UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Optimizing Deep Neural Networks for Deployment on Reconfigurable Architectures using Reinforcement Learning

Diploma Thesis

Ioanna-Maria Panagou

Supervisor: Nikolaos Bellas

July 2024

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Optimizing Deep Neural Networks for Deployment on Reconfigurable Architectures using Reinforcement Learning

Diploma Thesis

Ioanna-Maria Panagou

Supervisor: Nikolaos Bellas

July 2024

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Βελτιστοποίηση Μοντέλων Βαθιάς Μάθησης για Εφαρμογή σε Επαναπροσδιοριζόμενες Αρχιτεκτονικές χρησιμοποιώντας Ενισχυτική Μάθηση

Διπλωματική Εργασία

Ιωάννα-Μαρία Πανάγου

Επιβλέπων/πουσα: Νικόλαος Μπέλλας

Ιούλιος 2024

Approved by the Examination Committee:

Supervisor **Nikolaos Bellas**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Spyros Lalis**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Dimitrios Katsaros**
Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

First and foremost, I would like to thank my supervisor, Prof. Nikolaos Bellas, for his significant influence on my academic interests and career. His guidance helped me discover new paths and improve my technical, writing, and interpersonal skills. I attribute many of my academic achievements to his mentorship. In the same vein, I also want to thank the members of the Computer Systems Lab (CSL) for their support and collaboration. Working with all of you has been a rewarding experience.

Additionally, I want to express my gratitude to my family for their unwavering support and encouragement throughout this journey. A special thanks to my mother, whose love, sacrifices, and endless encouragement have been the foundation of all my accomplishments.

Lastly, I would like to extend my gratitude to my partner, Theodoros Aslanidis, for his understanding, patience, and support throughout this process. Your encouragement has been invaluable.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Ioanna-Maria Panagou

Diploma Thesis

Optimizing Deep Neural Networks for Deployment on Reconfigurable Architectures using Reinforcement Learning

Ioanna-Maria Panagou

Abstract

Deep neural networks (DNNs) have become ubiquitous in various applications, driven by their impressive performance in tasks ranging from image recognition [1] to natural language processing [2]. As these networks continue to grow in size and complexity, training is predominantly conducted on powerful GPUs within large data centers. However, inference often needs to occur on smaller, resource-constrained devices. Field-Programmable Gate Arrays (FPGAs) are ideal for such deployments due to their ability for hardware customization, low latency, and energy efficiency [3].

The mapping of a DNN to hardware is challenging, as DNNs prioritize accuracy over hardware efficiency, resulting in significant redundancy [4]. Compression techniques like quantization can exploit this redundancy to create hardware-friendly networks without sacrificing much accuracy [5]. FINN [6] is a compiler tool that maps quantized networks, particularly convolutional neural networks (CNNs), to hardware but requires extensive user input.

This thesis proposes automating the DNN-to-FPGA mapping process of FINN using reinforcement learning (RL). Our approach employs RL to explore the vast design space of DNN accelerators, trading off accuracy and throughput while adhering to resource constraints. By reducing the manual effort needed for FPGA deployment, our method simplifies the process and enhances the efficiency of DNNs in resource-limited environments.

Keywords:

Deep Learning, Accelerators, FPGA, FINN, Quantization, Reinforcement Learning

Διπλωματική Εργασία

Βελτιστοποίηση Μοντέλων Βαθιάς Μάθησης για Εφαρμογή σε Επαναπροσδιοριζόμενες Αρχιτεκτονικές χρησιμοποιώντας Ενισχυτική Μάθηση

Ιωάννα-Μαρία Πανάγου

Περίληψη

Τα Βαθιά Νευρωνικά Δίκτυα (DNNs) έχουν κυριαρχήσει σε πολλές εφαρμογές, λόγω της εντυπωσιακής τους απόδοσης σε καθήκοντα που εκτείνονται από την αναγνώριση εικόνων [1] μέχρι την επεξεργασία λόγου [2]. Όσο αυτά τα δίκτυα μεγαλώνουν σε μέγεθος και πολυπλοκότητα, η εκπαίδευσή τους γίνεται σε ισχυρές GPUs μέσα σε μεγάλα κέντρα δεδομένων. Όμως, η εφαρμογή τους συχνά πρέπει να τελεστεί σε μικρότερες συσκευές με περιορισμένους πόρους. Οι Συστοιχίες Πυλών Προγραμματιζόμενου Πεδίου (Field-Programmable Gate Arrays - FPGAs) είναι ιδανικές για τέτοιες υλοποιήσεις λόγω της προσαρμοστικότητας του υλικού, της χαμηλής καθυστέρησης και της ενεργειακής αποδοτικότητας [3].

Η υλοποίηση ενός DNN σε υλικό είναι πρόκληση, καθώς τα DNNs δίνουν προτεραιότητα στην ακρίβεια παρά στην αποδοτικότητα του υλικού, οδηγώντας σε σημαντικό πλεονασμό [4]. Τεχνικές συμπίεσης όπως ο κβαντισμός εκμεταλλεύονται αυτόν τον πλεονασμό, δημιουργώντας δίκτυα φιλικά προς το υλικό χωρίς να θυσιάσουν σημαντικά την ακρίβεια [5]. Το FINN [6] είναι μεταγλωτιστής που υλοποιεί ποσοτικοποιημένα δίκτυα και κυρίως συνελικτικά νευρωνικά δίκτυα (CNNs) σε υλικό, αλλά απαιτεί εκτενή ανθρώπινη παρέμβαση.

Αυτή η διπλωματική προτείνει την αυτοματοποίηση της διαδικασίας του FINN που υλοποιεί ένα DNN σε FPGA, κάνοντας χρήση ενισχυτικής μάθησης (Reinforcement Learning - RL). Η προσέγγισή μας χρησιμοποιεί RL για να εξερευνήσει τον τεράστιο σχεδιαστικό χώρο των επιταχυντών DNN, εξισορροπώντας την ακρίβεια και την απόδοση υπό περιορισμούς πόρων. Μειώνοντας την ανάγκη για χειροκίνητη παρέμβαση στην υλοποίηση για FPGA, η μέθοδός μας απλοποιεί τη διαδικασία και βελτιώνει την αποδοτικότητα των DNN σε περιβάλλοντα περιορισμένων πόρων.

Λέξεις-κλειδιά:

Βαθιά Μάθηση, Επιταχυντές, FPGA, FINN, Κβαντισμός, Ενισχυτική Μάθηση

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Motivation

Deep Neural Networks (DNNs) have revolutionized numerous fields by demonstrating, and often exceeding, human-level performance in various tasks. More specifically, Convolutional Neural Networks (CNNs) are arguably the most successful class of DNNs to date. Although particularly suited for visual tasks, such as image classification [8], object detection [9] and image segmentation [10], CNNs have also been used in natural language processing (NLP) tasks, like sentence classification [11], in recommendation systems [12] and even in speech and audio processing, such as music genre classification [13]. However, their impressive performance comes at the cost of significant computational and storage requirements, and therefore, extensive research has been done around the topic of hardware accelerators for DNNs [14] for various platforms. For example, CPU architectures have been extended with the addition of specialized AI Single Instruction Multiple Data (SIMD) units, but their performance is still inferior to other platforms [15]. GPUs have been the preferred platform for DNN training and inference, especially since they have added special tensor units, but are very power-hungry devices [16]. ASICs, such as Google Tensor Processing Units (TPUs), have also dominated the hardware accelerator domain but are complex and costly to design and validate [17]. Field Programmable Gate Arrays (FPGAs) appear to be a promising alternative to ASICs because of their reprogrammable nature and have also been used for DNN acceleration.

Interestingly, it has been proven that, even though DNNs owe their success to their deep structures and billions of parameters, there is still a high level of redundancy [4]. This redun-

dancy can be leveraged to create efficient hardware accelerators using network compression techniques, such as quantization. Quantization in neural networks involves reducing the precision of the weights and activations of the network from floating-point numbers to lower-bit integers, such as 8-bit or even binary values. This process significantly reduces the model size and computational requirements, making it more efficient for deployment on hardware with limited resources, such as mobile devices and embedded systems. Since FPGAs are customizable at the gate level, they are ideal candidates for implementing Quantized Neural Networks (QNNs).

The design space of DNN quantization is typically immense, making it impractical to explore using brute-force methods. Neural Architecture Search (NAS) encompasses algorithms and heuristics designed to explore the design space of DNNs efficiently. Most NAS methods ignore, however, the underlying hardware and are designed only to optimize ML accuracy and not other aspects, such as latency, throughput, energy consumption, or area. Reinforcement Learning (RL) can be used as a NAS technique to automatically discover optimal neural network architectures and, more specifically, quantization levels in each layer of the architecture. In this approach, an RL agent explores the space of possible architectures, using rewards based on the performance of each architecture and the hardware resources it requires to guide its search and iteratively improve the design.

FINN is an open-source framework developed by Xilinx for generating fast and efficient inference of quantized deep neural networks (QNNs) on FPGAs [6]. FINN allows the design of custom dataflow architectures to enable the creation of highly optimized hardware accelerators that can efficiently process DNNs on FPGAs. It was developed to initially target Binarized Neural Networks (BNNs) and, subsequently, QNNs for FPGA inference. Even though it is a highly automated tool, significant user input is still required for both the quantization strategy and determining the desired level of parallelization. A work that successfully performs the above exploration is HAQ [18], which leverages RL to determine an optimal quantization strategy for each layer under latency and energy constraints by receiving direct feedback from a simulator of the target platform [18]. Even if their framework exhibits outstanding performance in various settings, it only targets systolic array-like architectures and not streaming ones where the parallelism for each layer has to be explicitly defined.

Our work introduces, implements, and evaluates a new quantization-aware algorithmic flow that uses RL to explore the design space of neural networks and their mapping onto

reconfigurable platforms. The flow selects an optimal (or near-optimal) DNN FPGA implementation under latency, area, and accuracy constraints set by the user and the capabilities of the FPGA. Using quantization, this automation can assist a hardware design team in rapidly and efficiently evaluating the Pareto optimal space of a large number of DNN designs spanning latency, area, and accuracy.

1.2 Thesis Contributions

The contributions of this thesis can be summarized as follows:

- We introduce an RL-based algorithm that optimizes quantization strategies for neural networks. This algorithm is designed to maximize accuracy while adhering to resource and latency constraints, specifically targeting FINN-generated streaming architectures.
- We propose a method for defining the parallelism of each neural network layer. This method aims to enhance performance while ensuring that all proposed designs are feasible within the given constraints.
- We conduct comprehensive evaluations of our proposed methods using well-known CNN architectures and deploying our accelerators on a high-performance FPGA platform to demonstrate its effectiveness and efficiency.

1.3 Outline

This thesis is organized into the following chapters.

- Chapter 2 presents the necessary background information to facilitate a comprehensive understanding of the thesis.
- Chapter 3 provides a detailed description of all the individual components of our workflow.
- Chapter 4 presents the results of applying our workflow to implement well-known CNN architectures for image classification on an actual device.
- Chapter 5 concludes the thesis and points to future directions for improvement.

Chapter 2

Background

This chapter explores the fundamental concepts necessary to understand the key themes of the thesis. Following a brief overview of DNNs, CNNs, and their training and inference procedure, we elaborate on the techniques, architectures, and tools that can be utilized to develop efficient CNN accelerators for FPGAs. The emphasis is on FINN, a compiler framework developed by Xilinx, which provides an end-to-end flow for implementing QNNs on FPGAs. Lastly, we describe RL and its role in automating the design of neural network architectures through NAS.

2.1 Convolutional Neural Networks

This work focuses on deploying CNNs for image classification, although the underlying principles can be adapted to other types of neural networks and tasks. CNNs are a specialized type of DNNs primarily used for visual tasks, such as image classification. The goal of a CNN is to approximate a mapping $\mathbf{x}_i \rightarrow y_i \quad \forall i$, each pair $\{\mathbf{x}_i, y_i\}$ representing a training sample where x_i is an input image and y_i a number representing the class the image belongs to.

The input images are typically represented as three-dimensional arrays, known as feature maps. The first dimension is the channel axis, while the last two dimensions correspond to the spatial dimensions (height and width) of the image. CNNs operate by extracting hierarchical features or patterns from these images. Each layer in a CNN transforms the input feature map into a new feature map, emphasizing specific features such as corners and edges. The primary layers that make up CNNs include Fully-Connected Layers, Convolutional Layers, Pooling

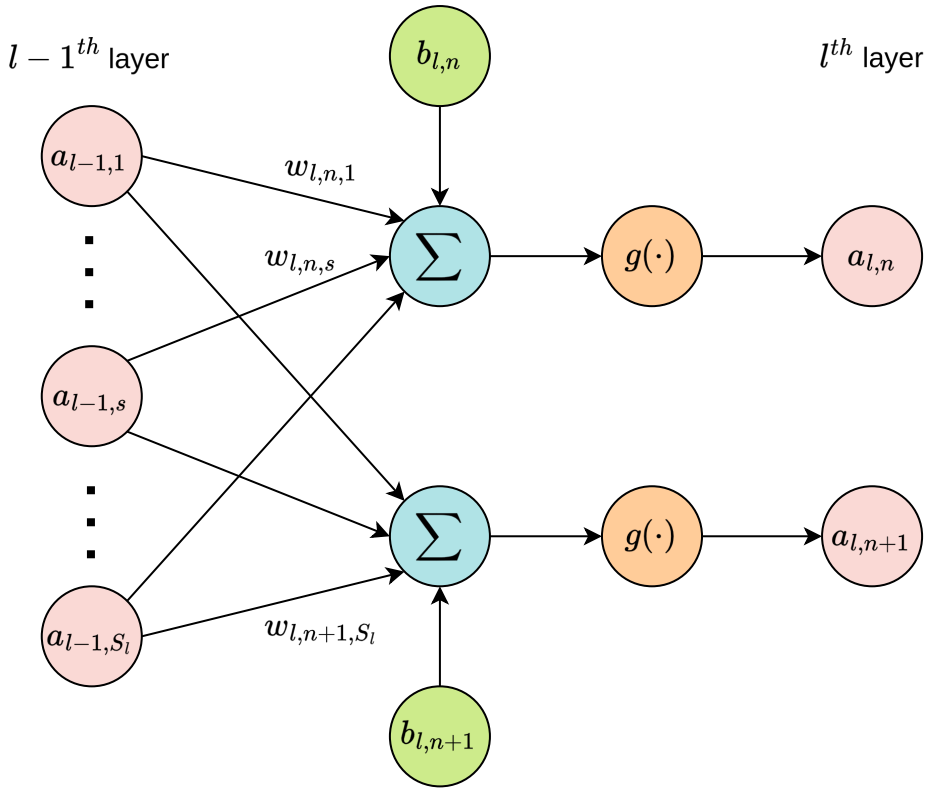


Figure 2.1: Computation of activations in an FC Layer.

Layers, and Batch Normalization (BN) Layers.

2.1.1 Fully-Connected Layers

Fully-Connected (FC) layers, also known as dense layers, are the traditional type of neural network layers where each neuron is connected to every neuron in the previous layer. These layers are typically found towards the end of the network and are used to integrate the features extracted by previous layers to perform the final classification. More formally, the output activation $a_{i,j}$ for the n^{th} neuron in the l^{th} layer can be computed as:

$$a_n^{(l)} = g\left(\sum_{s=1}^{S^{(l)}} w_{n,s}^{(l)} \cdot a_s^{(l-1)} + b_n^{(l)}\right) \tag{2.1}$$

where $w_{n,s}^{(l)}$ is the weight of the s^{th} synapse connected to the input of the n^{th} neuron in the l^{th} layer, $b_n^{(l)}$ is the bias term, $S^{(l)}$ is the number of synapses connected to each neuron in the l^{th} layer and g is a nonlinear activation function (see Figure 2.1).

The preferred choice of activation function in modern neural networks is the ReLU (Rec-

tified Linear Unit) activation function, which takes the form:

$$g(x) = (x)_+ = max(0, x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases} \tag{2.2}$$

Another popular choice, especially in early instances of neural networks, is the sigmoid activation function:

$$g(x) = \frac{e^x}{1 + e^x} = \frac{1}{1 + e^{-x}} \tag{2.3}$$

2.1.2 Convolution Layers

Convolution layers are the core building blocks of CNNs. They apply a set of filters (also known as kernels) to the input data to extract local features such as edges, textures, and patterns. Each filter slides across the input image, performing a convolution operation that results in a feature map. These layers are responsible for learning spatial hierarchies by detecting increasingly complex features at each layer. This layer takes advantage of the fact that features can exist anywhere in the image, structuring its neurons in a way that each neuron only looks at a local region of the input feature map. Hence, the output activation $a^{(l)}_{k,i,j}$ of the neuron in location (i, j) of channel k in convolution layer l is equal to:

$$a^{(l)}_{k,i,j} = g\left(\sum_{c=1}^{C_{in}} \sum_{k_i=0}^{K_H-1} \sum_{k_j=0}^{K_W-1} w^{(l)}_{k,c,k_i,k_j} \cdot a^{(l-1)}_{c,i+k_i,j+k_j} + b^{(l)}_k\right) \tag{2.4}$$

where C_{in} is the number of input channels and K_H and K_W are the dimensions of the convolution window (see Fig. 2.2).

2.1.3 Pooling Layers

Pooling layers are similar in nature to the convolution layers, but instead of convolution, they perform a non-linear operation. The most popular pooling layer is the max pooling layer, which downsamples the output activation by selecting the maximum value within a defined window or region, thereby retaining the most significant features. Pooling is an inherently lossy operation and is used to reduce the amount of computation of the subsequent layers.

2.1.4 Batch Normalization

Batch Normalization (BN) [19] is a technique used to standardize the inputs to a layer by adjusting and scaling the activations to have a mean of zero and a variance of one, thereby

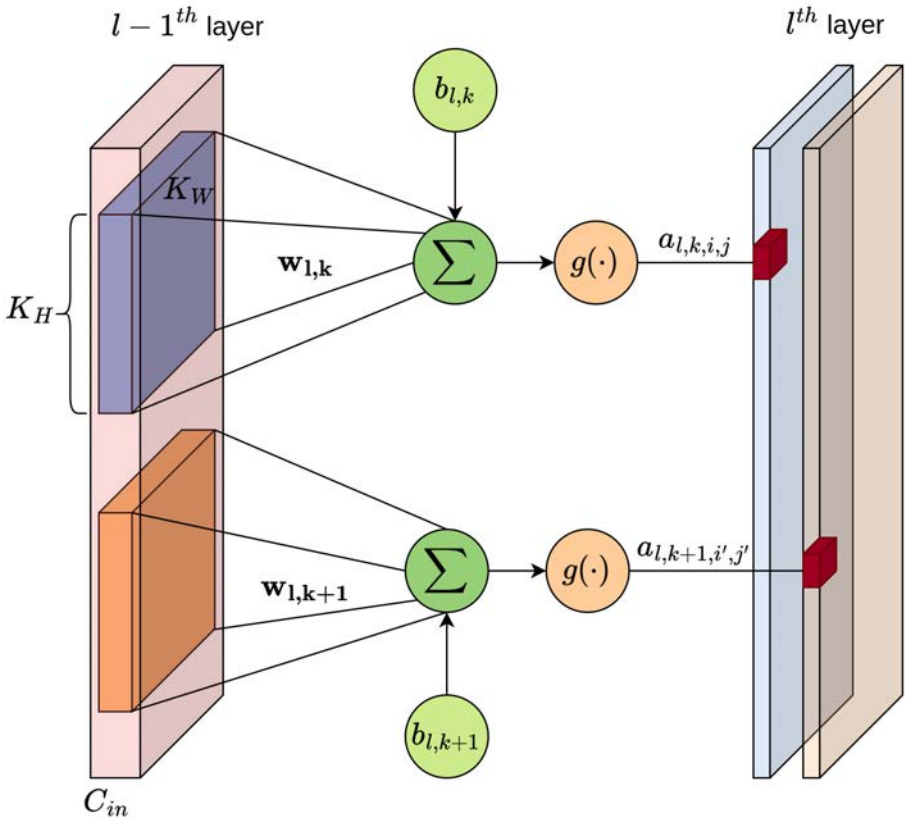


Figure 2.2: Computation of activations in a CNN layer.

improving training speed and stability. Inserting batch normalization layers after fully connected layers or convolution layers explicitly forces the activations through a network to take on a unit Gaussian distribution, which accelerates convergence. Let $\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$ be the mean of a batch of size m and $\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$ be the variance of the batch, the output of the batch normalization layer is normalized and the scaled and shifted by learnable parameters γ and β as so:

$$y_i = \gamma \cdot \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta \tag{2.5}$$

where ϵ is a very small value added for numerical stability.

BN layers exhibit different behaviors during training and inference. During the training passes, statistics are collected for the means and standard deviations of the training batches so that these metrics are used during inference. Thus, they are considered constant values, and, therefore, a BN layer may be merged with the preceding FC or convolution layer by updating the weights and biases of the latter in a process called BN folding or BN merging [20].

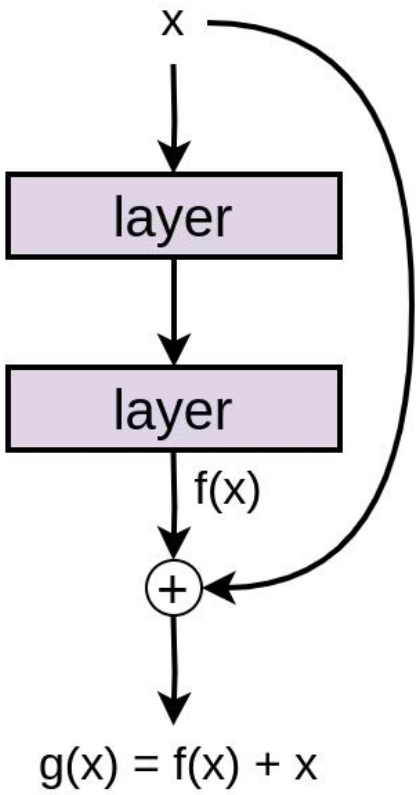


Figure 2.3: Example of Residual Connection

2.1.5 Residual Networks

A ResNet model, short for Residual Network, is a type of deep neural network architecture that addresses vanishing gradients, which often occur when training very deep networks [21]. The core idea of ResNets is the residual block, which allows networks to learn residual functions with reference to the layer inputs, which can be formulated as $y(x) = f(x) + x$, where x is the input (see Fig. 2.3).

2.1.6 Training and Inference

Neural Networks are deployed in a two-step process: training and inference. During training, batches from the input dataset are continuously fed to the network, the output of the network is compared to the ground truth label, and the weights w of the network are updated using gradient descent until no further error reduction in error is achieved (convergence). In CNNs, the inputs are images, and the output is the label of the class to which the network assigns the image.

The weight update rule in neural networks using gradient descent is given by:

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \frac{\partial L}{\partial w_{ij}^{(l)}} \tag{2.6}$$

where:

- $w_{ij}^{(l)}$ is the weight from neuron i in layer $l-1$ to neuron j in layer l.
- η is the learning rate, a hyperparameter that determines the step size for each iteration of weight updates.
- $\frac{\partial L}{\partial w_{ij}^{(l)}}$ is the gradient of the loss function L with respect to the weight $w_{ij}^{(l)}$.

Training is a very computationally intensive process and is usually performed on dedicated GPUs. There are multiple methods to accelerate training and improve convergence properties, such as data preprocessing, proper weight initialization and regularization. Inference is performed on a trained machine learning model. The trained model is fed input that it has not previously seen and, for the task of image classification, predicts which class the input images belong to. Since the task of inference does not involve backpropagation and weight updates, it is less expensive. Hence, inference can be performed on power- and memory-constrained devices.

2.2 DNN Deployment on FPGAs

Real-time deployment of Deep Neural Networks can be challenging due to significant computation, storage and energy costs. FPGAs are ideal candidates for DNN deployment, but porting a DNN on an FPGA is not a straightforward procedure, as it is subject to resource constraints. Network compression techniques have shown promising results in reducing the size of a trained neural network without sacrificing accuracy. A frequently used network compression technique is quantization.

2.2.1 Quantization

Quantization refers to the set of techniques for performing computation and storing vectors at lower bitwidths. Neural networks are typically trained in floating-point arithmetic, since numerical accuracy is critical to convergence, but contain a lot of redundant information, which can be exploited to reduce the cost of inference by moving to low-precision integer arithmetic. The benefits of integer-only inference are threefold:

1. Integer compute units require fewer resources than floating-point units, enabling the instantiation of more compute units, increasing parallelism, and reducing inference latency.

2. Quantized vectors have a significantly lower memory footprint than their floating-point representation, thus more weights can fit into on-chip memory, reducing both energy consumption for off-chip memory accesses and bandwidth requirements.

3. Integer computations consume less power than floating-point operations.

Note that to reap the benefits of integer-only inference, all activations, weights, and biases should be quantized.

Quantization formulation

Quantization maps a floating point number $x \in [a, b]$ to an integer in the range $x_q \in [a_q, b_q]$. The de-quantization process can be written as:

$$x = c(x_q + d) \tag{2.7}$$

and the quantization process as:

$$x_q = round\left(\frac{1}{c}x - d\right) \tag{2.8}$$

To find c and d, we just solve the system of equations:

$$\begin{aligned} a &= c(a_q + d) \\ b &= c(b_q + d) \end{aligned} \tag{2.9}$$

The solution for c and d is:

$$\begin{aligned} c &= \tfrac{b-a}{b_q-a_q} \\ d &= \tfrac{ab_q-ba_q}{b-a} \end{aligned} \tag{2.10}$$

We will also need to ensure that 0.0 remains precisely zero in both representations, so the following equation should apply:

$$x_q = round\left(\frac{1}{c}0.0 - d\right) \implies \tag{2.11}$$

$$x_q = round(-d) \tag{2.12}$$

so d should be equal to $round(d)$:

$$d = round\left(\frac{ab_q - ba_q}{b - a}\right) \tag{2.13}$$

Parameters c and d are called scale s and zero point z, respectively, so the de-quantization and quantization equations can be written as:

$$x = s(x_q + z) \tag{2.14}$$

and

$$x_q = clip(round(\frac{1}{s}x - z), a_q, b_q) \tag{2.15}$$

where the clipping operation constrains the quantized number to the predetermined range. Typically, a and b are determined using training statistics, whereas a_q and b_q are the minimum and maximum numbers that can be represented using a specified bitwidth. For example, if we wish to quantize to INTb, then $[a_q, b_q] = [-2^{b-1}, 2^{b-1} - 1]$ or $[0, 2^b - 1]$ for UINTb.

More details about quantization are provided in Appendix A.

Quantization-Aware Training (QAT) vs Post-Training Quantization (PTQ)

Depending on whether the reduced precision is considered during the training process, we can distinguish three approaches to working with quantized models:

1) **Quantization-Aware Training (QAT)** is a technique designed to minimize accuracy loss when reducing the precision of neural network computations. QAT simulates the effects of integer-only inference during the training phase, allowing the model to adapt to precision constraints before deployment. This is achieved by inserting fake quantization nodes after activations, weights, and biases. A fake quantization node is a pair of quantization and dequantization nodes (fig. 2.4), simulating the quantization process without actually converting the data to lower precision during training. The operation executed by the fake quantization node is then:

$$fq(x) = s \cdot clip\left(round\left(\frac{1}{s}x - z\right), a_q, b_q\right) + z \tag{2.16}$$

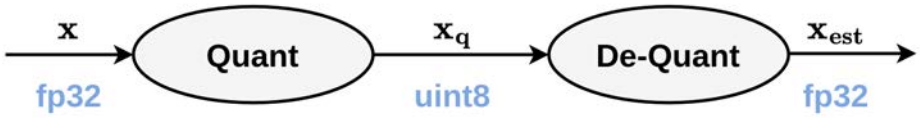


Figure 2.4: Fake Quantization node as Quant and De-Quant node connected in series

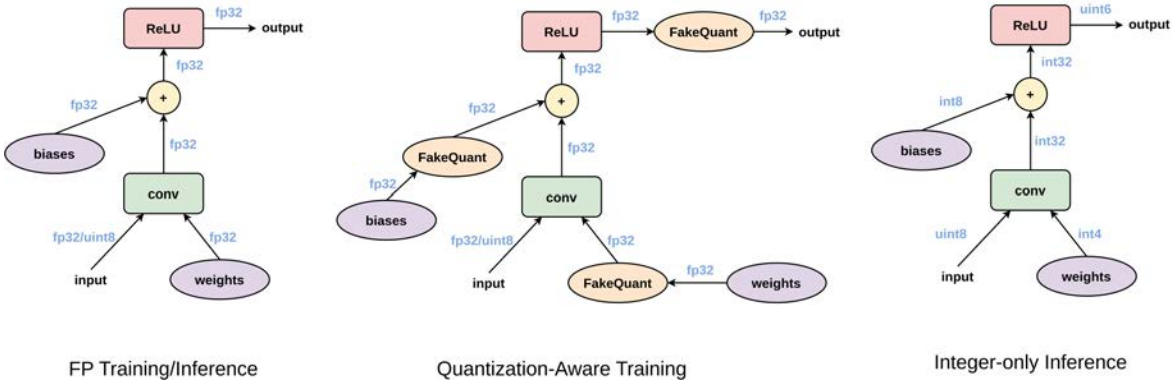


Figure 2.5: Layer-level training/inference for FP and Integer datatypes.

One can easily observe that the fake quantization operation is not differentiable with respect to x. Normally, this would make it impossible to perform the backpropagation algorithm, as it relies on gradients to update parameters. However, we employ the Straight Through Estimation (STE) derivative approximation technique to overcome this issue [22]. In STE, we treat the fake quantization operation as the identity function during backpropagation, where the derivative is set to 1.0 if the clipped number falls within the range $[a_q, b_q]$ and 0.0 otherwise.

During inference, we discard the fake quantization nodes, replacing the floating-point weights and biases with their quantized versions and quantizing the output of activations if needed (fig. 2.5). If we aim for integer-only inference, where all computations are performed using integer arithmetic, it is essential for all weights, biases, and activations to be quantized.

2) **Post-Training Quantization (PTQ)** applies reduced precision to a model after its initial training. Unlike Quantization-Aware Training (QAT), PTQ does not take into account the effects of quantization during the training phase. Instead, it focuses on compressing the pretrained model for deployment on resource-constrained devices. PTQ is generally considered less complex than QAT, as it does not necessitate any modifications to the training process. Calibration is a critical step in the PTQ process. During

calibration, a representative dataset is passed through the network, and statistics are collected and used to estimate scales and zero points correctly. Calibration ensures that the accuracy drop due to PTQ is minimized.

3) **PTQ followed by QAT** combines the advantages of the two approaches. A pretrained model is quantized using PTQ and then finetuned for a small number of epochs using QAT.

2.2.2 Mapping Neural Networks to Hardware

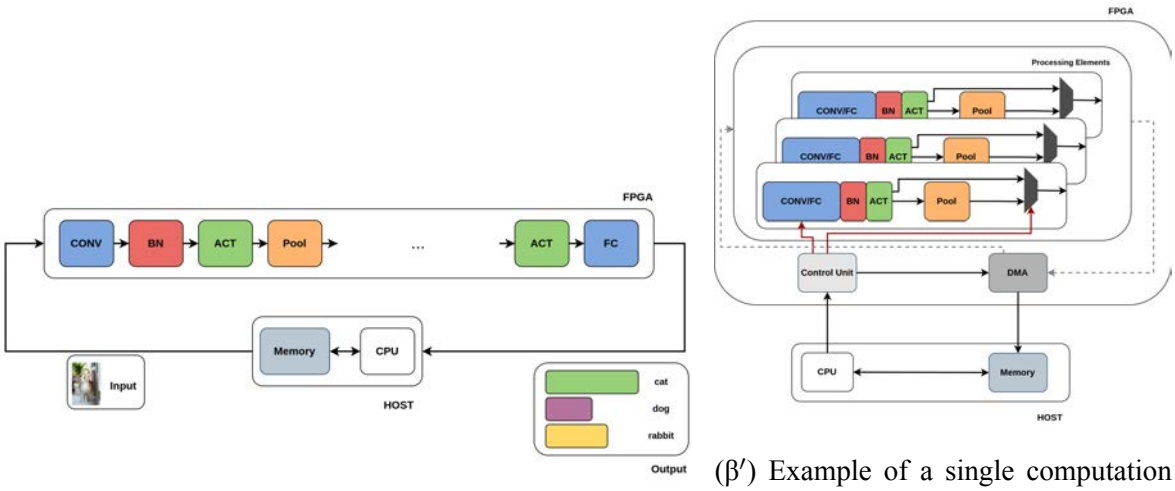
There is a lot of prior work on mapping DNNs on hardware for both FPGAs and ASICs. We can distinguish two major categories: streaming architectures and single computation engines [7].

Streaming Architectures

A streaming architecture usually consists of a dedicated hardware block for each layer, with each block optimized to maximize the parallelism of this particular layer. The blocks are connected in a pipeline fashion, and the data is streamed through the architecture. Pipelining maximizes inter-layer parallelism, enabling simultaneous execution of layers. This efficiency comes with increased design and compilation times since a new bitstream must be generated for each model configuration (see fig. 2.6α′).

Single computation engines

Conversely, at the opposite end of the spectrum, this architectural approach adopts a single computing engine, typically in the configuration of a systolic array of processing elements, which executes the various layers of the target DNN sequentially. This method promotes reusability and portability, as the design can be entirely agnostic to specific applications and is limited solely by the resources of a given device. A compiled bitstream can be utilized across multiple models, and the compiler can translate them into a sequence of instructions executable on the underlying hardware, similar to a CPU (see fig. 2.6β′).



(α′) Example of a streaming accelerator architecture

(β′) Example of a single computation engine accelerator.

Figure 2.6: Architectures of DNN accelerators. Adapted from [7]

2.2.3 FINN

FINN Architecture

FINN is a framework for building FPGA DNN accelerators using a heterogeneous streaming architecture customized for each target DNN. A computation engine is created for each layer, and then those engines are connected in a pipeline. Initially targeting only BNNs where all parameters and activations are quantized to 1-bit, FINN can now target QNNs as well. The basic premise behind FINN is that the original network, which is written in PyTorch, after a series of conversions and transformations, is mapped to parametrizable templates that are subsequently compiled using the Vitis toolchain into hardware accelerators and connected together into a single bitstream.

Optimizations

FINN leverages the inherent DNN computation properties and quantization properties to perform precise optimizations that do not affect accuracy. The two fundamental optimizations performed are:

1) **Convolution as GEneral Matrix Multiplication (GEMM) using the *im2col* approach:** The *im2col* approach [23] is a technique used to transform convolution operations into general matrix multiplication (GEMM) operations, which can be more efficiently computed on hardware optimized for matrix operations.

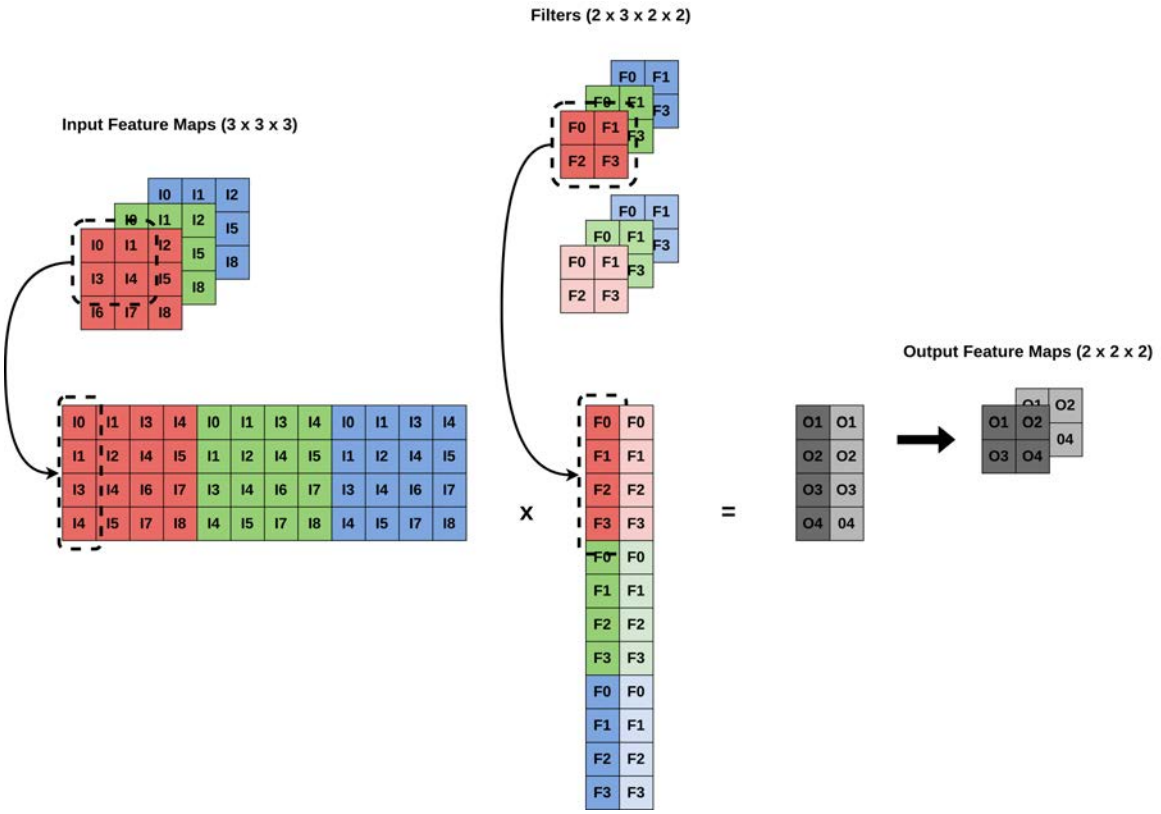


Figure 2.7: An example of *im2col* method for convolution

In CNNs, the *im2col* method reshapes the input image into a column matrix. Each patch of the image that the convolution filter slides over is unrolled into a column vector. These column vectors are then stacked together to form a larger matrix. This transformation enables the convolution operation to be represented as a matrix multiplication between the feature maps and the filter weights, which are also reshaped into a matrix (see fig. 2.7). Note that *im2col* may introduce redundancy by replicating the elements of the input feature maps and the filters (for example, *I4* of the red channel appears four times in the transformed matrix) to speedup 2D convolution.

By converting convolution to GEMM using *im2col*, we can leverage the many optimized libraries and implementations of highly efficient matrix multiplication methods.

2) **Batch Normalization BN as Thresholding:** BN layers typically follow FC and CONV layers and are followed by an activation layer. Through a process called streamlining [24], any QNN layer can be computed using only integer operations. Streamlining is performed in three steps (see fig. 2.9).

   1. **Activation Quantization as successive thresholding**: From figure 2.8, we can

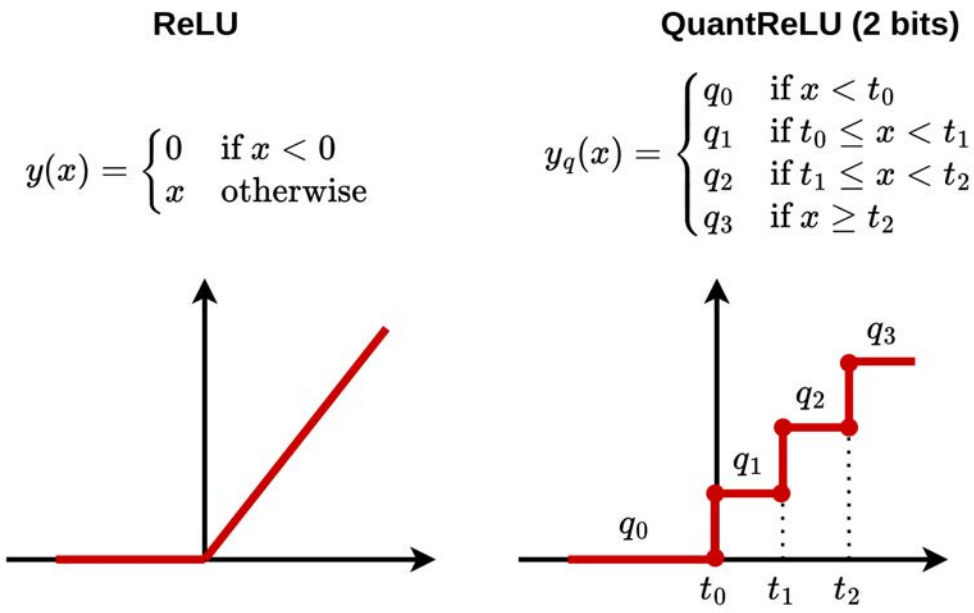


Figure 2.8: Activation Quantization as Thresholding

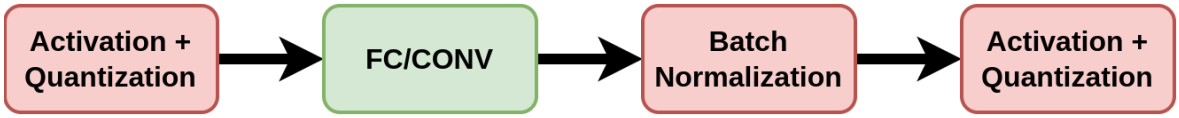
observe that activation plus successive quantization can be written as a linear transformation:

$$Q(x) = \alpha \cdot T(x) + \beta = \alpha \cdot \begin{cases} 0 & \text{if } x \leq t_0 \\ 1 & \text{if } t_0 < x \leq t_1 \\ \quad \cdots & \\ 2^b - 1 & \text{if } 2^{b-1} < x \end{cases} + \beta \tag{2.17}$$

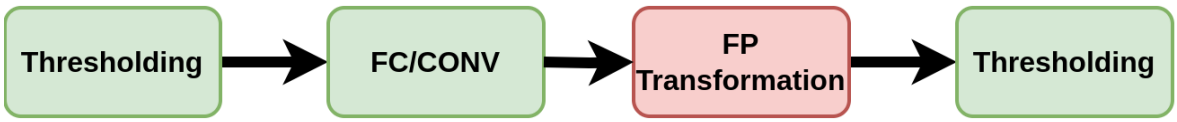
for b-bit quantization, where function t_i are predefined thresholds that are found during training and α and β can be floating point constants.

2. **Collapsing Linear Transformations**: Given that fully-connected and convolution layers perform linear operations, we can move all the floating-point operations to be positioned between those layers, which perform integer operations, and the next activation + quantization layer and collapse them into a single floating point linear transformation: $ax + b$.

3. **Absorbing floating-point linear transformations into thresholds:** To eliminate the remaining floating-point operations, we can absorb them into thresholds. More specifically, we can solve the inequalities $t_i \leq ax + b < t_{i+1}$, so the new thresholds become equal to $t'_i = \frac{t_i - b}{a}$



(α′) Initial network structure



(β′) After collapsing floating point operations in a single node

(γ′) After updating thresholds

Figure 2.9: Steps for streamlining process. Red blocks indicate floating-point operations and green blocks indicate integer operations

Mapping to Hardware

As stated in 2.2.3, the majority of operations in a quantized DNN can be expressed as matrix multiplications followed by thresholding. Hence, the core of every layer block is the Matrix-Vector-Threshold Unit (MVTU), which is shown in 2.11β′. The MVTU consists of an input and an output buffer and an array of P Processing Elements (PEs), each with a number of Q SIMD lanes. The layer parameters (weights, biases, and thresholds) are typically kept in On-Chip Memories. A PE first performs Q parallel multiplications. The results are subsequently reduced in an adder tree and then accumulated with the current dot-product. The accumulation result is then compared with the thresholds from the threshold memory to produce the final output.

A convolution unit is composed of a Sliding Window Unit (SWU) followed by an MVTU (see fig. 2.11α′). The former is tasked with generating the image representation required for a convolution lowered to a matrix multiplication. As shown in 2.11α′, the SWU is a collection of line buffers and logic for writing to and reading from those buffers.

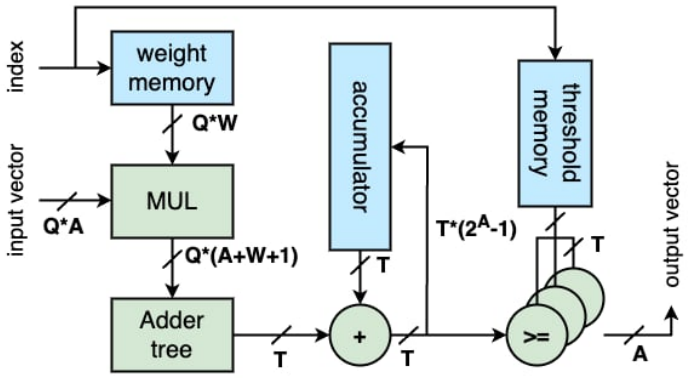
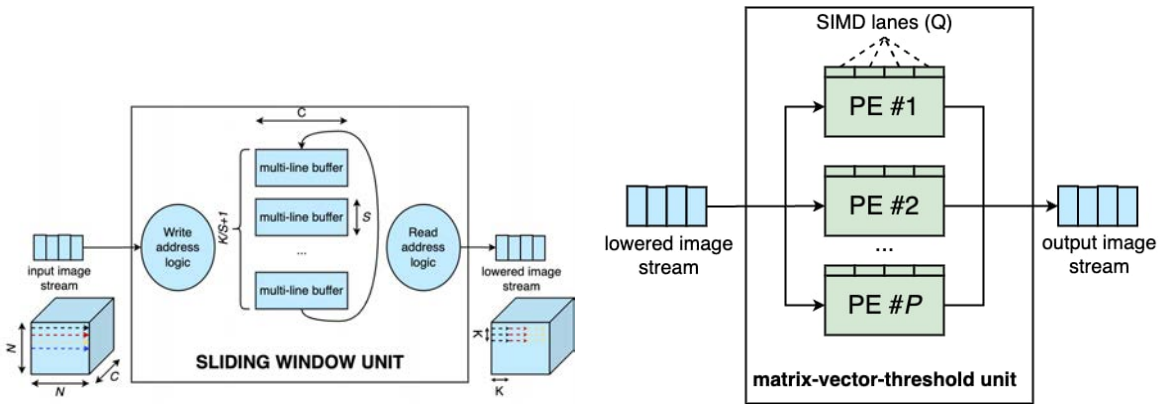


Figure 2.10: Processing Element (PE) Datapath. Bold letters denote the bitwidth of the data at that particular point. **Q** is the number of input vectors, **A** the bitwidth of the activations and **W** the bitwidth of the weights



(α′) Sliding Window Unit (SWU) (β′) Matrix-Vector-Threshold Unit (MVTU)

Figure 2.11: FINN Building blocks of Layers

Folding

FINN aims to achieve a given FPS requirement by controlling the degree of parallelism in each layer. Parallelism is defined by the number of PEs and SIMD lanes per PE in each layer. Each PE corresponds to a hardware neuron, and each SIMD lane acts as a hardware synapse. Had we had enough FPGA resources, we could instantiate layers with PEs and SIMDs equal to the number of neurons and synapses, resulting in a classification rate equal to the clock rate. However, due to resource constraints, we are required to have fewer PEs, which results in time-multiplexed resources and lower throughput. Folding refers to the degree of time-sharing resources. Given that the architecture is pipelined, the latency of the network is defined by the slowest layer, so it makes sense that the other layers are folded in a way that matches the latency of the bottleneck layer. FINN first computes the required latency of a frame in clock cycles and matches the throughput of all layers to the bottleneck layer.

End-to-End Flow

Figure 2.12 shows an example of end-to-end flow in FINN.

1) **Brevitas Export:** Brevitas [25] is a PyTorch library for neural network quantization, with support for both PTQ and QAT. In order to facilitate training using Brevitas, our target DNN model has to be either written using layers from the Brevitas library or rewritten from a PyTorch implementation by replacing the floating-point layers with their quantized implementations.

   After training, the model is exported to QONNX format. QONNX [26], short for Quantized ONNX, is an extension of the ONNX format that focuses on model quantization. ONNX [27] (Open Neural Network Exchange) is an open-source format designed to represent machine learning models that allows for interoperability between different machine learning frameworks. QONNX uses special nodes (Quant, BinaryQuant and Trunc) in order to represent arbitrary-precision quantization in ONNX. From QONNX, the model is transformed to FINN-ONNX format, which is the input expected by FINN.

2) **Network Preparation:** The input in this stage is a network in FINN-ONNX format, which is essentially a graph of nodes with each representing a computation. The main idea of this stage is to perform transformation passes to the network in order to convert

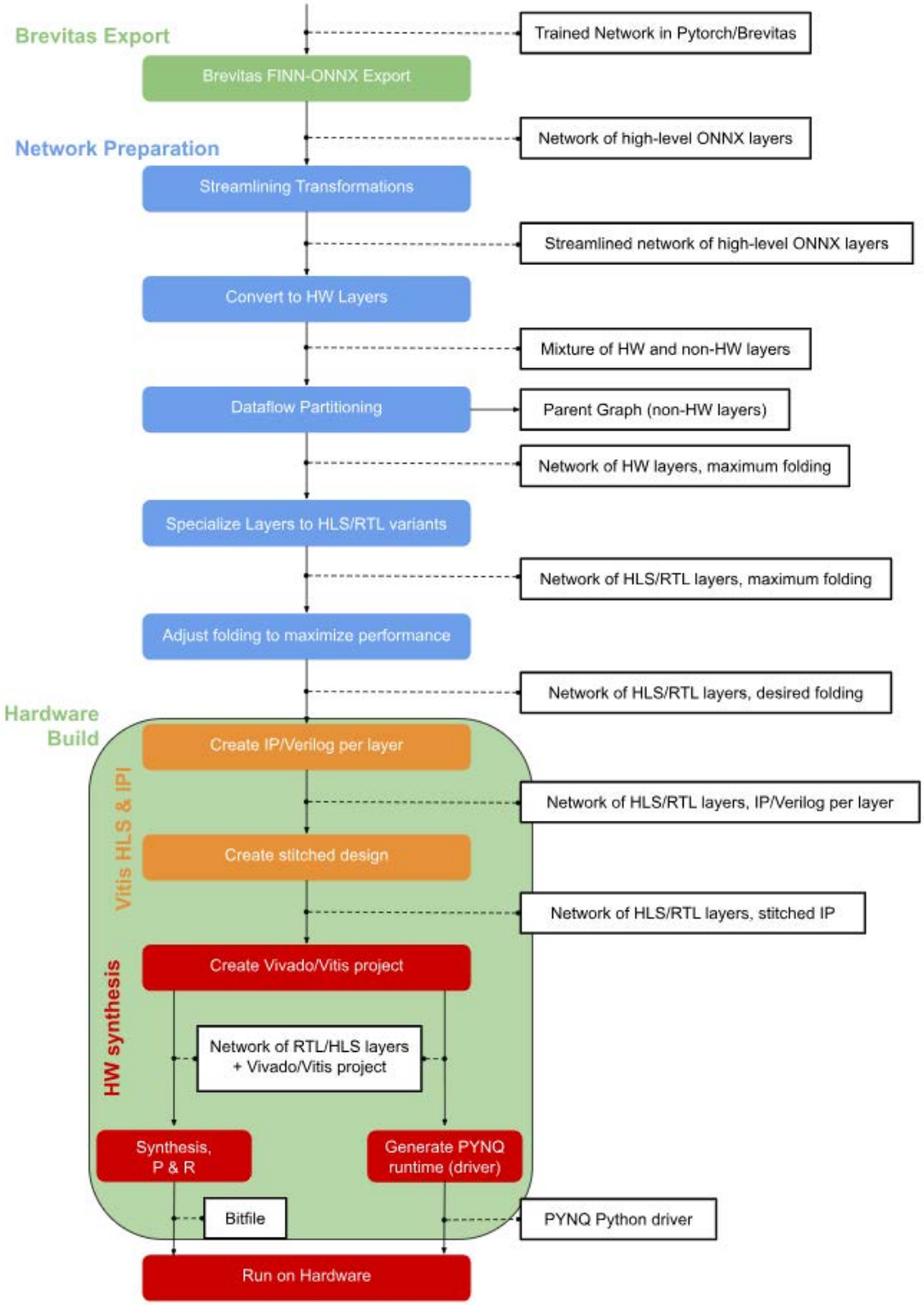


Figure 2.12: End-to-End FINN Flow.

the nodes to custom nodes that correspond to HLS or RTL library functions. Those transformations include:

- **Tidy-up transformations:** Those transformations typically do not alter the structure of the graph but add useful metadata to the nodes, such as their name, their input and output shapes, and their input and output datatypes.
- **Streamlining transformations:** The goal of those transformations is to move nodes around and collapse them into a single operation or absorb them into multithresholding nodes.
- **Convert to HW layers:** After streamlining, the network should only contain nodes that can be transformed into hardware. At this stage, each node is marked with the hardware RTL or HLS template that corresponds to. Note that at this point, the model is still in ONNX format, and no hardware has been built yet.
- **Dataflow Partitioning:** In this step, the graph is split so that only the part consisting of HW layers continues in the FINN flow.
- **Specialize Layers:** At this step, either FINN or the user decides if a layer will be implemented in HLS or RTL.
- **Folding:** This step adjusts the folding of each layer by either using the target FPS or using a custom user folding configuration.

For a more concrete visual example of the transformations, we refer the reader to Appendix B.

3) **Hardware Build and Deployment:** After all layers have been converted to HLS or RTL layers, the model can be processed by FINN, and a bitfile can be generated. Firstly, DMA nodes for transferring data to and from the accelerator, as well as Datawidth converters between layers, will be added to the graph. Next, FIFO nodes will be inserted between nodes, and IP blocks will be generated for each layer. A new project will be created that stitches all those components together, and a bitfile will be created. At the same time, FINN will generate a Python driver for the design, which will be responsible for initiating data movement to and from the accelerator.

2.3 Reinforcement Learning

2.3.1 Preliminaries

RL is a subset of ML where an agent learns by interacting with an environment and observing its feedback. More specifically, at each discrete timestep t, the agent observes the current state of the environment s_t, performs an action a_t according to a policy $\pi(s)$, and subsequently receives the next state s_{t+1} along with a reward r_{t+1}. The set of all possible states the agent can observe at any timestep is called the state space and is denoted as S. Similarly, the set of all possible actions the agent can take in a given environment is denoted as A. The reward given to the agent at each timestep is determined by the reward function R. A higher reward signifies more desirable behavior from the perspective of the agent. The goal of the agent is to optimize the total cumulative reward $R_t = \sum_{i=t}^{T} r_{t+1}$ accumulated over an episode. In addition to the policy, the agent uses two complementary actions to facilitate training, the state-value function $V_\pi(s_t)$ and the action-state value function $Q_\pi(s_t, a_t)$. The former estimates the expected cumulative reward of being in a specific state s_t and following a particular policy π thereafter, and the latter estimates the expected cumulative reward of taking a specific action a_t in a given state s_t and then following a particular policy π thereafter.

2.3.2 Deep Reinforcement Learning DRL

Deep Reinforcement Learning (DRL) is a subset of RL that combines traditional RL techniques with deep learning methods to handle more complex and high-dimensional environments. The primary motivation behind deep RL is to leverage the representational power of deep neural networks to approximate value functions, policies, and models, enabling agents to learn and make decisions from raw sensory inputs like images, sounds, and other high-dimensional data.

DRL extends traditional RL by using deep neural networks to approximate components of RL, making it feasible to apply RL to more complex problems. Traditional RL methods often struggle with high-dimensional state and action spaces due to high dimensionality. DRL addresses this by using neural networks to approximate the state-value function V_π (or state-action value function $Q_\pi(s, a)$) and/or the policy $\pi(a|s)$ itself.

2.3.3 Actor-Critic Algorithms

Although many RL algorithms can be used to train agents, we will focus on actor-critic methods because they were the most suitable for our problem and the ones we worked with.

Actor-critic methods are a type of reinforcement learning algorithm that combines two primary components: an actor and a critic. The actor is the entity that learns the policy π and performs actions according to that policy, and the critic is the entity that evaluates the actions of the actor. After the actor takes action a_t at timestep t, it receives the next state s_{t+1} and a reward r_t. Given the new reward r_t, the critic compares the value function in the new state s_{t+1} with the value function in the current state s_t plus the reward and updates its value function to give more precise estimates. It also provides feedback to the actor in order to improve its policy.

Actor-critic methods are known to lead to more stable and efficient learning and can model continuous action spaces. Examples of actor-critic methods are DDPG [28], TD3 [29], A2C [30], SAC [31] and PPO [32].

2.3.4 Neural Architecture Search

NAS is a method for automating the design of neural network architectures. Instead of manually selecting and tuning hyperparameters, NAS employs search algorithms to explore a vast space of possible architectures and identify the optimal configuration. This approach can significantly enhance the performance of neural networks by discovering architectures that might be overlooked by human designers. NAS methods generally consist of three main components: a search space, a search strategy, and an evaluation method. The search space defines the potential architectural elements and their connections. The search strategy, which can be based on reinforcement learning [33], evolutionary algorithms [34], or gradient-based methods [35], navigates this space to find promising candidates. Finally, the evaluation method assesses the performance of these candidates, typically using a validation dataset. By iterating through these components, NAS can efficiently identify high-performing architectures, optimizing both accuracy and computational efficiency. This automated process is particularly valuable in designing CNNs for hardware deployment, as it can balance the trade-offs between performance and resource constraints, leading to more efficient and effective network implementations.

2.3.5 HAQ: Hardware-Aware Automated Quantization with Mixed Precision

The authors of HAQ propose a learning-based RL framework to determine the bitwidth of both weights and activations for each layer on different hardware accelerators [18]. The novelty of their work lies in the fact that by using RL, they can target larger and deeper networks that rule-based strategies could not efficiently handle. They also adhere to constraints by involving the target hardware architecture in the loop.

They employ the Deep Deterministic Policy Gradient (DDPG) actor-critic algorithm to train an agent that decides the bitwidths of weights and activations in a layer-wise manner. Each network layer constitutes a different state that is described by a ten-dimensional feature vector S_k. If the k^{th} layer is a convolution layer, the state is equal to:

$$S_k = (k, c_{in}, c_{out}, s_{kernel}, s_{stride}, s_{feat}, n_{params}, i_{dw}, i_{w/a}, a_{k1}) \tag{2.18}$$

where k is the layer index, c_{in} is the number of input channels, c_{out} is the number of output channels, s_{kernel} is the kernel size, s_{stride} is the stride, s_{feat} is the input feature map size, n_{params} is the number of layer parameters, i_{dw} is a binary indicator for depthwise convolution, $i_{w/a}$ is a binary indicator for weight/activation, and a_{k1} is the action from the last time step.

If the k^{th} layer is a fully-connected layer, the state S_k is

$$S_k = (k, h_{in}, h_{out}, 1, 0, s_{feat}, n_{params}, 0, i_{w/a}, a_{k1}) \tag{2.19}$$

where k is the layer index, h_{in} is the number of input hidden units, h_{out} is the number of output hidden units, s_{feat} is the size of the input feature vector, n_{params} is the number of layer parameters, $i_{w/a}$ is a binary indicator for weight/activation, and a_{k1} is the action from the last step.

At each time step, the action taken is a number in the range $[0, 1]$ that is subsequently scaled and rounded to an integer in the range $[1, 8]$. The reason that a continuous action space is chosen as opposed to a discrete one is that the relative order of the bitwidths should have to be preserved; a 2-bit quantization, for instance, is more aggressive than an 8-bit quantization.

The reward function R is only related to accuracy and is equal to:

$$R = 0.1 \cdot (acc - acc_{orig}) \tag{2.20}$$

where acc_{orig} is the accuracy of the original model.

The authors enforce latency and energy constraints by limiting the action space. After all the layers have been quantized at the end of an episode, they use direct feedback (latency and energy) from the hardware accelerator, and, in case of a constraint violation, they progressively reduce the bitwidths of layers until the constraint is fulfilled.

Chapter 3

Tool flow for Architectural Exploration

3.1 High-Level Overview

Fig. 3.1 provides an overview of the end-to-end flow we have developed. The first step involves training the target DNN, assuming it has not been pretrained. We then train a RL agent on the model. The objective of the RL agent is to determine a quantization scheme that maximizes accuracy while accounting for both resource and latency constraints. Next, the model is quantized using the trained RL agent or, alternatively, a user-supplied quantization strategy. The agent also supplies the folding configuration that yields the target latency given the available platform resources. The model and its parameters are then exported to FINN, where necessary transformations are applied. The optimized design is implemented into hardware, and the FPGA device is programmed. We deploy our accelerator using the provided Python driver on the target platform.

3.2 Target CNN Training

Our flow accepts target CNNs written in PyTorch. We currently support training models for image classification using the MNIST [36] or CIFAR-10 [37] datasets from Torchvision [38]. The model may contain fully-connected, convolutional, pooling, and activation layers. If the model contains batch normalization layers, those should exist between a fully-connected or convolutional layer and an activation layer so that the streamlining can successfully take place. Attention should be paid to networks with residual connections, such as ResNet architectures [21], because FINN requires the same sign across residual adds, so it

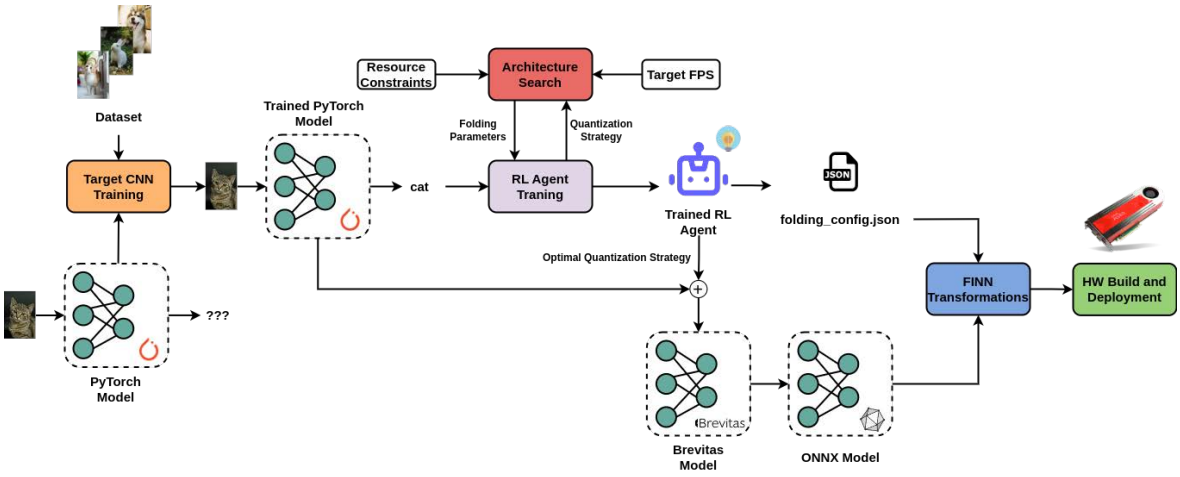


Figure 3.1: High-level overview of our approach

is possible that some ReLU activations may need to be added or removed [39].

3.2.1 RL Agent Training

Modelling our environment was mainly inspired by HAQ, but some changes had to be made to ensure that our model was compatible with FINN after exporting. We had to establish a proper methodology for quantizing a pretrained model correctly so that it fulfills the below requirements imposed by FINN:

1. FINN is not designed for hardware accelerators that contain FP operations since it requires integer-only inference. Therefore, the weights, biases, and inputs to all layers should be quantized to an integer representation.

2. The inputs to element-wise operators and concatenations should be quantized to have the same scale, sign, and bit-width.

3. The inputs to convolution layers should be quantized to a bit-width that is larger than 2 bits since it is mandatory that the number 0 should be represented exactly by the input datatype as the padding inserted by the convolution consists of 0s itself.

Given the above requirements, algorithm 1 enlists the order in which the network layers should be quantized for FINN compatibility.

As mentioned in 1, the bitwidth of the input and output is fixed at 8 bits. The bit widths of the quantization nodes inserted in intermediate steps (steps 3-4) are also predetermined by the user, with a recommended value of 4 bits. Those decisions constitute a minority of

Algorithm 1 FINN-Compatible Quantization Steps

1: **Input Quantization** The network input is quantized to *INT8* using a `QuantIdentity` node. Since `QuantIdentity` nodes only support signed integers, *UINT8* is not an option. Typically, input images have integer values in the range $[0, 255]$, which are scaled and normalized to floating point values in the range $[0, 1]$ during training. Although quantizing the input to *INT8* would suffice, the input only contains positive values, so we would be wasting half the *INT8* range. To address this, we insert multiplication and addition nodes at the beginning of the network to scale the input to the range $[-1, 1]$.

2: **Activation Quantization** To minimize the amount of re-quantizations and consecutive quantizations needed, we first quantize the output of every activation, so that the inputs to compute layers, such as Linear and Conv, are already quantized. For ReLU activations, we opt for unsigned quantization since we know that the output will consist of positive values only.

3: **Add Output Quantization** Similar to activation quantization, we quantize the outputs of addition operations if they are not immediately followed by a quantized activation function.

4: **Residual Handler** We ensure that the inputs to Add and Concat nodes are quantized to have the same scale, zero point and bit-widths by inserting quantization nodes where necessary.

5: **Layer Quantization** At this stage, the input to each computation layer is already quantized. The remaining task is to quantize the weights and biases of these layers, as well as their outputs if they are not followed by a quantized activation function. We quantize the biases to *INT8*. A special case occurs when a layer is followed by Batch Normalization (BN). Since BN is merged with the layer during BN folding, we ensure the output of the BN layer is quantized. If the output of the layer is also the output of the model, we quantize it to *INT8*.

the quantization process and are fixed in advance. The most critical quantization decisions, which significantly impact the results, involve determining the bitwidths of the activations and weights in the computation layers. Considering the above, we had to model the environment in a way that allows us to first predict the best bitwidth for the activation layers and afterwards for the compute layers and not concurrently as in the case of HAQ.

3.2.2 State Space

In HAQ, the quantizable layers include the Linear and Convolution layers, where two actions are taken at each step: one for the weights and one for the activations. In our approach, we traverse the network layer by layer, first identifying the activations that need to be quantized. We then perform quantization steps 3-4 of 1 and proceed to quantize the weights of the

compute layers. This requires a different representation. We decided on a six-dimensional feature vector. For the k^{th} quantizable node (activation or compute layer) k, the feature vector is:

$$S_k = (k, i_{a/c}, t_k, \#flops, \#params, a_{prev}) \tag{3.1}$$

where:

- k is the node index
- $i_{a/c}$ is a binary indicator for activation or computation node
- t_k is the type of activation or compute node
- $\#flops$ is the number of floating-point operations executed by this node
- $\#params$ is the number of parameters maintained by this node
- a_{prev} is the last action taken by our agent

The aforementioned state representation can be easily extended to include other types of nodes, such as attention or recurrent layers, beyond just Linear and Convolution layers. Note that each value of the state space vector is normalized to $[0, 1]$.

3.2.3 Action Space

At the k^{th} timestep, the agent decides on an action a_k, which is in the range of $[-1, 1]$ and rounds it into the dicrete bitwidth value b_k as:

$$b_k = \lceil (a_k + 1) \cdot (b_{max,k} - b_{min,k})/2 + b_{min,k} - 0.5 \rceil \tag{3.2}$$

where $b_{max,k}$ and $b_{min,k}$ is the maximum and minimum allowed bitwidths for node k. For activation layers, the minimum bitwidth is set to 2 to ensure zero can be represented in subsequent convolution layers. However, the minimum bitwidth for weights is kept at 1.

3.2.4 Constraints

According to [40], imposing large penalties to ensure the agent adheres to constraints may not be optimal. Following the HAQ approach, we aim to achieve the user-defined target FPS by restricting the action space. After quantizing all layers and defining the folding parameters,

we iteratively decrease the maximum bitwidth, prioritizing the layers towards the end of the network, which are typically larger due to having more neurons. We then re-quantize the model, recalculate the folding parameters, and reassess the target FPS. This process is repeated until the target is met (see section 3.3).

3.2.5 Reward Function

Our reward function focuses solely on the accuracy achieved by the model. At the conclusion of an episode, once all layers have been quantized and the target FPS has been achieved, we calibrate the model using a small subset of the training data. We then finetune the model for a specified number of epochs and evaluate its accuracy on the testing dataset. Let this accuracy be denoted as acc, ranging from 0.0 to 100.0. The reward R is calculated as follows:

$$R = acc \cdot 0.02 - 1.0 \tag{3.3}$$

This formulation ensures that R falls within the range $[-1, 1]$. During the episode, rewards assigned at each quantization step prior to the end of the episode are set to zero.

3.2.6 RL Implementation

In our approach, the implementation of RL involves parsing the target CNN into a custom Gymnasium [41] environment. We employ the TD3 algorithm, utilizing the stable-baselines framework [42] for training the RL agent.

3.3 Architecture Search

To determine the folding parameters for each layer, the quantized model in Brevitas format is first exported to QONNX and then to FINN-ONNX. A series of network preparation transformations are applied to convert the model into hardware (HW) nodes, similar to the process described in Appendix B. Each layer is mapped to a templated implementation, enabling accurate resource and latency estimates based on the methodology in [43].

FINN uses the `SetFolding` transformation to establish folding parameters by calculating the maximum allowable clock cycles based on the target frames per second (FPS). This process focuses on the bottleneck layer, which has the highest estimated latency. While effec-

tive for smaller target FPS values, this approach relies heavily on user-defined FPS requirements and does not thoroughly explore the design space, particularly neglecting optimization knobs like RAM type and memory mode.

Our approach draws inspiration from [43], using the critical path to determine overall latency. Initially, we configure the model with a single processing element (PE) and SIMD lane for each layer, estimating clock cycles for each. We then iteratively increase parallelism in the bottleneck layer until resources are exhausted or the target FPS is met. We systematically explore various configuration knobs per layer to optimize resource utilization. These configuration knobs include:

- FPGA Resource Type for Memories (`ram_type`)
- Memory Mode for MVAU (Matrix-Vector Activation Unit) Weights (`mem_mode`)
- Resource Type for MVAU Layers (`res_type`)

In addition to resource constraints, we also adhere to constraints related to allowed values for PE and SIMD for each layer, as described in [44]. To avoid issues during hardware generation, we ensure that the instream and outstream of each layer are within limits (lower than 8192 bits) and that any HLS layer is not set to use DSP resources, as this is currently suboptimal.

Before declaring a design infeasible, we identify and mitigate limiting resources by modifying the hardware attributes of nodes with the highest utilization of the respective resource. For instance, if BRAMs are the limiting resource, we alter the `ram_type` attribute of the node consuming the most BRAMs, switching it to URAMs or LUTRAMs. This process is repeated until the design becomes feasible or a maximum number of iterations is reached. We do not use the exact number of available resources on the target platform but instead apply a safety factor recommended by the vendor to ensure timing closure. While most FINN transformations are universally applicable, streamlining and conversion to hardware transformations are highly network-specific. There is no ”one-size-fits-all” approach. We have implemented these transformations for the models used in our experiments, but they may not apply to other models. Therefore, we provide instructions for adding custom models and their necessary transformations within our flow in the Github repository of our project.

Algorithm 2 SetFolding Transformation

```
1:  Input: model, platform, targetFrequency, targetFPS
2:  for all nodes in model.nodes do
3:      nodes.PE ← 1
4:      nodes.SIMD ← 1
5:  end for
6:  while isFeasible(model) do
7:      fps ← estimateTargetFPS(model, targetFrequency)
8:      if fps ≥ targetFPS then
9:          return model
10:     end if
11:     oldModel ← model
12:     bottleneck ← findBottleneck(model)
13:     bottleneck.PE ← next valid value
14:     bottleneck.SIMD → next valid value
15: end while
16: return oldModel
```

3.4 FINN Transformations and Hardware Build

After the agent quantizes the model using the optimal strategy and provides the quantized model along with the folding parameters that meet the resource and latency requirements, we use the standard FINN flow to build and deploy the accelerator, requiring no further user input.

Chapter 4

Experiments

In this section, all experiments are conducted on the Alveo U250 FPGA platform [45]. Alveo U250 is a high performance accelerator card developed by Xilinx, designed to boost the performance of data center applications. It features a large amount of programmable logic, high-bandwidth memory, and a versatile I/O architecture, making it ideal for workloads such as machine learning inference, data analytics, video processing, and database acceleration. The available resources of U250 are specified in Table 4.1.

4.1 LeNet5 for MNIST

LeNet5 [46], an early CNN architecture developed specifically for handwritten digit recognition, is composed of 2 convolutional layers followed by 3 fully connected layers (see Fig. 4.1). We used LeNet5 to classify the MNIST [36] dataset, achieving an initial precision of 98. 77% for the unquantized model after 10 training epochs.

Resource Type	Total Number
BRAM_18K	2384
LUTs	1488120
URAMs	1280
DSPs	10634

Table 4.1: Available Resources for Alveo U250

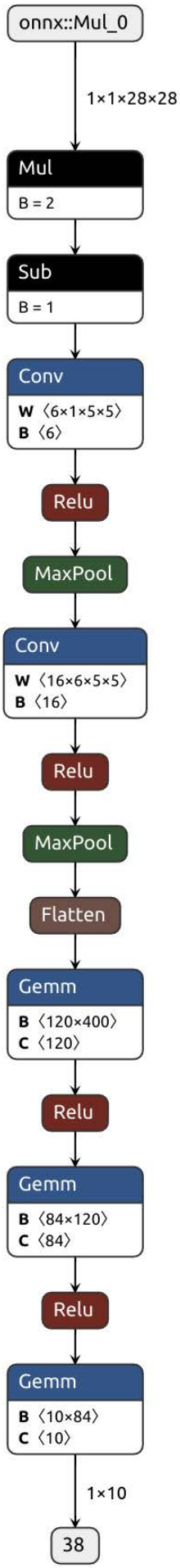


Figure 4.1: ONNX model for the LeNet5 CNN.

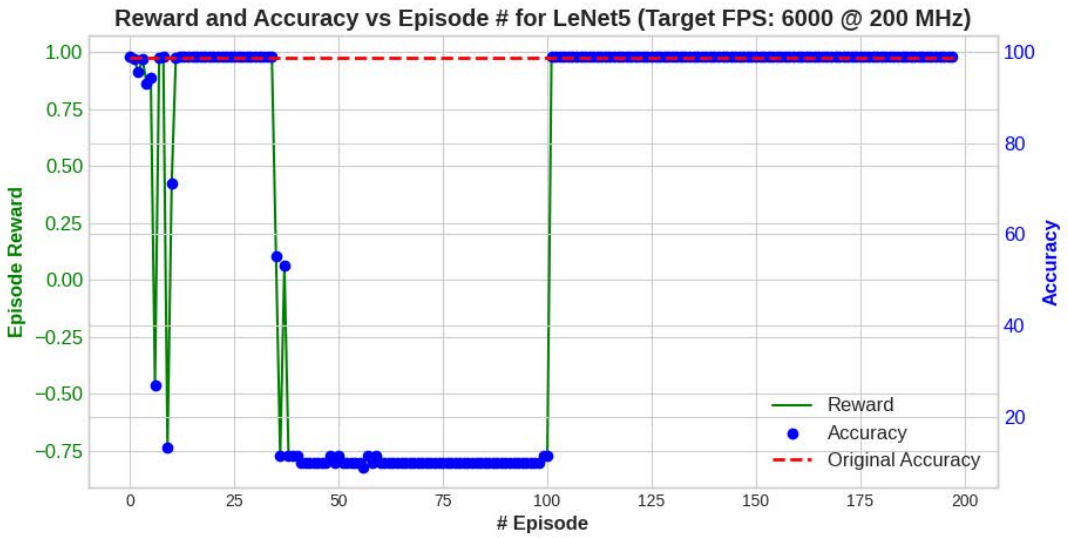


Figure 4.2: Reward and Accuracy per training episode for LeNet5 with target FPS = 6000 @ 200MHz

Quantization Method	w1a2	w4a4	w8a8	Custom	Unquantized
Accuracy (%)	9.74	99.09	99.15	**99.21**	98.77

Table 4.2: Accuracy for various quantization methods (LeNet5)

4.1.1 Agent Results

Figure 4.2 illustrates the training reward and network accuracy during the training of an agent targeting 6000 FPS at 200 MHz. The agent converged successfully to a quantization strategy that exceeded the accuracy of the original unquantified model. Table 4.2 presents the accuracies achieved through various quantization methods; “wXaY” denotes X bits for weights and Y bits for activations, while “Custom” signifies the optimized quantization strategy determined by our agent. Our quantization strategy outperforms even the unquantized floating-point model. Figure 4.3 also provides resource estimates for each strategy; our custom policy achieves higher accuracy with fewer resources than uniform 8-bit quantization, highlighting the efficiency gains of optimized bitwidth allocation.

4.1.2 Hardware Build and Deployment

The design was successfully synthesized, achieving the target frequency of 200 MHz. Figure 4.4 shows the measured throughput for the hardware accelerator. Although the achieved

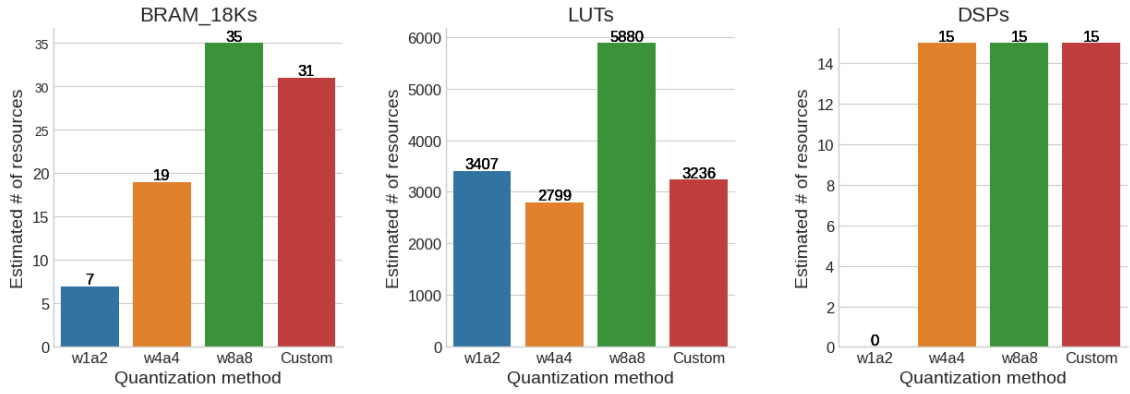


Figure 4.3: Estimated resources for various quantization methods

throughput slightly varies across batch sizes, the design consistently meets the target FPS.

4.2 ResNet18 for CIFAR10

We opted to implement the ResNet18 residual network, the smallest variant among ResNets. ResNet18 consists of a series of convolutions arranged in residual blocks. Figure 4.5 illustrates a residual block used in our network. This component is repeated four times within the network. Being larger and more complex than LeNet5, ResNet18 is well suited to classify challenging datasets like CIFAR10, which comprises color images categorized into distinct classes. ResNet18 achieves a classification accuracy of 85.25% in the CIFAR10 dataset after training for 100 epochs.

4.2.1 Agent Results

Since we were aware that this network would be more challenging, we chose a reasonable target FPS of 1000 and at a frequency of 100MHz. Due to the considerably larger design space, its larger size makes it hard for an agent to converge to an optimum strategy in a short number of epochs. However, we could leverage the exploration of the agent to construct the Pareto front of accuracy vs. network size using the experiences of the agent, as shown in Figure 4.6. In order to preserve as much accuracy as possible, we opted to implement the Pareto point with the highest accuracy at 83.31%. The accuracy of other quantization methods is listed in Table 4.3. It is worth noting that for w4a4 and w8a8 models the accuracy might be

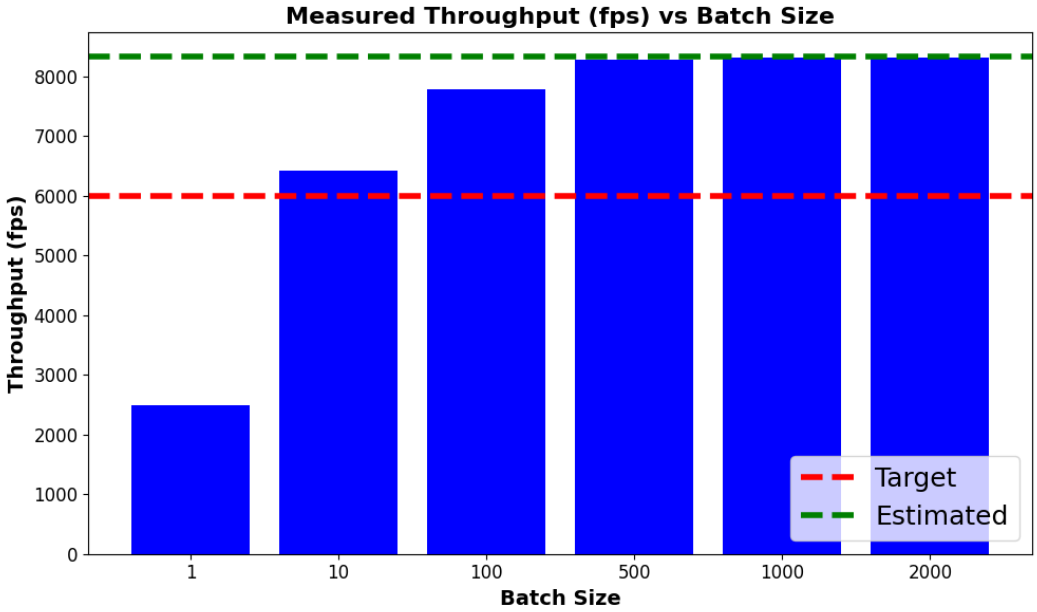


Figure 4.4: Measured FPS per Batch Size for LeNet5

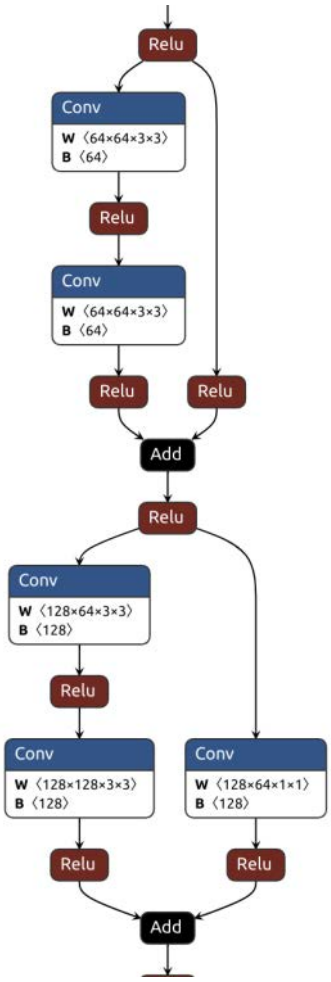


Figure 4.5: Residual Blocks in ResNet

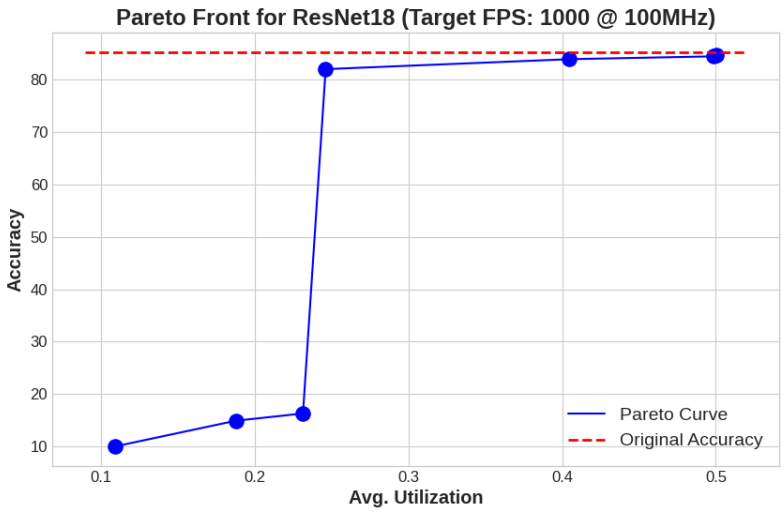


Figure 4.6: Pareto Front for ResNet18 (Target FPS: 1000 @ 100MHz)

Quantization Method	**w1a2**	**w4a4**	**w8a8**	**Custom**	**Unquantized**
Accuracy (%)	10.0	88.94	**89.95**	83.31	85.25

Table 4.3: Accuracy for various quantization methods (ResNet18)

higher than the unquantized model, but those quantization schemes are infeasible and cannot achieve 1000fps @ 100 fps with the uniform 8-bit quantization exceeding available resources even for the slowest model where all folding parameters are equal to 1.

4.2.2 Hardware Build and Deployment

Despite the low target frequency, the accelerator managed to operate at only 26MHz in practice and did not reach the target FPS of 1000. Its performance is scaled accordingly at a maximum of around 250FPS, as can be seen in Fig. 4.7.

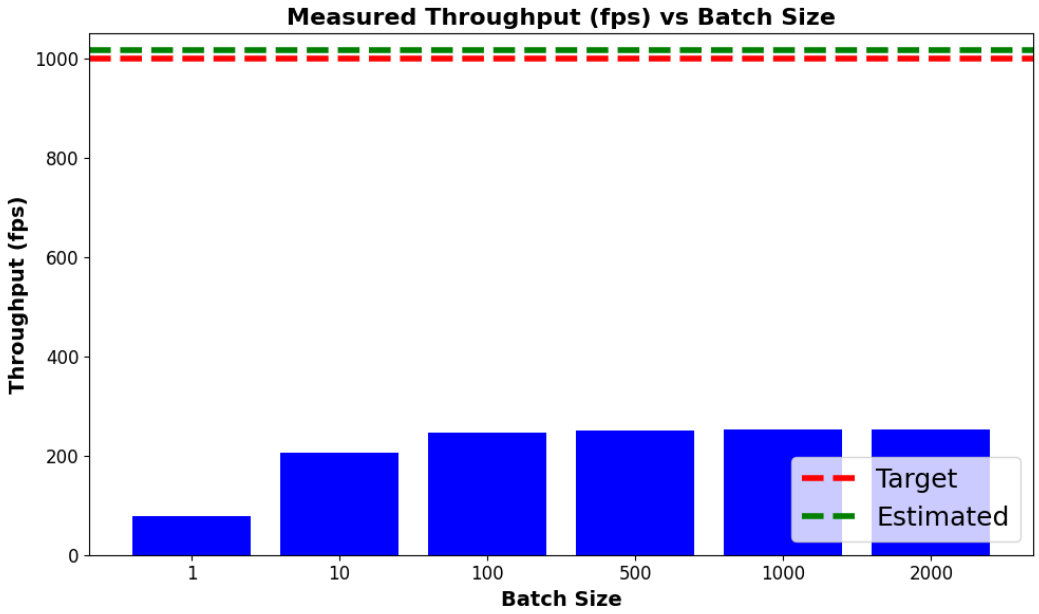


Figure 4.7: Measured FPS per Batch Size for ResNet18

Chapter 5

Conclusion

5.1 Summary

In this thesis, we developed a comprehensive end-to-end CNN-to-FPGA workflow that enhances existing frameworks by automating components that traditionally required significant user input. Our findings demonstrate that employing learning-based methods and mixed-precision quantization, although more time-intensive, yields superior accuracy and enhanced performance. Additionally, we highlighted the challenges of mapping very deep neural networks onto hardware. We emphasized the need for improved collaboration between the front-end design exploration tool and the back-end hardware implementation tool to achieve more accurate performance estimates.

5.2 Future Work

Our work can and is encouraged to be extended in multiple directions. To indicate a few:

1. Support for other DNN architectures, such as RNNs, LSTMs, Transformers by adding more options for layers in the state representation of the model. However, this requires support from the back-end tool (FINN), which, to our knowledge, is still not available.

2. Reinforcement Learning Agent Optimizations and hyperparameter tuning. Experiments using a various reinforcement learning algorithm or different state/action/reward formulations can be conducted to find out which combination on the above works best across models or for specific network architectures.

3. Support for more models and datasets. Although our flow will work for all models that satisfy the architecture constraints described in section 3.2, they have not yet been integrated into our flow.

Bibliography

[1] Salman Khan, Hossein Rahmani, Syed Afaq Ali Shah, Mohammed Bennamoun, Gerard Medioni, and Sven Dickinson. A guide to convolutional neural networks for computer vision. 2018.

[2] Yoav Goldberg. *Neural network methods for natural language processing*. Springer Nature, 2022.

[3] Ahmad Shawahna, Sadiq M Sait, and Aiman El-Maleh. Fpga-based accelerators of deep learning networks for learning and classification: A review. *ieee Access*, 7:7823–7859, 2018.

[4] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[5] Ruizhou Ding, Zeye Liu, RD Shawn Blanton, and Diana Marculescu. Quantized deep neural networks for energy efficient hardware-based inference. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 1–8. IEEE, 2018.

[6] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays*, pages 65–74, 2017.

[7] Stylianos I Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[9] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[10] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[11] Yoon Kim. Convolutional neural networks for sentence classification proceedings of the 2014 conference on empirical methods in natural language processing, emnlp 2014, october 25-29, 2014, doha, qatar, a meeting of sigdat, a special interest group of the acl. *Association for Computational Linguistics, Doha, Qatar*, 2014.

[12] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 974–983, 2018.

[13] Yu-Huei Cheng, Pang-Ching Chang, and Che-Nan Kuo. Convolutional neural networks approach for music genre classification. In *2020 International Symposium on Computer, Consumer and Control (IS3C)*, pages 399–403. IEEE, 2020.

[14] Sparsh Mittal. A survey on modeling and improving reliability of dnn algorithms and accelerators. *Journal of Systems Architecture*, 104:101689, 2020.

[15] Sparsh Mittal, Poonam Rajput, and Sreenivas Subramoney. A survey of deep learning on cpus: opportunities and co-optimizations. *IEEE Transactions on Neural Networks and Learning Systems*, 33(10):5095–5115, 2021.

[16] EunJin Jeong, Jangryul Kim, Samnieng Tan, Jaeseong Lee, and Soonhoi Ha. Deep learning inference parallelization on heterogeneous processors with tensorrt. *IEEE Embedded Systems Letters*, 14(1):15–18, 2021.

[17] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12. IEEE, 2017.

[18] Kai Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8612–8620, 2019.

[19] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr, 2015.

[20] Wonkyung Jung, Daejin Jung, Byeongho Kim, Sunjung Lee, Wonjong Rhee, and Jung Ho Ahn. Restructuring batch normalization to accelerate cnn training. *Proceedings of Machine Learning and Systems*, 1:14–26, 2019.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. corr abs/1512.03385 (2015), 2015.

[22] Penghang Yin, Jiancheng Lyu, Shuai Zhang, Stanley Osher, Yingyong Qi, and Jack Xin. Understanding straight-through estimator in training activation quantized neural nets. *arXiv preprint arXiv:1903.05662*, 2019.

[23] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *Tenth international workshop on frontiers in handwriting recognition*. Suvisoft, 2006.

[24] Yaman Umuroglu and Magnus Jahre. Streamlined deployment for quantized neural networks. *arXiv preprint arXiv:1709.04060*, 2017.

[25] Alessandro Pappalardo. Xilinx/brevitas, 2023.

[26] Alessandro Pappalardo, Yaman Umuroglu, Michaela Blott, Jovan Mitrevski, Ben Hawks, Nhan Tran, Vladimir Loncar, Sioni Summers, Hendrik Borras, Jules Muhizi, et al. Qonnx: Representing arbitrary-precision quantized neural networks. *arXiv preprint arXiv:2206.07527*, 2022.

[27] ONNX. Onnx github repository. `https://github.com/onnx/onnx`, n.d. Accessed: July 3, 2024.

[28] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[29] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

[30] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[31] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[32] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[33] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[34] Yuqiao Liu, Yanan Sun, Bing Xue, Mengjie Zhang, Gary G Yen, and Kay Chen Tan. A survey on evolutionary neural architecture search. *IEEE transactions on neural networks and learning systems*, 34(2):550–570, 2021.

[35] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *Knowledge-based systems*, 212:106622, 2021.

[36] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. Mnist handwritten digit database. `http://yann.lecun.com/exdb/mnist/`, 2010.

[37] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. `https://www.cs.toronto.edu/~kriz/cifar.html`, 2009.

[38] TorchVision Contributors. Torchvision: Datasets, transforms and models specific to computer vision. https://pytorch.org/vision/stable/index.html, 2024. Accessed: 2024-06-24.

[39] Brevitas. Brevitas resnet model. https://github.com/Xilinx/brevitas/blob/200456825f3b4b8db414f2b25b64311f82d3991a/src/brevitas_examples/bnn_pynq/models/resnet.py#L112, 2024. Accessed: 2024-06-24.

[40] Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. Constrained policy optimization. In *International conference on machine learning*, pages 22–31. PMLR, 2017.

[41] Farama Foundation. Gymnasium: A framework for reinforcement learning environments. https://github.com/Farama-Foundation/Gymnasium, 2024.

[42] DLR Institute of Robotics and Mechatronics. Stable Baselines3: Reinforcement learning in pytorch. https://github.com/DLR-RM/stable-baselines3, 2024.

[43] Michaela Blott, Thomas B Preußer, Nicholas J Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees 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)*, 11(3):1–23, 2018.

[44] Inc. Xilinx. *FINN Documentation: Constraints to Folding Factors per Layer*, 2024. https://finn.readthedocs.io/en/latest/internals.html#constraints-to-folding-factors-per-layer.

[45] Xilinx. Alveo u250 data center accelerator card. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html, n.d. Accessed: July 3, 2024.

[46] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

APPENDICES

Appendix A

Quantization

A.1 Uniform vs Non-Uniform Quantization

Quantization can be uniform or non-uniform. In the uniform case, the mapping from the input to the output is a linear function resulting in uniformly spaced outputs for uniformly spaced inputs. In the non-uniform case, the mapping from the input to the output is a non-linear function so the outputs won't be uniformly spaced for a uniform input (see fig. A.1). Uniform quantization is simpler and more efficient for hardware implementation but may sacrifice accuracy, especially for datasets with non-uniform distributions. Non-uniform quantization can offer better accuracy but requires more complex mapping functions and computational resources.

In this thesis, we mainly focus on uniform quantization, since it is more hardware-friendly.

A.2 Asymmetric and Symmetric Quantization

In asymmetric quantization mode, we map the min/max in the float range to the min/max of the integer range.

Conversely, in symmetric quantization mode, instead of mapping the exact min/max of the float range to the quantized range, we choose the maximum absolute value between min/max. This choice effectively centers the quantized integer range around zero, making the quantization symmetric with respect to zero, hence we do not need a zero-point for this type of quantization (see fig. A.2). For symmetric quantization, we can opt for a restricted/narrow-range that excludes the smallest negative value for hardware purposes.

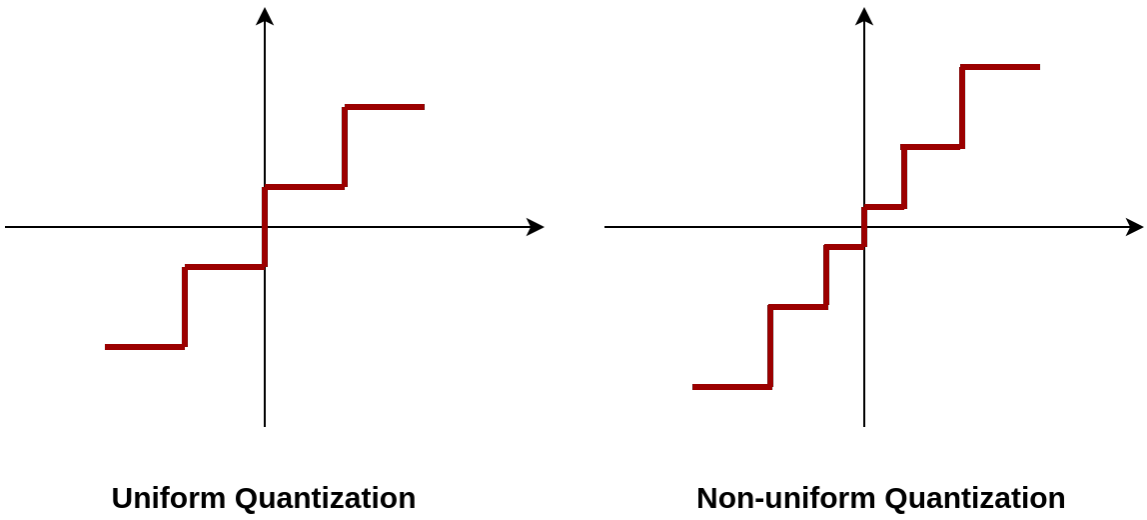


Figure A.1: Uniform vs Non-Uniform Quantization

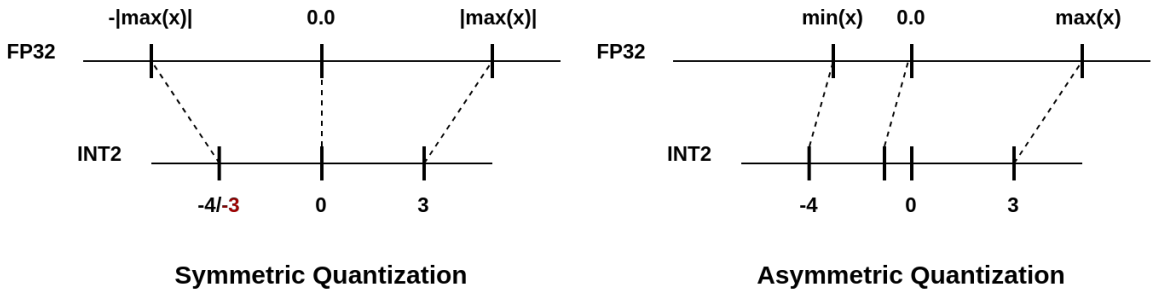


Figure A.2: Symmetric vs Asymmetric Quantization

In asymmetric quantization, we fully utilize the quantized range by directly mapping the minimum and maximum values from the floating-point range to the corresponding endpoints of the quantized range. This ensures that the entire dynamic range of the data is represented optimally.

However, in symmetric mode, if the floating-point values are biased towards one side, it can lead to inefficient utilization of the quantized range. For instance, after applying Rectified Linear Unit (ReLU), where all values are positive, symmetric quantization dedicates significant range to negative values that will never occur. Quantizing it in symmetric mode means we're effectively losing 1 bit.

Conversely, looking at the implementation complexities, symmetric quantization is simpler for convolutional and fully connected layers. In asymmetric mode, handling zero-points requires additional hardware logic, which can impact performance in terms of latency, power consumption, or area, depending on the specific implementation details.

A.3 Per-tensor and Per-channel Quantization

Another aspect of quantization to consider is the granularity of the quantization parameters. Quantization parameters can be computed either on a per-tensor basis or on a per-channel basis.

When quantization parameters are computed on a per-tensor basis, a single pair of (scale, zero-point) is used for the entire tensor. In contrast, with per-channel quantization, a pair of (scale, zero-point) is computed for each element along one of the tensor's dimensions. For instance, in a tensor with dimensions [N, C, H, W], using per-channel quantization parameters for the second dimension would mean computing and storing C pairs of (scale, zero-point).

A.4 Weights and Activations Quantization

Quantization ranges are treated differently for weight quantization vs. activation quantization:

- For weights, we choose the floating-point range to be quantized to be the same as the full weight range after training.
- For activations, we choose the floating-point activation range by statistics aggregated during training.

Appendix B

FINN Transformations

In this appendix, we will showcase examples of few of the FINN transformations on a simple CNN model. Figure B.1 shows the starting unquantized CNN.

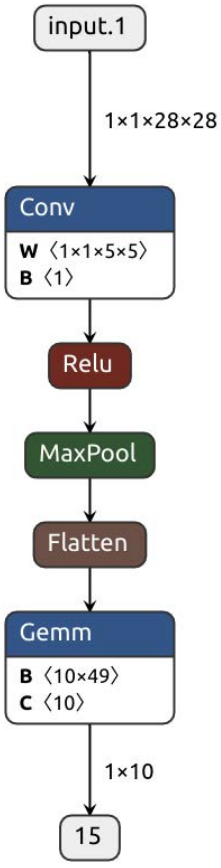


Figure B.1: Unquantized simple CNN

Figure B.2 depicts the model after exporting it to **QONNX** format.

Figure B.3 depicts the model after the **tidy-up** transformations. Notice that the input and output of the model now have readable names that correspond to their actual function and

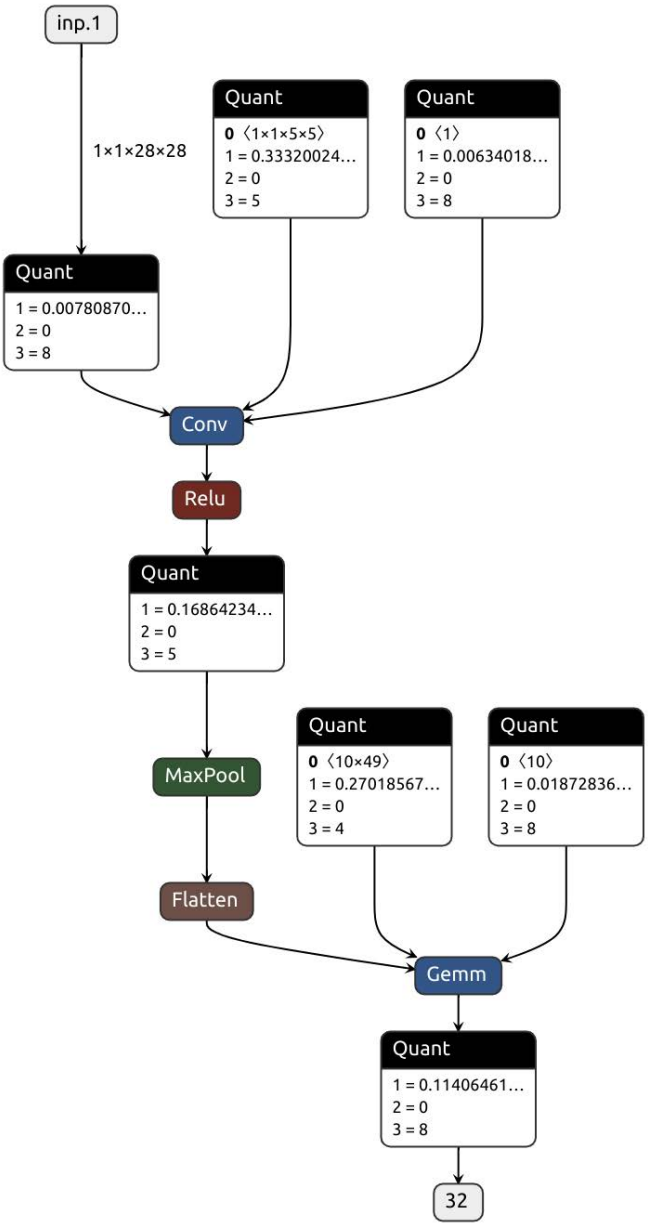


Figure B.2: CNN in QONNX format

that the output shapes of each node have been inferred.

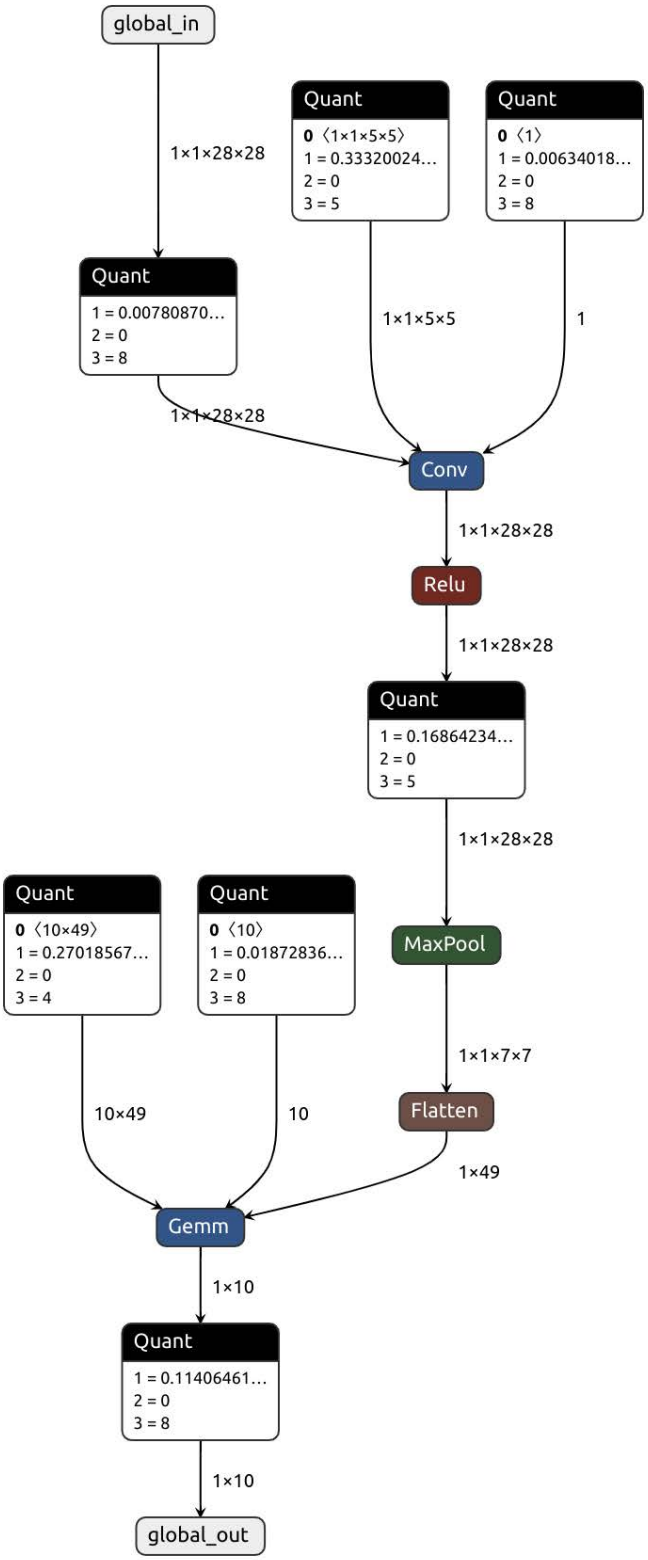


Figure B.3: CNN after tidy-up transformations

Figure B.4 depicts the model after the **pre-** and **post-** processing transformations. The

input dataset is in a channels-first format, but FINN uses a channels-last format, so a transpose node should be added at the beginning of the model to shape the input appropriately. In addition, the input that the model was trained on was normalized as a preprocessing step offline to a range (0, 1), but during inference the input will be in UINT8, so we need to divide it manually to the range (0, 1). As a postprocessing step, we insert a TopK node at the end of the network that extracts the index which corresponds to the top predicted class from the output of the last fully-connected layer.

Figure B.5 shows the model after it has been converted from QONNX to **FINN-ONNX** representation. Notice how the activation `Quant` nodes have been transformed to `MultiThreshold` nodes, the quantized weights have been absorbed inside the convolution and linear layers and the biases have been transformed to `Add` nodes.

Figures B.6, B.7, B.8 and B.9 show the model at various stages of the streamlining process. In figure B.6, `Div` and `Sub` nodes are converted to `Mul` and `Add` nodes. In figure B.7, the `Add` nodes after the `MultiThreshold` nodes are absorbed in the thresholds of the latter. In figure B.8, the nodes representing linear operations, such as `Mul` nodes are absorbed in the preceding nodes, such as `MatMul` if they are linear as well. Finally, in B.9, scalar operations are absorbed in the `TopK` node.

Figures B.10, B.11, B.12, B.13, B.14, B.15, B.16 depict the model in various stage of the conversion to hardware process. This process involves replacing nodes with HLS or RTL templates and performing streamlining steps in between when necessary. In figure B.10, we replace the `MaxPool` node with an `Im2Col` node and a `Pool` node plus `Transpose` nodes. In figure B.11, we replace the `Convolution` layers with `ConvolutionInputGenerator` nodes and `MatMul` nodes. In figure B.12, we replace the `MatMul` nodes with `MVAU` (Matrix-Vector Activation Unitss) and `VVAU` (Vector-Vector Activation Units). In figure B.13, we reorder the `Transpose` nodes in front of the `MultiThreshold` nodes, so that consecutive tranposes can cancel out each other. In figure B.14, we replace the `MultiThreshold` nodes with their corresponding `Thresholding` hardware implementation. In figure B.15, we infer a `LabelSelect` layer in place of the `TopK` node. Finally, in figure B.16, we remove the consecutive `Transpose` and `Flatten` layer by moving the reshaping operation in the succeeding `MVAU` node. The model now consists entirely of hardware nodes, so we can continue with the rest of the FINN transformations.

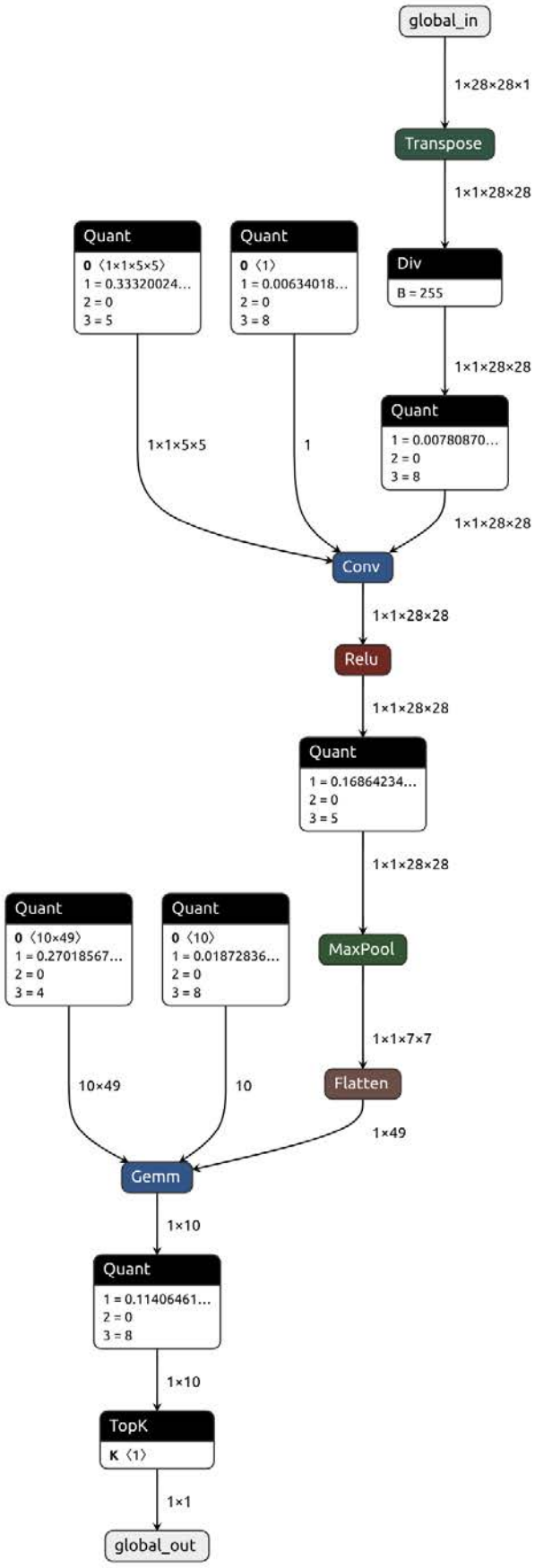


Figure B.4: CNN after pre- and post-processing transformations

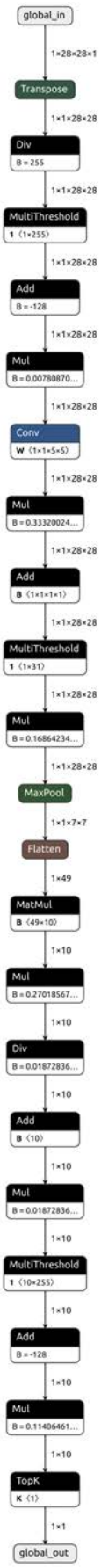


Figure B.5: CNN in FINN-ONNX format

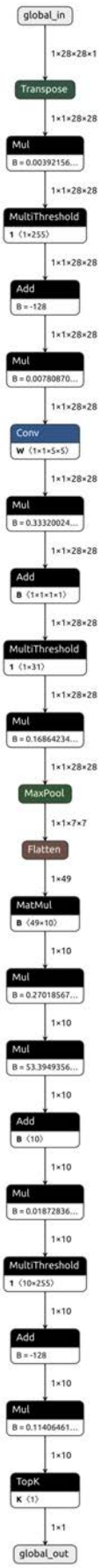


Figure B.6: CNN model after streamlining step 1

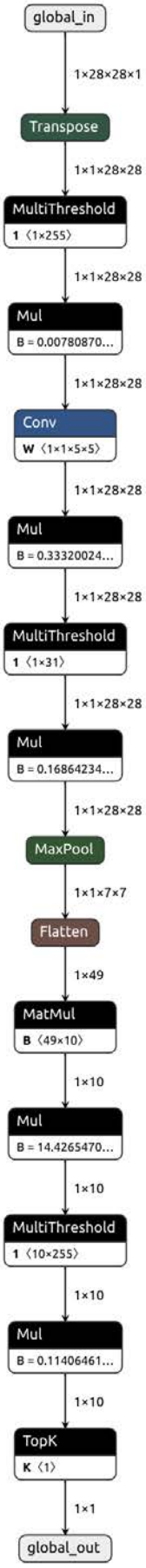


Figure B.7: CNN model after streamlining step 2

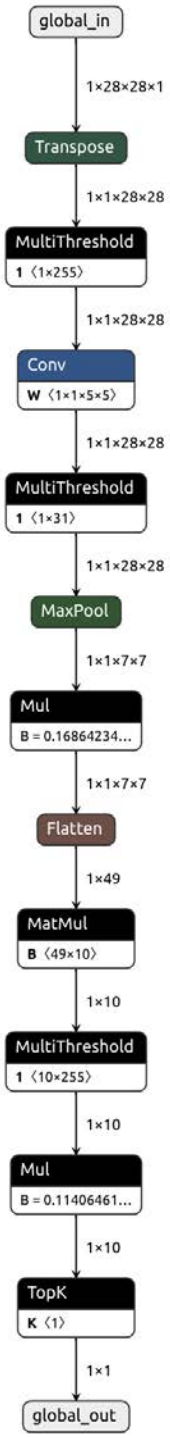


Figure B.8: CNN model after streamlining step 3

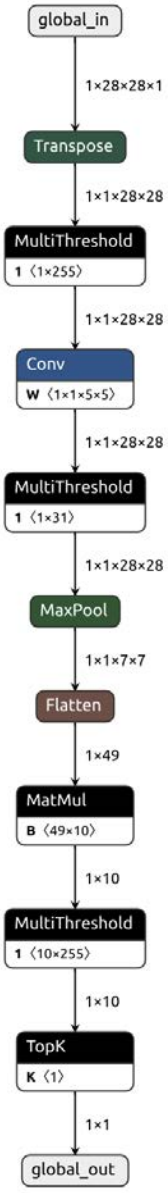


Figure B.9: CNN model after streamlining step 4

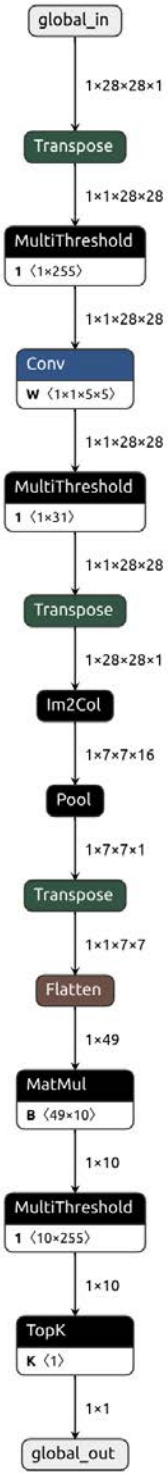


Figure B.10: CNN model after converting to hardware step 1

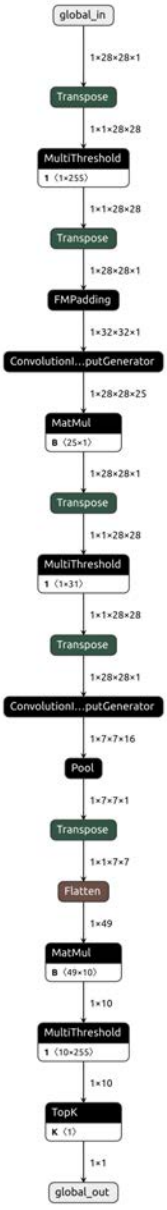


Figure B.11: CNN model after converting to hardware step 2

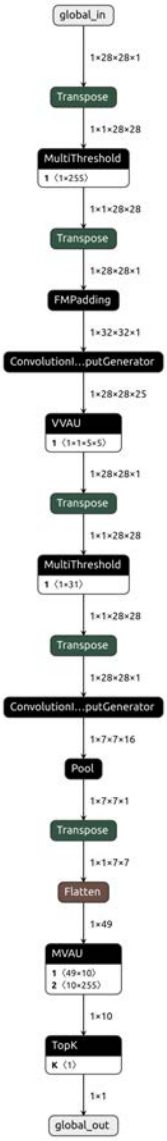


Figure B.12: CNN model after converting to hardware step 3

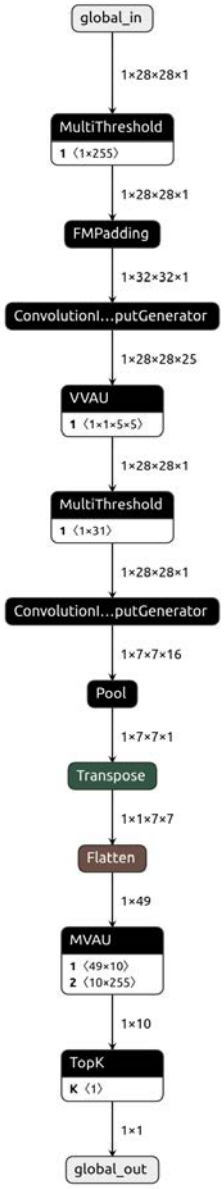


Figure B.13: CNN model after converting to hardware step 4

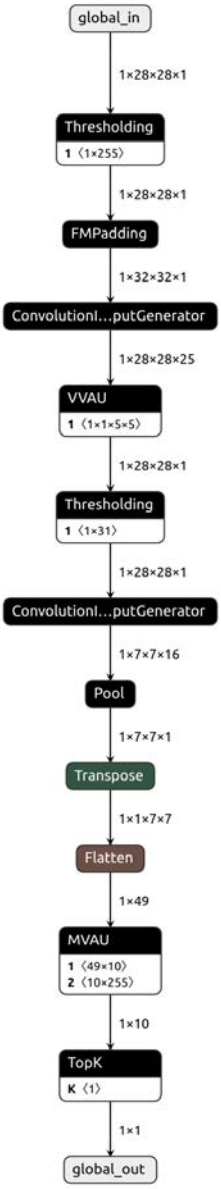


Figure B.14: CNN model after converting to hardware step 5

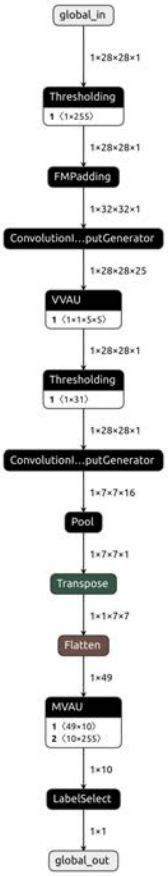


Figure B.15: CNN model after converting to hardware step 6

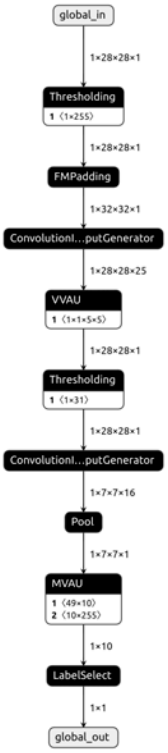


Figure B.16: CNN model after converting to hardware step 7