POLITECNICO DI TORINO

Master's Degree Course in Electronic Engineering

Politecnico
di Torino

Master's Degree Thesis

Hardware Acceleration of AdderNet via High-Level Synthesis for FPGA

Candidate

Valentina Marino

Supervisor

Prof. Luciano Lavagno

October 2024

Technology is the campfire around which we tell our stories

Alla roccia che sempre sostiene la mia anima
Alla forza che mi scaglia oltre ogni orizzonte
All'amore che mi illuminerà per tutta la vita
All'orgoglio di chi avrebbe voluto essere qui

Abstract

Convolutional Neural Networks (CNNs) are widely used for machine learning tasks but often come with high computational costs due to their reliance on resource-intensive Multiply-Accumulate (MAC) operations. As a more efficient alternative, AdderNet (AddNN) replaces these MAC operations with simpler Sum-of-Absolute-Difference (SAD) operations, employing an ℓ_1-norm-based approach. While this architecture reduces computational expenses, it has not yet achieved the same level of hardware optimization as CNNs, particularly in areas such as effective quantization, accelerator design, and efficient use of FPGA resources like DSP slices.

This thesis presents an efficient quantized implementation of the AddNN ResNet20 architecture using an 8-bit fixed-point quantization scheme. Developed with the Brevitas framework, this approach significantly reduces memory usage and computational overhead, enabling efficient deployment on FPGAs. The model's hyperparameters were fine-tuned and trained on the CIFAR-10 dataset, achieving an accuracy of 86.61%. The architecture was represented in a custom QONNX format to detail the layers, input-output quantization, and connections.

Additionally, a tailored high-level synthesis (HLS) code generation pipeline was created to transform the Python-based model into an FPGA bitstream using Vitis HLS. The implementation and validation of this architecture were demonstrated on the Kria KV260 FPGA board, utilizing ResNet20 on the CIFAR-10 dataset to showcase the efficiency of the proposed solution.

The contributions of this research aim to enhance the performance and flexibility of AddNNs in embedded systems, particularly those leveraging FPGA hardware, through innovations in both architectural design and implementation methodologies.

Contents

List of Figures viii

List of Tables x

Nomenclature xi

1 Introduction 1

1.1 Machine Learning and Deep Learning 1

1.2 Artificial Neural Networks 3

1.3 Neural Networks on Edge Devices 4

1.3.1 The Evolution of Neural Networks 5

1.3.2 Applications of Neural Networks 5

1.3.3 Challenges and Limitations 5

1.4 Convolutional Neural Network 6

1.5 Convolutional Layers 7

1.5.1 Padding and Stride 8

1.6 Normalization Layers 9

1.7 Activation Layers 12

1.8 Pooling Layers 13

1.9 Fully Connected Layers 14

2 AdderNet **15**

2.1 Introduction 15

2.2 Adder Layers: Theory and Implementation 16

2.3 Gradient Computation in AdderNets 16

2.4 Adaptive Learning Rate in AdderNets 16

2.5 Skip Connections in AdderNet 17

2.6 Comparisons with CNNs 18

2.7 Hardware Implementation 19

2.8 Applications and Future Work 21

3 Quantization-Aware Training using Brevitas **22**

3.1 Quantization in Brevitas 23

3.1.1 Quantization Classes 29

3.1.2 Quantizers 29

3.1.3 Quantization Function 29

3.1.4 Model Optimization 29

3.1.5 Deployment Support 30

3.2 AddNN ResNet20 Quantization using Brevitas 30

3.2.1 AddNN QuantResNet20 Code 30

3.2.2 Quantization of adder2d in Brevitas 35

3.2.3 QuantAdd2d Code 35

3.2.4 Training process and results 39

3.2.5 Quantization of BatchNorm2d in Brevitas 41

4 ONNX **44**

4.1 Model Exportation in ONNX 44

4.2 ONNX Model Visualization 45

4.3 QONNX 47

5 Integer Quantized AdderNet Implementation 52

5.1 NN2FPGA Framework 52

5.2 High-Level-Synthesis 52

5.3 Vitis HLS 53

5.3.1 Workflow 54

5.4 Framework and Reproduced CNN ResNet20 55

5.5 AdderNet Accelerator 55

5.6 DSP Packing 61

5.6.1 Adopted Solution 62

5.6.2 Future Work 65

6 Evaluation and Conclusions 66

6.1 Simulation and Analysis 66

6.2 Simulation Environment 66

6.3 Experimental Results 67

6.4 Conclusions 68

References 69

Appendix A ResNet 70

Appendix B adder2d 74

Appendix C Dataset 77

C.0.1 Features Structure 78

List of Figures

1.1 Types of Learning 2

1.2 ML, DL, AI visualization 3

1.3 Neuron: details 4

1.4 Example of Fully Connected Layers 6

1.5 Example of a 2D Convolution 8

1.6 Padding explained visually 9

1.7 Strides example with $s_w = 2$ and $s_h = 3$ 10

1.8 Batch Norm explained visually 11

1.9 ReLU and ReLU6 activations 12

1.10 Example of a pooling layer applied to a 4x4 tensor. The result depends on the chosen pooling method. 13

1.11 Fully Connected Artificial Neural Network 14

2.1 Basic Blocks: comparison between CNN and AddNN 18

2.2 ResNet20 Architecture 18

2.3 CNN Kernel 19

2.4 AdderNet Kernel 19

2.5 CNN Kernel compared to AddNN Kernel 19

2.6 Architecture of the universal AdderNet accelerator 20

3.1 Training result: Accuracy 40

3.2 Training result: Loss 40

4.1 Standard Node for Conv2d 46
4.2 Custom Node for adder2d 47
4.3 Detail of the first convolutional layer $Conv_0$ 48
4.4 Detail of the first *adder2d* layer 49
4.5 Detail of the second *adder2d* layer 50
4.6 Detail of the first *Add* operation node 51

5.1 Vitis HLS Diagram 53
5.2 Vitis HLS - Project Composition 54
5.3 Vitis HLS - Workflow 55
5.4 FPGA's DSP slice. 62
5.5 DSP packing for(a) Dual-INT8 CNN,and DSP packing and DSP-LUT co-packing for(b) Quad-INT8 and Octo-INT8 AdderNet designs 65

List of Tables

2.1 Comparison between CNN and AddNN on a ResNet20 with CIFAR-10 and CIFAR-100 18

3.1 Mapping of main PyTorch layers to available Brevitas layers 28

6.1 Post-Synthesis Performance for AdderNet ResNet-20 67

6.2 Post-Synthesis Resource Usage for AdderNet ResNet-20 68

Nomenclature

AddNN Adder Neural Network

AI Artificial Intelligence

ANN Artificial Neural Network

ASIC Application-Specific Integrated Circuits

BIP Binary Integer Programming

BN Batch Normalization

BRAM Block RAM

CNN Convolutional Neural Network

CPU Central Processing Units

DL Deep Learning

DNN Deep Neural Network

DRAM Dynamic Random-Access Memory

DSP Digital Signal Processors

FF Flip-Flop

FM Feature Map

FPGA Field Programmable Gate Array

GPU Graphic Processing Unit

HLS High-Level Synthesis

LUT Look-Up Table

MAC Multiply And Accumulate

ML Machine Learning

MLP Multi-Layer Perceptron

NLP Natural Language Processing

ONNX Open Neural Network Exchange

PTQ Post-Training Quantization

QAT Quantization-Aware Training

QONNX Quantized Open Neural Network Exchange

ReLU REctified Linear Unit

RTL Register-Transfer Level

SAD Sum of Absolute Differences

SIMD Single Instruction Multiple Data

URAM Ultra RAM

Chapter 1

Introduction

1.1 Machine Learning and Deep Learning

Artificial Intelligence (AI) is an innovative computer science field, enabling computers and digital devices to learn and perform tasks that optimize their chances of achieving specific goals. The remarkable progress in AI over recent years has significantly impacted various applications, including image recognition, object detection, language understanding, and problem-solving.

Machine Learning (ML) can be considered a sub-field of AI. It adopts algorithms trained on datasets to create self-learning models that are capable of predicting outcomes and classifying information without human intervention. ML methods are generally categorized into three primary types:

- **Supervised Learning:** This approach involves training a machine to predict an output value, either continuous (regression) or discrete (classification), corresponding to an input. Algorithms are trained on labeled datasets that include tags describing each piece of data. This means that the training phase is based on example pairs of inputs and outputs to instruct the machine on the correct output for each input.
- **Unsupervised Learning:** In this method, algorithms are trained on unlabeled datasets with no tags. This means that the machine has to identify on its own patterns and similarities within the data to categorize it into groups, without predicting a specific output value.

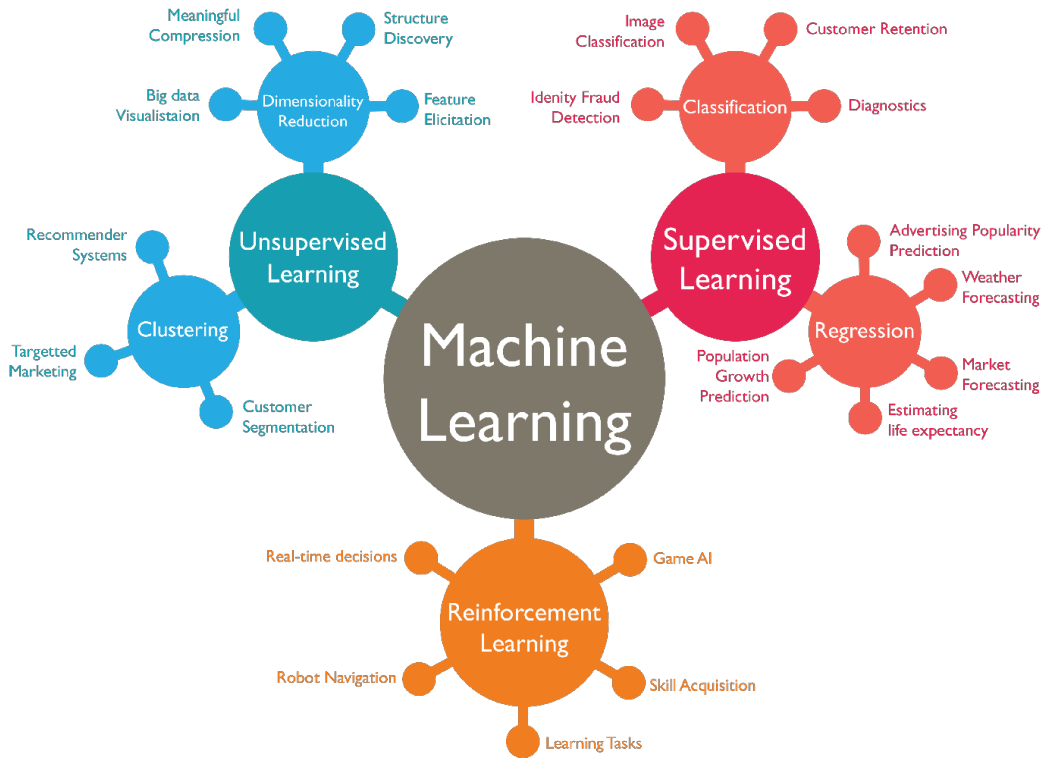


Fig. 1.1 Types of Learning

- **Reinforcement Learning:** This technique addresses sequential decision problems where the future state of the systems is determined by its current state and the action taken. The objective is to develop autonomous agents that select actions to achieve specific goals, with a "reward function" measuring the quality of the chosen action in order to reinforce the desirable behavior.

Deep Learning (DL) refers to a type of machine learning that employs multi-layered structures. What distinguishes DL is its ability to autonomously recognize high-level features from the input data, unlike traditional machine learning algorithms that rely on features designed by humans.

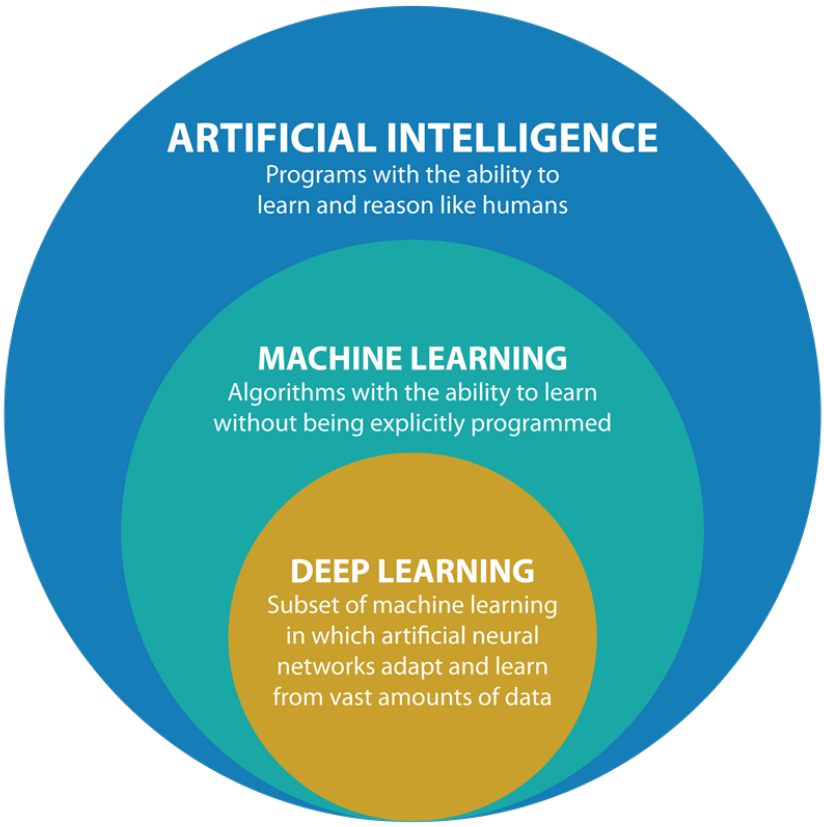


Fig. 1.2 ML, DL, AI visualization

1.2 Artificial Neural Networks

Artificial Neural Networks (ANNs) are mathematical models inspired by the structure of human brain neurons. In feedforward neural networks, each neuron performs an algebraic operation on its input x_i, represented by the equation:

$$y_i = w_i \cdot x_i + b_i$$

where w_i and b_i are the weight and bias of the i-th neuron, respectively. The output y_i is thus a linear function of the input x_i.

ANNs consist of neurons organized in layers, with each layer's neurons interconnected in a sequence-like structure.
Considering a neuron with multiple inputs, the equation can be generalized as:

$$y_i = \sum_{j=1}^{n} w_{ij} \cdot x_j + b_i$$

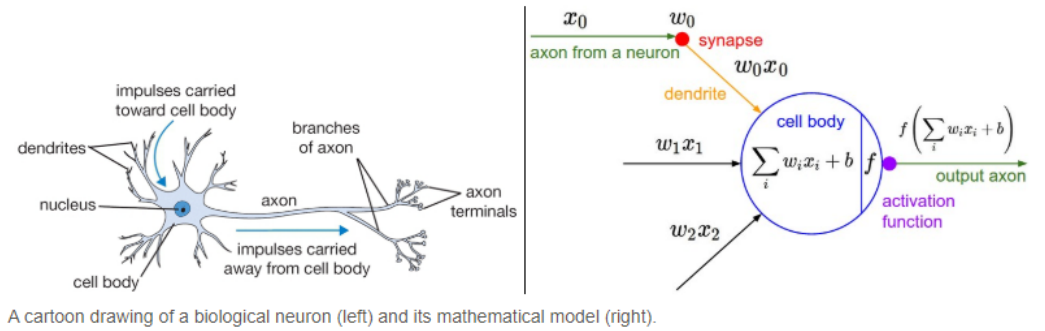


Fig. 1.3 Neuron: details

1.3 Neural Networks on Edge Devices

The advent of powerful Graphics Processing Units (GPUs) has made possible to deploy increasingly complex neural networks, reducing training and inference times in a significant way. However, the high power consumption of GPUs makes them impractical for edge devices like mobile phones and cameras, which have limited memory and computational capacity.

Edge computing is crucial for applications requiring real-time data processing, privacy, and low latency. Consequently, research has focused on optimizing neural networks for edge devices through techniques such as:

- **Pruning:** This technique reduces the size of neural networks by removing redundant neurons or connections, thereby decreasing memory usage and speeding up computations.
- **Quantization:** This method optimizes data and weight representation, balancing memory usage and accuracy.
- **Model Architecture Design:** Developing alternative neural network structures to create more efficient models.

1.3.1 The Evolution of Neural Networks

The evolution of neural networks has reached several important milestones. Early models, like the perceptron, struggled with non-linear problems. The creation of multi-layer perceptrons (MLPs) brought in hidden layers, allowing these networks to understand non-linear relationships. Still, MLPs faced challenges with computational efficiency and training deep architectures.

The early 2000s saw a major breakthrough with the introduction of deep learning, as deep neural networks (DNNs) began to deliver exceptional performance in a range of tasks. Improvements in hardware, particularly with GPUs, along with techniques like dropout and batch normalization, have significantly boosted the training and generalization abilities of these networks.

1.3.2 Applications of Neural Networks

Neural networks have revolutionized numerous fields. For instance, they play a crucial role in image processing, for tasks such as classification, object detection, and segmentation. In natural language processing (NLP), neural networks underpin models for translation, sentiment analysis, and text generation. Autonomous systems, including self-driving cars and robotic control, rely heavily on neural networks to interpret sensor data and make decisions in real-time.

1.3.3 Challenges and Limitations

Even with their achievements, neural networks encounter various challenges. The training process demands substantial computational resources and a vast amount of labeled data. Overfitting is a major concern, as it occurs when a model excels on training data but struggles with new, unseen data. Additionally, neural networks are frequently regarded as "black boxes," making their decision-making processes hard to understand.

1.4 Convolutional Neural Network

Convolutional Neural Networks (CNNs) have achieved state-of-the-art results in many tasks, including computer vision and speech recognition. This success is due to the high accuracy and performance of convolutional layers, which are more computationally efficient and require less memory bandwidth than fully connected layers. The choice of hardware for implementing convolutional layers has a significant impact on their applicability. Central Processing Units (CPUs) are versatile and easy to program, but are relatively inefficient due to their architecture. GPUs are designed for high levels of parallelism, they can handle multiple computations simultaneously, which is well suited to the parallel nature of CNNs at the cost of increased power consumption. Application-specific integrated circuits (ASICs) and field-programmable gate arrays (FPGAs) offer different trade-offs between cost and flexibility for algorithm acceleration. While FPGAs are less powerful and energy efficient due to their reprogrammability, they have lower design costs and can be customised for specific applications.

Given an image as input, the CNN recognizes the elements in the image and classifies them, by giving as output the probability that that image belongs to a particular class (as person, bike, cat, dog, car, ...). Some CNNs are able also to detect the position of the detected object in the image.

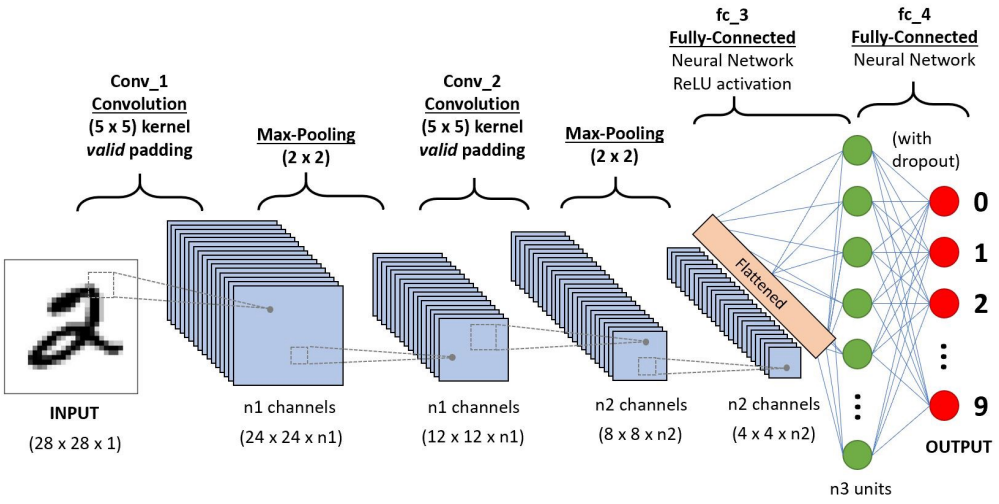


Fig. 1.4 Example of Fully Connected Layers

CNNs are made of different layers, each one with a specific function, that are repeated several times, depending on the CNN implementation. In order to classify the image, the CNN takes as input the related matrix[1], called Feature Map (FM), and makes it flow into these layers, where the FM is convoluted and the learnable parameters, called weights, are updated. Convolutional Neural Networks are a specialized type of ANN designed for processing structured grid data, such as images. CNNs have achieved remarkable success in tasks such as image classification and object detection.
A CNN typically comprises several types of layers:

- **Convolutional Layers:** These layers apply convolution operations to the input, using filters (kernels) to detect features such as edges, textures, and patterns.
- **Activation Layers:** These layers introduce non-linearity into the network, enabling it to model complex relationships.
- **Pooling Layers:** These layers reduce the spatial dimensions of the feature maps, thereby decreasing the computational load and helping to prevent overfitting.
- **Fully Connected Layers:** These layers connect every neuron in one layer to every neuron in the next layer, typically used in the final stages of the network for classification tasks.

1.5 Convolutional Layers

Convolutional Layers are used to detect any kind of shapes in an image. In order to do that, the image, represented by a $n_A \times n_A \times 3$ matrix (RGB coding), is convoluted with specific filter matrices, also called kernels, which store the learnable parameters of the network, i.e. the weights. The kernels are used to detect specific shapes inside an image, such as horizontal and vertical lines and, as the image, they are represented by square matrices, with different weights, depending on which shapes they have to

[1]CNN and, more in general, computer see images as matrices of pixels: if the image is coded in RGB, then CNN will decode it as a H×W×3 matrix, where H and W represent height and width respectively, while 3 is the number of channels, where every of them stores the value of the Red, Green, Blue color, which goes from 0 to 255.

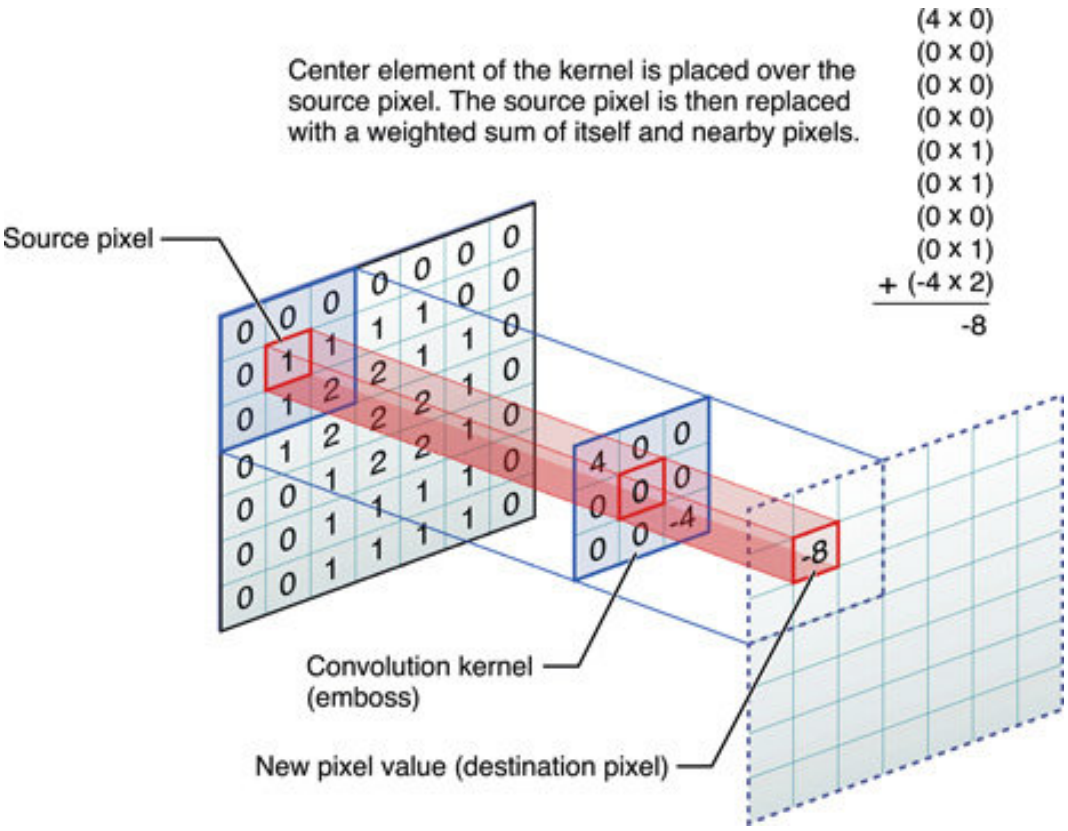


Fig. 1.5 Example of a 2D Convolution

detect. In order to understand what convolution is and how it works, an example is here reported. Considering two matrices (called tensor in Pytorch), A for the image and K for the kernel, both with dimensions 3×3 the convolution is given by:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}, K = \begin{pmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{pmatrix}$$

$$(A * K)_{ij} = \sum_{m=1}^{3} \sum_{n=1}^{3} a_{i+m-1,j+n-1} k_{m,n}$$

1.5.1 Padding and Stride

In convolutional neural networks, padding and stride are fundamental parameters that affect the spatial dimensions and computational characteristics of feature maps.

Padding involves adding extra rows and columns of zeros around the input matrix before performing convolution. This ensures that the spatial dimensions of the output feature map are preserved. Mathematically, if I represents the input size (width or height) and P denotes the amount of padding applied, the output size O of the feature map after convolution can be expressed as:

$$O = \left\lfloor \frac{I + 2P - F}{S} \right\rfloor + 1$$

where F is the filter size and S is the stride. Zero-padding (padding with zeros) is commonly used to maintain spatial dimensions and control the size of the output feature map.

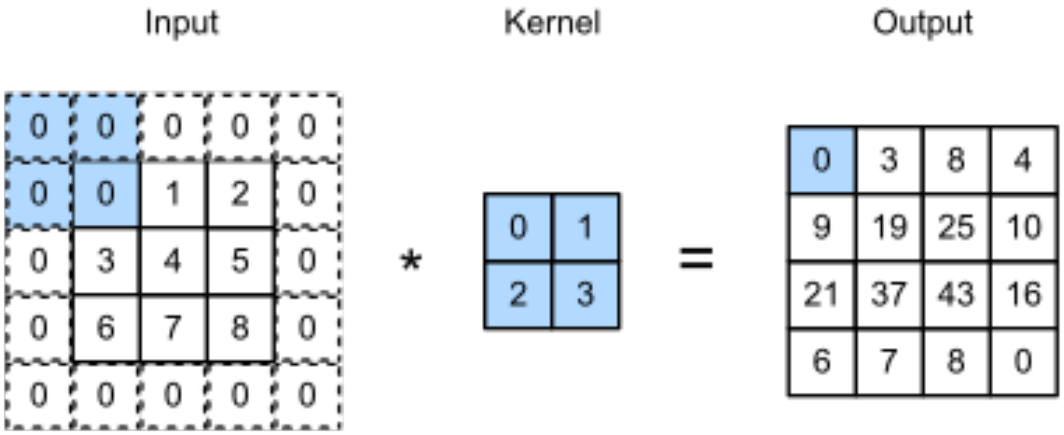


Fig. 1.6 Padding explained visually

Stride, on the other hand, determines the number of pixels the convolutional filter moves across the input matrix in each step. A stride of $S = 1$ means the filter moves one pixel at a time, resulting in overlapping receptive fields and higher computational intensity. Conversely, larger stride values $S > 1$ cause the filter to skip pixels, reducing the output size more quickly. The output size O can be computed using the formula above.

1.6 Normalization Layers

Batch normalization (BN) is a widely used technique in deep neural network training, introduced by Ioffe and Szegedy in 2015, to address the problem of *internal covariate*

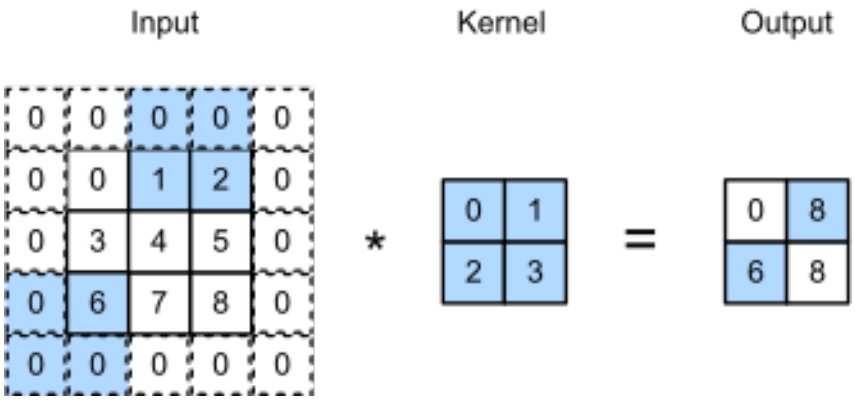


Fig. 1.7 Strides example with $s_w = 2$ and $s_h = 3$

shift. This phenomenon refers to the changes in the distribution of layer inputs during training, which can slow convergence and reduce stability. By normalizing the inputs of each layer to maintain consistent distributions, BN stabilizes and accelerates the training process.

BN can be inserted immediately before or after the activation function. The positioning of BN in relation to the activation function can influence the model's performance; typically, placing BN before non-linear activations like ReLU helps in preserving the learned features' distribution.

For each hidden layer, the mean value μ_B (Eq. 1.1) and the variance σ_B^2 (Eq. 1.3) of the activation values on the batch are computed:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{1.1}$$

Then the input is normalized as:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \tag{1.2}$$

where

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{1.3}$$

The parameter ε is used to prevent division by zero if σ_B becomes null during the training phase.

Finally, the output of the layer is calculated using two trainable parameters (γ and β) to perform a linear transformation:

$$y_i = \gamma \hat{x}_i + \beta \tag{1.4}$$

In each step, the model calculates μ_B and σ_B^2, and updates the trainable parameters γ (scale) and β (bias) using gradient descent. These parameters restore the network's representational capacity after normalization, ensuring that BN does not limit the network's ability to learn complex patterns. During the evaluation phase, instead of batch-specific statistics, the mean value and variance are computed using the running averages accumulated during the training phase, allowing for consistent predictions across varying batch sizes.

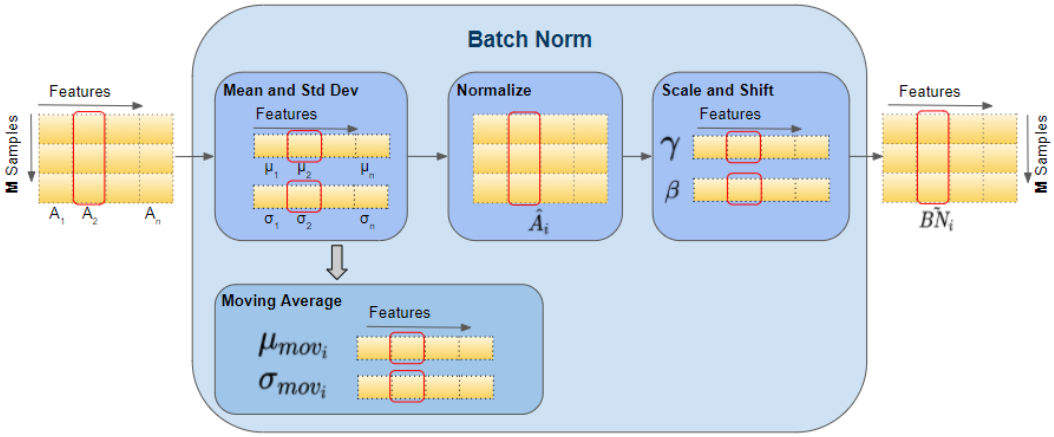


Fig. 1.8 Batch Norm explained visually

1.7 Activation Layers

After the convolutional layer, an activation layer is typically applied. The primary function of the activation layer is to introduce non-linearity into the network, which is critical for enabling the network to model and learn complex patterns in the data. The most commonly used activation function is the Rectified Linear Unit (ReLU), defined as:

$$\text{ReLU}(x) = \max(0, x)$$

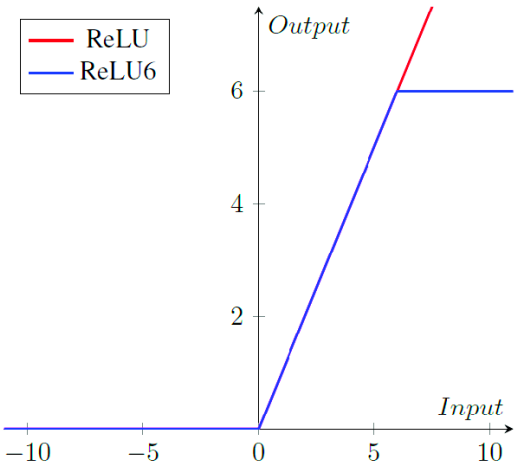


Fig. 1.9 ReLU and ReLU6 activations

As shown in Figure 1.9, ReLU and ReLU6 are commonly used in deep neural networks. ReLU is adopted because of its computational efficiency and its role in overcoming the vanishing gradient problem, which was prevalent in earlier activation functions like sigmoid and tanh. ReLU introduces non-linearity by outputting the input value when it is positive, and zero otherwise, allowing the network to efficiently propagate gradients during training. However, a potential drawback is the *dying ReLU* problem, where neurons can become inactive and slow down learning.

ReLU6 is a variant of ReLU and limits the maximum output to 6. This modification can improve model stability in certain scenarios, such as low-precision computing environments (e.g., mobile or embedded systems), by controlling the range of activations and increasing computational efficiency.

1.8 Pooling Layers

Pooling layers are integral components in convolutional neural networks, particularly effective in managing large input images by reducing their dimensionality. By downsampling the spatial dimensions of feature maps, pooling layers not only reduce computational complexity but also help retain essential spatial hierarchies, which is critical for learning robust features across varying scales. This reduction in spatial dimensions also reduces the risk of overfitting by making the network less sensitive to small shifts or biases in the input data.

Similar to convolutional layers, pooling layers are defined by a kernel with specific parameters such as kernel size n_K and stride s. However, unlike convolutional layers, which use learnable filters, pooling layers act as sliding windows over the feature map, aggregating values within the defined kernel region without introducing new parameters.

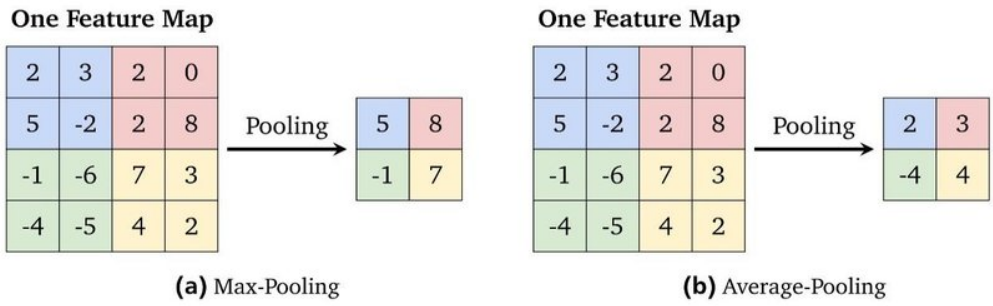


Fig. 1.10 Example of a pooling layer applied to a 4x4 tensor. The result depends on the chosen pooling method.

As illustrated in Figure 1.11, pooling layers reduce the size of the input tensor by summarizing the values in each window. The two primary pooling techniques are max pooling and average pooling. Max pooling selects the maximum value within the kernel window, which helps the model focus on the most prominent features. Max pooling is commonly used in practice because it emphasizes strong activations and helps detect key spatial features. Average pooling, on the other hand, computes the mean value of the elements within the window, resulting in a smoothed representation of the features. This method can be advantageous when a more generalized, less feature-specific summary is desired.

1.9 Fully Connected Layers

In CNNs, the final layer typically consists of a fully connected layer responsible for classification. Prior to this layer, the network processes input through convolutional and pooling layers to extract hierarchical features. The output of the last convolutional layer is a three-dimensional feature map. To prepare for classification, this feature map is flattened into a one-dimensional vector, preserving extracted features while discarding spatial relationships. The flattened vector is then fed into the fully connected layer, which computes weighted sums of its inputs to generate class predictions or scores. This layer's architecture matches the number of classes, enabling the network to output probabilities or scores for each class, facilitating classification based on learned features.

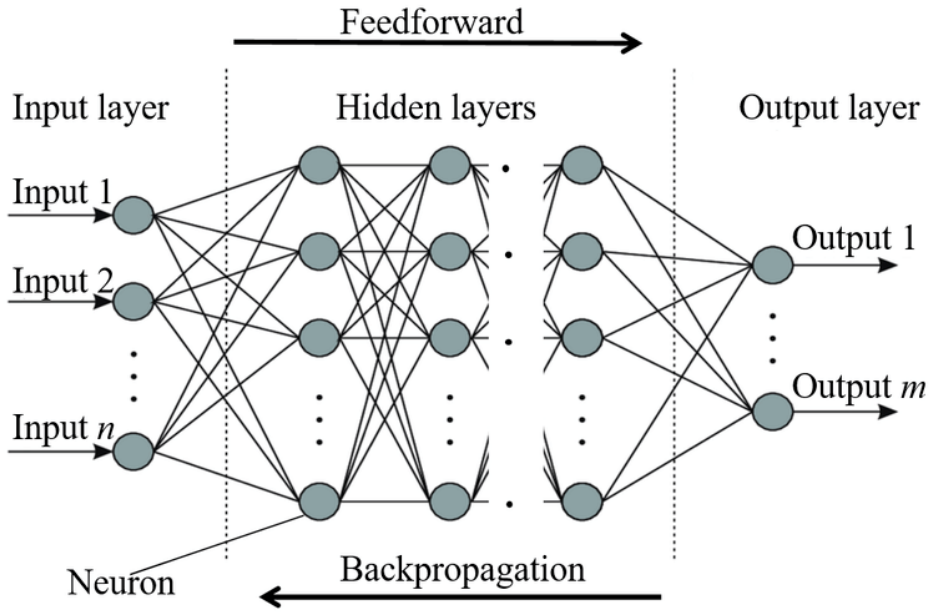


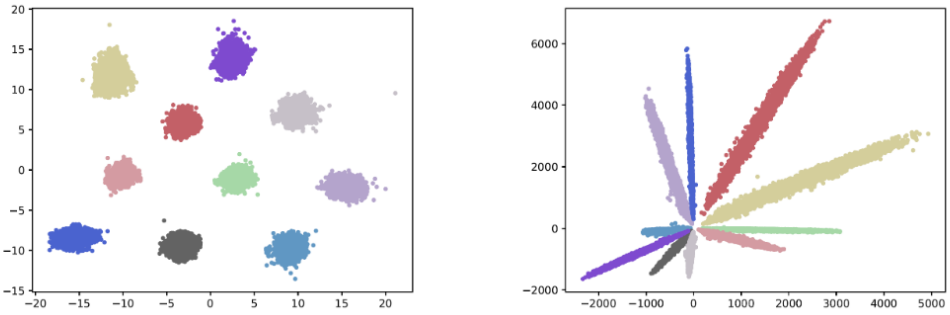
Fig. 1.11 Fully Connected Artificial Neural Network

Chapter 2

AdderNet

2.1 Introduction

The rapid progress of neural networks over the recent years has led to significant breakthroughs in different fields such as image recognition, natural language processing, and autonomous driving. However, these developments come at the cost of higher computational complexity and power consumption. This is particularly the case of Convolutional Neural Networks where the convolution operations are multiplication-intensive. AdderNet represents an innovative solution by substituting these multiplications with additions, which are computationally less expensive [1].



(a) Visualization of features in AdderNets

(b) Visualization of features in CNNs

Figure 1. Visualization of features in AdderNets and CNNs. Features of CNNs in different classes are divided by their angles. In contrast, features of AdderNets tend to be clustered towards different class centers, since AdderNets use the ℓ_1-norm to distinguish different classes. The visualization results suggest that ℓ_1-distance can served as a similarity measure the distance between the filter and the input feature in deep neural networks

2.2 Adder Layers: Theory and Implementation

The core innovation in AdderNet is the adder layer, which replaces the traditional convolution operation. Instead of computing the dot product between input features and filters, the adder layer calculates the ℓ_1 norm of their differences. Formally, for an input feature map $\mathbf{X}$ and a filter $\mathbf{W}$, the adder operation is defined as:

$$Y_{i,j,k} = -\sum_{m=1}^{M}\sum_{n=1}^{N}\sum_{c=1}^{C}\left|X_{i+m-1,j+n-1,c} - W_{m,n,c,k}\right| + b_k$$

where M and N are the height and width of the filter, C is the number of input channels, k indexes the filter, and b_k is the bias term.

2.3 Gradient Computation in AdderNets

The training of AdderNet involves backpropagation, where gradients must be computed for weight updates. Given the loss function L, the gradient of the weights $\mathbf{W}$ with respect to the loss can be derived using the chain rule. The gradient of the adder operation can be expressed as:

$$\frac{\partial L}{\partial W_{m,n,c,k}} = \sum_{i,j}\frac{\partial L}{\partial Y_{i,j,k}} \cdot \text{sign}(X_{i+m-1,j+n-1,c} - W_{m,n,c,k})$$

where sign denotes the sign function. This gradient computation leverages the absolute differences, allowing for efficient weight updates while maintaining the simplicity of addition operations.

2.4 Adaptive Learning Rate in AdderNets

To enhance the training efficiency of AdderNet, an adaptive learning rate strategy is employed. This strategy adjusts the learning rate based on the magnitude of the gradients, ensuring stable convergence, using a cosine annealing schedule. Given an initial learning rate η_{max}, the learning rate η_t at epoch t is calculated as follows:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min}) \left(1 + \cos\left(\frac{t\pi}{T_{max}}\right)\right)$$

where:

- η_{min} is the minimum learning rate.
- η_{max} is the maximum learning rate (usually the initial learning rate).
- t is the current epoch.
- T_{max} is the maximum number of epochs.

For the given scheduler:

$$\eta_t = 0.001 + \frac{1}{2}(\eta_{max} - 0.001) \left(1 + \cos\left(\frac{t\pi}{400}\right)\right)$$

2.5 Skip Connections in AdderNet

Skip connections, also known as residual connections, are crucial in AdderNet for enabling efficient training of deep networks. These connections help in mitigating the vanishing gradient problem by allowing gradients to flow directly through the network layers.

In AdderNet, a residual block can be defined as:

$$\mathbf{Y} = \mathscr{F}(\mathbf{X}, \{\mathbf{W}_i\}) + \mathbf{X}$$

where $\mathbf{X}$ is the input to the residual block, $\mathscr{F}(\mathbf{X}, \{\mathbf{W}_i\})$ represents the series of adder layers within the block, and $\mathbf{Y}$ is the output. The addition of $\mathbf{X}$ ensures that the gradients can bypass certain layers during backpropagation, thus enhancing the training stability and efficiency.

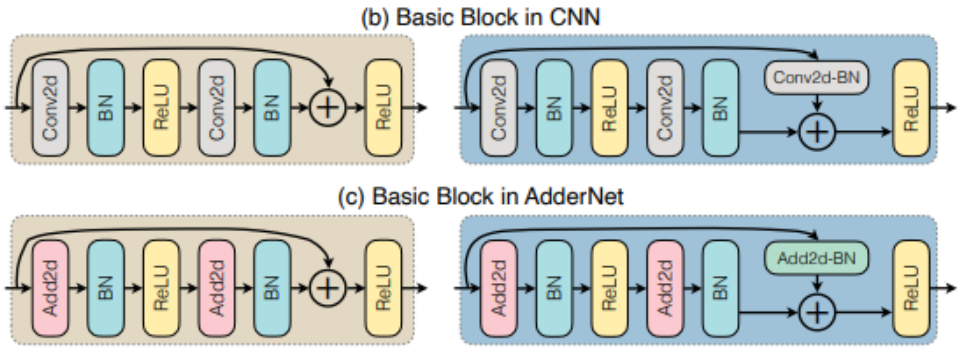


Fig. 2.1 Basic Blocks: comparison between CNN and AddNN

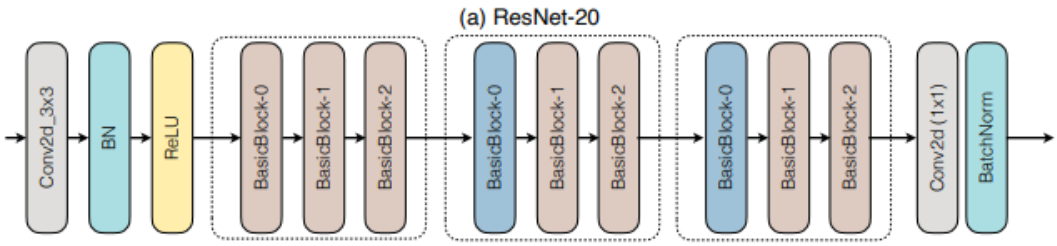


Fig. 2.2 ResNet20 Architecture

2.6 Comparisons with CNNs

Extensive experiments have been conducted to evaluate the performance of AdderNet compared to traditional CNNs.

Metrics such as classification accuracy and loss are considered. Results demonstrate that AdderNet achieves competitive accuracy with significantly reduced computational costs. Table 2.1 summarizes the key performance differences.

Table 2.1 Comparison between CNN and AddNN on a ResNet20 with CIFAR-10 and CIFAR-100

ResNet20	**CIFAR-10**		**CIFAR-100**	
	Accuracy (%)	**Loss**	**Accuracy (%)**	**Loss**
CNN	92.3	0.28	68.4	1.35
AddNN	91.8	0.35	66.8	1.45

AdderNets do not inherently reduce the number of parameters or memory usage. The main advantage of this approach is that, when batch normalization layers are

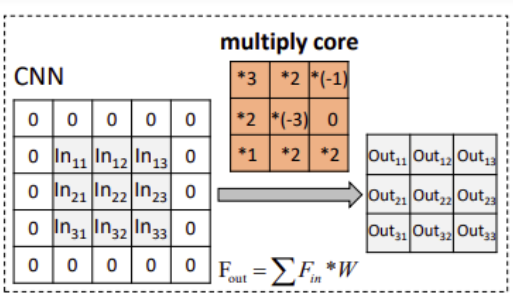


Fig. 2.3 CNN Kernel

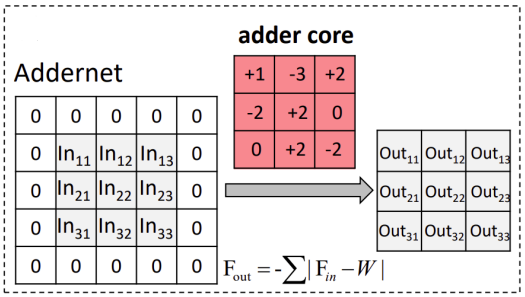


Fig. 2.4 AdderNet Kernel

Fig. 2.5 CNN Kernel compared to AddNN Kernel

excluded, the model eliminates multiplications entirely. Although this may result in a slight loss of accuracy, AdderNet provides a faster computational model.

Furthermore, unlike multiplication operations, the results of addition (or subtraction, in this context) do not require rescaling, as the weights, inputs, and outputs typically remain within the same range. This characteristic not only conserves logic resources but also contributes to a reduction in power consumption.

2.7 Hardware Implementation

It is important to recognize that modern GPUs are specifically optimized for convolution operations, particularly through dedicated modules that execute multiply-accumulate operations. Consequently, AdderNet exhibits slower performance compared to convolutional neural networks on such hardware, especially during the training phase.

Nevertheless, GPUs are notably costly in terms of energy consumption, with their power requirements far exceeding the capabilities of current mobile and edge devices. By implementing an adder kernel on hardware that is capable of executing additions more efficiently, such as FPGAs or ASICs, both the time and resource usage can be significantly minimized compared to CNNs. The fundamental structure of a convolutional kernel relies on a MAC unit, while the adder layer can utilize either a comparator and an adder or two adders.

This architecture is organized into four key components: a data storage unit along with an input/output port, a data path control module, structures for auxiliary

operations (such as pooling and batch normalization), and a parallel kernel operation core. The latter typically has the highest cost in terms of logic resources due to the Single Instruction Multiple Data (SIMD) architecture characteristic of FPGAs. Notably, with a data parallelism of 16 bits and 64 input channels aggregated in the adder tree, AdderNet could potentially utilize 81.6% less logical resources and power compared to convolutional networks [2].

However, the accurate energy benefit estimation of AdderNet still remains a challenge. The overhead associated with data transfer is a critical factor in implementing the model on FPGAs, as the transfer of data from dynamic random-access memory (DRAM) to computational resources represents the primary bottleneck in terms of both energy consumption and time.

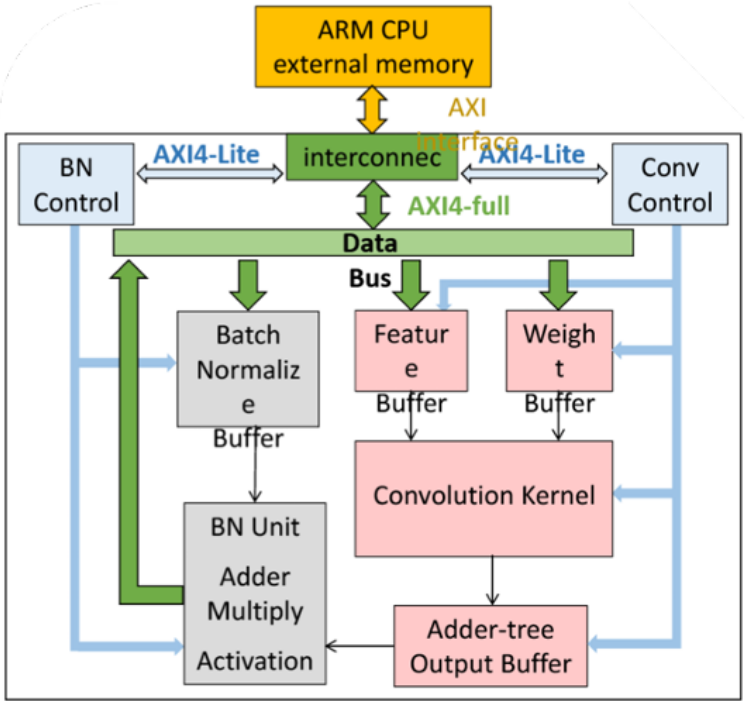


Fig. 2.6 Architecture of the universal AdderNet accelerator

2.8 Applications and Future Work

AdderNet represents a significant advancement in neural network design by reducing the computational cost associated with traditional CNNs.
By replacing multiplications with additions, AdderNet achieves lower power consumption and faster inference times, making it suitable for deployment in resource-constrained environments.

The promising results and potential for further optimization suggest that AdderNet could play a decisive role in the next generation of AI applications, particularly in edge computing scenarios where power efficiency is crucial.

Future work involves exploring advanced quantization techniques, optimizing the adder layer for different hardware platforms, and expanding the application scope to include more complex tasks such as object detection and segmentation.

Chapter 3

Quantization-Aware Training using Brevitas

The primary objective of this research project is to create an 8-bit fixed-point quantized version of the AddNN ResNet20 network [A], while preserving its original structure and features. In order to achieve this, the classes and functions from the Brevitas library have been adopted as tools for quantization.

Brevitas [3] is a PyTorch library specifically designed for neural network quantization, providing extensive support for both post-training quantization (PTQ) and quantization-aware training (QAT). It serves a diverse range of users and purposes. Regarding the creation of quantized models, Brevitas facilitates two primary approaches:

- Creating a quantized model manually involves using Brevitas' specialized quantized layers found in the brevitas.nn module. This process may require adjustments to an existing PyTorch model that was initially designed for floating-point operations.
- Automated generation, on the other hand, involves inputting a floating-point model, which allows Brevitas to automatically create a quantized model definition based on the criteria set by the user.

Once a quantized model is created through either of these methods, it can be used for various applications:

- PTQ (Post-Training Quantization): Starting from a pre-trained floating-point model, the quantized model can undergo post-training quantization;
- QAT (Quantization Aware Training): The quantized model can be utilized for training from scratch or fine-tuning, either independently or on a pre-trained floating-point model;
- PTQ followed by QAT fine-tuning: This approach combines the strengths of both post-training quantization and quantization-aware training for optimal performance.

3.1 Quantization in Brevitas

The Brevitas library is built upon the PyTorch framework, enabling quantization of standard PyTorch layers by providing them with quantized parameters. Considering the following implementation of QuantConv2d, it is evident that the layer is built up by inheriting the standard PyTorch Conv2d layer and instantiating a quantization class called QuantWBIOL (QuantWeightBiasInputOutputLayer). This class receives the input, bias, and weights of the Conv2d layer and returns their quantized versions, ensuring that the convolution performed by Conv2d operates with quantized parameters.

```
1  import math
2  from typing import Optional, Tuple, Type, Union
3
4  import torch
5  from torch import Tensor
6  from torch.nn import Conv1d
7  from torch.nn import Conv2d
8  from torch.nn import functional as F
9
10 from brevitas.function.ops import max_int
11 from brevitas.function.ops_ste import ceil_ste
12 from brevitas.inject.defaults import
       Int8WeightPerTensorFloat
13 from brevitas.quant_tensor import QuantTensor
14
```

```
from .quant_layer import ActQuantType
from .quant_layer import BiasQuantType
from .quant_layer import QuantWeightBiasInputOutputLayer
    as QuantWBIOL
from .quant_layer import WeightQuantType

__all__ = ['QuantConv1d', 'QuantConv2d']
[...]
class QuantConv2d(QuantWBIOL, Conv2d):

    def __init__(
            self,
            in_channels: int,
            out_channels: int,
            kernel_size: Union[int, Tuple[int, int]],
            stride: Union[int, Tuple[int, int]] = 1,
            padding: Union[int, Tuple[int, int]] = 0,
            dilation: Union[int, Tuple[int, int]] = 1,
            groups: int = 1,
            padding_mode: str = 'zeros',
            bias: Optional[bool] = True,
            weight_quant: Optional[WeightQuantType] =
               Int8WeightPerTensorFloat,
            bias_quant: Optional[BiasQuantType] = None,
            input_quant: Optional[ActQuantType] = None,
            output_quant: Optional[ActQuantType] = None,
            return_quant_tensor: bool = False,
            device: Optional[torch.device] = None,
            dtype: Optional[torch.dtype] = None,
            **kwargs) -> None:
        # avoid an init error in the super class by
           setting padding to 0
        if padding_mode == 'zeros' and padding == 'same'
            and stride > 1:
            padding = 0
            is_same_padded_strided = True
        else:
            is_same_padded_strided = False
```

```
49          Conv2d.__init__(
50              self,
51              in_channels=in_channels,
52              out_channels=out_channels,
53              kernel_size=kernel_size,
54              stride=stride,
55              padding=padding,
56              padding_mode=padding_mode,
57              dilation=dilation,
58              groups=groups,
59              bias=bias,
60              device=device,
61              dtype=dtype)
62          QuantWBIOL.__init__(
63              self,
64              weight_quant=weight_quant,
65              bias_quant=bias_quant,
66              input_quant=input_quant,
67              output_quant=output_quant,
68              return_quant_tensor=return_quant_tensor,
69              **kwargs)
70          self.is_same_padded_strided =
               is_same_padded_strided
71  [...]
```

Listing 3.1 Brevitas implementation of QuantConv2d

Brevitas already provides several quantizers (located in the brevitas.quant folder in [4]), each of which is widely configurable by the user according to their requirements. Each quantizer is characterized by different parameters that define its functionality; the main ones are:

- **Quant Type**: The type of quantization implemented for the parameter. The most commonly used types are:
  - **QuantType.INT**: Integer quantization implemented by the IntQuant() module. Given an input tensor, IntQuant() performs scaled, shifted, uniform integer quantization based on the parameters scale, zero-point,

and bit-width, which are provided as arguments. It returns the quantized tensor in a de-quantized format.

  - **QuantType.BINARY**: Binary quantization implemented by the BinaryQuant() module. It returns the quantized output in the de-quantized format, along with the scale, zero-point, and bit width, which is 1 in this case,
  - **QuantType.TERNARY**: Ternary quantization implemented by the TernaryQuant() module. Given an input tensor, it returns its quantized output in de-quantized format, with scale, zero-point, and bit width, which is always 2 in this case.

- **Bit Width**: The number of bits used to quantize the original parameter.
- **Narrow Range**: A boolean parameter that, if set to True, implements the value in a range from $(-2^{N-1}+1)$ to (2^N-1), instead of -2^{N-1} to (2^N-1), where N corresponds to the bit width. For instance, if N=8 and narrow range=True, the quantized value will range from -127 to 127 instead of -128 to 127; this enhances hardware inference efficiency.
- **Signed**: If set to True, the quantized value can be both positive and negative.

In the specific case of the QuantConv2d layer, it is used by default the *Int8WeightPerTensorFloat* quantizer for the weights parameter. As reported in [4], this is an "8-bit narrow per-tensor signed integer weight quantizer with a floating-point scale factor computed from backpropagated statistics of the weight tensor," meaning the convolution kernel weights are quantized to 8 bits in a range from -127 to 127, with a floating-point scale factor.

The formula used by *Int8WeightPerTensorFloat* to compute the scale is:

$$\text{scale} = \frac{\text{th}}{\text{int th}}$$

where th is the threshold defined as the maximum absolute value in an input tensor X:

$$\text{th} = \max_{i,j=1,\ldots,\dim(X)} \{|x_{i,j}|\}$$

and int th is the integer threshold given by:

$$\text{int th} = \begin{cases} 2^{N-1} - 1 & \text{if signed=True} \\ 2^{N} - 1 & \text{if signed=False} \end{cases}$$

The quantization is then performed by dividing the floating-point value by the scale factor:

$$\text{IntW} = \frac{\text{FPW}}{\text{scale}}$$

Considering the following numerical example, where quantization is performed on 8 bits with *signed=True*, the steps are:

$$\text{FPW} = \begin{pmatrix} 6.789 & 2.310 & 7.938 \\ -4.567 & 1.234 & 0.876 \\ -2.345 & 5.678 & -6.789 \end{pmatrix}$$

$$\text{th} = \max |\text{FPW}_{ij}| = 7.938$$

$$\text{int th} = 2^{N-1} - 1 = 2^{8-1} - 1 = 127$$

$$\text{scale} = \frac{\text{th}}{\text{int th}} = \frac{7.938}{127} = 0.0625$$

To compute the quantized weight:

$$\text{IntW} = \frac{\text{FPW}}{\text{scale}} \approx \begin{pmatrix} 109 & 37 & 127 \\ -73 & 20 & 14 \\ -38 & 91 & -109 \end{pmatrix}$$

It is important to note that during network training, both the quantized parameters and scales are recalculated each time the optimizer updates the original non-quantized parameters (FPW in the example). Also, it is crucial to highlight that the convolution is not performed between the input and the integer representation of the weight, but with the quantized weight in the de-quantized format. The quantizer, given the scale, zero point, bit width, and input tensor X, computes its integer representation y_{int}, but then returns the quantized parameter in the de-quantized float representation. Thus, during training, the quantized layers' operations are performed among floating-point values.

The de-quantized format of the weights is given by:

$$\text{DeQuantW} = \text{IntW} \cdot \text{scale}$$

which in this specific case is:

$$\text{DeQuantW} = \text{IntW} \cdot \text{scale} = \begin{pmatrix} 109 & 37 & 127 \\ -73 & 20 & 14 \\ -38 & 91 & -109 \end{pmatrix} \cdot 0.0625 = \begin{pmatrix} 6.789 & 2.310 & 7.938 \\ -4.567 & 1.234 & 0.876 \\ -2.345 & 5.678 & -6.789 \end{pmatrix}$$

When inferring the network on FPGA, the weights are exported and stored in the integer quantized format, and to maintain the same accuracy as during training, the output feature map (FM) will be multiplied by the scale factor, since:

$$\text{InputFM} * \text{DeQuantW} = \text{InputFM} * \text{IntW} \cdot \text{scale}$$

A similar layer construction is adopted for other Brevitas layers. The following table outlines several quantized layers and their corresponding unquantized PyTorch equivalents.

PyTorch Layer	Brevitas Layer
Convolutional Layers	
nn.Conv1d	QuantConv1d
nn.Conv2d	QuantConv2d
nn.ConvTranspose1d	QuantConvTranspose1d
nn.ConvTranspose2d	QuantConvTranspose2d
Pooling Layers	
nn.MaxPool1d	QuantMaxPool1d
nn.MaxPool2d	QuantMaxPool2d
nn.AvgPool2d	QuantAvgPool2d
nn.AdaptiveAvgPool2d	QuantAdaptiveAvgPool2d
Non-linear Activations	
nn.Hardtanh	QuantHardTanh
nn.ReLU	QuantRelu
nn.Sigmoid	QuantSigmoid
nn.Tanh	QuantTanh
Dropout Layers	
nn.Dropout	QuantDropout

Table 3.1 Mapping of main PyTorch layers to available Brevitas layers

3.1.1 Quantization Classes

Brevitas provides various quantization classes, which are tailored for different aspects of neural networks. These include $Int8WeightPerTensorFixedPoint$ for weight quantization and $Int8ActPerTensorFixedPoint$ for activation quantization, each offering specific benefits in terms of precision and computational efficiency.

3.1.2 Quantizers

In a broad context, a quantizer refers to any implementation of a quantization technique. Brevitas offers flexibility in employing various quantization methods. To be compliant with Brevitas documentation, a quantizer is a subclass of *brevitas.inject.ExtendedInjector* that contains a *tensor_quant* attribute. This attribute refers to an instance of a *torch.Module* responsible for managing quantization. Quantizers in Brevitas define the process of converting floating-point numbers into fixed-point integers suitable for deployment on hardware accelerators. They aim to minimize loss in model accuracy while optimizing the efficiency of hardware utilization.

3.1.3 Quantization Function

The quantization function in Brevitas integrates smoothly into the forward pass of the neural network, applying quantization to weights, activations, and biases according to what has been defined by the user. This functionality is crucial for maintaining model performance across different quantization schemes.

3.1.4 Model Optimization

Quantization-aware optimization techniques in Brevitas, such as the integration of quantized layers and functions, contribute to minimizing computational overhead and memory usage without compromising model accuracy. This optimization is particularly beneficial for resource-constrained deployment scenarios.

3.1.5 Deployment Support

Brevitas supports deployment on various platforms and accelerators by ensuring compatibility with ONNX, an open format for AI models. This support enables the integration into existing software ecosystems and facilitate an efficient execution on hardware.

3.2 AddNN ResNet20 Quantization using Brevitas

The chosen strategy includes an initial step involving the creation of a custom layer tailored for the quantized version of the adder2d, which is currently unavailable in the Brevitas library. Regarding the existing Brevitas quantized layers [4], they have been substituted for the unquantized counterparts, as needed, with the main exception of the *nn.BatchNorm2d* quantized alternative, named *BatchNorm2dToQuantScaleBias*.
This exception is made to ensure the integrity and convergence of the neural network during training. Presented below is the code for the final quantized network, AdNN QuantResNet20.

3.2.1 AddNN QuantResNet20 Code

As evident from the provided code, the chosen quantization method utilizes the *Int8WeightPerTensorFixedPoint* quantizer for weights and *Int8ActPerTensorFixedPoint* for input and output activations. The former, as described in [4], is an "8-bit narrow per-tensor signed fixed-point weight quantizer with the radix point computed from backpropagated statistics of the weight tensor."

Meanwhile, the latter is defined as an "8-bit per-tensor signed int activations fixed-point quantizer with learned radix point initialized from runtime statistics." The quantized model of AddNN ResNet20, named **QuantResNet20**, has been developed with the following code:

```
import torch
import torch.nn as nn
import brevitas
import brevitas.nn as qnn
from QuantAdd2d import QuantAdd2d
from brevitas.quant import Int8WeightPerTensorFixedPoint
from brevitas.quant import Uint8ActPerTensorFixedPoint
from brevitas.quant import Int8ActPerTensorFixedPoint


def conv3x3(in_planes, out_planes, stride=1):
    " 3x3 convolution with padding "
    return QuantAdd2d(in_planes, out_planes, kernel_size
       =3, stride=stride, padding=1, bias=False)

class BasicBlock(nn.Module):
    expansion=1

    def __init__(self, inplanes, planes, stride=1,
       downsample=None):
        super(BasicBlock, self).__init__()
        self.conv1 = conv3x3(inplanes, planes, stride =
           stride)
        self.bn1 = nn.BatchNorm2d(planes)
        self.relu = qnn.QuantReLU(bit_width=8, act_quant
           =Uint8ActPerTensorFixedPoint, input_quant=
           Uint8ActPerTensorFixedPoint,
           return_quant_tensor=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = nn.BatchNorm2d(planes)
        self.downsample = downsample
        self.stride = stride
        self.input_quant = qnn.QuantIdentity(
            act_quant=Int8ActPerTensorFixedPoint
        )

    def forward(self, x):
```

```
32          residual = x
33
34          out = self.conv1(x)
35          out = self.bn1(out)
36          out = self.relu(out)
37
38          out = self.conv2(out)
39          out = self.bn2(out)
40          out = self.input_quant(out)
41
42          if self.downsample is not None:
43              residual = self.downsample(x)
44
45          residual = self.input_quant(residual)
46
47          out += residual
48          out = self.relu(out)
49
50          return out
51
52
53  class ResNet(nn.Module):
54
55      def __init__(self, block, layers, num_classes=10):
56          super(ResNet, self).__init__()
57          self.inplanes = 16
58          self.conv1 = qnn.QuantConv2d(3, 16, kernel_size
                =3, stride=1, padding=1, bias=False,
                weight_quant=Int8WeightPerTensorFixedPoint,
59               input_quant=Int8ActPerTensorFixedPoint,
60               output_quant=Int8ActPerTensorFixedPoint)
61          self.bn1 = nn.BatchNorm2d(16)
62          self.relu = qnn.QuantReLU(bit_width=8, act_quant
                =Uint8ActPerTensorFixedPoint, input_quant=
                Uint8ActPerTensorFixedPoint,
                return_quant_tensor=True)
63          self.layer1 = self._make_layer(block, 16, layers
                [0])
```

```
64          self.layer2 = self._make_layer(block, 32, layers
               [1], stride=2)
65          self.layer3 = self._make_layer(block, 64, layers
               [2], stride=2)
66          self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
67          self.fc = qnn.QuantConv2d(64 * block.expansion,
               num_classes, 1, bias=False, weight_quant=
               Int8WeightPerTensorFixedPoint,
68              input_quant=Int8ActPerTensorFixedPoint,
69              output_quant=Int8ActPerTensorFixedPoint)
70          self.bn2 = nn.BatchNorm2d(num_classes)
71
72          for m in self.modules():
73              if isinstance(m, nn.BatchNorm2d):
74                  m.weight.data.fill_(1)
75                  m.bias.data.zero_()
76
77      def _make_layer(self, block, planes, blocks, stride
           =1):
78          downsample = None
79          if stride != 1 or self.inplanes != planes *
               block.expansion:
80              downsample = nn.Sequential(
81                  QuantAdd2d(self.inplanes, planes * block
                       .expansion, kernel_size=1, stride=
                       stride, bias=False),
82                  nn.BatchNorm2d(planes * block.expansion)
83              )
84
85          layers = []
86          layers.append(block(inplanes = self.inplanes,
               planes = planes, stride = stride, downsample
               = downsample))
87
88          if stride != 1:
89              input_quant = qnn.QuantIdentity(act_quant=
                   Int8ActPerTensorFixedPoint)
90              layers.append(input_quant)
```

```
91
92          self.inplanes = planes * block.expansion
93          for _ in range(1, blocks):
94              layers.append(block(inplanes = self.inplanes
                   , planes = planes))
95
96          return nn.Sequential(*layers)
97
98      def forward(self, x):
99
100         x = self.conv1(x)
101         x = self.bn1(x)
102         x = self.relu(x)
103
104         x = self.layer1(x)
105         x = self.layer2(x)
106         x = self.layer3(x)
107
108         x = self.avgpool(x)
109         x = self.fc(x)
110         x = self.bn2(x)
111
112         return x
113
114
115 def resnet20(**kwargs):
116     return ResNet(BasicBlock, [3, 3, 3], **kwargs)
```

Listing 3.2 Python Code for QuantAddResNet20

As highlighted by the code, the standard PyTorch layers such as `Conv2d` and `ReLU` have been substituted with their Brevitas counterparts. However, the `BatchNorm2d` layer from PyTorch has been retained for reasons delineated in *Section 3.2.5*, while the Global AvgPool `AdaptiveAvgPool2d` (used instead of the classical AvgPool2d of the CNN ResNet20) has been left unchanged, since there is no need to quantize its values. Furthermore, the activation function has been quantized to 8 bits.

Specifically, the `QuantReLU` layer operates similarly to the conventional `ReLU` layer, with the primary distinction that its output is quantized to 8 bits.

3.2.2 Quantization of adder2d in Brevitas

The development of a quantized adder2d layer has sparked numerous experiments, commencing with the creation of a bespoke QuantAdd2d akin to QuantConv2d. Regrettably, neither the compiler during the initial training phase nor subsequent export functions could recognize this product. This is likely due to the inherent limitation of the built-in class *QuantWeightBiasInputOutputLayer*, commonly referred to as *QuantWBIOL*, which is designed to operate exclusively on layers native to the PyTorch library.
To address this significant obstacle, the selected approach involved extending the original adder2d [B] class definition to incorporate novel functionalities for the quantization of weights, biases, input, and output activations.
Below is the code excerpt for the finalized QuantAdd2d.

3.2.3 QuantAdd2d Code

```
1
2  import torch
3  import torch.nn as nn
4  import numpy as np
5  from torch.autograd import Function
6  import math
7  import brevitas.nn as qnn
8  from brevitas.quant import Int8WeightPerTensorFixedPoint
9  from brevitas.quant import Int8ActPerTensorFixedPoint
10
11 def adder2d_function(X, W, stride=1, padding=0):
12     n_filters, d_filter, h_filter, w_filter = W.size()
13     n_x, d_x, h_x, w_x = X.size()
14
15     h_out = (h_x - h_filter + 2 * padding) / stride + 1
16     w_out = (w_x - w_filter + 2 * padding) / stride + 1
17
```

```
18      h_out, w_out = int(h_out), int(w_out)
19
20      X_col = torch.nn.functional.unfold(X.view(1, -1, h_x
            , w_x), int(h_filter), dilation=1, padding=
            padding, stride=stride).view(n_x, -1, h_out*w_out
            )
21      X_col = X_col.permute(1,2,0).contiguous().view(X_col
            .size(1),-1)
22      W_col = W.view(n_filters, -1)
23
24      out = adder.apply(W_col,X_col)
25
26      out = out.view(n_filters , h_out, w_out, n_x)
27      out = out.permute(3, 0, 1, 2).contiguous()
28
29      return out
30
31
32  class adder(Function):
33      @staticmethod
34      def forward(ctx, W_col, X_col):
35          ctx.save_for_backward(W_col,X_col)
36          output = -(W_col.unsqueeze(2)-X_col.unsqueeze(0)
                ).abs().sum(1)
37          return output
38
39      @staticmethod
40      def backward(ctx,grad_output):
41          W_col,X_col = ctx.saved_tensors
42          grad_W_col = ((X_col.unsqueeze(0)-W_col.
                unsqueeze(2))*grad_output.unsqueeze(1)).sum
                (2)
43          grad_W_col = grad_W_col/grad_W_col.norm(p=2).
                clamp(min=1e-12)*math.sqrt(W_col.size(1)*
                W_col.size(0))/5
44          grad_X_col = (-(X_col.unsqueeze(0)-W_col.
                unsqueeze(2)).clamp(-1,1)*grad_output.
                unsqueeze(1)).sum(0)
```

```
45
46          return grad_W_col, grad_X_col
47
48  class QuantAdd2d(nn.Module):
49      def __init__(self, input_channel, output_channel,
            kernel_size, stride=1, padding=0, bias=False):
50          super(QuantAdd2d, self).__init__()
51          self.stride = stride
52          self.padding = padding
53          self.input_channel = input_channel
54          self.output_channel = output_channel
55          self.kernel_size = kernel_size
56          self.adder = torch.nn.Parameter(nn.init.normal_(
               torch.randn(output_channel, input_channel,
               kernel_size, kernel_size)))
57          self.bias = bias
58          if bias:
59              self.b = torch.nn.Parameter(nn.init.uniform_
                   (torch.zeros(output_channel)))
60
61
62          self.adder_quant = qnn.QuantIdentity(
63              weight_quant=Int8WeightPerTensorFixedPoint,
64              act_quant=Int8ActPerTensorFixedPoint
65
66          )
67
68          self.input_quant = qnn.QuantIdentity(
69              act_quant=Int8ActPerTensorFixedPoint
70          )
71
72          self.output_quant = qnn.QuantIdentity(
73              act_quant=Int8ActPerTensorFixedPoint
74          )
75
76      def forward(self, x):
77          x_quant = self.input_quant(x)
```

```
78          adder_quant_output = self.adder_quant(self.adder
               )
79
80          output = adder2d_function(x_quant,
               adder_quant_output, self.stride, self.padding
               )
81
82          if self.bias:
83              output += self.b.unsqueeze(0).unsqueeze(2).
                   unsqueeze(3)
84
85          output_q = self.output_quant(output)
86
87          return output_q
```

Listing 3.3 Python Code for QuantAdd2d Layer

The given code demonstrates the implementation of the custom adder2d layer, leveraging both PyTorch and Brevitas libraries to facilitate quantization. The primary function, *adder2d_function*, defines the forward pass of the adder layer, calculating the output dimensions based on input dimensions, filter size, stride, and padding. It utilizes PyTorch's *unfold* function to convert the input tensor into column format for efficient matrix multiplication with the weight tensor, followed by a custom *adder* function to perform element-wise addition and subsequent operations. The *adder* class inherits from *torch.autograd.Function* to enable the implementation of both forward and backward passes. In the forward pass, the method computes the output as the sum of absolute differences between input and weights, whereas the backward pass calculates the gradients with respect to the input and weight tensors, ensuring gradient flow during backpropagation.

The *QuantAdd2d* class, derived from *nn.Module*, integrates quantization into the custom adder layer. The constructor initializes various parameters, including stride, padding, input and output channels, kernel size, and an optional bias.
It also defines the quantization operations using Brevitas' *QuantIdentity* with *Int8WeightPerTensorFixedPoint* and *Int8ActPerTensorFixedPoint* quantizers for both weights and activations. During the forward pass, the input tensor is first quantized,

followed by quantization of the adder parameters. The custom *adder2d_function* then computes the output, which is subsequently quantized before being returned.

3.2.4 Training process and results

The training process of the quantized AdderNet on ResNet20 involved rigorous optimization experiments over 500 epochs to achieve the best performance. Initial configurations using the SGD optimizer faced significant issues, such as loss function and gradient explosions, which impeded effective training. Similarly, using the Adam optimizer with the CyclicCosAnnealingLR scheduler resulted in early model stalling, indicating that this learning rate schedule was not conducive to stable training. The combination of AdamW optimizer with a StepLR scheduler also proved challenging due to an unstable loss function.

Subsequent tests with the Adam optimizer and ReduceLROnPlateau scheduler produced an accuracy of 57% accuracy, indicating insufficient improvement with adaptive learning rate adjustments. Additionally, using the Adam optimizer with a lower learning rate of 0.001 and the CosineAnnealingLR scheduler resulted in an accuracy of 81%. The Adam optimizer with the CyclicLR scheduler achieved an accuracy of 82%, suggesting that cyclical learning rate adjustments were not optimal for this model.

The final training process for the quantized AdderNet on ResNet20 utilized this optimal configuration with the Adam optimizer and cosine annealing scheduler. Despite applying dataset augmentation techniques during preprocessing to enhance robustness, these did not lead to significant performance gains. The final training regimen, which involved gradually decreasing the learning rate from 0.01 to 0.001 over 500 epochs, achieved an accuracy of 86.61%, compared to the base architecture's accuracy of 91.4%. This result underscores the AdderNet's ability to balance computational efficiency with competitive accuracy, highlighting its potential for deployment in resource-constrained environments.

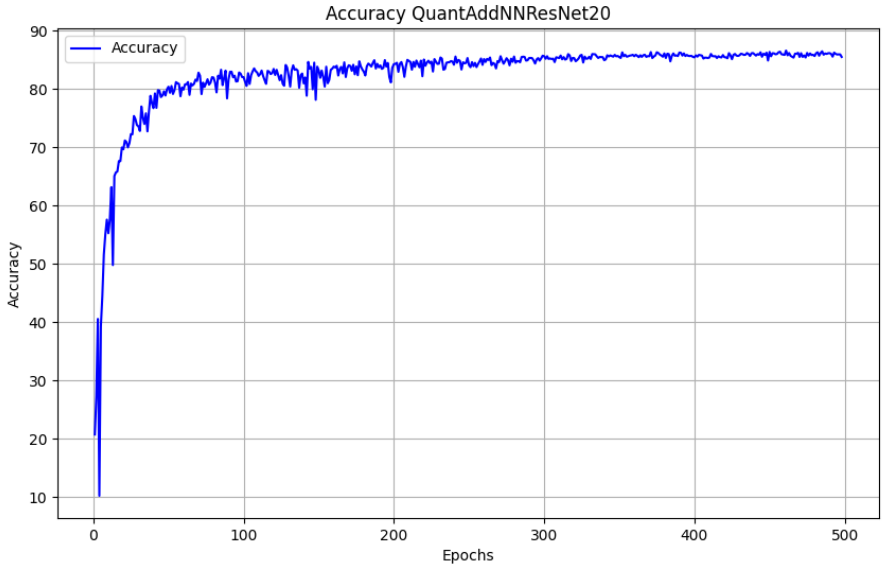


Fig. 3.1 Training result: Accuracy

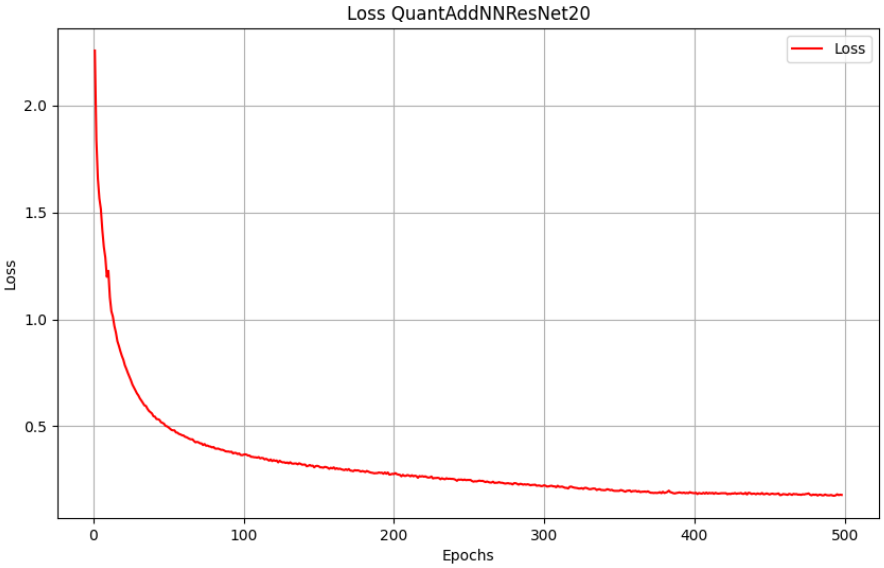


Fig. 3.2 Training result: Loss

3.2.5 Quantization of BatchNorm2d in Brevitas

Batch normalization (BN) fusion offers a practical solution for implementing resource-efficient CNN accelerators without adding hardware overhead. During inference, the running mean and variance parameters of BN layers are fixed and can be realized using 1×1 convolutions. These fixed BN layers can be fused with the preceding convolutional layers, further reducing hardware and runtime costs. Subsequently, post-training quantization can be applied to the fused weight parameters to minimize hardware expenses, with negligible loss in inference accuracy. However, BN fusion is unsuitable for AdderNet due to the incompatibility between the Sum of Absolute Difference operation in AdderNet and the 1×1 convolution operation in the fixed BN layer.

To prevent excessive consumption of DSPs in subsequent hardware implementations, it is necessary to quantize the Batch Normalization (BN) layers. However, given that Brevitas is a relatively new research tool, the provided quantized version, *BatchNorm2dToQuantScaleBias*, has been widely reported to exhibit malfunctioning behavior. This issue was corroborated by the undersigned, as the introduction of this version caused the model to lose all prediction accuracy drastically. Consequently, alternative approaches were considered, including the PyTorch quantization library and the onnxruntime quantization tools. However, it was found that the PyTorch library currently does not support the quantization of BatchNorm2d either statically or dynamically, and the same limitation applies to the onnxruntime.

Drawing from the insights provided in [5], in response to the pivotal challenge at hand, a strategic adaptation has been undertaken. Specifically, all BatchNorm2d layers, with the notable exception of the initial and terminal ones, which can be still merged in their previous Conv2d layers respectively, have been substituted with Conv2d depthwise counterparts, realized through 1x1 convolutions. The replacements have been done respecting the relation among the two different kind of layers' parameters. Notice that child is the label of the original BatchNorm2d layer, while conv1x1 is the new Conv2d layer's label.

$$\text{scale} = \frac{\text{child.weight}}{\sqrt{\text{child.running\_var} + \text{child.eps}}} \tag{3.1}$$

$$\text{weight\_tensor} = \text{scale.view}(\text{num\_features}, 1, 1, 1) \tag{3.2}$$

$$\text{conv1x1.weight.copy\_(weight\_tensor)} \tag{3.3}$$

$$\text{bias\_tensor} = \text{child.bias} - \text{scale} \times \text{child.running\_mean} \tag{3.4}$$

$$\text{conv1x1.bias.copy\_(bias\_tensor)} \tag{3.5}$$

Following this manipulation, the revised model underwent evaluation to make sure that no performance degradation occurred, compared to its predecessor. Notably, the prediction accuracy remained unchanged, affirming the logical equivalence between the model before and after the substitution was carried out.
Having circumvented the BN quantization challenge, different alternatives for the quantization of the newly inserted Conv2d layers have been explored. Regrettably, neither the PyTorch nor the ONNX Runtime libraries currently offer support for quantizing Conv2d depthwise layers. Consequently, Brevitas emerges as the primary solution once more. Nevertheless, distinguishing itself from the aforementioned implementations, a per-channel quantization strategy has been adopted for the layers' weights, while input, output activations, and bias persist in being quantized via a per-tensor approach.
This decision stems from the observation that varying numbers of input/output channel groups can result in weights exhibiting high variances across groups. Such discrepancies can lead to a significant reduction in accuracy when employing a per-tensor quantization approach for weights. In contrast, the selected strategy mitigates this issue, thereby preserving the final model performance. However, this comes at the expense of slightly increased memory usage due to the larger number of parameters that need to be stored.

In consideration of resource optimization, a conventional batch normalization layer has been adopted for future steps despite the potential advantages of the proposed Conv2D quantized per-channel approach. This decision stems from concerns over the operational efficiency of DSP slices, where the adoption of the depthwise Conv2d quantized per-channel strategy could lead to an increase in floating-point operations (Float8), potentially offsetting the resource savings achieved through quantization of the AdderNet. This cautious approach aims to balance the preservation of model performance with the efficient utilization of FPGA resources. While

the per-channel quantization strategy offers benefits in mitigating weight variances across input/output channel groups, its implementation would necessitate careful management to avoid excessive resource consumption and maintain the overall efficiency of the FPGA-based system. In summary, due to considerations of trade-offs, it has been determined that *Float32 BN* is likely to be the most significant bottleneck in the design.

Chapter 4

ONNX

ONNX can be compared to a programming language specialized in mathematical functions. It defines all the necessary operations a machine learning model needs to implement its inference function with this language. ONNX aims at providing a common language any machine learning framework can use to describe its models. The first scenario is to make it easier to deploy a machine learning model in production. An ONNX interpreter (or runtime) can be specifically implemented and optimized for this task in the environment where it is deployed. With ONNX, it is possible to build a unique process to deploy a model in production and independent from the learning framework used to build the model *.onnx* implements a python runtime that can be used to evaluate ONNX models and to evaluate ONNX operators.

4.1 Model Exportation in ONNX

The transformation of the quantized ResNet20 model into an ONNX representation is essential for deploying and performing inference on hardware accelerators.
Python code is presented for exporting a quantized ResNet20 model to the ONNX format. The *QuantResNet20* model is instantiated and loaded with weights from the trained model checkpoint using PyTorch. Subsequently, the model is set to evaluation mode and an ONNX export path is specified. To facilitate the export, a dummy input tensor is generated to define the input shape required by the model `(1, 3, 32, 32)`.

```
1
2  import os
3  import torch
4  from brevitas.export import export_onnx_qcdq
5  from QuantResNet20 import resnet20
6
7  # Best Accuracy-Checkpoint's Path
8  checkpoint_path = './output/checkpoint.pth'
9
10 # Loading of QuantResNet20 Model
11 net = resnet20().cuda()
12
13 # Loading of the Best Accuracy-Checkpoint's Weights
14 checkpoint = torch.load(checkpoint_path)
15 net.load_state_dict(checkpoint['model_state_dict'])
16
17 # Exporting the ONNX model
18 net.eval()  # Model has to be set in evaluation model
19 export_onnx_path = "./output/QuantResNet20.onnx"  # Path
       of the exported ONNX file
20 input_shape = (1, 3, 32, 32)    # Shape of the inputs
     required by the proposed model
21 dummy_input = torch.randn(input_shape, device='cuda')
     # Random input for the model to be exported
22 export_onnx_qcdq(net, dummy_input, export_path=
     export_onnx_path, opset_version=13)  # Final export
     instruction
```

Listing 4.1 Python Code for Exportation

4.2 ONNX Model Visualization

To visualize the exported ONNX model, Netron has been utilized.
Netron is a neural network, deep learning and machine learning models visualization tool developed basing on the Electron platform [6].

It supports the visualization of many mainstream AI framework models and supports multiple platforms (such as Mac, Windows, and Linux).

Given that the *adder2d* layer is not part of the original PyTorch library, there is currently no corresponding operator in the ONNX standard. This presents a potential obstacle when moving towards the inference stage. However, all the individual operations comprising the layer are recognized as standard operators by ONNX. Therefore, the adopted approach involves replacing all the subgraphs related to the *adder2d* layers with custom nodes. For simplicity, these custom nodes must align with the attributes of the standard *conv2d* operator, as illustrated in Figures 4.1 and 4.2.

To implement this solution, the replacement functions provided by the *onnx – graphsurgeon* tool [7] have been adapted and utilized. Initially, some placeholder *Clip* nodes were used to substitute the relevant subgraphs, after which the necessary attributes were assigned to it by defining the new custom node, *adder2d*. This custom node's features are now configurable according to specific requirements.

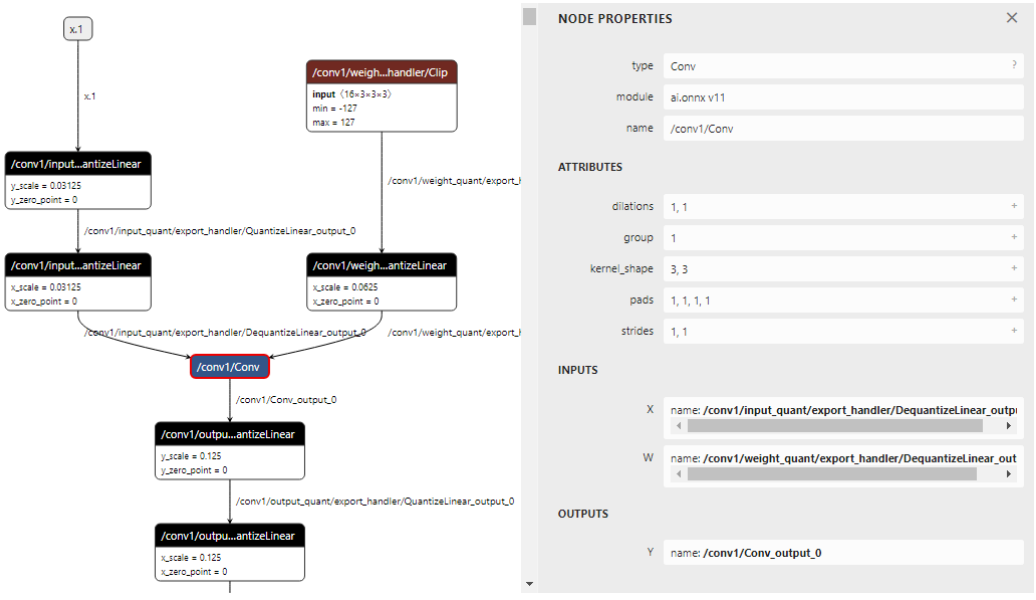


Fig. 4.1 Standard Node for Conv2d

Fig. 4.2 Custom Node for adder2d

4.3 QONNX

Quantized ONNX (QONNX) introduces three novel custom operators—Quant, BipolarQuant, and Trunc—to facilitate arbitrary-precision uniform quantization within the ONNX framework. These operators empower the representation of diverse quantization schemes, encompassing binary, ternary, 3-bit, 4-bit, 6-bit, or any other specified precision levels.

Quantization seamlessly integrates into the neural network model, applicable across parameters or layer inputs.

QONNX offers extensive flexibility in selecting scaling factors and zero-point granularity, essential for optimizing model performance and enhancing accuracy.

It is notable that quantized values adhere to standard floating-point datatypes, ensuring alignment with ONNX protobuf specifications. As a result, QONNX significantly enhances the capabilities of ONNX models, enabling efficient deployment on resource-constrained platforms while maintaining robust interoperability.

Upon availability of the ONNX model, the corresponding QONNX version can be exported using the following commands:

```
qonnx-cleanup ${ONNX_FILE}_bnfuse.onnx --out-file=${
    ONNX_FILE}_clean.onnx
qonnx-convert ${ONNX_FILE}_clean.onnx --output-style
    quant --output-file=${ONNX_FILE}_clean_quant.onnx
```

In the following pictures the major nodes of AddNN basic block are shown in their QONNX graph representation, visualized on Netron. It is also evident the aforementioned skip connection.

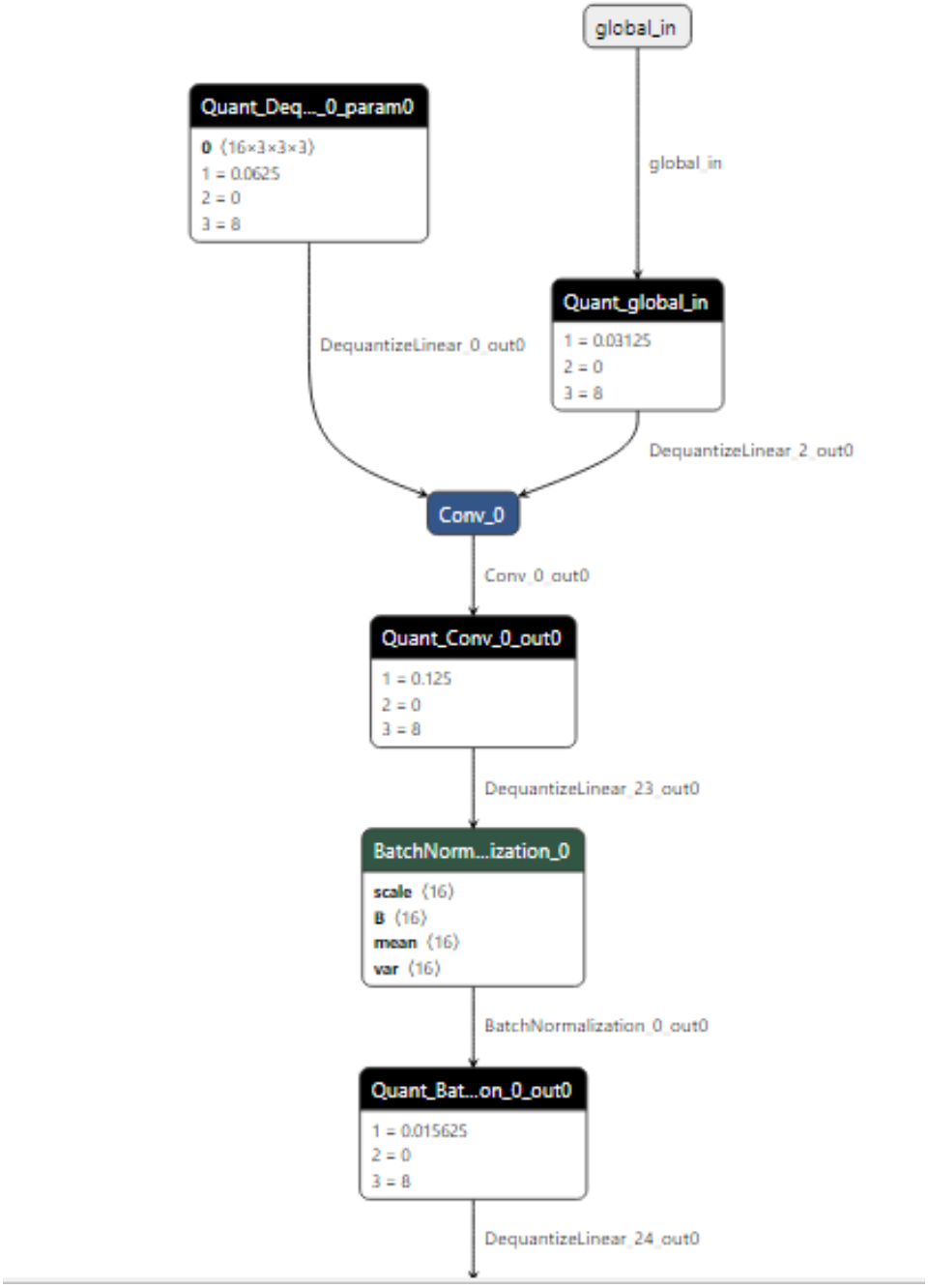


Fig. 4.3 Detail of the first convolutional layer $Conv_0$

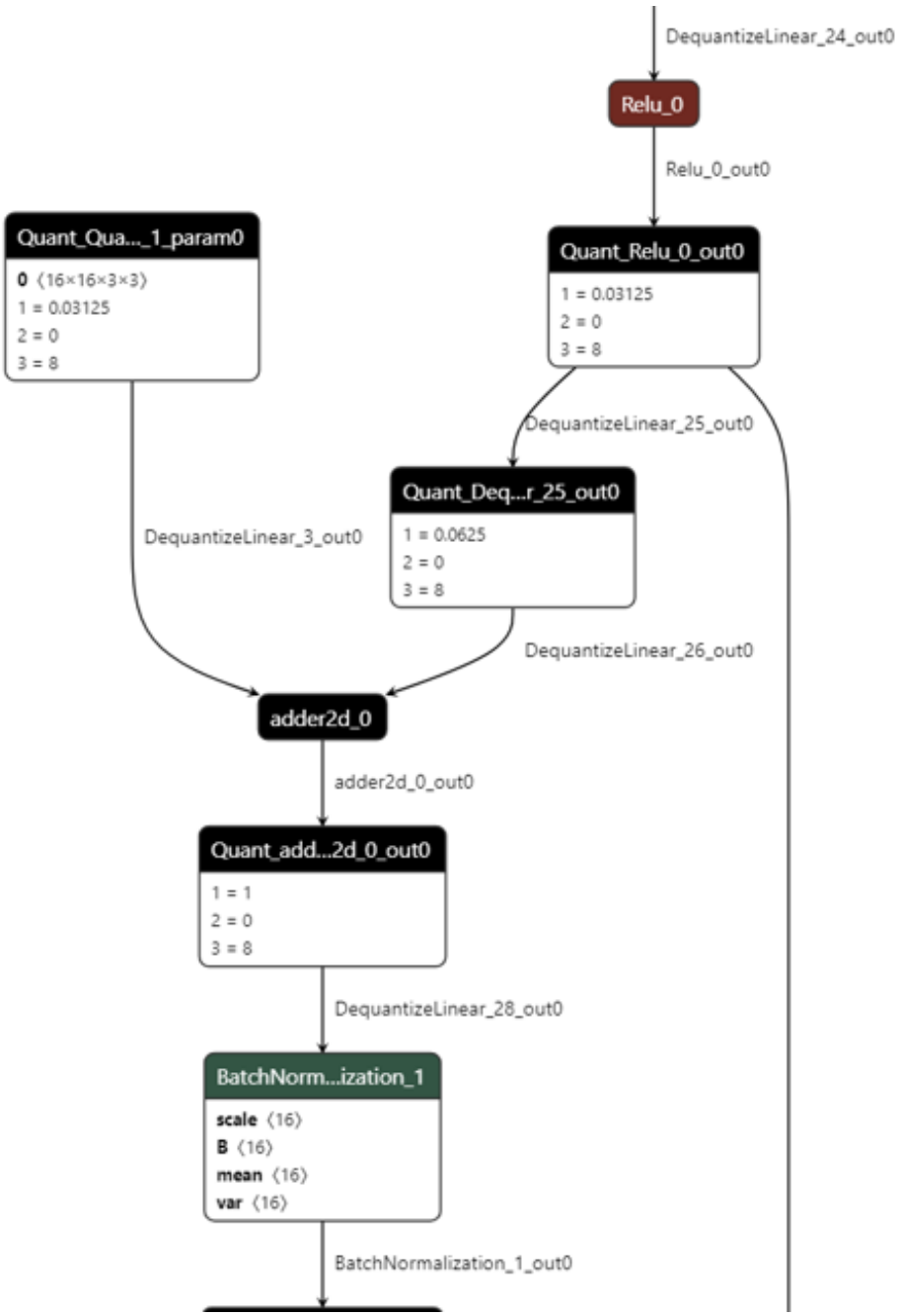


Fig. 4.4 Detail of the first *adder2d* layer

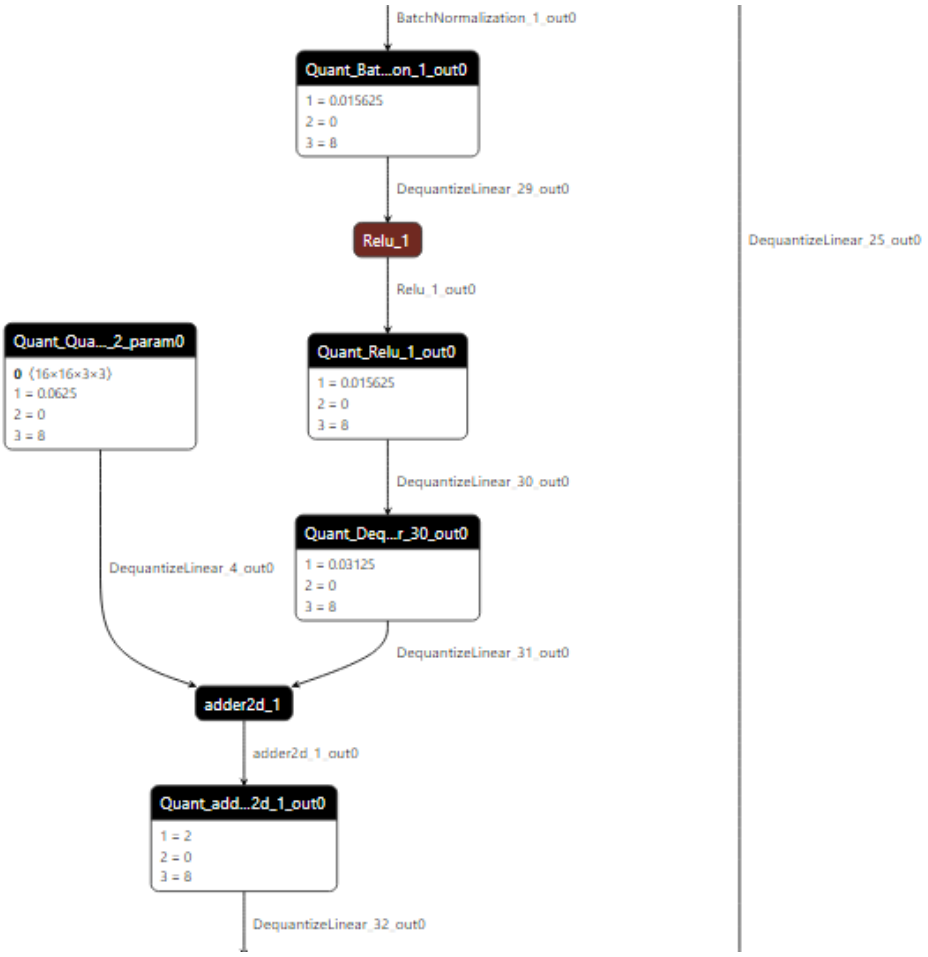


Fig. 4.5 Detail of the second *adder2d* layer

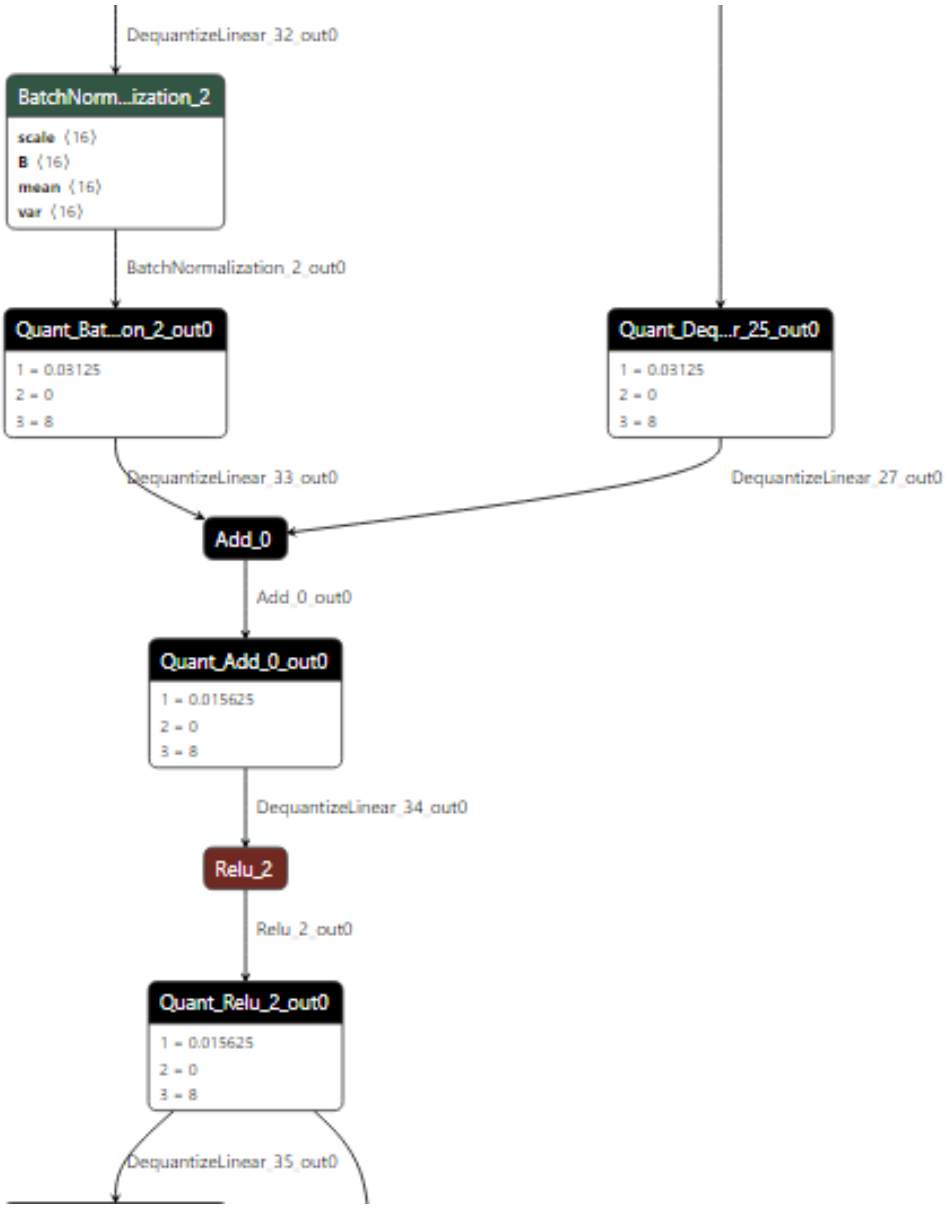


Fig. 4.6 Detail of the first *Add* operation node

Chapter 5

Integer Quantized AdderNet Implementation

5.1 NN2FPGA Framework

NN2FPGA is a framework developed within the Microelectronics Research Group at the Politecnico di Torino's Department of Electronics and Telecommunications. It is designed to generate quantized convolutional neural network accelerators in C++ for AMD FPGAs. The primary objective of this project is to provide a reliable tool that targets embedded FPGAs while maintaining state-of-the-art performance metrics. This framework supports ResNet-like models and includes optimizations for skip connections. Additionally, it ensures optimal resource allocation using the Binary Integer Programming (BIP) algorithm. A notable feature of the project is its commitment to open science, as it is fully open-source and released under the MIT license.

5.2 High-Level-Synthesis

High-Level Synthesis (HLS) is an Electronic Design Automation technique that aims to translate an algorithm description in a high-level software programming language, such as C and C++, into a HDL description. HLS allows for the design of

more complex systems in less time than HDL design, and facilitates the co-design of hardware and software, although it does so at the cost of limited low-level control.

5.3 Vitis HLS

Vitis HLS (High-Level Synthesis) [8] is a design tool developed by Xilinx that facilitates the conversion of high-level programming languages, such as C and C++, into hardware description language (HDL). This tool abstracts the complexities associated with HDL coding, enabling efficient implementation of hardware algorithms. Vitis HLS allows for the use of pragmas to guide the synthesis process, optimizing hardware design for performance, area, or power consumption. A critical feature of Vitis HLS is its support for streaming data between processes, essential for designing high-throughput, low-latency systems. Streams in Vitis HLS ensure efficient data transfer between different blocks of the hardware design, preventing bottlenecks and maintaining smooth data flow.

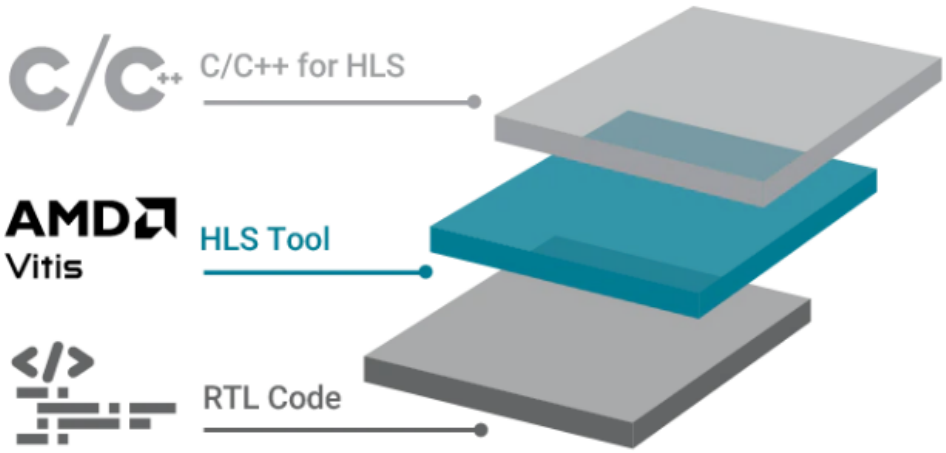


Fig. 5.1 Vitis HLS Diagram

5.3.1 Workflow

The typical HLS workflow is comprised of the following stages:

1. The implementation of the software (SW) is performed at the top-level entity, which is a C function. The function arguments are the entity ports, and the functionality is implemented in the SW. In order to guarantee synthesizability, certain constraints must be respected.

2. The verification of the software can be conducted by developing the testbench as a simple main function that calls the top-level entity function. Consequently, the functionality can be verified in a manner analogous to that of any SW, with the potential to utilise traditional tools, for instance debuggers and print statements.

3. Hardware synthesis: the synthesizer generates a register-transfer level (RTL) description of the top-level entity. It is possible to generate different architectures by setting up some parameters through dedicated directives.

4. Hardware verification: the RTL description is simulated to ensure that the software and hardware outputs are in alignment.

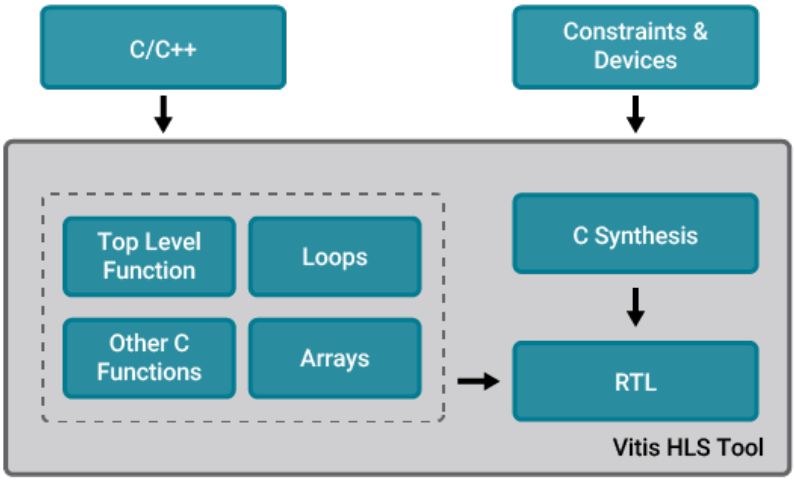


Fig. 5.2 Vitis HLS - Project Composition

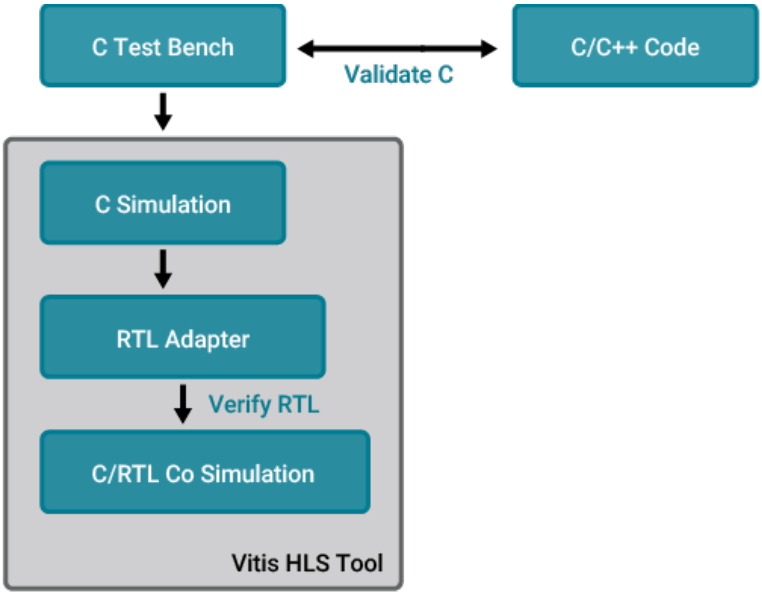


Fig. 5.3 Vitis HLS - Workflow

5.4 Framework and Reproduced CNN ResNet20

The framework used for reproducing the convolutional neural network ResNet20 is fundamental for this resarch. ResNet20, a variant of the Residual Network, is designed for deep learning tasks, particularly image classification. Reproducing ResNet20 involves implementing its architecture, including convolutional layers, batch normalization, and residual connections, using a high-level framework such as PyTorch. Accurate reproduction of ResNet20 is essential to ensure that subsequent modifications and experiments are based on a reliable and well-understood model. This framework allows for easy manipulation and extension of the network, facilitating the integration of novel components and exploration of new ideas, such as the transition to AdderNet.

5.5 AdderNet Accelerator

This thesis contributes to the adaptation of the convolutional ResNet20 generated by the framework to implement the proposed quantized AdderNet. The primary steps, starting from a CNN ResNet20 generated by the aforementioned Framework, involve

deactivating the ReLU function, removing intermediate Add operators, and inserting a newly defined batch normalization function after each convolutional layer through appropriate modifications. These changes are necessary because BatchNorm2d layers cannot be merged with the adder2d ones, unlike in traditional CNNs, and consequently, neither can ReLU layers. The three main types of transformations performed are as follows:

- **Deactivation of ReLU Functions:** The ReLU activation functions are deactivated in the convolutional nodes in order to remove their influence on the overall network.
- **Removal of Intermediate Add Layers:** The provided intermediate *Add* nodes have been removed by deactivating the proper parameters in the convolutional nodes, to fit the final AdderNet architecture for which a tailored *Add* function has been implemented.
- **Insertion of Batch Normalization:** A custom Batch Normalization function is introduced after each convolutional layer to ensure network stability and performance. Two versions of the normalization layer have been developed: the first, *realbn*, is designed for standard cases where the scale factor applied to the data (after a convolutional layer) is less than or equal to 1.
  This often occurs when a DeQuant node is applied after the adder2d layer to convert fixed-point values back to floating-point for further processing. In this scenario, the scaling is minimal, so the regular batch normalization operation can be applied directly; the latter, *realbn_scaled*, is tailored for situations where the scaling factor exceeds 1. This can happen due to the limitations of fixed-point arithmetic in hardware. Vitis HLS is not able to handle such scenarios efficiently, so this function includes additional steps to handle fixed-point scaling before applying batch normalization.

  The implementation of the new functions for the three explicit nodes is detailed in the codes below.

```
template<typename T>
T relu(T& input) {
    T output = input;
    output = (input > 0) ? input : typename std::
        remove_reference<decltype(input)>::type(0);
```

```
    return output;
}

template<typename data_in_t, typename data_out_t, int CH
    , int H, int W, int ch_step, int w_step>
void apply_relu(hls::stream<data_in_t> dinStream[w_step
    ], hls::stream<data_out_t> doutStream[w_step])
{
    for (auto h = 0; h < H; h++) {
        for (auto w = 0; w < W; w += w_step) {
            for (auto ch = 0; ch < CH; ch += ch_step) {
                for (auto ow = 0; ow < w_step; ow++) {
                    data_in_t data = dinStream[ow].read
                        ();
                    data_out_t relu_result;
                    for (auto op = 0; op < ch_step; op
                        ++) {
                        relu_result.data[0][op] = relu(
                            data.data[0][op]);
                    }
                    relu_result.last = data.last;
                    doutStream[ow].write(relu_result);
                }
            }
        }
    }
}
```

Listing 5.1 ReLU - Code details

```
template<typename data_t, typename data_tt, typename
    data_out_t, int CH, int H, int W, int ch_step, int
    w_step>
void Add(hls::stream<data_t> din1Stream[w_step],
         hls::stream<data_tt> din2Stream[w_step],
         hls::stream<data_out_t> doutStream[w_step])
{  for (auto h = 0; h < H; h++) {
        for (auto w = 0; w < W; w += w_step) {
```

```
            for (auto ch = 0; ch < CH; ch += ch_step) {
                for (auto ow = 0; ow < w_step; ow++) {
                    data_t data1 = din1Stream[ow].read()
                        ;
                    data_tt data2 = din2Stream[ow].read
                        ();
                    data_out_t result;
                    for(auto op = 0; op < ch_step; op++)
                        {
                        result.data[0][op] = data1.data
                            [0][op] + data2.data[0][op];
                    }
                    result.last = data1.last;
                    doutStream[ow].write(result);
                }
            }
        }
    }
}
```

Listing 5.2 Add - Code details

```
template <typename data_t_in, typename data_t_out, int
    layer, int CH, int H, int W, int ch_step, int w_step>
void realbn(hls::stream<data_t_in> dinStream[w_step],
            hls::stream<data_t_out> doutStream[w_step],
            const float scale[CH],
            const float bias[CH],
            const float mean[CH],
            const float variance[CH],
            const float epsilon)
{
    for (int h = 0; h < H; h++)
    {
        for (int w = 0; w < W; w += w_step)
        {
            for (int ch = 0; ch < CH; ch += ch_step)
            {
```

```
16 #pragma HLS unroll
17                 for (int ow = 0; ow < w_step; ow++)
18                 {
19                     data_t_in data = dinStream[ow].read
                           ();
20                     data_t_out scaled;
21                     scaled.last = data.last;
22                     for (auto op = 0; op < ch_step; op
                           ++)
23                     {
24                         float normalized_data = (data.
                               data[0][op].to_float() - mean
                               [op + ch]) / std::sqrt(
                               variance[op + ch] + epsilon);
25                         scaled.data[0][op] = scale[ch +
                               op] * normalized_data + bias[
                               ch + op];
26                     }
27                     doutStream[ow].write(scaled);
28                 }
29             }
30         }
31     }
32 }
33
34 template <typename data_t_in, typename data_t_out, int
       layer, int bit_scaling, int CH, int H, int W, int
       ch_step, int w_step>
35 void realbn_scaled(hls::stream<data_t_in> dinStream[
       w_step],
36             hls::stream<data_t_out> doutStream[w_step],
37             const float scale[CH],
38             const float bias[CH],
39             const float mean[CH],
40             const float variance[CH],
41             const float epsilon)
42 {
43     for (int h = 0; h < H; h++)
```

```
44      {
45          for (int w = 0; w < W; w += w_step)
46          {
47              for (int ch = 0; ch < CH; ch += ch_step)
48              {
49  #pragma HLS unroll
50                  for (int ow = 0; ow < w_step; ow++)
51                  {
52                      data_t_in data = dinStream[ow].read
                            ();
53                      data_t_out scaled;
54                      scaled.last = data.last;
55                      for (auto op = 0; op < ch_step; op
                            ++)
56                      {
57                          ap_fixed<8, 8, AP_RND_CONV,
                                AP_SAT> unscaled_data = data.
                                data[0][op] / (1 <<
                                bit_scaling);
58                          ap_int<8 + bit_scaling>
                                scaled_data = unscaled_data *
                                 (1 << bit_scaling);
59                          float normalized_data = (
                                scaled_data.to_float() - mean
                                [op + ch]) / std::sqrt(
                                variance[op + ch] + epsilon);
60                          scaled.data[0][op] = scale[ch +
                                op] * normalized_data + bias[
                                ch + op];
61                      }
62                      doutStream[ow].write(scaled);
63                  }
64              }
65          }
66      }
67  }
```

Listing 5.3 BN - Code details

The provided detail on batch normalization function demonstrates that all statistical parameters required by the layer's computations—mean, variance, bias, epsilon, and scale factor—are directly supplied as inputs to the function. These parameters are meticulously extracted from the original QONNX model and appropriately utilized in the defined function.

To convert the current network into the final AdderNet, two essential steps are required. First, it is necessary to substitute the existing convolutional layers with the new custom adder2d layers. These layers are key to the AdderNet architecture, which uses SAD operations instead of traditional MAC ones to learn features in the network. Next, the input and output parameter types must be adapted to match those of the Quant nodes, as clearly defined in the final QONNX graph. This ensures compatibility across layers and maintains the precision required during the quantization process, allowing the network to function efficiently as an AdderNet.

Following preliminary tests to verify the correctness of these procedures, the contribution involved introducing new data streams for quantization nodes between each pair of layers. The type of each stream was derived from the values present in the final QONNX graph, along with the dimensions of the adder2d accumulators. These accumulators were specifically adjusted, transitioning from their previous configuration tailored to MAC operations in CNNs, to now supporting SAD operations typical of AddNN.

5.6 DSP Packing

Digital Signal Processing (DSP) packing with High-Level Synthesis involves a critical optimization technique aimed at enhancing the efficiency of FPGA-based DSP applications. HLS tools like Vitis HLS enable developers to convert high-level DSP algorithms, typically written in languages such as C or C++, into optimized hardware implementations. DSP packing specifically focuses on maximizing the utilization of FPGA resources, such as DSP blocks, by efficiently mapping algorithmic operations onto these specialized hardware units. By packing multiple DSP operations into a single DSP block where possible, HLS tools can reduce resource consumption, improve performance, and minimize power consumption, making FPGA-based DSP implementations more cost-effective and scalable for a wide range of signal processing applications. This technique is essential for achieving high-throughput, low-latency

processing capabilities required in modern digital communication systems, image and video processing, radar systems, and many other DSP-intensive domains.

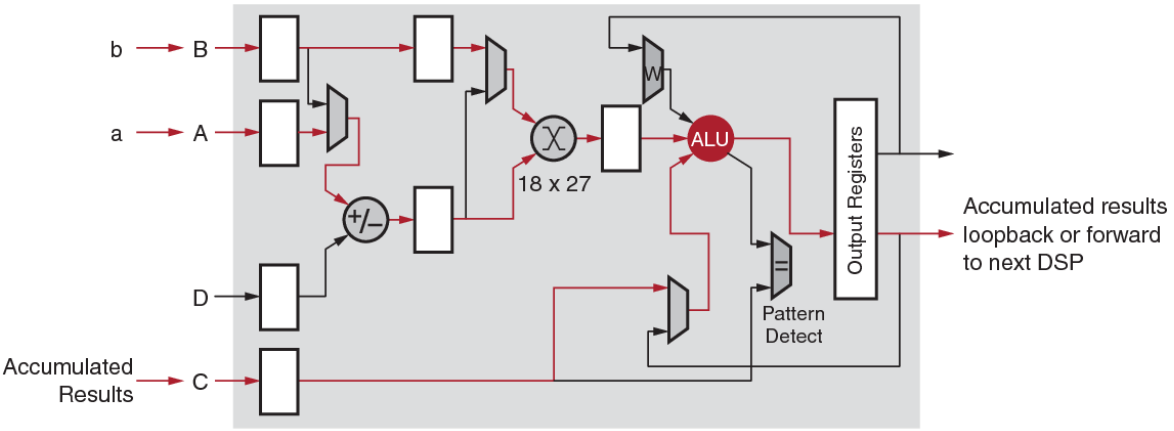


Fig. 5.4 FPGA's DSP slice.

5.6.1 Adopted Solution

To fully leverage all resources on FPGA, various DSP packing techniques have been explored, ranging from fundamental INT8 Optimization for CNN available for Xilinx Devices [9]-[10] to the latest methods tailored for AddNN [11]. In the context of the proposed quantized AdderNet, the main aim is a first optimization of DSP resources to enable concurrent SAD operations.

DSP48E2 slices in Xilinx FPGAs offer native DSP packing capabilities for addition operations [12]. In Single Instruction Multiple Data (SIMD) mode, a DSP slice can support up to Quad-INT12 adder/subtractor/accumulator with four separate CARRYOUT signals. Given that the proposed AdderNet is quantized to 8-bit, only the LSB portion of the INT12 adder has been utilized.
DSP packing with Quad INT12 represents an advanced optimization strategy within High-Level Synthesis (HLS) frameworks like Vitis HLS.

This technique focuses on efficiently utilizing FPGA resources, particularly DSP blocks, by packing four INT12 operations into a single DSP block. By leveraging the parallel processing capabilities inherent in FPGA architectures, Quad INT12 packing significantly enhances the throughput and efficiency of DSP applications. This approach not only improves performance but also conserves FPGA resources and reduces power consumption, making it highly suitable for demanding signal

processing tasks such as deep learning inference, image and video processing, and telecommunications.
The integration of Quad INT12 packing in HLS would facilitate the implementation of high-performance, low-latency DSP systems on FPGA platforms, meeting the stringent requirements of modern applications. The following code has been written in order to implement the Quad INT12 packing strategy.

```
(* use_dsp = "yes" *) module dsp_quadsimd (
    input logic CLK,
    input logic RSTP,
    input logic [29:0] A,
    input logic [17:0] B,
    input logic [17:0] C,
    input logic [26:0] D,
    output logic [47:0] P

);

    logic [26:0] X1, X2;
    logic [26:0] W1, W2;
    logic [26:0] double1, double2;
    logic [11:0] sum1, sum2, sum3, sum4;


    always_ff @(posedge CLK) begin

        if (RSTP) begin

            X1 <= 27'b0;
            W1 <= 27'b0;
            X2 <= 27'b0;
            W2 <= 27'b0;
            double1 <= 27'b0;
            double2 <= 27'b0;
            sum1 <= 12b'0;
```

```
29              sum2 <= 12b'0;
30              sum3 <= 12b'0;
31              sum4 <= 12b'0;
32
33              P <= 48'b0;
34
35          end else begin
36
37              X1 <= (A[11:0] << 14) + B[11:0];
38              W1 <= (C[11:0] << 14) + C[11:0];
39              double1 <= X1 - W1;
40
41              X2 <= (A[11:0] << 14) + B[11:0];
42              W2 <= (D[11:0] << 14) + D[11:0];
43              double2 <= X2 - W2;
44
45              sum1 <= double1[26:14];
46              sum2 <= double1[11:0];
47              sum3 <= double2[26:14];
48              sum4 <= double2[11:0];
49
50              P <= {sum1, sum2, sum3, sum4};
51
52          end
53      end
54
55  endmodule
```

Listing 5.4 RTL code for Quad-INT12 DSP packing

5.6.2 Future Work

Future work will mainly focus on the integration of the aforementioned DSP packing approach. A consequent implementation strategy would consist in an aggressive DSP-LUT co-packing to enable Octo-INT8 SIMD operation with one DSP48E2 slice and 16 LUTs on FPGA.

To achieve this desired configuration, the INT12 adder has been further divided into an 8-bit adder and a 3-bit adder, with one guard bit between them to ensure accurate addition of both components [13]. The upper 8-bit adder will utilize the CARRYOUT of the INT12 adder, while the lower 8-bit adder will be combined with a 5-bit adder implemented using LUTs.

As Vitis HLS lacks native support for packing Quad-INT12 and Octo-INT8 operations onto a single DSP48E2 slice, a black-box implementation strategy would be adopted. In that approach, a structural System Verilog RTL representation of the packed unit would be developed to serve as a well-defined processing element interface for Octo operations in Vitis HLS.

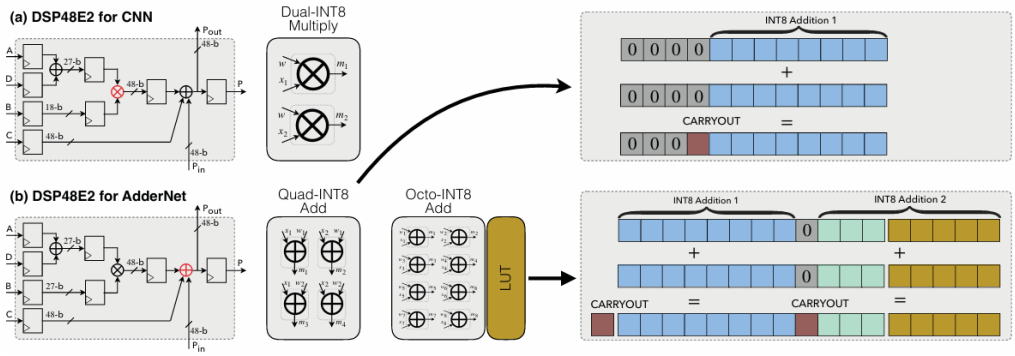


Fig. 5.5 DSP packing for(a) Dual-INT8 CNN,and DSP packing and DSP-LUT co-packing for(b) Quad-INT8 and Octo-INT8 AdderNet designs

Chapter 6

Evaluation and Conclusions

6.1 Simulation and Analysis

The simulation and analysis phase is fundamental for validating the modifications and estimating the adapted network performance. During this phase, various metrics such as accuracy, latency, and resource utilization are measured and compared against the original ResNet20 model. Simulation involves running the modified network on a dataset to observe its behavior and performance. The analysis provides insights into the effectiveness of the changes, highlighting any improvements or drawbacks. This process, beside a first network simulation on the CIFAR-10 dataset and final QONNX inferencing, comprises both synthesis and co-simulation. The results are presented and commented in the following paragraphs.

6.2 Simulation Environment

The project was developed using *Vitis™ HLS* 2023.2. Upon adding the *include*, *source*, and *testbench* files to the workspace, a C simulation was executed to verify the expected behavior of the input network.

Next, synthesis was initiated to review the key results of the synthesized architecture by consulting the generated reports. Following this, a C/RTL co-simulation was performed, allowing the network to be simulated using the HDL-generated model. At this stage, it is possible to choose between performing the verification

with Verilog or VHDL. These figures are approximate, as they are derived from C/RTL Co-Simulation and estimations made during C Synthesis; however, they remain valuable for recognizing the main trends of the final architecture.

6.3 Experimental Results

The data that has been gathered throughout the overall procedure is primarily focused on the following aspects:

- Performance: assessed based on metrics such as throughput and latency. Throughput refers to the number of frames processed per second (FPS), while latency measures the time required for the network to process a single input from start to finish. These metrics provide insights into how efficiently the network operates in real-time scenarios, which is crucial for applications requiring low-latency responses, such as image recognition or edge computing.
- Resource utilization: measured by the quantity of utilized Block RAMs (BRAMs), Ultra RAM (URAMs), Digital Signal Processors (DSPs), Look-up Tables (LUTs), and Flip-Flops (FFs). These resources indicate how much of the FPGA's available hardware is consumed by the network. Efficient resource utilization is critical for optimizing the hardware design, as it ensures that the network can operate within the constraints of the FPGA while maximizing performance. Higher resource usage, particularly DSPs and BRAMs, can indicate computational bottlenecks or areas for optimization.

Metric	Value	Units
Frequency	263.16	MHz
Throughput	8028.25	FPS
Latency	0.125	ms

Table 6.1 Post-Synthesis Performance for AdderNet ResNet-20

It is important to highlight that the final network's performance is constrained by the presence, not yet overcome, of unquantized Batch Normalization layers which represent the actual bottleneck. The principal consequence of this design decision is a considerable increase in the number of DSP slices required, due to the necessity of

Resource	Usage
DSP	682
LUT	1,263,467
FF	648,373
BRAM	116
URAM	12

Table 6.2 Post-Synthesis Resource Usage for AdderNet ResNet-20

computing all the BN's multiplications in 32-bit floating point, which represents a significant cost in terms of higher precision. This cost cannot be currently reduced for the reasons previously outlined in 3.2.5.

6.4 Conclusions

In conclusion, while the presented quantized AdderNet ResNet-20 design successfully implements the neural network model on FPGA hardware, there is still significant room for optimization. The architecture has demonstrated its potential for resource efficiency and performance improvement, but the results obtained have not fully met initial expectations. This highlights the need for further refinement to achieve optimal performance.

Going forward, the primary focus will be on optimising the performance of the current design, particularly through the integration of the already implemented DSP packing strategies. As outlined in the literature [14], these methods can significantly reduce resource overhead and improve efficiency, allowing the architecture to operate closer to its full potential by making better use of FPGA resources.

In addition, the refinement of quantization strategies will be a crucial area for future development. This will involve exploring more advanced quantization techniques and fine-tuning the network to improve results in both loss and accuracy metrics while maintaining efficiency.

Another key priority will be the optimization of the C++ code that describes the network, in particular the resource allocations and dependencies between layers, such as the streams used for quantization nodes.

In summary, while the AdderNet architecture shows promising results, significant improvements are both necessary and possible. The future work outlined aims to unlock the full potential of this architecture for use in resource-constrained environments.

References

[1] Hanting Chen, Yunhe Wang, Chunjing Xu, Boxin Shi, Chao Xu, Qi Tian, and Chang Xu. Addernet: Do we really need multiplications in deep learning? December 2019.

[2] Addernet and its minimalist hardware design for energy-efficient artificial intelligence.

[3] Xilinx/brevitas. First Release: January 2021.

[4] Alessandro Pappalardo. Xilinx/brevitas.

[5] A novel fpga-based convolution accelerator for addernet. 2021.

[6] https://github.com/lutzroeder/netron.git.

[7] https://github.com/nvidia/tensorrt/tree/master/tools/onnx-graphsurgeon.

[8] https://docs.amd.com/r/en-us/ug1399-vitis-hls/introduction.

[9] Convolutional neural network with int4 optimization on xilinx devices. June 24, 2020.

[10] Deep learning with int8 optimization on xilinx devices. April 24, 2017.

[11] Dsp-packing: Squeezing low-precision arithmetic into fpga dsp blocks. March 2022.

[12] Ultrascale architecture dsp slice user guide. 2021.

[13] Dsp-packing: Squeezing low-precision arithmetic into fpga dsp blocks. 2022.

[14] Wsq-addernet: Efficient weight standardization based quantized addernet fpga accelerator design with high-density int8 dsp-lut co-packing optimization. December 2022.

Appendix A

ResNet

AddNN ResNet20

```
import adder
import torch.nn as nn

def conv3x3(in_planes, out_planes, stride=1):
    " 3x3 convolution with padding "
    return adder.adder2d(in_planes, out_planes,
       kernel_size=3, stride=stride, padding=1, bias=
       False)

class BasicBlock(nn.Module):
    expansion=1

    def __init__(self, inplanes, planes, stride=1,
       downsample=None):
        super(BasicBlock, self).__init__()
        self.conv1 = conv3x3(inplanes, planes, stride=
           stride)
        self.bn1 = nn.BatchNorm2d(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = nn.BatchNorm2d(planes)
        self.downsample = downsample
```

```
19          self.stride = stride
20
21      def forward(self, x):
22          residual = x
23
24          out = self.conv1(x)
25          out = self.bn1(out)
26          out = self.relu(out)
27
28          out = self.conv2(out)
29          out = self.bn2(out)
30
31          if self.downsample is not None:
32              residual = self.downsample(x)
33
34          out += residual
35          out = self.relu(out)
36
37          return out
38
39  class ResNet(nn.Module):
40
41      def __init__(self, block, layers, num_classes=10):
42          super(ResNet, self).__init__()
43          self.inplanes = 16
44          self.conv1 = nn.Conv2d(3, 16, kernel_size=3,
                stride=1, padding=1, bias=False)
45          self.bn1 = nn.BatchNorm2d(16)
46          self.relu = nn.ReLU(inplace=True)
47          self.layer1 = self._make_layer(block, 16, layers
                [0])
48          self.layer2 = self._make_layer(block, 32, layers
                [1], stride=2)
49          self.layer3 = self._make_layer(block, 64, layers
                [2], stride=2)
50          self.avgpool = nn.AvgPool2d(8, stride=1)
51          self.fc = nn.Conv2d(64 * block.expansion,
                num_classes, 1, bias=False)
```

```
52          self.bn2 = nn.BatchNorm2d(num_classes)
53
54          for m in self.modules():
55              if isinstance(m, nn.BatchNorm2d):
56                  m.weight.data.fill_(1)
57                  m.bias.data.zero_()
58
59      def _make_layer(self, block, planes, blocks, stride
            =1):
60          downsample = None
61          if stride != 1 or self.inplanes != planes *
               block.expansion:
62              downsample = nn.Sequential(
63                  adder.adder2d(self.inplanes, planes *
                       block.expansion, kernel_size=1,
                       stride=stride, bias=False),
64                  nn.BatchNorm2d(planes * block.expansion)
65              )
66
67          layers = []
68          layers.append(block(inplanes=self.inplanes,
               planes=planes, stride=stride, downsample=
               downsample))
69          self.inplanes = planes * block.expansion
70          for _ in range(1, blocks):
71              layers.append(block(inplanes=self.inplanes,
                   planes=planes))
72
73          return nn.Sequential(*layers)
74
75      def forward(self, x):
76          x = self.conv1(x)
77          x = self.bn1(x)
78          x = self.relu(x)
79
80          x = self.layer1(x)
81          x = self.layer2(x)
82          x = self.layer3(x)
```

```
83
84          x = self.avgpool(x)
85          x = self.fc(x)
86          x = self.bn2(x)
87
88          return x.view(x.size(0), -1)
89
90  def resnet20(**kwargs):
91      return ResNet(BasicBlock, [3, 3, 3], **kwargs)
```

Listing A.1 Python Code

Appendix B

adder2d

adder2d layer

```
import torch
import torch.nn as nn
import numpy as np
from torch.autograd import Function
import math

def adder2d_function(X, W, stride=1, padding=0):
    n_filters, d_filter, h_filter, w_filter = W.size()
    n_x, d_x, h_x, w_x = X.size()

    h_out = (h_x - h_filter + 2 * padding) / stride + 1
    w_out = (w_x - w_filter + 2 * padding) / stride + 1

    h_out, w_out = int(h_out), int(w_out)
    X_col = torch.nn.functional.unfold(X.view(1, -1, h_x
        , w_x), h_filter, dilation=1, padding=padding,
        stride=stride).view(n_x, -1, h_out*w_out)
    X_col = X_col.permute(1,2,0).contiguous().view(X_col
        .size(1),-1)
    W_col = W.view(n_filters, -1)

    out = adder.apply(W_col,X_col)
```

```
20
21      out = out.view(n_filters, h_out, w_out, n_x)
22      out = out.permute(3, 0, 1, 2).contiguous()
23
24      return out
25
26  class adder(Function):
27      @staticmethod
28      def forward(ctx, W_col, X_col):
29          ctx.save_for_backward(W_col,X_col)
30          output = -(W_col.unsqueeze(2)-X_col.unsqueeze(0)
                ).abs().sum(1)
31          return output
32
33      @staticmethod
34      def backward(ctx,grad_output):
35          W_col,X_col = ctx.saved_tensors
36          grad_W_col = ((X_col.unsqueeze(0)-W_col.
                unsqueeze(2))*grad_output.unsqueeze(1)).sum
                (2)
37          grad_W_col = grad_W_col/grad_W_col.norm(p=2).
                clamp(min=1e-12)*math.sqrt(W_col.size(1)*
                W_col.size(0))/5
38          grad_X_col = (-(X_col.unsqueeze(0)-W_col.
                unsqueeze(2)).clamp(-1,1)*grad_output.
                unsqueeze(1)).sum(0)
39
40          return grad_W_col, grad_X_col
41
42  class adder2d(nn.Module):
43
44      def __init__(self,input_channel,output_channel,
            kernel_size, stride=1, padding=0, bias = False):
45          super(adder2d, self).__init__()
46          self.stride = stride
47          self.padding = padding
48          self.input_channel = input_channel
49          self.output_channel = output_channel
```

```
50          self.kernel_size = kernel_size
51          self.adder = torch.nn.Parameter(nn.init.normal_(
               torch.randn(output_channel,input_channel,
               kernel_size,kernel_size)))
52          self.bias = bias
53          if bias:
54              self.b = torch.nn.Parameter(nn.init.uniform_
                   (torch.zeros(output_channel)))
55
56      def forward(self, x):
57          output = adder2d_function(x,self.adder, self.
               stride, self.padding)
58          if self.bias:
59              output += self.b.unsqueeze(0).unsqueeze(2).
                   unsqueeze(3)
60
61          return output
```

Listing B.1 Python Code

Appendix C

Dataset

CIFAR-10

The CIFAR-10 dataset, short for the Canadian Institute for Advanced Research, serves as a pivotal resource in the realm of machine learning and computer vision. Comprising 60000 32x32 color images distributed across 10 distinct classes, it stands as a cornerstone dataset in the field, facilitating diverse research endeavors. These classes encompass airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks, each category comprising 6,000 images.

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

For algorithms designed to discern objects within images, CIFAR-10 proves invaluable as a teaching tool. Its low-resolution nature enables swift experimentation with various recognition algorithms, facilitating rapid iteration and analysis of efficacy. Originating as a labeled subset of the 80 Million Tiny Images dataset from 2008, CIFAR-10 emerged following meticulous labeling efforts, often incentivized through student compensation. Convolutional neural networks, in their manifold configurations, consistently excel in recognizing objects within CIFAR-10 images, cementing their status as the preferred approach for tackling this dataset's challenges.

C.0.1 Features Structure

```
FeaturesDict({
    'id': Text(shape=(), dtype=string),
    'image': Image(shape=(32, 32, 3), dtype=uint8),
    'label': ClassLabel(shape=(), dtype=int64,
       num_classes=10),
})
```

Listing C.1 Characteristics