodfellow *et al.*, *Deep Learning*. MIT Press, 2016.

[3] K. Minje and S. Paris, "Bitwise neural networks," *arXiv preprint arXiv:1601.06071*, 2016.

[4] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," *arXiv preprint arXiv:1704.04760*, 2017.

[5] M. Courbariaux *et al.*, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," *arXiv preprint arXiv:1602.02830*, 2016.

[6] S. Zhou *et al.*, "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[7] M. Courbariaux *et al.*, "Training deep neural networks with low precision multiplications," in *ICLR 2014*.