CNN implementation in Resource Limited FPGAs - Key Concepts and Techniques

José Rosa*, Mónica Figueiredo*† and Luis Conde Bento*‡
*DEE/ESTG, Politécnico de Leiria, Portugal
Email:2190383@my.ipleiria.pt, {monica.figueiredo, luis.conde}@ipleiria.pt
†IT, Leiria, Portugal, ‡ISR-UC, Coimbra, Portugal

***Abstract*—Convolutional Neural Network (CNN) is a specific type of algorithm that has become dominant in image recognition and classification. Currently, there is a tendency to migrate CNN implementations from the *cloud* to the *edge* (closer to the data source) in order to reduce both latency and communication bandwidth and at the same time, increase security and system efficiency. Field Programmable Gate Array (FPGA) is a good option for implementing CNN in the *edge*, since even the lowest cost FPGAs have a good energy efficiency and a sufficient throughput to enable real-time applications. In this paper, key concepts about CNN are reviewed. Next, the most popular compression methods used in the CNN training phase are described. Finally, we present the most popular frameworks for hardware accelerator design and key hardware optimization techniques used by many of those frameworks to enable CNN inference on resource limited devices.**

***Index Terms*—CNN, FPGA, *edge*, AI**

I. Introduction

Artificial Intelligence (AI) is a science that aims to provide machines with some degree of intelligence. Machine Learning (ML) is a branch of AI in which machines learn in order to achieve a certain goal without being explicitly programmed to do so [1], [2]. Nowadays, this type of algorithm is present in a wide range of applications, namely in digital assistance, fraud detection, medical image analysis and autonomous driving [3]. In order to learn, ML algorithms must undergo a training phase, in which the parameters are adjusted according to a given dataset. After the training phase, the algorithm can be used by the machine to extract information from new data - this process being called inference. Although ML algorithms are sufficient to solve most simple problems, Artificial Neural Network (ANN), often with multiple hidden layers, known as Deep Neural Network (DNN), are best suited for solving problems with a higher degree of complexity. Figure 1 depicts the hierarchy of the AI science [4].

The most common DNN architectures are Feedforward Deep Neural Networks, also known as Multilayer Perceptrons (MLP) [5]–[8], CNN [9], [10], and Recurrent Neural Network (RNN) [11], [12]. MLP are mainly used for approximation tasks, being suitable for processing tabular data (*e.g.* parametric tables). CNN are used to extract features from structural data (*e.g.* images) as convolution operations are reasonably insensible to simple geometric variations (*i.e.* translation, rotation, scaling) and distortion [13]. Finally, RNN are mostly used for correcting and predicting sequential data (*e.g.* audio) [14].

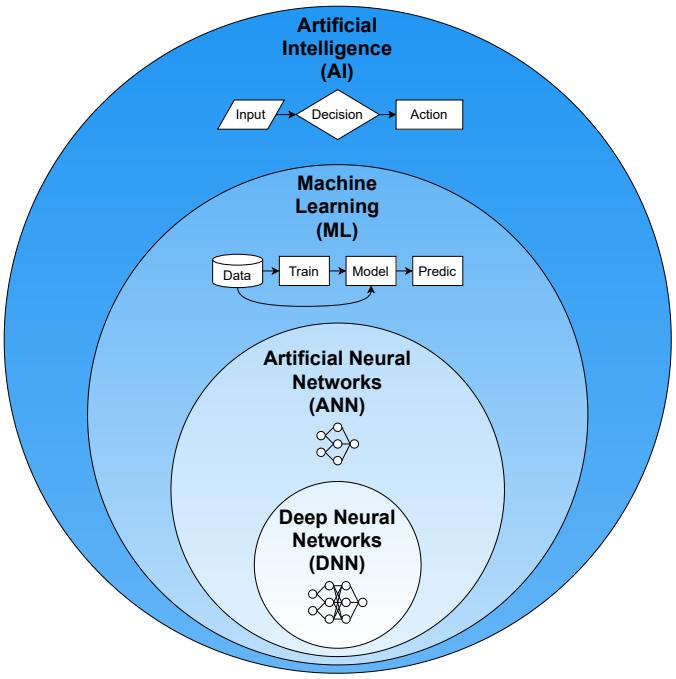


Figure 1: Artificial Intelligence landscape.

Traditionally, these networks are trained and implemented in the *cloud*, where computational and energy resources are virtually unlimited. However, inference in the *cloud* presents several problems: 1) the system becomes dependent on the network; 2) confidential data is transmitted, reducing system security; 3) redundant data may be transmitted since the processing is done in the *cloud*, wasting bandwidth; and 4) latency may prevent real-time inference. These problems can be mitigated if the inference is performed close to the data source or the inference request (*e.g.* sensor, mobile device or network node), *i.e.*, in the *edge*.

Unlike what is seen in *cloud*, DNN inference in the *edge* is hampered by the limited amount of computational and energy resources, and at the same time pushed for delivering high throughput and low latency. For this reason, there has been a growing interest in developing compression methods able to reduce hardware and energy footprint of DNN accelerators for real-time *edge* inference. Depending on the required compression, there may be a loss of accuracy so the designer should choose the most appropriate model and compression

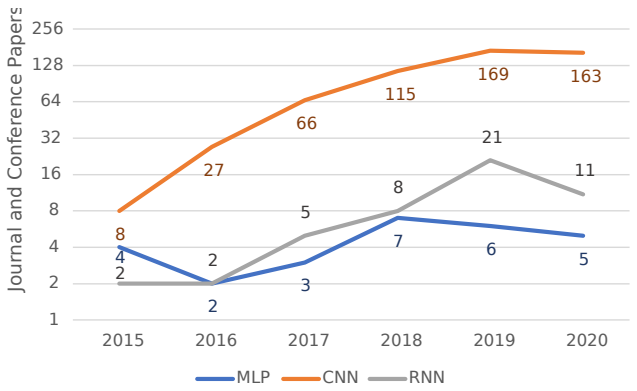


Figure 2: FPGA implementation trends according to published papers [15].

Table I: CNN Implementation Solutions Comparison [23]

	CPU	GPU	FPGA	ASIC
Adaptability (to a variety of situations)	High	Medium	Low	None
Compute Power	Medium	High	High	Medium
Latency	Medium	High	Low	Ultra Low
Throughput	Low	High	High	High
Parallelism	Low	High	High	High
Power efficiency	Medium	Low	Medium	High

techniques according to the application requirements.

Very simple CNN models could be implemented in Central Processing Unit (CPU) but in terms of throughput and latency CPU perform poorly. The use of Graphics Processing Unit (GPU) offer higher throughput, but incur in a high energy footprint. Application Specific Integrated Circuit (ASIC) have an optimal performance in terms of throughput, latency and energy efficiency, but Non-Recurring Engineering (NRE) and production costs are high. FPGA are a cheaper alternative, offering high performance in terms of throughput, latency and energy efficiency. Also, they have the advantage of being reconfigurable, which allows the DNN model to be updated and/or adjusted after initial deployment (Table I).

In recent years, there has been a growing interest in using FPGA to implement neural networks accelerators, especially CNN, in both *edge* and *cloud*. In fact, the number of conference and journal articles available on IEEExplore [15] related to FPGA and CNN is significantly higher than the number of articles related to FPGA and other DNN architectures (Fig. 2). Although many of these works are focused on the implementation of CNN in high performance FPGA, it is already very common to find implementations in limited resources devices (suitable for the *edge*) [16]–[22], which is the focus of this work.

This document aims to describe various techniques for CNN model compression, optimization and implementation in resource limited FPGA. In chapter II, basic DNN and CNN concepts are presented. In chapter III, the most common CNN model compression techniques are discussed and, finally, chapter IV describes the landscape of automated implementation frameworks and optimization techniques for FPGA inference in the *edge*. Conclusions are given in chapter V.

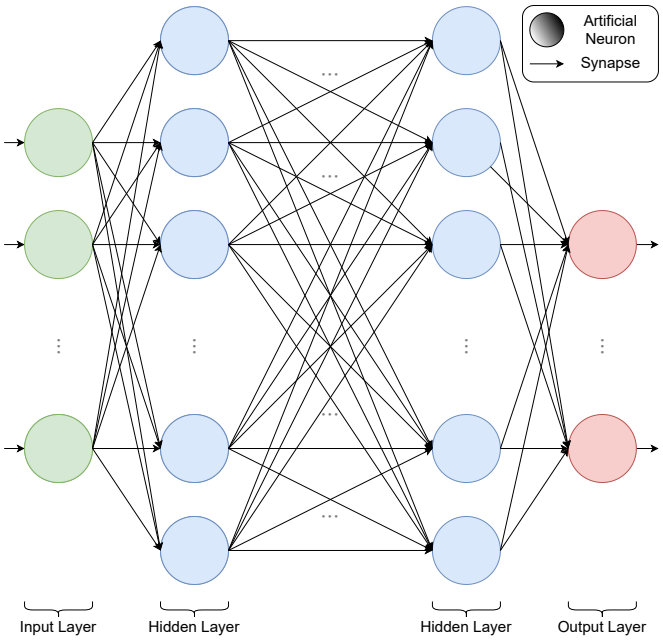


Figure 3: Artificial Neural Network (ANN) diagram

II. Neural Networks Background

CNN is a type of ANN that resort to convolutional layers, pooling layers and fully-connected layers. Basic concepts common to all ANNs are introduced in section II-A while key CNN concepts are presented in section II-B. Finally, the most popular and innovative CNN topologies, are described in II-C.

A. Artificial Neural Networks

ANN is inspired by the human brain, and its structure consists of an input layer, an output layer and one or more hidden layers. ANN with more than one hidden layer is referred to as DNN. All layers of an ANN are made up of artificial neurons interconnected to neurons in the neighboring layers (Fig. 3). The input layer receives input data and the output layer delivers the prediction result generated by the network. The hidden layers process the data so that predictions are generated. The output of each layer (except for the last) is called Output Feature Map (OFM) or activation map. All layers, except the first one, receive the OFM of the previous layer, in this case called Input Feature Map (IFM).

The internal structure of an artificial neuron is illustrated in Figure 4. A summation is applied to the product of the N entries (x_i) by the N weights (w_i) and the independent bias ($bias$). The result of the sum is then passed by an activation function (φ) that produces the output of the neuron (y), also called activation, as shown in equation (1).

$$y = \varphi(\cdot) = \varphi(x_1 \times w_1 + x_2 \times w_2 + \cdots + x_N \times w_N + bias) \quad (1)$$

There are several activation functions (φ), which should be chosen according to the application [24]. The most commonly used activation functions are the Rectified Liner Unit (ReLU) and Softmax. ReLU is the most popular in hidden layers as

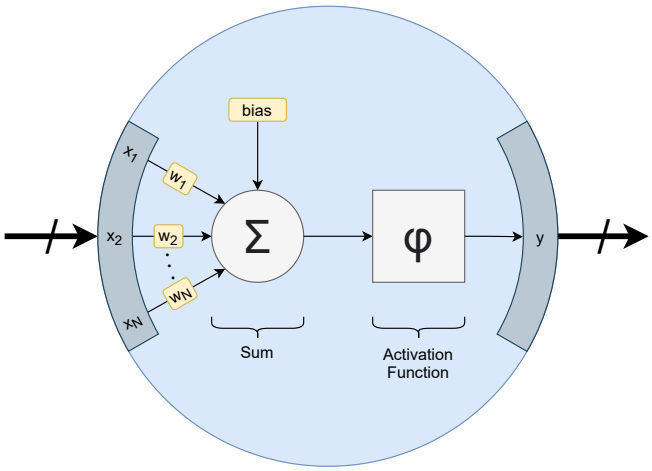


Figure 4: Artificial neuron structure.

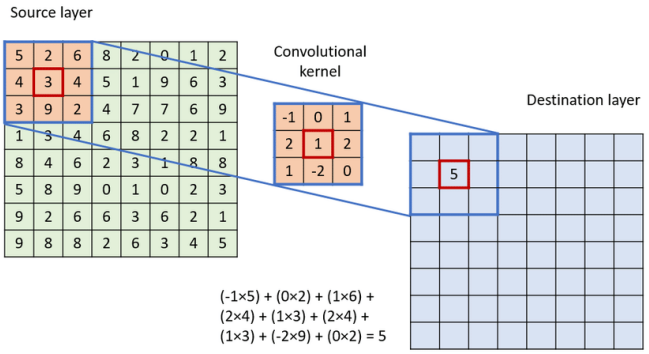


Figure 5: Convolution with a 3×3 filter kernel: center-kernel; left-input data; right-output convolution value [29].

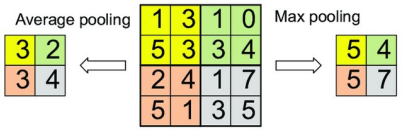


Figure 6: Pooling: max pooling (right) and average pooling (left) [30].

it usually outperforms other activation functions. When the network is used to make multi-class predictions it is common to use the Softmax function in the last layer. This function returns the probability of the predicted label belonging to a given class.

ANN prediction capability is acquired during the training stage. A representative set of labeled data feeds the network, and the error between the real and the predicted label will, in an iterative way, be used to update network parameters (backpropagation) [25]. At the end of the training phase, the model can be implemented to make predictions (inference) based on a given input. To help the designer in this process and increase productivity many open-source software libraries and deep learning frameworks have been released. Toolflows like TensorFlow, Keras, Caffe, PyTorch and others, provide high-level APIs and execution models to efficiently design and train neural networks [26].

B. CNN Key Concepts

CNN resort to convolution operations to automatically and efficiently extract features from structured data [27]. Convolution operations are implemented by sliding a filter (kernel), with $N \times N$ weights, across an IFM (or an image channel if the convolution is the first to be computed). Figure 5 illustrates the convolution process. A 3×3 kernel slides over the input data, and outputs a convolution value for each position. After the convolution operation, data is passed through an activation function, here omitted for simplicity.

CNN also include pooling and fully connected layers (FC). The first serve to sub-sample the image, reducing the complexity of the CNN and the risk of overfitting [28]. The most common pooling strategies are max pooling and average pooling. Figure 6 illustrates this process: in max pooling the highest value of a region is passed; while in average pooling the average of all values in the region is passed. FC layers are used to classify the features extracted in the convolution layers. In these layers all artificial neurons are connected to all previous layer's activations.

The number of parameters and operations on CNN is not evenly distributed across the layers. Convolutional layers are responsible for most operations (approximately 98%) and most parameters are associated with FC layers (approximately 93%), the remaining layers have a small contribution to the number of networks parameters and operations. Therefore, compression (section III) and optimization (section IV) techniques have a greater impact if applied in these layers. These intrinsic characteristics are shared among most of the CNN models [31].

C. CNN architectures

In recent years, several CNN network architectures have been proposed for different application requirements and constraints. Figure 7 illustrates the compromise between throughput and accuracy of some architectures for image classification using ImageNet dataset. The area of each circle is proportional to the required implementation memory (in Mega Bytes). As shown, there is a large number of alternative architectures with different performance and computational requirements. In this section we briefly review some of the most popular CNN architectures, showcasing their performance achievements but also their resource requirements [30], [32]–[35]. Most of these models (and their pre-trained weights) are also offered by deep learning frameworks and/or have shown success in competitions like the ImageNet Large Scale Visual Recognition Challenge (ILSVRC).

LeNet-5 [36] is seen as the classic architecture of CNN, having been designed for classification of handwritten digits. The network inputs 32×32 grayscaled images and consists of five hidden layers - the first three are convolutional and the last two are FC. With 60k parameters LeNet-5 achieved a test error of less than 1% in all its variants. Due to the results achieved, LeNet boosted the development of many other architectures and variants.

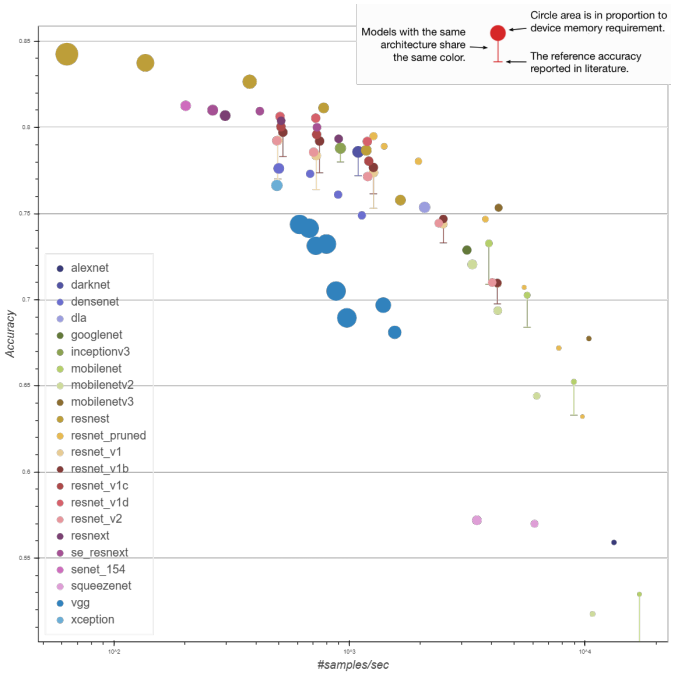


Figure 7: Throughput vs TOP-1 accuracy in popular CNN architectures [37].

AlexNet [38] was the first CNN to achieve good results for large image classification (227×227), serving as a basis for future implementations. This network consists of eight layers, five of which are convolutional and three are FC. Several innovative techniques were implemented, including the use of ReLU as an activation function in the hidden layers, to speed up the training process, and the dropout and data augmentation strategy to prevent overfitting. In 2012 AlexNet, got a TOP-5 validation error[1] of 16.4% in ILSVRC using ensemble prediction[2] and training data from the 2011 dataset. Individually, an AlexNet gets a TOP-5 validation error of 18.2% with 60M parameters.

VGG [39] achieved better results than AlexNet by increasing the number of hidden layers (depth). The authors propose VGG configurations with different depths, ranging from 11 to 19 deep layers, VGG-11 and VGG-19 respectively. It was demonstrated that the use of reduced kernels (*e.g.* 3×3) in cascade produces the same effect as a larger kernel (*e.g.* 5×5), using less parameters. In ILSVRC-2014 an ensemble of 7 VGG networks managed to achieve a TOP-5 validation error of 7.3%. A VGG-19 can achieve TOP-5 validation errors of 8.0% with 144M parameters.

GoogLeNet [40] was the winner of the ILSVRC-2014. This architecture introduced Inception Modules which, among other innovations, uses kernels of different sizes in order to extract features with different scales. The results achieved in ILSVRC-2014 were based on ensemble prediction, which guaranteed a TOP-5 validation error of 6.67% using a set of 7 models and reducing the parameters by around $12\times$ when compared to to AlexNet. Individually, a GoogLeNet model can achieve a TOP-5 error of 10.07%.

In 2015, an ensemble ResNet [41] won the ILSVRC competition with a TOP-5 validation error of 3.57%. This architecture uses additional shortcut connections, which skip one or more layers in order to eliminate the vanishing-gradient problem[3]. Individually, a ResNet with 152 layers gets a TOP-5 validation error of 4.49%, with 60M parameters [33].

In DenseNet [42] all layers are connected to the following layers. These networks can reduce the vanishing-gradient problem, are efficient in relation to the number of parameters, allow feature reuse, and improve their propagation. DenseNet can achieve better accuracy than RestNet using fewer parameters and computations. A DenseNet with 201 layers gets a TOP-5 validation error similar to that of a ResNet-101 ($\approx 6.34\%$), using about 20M parameters (about half the parameters of ResNet-101).

SqueezeNet [43] was designed with the objective of maintaining the accuracy of known CNN models, while reducing the number of parameters. This architecture introduces the Fire Module consisting of a squeeze layer, which uses 1×1 filters instead of 3×3 filters to reduce the number of parameters, followed by an expand layer, which applies 1×1 convolutions in parallel to 3×3 convolutions. This architecture achieved a TOP-5 validation error of 19.7%, comparable to AlexNet's TOP-5 validation error, with only 1.2M parameters.

MobileNet [44] was designed to be deployed on mobile devices and embedded vision applications. MobileNet uses separable convolutions in order to reduce architectural complexity in exchange for some reduction in accuracy. A MobileNet with 224 hidden layers got a TOP-5 validation error of 10.5% on ImageNet with 4.2M parameters.

Some architectures, such as ResNet and DenseNet, aim to achieve high accuracy leading to a reduction in throughput and, in general, to an increase in network size. Other architectures, such as SqueezeNet and MobileNet, exchange accuracy by a reduced network complexity, enabling real-time inference. Although any model can be compressed and optimized, these lighter architectures are generally preferred for inference in the *edge*, where resources are limited and accuracy requirements are less stringent.

III. Model Compression Techniques

CNN inference in the *edge* usually requires the use of a set of compression techniques that, if correctly applied, lead to a drastic reduction of used resources with variable degrees of accuracy loss. These techniques are most commonly applied during training and can be divided into 3 categories: pruning; quantization; and encoding.

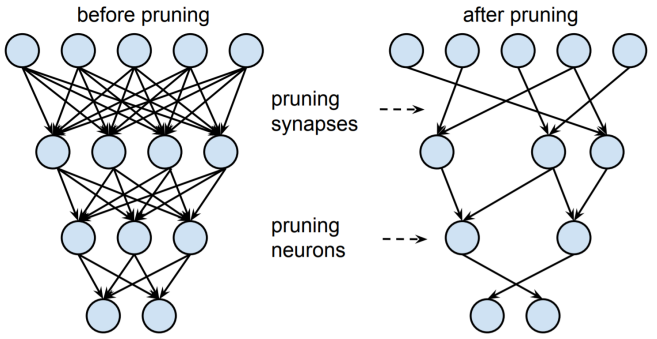
A. Pruning

Pruning is a strategy that aims to suppress weights and/or activations that have a negligible impact on the model accuracy

[1]TOP-5 Validation Error: when the correct label is not contained in the five most likely predicted labels

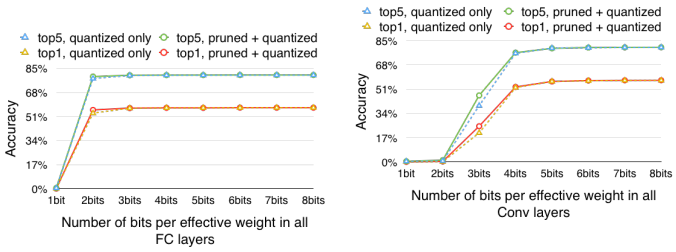
[2]Ensemble Prediction: the final output is the result of averaging the individual outputs of several independently trained models.

[3]Vanishing-Gradient Problem: During the training process a portion of the error is used to update the weights of an ANN. Sometimes this portion is too small making it hard to effectively train an ANN.



(a) Original neural network. (b) Sparse Neural Network.

Figure 8: Pruning effect on a generic neural network. [45].



(a) Compression in FC layers (b) Compression in convolutional layers

Figure 9: Quantization and pruning effect [50].

[45]. Figure 8 illustrates the pruning process of a generic DNN. The sparse network (Fig. 8b) results from the suppression of neurons and synapses from the original network (Fig. 8a). Pruning can be structural or non-structural, depending on whether it is applied to a weight or a layer, respectively [46]. Also, it can be applied to neurons or connections during the training phase (static pruning) or during inference (dynamic pruning) [47], although static pruning is the most effective and common approach.

Static pruning suppresses network parameters that do not meet a certain criterion. Usually, the model is (re)trained after the first pruning so that the remaining parameters are fine-tuned. The (re)training process aims to approximate the accuracy of the sparse and original networks [45], and usually is very successful. In [45], authors implement a static pruning strategy, and they manage to achieve a parameter reduction of $9\times$ and $13\times$ for AlexNet and VGG-16, respectively, with negligible accuracy loss.

While static pruning is applied in the training stage, dynamic pruning is applied during the inference. Depending on the model input (e.g. images with different qualities), different structures can be suppressed targeting to the smaller possible computational footprint. Although this technique can be adequate in some applications [48], globally it does not provide the same performance and compression (in computational resources and memory) when compared to a model that has undergone static pruning during the training phase.

B. *Quantization*

After training, CNN parameters are traditionally represented with a 32-bit floating point format (FP-32). However, most CNN implementations do not require this much precision [49] and quantization can be used to represent weights and/or activations using fewer bits. This reduces memory, bandwidth and energy at the expense of a possible loss of network accuracy. It can be done after training (Post Training Quantization), or it can be considered during the training phase (Quantization Aware Training), with a lower accuracy penalty [47].

In [50], authors show that quantizing up to 2 bits in fully-connected layers and up to 4 bits in convolutional layers has a negligible impact on the network accuracy. In the same paper authors show that a quantization with 8 bits in convolutional layers and 5 bits in the FC layers has no negative impact on the network accuracy (Fig. 9), and results in a $18\times$ compression rate. Although quantization has proven to be a very effective compression technique, it should be noted that different types of layers show different sensitivities to quantization, with convolutional layers being the most sensitive to this operation.

Extreme quantization has also been proposed to radically compress CNN models, maximizing efficiency in terms of memory and resource requirements in exchange for a degradation of accuracy. One or two bit quantized neural networks exchange accuracy for a high model compression [51], being good candidates for real-time deep learning implementations on FPGA and ASIC due to their bitwise efficiency.

Binary Neural Network (BNN) resort to 1-bit quantization in both weights and activations. This technique allows the implementation of convolution operations using binary operations (XNOR and bit-count), which makes it possible to reduce the size of networks and memory accesses. In [52] the authors quantized AlexNet achieving a TOP-5 accuracy of 60.1% on ImageNet with compressions of $32\times$ in memory. The authors in [49] also achieved a compression of $32\times$ by applying a quantization strategy for weights and activations on an AlexNet, achieving a TOP-5 accuracy of 69.2% on ImageNet. The same authors also explored the possibility of using 1-bit quantized weights and multi-bit single-precision activations to improve accuracy. To differentiate from BNN, these were called Binary Weight Network (BWN).

Ternary Weight Neural Network (TWN) have also been proposed and shown to have better accuracy than BNNs with similar computational complexity [53]. The weights of this type of network are coded with the value -1, 0 and +1, making its hardware implementation very efficient (since the additional value (zero) does not participate in the computations). The results show that TWN achieve compression rates $2\times$ lower than BWN, but with a degradation of the TOP-5 accuracy of only 1.8% with a ResNet-18B on ImageNet.

C. *Encoding*

Encoding takes advantage of parameter distribution to compress the model. In fact, after the training phase there are many parameters that share the same value. If the network is quantized, the sharing of values is even more accentuated, which makes encoding more advantageous. In addition to

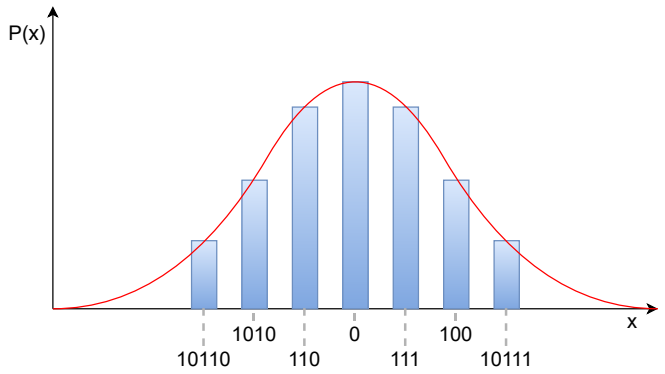


Figure 10: Illustration of Huffman Encoding based on the probabilistic distribution of weights.

having shared values, knowing their distribution provides an opportunity to optimize their storage [50].

There are several techniques proposed to this end. One of these techniques is based on cluster aggregation, enabling compression levels of $9\times$ with negligible losses in accuracy [54]. Another possibility is to use source codes, such as Huffman codes [50], where variable length binary codes are used to represent symbols, with the most common symbols being encoded with fewer bits (Fig. 10). Because the Huffman encoding is lossless, this technique does not degrade the accuracy of the network. In [50], authors take advantage of the distribution of weight values after quantization and pruning and employ Huffman encoding to achieve compressions of $8\times$ and $18\times$ for AlexNet and VGG-16, respectively, without accuracy degradation.

IV. CNN Implementation in Resource Limited FPGA

To deploy trained and compressed models in resource limited FPGA, the designer must choose the right methodology to efficiently convert his model to a hardware design. He may be worried about throughput, energy efficiency, hardware footprint, development time or design portability. At this section, different implementation frameworks and optimization techniques are discussed.

A. FPGA Accelerator Frameworks

FPGA accelerators exploit different levels of parallelism to increase throughput and enable low-cost real-time inference. Accelerator architectures can be generally classified as Single Computation Unit (SCU) or streaming. SCUs are very flexible as they can execute different networks using the same hardware. However, the hardware is not tailored to the target network and thus is not as optimized as in streaming architectures. Streaming accelerators offer a good throughput-latency tradeoff but at the cost of a larger hardware footprint, making these architectures better suited for smaller or highly quantized networks.

The most relevant FPGA vendors, Xilinx and Intel, offer AI development environments based on optimized IP cores, tools, libraries and models, making it simpler for users without FPGA knowledge to develop deep-learning inference applications. Vitis AI, from Xilinx, supports model quantization and prunning and other more sophisticated optimizations such as layer fusion, instruction scheduling, and memory tiling. Subsequent compilation tools enable DNN algorithms to be deployed in their Deep Learning Processor Unit (DPU) - a programmable engine with a specialized instruction set. OpenVINO toolkit offers a similar inference workflow for Intel Vision Processing Units (VPU) accelerators. Although very powerful, these AI development environments are generically based on SCUs, so they are not optimized for a specific network and usually target sophisticated hardware SoC FPGA.

Regarding streaming architectures, Xilinx Research Labs have also recently released an experimental framework (FINN) to automate the creation of fully customized inference engines for design space exploration in extreme quantized networks, both for the *edge* and *cloud* [55] [56]. CHaiDNN is also a fixed point precision open source accelerator framework from Xilinx, but specifically targeting the more sophisticated Xilinx UltraScale MPSoCs [57]. It uses High Level Synthesis (HLS) and runs convolutional layers in the FPGA and fully connected layers on CPU, therefore its architecture resembles a custom architecture rather than a streaming architecture.

HLS based toolkits, such as Vivado HLS or Intel FPGA OpenCL SDK, are also very popular in generating FPGA-based hardware designs from a high level of abstraction. Nevertheless, the efficiency of these HLS tools depend on the designer proficiency to map and schedule low-level primitive operations [58]. To circumvent this, hls4ml [59] has been developed to automatically generate streaming accelerators using Vivado HLS. By supporting Keras, Tensorflow, PyTorch and Onnx models as inputs, hls4ml provides a means to obtain customized hardware implementations of CNN, requiring minimal hardware design expertise. Other works have also resorted to HLS tools [60] [61], OpenCL based tools [62], or other less popular tools [63], to describe custom neural networks in a high-level programming language, and then algorithmically compile that code down to a Register Transfer Level (RTL) design specification. The end goal is to implement FPGA-based CNN accelerators with reduced human intervention. However, most of these frameworks are not targeting resource limited FPGA as the target is to build high-performance accelerators [64].

HADDOC2 [65] and DNNWEAVER [66] are also open-source frameworks to implement CNN hardware accelerators in Xilinx and Intel FPGA. They take Caffe high level description as input and automatically generate Hardware Description Language (HDL) code. In HADDOC2, the streaming architecture is generated following the exact topology of the network, so there are no configurable parameters. On the contrary, DNNWEAVER favors flexibility over customization with its SCU approach, but inefficiencies are introduced due to control mechanisms that resemble those of a processor. Other FPGA accelerator frameworks were proposed to generate synthesizable hardware, but are not publicly available [67] [68] [69] [70].

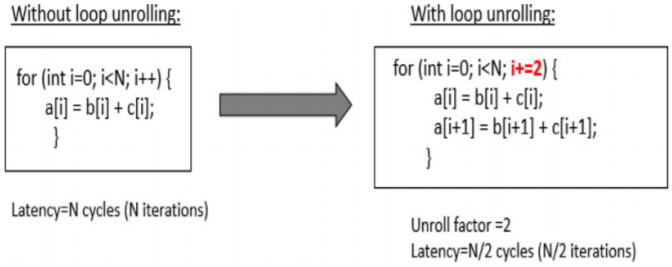


Figure 11: Illustrating loop unrolling technique with an unrolling factor of 2 [72].

B. Optimization Techniques

When resorting to HLS tools, low level optimization techniques are required. Loop unrolling (or unfolding) is one of those techniques that is used to parallelize a set of sequential instructions [71], whenever there is independence between iterations. This technique is usually associated with an unrolling factor (or unrolling parameter), which determines how many parallel tasks are executed in one iteration. Figure 11 illustrates the unrolling of a loop with an unrolling factor of 2. The original loop, with N iterations, is executed in N clock cycles, while the unrolled loop is executed in half that time. The disadvantage associated with this technique is the greater use of hardware resources and, consequently, of energy, which is relevant if the size of the loop or unrolling factor is large [72].

Pipelining is a technique that allows a task to be executed without the previous one being completed, that is, as soon as the necessary data for its execution is available. It can be applied whenever the following N tasks do not depend on data provided by the executing task. The Initiation Interval (II) defines the number of clock cycles between the beginning of consecutive tasks. Figure 12 illustrates the process of pipelining of a loop with an II of 1 clock cycle. The tasks for the next iteration start running as soon as it is possible to do a new read, compute, or write. Pipelining allows the designer to improve throughput without increasing the hardware resources [73].

Tiling methods, also known as loop tiling or loop blocking, aim to optimize external memory access times by efficiently storing data in cache tiles with temporal and/or spatial dependence in the processing cycles [74]. These methods change the order of data access in external memory so that they can be stored in internal memory until used [75]. Figure 13 illustrates the loop tiling process. If all rows and columns of a structure are loaded, it would not be possible to store that information in internal memory. Using the tiling strategy, small blocks of data can be loaded and stored in internal memory until they are needed in computations. Tiling requires data to be stored in memory with the organization imposed by this technique, otherwise its use can reduce the efficiency of memory accesses [74].

The Winograd algorithm is also a popular technique to reduce the hardware footprint of convolution operations. It optimizes the hardware implementation of convolutions by reducing the number of multiplications, and replacing them with

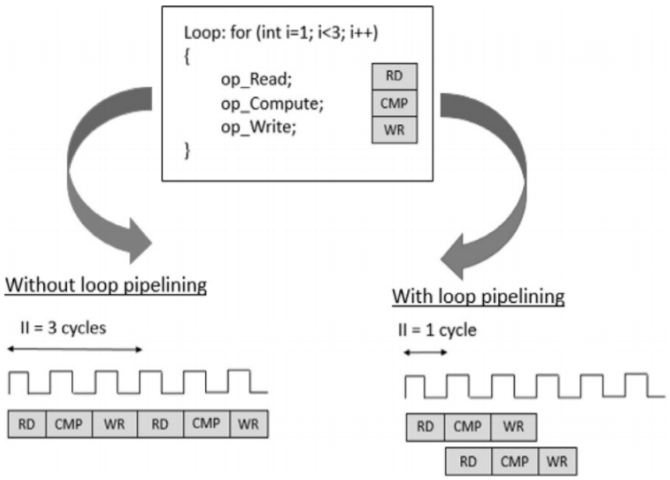


Figure 12: Pipelining technique with 1 clock cycle Initiation Interval [72].

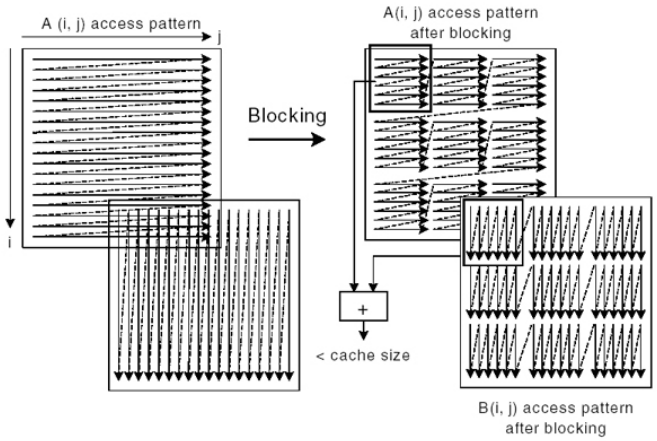


Figure 13: Tiling strategy illustrated. [76]

addition operations [77]. Although this algorithm reduces the number of computations, it increases the necessary bandwidth, being only beneficial for small kernels and steps strides [74].

V. Conclusion

This paper reviews fundamental AI concepts, CNN model compression techniques and presents an overview of FPGA Accelerator Frameworks, that should guide a system designer aiming to implement CNN models in limited resources FPGA. The CNN architecture should be chosen based on the application, taking advantage of the compromise that exists between *accuracy* and *throughput*. During the training and implementation phases, compression and optimization strategies can be adopted to reduce the hardware requirements.

In the training phase, pruning, quantization and coding strategies can be used together to significantly compress the model with limited accuracy degradation [50]. The (re)training of the network after applying techniques that cause loss of information, such as pruning and quantization, causes a negligible or inexistent loss on the model's accuracy. The use of lossless coding methods allows the model to be further compressed without affecting its accuracy.

In the implementation phase, techniques can be adopted to increase the throughput of the network, such as the parallelization of operations and the efficient use of available memory. These compression and optimization strategies, along with growing landscape of automated implementation frameworks, will continue to fuel the implementation of complex CNN in FPGA with reduced resources. Hence, CNN inference is no longer linked with computation exclusively in *cloud*, but being increasingly present in the *edge* paradigm.

REFERENCES

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[2] C. François, *Deep Learning with Python*, 1st ed. New York, NY: Manning Publications, 2017.

[3] IBM, "What is Machine Learning? — IBM." [Online]. Available: https://www.ibm.com/cloud/learn/machine-learning

[4] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015. [Online]. Available: https://doi.org/10.1038/nature14539

[5] D. Svozil, V. Kvasnicka, and J. Pospichal, "Introduction to multi-layer feed-forward neural networks," *Chemometrics and Intelligent Laboratory Systems*, vol. 39, no. 1, pp. 43–62, 1997. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169743997000610

[6] T. L. Fine, *Feedforward neural network methodology*. Springer Science & Business Media, 2006.

[7] L. Noriega, "Multilayer perceptron tutorial," *School of Computing. Staffordshire University*, 2005.

[8] E. Bisong, *The Multilayer Perceptron (MLP)*. Berkeley, CA: Apress, 2019, pp. 401–405. [Online]. Available: https://doi.org/10.1007/978-1-4842-4470-8_31

[9] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *CoRR*, vol. abs/1511.08458, 2015. [Online]. Available: http://arxiv.org/abs/1511.08458

[10] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 International Conference on Engineering and Technology (ICET)*, 2017, pp. 1–6.

[11] L. C. Jain and L. R. Medsker, *Recurrent Neural Networks: Design and Applications*, 1st ed. USA: CRC Press, Inc., 1999.

[12] I. Sutskever, *Training recurrent neural networks*. University of Toronto Toronto, Canada, 2013.

[13] Y. LeCun, "MNIST Demos on Yann LeCun's website." [Online]. Available: http://yann.lecun.com/exdb/lenet/index.html

[14] Y. Han, X. Wang, V. C. M. Leung, D. Niyato, X. Yan, and X. Chen, "Convergence of edge computing and deep learning: A comprehensive survey," *CoRR*, vol. abs/1907.08349, 2019. [Online]. Available: http://arxiv.org/abs/1907.08349

[15] "IEEE Xplore." [Online]. Available: https://ieeexplore.ieee.org/

[16] C. Lammie, A. Olsen, T. Carrick, and M. Rahimi Azghadi, "Low-power and high-speed deep fpga inference engines for weed classification at the edge," *IEEE Access*, vol. 7, pp. 51 171–51 184, 2019.

[17] C. Bao, T. Xie, W. Feng, L. Chang, and C. Yu, "A power-efficient optimizing framework fpga accelerator based on winograd for yolo," *IEEE Access*, vol. 8, pp. 94 307–94 317, 2020.

[18] G. Dinelli, G. Meoni, E. Rapuano, G. Benelli, and L. Fanucci, "An fpga-based hardware accelerator for cnns using on-chip memories only: Design and benchmarking with intel movidius neural compute stick," *International Journal of Reconfigurable Computing*, vol. 2019, p. 7218758, Oct 2019. [Online]. Available: https://doi.org/10.1155/2019/7218758

[19] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W. Hwu, and D. Chen, "FPGA/DNN co-design: An efficient design methodology for iot intelligence on the edge," *CoRR*, vol. abs/1904.04421, 2019. [Online]. Available: http://arxiv.org/abs/1904.04421

[20] X. Liu, D. H. Kim, C. Wu, and O. Chen, "Resource and data optimization for hardware implementation of deep neural networks targeting fpga-based edge devices," in *2018 ACM/IEEE International Workshop on System Level Interconnect Prediction (SLIP)*, 2018, pp. 1–8.

[21] N. Lin, H. Lu, X. Hu, J. Gao, M. Zhang, and X. Li, "When deep learning meets the edge: Auto-masking deep neural networks for efficient machine learning on edge devices," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, 2019, pp. 506–514.

[22] L. Yang, Z. He, and D. Fan, "A fully onchip binarized convolutional neural network fpga impelmentation with accurate inference," in *Proceedings of the International Symposium on Low Power Electronics and Design*, ser. ISLPED '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3218603.3218615

[23] A. Shimoni, "A gentle introduction to hardware accelerated data processing," pp. 1–12, 2018. [Online]. Available: https://hackernoon.com/a-gentle-introduction-to-hardware-accelerated-data-processing-81ac79c2105

[24] S. Sharma, S. Sharma, and A. Athaiya, "ACTIVATION FUNCTIONS IN NEURAL NETWORKS," *International Journal of Engineering Applied Sciences and Technology*, vol. 4, no. 12, pp. 310–316, 2020. [Online]. Available: https://www.ijeast.com/papers/310-316,Tesma412,IJEAST.pdf

[25] "Backpropagation: Intuition and Explanation — by Max Reynolds — Towards Data Science." [Online]. Available: https://towardsdatascience.com/backpropagation-intuition-and-derivation-97851c87eece

[26] K. Dinghofer and F. Hartung, "Analysis of criteria for the selection of machine learning frameworks," in *2020 International Conference on Computing, Networking and Communications (ICNC)*, 2020, pp. 373–377.

[27] IBM Cloud Education, "What are Convolutional Neural Networks? — IBM," 2020. [Online]. Available: https://www.ibm.com/cloud/learn/convolutional-neural-networks

[28] "What is Overfitting? — IBM." [Online]. Available: https://www.ibm.com/cloud/learn/overfitting

[29] D. Podareanu, V. Codreanu, S. Aigner, C. Leeuwen, and V. Weinberg, "Best practice guide - deep learning," Partnership for Advanced Computing in Europe (PRACE), Tech. Rep., 02 2019.

[30] M. Véstias, "A survey of convolutional neural networks on edge with reconfigurable computing," *Algorithms*, vol. 12, p. 154, 07 2019.

[31] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A survey of FPGA based neural network accelerator," *CoRR*, vol. abs/1712.08934, 2017. [Online]. Available: http://arxiv.org/abs/1712.08934

[32] M. Talib, S. Majzoub, Q. Nasir, and D. Jamal, "A systematic literature review on hardware implementation of artificial intelligence algorithms," *The Journal of Supercomputing*, vol. 77, 02 2021.

[33] M. Véstias, R. Duarte, J. Sousa, and H. Neto, "Moving deep learning to the edge," *Algorithms*, vol. 13, p. 125, 05 2020.

[34] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi, "A survey of the recent architectures of deep convolutional neural networks," *Artificial Intelligence Review*, vol. 53, no. 8, pp. 5455–5516, Dec 2020. [Online]. Available: https://doi.org/10.1007/s10462-020-09825-6

[35] S. Bianco, R. Cadène, L. Celona, and P. Napoletano, "Benchmark analysis of representative deep neural network architectures," *CoRR*, vol. abs/1810.00736, 2018. [Online]. Available: http://arxiv.org/abs/1810.00736

[36] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[37] "Classification — gluoncv 0.11.0 documentation." [Online]. Available: https://cv.gluon.ai/model_zoo/classification.html

[38] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, p. 84–90, May 2017. [Online]. Available: https://doi.org/10.1145/3065386

[39] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015.

[40] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. [Online]. Available: http://arxiv.org/abs/1409.4842

[41] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[42] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 2261–2269.

[43] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016. [Online]. Available: http://arxiv.org/abs/1602.07360

[44] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[45] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," *ArXiv*, vol. abs/1506.02626, 2015.

[46] S. Lin, R. Ji, C. Yan, B. Zhang, L. Cao, Q. Ye, F. Huang, and D. Doermann, "Towards optimal structured cnn pruning via generative adversarial learning," 2019.

[47] T. Liang, J. Glossner, L. Wang, and S. Shi, "Pruning and Quantization for Deep Neural Network Acceleration: A Survey," Tech. Rep., 2021.

[48] F. Nikolaos, I. Theodorakopoulos, V. Pothos, and E. Vassalos, "Dynamic pruning of cnn networks," in *2019 10th International Conference on Information, Intelligence, Systems and Applications (IISA)*, 2019, pp. 1–5.

[49] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," 2016.

[50] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv: Computer Vision and Pattern Recognition*, 2016.

[51] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," 2018.

[52] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, Eds., vol. 29. Curran Associates, Inc., 2016. [Online]. Available: https://proceedings.neurips.cc/paper/2016/file/d8330f857a17c53d217014ee776bfd50-Paper.pdf

[53] F. Li and B. Liu, "Ternary weight networks," *CoRR*, vol. abs/1605.04711, 2016. [Online]. Available: http://arxiv.org/abs/1605.04711

[54] M. Samragh, M. Ghasemzadeh, and F. Koushanfar, "Customizing neural networks for efficient fpga implementation," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, pp. 85–92.

[55] 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 Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 3, Dec. 2018. [Online]. Available: https://doi.org/10.1145/3242897

[56] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers, "Finn-¡i¿r¡/i¿: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 3, Dec. 2018. [Online]. Available: https://doi.org/10.1145/3242897

[57] "CHaiDNN." [Online]. Available: https://github.com/Xilinx/CHaiDNN

[58] S. I. Venieris, A. Kouris, and C. Bouganis, "Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions," *CoRR*, vol. abs/1803.05900, 2018. [Online]. Available: http://arxiv.org/abs/1803.05900

[59] "hls4ml." [Online]. Available: https://fastmachinelearning.org/hls4ml/

[60] D. Gschwend, "Zynqnet: An fpga-accelerated embedded convolutional neural network," *CoRR*, vol. abs/2005.06892, 2020. [Online]. Available: https://arxiv.org/abs/2005.06892

[61] "RFNoC Neural Network." [Online]. Available: https://github.com/Xilinx/RFNoC-HLS-NeuralNet

[62] A. Ghaffari and Y. Savaria, "Cnn2gate: An implementation of convolutional neural networks inference on fpgas with automated design space exploration," *Electronics*, vol. 9, no. 12, 2020. [Online]. Available: https://www.mdpi.com/2079-9292/9/12/2200

[63] A. Jahanshahi, "Tinycnn: A tiny modular CNN accelerator for embedded FPGA," *CoRR*, vol. abs/1911.06777, 2019. [Online]. Available: http://arxiv.org/abs/1911.06777

[64] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen, "Cloud-dnn: An open framework for mapping dnn models to cloud fpgas," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 73–82. [Online]. Available: https://doi.org/10.1145/3289602.3293915

[65] "Haddoc2 : Hardware Automated Dataflow Description of CNNs." [Online]. Available: https://github.com/DreamIP/haddoc2

[66] "dnnweaver." [Online]. Available: http://dnnweaver.org/

[67] S. I. Venieris and C.-S. Bouganis, "fpgaconvnet: A framework for mapping convolutional neural networks on fpgas," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 40–47.

[68] S. Liang, C. Liu, Y. Wang, H. Li, and X. Li, "Deepburning-gl: an automated framework for generating graph neural network accelerators," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–9.

[69] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping cnn onto embedded fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 35–47, 2018.

[70] V. Gokhale, A. Zaidy, A. X. M. Chang, and E. Culurciello, "Snowflake: An efficient hardware accelerator for convolutional neural networks," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017, pp. 1–4.

[71] M. Wolf, "Chapter 5 - program design and analysis," in *Computers as Components (Third Edition)*, 3rd ed., ser. The Morgan Kaufmann Series in Computer Architecture and Design, M. Wolf, Ed. Boston: Morgan Kaufmann, 2012, pp. 213–306. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123884367000052

[72] M. Makni, S. Niar, M. Baklouti, and M. Abid, "Hape: A high-level area-power estimation framework for fpga-based accelerators," *Microprocessors and Microsystems*, vol. 63, pp. 11–27, 08 2018.

[73] "Intel High Level Synthesis Compiler Pro Edition: Best Practices Guide." [Online]. Available: https://www.intel.com/content/www/us/en/programmable/documentation/nml1505158467345.html#uny1508901514896

[74] S. Mittal, "A survey of fpga-based accelerators for convolutional neural networks," *Neural Computing and Applications*, vol. 32, 02 2020.

[75] R. Ayachi, Y. Said, and A. Ben Abdelali, "Optimizing neural networks for efficient fpga implementation: A survey," *Archives of Computational Methods in Engineering*, Jan 2021. [Online]. Available: https://doi.org/10.1007/s11831-021-09530-9

[76] "How to Use Loop Blocking to Optimize Memory Use on 32-Bit Intel®..." [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/how-to-use-loop-blocking-to-optimize-memory-use-on-32-bit-intel-architecture.html?wapkw=(ia-32+manual)

[77] C. Bao, T. Xie, W. Feng, L. Chang, and C. Yu, "A power-efficient optimizing framework fpga accelerator based on winograd for yolo," *IEEE Access*, vol. PP, pp. 1–1, 05 2020.