Minimizing Power for Neural Network Training with Logarithm-Approximate Floating-Point Multiplier

TaiYu Cheng, Jaehoon Yu, Masanori Hashimoto
Department of Information Systems Engineering, Osaka University
t-cheng@ist.osaka-u.ac.jp, yu.jaehoon@ist.osaka-u.ac.jp, hasimoto@ist.osaka-u.ac.jp

***Abstract*—This paper proposes to adopt logarithm-approximate multiplier (LAM) for multiply-accumulate (MAC) computation in neural network (NN) training engine, where LAM approximates a floating-point multiplication as an addition resulting in smaller delay, fewer gates, and lower power consumption. Our implementation of NN training engine for a 2-D classification dataset achieves 10% speed-up and 2.5X and 2.3X efficiency improvement in power and area, respectively. LAM is also highly compatible with conventional bit-width scaling (BWS). When BWS is applied with LAM in four test datasets, more than 5.2X power efficiency improvement is achievable with only 1% accuracy degradation, where 2.3X improvement originates from LAM.**

***Index Terms*—approximate computing, neural network, training engine, floating-point unit, logarithmic multiplier, multiply-accumulate operation**

I. Introduction

Advanced artificial intelligent (AI) technology has driven Big Data analysis to benefit industries such as finance, medicine, and manufacturing. For delivering AI efficacy to our daily life, emerging edge computing moves the data and services from the cloud to the vicinity of edge terminals, where the data can be analyzed in the edge server before sending to the cloud [1]. Edge computing is expected to provide more local and real-time services than cloud, but the servers are required to enable in-situ data processing and judgment. With this trend, energy-efficient hardware training engines that can be accommodated in edge servers are demanded and widely studied [2] [3].

Neural network (NN) is one of the most widely-used techniques in machine learning [4]. A feedforward NN model is composed of a few to many layers, each of which includes many neurons. The neurons are connected layer by layer through synaptic weights. The synaptic weights are optimized to provide sufficiently high accuracy through computationally expensive training phase. Hardware NN system is mainly categorized into two types. The first one is inference engine that processes a network with given pre-trained weights, and the latter is training engine that has additional capability of weight optimization in training phase. Regardless of inference or training engine, multiply-accumulate (MAC) arithmetic computation is the primary operation. Rapidly increasing trend of NN size to deal with more difficult and sophisticated problems explodes the amount of MAC computation, resulting in a strong demand for dedicated hardware engines.

Inference engines in several studies exploit inherent error-tolerant property in machine learning and introduce approximate computing (AC) techniques for gaining performance and reducing cost [5]–[8]. Among various AC techniques for inference engine, bit-width scaling (BWS), which reduces the bit width of data representation, is the most popular way that trades computation reduction with accuracy degradation [5], [9]–[11]. Even binarized neural networks are studied.

In contrast to inference, training engine needs to perform more arithmetic computation with a wider dynamic range since the gradient, which is numerically computed and used to guide the weight update, spreads in a large dynamic range [9]. Due to this, adopting floating-point units (FPUs) is beneficial to training engine to accommodate such gradient computation needing a large dynamic range, but FPUs are known power hungry and area expensive [12]. Especially, multiplication is one of the most power-hungry and space-demanding arithmetic operators in FPUs, and hence massive MAC computation in NN training deteriorates the power and design efficiency. Therefore, power-efficient floating-point multiplication is highly demanded by training engine development.

In this paper, we propose to adopt logarithm approximate floating-point multipliers (LAMs [13]) in training engine. The advantages for adopting LAMs in training NN include: (1) LAM approximates expensive floating-point multiplication as cheap fixed-point addition resulting in significant power and area saving, (2) even adopting LAM, BWS is applicable, and both advantages of LAM and BWS can be obtained. To the best of our knowledge, this paper reports the first attempt that incorporates LAM with BWS into floating-point training engine. In a 2-D classification case [14], LAM based training engine provides 10% speed-up and 2.5X power reduction in addition to 2.3X area saving. Combining LAM with BWS, the improvement increases to 32% speed-up and 5.5X power reduction in addition to 5X area saving. Moreover, even BWS is applied with LAM, training accuracies for several datasets are comparable, which means they are sometimes better or worse, and the accuracy degradation is at most 1.0%. A projection of the hardware performance from the simple 2-D classification case to larger-size NNs shows that more than 5.2X power reduction is attainable, where 2.3X improvement originates from LAM.

The rest of this paper is organized as follows. Section II reviews NN with its training and related works. Section III introduces LAM and discusses its approximation error.

978-1-7281-2103-1/19/$31.00

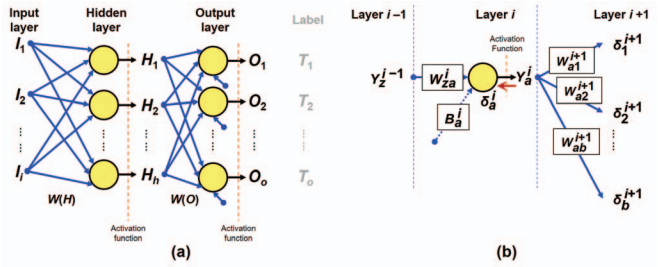


Fig. 1. Schematics of (a) forward-propagation, and (b) back-propagation in a feed-forward neural network.

Experimental results of adopting LAM in training engine are presented in Section IV. and Section V concludes this paper.

II. Preliminaries, Related Work and Research Motivation

A. Basics of Neural Network

Fig. 1(a) illustrates a multilayer perceptron (MLP) structure, which is known as a basic feedforward NN [15]. Each neuron in the network computes a sum of all the states of neurons in the previous layer multiplied with corresponding synaptic weights, passes the sum through a non-linear activation function to determine its state, and then propagates the state to the next layer. This procedure can be expressed as:

$$Y = \text{Act}(\sum WX + B), \tag{1}$$

where X denotes the state of each neuron in the previous layer, W and B denote corresponding synaptic weights and bias, Act() represents a nonlinear activation function, which usually allows passing > 0 values or limites the values between -1 to 1, and Y is the state of the neuron. Referring to Fig. 1(a), (X,W,Y) could be either (I,$W(H)$,H) or (H,$W(O)$,O). This procedure, so-called forward propagation, keeps going until the states of all the neurons in the output layer are determined, which is the core and dominant operation that an inference engine with pre-trained weights executes.

On the other hand, training NN aims at finding a set of synaptic weights and bias values to minimize the loss function (Loss), which is usually defined as the error squared between the state from the forward propagation results in output layer O and target T [15]. In the beginning of training phase, all the weights are randomly initialized (biases can be initially set to 0) and then forward propagation is launched. The next step is to distribute the loss according to the contribution of each synaptic weight (W) and bias (B), which can be obtained through computing their gradient (∂Loss/∂W and ∂Loss/∂B). Based on the computed gradient, each synaptic weight and bias can be numerically updated during each iteration. Let us take Fig. 1(b) as an example. Suppose a synaptic weight W^i_{za} connects the z-th neuron in the (i-1)-th layer (state = Y^{i-1}_z) with the a-th neuron in the i-th layer (state = Y^i_a) and a bias B^i_a is in the a-th neuron in the i-th layer. Then, W^i_{za} and B^i_a can be trained by:

$$W^i_{za} \mathrel{+}= -\eta \frac{\partial \text{Loss}}{\partial W^i_{za}} \text{ where } \frac{\partial \text{Loss}}{\partial W^i_{za}} = \delta^i_a Y^{i-1}_z, \tag{2}$$

$$B^i_a \mathrel{+}= -\eta \frac{\partial \text{Loss}}{\partial B^i_a} \text{ where } \frac{\partial \text{Loss}}{\partial B^i_a} = \delta^i_a, \tag{3}$$

The gradient terms ∂Loss/∂W^i_{za} and ∂Loss/∂B^i_a in (2) and (3) share the same term δ^i_a while ∂Loss/∂W^i_{za} further includes the term Y^{i-1}_z. Basically, the gradient terms would decay during the weight and bias update, and thus these are also called gradient decent method. η is the learning rate, and δ^i_a is conditionally formulated as follows. If the i-th layer is the output layer, δ^i_a is:

$$\delta^i_a = \text{Act}'(O_a)(O_a - T_a), \tag{4}$$

where O_a represents the state computed through forward propagation, T_a means its target state, and Act' means the derivative of activation function. If the i-th layer is not the output layer, referring to Fig. 1(b), δ^i_a is formed as:

$$\delta^i_a = \text{Act}'(Y^i_a)(\sum_{n=1}^{b} W^{i+1}_{an}\delta^{i+1}_n), \tag{5}$$

where W^{i+1}_{an} denotes the synaptic weight to the n-th neuron in the (i+1)-th layer. δ^{i+1}_n can be recursively computed through (4) and (5). Note that, with (4) and (5), the output loss is propagating backward from the output layer, and thus this procedure is named as back-propagation [15]. In addition, (5) indicates that the δ^i_a in the non-output layer needs to compute all the weighted sum of δ^{i+1}_n, meaning that MAC computation is also primary in back propagation. Therefore, training phase executes huge amount of MAC computations during the iteration loops of forward and back propagation. In addition, the gradient terms have a large dynamic range, and thus adopt floating-point arithmetic is beneficial in training.

B. Related Work and Research Motivation

Several papers propose to introduce bit-width scaling [6] [7] and approximate multipliers [16] [17] into NN. The former uses fewer bits in computation while the latter aims at reducing the resource for multiplication operation.

Logarithm based multiplier is a typical type of approximate multiplier since logarithm converts multiplication to addition. Some researches exploit logarithm based multiplier in NN. Reference [11] applies iterative logarithmic multipliers (ILM) with error tolerant algorithm whereas [16] directly converts multiplicand and multiplier into log domain to do multiplication as an addition. The above studies are all based on fixed-point units. Recent works [18] and [19] propose to adopt logarithm based floating-point multiplier in NN, but [19] only adopts it in inference engine. As for floating-point training engine, existing works study BWS only [5], [9], [10]. Although [18] applies logarithm based multiplier to training, logarithm based multiplier is used only in the early stage of training, and the exact multiplier is used in the latter training stages. Therefore, [18] demands an exact multiplier in addition to an approximate one and relies on the exact multiplier for training. Also, cooperative design with BWS is not addressed.

In summary, previous studies for floating-point NN training intensively focus on BWS, and it left the space for evaluating

the efficacy of logarithm based multiplier in training. Also, BWS is still the primary choice in training engine design, and hence the compatibility between logarithm based multiplier and BWS must be investigated. Taking into account the tremendous number of MAC operations in training engine and the limited power budget for edge computing, exploring a useful approximate technique for pursuing higher energy efficiency and examining its compatibility with BWS could give a useful implication for training engine designers, which is the objective of this work.

III. Logarithmic Approximate Multiplier

Logarithmic approximate multiplier, LAM in short, is developed by [13]. With an approximation, a floating-point value in linear domain can be regarded as its value taken by logarithm of base 2 in fixed-point format. Thanks to the log-domain property, floating-point multiplication can be simplified to fixed-point addition. This section introduces LAM and analyzes its approximation error.

A. Floating-point Multiplication

Floating-point format consists of three parts to represent a value in scientific notation; one bit for sign, several bits for exponent, and the remaining bits for fraction. When a floating-point contains N exponent bits and M fraction bits, a value is represented by:

$$(-1)^{\text{sign}} * (1 + \text{fraction}/2^M) * 2^{(\text{exponent}-\text{bias})}, \quad (6)$$

where bias equals to $2^{(N-1)}$-1. Fig. 2 explains the multiplication of two floating-point values, where these three parts in the floating-point representation are processed individually. The sign parts take an exclusive-or function, then the exponent parts are added, and the fraction parts are multiplied after adding 1 to make mantissas. The multiplier for the mantissas consumes large power and area, and it often limits the speed since it contains many full adders in both width and depth.

B. Logarithmic Approximate Multiplier

Focusing on positive floating-point number, we simplify (6) for the sake of clarity in the following discussion.

$$A = 2^E(1.F), \quad (7)$$

When converting it into log domain, (7) becomes

$$\log_2 A = E + \log_2(1.F) \cong E.F, \quad (8)$$

where the approximated representation in the right term utilizes the approximation below.

$$\log_2(1 + x) \cong x, \text{ for } 0 \le x \le 1. \quad (9)$$

Reminding the alignment of E and F in the floating-point number format in Fig. 2, (7) and (8) indicate that a floating-point value A can be approximately regarded as $\log_2 A$ in a fixed-point format without any modification.

Logarithmic domain is beneficial in multiplication as mentioned in Section II-B since it can convert multiplication to addition. Fig. 3 illustrates the approximate multiplier which

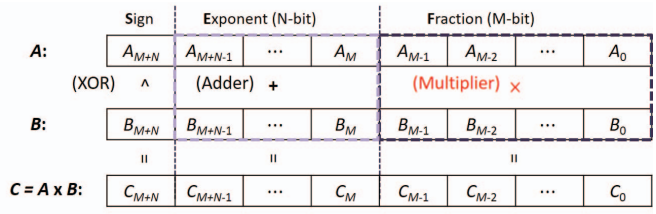


Fig. 2. Exact floating-point multiplier.

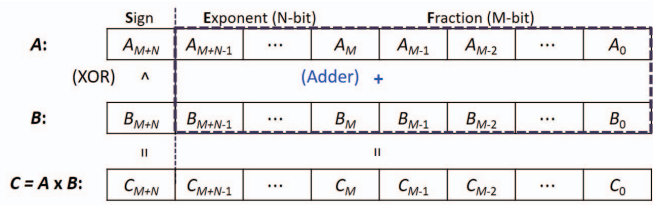


Fig. 3. Logarithmic approximate multiplier (LAM).

we name as logarithmic approximate multiplier (LAM). There is no more multiplication in LAM, just adding up the exponent and fraction parts of the two values to obtain the approximate multiplication result.

We can also analyze the approximation error of LAM by directly comparing the computations of exact multiplication and LAM. If there are two values A and B in floating-point format as $A = 2^{e_A}(1 + f_A)$ and $B = 2^{e_B}(1 + f_B)$, where f_A, f_B denote the fraction values of A, B and e_A, e_B denote the exponent values of A and B, the exact multiplication of A and B becomes:

$$A \times B = 2^{e_A+e_B}(1 + f_A)(1 + f_B), \quad (10)$$

Equation (10) may be changed after normalizing to follow the format of (6) since the non-exponent part $(1 + f_A)(1 + f_B)$ must be limited between 1 and 2. If $(1 + f_A)(1 + f_B) \ge 2$, then this value should be divided by 2 and carry 1 is delivered to exponent. Similarly, the result of LAM also depends on $f_A + f_B$. Basically, LAM(A,B) equals to $2^{e_A+e_B}(1 + f_A + f_B)$, but if $f_A + f_B \ge 1$, which means the sum of fraction values causes overflow and generates carry to the integer (exponent) part, LAM comes out to be $2^{e_A+e_B+1}(f_A + f_B)$. Comparing the expressions from varied conditions, approximate error *ErrLAM* can be derived as a conditional function of f_A & f_B:

$$ErrLAM = \begin{cases} f_A f_B, & (1 + f_A)(1 + f_B) < 2 \\ (1 - f_A)(1 - f_B)/2, & \text{otherwise.} \end{cases} \quad (11)$$

Here, both f_A and f_B are between 0 and 1, and hence *ErrLAM* is always above 0. Fig. 4 shows a contour map of *ErrLAM* as a function of f_A and f_B. The maximum value of *ErrLAM* is 3-2$\sqrt{2}$ $\cong$ 0.172 when both f_A and f_B equal to $\sqrt{2}$-1 $\cong$ 0.414. Note that this value is just the gap of fraction term, and the actual error needs to take the sum of exponent terms into consideration. Fig. 4 also indicates that, as long as either f_A or f_B is closed to 0 or 1, the approximate error is well suppressed. The advantage of LAM in terms of the speed, power, and area will be discussed in Section IV-A, and the

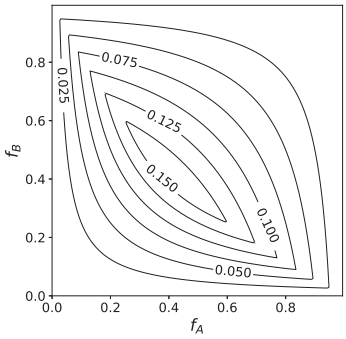


Fig. 4. Contour plot of *ErrLAM*.

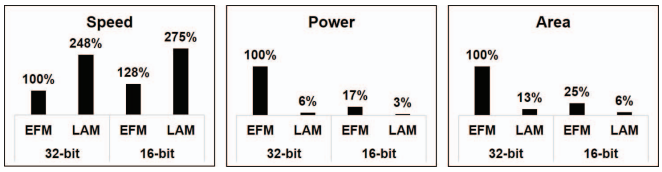


Fig. 5. Performance comparison results of LAM and EFM.

impact of the approximation error on the NN training will be investigated in Sections IV-C and IV-D.

IV. Experimental Results

This section shows the advantage of LAM as an arithmetic unit and demonstrates the impact of LAM in the NN training engine on the classification accuracy and hardware resource.

A. LAM Performance

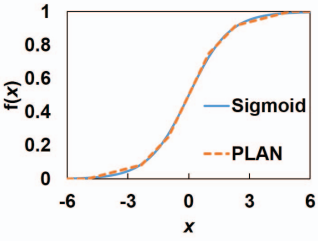
We first evaluate the performance of LAM as a multiplier by implementing two multipliers; one is LAM, and another is baseline exact floating-point multiplier, denoted as EFM. LAM is manually implemented by RTL with verlog while EFM directly adopts Synopsys DesignWare IP DW_fp_mult for functional credibility and sophisticated quality. The two designs are synthesized by Design Compiler with 45nm Nangate cell libraries. We examine their highest achievable frequencies during speed evaluation. In contrary, we align the synthesized frequency for power and area benchmarking.

The evaluation results are shown in Fig. 5. For the sake of clarity, all the results are normalized by that of 32-bit EFM. Fig. 5 shows LAM achieves 2.5X speed compared with EFM and consumes 17X less power and 8X less area in case of 32-bit floating-point expression (sign-exponent-fraction = 1-8-23). It is noted that all the power values in this paper are obtained by Design Compiler without annotating actual toggle information. The results of 16-bit version (1-5-10) are also presented. An interesting observation is that 32-bit LAM operates faster and consumes less power and area than even 16-bit EFM. Thus, LAM can improve energy efficiency remarkably. The following sections evaluate the impact of LAM on NN training with BWS in terms of the NN classification accuracy and hardware performance.

TABLE I
NN structure for testcases.

Dataset	#Neurons (I,H,O)	batch size
FOURCLASS	(2,8,2)	100
MNIST	(400,300,10)	100
HARS	(561,40,6)	40
ISOLET	(617,100,26)	60

PLAN Sigmoid f(x), f(-x) = 1 – f(x)	
Criterion	Formula
x ≥ 5	1
2.375 ≤ x < 5	x/32 + 0.84375
1 ≤ x < 2.375	x/8 + 0.625
0 ≤ x < 1	x/4 + 0.5



(a) (b)

Fig. 6. PLAN [24]. (a) Expression of PLAN and (b) original and PLAN sigmoid plots.

B. Experimental Setup for NN Training

Table I lists the datasets [14], [20]–[22] used for the experiments in this paper and the numbers of neurons in the NNs for each dataset. For the experiments, we prepare a 3-layer structure, which means 1 hidden layer, and adopt ReLU (Rectified Linear Unit, $y = \max(x,0)$) and sigmoid function ($y = 1/(1+\exp(-x))$) as the activation function of hidden and output layers, respectively. Cross entropy is chosen as the loss function since it can combine with the sigmoid function to simplify (4) to a simple linear relation, $\delta_a^i = O_a - T_a$ [23]. We also adopt stochastic gradient decent with mini-batch (update weights after accumulating $\partial\text{Loss}/\partial W$ from a batch size of training data), and apply learning rate decay (gradually decline learning rate along with iterations) as well for better convergent speed during the training phase.

Besides, the hardware implementation of non-linear sigmoid function is costly, and hence we adopt PLAN (Piecewise Linear Approximation of a Nonlinear function) sigmoid function proposed in [24]. PLAN sigmoid requires only shifter and adder, and it is friendly to hardware implementation. Fig. 6(a) lists the conditional equations used for sigmoid approximation, and Fig. 6(b) shows the curves of the original and PLAN sigmoid functions. We can see that PLAN sigmoid is well correlated with the original one.

Fig. 7 plots the diagram for our training NN hardware engine containing arithmetic units such as $\otimes$ for multipliers and $\oplus$ for adders. Forward- block computes (1), Back- block calculates (4) and (5), and then updating block computes (2) and (3). Here a, b, and c denote the numbers of neurons in Layer I, H, O, ReLU' means the derivative function of ReLU and Reg is register. The subtractor and PLAN function are annotated as adders since their functionality is similar while ReLU is drawn by $\bigcirc$. In updating block, we use the accumulators to add up the gradient δ values for the number of batch size and then update the associated weights and biases based on the accumulated gradients.

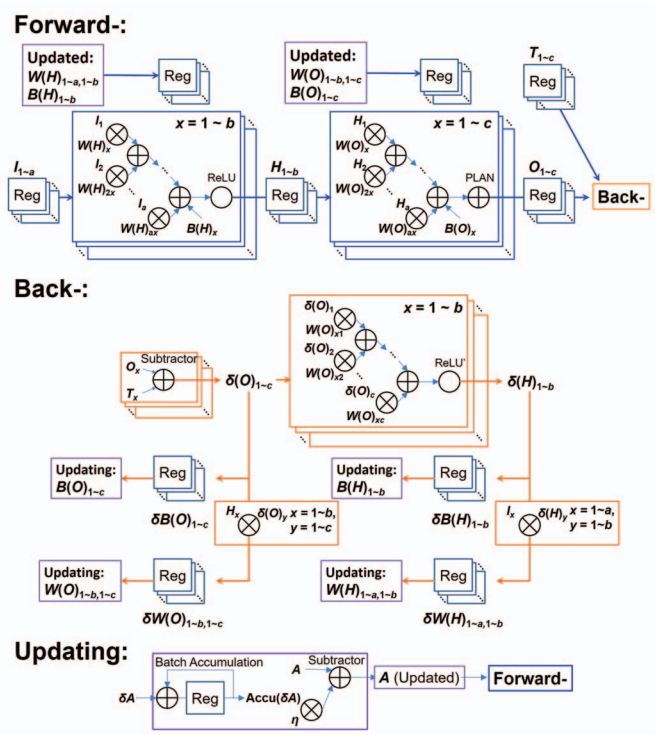


Fig. 7. Diagram of NN training hardware.

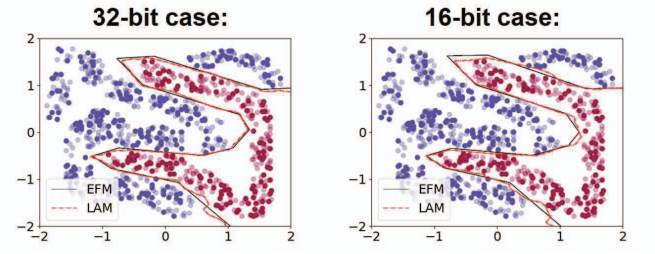


Fig. 8. Boundaries trained for 2-D classification FOURCLASS.

C. Evaluation for FOURCLASS Dataset

Now we evaluate the NN training engine with FOURCLASS dataset [14], which is a simple 2-D classification problem and its training results can be graphically illustrated with the boundary line to separate the two classified groups. Fig. 8 shows the training results of 32-bit and 16-bit floating-point cases, where the samples in the same group share the same color. Note that the training and inference are both carried out with EFM or LAM. Although the boundary lines of EFM and LAM show some discrepancy, both of them successfully distinguish the groups of samples. In 32-bit case, both EFM and LAM achieve 100% classification accuracy. In 16-bit case, which corresponds to BWS adoption, LAM experiences 0.4% classification accuracy drop while it is negligibly small. This result suggests LAM is quite compatible with BWS.

The hardware evaluation results are shown in Fig. 9, where a normalization to 32-bit EFM engine is applied similarly to

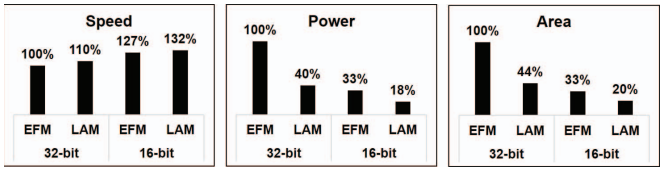


Fig. 9. Benchmarking results of adopting LAM for training FOURCLASS.

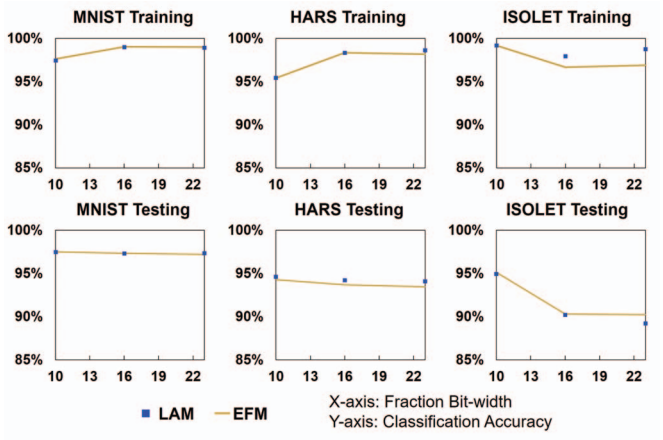


Fig. 10. Training results under BWS.

Section IV-A. The training engine adopting LAM gains 10% speed-up over EFM engine, where this speed-up is smaller than Fig. 5 since the floating-point adder is now a speed-limiting module. In contrast to speed, 2.5X (=100%/40%) power and 2.3X (=100%/44%) area efficiency enhancements are attained by LAM. These results confirm that the multiplier is so power and space demanding that reducing its computational cost improves the power and area efficiency considerably. On the other hand, when LAM + BWS (16-bit LAM) is applied, 32% speed-up, 5.5X power, and 5X area efficiency enhancements are attained, where 5% speed-up, 2.5X (=100%/18%−100%/33%) power reduction and 2X (=100%/20%−100%/33%) area saving originate from LAM. The benefits of training with LAM are quantitatively clarified.

D. Evaluation for Higher Dimensional Datasets

We further evaluate the effectiveness of LAM with other three higher dimensional datasets. Fig. 10 shows the training results for the three datasets. To test the compatibility of LAM to BWS, we varied the number of fraction bits as 10, 16 and 23 while the bit-width for exponent remains 8-bit to keep the dynamic range. The results show that only in ISOLET and 23-bit fraction case, training with LAM might be trapped into over-fitting resulting in 1% accuracy degradation. On the other hand, in other combinations of dataset and fraction bit width, LAM provides very close or even better classification accuracy compared to EFM. In overall, our test cases show that there is no reason to use EFM for training and LAM can provide a trained NN model whose classification accuracy is comparable whereas [18] uses EFM at the latter stages of training for the

TABLE II
STATISTICS FOR USAGE OF MULTIPLIERS, ADDERS, AND REGISTERS IN HARDWARE.

Usage			
Dataset	#Multipliers	#Adders	#Registers
FOURCLASS	122	128	142
MNIST	372,310	372,340	371,050
HARS	68,326	68,344	69,352
ISOLET	195,626	195,704	194,664

TABLE III
PERFORMANCE EVALUATION RESULTS FOR TRAINING LARGER-SIZE OF NN (PROJECTED FROM FOURCASE RESULT).

Power (Normalized)			
32-bit		19-bit	
EFM	LAM	EFM	LAM
100%	40%	35%	19%

same datasets. Our results reveal that EFM is not necessary at least for the datasets used in our experiments.

For estimating the hardware improvement thanks to LAM, a simple projection is performed based on the FOURCLASS results. The power consumption of the training engine can be estimated by the numbers of arithmetic units and registers. The power values of each arithmetic unit and register are obtained from the synthesis results. The remaining thing is counting the number of arithmetic units and registers. We can refer to Fig. 7 for counting the numbers of multipliers, adders, and registers. The usages are dominated by weights and biases once the NN size grows up. Also, according to Fig. 7, the usage of multipliers, adders, and registers for weights and biases are nearly even (closed to 1:1:1). Table II lists our statistics for the four datasets, proving that the usage of three parts closed to 1:1:1. As the number of registers shrinks in higher dimensional datasets, the power improvement rate by LAM for training a larger NN-size structure is roughly equal or slightly larger than that of FOURCLASS. In principle, we can conservatively estimate the least improvement rate based on the FOURCLASS benefit. Table III shows the projected values. LAM in 32-bit attains 2.5X and in 19-bit case (BWS + LAM) achieves 5.2X power efficiency, where 2.3X originates by LAM.

V. CONCLUSIONS

Floating-point computation is often adopted in NN hardware training engine, and then we evaluated whether approximate floating-point multiplier could be used in NN training for achieving higher energy efficiency. We focused on logarithmic approximate multiplier (LAM) incorporating with bit-width scaling (BWS) to reduce primary MAC computation complexity. The experimental results show that training NN with LAM can achieve 10% speed-up and 2.5X power reduction in addition to 2.3X area saving. Even when training with LAM + BWS, there is no more than 1% accuracy discrepancy compared with exact multiplier, where LAM + BWS outperforms, rather than degrades, the accuracy frequently. As for the hardware performance, 5.2X power efficiency is attained, where 2.3X originates by LAM. Our future work includes evaluating the applicability of LAM to further extended or other types of NN and other larger datasets.

ACKNOWLEDGMENTS

This work is supported by Grant-in-Aid for Scientific Research (B) from JSPS under Grant 19H04079.

REFERENCES

[1] G. L. Pedro *et. al*, "Edge-centric Computing: Vision and Challenges," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 5, pp. 37–42, Oct. 2015.
[2] P. Grulich *et. al*, "Collaborative edge and cloud neural networks for real-time video processing," in *VLDB Endowment*, vol. 11, pp. 2046–2049, 2018.
[3] Y. Huang *et. al*, "When deep learning meets edge computing," in *ICNP*, 2017, pp, 1–2.
[4] A. Krizhevsky *et. al*, "Imagenet classification with deep convolutional neural networks," in *NIPS*, '12 2012, 1097-1105.
[5] J. Y. F. Tong *et. al*," Reducing power by optimizing the necessary precision/range of floating-point arithmetic," *IEEE TVLSI*, vol. 8, no. 3, pp. 273–286, 2000.
[6] S. Venkataramani *et. al*," Axnn: Energy-efficient neuromorphic systems using approximate computing." in *ISLPED*, August 2014, pp. 27–32.
[7] Q. Zhang *et. al*, "Approxann: An approximate computing framework for artificial neural network." in *DATE*, March 2015, pp. 701–706.
[8] J. Kung *et. al*," A power-aware digital feedforward neural network platform with backpropagation driven approximate synapses," in *ISLPED*, July 2015, pp. 85–90.
[9] J. David *et. al*," Training deep neural networks with low precision multiplications," *arXiv:1412.7024*, 2014.
[10] N. Wang *et. al*," Training deep neural networks with 8-bit floating point numbers," In *NIPS*, pp. 7685–7694, December 2018.
[11] D. Kim *et. al*," A power-aware digital multilayer perceptron accelerator with on-chip training based on approximate computing," *IEEE TETIC*, vol. 5, no. 2, pp. 164–178, February 2017.
[12] M. Horowitz. Energy table for 45nm process. Stanford VLSI wiki.
[13] M. Gao *et. al*," Energy efficient runtime approximate computing on data flow graphs," In *ICCAD*, November 2017, pp. 444–449.
[14] C. Chang and C. Lin. Fourclass, 1996. URL https://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/binary.html.
[15] S. Haykin. Neural Networks and Learning Machines. PEARSON Education 3 edition, 2008.
[16] K. Ueyoshi *et. al*," QUEST: A 7.49TOPS multi-purpose log-quantized DNN inference engine stacked on 9MB 3D SRAM using inductive coupling technology in 40nm CMOS," in *ISSCC*, February 2018, pp. 186–196.
[17] U. Lotrič and P. Bulić," Applicability of approximate multipliers in hardware neural networks," *Neurocomputing*, vol. 96, pp. 57–65, November 2012.
[18] M. Imani *et. al*," RMAC: Runtime configurable floating point multiplier for approximate computing," in *ISLPED*, July 2018.
[19] J. Johnson, "Rethinking floating point for deep learning," *arXiv:1811.01721v1*, 2018.
[20] Y. LeCun, C. Cortes, and C. J.C. Burges, "Mnist", http://yann.lecun.com/exdb/mnist, (1998).
[21] Uci machine learning repository, "Human Activity Recognition Using Smartphones Data Set," http://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones (2012).
[22] Uci machine learning repository, "ISOLET Data Set," http://archive.ics.uci.edu/ml/datasets/ISOLET (1994)
[23] K. Murphy. Murphy, K. (2012). Machine learning. Cambridge, Mass.: MIT Press.
[24] A. Tisan *et. al*," Digital implementation of the sigmoid function for FPGA circuits," *Acta Technica Napocensis Electronics and Telecommunications*, 2009.