# Exact Neural Networks from Inexact Multipliers via Fibonacci Weight Encoding

William Andrew Simon*, Valérian Ray, Alexandre Levisse*,
Giovanni Ansaloni*, Marina Zapater†*, and David Atienza*
*Embedded Systems Laboratory (ESL), Swiss Federal Institute of Technology Lausanne (EPFL)
†University of Applied Sciences Western Switzerland (HEIG-VD / HES-SO)
Email: {william.simon, alexandre.levisse, giovanni.ansaloni, marina.zapater, david.atienza}@epfl.ch

***Abstract*—Edge devices must support computationally demanding algorithms, such as neural networks, within tight area/energy budgets. While approximate computing may alleviate these constraints, limiting induced errors remains an open challenge. In this paper, we propose a hardware/software co-design solution via an inexact multiplier, reducing area/power-delay-product requirements by 73/43%, respectively, while still computing exact results when one input is a Fibonacci encoded value. We introduce a retraining strategy to quantize neural network weights to Fibonacci encoded values, ensuring exact computation during inference. We benchmark our strategy on Squeezenet 1.0, DenseNet-121, and ResNet-18, measuring accuracy degradations of only 0.4/1.1/1.7%.**

***Index Terms*—neural networks, quantization, accelerators, approximate computing**

## I. Introduction

With the continued expansion of the Internet of Things, computationally intensive applications are becoming ubiquitous on embedded devices [1]. Particularly, Deep Neural Networks (DNNs) are nowadays applied to a wide range of applications, including image recognition [2], object detection [3], and natural language processing [4]. To support complex DNNs within the tight constraints characterizing IoT systems, a wide range of optimization strategies have been proposed, including algorithmic [5], [6] and hardware [7], [8] approaches.

From the hardware perspective, approximate computing is well suited to DNNs, as they are naturally robust against minor runtime perturbations [9]. As such, various approximate computing architectures have been proposed for enhancing DNN efficiency. Even so, approximation error still degrades accuracy, necessitating an assortment of solutions such as limited neuron approximation [10], [11], or ensemble networks [12]. Accuracy degradation can also be mitigated through retraining while simulating the functionality of the Approximate Multiplier (AM); however, such retraining cannot be efficiently accelerated by, for example, GPUs, thus increasing training time by hours or days [13]. Further, many works do not utilize AMs uniformly across the network [11], [13], reducing generalizability in the case of hardware implementation.

In this work, we address the challenges of DNN retraining and uniformity via a hardware/software co-design solution, coupling hardware approximation with DNN optimization. We first propose a carryless partial sum AM to increase DNN matrix multiplication efficiency. By discarding the carry bit during partial product summation, we convert 58% of the multiplier's full adders into `xor` or `or` gates, greatly reducing area, power consumption, and delay. This AM has the unique benefit of computing exact values if at least one input is Fibonacci encoded; that is, its binary form contains no consecutive ones.

To eliminate accuracy degradation due to error injected by the AM, we then propose a weight quantization strategy to guarantee exact multiplication results during inference. First, Fibonacci Code Quantization (FCQ) encodes weights to the closest Fibonacci code word, thus eliminating errors during partial product summation. We then use Incremental Network Quantization (INQ) and retraining to eliminate accuracy loss due to FCQ. As FCQ guarantees exact outputs from our inexact AM, we can perform retraining without AM simulation, drastically reducing retraining time. We explore three INQ strategies across three DNNs, namely, ResNet-18, DenseNet-121, and Squeezenet 1.0, on the CIFAR-100 dataset. Our results demonstrate the possibility to achieve full FCQ with only 0.4%, 1.1%, and 1.7% accuracy degradation, respectively.

The contributions of this paper are as follows:

- We propose a carryless partial sum Approximate Multiplier (AM) that replaces 58% of an 8-bit standard multiplier's full adder operations with `or` logic. We implement our AM in 65nm TSMC CMOS and analyze its area and power-delay-product reductions (73/43%, respectively), as well as its mean relative error distance (0.054) against other AMs.
- We propose Fibonacci Code Quantization (FCQ), a strategy for weight quantization such that weights produce exact results when multiplied via our AM. FCQ reduces retraining time by 300x compared to retraining with other AMs.
- We utilize Incremental Network Quantization (INQ) to recover accuracy lost due to FCQ. We explore multiple INQ strategies across three benchmarks, namely, Squeezenet 1.0, DenseNet-121, and ResNet-18, demonstrating full FCQ with accuracy losses of 0.4%/1.1%/1.7%, respectively.

The remainder of this paper is organized as follows. Section II discusses background work. Section III details our AM implementation. Section IV explains how FCQ eliminates AM errors. Section V details our INQ strategy for regaining accuracy. Sections VI and VII detail our benchmarking process and analysis, and contextualizes this work among other approximate DNN publications. Finally, Section VIII concludes the paper.

This work has been partially supported by EC H2020 RECIPE project (GA No. 801137), EC H2020 WiPLASH project (GA No. 863337), ERC Consolidator Grant COMPUSAPIEN (GA No. 725657), and by the Swiss NSF ML-Edge Project (GA No. 200020_182009).

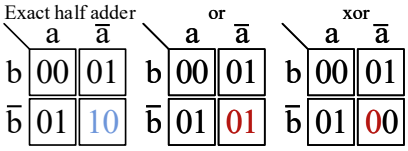


Fig. 1. Karnaugh maps for an exact half adder, as well as `or` and `xor` gates. While `xor` gates compute the sum value exactly in all instances, `or` gates provide the least error when the carry value is considered.

## II. Background Literature

This section introduces background concepts relating to this work. Section VII-C will contextualize this work among previous works proposing AMs for use in DNNs.

### A. Approximate Multiplication

The objective of approximate multiplication is to reduce the area footprint, power consumption, and delay of a multiplier by approximating/simplifying aspects of the architecture. Kulkarni *et al.* [14], for example, modifies the Karnaugh Map of a 2x2 building block multiplier to halve its area. Another approximation method is the reduction of the partial product summation tree complexity, e.g., by performing partial product summation for only a portion of product MSBs, while combining the LSBs via a low cost combinatorial method [9]. Thirdly, the full adder blocks can be approximated through a variety of methods [15]. Finally, use of genetic algorithms to generate Pareto optimal AMs has been demonstrated to be effective [11], [13]. This work belongs to the third category, as it modifies the partial product adder array by replacing a portion of full adder elements with simpler gates to reduce area, power, and latency.

### B. Neural Network Quantization

Previous works have proposed various neural network weight quantization schemes to better meet the power, performance, and memory constraints of embedded devices. Low precision quantization of weights to fixed-point integers could be considered a baseline [16], but more aggressive weight quantization strategies have been proposed, such as power of 2 quantization [5] or weight+activation quantization [17]. Aggressive quantization strategies require care to maintain acceptable accuracy. One such method that maintains high accuracy under aggressive quantization is Incremental Network Quantization (INQ) [5]. INQ quantizes a fraction of weights at each training epoch, before retraining the remaining weights to regain lost accuracy. In this work, we utilize INQ to quantize DNN weights to values that do not incur error when utilized with our AM.

## III. Carryless Partial Sum Approximate Multiplier

In the scope of improving DNN efficiency via approximate computing, we introduce a carryless partial sum multiplier. We accomplish this by simplifying a portion of Full Adders (FAs) within the adder tree to `or` or `xor` logic, as shown in Figure 2-a. This modification greatly reduces multiplier area, power consumption, and delay, at the cost of potentially introducing approximation error. We describe a weight quantization methodology for avoiding such errors in Section IV.

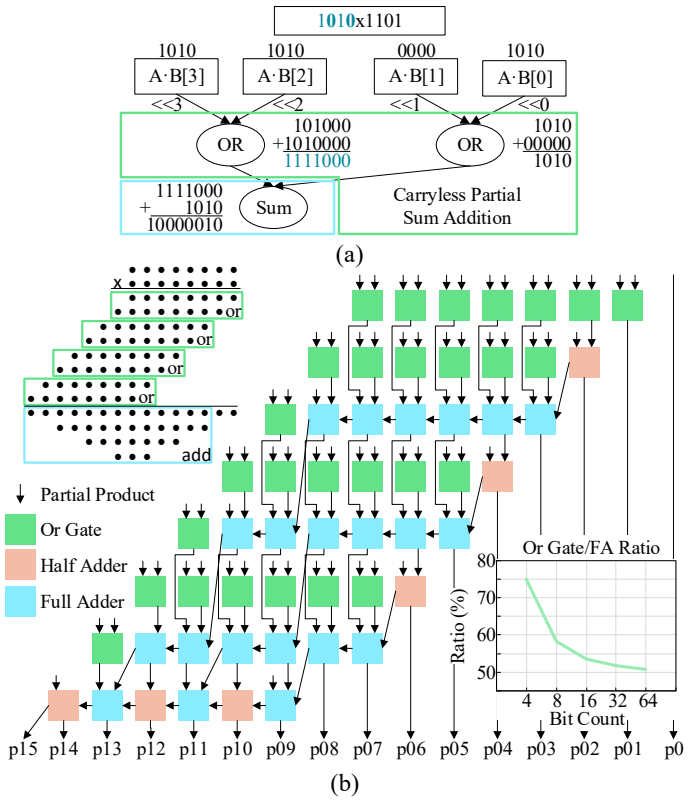


Fig. 2. (a) 4-bit multiplication with carryless partial product summation enabled by `or` gates. (b) 8x8 adder array with FA gates replaced by `or` gates to reduce area and power consumption. The ratio of replaced adders approaches a limit of 50% as bit width increases.

### A. Selection of Reduction Operator

When simplifying the FAs, two viable replacement options consist of `or` and `xor` gates. A `xor` gate advantageously produces only one incorrect bit with inputs $ab = 11$, as seen in Figure 1. In contrast, `or` gates require fewer transistors and produce a closer exact value for $ab = 11$. Indeed, we find that for all input combinations for an 8-bit multiplier, the Mean Relative Error Distance (MRED), or average distance for every approximate and expected product, is 0.99 for a `xor` based AM, while the MRED of an `or` based AM is only 0.054. This work therefore utilizes `or` gates for partial product reduction [18].

### B. Approximate Multiplier with Partial Product Or Reduction

Figure 2-b illustrates our AM architecture applied to a carry save multiplier. Initially, an $n$-bit carry save architecture contains $(n-2) * n$ FAs and $n-1$ half adders. To implement our AM, we replace the adders responsible for partial product summation with `or` gates, visible as green boxes. The remainder of adders, symbolized by blue and orange boxes, accumulate the intermediate values into the final product and are left unchanged. Hence, the number of FAs replaced can be calculated via the equation $\frac{n^2-n}{2}$. At 4 bits, the ratio of replaced FAs is 75%; this ratio decreases to 50% as multiplier width is increased.

One setback of this (and indeed, most) AMs is their injection of approximation errors into applications. It would be preferable to be able to predict for which inputs an AM will produce exact or erroneous results. For a carryless partial sum AM, such

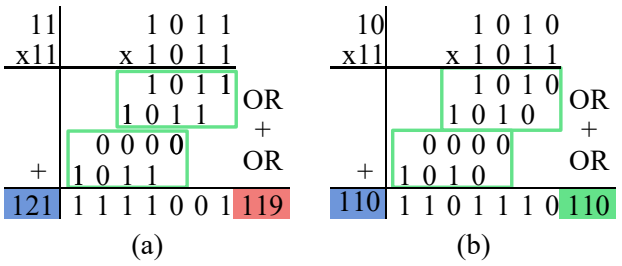


Fig. 3. Naive carryless partial sum multiplication will lead to errors and reduced accuracy (a). Such errors can be avoided by quantizing NN weights according to Fibonacci code words (b).

prediction is possible, a fact which we exploit in the next section to enhance neural networks with approximate multiplication while maintaining a high output accuracy.

## IV. Fibonacci Code Word Quantization for DNNs

### A. Motivation

The AM proposed in Section III provides significant reductions in area, power consumption, and delay, which will be quantified in Section VII; however, it's naive utilization in a neural network would introduce unacceptable accuracy degradation. It has been demonstrated previously that retraining an AM-enhanced DNN recoups lost accuracy [19]. However, in order to retrain the network, it is necessary to simulate the functionality of the AM, for example via a lookup table [13]. AM simulation while retraining prohibitively increases training time (15 days for ResNet-18 retraining [13]) by precluding the use of hardware components such as vector processors or GPUs. While this has proven a challenge for previous works, the AM proposed in this paper enables a novel method of retraining without AM simulation via Fibonacci code word quantization.

### B. Countering Approximation Errors Via Weight Fibonacci Coding Quantization

Approximate multipliers by their nature introduce errors into the product of the operation. For the AM proposed in Section III, the introduced error is illustrated in Figure 3-a, namely, if two partial products contain overlapping ones, the resulting value will be incorrect. To avoid such errors, we propose to quantize weights such that errors will not occur when reducing partial products. This is accomplished by quantizing weight values to the closest Fibonacci code word [20], that is, the closest value such that no consecutive ones appear in the binary representation of at least one of the operands, as illustrated in Figure 2. The result of such a quantization is that the sum of any two partial products is equivalent to a bitwise `or` operation between them, enabling them to be summed exactly. Such a quantization and multiplication is illustrated in Figure 3-b. Importantly, *only one* multiplier input need be a Fibonacci code word to guarantee an exact output; the other input can be any value between 0 and $2^n - 1$. This qualification is necessary for utilization in a DNN without needing to also modify layer inputs. We coin this method of quantization Fibonacci Coding Quantization (FCQ).

**Algorithm 1** Incremental Fibonacci code quantization and retraining flow.

**Input: f_model: Float Model, strategy: INQ Strategy, q_steps: INQ Steps, r_epochs: # Retraining Epochs**
**Output: Fibonacci Code Quantized Model**

```
def fib_quantize(f_model, strategy, q_steps, r_epochs):
    q_model = quantize(f_model)
    for i = 0; i < len(q_steps); i++ do
        q_model = fib_enc_and_freeze(q_model, q_steps[i])
        f_model = dequantize(q_model)
        for j = 0; j < r_epochs; j++ do
            f_model = train(f_model)
        end for
        q_model = quantize(f_model)
    end for
    return q_model
```

### C. Quantization Parameters for Fibonacci Code Quantization

FCQ is based on the low-precision general matrix multiplication (gemmlowp [16]) method. In gemmlowp, a scale and zero-point value for the weights of each layer are calculated such that the weight matrix can be scaled between minimum and maximum fixed-point values and the real value of 0 is exactly representable. FCQ builds on gemmlowp by further quantizing the fixed-point values to Fibonacci code words. A few considerations must be made to ensure that FCQ can be co-implemented with gemmlowp.

First, when quantizing a network, the range of values to which the weights are quantized must be considered. In gemmlowp, this is typically the range of signed values a given $n$-bit binary value can represent, e.g. -128 to 127 for 8-bit signed values. In the case of FCQ, asymmetric quantization is used, with a minimum value of 0 and a maximum value $quant_{max}$ of $2^n - 1$. This is necessary as small two's complement negative values contain many consecutive ones and thus are poorly represented as Fibonacci code words. Bias values are not encoded to Fibonacci code words as they are not involved in multiplication.

FCQ imposes further constraints on the upper range of quantizable values, as the maximum possible unsigned $n$-bit value $F_{max}$ a weight can take is '10' repeating for $n/2$ bits (e.g. 10101010 or 170 for 8-bit values), with any value above this being clamped to $F_{max}$. In order to maintain weight value variance while preventing an excessive quantity of weights from being clamped to this maximum value, we select a $q_{max}$ value of $\frac{(quant_{max}+F_{max})}{2}$, midway between the max quantized and max Fibonacci values (212 for 8-bit values).

Finally, in conjunction with asymmetric quantization, we find that using max pooling layers in contrast with average pooling layers provides higher network accuracy, as such layers are less impacted by the elimination of negative values.

Extreme forms of quantization, e.g. FCQ as presented here, typically induce unacceptable levels of accuracy loss if implemented in isolation. We overcome this barrier through the use of Incremental Network Quantization (INQ) [5] to recover lost accuracy due to FCQ.

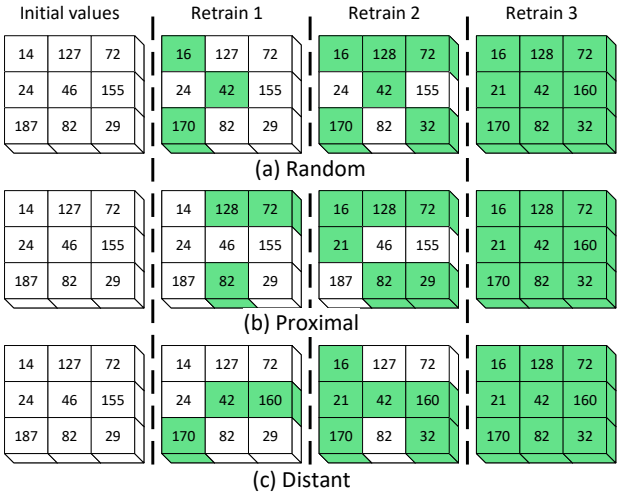


Fig. 4. Three incremental quantization strategies for Fibonacci incremental retraining; (a) Random, (b) Proximal, and (c) Distant. In each retraining step, a fraction of weight values are quantized to the nearest Fibonacci code word and frozen, represented by green boxes.

## V. Improving Accuracy through Incremental Network Quantization

### A. FCQ as an Enabler for Approximate INQ

In INQ, weight values are incrementally quantized to a range or set of values and then frozen while the remainder of the network is retrained to mitigate the subsequent loss in accuracy. In previous works, INQ as a method to recover lost accuracy due to approximate multiplication would lead to prohibitively long training times, as the AM must be simulated via a lookup table during training [13]. However, as discussed in Section IV-B, the AM presented in Section III necessitates that only one input value be a Fibonacci code word, while the other input may take any value. This characteristic enables us to retrain the network without needing to simulate the AM, as Fibonacci quantized weights are guaranteed to produce exact products.

Algorithm 1 illustrates our methodology for performing FCQ via INQ. At each iteration, we quantize the network to 8-bits, perform FCQ on a fraction of the weights, freeze their values, then convert the model back to floating point. We then retrain the remaining values to regain accuracy. These steps are repeated until FCQ has been applied to all weights.

### B. Incremental Quantization Strategies

In order to implement INQ, a strategy to iteratively quantize weights must be selected. We propose three such strategies, random, proximal, and distant, as illustrated in Figure 4.
*Random* allocates weights randomly to quantization steps. This has the advantage of being easy to implement, but is simplistic.
*Proximal* allocation orders weights for quantization by proximity to the nearest Fibonacci code word. This strategy results in the least perturbation in the early stages of training, reducing the chance of placing the network in an unrecoverable state.
*Distant* allocation quantizes weights in reverse order of the proximal strategy. This strategy performs most retraining in the early steps, allowing more potential for recovering lost accuracy as fewer weights are frozen during steps of greatest quantization.

TABLE I
Cumulative Fraction of Weights to Quantize at Each Incremental Quantize-and-Retrain Step

| Strategy | Cumulative Fraction of Weights Quantized |
|---|---|
| Random | [0.05,0.1,0.15,0.2,0.25,0.3,0.35,0.4,0.45,0.5, 0.55,0.60,0.65,0.7,0.75,0.8,0.85,0.9,0.95,1.0] |
| Proximal | [0.3,0.4,0.5,0.6,0.7,0.8,0.85,0.9,0.95,0.98,0.99, 0.995,0.998,0.999,0.9995,0.9998,0.9999,1.0] |
| Distant | [0.001,0.0025,0.005,0.01,0.025,0.05,0.1,0.15, 0.2,0.25,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0] |

For each quantization strategy, we define the cumulative fraction of weights to be quantized at each step. For the random strategy, the fraction of weights increases uniformly from 0.0 to 1.0. For proximal, we take an aggressive approach towards initial quantization, as the first weights quantized are closer to a Fibonacci code word. Conversely, for the distant strategy, we perform initial quantization conservatively, as weights furthest from a Fibonacci code word are first encoded. Table I details the fraction of weights to quantize at each step for random, proximal, and distant FCQ strategies.

## VI. Experimental Setup

We perform a hardware assessment of our proposed AM via implementation in 65nm TSMC CMOS [21] to allow area, Power-Delay-Product (PDP), and MRED comparisons against state-of-the-art AM multipliers [15]. We also compare against a subset of AMs from the EvoApprox8b [22] library which are optimized with respect to MRED and power consumption. EvoApprox8b is a set of Pareto-optimal evolved AMs generated by genetic algorithms that have been utilized in previous works to improve neural network efficiency [13]. Finally, we compare against a 6-bit incrementally quantized network, as 6 bit weights enable 64 unique values, comparable to the 55 weight values enabled by FCQ quantization.

To assess our weight quantization and retraining strategy, we implement FCQ-INQ on the DenseNet-121 [23], Squeezenet 1.0 [24], and ResNet-18 [25] DNNs in PyTorch, over the CIFAR-100 database, and analyze accuracy over random, proximal, and distant quantization strategies against standard 8-bit quantization. We also perform OneShot FCQ in which we quantize all weights in one go. We add a hyperparameter, Iterative Steps (IS), which contains the fraction of weights to quantize at each step, defined by the values detailed in Section V. We start the retraining stage with a learning rate of 8E-4 and decrease it by a factor of 0.2 when the loss value plateaus, stopping once either the learning rate drops below 1E-6 or 24 epochs are completed. All trainings and inferences are performed with an NVIDIA Tesla T4 GPU.

## VII. Experimental Results and Analysis

### A. Hardware Synthesis and Analysis

As we are modifying the partial product adder tree, we compare our design against those described in [15], hereafter called the FA Approx library, as well as AMs from the EvoApprox8b library. Area and PDP results are illustrated in Figure 5-a. It should be noted that the EvoApprox library is

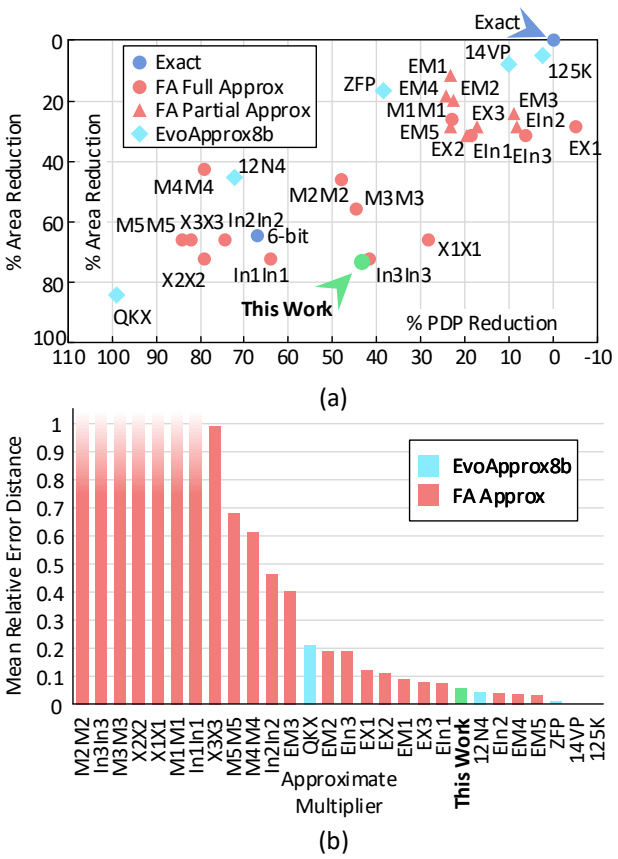


Fig. 5. (a) Area and Power-Delay-Product (PDP) reduction comparison between this work against other AMs for 8x8 multiplier architectures. (b) Mean Relative Error Distance (MRED) of this work in comparison to other AMs. This work presents high area and PDP reductions in relation to its low MRED.

implemented in 45nm CMOS; however, the relative area/PDP reductions are still relevant to this analysis. Red points belong to the FA Approx library, which contains two subsets of AMs, one with a fully approximated adder tree, and the other with only LSB approximation. Blue points represent the EvoApprox8b AMs. As can be seen, our AM design provides area/PDP reductions of 73/43%, respectively, with greater area reduction than any FA Approx AM and all but the smallest Evo AM. These reductions are impressive given that only 58% of the FAs were converted into `or` gates, whereas the works in comparison with better reductions consist entirely of approximate FAs.

Figure 5-b illustrates the MRED of the AMs in comparison. As can be seen, this work outperforms all fully and most partially approximate AMs from the FA Approx library. Most EvoApprox AMs provide better MRED; however, when taken in conjunction with Figure 5-a, this work provides a stronger trade-off between area, PDP, and MRED. Even with a low MRED, retraining is still necessary to recoup lost accuracy. We therefore analyze FCQ-INQ retraining features in the next section.

### B. Retraining Analysis

In order to demonstrate the importance of FCQ, we perform inference with DenseNet-121 on a set of 10000 images divided into 40 batches of 256 images per batch, while simulating an AM from the FA Approx library via the technique described by Mrazeck *et al.* [13]. To complete the entire set with AM

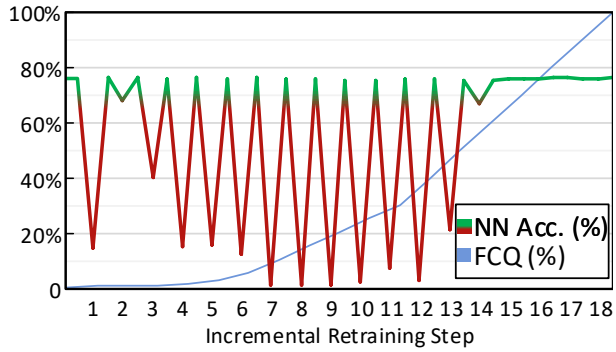


Fig. 6. DenseNet-121 accuracy during the quantization/retraining process utilizing the distant retraining strategy. Retraining enables to recoup nearly all accuracy loss. Weights are quantized conservatively, as described in Section V-B. The blue line represents the percentage of weights quantized at each step.

TABLE II
ACCURACY DEGRADATION FOR QUANTIZATION STRATEGIES AFTER INCREMENTAL QUANTIZATION

| Network | ResNet-18 | DenseNet-121 | Squeezenet 1.0 |
|---|---|---|---|
| Float Acc. (%) | 75.28 | 77.47 | 69.24 |
| Quantized Acc. (%) | 75.16 (-0.43) | 76.37 (-0.10) | 68.85 (-0.39) |
| 6b Quant. Acc. (%) | 73.78 (-1.5) | 67.28 (-1.96) | 71.42 (-6.05) |
| OneShot Acc. (%) | 72.42 (-2.86) | 64.76 (-12.71) | 60.2 (-9.04) |
| Random Acc. (%) | 73.05 (-2.23) | 74.62 (-2.85) | 66.83 (-2.41) |
| Proximal Acc. (%) | 73.04 (-2.24) | 71.64 (-5.83) | 65.25 (-3.99) |
| Distant Acc. (%) | 73.54 (-1.74) | 76.37 (-1.1) | 68.86 (-0.38) |

simulation takes ∼20 minutes. In comparison, the same set of inferences without AM simulation takes only 4-5 seconds, or *240-300x* faster. Through FCQ, we avoid AM simulation, and, hence, we can retrain our network without drastic slowdown.

Figure 6 illustrates the accuracy loss during FCQ and subsequent recovery during retraining for DenseNet-121. It is clear that retraining is necessary to maintain network accuracy, as even FCQ for 0.1% of weight values reduces accuracy by 62%. Via fast retraining, however, we recover almost all lost accuracy. The ability to fast retrain is nearly unique to the AM presented in this work; all AMs in the FA Approx library, and all but 2 AMs in whole EvoApprox8b library (totaling 46 multipliers) depend on both inputs to ascertain the accuracy of the output, with the 2 Evo AMs providing only 4/7% and 0.7/5% area/power reductions.

Table II details the results of FCQ on the selected benchmarks. As can be seen, 8-bit weight quantization leads to little accuracy degradation. We then perform FCQ using the aforementioned ICQ strategies. Accuracy degradation for the OneShot strategy is predictably poor, as only *bias* and *batchnorm* hyperparameters can be retrained. Next, the random strategy provides reasonable results across all networks. Interestingly, random FCQ performs better than the proximal strategy, due to the aforementioned fact that little training is done in the early retraining stages. In contrast, the distant strategy recovers nearly all lost accuracy across all networks, resulting in degradations of only 0.4%, 1.1%, and 1.7%. FCQ also generally outperforms 6-bit quantization as the variance of quantized values is higher in FCQ.

TABLE III
COMPARISON OF THIS WORK TO OTHER APPROXIMATE MULTIPLIER DNNS

| Work | Dataset | Retrain/Uni./ Depth | Energy Reduction | Accuracy Loss |
|---|---|---|---|---|
| ApproxANN [10] 2015 | MNIST | Slow / No / | -35% | -0.5% |
| | CIFAR-10 | Unspecified | -51% | -0.5% |
| Jiao [26] 2018 | MNIST | No / Yes / Low | -48% | -1.0% |
| ALWANN [13] 2019 | CIFAR-10 | No / No / High | -30% | -0.6% |
| | | | | -0.9% |
| | | | | -1.7% |
| This work | CIFAR-100 | Fast / Yes / High | -39% | -0.4% |
| | | | | -1.1% |
| | | | | -1.7% |

### C. Comparison to the State-of-the-Art

As the multiplication operation consumes a sizable portion of total energy cost of inference [13], significant research has been performed to implement approximate multiplication for DNNs. Several works [11], [13] utilize genetic algorithms to explore the multiplier design space and simulate the impact of approximation on each layer and neuron of the neural network. These works utilize non-uniform AM architectures across the network as well as in some examples mixing AM architectures within a network, demonstrating impressive energy, area, and delay values for the network and dataset but eliminating flexibility to other networks and datasets when implemented in hardware. Many works also require retraining during or after insertion of AMs [10], [11]. Such retraining requires that the approximate multiplier be simulated, precluding the use of hardware optimizations for vectorized multiplication and increasing inference time to hours or days for deeper networks [13]. Finally, partly as a result of the prohibitive cost of retraining, most works thus far have targeted simpler datasets such as MNIST, CIFAR-10, or SVHN.

Therefore, our work differs from the majority of AM based neural networks in 3 aspects: 1) Our AM architecture guarantees exact outputs when weight values are properly quantized. This means that it is not necessary to simulate the AM, which leads to the second benefit of this work, 2) network retraining can be performed with hardware optimizations via standard vectorized multiplication. This enables for much deeper approximated networks than has been demonstrated in the state-of-the-art, as well as fast exploration of the hyperparameter design space, while still providing large area and power reductions. Finally, 3) The design is applied uniformly across the network, and has thus been demonstrated as generalizable to networks of various depths and feature types such as Squeezenet's Fire modules and ResNet's residual functions.

Table III compares this work with other state-of-the-art works across various features, such as dataset, necessity of retraining, uniformity and network depth. As can be seen, the presented work demonstrates state-of-the-art accuracy while maintaining hardware uniformity across the network. While incremental retraining is utilized, it is performed with hardware acceleration, enabling approximation even in deep networks.

## VIII. CONCLUSION

In this work, we have presented a hardware/software co-design solution for reducing the area, power, and delay costs of DNN multiplications. Our carryless partial sum AM replaces over half of a multiplier's full adders with `or` gates, reducing area and PDP by 73/43%. This AM also performs exact multiplications when one input is a Fibonacci code word. We exploit this characteristic by incrementally Fibonacci quantizing and retraining DNN weights. Our methodology reduces inference runtime during retraining by 240-300x, while our benchmarks, Squeezenet 1.0, DenseNet-121, and ResNet-18, incur very small accuracy losses of 0.4/1.1/1.7% for the CIFAR-100 dataset, while still benefiting from approximate multiplication.

## REFERENCES


[1] GfK, *Gfk smart home study 2018*, Mar. 2018.
[2] A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, 2017.
[3] S. Ren, K. He, *et al.*, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *NIPS 28*, 2015.
[4] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," *ICML*, 2008.
[5] A. Zhou, A. Yao, *et al.*, "Incremental network quantization: Towards lossless cnns with low-precision weights," 2017. arXiv: 1702.03044.
[6] X. Lin, C. Zhao, and W. Pan, "Towards accurate binary convolutional neural network," in *NIPS*, 2017.
[7] B. Reagen, P. Whatmough, *et al.*, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *ISCA*, 2016.
[8] W. Khwa, J. Chen, *et al.*, "A 65nm 4kb algorithm-dependent computing-in-memory sram unit-macro with 2.3ns and 55.8tops/w fully parallel product-sum operation for binary dnn edge processors," in *ISSCC*, 2018.
[9] Khaing Yin Kyaw, Wang Ling Goh, and Kiat Seng Yeo, "Low-power high-speed multiplier for error-tolerant application," in *EDSSC*, 2010.
[10] Q. Zhang, T. Wang, *et al.*, "Approxann: An approximate computing framework for artificial neural network," in *DATE*, 2015.
[11] S. Venkataramani, A. Ranjan, *et al.*, "Axnn: Energy-efficient neuromorphic systems using approximate computing," in *ISLPED*, 2014.
[12] F. Ponzina, M. Peon, *et al.*, "E2cnns: Ensembles of convolutional neural networks to improve robustness against memory errors in edge-computing devices," *IEEE TC*, 2021.
[13] V. Mrazek, Z. Vasicek, *et al.*, "Alwann: Automatic layer-wise approximation of deep neural network accelerators without retraining," in *ICCAD*, 2019.
[14] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *VLSID 24*, 2011.
[15] M. Masadeh, O. Hasan, and S. Tahar, "Comparative study of approximate multipliers," ser. GLSVLSI, 2018.
[16] B. Jacob, P. Warden, *et al.* (2015), [Online]. Available: https://github.com/google/gemmlowp.
[17] D. Zhang, J. Yang, *et al.*, "Lq-nets: Learned quantization for highly accurate and compact deep neural networks," in *ECCV*, 2018.
[18] L. Ni and K. Hwang, "Vector-reduction techniques for arithmetic pipelines," *IEEE Transactions on Computers*, 1985.
[19] V. Mrazek, S. S. Sarwar, *et al.*, "Design of power-efficient approximate multipliers for approximate artificial neural networks," in *ICCAD*, 2016.
[20] A. S. Fraenkel and S. T. Kleinb, "Robust universal complete codes for transmission and compression," *Discrete Applied Mathematics*, 1996.
[21] (2005), [Online]. Available: https : / / www . tsmc . com / english / dedicatedFoundry/technology/logic/l_65nm.
[22] V. Mrazek, R. Hrbacek, *et al.*, "Evoapprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods," in *DATE*, 2017.
[23] G. Huang, Z. Liu, *et al.*, "Densely connected convolutional networks," in *CVPR*, 2017.
[24] F. N. Iandola, M. W. Moskewicz, *et al.*, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size,"
[25] K. He, X. Zhang, *et al.*, "Deep residual learning for image recognition," in *CVPR*, 2016.
[26] X. Jiao, V. Akhlaghi, *et al.*, "Energy-efficient neural networks using approximate computation reuse," in *DATE*, 2018.