Vol.2021-ARC-244 No.44
Vol.2021-SLDM-194 No.44
Vol.2021-EMB-56 No.44
2021/3/26

量子化深層学習のための精度シミュレーション

田宮 豊* 橋本 鉄太郎* 川辺 幸仁*

概要：深層学習の高速化手法として、量子化によるデータ削減の有効性が知られている。その一方で、学習精度劣化の可能性があるため、実装対象であるハードウェアとニューラルネットワークに適した量子化方式を求める必要がある。 本論文では、最適な量子化方式を決定するために、汎用の量子化精度シミュレータを開発した。精度シミュレーションでは、様々な量子化方式(データ型、スキーム、量子化パラメタ)を指定でき、さらに、複数の量子化方式をネットワークのレイヤ毎に、もしくは、activation, gradient, weight 毎に組み合わせることが可能である。 本精度シミュレータは、 PyTorch の Quantization-Aware Training (QAT) を用いて実装したことにより、シミュレーション時間は、量子化無しの場合の 2.7 倍以下に抑えることが可能となった。

キーワード：深層学習，量子化, Quantization-Aware Training, PyTorch

A Precision Simulator of Deep Neural Network Quantization

TAMIYA Yutaka* HASHIMOT Tetsutaro* KAWABE Yukihito*

Abstract: The effectiveness of data reduction by quantization is known as a method for speeding up deep learning. On the other hand, since there is a possibility that the learning accuracy tends to be worse, it is necessary to find a quantization method suitable for the hardware to be implemented and the neural network. In this paper, we have developed a general-purpose quantization accuracy simulator to determine the optimum quantization method. In the accuracy simulation, various quantization methods (data type, scheme, quantization parameter) can be specified, and multiple quantization methods can be combined for each layer of the network or for each activation, gradient, weight. Is. By implementing this precision simulator based on PyTorch's Quantization-Aware Training (QAT), the simulation time can be suppressed to 2.7 times or less of his time without quantization.

Keywords: Deep Learning, Quantization, Quantization-Aware Training, PyTorch

* (株)富士通研究所 Fujitsu Laboratories, Ltd.

Vol.2021-ARC-244 No.44
Vol.2021-SLDM-194 No.44
Vol.2021-EMB-56 No.44
2021/3/26

1. はじめに

深層学習の高速化手法として、量子化の有効性が知られている。量子化が最初に注目された用途は推論である[1][2]。これらの取り組みでは、監視カメラの画像認識のような計算資源が十分でない環境でもリアルタイム処理を実現するために量子化が有用である。

データ量の削減の需要が高い。最近の研究では、学習用途の量子化も試みられている[3][4]。学習には膨大な計算量が必要なため、大量の計算機資源 (CPU, GPU, メモリ, ストレージなど)を備えた並列計算機環境で処理されることが多い。

いずれの用途でも、量子化による効果は、データ量削減による処理の高速化である。例えば、 32 ビット浮動小数点データ(FP32)を 8 ビットに量子化できれば、ニューラルネットワークもテンソルデータもサイズが 1/4 となり、使用メモリとデータ転送量の削減が可能となる。更に、 8 ビット演算命令を備える計算機の場合は、計算時間の削減も可能である。

一方、量子化ではデータ量が削減されるため、量子化しない場合と同じ学習精度を達成するのは困難である。一般的な量子化の適用では、ニューラルネットワークに対して学習精度を維持可能な量子化方式を決める必要がある。ここで指す量子化方式では､データ型の他に､量子化スキーム、量子化パラメタ等、様々な種類が存在する。また、ニューラルネットワークへの量子化適用についても、レイヤ毎で異なる量子化方式を適用する場合や、activation, weight, gradient で量子化方式を変える場合もある[3][4]。このように種類と適用箇所に選択の余地が存在する量子化方式の中で、学習精度を維持できる組み合わせをニューラルネットワーク毎に決める必要がある。

上記のように様々な組み合わせが存在する量子化方式であるが、全ての種類の量子化方式を実行環境である計算機環境に実装するのは困難である。量子化の実装は、機械語レベル、Deep Learning ライブラリレベル、フレームワークレベルの 3 レベルから構成される。機械語レベルでは、量子化されたデータ型の演算命令を計算機アーキテクチャがサポートしているかに依存する。例えば、8 ビット整数 FMA (Fused Multply-Add) 命令を持つ計算機の場合、量子化 Convolution 演算を高速に実行できる。

Deep Leaning レベルの実装では、ニューラルネットワークの各レイヤが量子化演算命令を使用しているかに依存する。例えば、前述の Convolution 演算では、与えられた入力テンソルと重みテンソルのサイズに応じて、中間データのメモリ配置と 8 ビット整数 FMA 命令の適用順序を最適化することが必要になる。

最後のフレームワークレベルの実装では、量子化されたレイヤ同士の接続をサポートしてニューラルネットワークとしての全体動作を実現することである。例えば、FP32 データを量子化ニューラルネットワークに入力する場合や、レイヤ間で量子化方式が異なる場合は、適切なデータ型変換が行われる必要がある。

このように量子化ニューラルネットワークの実装では機械語レベルからフレームワークレベルに至るまで、ソフトウェアスタックを広範囲に変更することが必要であり、その作業には多大な開発工数が必要となる。そのため、現行の Deep Learning 環境への量子化実装は積極的に行われていない(特に推論より学習)。この状況は、量子化ニューラルネットワークの県有開発を難しくしている。

以上の背景から、我々はニューラルネットワークの量子化学習精度の評価を目的として、量子化精度シミュレータを開発した。量子化をソフトウェア的にシミュレートすることにより、一般的な計算機環境で量子化ニューラルネットワークを実行できる。今後開発予定の計算機環境および量子化方式を想定した学習精度の評価も可能となる。

本シミュレータの特長は以下の 3 点である：

- 様々な量子化方式に対応している。
- PyTorch Quntization-Aware Trainig (QAT) をベースにしており、量子化方式の拡張が容易。
- 分散/並列実行による高速実行が可能。

本論文の構成は以下のとおりである。第 2 章で量子化方式について述べる。第 3 章でベースとなる Quantization-Aware Training を説明し、第 4 章で本精度シミュレータの技術を説明する。そして、実験評価とまとめを述べる。

2. 量子化方式

本精度シミュレータがサポートする量子化方式は、Qint と FlexFP (Flexible Floating-Point)である。

Qint は int 型テンソルと、量子化パラメタでデータを表す方式である。量子化は、量子化パラメタ(*scale, zero_point*)を使って定義される：

$$Q(x,\ scale,\ zero\_point) = round(\frac{x}{scale} + zero\_point)$$

テンソル内の値を観測して最大値と最小値を求め、それらを int 型データの最大値と最小値に写像するように (*scale, zero_point*) を計算する。量子化方法として、量子化パラメタをテンソル毎に 1 つ持つ Per Tensor と、テンソルのチャネル毎に持つ Per Channel の 2 種類ある。更に、テンソルの最大値/最小値を観測する MinMaxObserver と、最大値と最小値のそれぞれの移動平均を観測する MovingAverageMinMaxObserver の 2 種類が Per Tensor 用と Per Channel 用のそれぞれに定義される。

Vol.2021-ARC-244 No.44
Vol.2021-SLDM-194 No.44
Vol.2021-EMB-56 No.44
2021/3/26

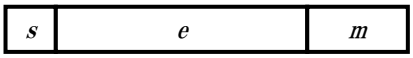

図 1　浮動小数点のビットフォーマット

FlexFP は、任意ビットフォーマットを持つ浮動小数点である。浮動小数点のビットフォーマットは、符号部 *s*、指数部 *e*、および、仮数部 *m* から構成される(図 1)。これらのビット数 *s-ebit-mbit* の組み合わせを自由に設定できるものが FlexFP である(*s* は符号なので 1 ビット固定)。量子化対象テンソルの値レンジに合せるため、指数部バイアス *b* を設ける。従って、FlexFP が表現する値は、以下の式で与えられる:

$$x = (-1)^s\left(1 + \sum_{i=1}^{mbit} m_i 2^{-i}\right) \times 2^{e-(2^{ebit-1}-1)+b}$$

尚、Bfloat16 は (*ebit*, *mbit*, *b*) = (8, 7, 0) の特殊な FlexFP である。

指数部バイアス *b* は、定数固定の FlexFP と、テンソルの値を観測して動的に計算する FlexFP DSE (FlexFP with dynamic shared bias)[5]の 2 種類ある。

3. Quantization-Aware Training (QAT)

本精度シミュレータは、Quantization-Aware Training (QAT) をベース技術としている。本章では、特に PyTorch[6]に実装されている QAT について説明する。

QAT は、量子化を非量子化ニューラルネットワーク(通常は FP32 データ型)でシミュレートする仕組みである。

Forward 伝播は、量子化レイヤの演算を FP32 で行う。演算結果は指定された量子化方式で丸められ、これを FP32 値として保持する。本来の量子化は行わず、量子化を模倣するだけなため、量子化によって期待されるデータ量削減や計算時間の削減効果は無い。逆に、量子化レイヤの挿入によってデータ量も計算時間も増加する。

Backward 伝播は、Forward 伝播と同様に演算を FP32 で行い、演算結果を FP32 値として保持する。量子化レイヤでは、量子化の逆関数を使う点が Forward 伝播と異なっている。

PyTorch の QAT の特長として、FP32 で記述されたニューラルネットワークモデルを簡単に QAT に変換できる点にある。即ち、その FP32 モデルに対して PyThon 関数 prepare_qat() を呼ぶだけで QAT 変換が完了する。そして、FP32 用の学習ループと評価関数は、無修正で使用することが可能である。なお、prepare_qat()関数は、FP32 ニューラルネットワークに対して量子化レイヤを自動挿入する(図 2)。量子化対象レイヤの挿入箇所は、Conv, Linear (full connected レイヤ), BatchNorm, ReLU の出力テンソルと weight テンソル(Conv と Linear のようにレイヤ自身が持っている場合)である。それらの箇所に量子化を担う FakeQuantize クラスが挿入される。更に、各 FakeQuantize クラスは Observer クラスを伴う。 Observer クラスは、指定された量子化方式に従って、量子化パラメタを計算する役目を持つ。その計算において、必要ならば関係づけられたテンソル値を観測する。例えば、 Qint では、この観測によってテンソル内の最大値と最小値を求めることができる。なお、ユーザが QuantStub クラスを手動でニューラルネットワーク中に挿入することで、任意テンソルを量子化可能である。

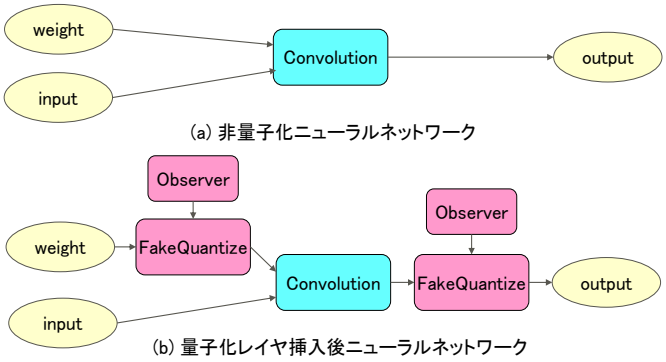

図 2　量子化レイヤの挿入

4. 量子化学習精度シミュレータ

本精度シミュレータでは、オリジナル QAT 実装に対して以下の 3 点の機能拡張をした:

① 量子化方式 FlexFP の追加
② Backward 時の gradient 量子化
③ 量子化学習時に確率的丸めを行う

これらの機能拡張は、図 3 のクラス関係図の色付け箇所に当たる。

追加する量子化方式 FlexFP は Observer クラスの派生クラス(FlexFpObserver, FlexFpDynEbiasObserver) に実装する。指定されたビット数 (*ebit*, *mbit*, *b*)に従ってテンソルの値を丸める。その際、テンソルの値は FP32 が(*ebit*, *mbit*, *b*)=(8, 23, 0)の FP フォーマットで保持されていることを利用すると、FlexFP の丸め処理は整数データに対するビットシフト、ビット AND、およびビット OR の演算で計算できる。更に、これらのビット演算はテンソル内の全ての要素に共通であるため、近年の CPU と GPU に備わる SIMD 命令で並列計算が可能である。また、FleFP8 DSE では、 shared exponent の計算のため、テンソル内の最大絶対値を持つ要素を求める必要がある。この計算もテンソル SIMD 命令で効率的に計算可能である。

Vol.2021-ARC-244 No.44
Vol.2021-SLDM-194 No.44
Vol.2021-EMB-56 No.44
2021/3/26

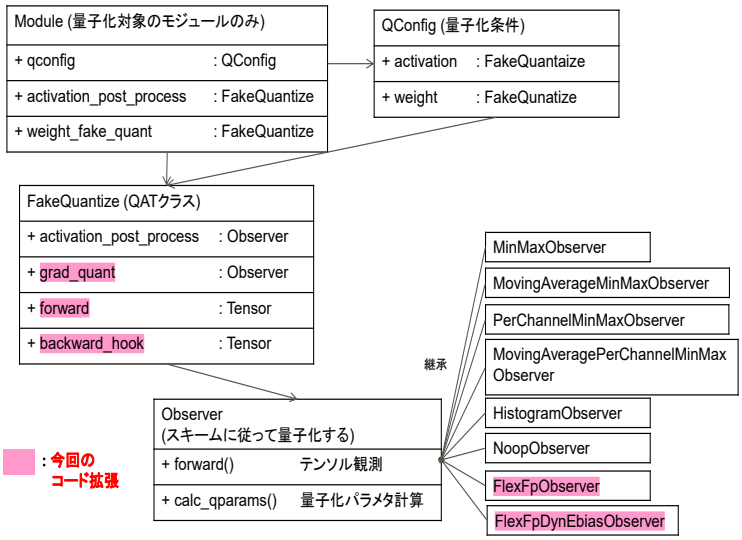

図 4 QAT 関連のクラス図

```
class FakeQuantize(torch.nn.Module):
  def __init__(self, ...):
    ...
    self.register_backward_hook(.backward_hook)

  def backward_hook(self, dX, dY):
    self.grad_quant(dY[0])
    _scale, _zero_point =  ¥
        self.grad_quant.calculate_qparams())
    dx = torch.fake_quantize_per_tensor_affine(dY[0],
            scale, zero_point,
            self.grad_quant_min, self.grad_quant_max,
            self.training)
    return (dx,)
```

図 4 FakeQunatize クラスの gradient 量子化

Backward 時に量子化レイヤの gradient テンソルに対しても Forward 時と同じ丸め処理を行う。これを実現するため、PyTorch からレイヤの backward 伝播が呼び出される際に hook 関数を呼ぶ仕組み “backward hook” を用いた。Python コードの抜粋を図 4 に示す。FakeQuantize クラスのコンストラクタ(__init__()関数)において、backward hook として backward_hook()関数を登録する。backward_hoook()関数では、与えられた gradient テンソル (dY)を Observer クラス(変数 self.grad_quant) で観測して、量子化パラメタを計算する。この量子化パラメタと gradient テンソルは、forward 時と同じように fake_quntize_per_tensor_affine() 関数に渡して量子化に相当する丸め処理が行われる。

なお、PyTorch には autograd という gradient を自動計算する仕組みがあるが、量子化学習では使わない。autograd には gradient テンソルを観測できないという制約が有るからである。よって、量子化レイヤの gradient 量子化は図 4 のように backward hook を使って実現した。

量子化学習時に確率的丸めは、学習精度の向上のために確率的丸めは必須であるため[3]、本精度シミュレータに取り入れた。確率的丸めとは、データの有効桁数以下に [0, 1)の範囲で発生させた乱数を加算して丸め処理を行う手法である。繰り上がりの有無をランダムにすることで、学習時の量子化誤差が緩和される。確率的丸めは学習時のみに適用し、推論時は使わない。その制御は、図 4 に示すよう、量子化関数 fake_quntize_per_tensor_affine() に対して、学習と推論のモードを区別する変数 self.training を引数として渡すことで実現する。

5. 実験評価

本シミュレータを PyTorh v1.6 に実装し、代表的なニューラルネットワーク[7]で量子化学習精度を評価した。

ResNet50 に対して、NVIDA V100 GPU を 4 台搭載したサーバ上で、各種量子化方式を 4 epoch 分実行した結果を表 1 に示す。FP32 をベースラインとして、6 種類の量子化方式を試している。(1) オリジナルの Qint8 の QAT (activation (forward) が Qint8 で gradient が FP32)、(2) activation も gradient も Qint8、(3) Bfloat16、 (4) [4]が提案する HFP8 (forward が 1-4-3, gradient が 1-5-2 の FlexFp8)、 (5) (4)に

表 1 ResNet50 に対する量子化精度シミュレーション結果(4epoch 分)

	FP32	Qint8 w/ grad=fp32	Qint8 w/ grad=qint8	Bfloat16	FlexFP8 fwd=(1-4-3), grad=(1-5-2)	FlexFP8 w/ grad-scaling=10k	FlexFP8 w/ dyn shared bias
認識精度	53.1%	60.7%	44.9%	59.6%	0.5%	0.5%	58.1%
Loss	3.24	3.47	4.09	3.49	6.91	6.91	3.43
実行時間	268m42s	523m03s	724m26s	507m33s	505m32s	513m25s	653m44s
実行時間比	1.00	1.95	2.70	1.89	1.88	1.91	2.43

Vol.2021-ARC-244 No.44
Vol.2021-SLDM-194 No.44
Vol.2021-EMB-56 No.44
2021/3/26

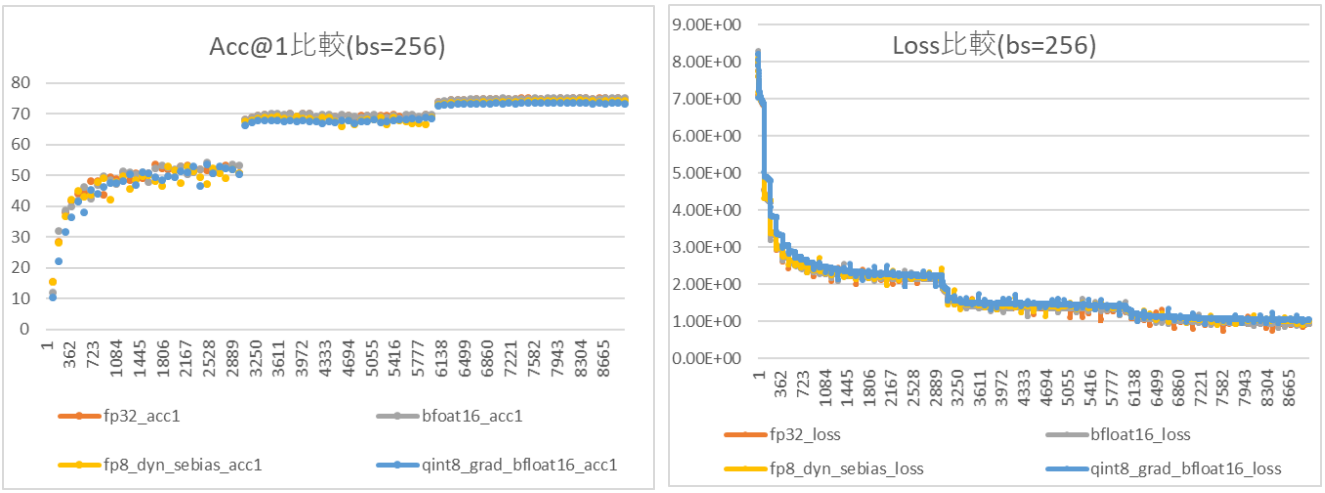

図 5　ResNet50 の量子化学習精度シミュレーション結果

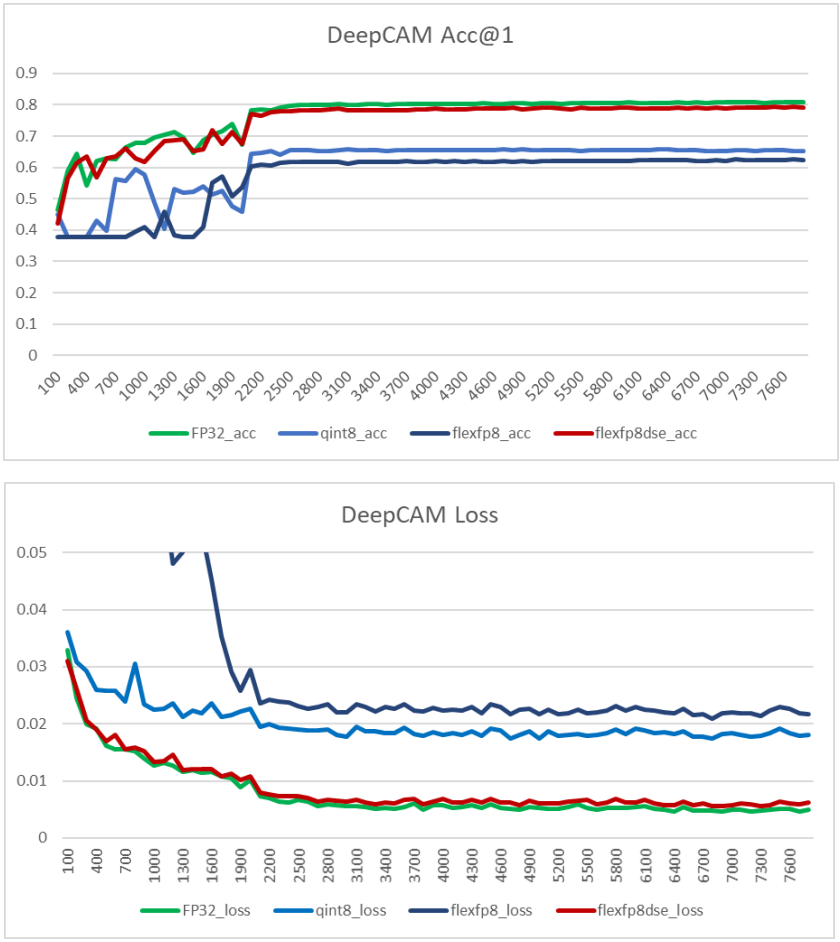

図 6　DeepCAM の量子化学習精度シミュレーション結果

grad scaling を適用したもの、そして、 FlexFP8 DSE (dynamic shared exponent) である。

どの量子化方式でも、FP32 に比べて 2.7 倍以下の実行時間となった。これは、本シミュレータが SIMD 演算に最適化されたテンソル演算を最大限に使って実装した効果が表れたため、2.7 倍で収まったと言える。Qint8 は、 量子化パラメタの scale を求めるための除算で実行時間が掛かっている。特に、gradient も Qint8 になると、更に遅くなり、

Vol.2021-ARC-244 No.44
Vol.2021-SLDM-194 No.44
Vol.2021-EMB-56 No.44
2021/3/26

FP32 よりも 2.7 倍のシミュレーション時間になった。他の Bfloat16, FlexFP8 は整数 SIMD 演算の効果により、2 倍程度の実行時間となっている。但し、FlexFP8 DSE は 2.43 倍の時間が掛かった。これは、FlexFP8 DSE はテンソル内の最大絶対値を求める reduce 演算が原因と考えられる。

次に、ResNet50 を 90 epoch 分実行した結果を図 5 のグラフに示す。左図は Top 1 の正答率(Acc@1)、右図は loss 値の推移を表している。Bfloat16 の学習曲線はかなり FP32 と酷似していることが分かる。FlexFP8 DSE は Bfloab16 よりも FP32 に近い学習精度を達成しており、効果的な量子化ということが分かる。

表 2　DeepCAM に対する量子化精度シミュレーション結果(4epoch 分)

@abci 64nodes	FP32	Qint8	FlexFp8	FlexFp8 dse
実行時間(sec)	1,759	3,667	2,535	2,608
実行時間比	1.00	2.08	1.44	1.48

更に、ML-Perf DeepCAM を産業技術総合研究所の GPU クラウド ABCI における 64 ノード(NVIDIA V100 256 台)で評価した(表 2)。対 FP32 で、Qint8 は 2.08 倍、FlexFP8 DSE は 1.48 倍の実行時間で収まった。量子化方式ごとの学習曲線を図 6 に示す。ここでも、FlexFP8 DSE が FP32 に非常に近い学習曲線を示しており、その有効性を確かめることができた。DeepCAM は大規模データを扱うベンチマークであり、FP32 ですら実行が困難である。そのような状況でも、量子化精度シミュレーションが実用時間で実行できることは有意義である。

6. おわりに

本論文では PyTorch QAT をベースとした量子化学習精度シミュレータについて述べた。QAT に対して僅かなコード拡張で様々な量子化方式の組み合わせを試すことが可能なことを示した。また、その拡張は PyTorch の分散並列実行機構である DataParallel および DistributedDataParallel と親和性が高く、非量子化と比べて実行時間が 2.7 倍以内に収まる等、高い実用性も示した。

量子化学習では未熟な研究分野であり、未だに最適な量子化方式の探索が続いている。このような状況では、本精度シミュレータのように高い拡張性と実行性能を有するツールは有用と考える。今後我々は、本精度シミュレータを用いて、最適な量子化方式の開発に取り組みたいと考える。

参考文献

[1] Intel, "OpenVINO: Deploy High-Performance Deep Learning Inference", https://software.intel.com/content/www/us/en/develop/tools/openvino-toolkit.html
[2] Xilinx, "Vitus AI: Adaptable and Real-Time AI Inference Acceleration", https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html
[3] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi, "Training and Inference with Integers in Deep Neural Networks", https://arxiv.org/pdf/1802.04680.pdf, Proc. of ICLR 2018.
[4] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan, "Hybrid 8-bit Floating Point (HFP8) Training and Inference for Deep Neural Networks", Proc. of NIPS 2019.
[5] Hisakatsu Yamaguchi, Makiko Ito, Katsuhiro Yoda and Atsushi Ike, "Training Deep Neural Networks in 8-bit Fixed Point with Dynamic Shared Exponent Management", Proc. of Design Automation and Test in Euro Conference, 2021 (掲載予定).
[6] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer, "Automatic differentiation in PyTorch", Proc. of NIPS 2017 Autodiff Workshop, 2017
[7] ML Perf Training, https://mlcommons.org/en/news/mlperf-training-v07/