XbNN: Enabling CNNs on Edge Devices by Approximate On-Chip Dot Product Encoding

Lucas Klemmer[1] Saman Froehlich[2] Rolf Drechsler[2,3] Daniel Große[1,3]
[1]Institute for Complex Systems, Johannes Kepler University Linz, Austria
[2]Institute of Computer Science, University of Bremen, Bremen, Germany
[3]Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
lucas.klemmer@jku.at, froehlich@uni-bremen.de, drechsler@uni-bremen.de, daniel.grosse@jku.at

***Abstract*—Only a few trends have gained as much traction as Edge Computing and *Neural Networks* (NN). Both have the potential to radically change how technology influences us. However, since edge devices feature only very limited resources, the sheer amount of performance required by modern NNs limits their use on the edge. Especially, the conversion of *Convolutional Neural Networks* (CNN) into feasible on-chip designs remains a hard task. Currently, hand-crafted and most-often very heavy architectures have to be used as existing *High-Level Synthesis* (HLS) frameworks provide only inefficient solutions.**

In this paper, we introduce the *Crossbar Neural Network* (XbNN) architecture. Our architecture employs a novel approximate on-chip dot product encoding for the efficient synthesis of CNNs on hardware. This encoding embeds the weights used in CNNs into the hardware design itself, significantly reducing the required memory and computation time. In addition, we present a methodology for the automated conversion of traditional CNNs given in TensorFlow into accelerators on top of the XbNN architecture. To demonstrate the effectiveness of XbNN, we conduct experiments on a common CNN test dataset and analyze the accuracy and performance of the resulting XbNN accelerators. We show that XbNN (a) achieves similar accuracies compared to TensorFlow CNNs and (b) provides much better area and performance results in comparison to a state-of-the-art HLS flow.

I. Introduction

The process to shift computation from data centers to the edge picks up speed steadily. At the edge more and more but at the same time smaller and smaller devices are found enabling a variety of unique applications. However, while the number of edge devices has heavily increased, they are inherently limited by their available computation power and energy consumption [1].

In the recent years, machine learning has become a key part of modern technologies, such as self-driving cars [2], general image recognition tasks [3] or natural language processing [4]. *Neural Networks* (NNs) have been successfully utilized in the context of machine learning and are the state-of-the-art technique used. NNs consist of multiple layers including countless neurons, which ultimately lead to very complex structures with many parameters. The parameters of NNs are trained during a time consuming training phase such that a given NN structure performs well on given training and test data sets in terms of accuracy. Different classes of NNs exist, however, especially *Convolutional Neural Networks* (CNNs) are widely used for image recognition tasks [5]–[7]. Filters, which play a key role in the convolutional layers of CNNs, require a lot of computation time and energy. During execution, they are mapped to a multitude of dot products, which have to be computed. Consequently, optimizations of the dot product have gained a lot of research interest [8]–[10].

The conversion of existing CNNs into on-chip designs feasible for the use in edge devices remains a hard task. Currently, hand-crafted architectures have to be employed for on-chip implementations as existing *High-Level Synthesis* (HLS) frameworks provide only inefficient solutions. Thus, to enable CNNs for the edge in practice, reducing the complexity of CNNs by quantization [11], [12] or even binarization in *Binary Neural Networks* (BNNs) [13] has come into focus in several recent works. However, CNNs can hardly be translated into BNNs efficiently and without significant loss of accuracy. Therefore, a major drawback of BNNs is their requirement of custom models and training processes which are time consuming and difficult to automate.

In this paper, we propose a novel NN structure called *Crossbar Neural Network* (XbNN). XbNNs use approximation techniques and leverage characteristics inherent to digital logic combing the expressiveness of CNNs with the high performance of BNNs. In the XbNN architecture, the computation of the filters are mapped to an approximate dot product encoding. To find the encoding we map the problem to an *Integer Linear Programming* (ILP) instance under given resource constraints. Overall, this results in a highly efficient hardware implementation of convolutional layers.

In addition, we present a methodology for the automated conversion of traditional CNNs given in TensorFlow [14] into accelerators on top of the XbNN architecture. This allows the direct adoption of XbNNs for SW developers, since no knowledge about HDL programming is required. To demonstrate the effectiveness of XbNN, we conduct experiments on a common CNN test data set and analyze the accuracy and performance of the resulting XbNN accelerators. We show that XbNN achieves similar accuracy compared to TensorFlow CNNs. Moreover, XbNN provides much better area and performance results in comparison to a state-of-the-art HLS flow.

II. Related Work

First, we consider related work which targets the "optimization" of the core operation (dot product) of NNs, for instance [8]–[10]. Essentially, these approaches use quantization or some form of approximation (often based on a fixed-point representation) of the weights. In contrast in this paper, we

encode the weights and a large part of the arithmetic operations implicitly into the hardware logic and hence can significantly reduce area since no memory is needed and the number of computation blocks such as multipliers and adders is reduced.

Second, there is a recent approach targeting CNN acceleration on the edge [15]. The authors devise a method to store all weights on FPGA memories, but they still keep the traditional separation of memory and computational logic. In this paper, we break with this separation and remove the memory dependency as the major performance bottleneck.

Next, we have to look on papers related to HLS of NNs. In [16] LeFlow has been introduced. LeFlow is a HLS design flow for NNs which allows to synthesize NNs which are given in the TensorFlow framework [14]. LeFlow first translates the given NN definition to an optimized LLVM-IR [17] using Googles XLA compiler [18]. Then, it uses an enhanced version of LegUp [19] to generate synthesizable Verilog from the LLVM-IR representation. We show in the experiments that we clearly outperform LeFlow. Integrating given approximate components (e.g. adders, multipliers [20]–[22]) into LeFlow has been considered in [23]. In [24], the authors proposed an approximation resilience exploration metric and an approximate accelerator for convolutional layers targeted for edge devices. FINN [25], [26] is a HLS-framework for BNNs which uses a streaming architecture. However, FINN is tied to the usage of PYNQ boards, and is therefore not applicable to arbitrary hardware. In contrast, we present a method to generate synthesizable Verilog code for our proposed XbNN architecture. Hence, our approach is much more flexible and can be used in any FPGA or ASIC flow.

III. Preliminaries

We briefly review CNNs and then focus on the dot product as the core operation when going into hardware. CNNs are state-of-the-art for many classification problems, for instance in image recognition. In contrast to conventional NNs, CNNs feature convolutional layers. Based on filters, these convolutional layers detect patterns in input images, which can then be combined to more complex features of these images. The filters consist of two or three dimensional matrices, which are convolved across the input. The output of one convolutional layer can be fed into another convolutional layer to combine the detected patterns. In addition to convolutional layers, CNNs also feature pooling layers, which reduce the size of the output of a convolutional layer by summarizing multiple adjacent outputs.

In practice, the computation of the convolution of the filters is mapped to the computation of numerous dot products and is a very computational intensive task. The *dot product* $< x, y >$ of two vectors $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ is defined as

$$< x, y >= \sum_{i=1}^{n} x_i \cdot y_i \tag{1}$$

and thus can be computed by *Multiply-Accumulate* (MAC) operations. In CNNs, one of these vectors represents the fixed weights of a filter, while the other vector is the input.

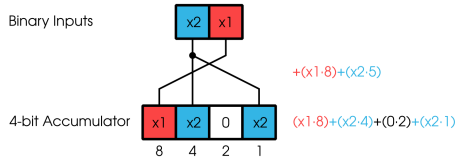

Fig. 1. Implementing multiple MACs that depend on more than one input and factors that are not a power of two.

IV. Approximate On-Chip Dot Product Encoding and XbNN Unit

In this section we lay the foundation for the proposed XbNN architecture which enables the synthesis of very efficient CNN accelerators by a *novel hardware approximation of the dot product*. For this, we leverage the encoding of binary numbers to implement all MACs of a dot product operation. We use the fact that each bit of a binary number represents a special case of the MAC operation, in which one operand is always binary, to encode an approximation of the dot product into a composition of wires and a single adder.

The foundation of the XbNN architecture is the encoding of integers in the binary system. The decimal value of an unsigned n-bit integer $y \in \{0,1\}^n$ is defined as $y = \sum_{i=0}^{n-1} 2^i \cdot y_i$. We can see, that the value is composed of multiple MAC operations and thus is similar to the definition of the dot product in Eq. 1.

Before we formalize the XbNN architecture, we illustrate the idea using the example in Fig. 1. In this example we assume a 4-bit integer, which is from now on referred to as the accumulator, because it sums the results of the multiplication of the dot product (cf. Eq. 1). We also assume two binary inputs which represent the input vector $x = (x_2, x_1)$ from Eq. 1. Each input x_i should implement a MAC operation of the form $+(x_i \cdot w)$ with $x_i \in \{0,1\}$ and $w \in \mathbb{N}$. We assume x_1 should implement the MAC operation $+(x_1 \cdot 8)$ and x_2 should implement $+(x_2 \cdot 5)$. To implement the MAC operation for x_1, the most significant bit of the accumulator is connected to x_1 as can be seen in Fig. 1 (red). Therefore, whenever $x_1 = 1$, the value of the accumulator is increased by 8.

The MAC operation $+(x_2 \cdot 5)$ for input x_2 (blue) cannot be expressed by connecting x_2 to a single bit, because the accumulator bits all encode values which are a power of two. To implement this MAC operation, x_2 can be connected to multiple accumulator bits whose summed up values result in 5. Accumulator bits which are not needed to form the necessary MAC operations are set to constant 0.

As can be seen from the example, this model allows to implement multiple MAC operations. However, there are two major issues: First, negative weights cannot be implemented and second, a weight can only be implemented once because each integer value can be encoded by exactly one combination of accumulator bits. Therefore, we extend our model to mitigate these problems. For this extension, the accumulator is split in two parts of which one is interpreted as a positive integer while the other is interpreted as a negative integer. The value of the accumulator is then calculated by summing the positive and negative parts. Fig. 2 shows the architecture

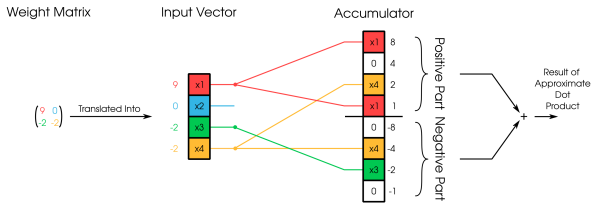

Fig. 2. Exemplary mapping of weight matrix to wiring of XbNN unit

of this extended model; due to the *crossbar structure* of the connections we denote it as *XbNN unit*.

Negative weights can be implemented by connecting inputs to the negative part of the accumulator (cf. x_3 in green). If two inputs have the same factor, the positive and negative parts can be combined to implement this weight multiple times. This can be seen in Fig. 2 with x_3 (green) and x_4 (yellow), which both realize the weight -2.

We now formalize the XbNN unit as the *approximate dot product*:

$$\vec{x} \odot B = \sum_{j=1}^{i} \left(\sum_{k=1}^{\frac{n}{2}} b_{kj} \cdot 2^{\frac{n}{2}-k} - \sum_{k=\frac{n}{2}+1}^{n} b_{kj} \cdot 2^{n-k} \right) \cdot x_j \quad (2)$$

The approximate dot product is defined on a binary input vector $\vec{x} \in \{0,1\}^i$ and a binary connection matrix $B \in \{0,1\}^{n \times i}$ where i is the number of inputs and n is the accumulator size. The binary connection matrix encodes which inputs are connected to which accumulator bits.

In hardware a bit can only be driven by a single source. Therefore, to be able to generate synthesizable hardware that implements an XbNN unit, we must constrain the binary connection matrix B as follows:

$$\forall j \in \{0, 1, \ldots, n\} : \sum_{k=1}^{i} b_{kj} \leq 1 \quad (3)$$

The constraint in Eq. 3 specifies that each accumulator bit can only be connected to at most one input.

Not every combination of MACs can be mapped to the proposed approximate dot product. For some combinations (e.g. all weights are equal) no mapping may exists, which means that some weights have to be approximated to be able to create an XbNN unit. Furthermore, the size of the accumulator has a significant impact on the quality of the mapping as well as the performance of the hardware implementation.

In the next section, we present a methodology for the conversion of CNNs and their trained weights to the approximate dot product using ILP and a hardware generator.

V. Designing Accelerators on top of the XbNN Architecture

In this section, we first present a methodology for converting CNNs into the XbNN architecture. In the second part, we propose an efficient accelerator design on top of the XbNN architecture.

A. Conversion of CNNs to the XbNN Architecture

Converting CNNs to the XbNN architecture is an approximation, because not every CNN filter can be directly mapped to an XbNN unit due to the limited possibilities of representing weights in an XbNN unit. Therefore, a methodology for mapping existing CNNs to the XbNN architecture is needed. To find mappings of CNN weights to connections in the XbNN unit, we propose to formulate the mapping task as an ILP optimization problem. Our proposed formulation employs a variable binary connection matrix (cf. Eq. 2) which encodes the connections of the XbNN unit. The solver then finds an assignment of this matrix, which is the best approximation of the CNN weights for a fixed accumulator size, i.e. the optimization goal is to minimize the difference between the CNN and XBNN weights.

XbNN units can only implement dot products for binary inputs and integer weights. To enable the conversion of floating-point CNNs into the XbNN architecture, we propose to first convert the floating-point weights to integers. For this, we make use of the observation, that the relative size of the weights is more important than their actual values. Our proposed method of converting floating-point weights first finds the largest absolute value *l_max* of all weights in a convolutional layer. We then map every weight of the layer from the number range *(-l_max, l_max)* into the range *(-xbnn_max, xbnn_max)* where *xbnn_max*, usually related to the accumulator size, can be chosen by the user during the conversion process.

Converting CNNs to XbNNs introduces approximations in the convolutional layers. This leads to the problem that the features found by the convolutional layers and the features expected by the fully connected layers can be different, which can result in a significantly lower prediction accuracy. To reduce this problem, we propose to perform a second training phase in which the weights of the XbNN modules are converted back into an CNN. The convolutional layers of this model are then fixed such that the second training phase only updates the fully connected layers. This approach is similar to transfer learning [27], which is commonly used for reusable feature extractors [28]. Since only the last layers have to be trained again the training time is significantly shorter.

Finally, when converting CNNs to the XbNN architecture the size of the accumulators of the XbNN units can be chosen for each convolutional layer. A larger size has the advantage of a smaller difference between CNN and XbNN weights. On the other hand a larger accumulator also requires more area and a larger adder. Considering the potentially hundreds or thousands of XbNN units in a design, this can add up to significant amounts of added area and delay. Choosing a smaller accumulator size can be useful to achieve a higher performance but at the cost of potential accuracy losses.

B. Accelerator Architecture

Our proposed accelerators are based on a streaming architecture in which each layer is an independent module. Each of these modules is connected to its predecessor and the following module using streams and a buffer that stores the currently required data for each module. The backbone of each accelerator are the XbNN Conv2D modules that implement

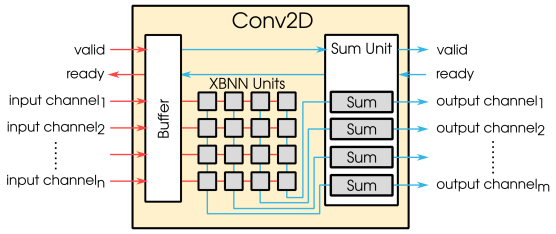

Fig. 3. Schematic of the Conv2D module. Each Conv2D module implements a convolutional layer and is connected to other modules using handshake-synchronized streams.

convolutional layers. The structure of XbNN modules is shown in Fig. 3. The buffer contains the control logic of the module and manages reads from the input stream. As soon as the buffer is filled with enough data the XbNN units start calculating the dot products of the input data and the weights. The weights are encoded into the XbNN units and each of them implements one 2D-filter of the layer. After the dot products are calculated the channels are merged together by the Sum Unit and the result is put on the output stream.

VI. Experimental Results

In this section we present our experimental results of converting CNNs to the XbNN architecture. We analyze the impact of the conversion on the prediction quality and the performance characteristics of the accelerators. Finally, we compare our XbNN architecture to LeFlow.

A. Performance on Fashion MNIST

In many scenarios, a pretrained CNN model is available or already in use. The XbNN architecture specifically aims at generating accelerators for these models. To simulate this use case, we trained a conventional floating point CNN model (cf. Table II) on the fashion MNIST data set [29].

TABLE I
Characteristics of XbNN accelerators A and B.

Model	Accuracy	Relearned	LUT %	Frequency	FPS
TF	91.23%	-	-	-	-
A	60.60%	86.26%	51.10%	100.00 MHz	$\sim$ 31,000
B	56.47%	85.54%	36.68%	111.11 MHz	$\sim$ 34,000

The accuracies of the baseline TensorFlow (*TF*) model and two selected XbNN configurations (*A*, *B*) and their accelerator performance are shown in Table I. We chose the configurations to represent two common scenarios: *(A)* a higher accuracy scenario with moderate approximation and *(B)* a high performance scenario with high approximation. The difference between configuration *A* and *B* is the size of the accumulators in the XbNN units. In configuration *A* all XbNN units feature an accumulator size of 10 bits (5 positive and 5 negative). In configuration *B* only the first convolutional layer uses XbNN units with 10 bit accumulators. In the remaining two layers the accumulator size is reduced to only 8 bits.

Compared to the baseline accuracy of 91.23% both XbNN configurations have significantly lower accuracies. However, after the second learning phase, the accuracies of both configurations can be significantly improved *(Relearned)*. We synthesized the accelerators for the Pynq Z2 development board and measured their performance in terms of LUT utilization, maximum frequency, and maximum theoretical *Frames Per Second* (FPS). The FPS are calculated by dividing the clock cycles per second by the clock cycles the accelerator needs to process one image. A clear performance improvement, both in utilization, frequency and FPS, can be observed for the high approximation configuration *B*.

TABLE II
Conv2D and Pooling layers of the Fashion MNIST XbNN model.

Type	Filter Shape	Input Shape
Conv2D	$1 \times 3 \times 3 \times 16$	$1 \times 28 \times 28$
Conv2D	$1 \times 3 \times 3 \times 32$	$16 \times 26 \times 26$
MaxPool2D	2×2	$32 \times 24 \times 24$
Conv2D	$32 \times 3 \times 3 \times 64$	$32 \times 12 \times 12$

B. Comparison to LeFlow

Table III shows a comparison of the accelerators generated by LeFlow and XbNN for a three layer MNIST [30] CNN (Conv2D $3\times3\times32$, Maxpool 2×2, Conv2D$3\times3\times64$). The accelerator generated by LeFlow requires nearly 70 million clock cycles to process the complete network. Compared to this, the XbNN accelerator requires only 1816 clock cycles to process the same network. The significantly lower required number of cycles of the XbNN accelerator and a maximum frequency more than three times higher result in an FPS performance advantage of multiple orders of magnitude. This advantage is a result of the very high degree of parallelization possible with the XbNN architecture. Also, due to the area efficient XbNN architecture the amount of required logic elements is nearly identical for both accelerators (19% vs. 21%). Further, XbNN requires no on-chip memory, since all parameters are encoded in the accelerator.

TABLE III
Characteristics of XbNN and LeFlow accelerators.

Property	LeFlow	XbNN
Cycles	69,289,300	1816
Frequency	51.7 MHz	183.62 MHz
FPS	0.74	101,112
Logics Util.	19%	21%
Memory Util.	66%	0%
Accuracy	97.42%	95.13%

VII. Conclusion

In this paper we have proposed the novel XbNN architecture. We have shown how to derive the XbNN architecture for a given CNN model and how to create the final accelerator implementation. In the experiments, we have demonstrated that the accuracy penalty of XbNNs compared to CNNs lies within an acceptable limit for many applications (5-6%) while performance improvements of multiple orders of magnitude are achieved.

Acknowledgments

This work was partly supported by the German Research Foundation (DFG) within the project PLiM (DR 287/35-1).

REFERENCES

[1] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[2] V. Swaminathan, S. Arora, R. Bansal, and R. Rajalakshmi, "Autonomous driving system with road sign recognition using convolutional neural networks," in *International Conference on Computational Intelligence in Data Science (ICCIDS)*, 2019, pp. 1–4.

[3] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *Conference on Computer Vision and Pattern Recognition*, 2014, pp. 1725–1732.

[4] O. Abdel-Hamid, A. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, "Convolutional neural networks for speech recognition," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 22, no. 10, pp. 1533–1545, 2014.

[5] H. Shin, H. R. Roth, M. Gao, L. Lu, Z. Xu, I. Nogues, J. Yao, D. Mollura, and R. M. Summers, "Deep convolutional neural networks for computer-aided detection: Cnn architectures, dataset characteristics and transfer learning," *IEEE Transactions on Medical Imaging*, vol. 35, no. 5, pp. 1285–1298, 2016.

[6] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *Conference on Computer Vision and Pattern Recognition*, 2014, pp. 1725–1732.

[7] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 815–823.

[8] M. Véstias, R. P. Duarte, J. T. de Sousa, and H. Neto, "Parallel dot-products for deep learning on FPGA," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4.

[9] M. P. Véstias, R. Policarpo Duarte, J. T. de Sousa, and H. Neto, "Hybrid dot-product calculation for convolutional neural networks in FPGA," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 350–353.

[10] L. Fiolhais and H. Neto, "An efficient exact fused dot product processor in FPGA," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 327–3273.

[11] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *CoRR*, vol. abs/1609.07061, 2016. [Online]. Available: http://arxiv.org/abs/1609.07061

[12] B. Moons, K. Goetschalckx, N. Van Berckelaer, and M. Verhelst, "Minimum energy quantized neural networks," in *Asilomar Conference on Signals, Systems, and Computers*, Oct 2017, pp. 1921–1925.

[13] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," in *arXiv*, 2016.

[14] Martín Abadi et. al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: https://www.tensorflow.org/

[15] G. Dinelli, G. Meoni, E. Rapuano, and L. Fanucci, "Advantages and limitations of fully on-chip cnn fpga-based hardware accelerator," in *International Symposium on Circuits and Systems (ISCAS)*, 2020, pp. 1–5.

[16] D. H. Noronha, B. Salehpour, and S. J. E. Wilton, "LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks," *ArXiv e-prints*, Jul. 2018.

[17] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[18] Google - TensorFlow, "XLA," https://www.tensorflow.org/xla/, 2018.

[19] A. Canis et. al., "Legup: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, pp. 24:1–24:27, Sep. 2013.

[20] S. Froehlich, D. Große, and R. Drechsler, "Approximate hardware generation using formal techniques," in *Approximate Circuits: Methodologies and CAD*, S. Reda and M. Shafique, Eds. Springer, 2019, pp. 155–174.

[21] A. Chandrasekharan, D. Große, and R. Drechsler, *Design Automation Techniques for Approximation Circuits*. Springer, 2018.

[22] S. Froehlich, D. Große, and R. Drechsler, "Approximate hardware generation using symbolic computer algebra employing Gröbner basis," in *Design, Automation and Test in Europe*, 2018, pp. 889–892.

[23] S. Froehlich, L. Klemmer, D. Große, and R. Drechsler, "ASNet: Introducing approximate hardware to high-level synthesis of neural networks," in *International Symposium on Multi-Valued Logic*, 2020, pp. 64–69.

[24] J. Castro-Godínez, D. Hernández-Araya, M. Shafique, and J. Henkel, "Approximate acceleration for cnn-based applications on iot edge devices," in *Latin American Symposium on Circuits Systems (LASCAS)*, 2020, pp. 1–4.

[25] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. ACM, 2017, pp. 65–74.

[26] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers, "Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–23, 2018.

[27] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345–1359, 2010.

[28] "Transfer learning and fine-tuning," accessed: 2020-11-06. [Online]. Available: https://www.tensorflow.org/tutorials/images/transfer_learning

[29] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," in *arXiv*, 2017.

[30] Mnist dataset. [Online]. Available: http://yann.lecun.com/exdb/mnist/