TaxoNN: A Light-Weight Accelerator for Deep Neural Network Training

Reza Hojabr[†§], Kamyar Givaki[†,1], Kossar Pourahmadi[†,1], Parsa Nooralinejad[†,1],
Ahmad Khonsari[†‡], Dara Rahmati[‡], M. Hassan Najafi[§]
[†]School of Electrical and Computer Engineering, University of Tehran, Tehran, Iran
[‡]School of Computer Sciences, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran
[§]School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, LA, USA
{r.hojabr,givakik,kosar.pourahmadi,p.nooralinejad}@ut.ac.ir, {ak,dara.rahmati}@ipm.ir, najafi@louisiana.edu

***Abstract*—Emerging intelligent embedded devices rely on Deep Neural Networks (DNNs) to be able to interact with the real-world environment. This interaction comes with the ability to retrain DNNs, since environmental conditions change continuously in time. Stochastic Gradient Descent (SGD) is a widely used algorithm to train DNNs by optimizing the parameters over the training data iteratively. In this work, first we present a novel approach to add the training ability to a baseline DNN accelerator (inference only) by splitting the SGD algorithm into simple computational elements. Then, based on this heuristic approach we propose *TaxoNN, a light-weight accelerator for DNN training*. TaxoNN can easily tune the DNN weights by reusing the hardware resources used in the inference process using a time-multiplexing approach and low-bitwidth units. Our experimental results show that TaxoNN delivers, on average, 0.97% higher misclassification rate compared to a full-precision implementation. Moreover, TaxoNN provides 2.1× power saving and 1.65× area reduction over the state-of-the-art DNN training accelerator.**

I. Introduction

Driven by the availability of large datasets, deep learning applications are increasingly growing in various fields such as speech recognition, computer vision, control and robotics. Meanwhile, time-consuming computations of DNNs and the need for power-efficient hardware implementations have made the semiconductor industry to rethink the customized hardware for deep learning algorithms. As a result, DNN hardware accelerators have been emerged as a promising solution to tackle efficient implementation of these compute-intensive and energy-hungry algorithms [1]–[3].

Employing deep learning algorithms in building intelligent embedded devices that interact with the environment requires customized accelerators that support both training and inference processes. For instance, in deep reinforcement learning algorithms, an agent uses a neural network (NN) to predict the proper action regarding the current state and the reward obtained from the environment. In such algorithms that a NN-based agent is interacting with the environment and the environmental conditions are changing continuously, the training process is performed repeatedly to tune the agent. Implementing the prohibitive computations of the training process seeks an efficient yet low-power trainable architecture. Although the recently proposed accelerators have significantly improved the performance of the inference process [1], there is still a growing demand for low-power DNN accelerators that support both training and inference processes.

Training process can be interpreted as an optimization problem that aims to minimize an objective function (network error function) by finding a set of network parameters. Stochastic Gradient Descent (SGD) is a common approach to solve this optimization problem [4]. SGD moves towards the optimum point in the decreasing direction of the error function's gradient. Calculating the gradients during SGD requires high-cost hardware resources which cannot be provided in embedded devices with limited power and area budget. Therefore, adding the training capability to the conventional inference-only accelerators is a challenging issue and needs a complete rethink. This work seeks a solution to enable inference-only accelerators to perform SGD computations with minimum hardware resources.

Relying on the approximate nature of NNs, several methods have been proposed replacing floating point units of NN with low-bitwidth ones [5]–[8]. Prior work have shown that employing low-bitwidth operations in DNN accelerators can result in a substantial power and area saving while maintaining the quality of the results [4], [5], [7]. While it is more common to employ low-bitwidth data in the inference process, recent works have demonstrated that the training process (i.e., SGD) can also be performed using quantized parameters [4]. Our observations confirm that the desired accuracy can be achieved, without sacrificing the network convergence, when using low-bitwidth operations during the training process. An important point, however, is that the required bitwidth can vary from layer to layer. As we get closer to final layers of DNNs, the extracted features become more valuable. While the early layers produce satisfying results with small bitwidths, a more precise computation is necessary in the final layers. Leveraging this observation, by proper adjustment of the bitwidth in each layer we can reduce a significant amount of power and area while maintaining the quality of the results.

This work proposes a novel low-cost accelerator that supports both training and inference processes. We first propose a novel method to split the SGD algorithm into smaller computational elements by unrolling this compute-intensive algorithm. Using our proposed method, a fine-grained inter-layer parallelism can be used in the training process. We then leverage this method and introduce TaxoNN, *a Light-Weight Accelerator for DNN training* which is able to perform training and inference processes using shared resources. TaxoNN utilizes an optimized datapath in a pipelined manner that minimizes the hardware cost. We show how bitwidth optimization in different layers of NN can reduce the implementation cost while keeping the quality of the results. In summary, the main contributions of this work are as follows:

i. We propose a novel heuristic method to minimize the implementation cost of the SGD algorithm by unrolling its computations. The proposed method reduces the hardware cost by time-division multiplexing (TDM) of multiply-and-accumulate (MAC) units.
ii. We introduce an accelerator for DNN training, called TaxoNN, that supports training and inference using this method. TaxoNN parallelizes the SGD algorithm while minimizes the required arithmetic units.
iii. We evaluate TaxoNN in terms of network convergence, power consumption, and area using low-bitwidth computational units for different layers. Our experimental results show that TaxoNN offers 2.1× power and 1.65× area saving at the cost of, on average, 0.97% higher misclassification rate compared to the full-precision implementation.

II. Related Work

Plenty of work have introduced specialized accelerators for deep learning [2], [3], [9]. Motivated by the processing characteristics of DNNs, Eyeriss [1] introduced a novel data-flow to maximize data reuse between neural Processing Elements (PEs) and hence to minimize the energy consumption wasted on

[1] These authors contributed equally.

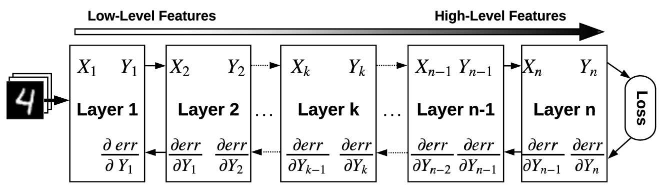

Low-Level Features
High-Level Features
X_1 Y_1
Layer 1
$\frac{\partial err}{\partial Y_1}$
X_2 Y_2
Layer 2
$\frac{\partial err}{\partial Y_1}$ $\frac{\partial err}{\partial Y_2}$
X_k Y_k
Layer k
$\frac{\partial err}{\partial Y_{k-1}}$ $\frac{\partial err}{\partial Y_k}$
X_{n-1} Y_{n-1}
Layer n-1
$\frac{\partial err}{\partial Y_{n-2}}$ $\frac{\partial err}{\partial Y_{n-1}}$
X_n Y_n
Layer n
$\frac{\partial err}{\partial Y_{n-1}}$ $\frac{\partial err}{\partial Y_n}$
Loss

Fig. 1. Back-propagation in the layers of a DNN.

data movements. As there are various types of layers in DNNs (convolutional, pooling and fully connected), MAERI [10] and FlexFlow [11] proposed new design methodologies to enable flexible data-flow mapping over neural accelerators. Moreover, eliminating unnecessary multiplications in sparse layers [8], [12], [13] and computation reuse [6], [14], [15] are promising solutions to reduce the cost of DNN accelerators.

Recently, replacing full-precision operations with low-bitwidth ones has been used as an effective approach to save energy consumption of DNNs [5], [16], [17]. Experimental observations have shown that the approximate nature of DNNs makes them tolerable to the quantization noise [5], [6], [18]. Hence, costly floating-point arithmetic units are replaced by fixed-point ones at no considerable accuracy loss. Bit Fusion [5] presents a bit-level flexible accelerator that dynamically sets the bitwidth to minimize the computation cost.

While the focus of most prior work has been on developing high-performance architectures for the inference process, some recent work proposed accelerators for training DNNs [19]–[21]. TIME [20] and Pipelayer [19] utilized Process-In-Memory (PIM) techniques to accelerate the training process. Performing the operations near memory helps to alleviate the data movement overhead during the DNN computations. However, to the best of our knowledge, no hardware architecture and datapath have been developed to reduce the processing time of the SGD algorithm by exploiting parallelism in its heavy computations. Some recent work have also shown that training can be performed using low-bitwidth gradients [4], [22]–[24]. In this work, we minimize the overall cost of the proposed accelerator, TaxoNN, by proper adjustment of bitwidth in each layer of the network.

III. Motivation and Background

The training process has the most prominent role in designing an accurate DNN. The underlying principle in training methods arises from what occurs in the human brain. To distinguish a certain object, a set of various pictures demonstrating the object in different gestures are fed to the network in an iterative manner. The network gradually learns to identify an object by extracting its features in multiple iterations. By comparing the output to the desired result, the network learns how to change the parameters. This procedure continues until the network finds the best weights that maximize the recognition accuracy.

From mathematical point of view, training procedure is performed by an error Back-Propagation (BP) method. As depicted in Fig. 1, an input data is fed to the network and forwarded through the layers. The produced output is fed to a loss function to calculate the gradient of the error. The computed gradient is then back-propagated through the layers to update the weights. During back-propagation, the gradient of the error tends gradually to zero. This method is called Gradient Descent. Eq. 1 shows how the weights in layer i are updated by the gradient. Learning rate is shown with α which determines the rate of network convergence by controlling the impact of gradients during the training process. Due to large amount of data, feeding all inputs to the network is very time-consuming. Therefore, a subset of data is picked up randomly in each iteration to train the network. This method, called Stochastic Gradient Descent (SGD), is the most common approach to train DNNs.

$$W_i = W_i - \alpha \frac{\partial error}{\partial W_i} \tag{1}$$

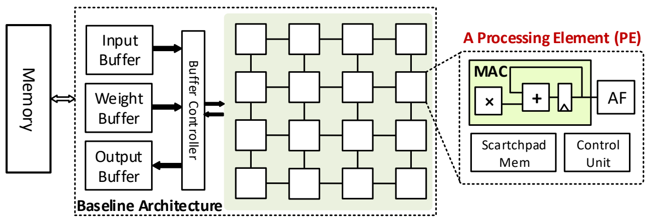


Fig. 2. The baseline architecture of TaxoNN to execute the inference of DNNs

Training often takes a long time to be completed as its processing time is directly proportional to the number of layers. Conventional DNNs are composed of a large number of layers (may even more than a thousand layers). Convolutional layers constitute the most portion of the computation load in DNNs. These layers are obligated to extract the features of the input data. Normally, the early layers extract general features that can be used in distinguishing any object. As we get closer to final layers, we extract more valuable features that help to recognize specific objects.

IV. Proposed Architecture

Due to limited processing resources in embedded systems, TaxoNN aims to train the network by reusing hardware resources used in inference to minimize the hardware cost. In what follows, we describe the micro-architecture of TaxoNN.

A. *Inference Architecture*

The baseline architecture of TaxoNN, designed to perform the inference process, is shown in Fig. 2. Similar to the state-of-the-art accelerator [1], it is composed of a 2D array of Processing Elements (PEs) used in both convolutional and fully-connected (FC) layers. In general, the output of each neuron (a.k.a filter in the convolutional layers) is achieved by a weighted summation, $y = f(\sum_{i=0}^{i=k} x_i w_i)$, where x_i is the input vector, w_i is the weight vector and f denotes the activation function. The activation function is typically Sigmoid in FC layers and ReLU (Rectified Linear Unit) in convolutional layers. In TaxoNN, each layer is equipped with input/output buffers that fetch/store the input/output data. Each PE can access to the weight buffer that holds the weight vector. To decrease the number of data accesses to the input buffer, the fetched values are forwarded through the PEs in a pipelined manner. PEs are equipped with a local scratchpad memory to hold the weights and partial results.

In the FC layer L_i, the required time to complete the computations of the neurons is $N_{i-1} + N_i$ clock cycles where N_{i-1} and N_i are the number of neurons in the $(i-1)^{th}$ and i^{th} layers, respectively (provided that we have N_i PEs). In the convolutional layer L_j, where the input data size is $h \times w \times d$ and the filter size is $k \times k \times d$, the convolution is achieved in $kd(w-k)(h-k)$ clock cycles. Similar to the *Row-Stationary* dataflow proposed in Eyeriss [1], each compute lane in the PE array is dedicated to a single row of the filter to maximize the data reuse in the architecture.

B. *Simplifying SGD algorithm*

As mentioned in Eq. (1), weights are updated in each layer by subtracting the term $\alpha\frac{\partial error}{\partial W_i}$ from their current value. The first step towards enabling training in the porposed accelerator is to simplify the term $\frac{\partial error}{\partial W_i}$ to implement it with the minimum hardware resources. Leveraging the chain rule we can partition $\frac{\partial error}{\partial W_i}$ into three small parts as follows:

$$\frac{\partial error}{\partial W_i} = \frac{\partial error}{\partial Y_{i+1}} \times \frac{\partial Y_{i+1}}{\partial Y_i} \times \frac{\partial Y_i}{\partial W_i} \tag{2}$$

where Y_i is the output of the i^{th} layer. Note that all the notations are written in the matrix form. The terms $\frac{\partial Y_{i+1}}{\partial Y_i}$ and $\frac{\partial Y_i}{\partial W_i}$ can further be expanded as follows:

$$\frac{\partial Y_{i+1}}{\partial Y_i} = \frac{\partial f_{i+1}(W_{i+1}Y_i)}{\partial Y_i} = W_{i+1}f'_{i+1}(W_{i+1}Y_i) \quad (3)$$

$$\frac{\partial Y_i}{\partial W_i} = \frac{\partial f_i(W_iY_{i-1})}{\partial W_i} = Y_{i-1}f'_i(W_iY_{i-1}) \quad (4)$$

where $f_{i+1}(.)$ denotes the activation function of the $(i+1)^{th}$ layer and f' refers to the derivation of the activation function. Combining Eq. (3) and Eq. (4) leads to Eq. (5):

$$\frac{\partial error}{\partial W_i} = \frac{\partial error}{\partial Y_{i+1}} \times f'_{i+1} \times W_{i+1} \times f'_i \times Y_{i-1} \quad (5)$$

We define G_{i+1} as the product of the first two terms in the right hand side (RHS) of Eq. (5) that is computed in the $(i+1)^{th}$ layer and passed backward to the i^{th} layer.

$$G_{i+1} = \frac{\partial error}{\partial Y_{i+1}} \times f'_{i+1} \quad (6)$$

Clearly, the input of i^{th} layer is the output of $(i-1)^{th}$ layer ($X_i = Y_{i-1}$, where X_i is the input of i^{th} layer). As a result, we can rewrite Eq. (2) as follows:

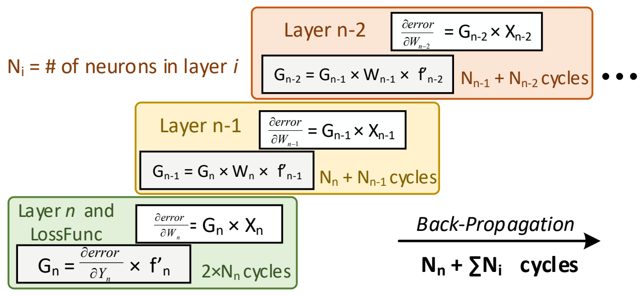
$$\frac{\partial error}{\partial W_i} = G_{i+1} \times W_{i+1} \times f'_i \times X_i \quad (7)$$

To facilitate the hardware implementation, Eq. (7) can be split into Eq. (8) and Eq. (9). As shown in Eq. (9), multiplying Eq. (8) by the input of i^{th}layer, X_i, results in term $\frac{\partial error}{\partial W_i}$ in Eq. (1).

$$G_i = G_{i+1} \times W_{i+1} \times f'_i \quad (8)$$

$$\frac{\partial error}{\partial W_i} = G_i \times X_i \quad (9)$$

Consequently, G_i has a key role in the training process. As shown in Eq. (8), G_i is achieved recursively by calculating in each layer and passing backward to the previous layer. We use this unrolling method to distinguish between the operations in the SGD and to properly map them to the hardware resources.


Layer n-2
$\frac{\partial error}{\partial W_{n-2}} = G_{n-2} \times X_{n-2}$
N_i = # of neurons in layer i
$G_{n-2} = G_{n-1} \times W_{n-1} \times f'_{n-2}$
$N_{n-1} + N_{n-2}$ cycles
Layer n-1
$\frac{\partial error}{\partial W_{n-1}} = G_{n-1} \times X_{n-1}$
$G_{n-1} = G_n \times W_n \times f'_{n-1}$
$N_n + N_{n-1}$ cycles
Layer n and LossFunc
$\frac{\partial error}{\partial W_n} = G_n \times X_n$
$G_n = \frac{\partial error}{\partial Y_n} \times f'_n$
$2 \times N_n$ cycles
Back-Propagation
$N_n + \sum N_i$ cycles
Fig. 3. Timing diagram of the training process in TaxoNN.

C. Training Architecture

To implement the BP computations, the baseline architecture must be modified by adding some simple logical components. Fig. 4 illustrates the micro-architecture of the proposed PEs in TaxoNN. The gray components are added to the baseline architecture to enable training. To minimize the needed resources, we employ a TDM approach to improve the resource utilization of the main components (e.g., the multipliers) in the datapath. In what follows, we describe each component in detail.

Multiplexers. As depicted in Fig. 4, the architecture is equipped with three multiplexers to enable TDM. The inference process is still performed using the main multiplier. All the needed parameters of Eq. (8) and Eq. (9) can be provided by a proper timing management of MUX1, MUX2 and the multiplier as follows: ❶ MUX1 provides G_{i+1} and MUX2 provides W_{i+1}. Then, $G_{i+1} \times W_{i+1}$ will be calculated and stored in register R1. ❷ MUX1 forwards f'_i and MUX2 forwards $G_{i+1} \times W_{i+1}$ to the multiplier to calculate $G_i = G_{i+1} \times W_{i+1} \times f'_i$. ❸ MUX1 forwards X_i to multiply it by G_i and hence produce $\frac{\partial error}{\partial W_i}$.

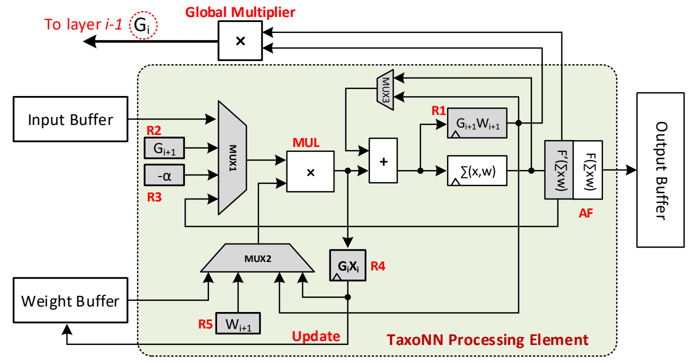


Fig. 4. The micro-architecture of a PE in TaxoNN with the training capability.

❹ Finally, the result is multiplied by the learning rate, α, that is already stored in a register behind MUX1.

In this manner, $-\alpha\frac{\partial error}{\partial W_i}$ as the most important parameter for updating the weights, is prepared through a TDM of the PE's multiplier. Note that G_{i+1} and W_{i+1} have been provided and sent to the current layer by the $(i+1)^{th}$ layer. Since all the computations are done in the matrix form, calculating $G_{i+1} \times W_{i+1}$ needs N_{i+1} cycles where N_{i+1} is the number of neurons in the $(i+1)^{th}$ layer. After each multiplication, the result is accumulated in the corresponding register.

Activation Function. The activation function unit of the baseline architecture (Fig. 2) is equipped with an internal unit to calculate the derivation of the activation functions. There are three types of activation functions which are commonly used in the modern DNNs: ReLU, Sigmoid and $tanh$. The derivation of sigmoid $\sigma(x)$ can be easily achieved by $\sigma'(x) = \sigma(x)(1-\sigma(x))$. Also, $tanh$ is simply achieved from $\sigma(x)$ as $tanh(x) = 2\sigma(2x) - 1$ and consequently, $tanh'(x)$ can be achieved as $tanh'(x) = 4\sigma'(2x)$. Finally, the derivation of the ReLU is 0 for negative inputs and 1 for positive ones.

Global Multiplier. In TaxoNN, each layer i has a single global multiplier to produce G_i. This multiplier is shared between all the neurons of the layer. Therefore, the number of cycles needed to produce G_i equals the number of neurons in that layer.

Consequently, the following components are added to the baseline PE: *(i)* three multiplexers, *(ii)* five registers (located in the scratchpad memory to hold the intermediate values during training), and *(iii)* activation function's derivation unit. The overhead cost of these components will be discussed in Section V.

D. Timing and Pipeline

TaxoNN benefits from an optimized and pipelined architecture. Fig. 3 shows the timing diagram of the training process composed of the forwarding phase followed by the error BP and weight updating. As mentioned in Section IV-B, G_i is the main precedence for calculating $-\alpha\frac{\partial error}{\partial W_i}$. In layer i, G_i is a vector of size N_i, where N_i is the number of neurons in that layer. Whenever $G_{i,1}$ (the first element of the matrix G_i) becomes ready, it will be sent to the previous layer, L_{i-1}, that needs the elements of G_i to calculate $\frac{\partial error}{\partial W_{i-1}}$. Therefore, producing $\frac{\partial error}{\partial W_i}$ has a timing overlap with producing G_i in the $(i)^{th}$ layer. Leveraging this pipelining, TaxoNN performs an iteration of the BP in $N_n + \sum_{i=1}^{i=n} N_i$ clock cycles, where n is the number of layers. The extra N_n is for the computations of the loss function and is equal to the processing time of the last layer (n^{th} layer).

V. Performance Evaluation

We used the LeNet architecture [25] to evaluate the performance of TaxoNN using MNIST, CIFAR10 and SVHN datasets. We extracted the results of full-precision computations using TensorFlow. In what follows, we analyze TaxoNN in terms of accuracy, network convergence, and hardware cost.

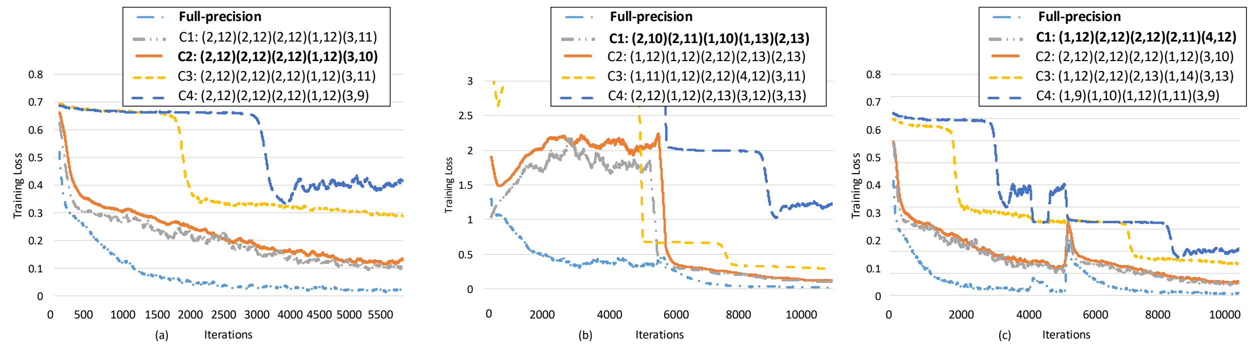


Fig. 5. The network loss: (a) MNIST and (b) CIFAR10, (c) SVHN.

TABLE I
THE NETWORK ACCURACY (%) OF DIFFERENT BITWIDTH VERSUS THE FLOATING-POINT IMPLEMENTATION.

Dataset	Precision per Layer (I,F)	TaxoNN Accuracy	Full-precision Accuracy
MNIST	(2,12)(2,12)(2,12)(1,12)(3,10)	99.1	99.4
CIFAR10	(2,10)(2,11)(1,10)(1,13)(2,13)	84.1	85.4
SVHN	(1,12)(2,12)(2,12)(2,11)(4,12)	94.7	96.0

TABLE II
THE AREA ($um^2 \times 10^3$) OF A PROCESSING ELEMENT OF TAXONN VERSUS THAT OF THE BASELINE ARCHITECTURE.

Bitwidth	21	20	19	18	17	16	15	14	13	Average Area Overhead
Eyeriss	14.3	13.1	11.8	11.1	10.6	10.1	9.7	9.0	8.1	
TaxoNN	15.5	14.3	12.9	12.1	11.7	11.2	10.6	9.9	9.0	
Overhead	8.3%	9.2%	9.1%	8.6%	10.0%	10.8%	8.8%	9.8%	10.5%	9.5%

TABLE III
THE POWER CONSUMPTION (mW) OF A PROCESSING ELEMENT OF TAXONN VERSUS THAT OF THE BASELINE ARCHITECTURE.

Bitwidth	21	20	19	18	17	16	15	14	13	Average Power Overhead
Eyeriss	4.54	4.48	4.42	4.31	4.22	4.10	3.98	3.88	3.75	
TaxoNN	4.84	4.78	4.70	4.65	4.49	4.31	4.15	4.13	4.04	
Overhead	6.5%	6.7%	6.2%	7.9%	6.5%	5.2%	4.3%	6.5%	7.7%	6.4%

TABLE IV
POWER AND AREA REDUCTION OF TAXONN COMPARED TO THE FULL-PRECISION TRAINING IMPLEMENTATION

Dataset	Power Reduction	Area Reduction
MNIST	2.1×	1.7×
CIFAR10	2.3×	1.8×
SVHN	1.9×	1.5×

A. *Bitwidth Optimization*

Fig. 5 demonstrates the network loss during different iterations of the training process using TaxoNN with different bitwidths (optimized for each layer) versus the case of training using the full-precision implementation. MNIST and SVHN are two datasets consist of 28×28 images from hand-written digits (0..9) and house numbers, respectively. CIFAR10 is a set of 32×32 color images in 10 classes. The training performance is evaluated over 10,000 test images and the network accuracies are extracted by TensorFlow.

The results shown in Fig. 5 confirm that the low-bitwidth training can have a comparable accuracy for the same number of iterations. The optimum bitwidth for each layer can be different from other layers. For each dataset, we evaluated the network accuracy for a large number of design points. Fig. 5 shows four design points for each dataset, each point representing the adopted precision for a layer. The number representation (I,F) indicates a fixed-point number with I bits for the integer part and F bits for the fractional part.

For instance, during the training of MNIST, the configuraction set C2 converges similar to the floating-point implementation. Lower bitwidths, however, may cause under-fitting. The speed of the network convergence gets reduced as the bitwidth gets shorter. This phenomenon implies that the network confidence is directly related to the precision of the arithmetic operations. An observation is that there is a lower bound that limits the bitwidth of the training. The bitwidths lower than these thresholds cause under-fitting while the bitwidths higher than them are not necessary and will only cost additional area and power consumption.

Table I shows the neural network accuracy when using TaxoNN with various bitwidths compared to the case of using 32-bit floating-point implementation. Decreasing the bitwidth down to the identified numbers in each configuration set has no considerable impact on the network accuracy. Using a bitwidth lower than the specified one in the configuration sets results in a dramatic accuracy loss as the network can not converge to the desired point.

B. *Hardware Cost*

To evaluate the hardware cost of the proposed architecture, we implemented TaxoNN in RTL Verilog and synthesized using the Synopsys Design Compiler with a 45-nm gate library. Table II shows the area cost of the synthesized TaxoNN PE (which supports training) versus the state-of-the-art accelerator, Eyeriss [1], as the baseline architecture (without supporting training). The average area overhead compared to Eyeriss is less than 10%. The activation functions' derivation unit contributed the most portion of this area overhead and the other units such as the multiplexers had a negligible cost.

Table III shows the power consumption of TaxoNN PE compared to that of the Eyeriss architecture (without supporting training) using fixed-point operations. As can be seen, the power consumption is not a concern for TaxoNN due to its pipelines and regular structure. The synthesis results show that the power consumption is, on average, less than 7% over that of the baseline architecture. Table IV summarizes the overall power and area improvement offered by TaxoNN with low-bitwidth operations compared to the full-precision architecture.

Moreover, the processing cycles needed for the back-propagation is relatively close to that of feed-forward. Therefore, TaxoNN improves the energy consumption of the training process. These privileges make TaxoNN an appealing accelerator for embedded devices with tight energy constraints.

VI. CONCLUSIONS

In this work, we proposed a light-weight DNN accelerator, called TaxoNN, that supports both inference and training processes. We introduced a novel method to unroll and parallelize the SGD computations. Using this method, we proposed a fine-grained and optimized datapath to perform the matrix operations of SGD. TaxoNN considerably reduces the computation resources required in DNN training by reusing the arithmetic units used in the inference. We evaluated TaxoNN with low-bitwidth operations for each layer. The proposed accelerator offers 1.65× area and 2.1× power saving at the cost of, on average, 0.97% higher misclassification rate compared to the full-precision implementation.

References

[1] Y.-H. Chen, T. Krishna, J. Emer *et al.*, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, 2016, pp. 262–263.

[2] T. Chen, Z. Du, N. Sun *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM Sigplan Notices*, vol. 49, no. 4, pp. 269–284, 2014.

[3] Z. Du, R. Fasthuber, T. Chen *et al.*, "Shidiannao: Shifting vision processing closer to the sensor," in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 2015, pp. 92–104.

[4] S. Zhou, Y. Wu, Z. Ni *et al.*, "DoReFa-Net: Training low bitwidth convolutional NNs with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[5] H. Sharma, J. Park, N. Suda *et al.*, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[6] A. Yasoubi, R. Hojabr, and M. Modarressi, "Power-efficient accelerator design for neural networks using computation reuse," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 72–75, 2017.

[7] P. Judd, J. Albericio, T. Hetherington *et al.*, "Stripes: Bit-serial deep neural network computing," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

[8] S. Zhang, Z. Du, L. Zhang *et al.*, "Cambricon-x: An accelerator for sparse neural networks," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

[9] B. Reagen, P. Whatmough, R. Adolf *et al.*, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Computer Architecture (ISCA), ACM/IEEE 43rd Annual International Symp. on*, 2016, pp. 267–278.

[10] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018, pp. 461–475.

[11] W. Lu, G. Yan, J. Li *et al.*, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symp. on*, pp. 553–564.

[12] A. Parashar, M. Rhu, A. Mukkara *et al.*, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Computer Architecture (ISCA), ACM/IEEE 44th Annual International Symposium on*, 2017, pp. 27–40.

[13] J. Albericio, P. Judd, T. Hetherington *et al.*, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 1–13.

[14] M. Riera, J.-M. Arnau, and A. Gonzalez, "Computation reuse in dnns by exploiting input similarity," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018.

[15] K. Hegde, J. Yu, R. Agrawal *et al.*, "Ucnn: Exploiting computational reuse in deep neural networks via weight repetition," *arXiv preprint arXiv:1804.06508*, 2018.

[16] D. J. Pagliari, E. Macii, and M. Poncino, "Dynamic bit-width reconfiguration for energy-efficient deep learning hardware," in *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM, 2018, p. 47.

[17] R. Hojabr, K. Givaki, S. R. Tayaranian *et al.*, "Skippynn: An embedded stochastic-computing accelerator for convolutional neural networks," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[18] V. Mrazek, S. S. Sarwar, L. Sekanina *et al.*, "Design of power-efficient approximate multipliers for approximate artificial neural networks," in *Proceedings of the 35th International Conference on Computer-Aided Design*, 2016, p. 81.

[19] L. Song, X. Qian, H. Li *et al.*, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 541–552.

[20] M. Cheng, L. Xia, Z. Zhu *et al.*, "Time: A training-in-memory architecture for memristor-based deep neural networks," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 26.

[21] J. Li, G. Yan, W. Lu *et al.*, "Tnpu: an efficient accelerator architecture for training convolutional neural networks," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, 2019, pp. 450–455.

[22] I. Hubara, M. Courbariaux, D. Soudry *et al.*, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[23] S. Wu, G. Li, F. Chen *et al.*, "Training and inference with integers in deep neural networks," *arXiv preprint arXiv:1802.04680*, 2018.

[24] U. Köster, T. Webb, X. Wang *et al.*, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *Advances in Neural Information Processing Systems*, 2017, pp. 1742–1752.

[25] Y. LeCun, L. Bottou, Y. Bengio *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.