

An FPGA-based On-device Fine-tuning Accelerator for Edge CNN Adaptation

YUNRUI FAN AND MENGJING TANG

MASTER'S THESIS

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY

FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY



An FPGA-based On-device Fine-tuning Accelerator for Edge CNN Adaptation

Yunrui Fan
Mengjing Tang

Department of Electrical and Information Technology
Lund University

Supervisor:
Sergio Castillo Mohedano
Joachim Rodrigues

Examiner: Pietro Andreani

June 7, 2026



Printed in Sweden
E-huset, Lund, 2026

Abstract

With the widespread deployment of convolutional neural networks (CNNs) at the edge, on-device fine-tuning has become essential for real-time model adaptation to address environmental accuracy degradation. However, on-device training is constrained by the "memory wall" energy bottleneck and numerical instability from low-precision fixed-point arithmetic. To tackle these challenges, this thesis presents an energy-efficient, FPGA-based on-device fine-tuning accelerator featuring a deep hardware-algorithm co-design.

At the algorithmic level, a partial fine-tuning strategy is adopted to freeze shallow feature extractors while only updating deeper task-specific layers, reducing the activation-storage overhead of full backpropagation. Additionally, a hardware-aware quantization-aware training (HA-QAT) pipeline incorporates LFSR-based Stochastic Rounding and Stochastic Approximate Pruning (SAP), achieving over 70% backward gradient sparsity with negligible accuracy loss. At the architectural level, forward and backward computations are modeled as a unified seven-dimensional nested loop mapped onto a custom 2D Processing Element (PE) array. By employing a Row-Stationary (RS) dataflow and adaptive spatial unrolling, the design maximizes data reuse and minimizes memory access.

The system is prototyped and verified on the Xilinx Zynq UltraScale+ ZCU104 platform. Evaluations using a lightweight VGG-style network on a perturbed CIFAR-10 dataset show that the accelerator recovers model accuracy from a degraded 69.02% to 84.85% with stochastic pruning enabled (a mere 0.30% drop versus the unpruned baseline). Operating at 166.67 MHz, it delivers an effective throughput of 21.93 GOPS with a power consumption of only 2.085 W, yielding an energy efficiency of 10.52 GOPS/W. This work demonstrates the feasibility of practical CNN on-device fine-tuning on resource-constrained edge FPGAs, providing an efficient hardware-software co-design solution for local continuous learning in edge intelligence.

Keywords: Edge Computing; On-device Fine-tuning; FPGA; Convolutional Neural Network (CNN); Hardware-Algorithm Co-design; Dataflow Optimization

Popular Science Summary

Deep learning is a key technology behind applications such as image recognition, autonomous driving, and intelligent monitoring systems. Many of these systems are based on *Convolutional Neural Networks (CNNs)*, which are typically trained in the cloud and then deployed to edge devices.

Edge devices often operate in dynamic environments, which can reduce the accuracy of the model over time. Cloud-based retraining is impractical due to communication costs and privacy concerns. Enabling on-device training is therefore essential for real-time local adaptation.

On-device training is more demanding than inference because the energy required for data movement can dominate total power consumption, creating the “memory wall”. To address this problem, this project proposes an energy-efficient FPGA accelerator that utilizes an optimized training dataflow. The architecture adopts a Row-Stationary dataflow strategy to maximize local data reuse and minimize costly memory traffic.

Another major challenge in edge training is numerical stability. Hardware replaces floating-point computation with fixed-point arithmetic, a trade-off that can lead to gradient underflow and unstable convergence. To ensure reliable learning despite these precision limits, the project explores gradient-stabilization techniques to improve training stability under low-precision conditions.

In general, this project presents a practical approach for enabling neural network training on resource-limited hardware. The proposed accelerator addresses key challenges such as the memory wall and numerical instability, providing an energy-efficient and reliable solution for on-device adaptation.

List of Acronyms

AI:	Artificial Intelligence
ALU:	Arithmetic Logic Unit
ASIC:	Application-Specific Integrated Circuit
AXI:	Advanced eXtensible Interface
BN:	Batch Normalization
BRAM:	Block Random Access Memory
BW:	Backward Propagation
CNN:	Convolutional Neural Network
CPU:	Central Processing Unit
DDR:	Double Data Rate
DGC:	Deep Gradient Compression
DNN:	Deep Neural Network
DSP:	Digital Signal Processing
FC:	Fully Connected
FIFO:	First-In First-Out
FPGA:	Field-Programmable Gate Array
FP32:	32-bit Floating Point
FSBL:	First Stage Boot Loader
FSM:	Finite State Machine
FW:	Forward Propagation
GAP:	Global Average Pooling
GEMM:	General Matrix Multiply
GLB:	Global Buffer
GPU:	Graphics Processing Unit
GOPS:	Giga Operations Per Second
HA-QAT:	Hardware-Aware Quantization-Aware Training
IR:	Input Reuse
IS:	Input Stationary
LFSR:	Linear Feedback Shift Register
LUT:	Look-Up Table
MAC:	Multiply-and-Accumulate
MAE:	Mean Absolute Error

MLP:	Multilayer Perceptron
MPSoC:	Multiprocessor System-on-Chip
OFMAP:	Output Feature Map
OR:	Output Reuse
OS:	Output Stationary
PE:	Processing Element
PISO:	Parallel-In Serial-Out
PL:	Programmable Logic
PRNG:	Pseudo-Random Number Generator
PS:	Processing System
PTQ:	Post-Training Quantization
QAS:	Quantization-Aware Scaling
QAT:	Quantization-Aware Training
ReLU:	Rectified Linear Unit
RF:	Register File
RS:	Row-Stationary
RTL:	Register Transfer Level
SAP:	Stochastic Approximate Pruning
SGD:	Stochastic Gradient Descent
SoC:	System-on-Chip
SRAM:	Static Random Access Memory
STE:	Straight-Through Estimator
UART:	Universal Asynchronous Receiver/Transmitter
URAM:	UltraRAM
WG:	Weight Gradient
WR:	Weight Reuse
WS:	Weight Stationary

Table of Contents

1	Introduction	1
1.1	Thesis Scope	1
1.2	Related Work	2
1.3	Thesis Outline	3
2	Background	5
2.1	Fundamentals of Convolutional Neural Networks	5
2.1.1	Convolutional Layer	5
2.1.2	Normalization Layer	8
2.1.3	Activation Layer	10
2.1.4	Pooling Layer	11
2.1.5	Fully Connected Layer	13
2.1.6	Softmax Layer	14
2.2	Training versus Inference	15
2.2.1	Inference	16
2.2.2	Training	16
2.3	Spatial Architecture and Dataflow Mapping	19
2.3.1	Convolution as Nested Loops and Data Reuse Opportunities	19
2.3.2	The Memory Wall: From Temporal to Spatial Architectures	20
2.3.3	Loop Mapping: Tiling, Unrolling, and Interchange	22
2.3.4	Taxonomy of Dataflows: Input, Output, and Weight Stationary	23
2.3.5	Row-Stationary Dataflow for On-Device Training	25
3	Design Methodology	29
3.1	Task Modeling and Network Configuration	29
3.1.1	Edge Fine-Tuning Task	29
3.1.2	Target CNN Architecture	30
3.2	Unified Training Computation Model	31
3.2.1	Training Computation Primitives	32
3.2.2	Forward Softmax Approximation	33
3.2.3	Layer-wise Backpropagation Rules	35
3.2.4	Weight Update	38
3.2.5	End-to-End Computational Mapping	39

3.3	Hardware-Aware Quantization Strategy	40
3.3.1	Fixed-Point Precision Baseline	40
3.3.2	Hardware-Aware Quantization-Aware Training	41
3.3.3	Fixed-Point Arithmetic and Saturation Rules	42
3.3.4	Quantization and Sparsification in Training	44
3.4	Dataflow and Tensor Reuse Analysis	50
3.4.1	Loop Transformations	50
3.4.2	Tensor Reuse Formulation	52
3.4.3	Row-Stationary Dataflow for Training	54
3.5	Chapter Summary	56
4	Hardware Architecture Design	57
4.1	Overall Architecture	57
4.2	Controller Module	59
4.3	Global Buffer Module	60
4.3.1	On-Chip Storage Mapping	60
4.3.2	Auxiliary Training Buffers	61
4.3.3	Automated Bit-Width Derivation	62
4.4	Convolution Module	62
4.4.1	Router Cluster	62
4.4.2	Processing Element Cluster	65
4.4.3	Processing Element	67
4.4.4	PISO FIFO	68
4.5	Data Post-processing Module	70
4.6	Functional Operation Module	71
4.6.1	Max Pooling	71
4.6.2	Global Average Pooling	72
4.6.3	Fully-Connected Layer	73
4.6.4	SoftMax	73
4.6.5	Stochastic Rounding & LFSR	75
4.6.6	Stochastic Approximate Pruning	77
4.6.7	Parameter Update	79
5	System Integration and Evaluation	81
5.1	FPGA Platform Overview	81
5.2	Verification System Design and Workflow	82
5.3	Vivado Verification	83
5.4	Functional Verification	86
5.5	Performance Analysis	90
6	Conclusions and Future Works	93
6.1	Conclusions	93
6.2	Future Works	94
	References	95

List of Figures

2.1	Schematic diagram of a basic CNN architecture.	6
2.2	Numerical schematic of 2D convolution operation for $H = W = 5$, $R = S = 3$, $s = 2$, and $p = 1$	7
2.3	Schematic of multi-channel 2D convolution operation.	8
2.4	Computational pipeline of the BN layer.	9
2.5	Activation Functions	10
2.6	Numerical schematic of 2×2 max pooling and average pooling op- erations applied to a 4×4 ifmap with a stride of 2.	12
2.7	Schematic of a FC layer.	13
2.8	Computational pipeline of the Softmax layer.	14
2.9	Conceptual framework of training and inference phases.	16
2.10	Dynamic memory footprint and data access patterns.	17
2.11	Computational graph of the deep neural network training cycle, illus- trating the execution flow, contrasting the forward evaluation of the loss $\mathcal{L}$ with the backward derivation of parameter gradients $(\nabla w, \nabla b)$	18
2.12	Geometric mapping of the 7-nested loop indices over the spatial co- ordinates of ifmaps, weight kernels, and ofmaps.	20
2.13	Normalized energy cost of data movement across different levels of the memory hierarchy relative to a single MAC operation ($1\times$).	21
2.14	Comparison of parallel computing paradigms.	21
2.15	Hardware mapping of the 7-nested convolution loops via temporal tiling and spatial unrolling.	22
2.16	Three typical data reuse modes enabled by loop interchange for con- volutional hardware mapping.	24
2.17	Hardware mapping of three foundational 1D dataflow paradigms onto a 2D systolic array.	24
2.18	Execution of the RS dataflow on a 2D PE array.	26
2.19	Microarchitecture of a single PE within the RS spatial array.	27
2.20	Temporal multiplexing and accumulation strategies within a single PE for handling high-dimensional convolutions.	27
3.1	Algorithmic workflow of the hardware-aware partial fine-tuning strategy.	30
3.2	Cumulative signed error of gradient quantization over training steps.	45

3.3	Training and validation trajectories comparing hard quantization (QAS OFF, red) and dynamic scaling (QAS ON, blue).	46
3.4	Gradient distribution before and after applying SAP. The peaks in the pruned distribution correspond to the threshold values $\pm\tau$ of each convolution layer.	48
3.5	Training and validation accuracy trajectories comparing dense backpropagation (Pruning OFF, red) with highly sparse backpropagation (Pruning ON, blue).	49
3.6	Adaptive spatial and channel unrolling strategy driven by the Layer Controller. As the network goes deeper, the mapping smoothly transitions from pure spatial parallelism (4-way) to hybrid (2-way) and channel parallelism (1-way) to maintain maximum utilization of the 4 PE Clusters.	51
3.7	Halo dataflow mechanism and cross-tile dependency resolution. . . .	52
3.8	Compute-in-read index transformation for geometric duality.	52
3.9	Spatio-temporal mapping of the Row-Stationary dataflow onto a 2D logical array.	55
4.1	Accelerator overall architecture.	58
4.2	Top controller state transition diagram.	59
4.3	Overall Physical Topology.	63
4.4	Dynamic Routing Configurations	64
4.5	Data reading pattern for activation-gradient calculation in the sixth convolution layer.	65
4.6	Data reading pattern for weight-gradient calculation in the sixth convolution layer.	65
4.7	Structure of single PE cluster.	66
4.8	Control flow of the PE cluster controller.	66
4.9	Datapath of single Processing Element. The first three multiplier lanes are used for forward convolution and activation-gradient calculation, while the fourth lane is additionally enabled during weight-gradient calculation.	68
4.10	Structure of the PISO FIFO used at the output side of each PE cluster.	69
4.11	State transition diagram of the data post-processing module.	70
4.12	Partial-sum recombination in 2-way and 1-way spatial unrolling modes.	71
4.13	Streaming dataflow of the 2×2 max-pooling with a FIFO line buffer.	72
4.14	Forward and backward dataflow of global average pooling over a 4×4 feature map.	73
4.15	Accuracy evaluation of the fixed-point exponential approximation over $x \in [-10, 0]$	75
4.16	18-bit Fibonacci LFSR used as the pseudo-random source.	76
4.17	Layer-wise activation error comparison between RTL results and floating-point reference with and without stochastic rounding.	77
4.18	Activation-gradient distribution before and after RTL stochastic pruning.	78
4.19	Layer-wise activation-gradient distributions before and after RTL stochastic pruning.	79

4.20	Residual-based parameter update datapath.	80
5.1	Architecture of the FPGA prototype verification system.	82
5.2	Vivado block design.	84
5.3	Utilization report.	84
5.4	Timing report.	85
5.5	Post-implementation power report before enabling stochastic pruning.	86
5.6	Post-implementation power report after enabling stochastic pruning.	86
5.7	UART output example in inference mode before on-chip training.	87
5.8	UART output example during on-chip training.	88
5.9	Training and test loss comparison of RTL on-chip training with pruning disabled and enabled.	89
5.10	Training and test accuracy comparison of RTL on-chip training with pruning disabled and enabled.	89

List of Tables

2.1	Convolutional parameters	6
2.2	Loop Optimization Parameters for Hardware Mapping	22
3.1	Target CNN configuration with Partial Fine-tuning segmentation. . .	31
3.2	Mathematical Notations for the Unified Training Computation Model	38
3.3	Forward and Backward Dataflow for the Target Partial Fine-Tuning CNN	39
3.4	Software Baseline vs. Optimized Hardware Precision Allocation ($Q_{m,n}$)	41
3.5	Cumulative Signed Error of Gradient Quantization After 30 Epochs .	45
3.6	Performance and Overhead Comparison: QAS ON vs. OFF	47
3.7	QAS Dynamic Scaling Telemetry (Averaged Over Batches)	47
3.8	Gradient Distribution and STE Internal Statistics (1 sample, Target Sparsity=0.99)	49
3.9	SAP Pruning Accuracy and Edge Recovery Comparison	50
4.1	On-Chip Buffer Configurations and Address Quantization Summary	61
5.1	AXI4-Lite register map of the PL control interface.	83
5.2	Peak test accuracy comparison between floating-point training and RTL on-board training	89
5.3	MAC count of the CNN training workload.	90
5.4	Effective throughput of the CNN training workload.	91
5.5	Comparison with FPGA-based CNN training accelerators.	92

Introduction

1.1 Thesis Scope

This thesis investigates the hardware–algorithm co-design of a resource-efficient on-device training accelerator targeting edge FPGA platforms. The primary objective is to enable deployed convolutional neural networks (CNNs) to recover from accuracy degradation induced by post-deployment environmental domain shifts. To provide a controlled and representative evaluation framework, the proposed methodology is developed and validated based on the CIFAR-10 image classification dataset and a lightweight VGG-style CNN architecture. Given the severe constraints on memory capacity and computational resources in embedded edge devices, the scope of this work is intentionally restricted to a partial fine-tuning setting, in which shallow feature extractors remain frozen while only the deeper task-specific layers participate in on-device adaptation and parameter updates.

From an algorithmic perspective, this thesis proposes a hardware-aware Quantization-Aware Training (HA-QAT) pipeline. By enforcing strict fixed-point constraints during forward propagation and error backpropagation, and utilizing continuous parameters with clamping for weight updates, it establishes a closed-loop algorithmic simulation of the underlying hardware dataflow, thereby improving numerical consistency between software-level training and hardware execution. Furthermore, the framework incorporates hardware-friendly numerical refinements, including Linear Feedback Shift Register (LFSR)-based stochastic rounding and Stochastic Approximate Pruning (SAP), to mitigate quantization bias and exploit gradient sparsity during backward propagation without introducing expensive sorting or control logic.

From an architectural perspective, this research abstracts the disparate training procedure into a unified 7-dimensional nested-loop computation model. This formulation is then mapped onto a custom 2D processing element (PE) array employing a Row-Stationary (RS) dataflow, designed to maximize spatio-temporal data reuse and minimize costly off-chip memory accesses. The physical implementation scope of this research is further bounded to system-level prototype validation on a Xilinx Zynq UltraScale+ ZCU104 MPSoC platform. A complete

hardware-software co-design is realized through the interaction between the Processing System (PS) and Programmable Logic (PL), enabling evaluation in terms of hardware resource utilization, power consumption, throughput, and accuracy recovery capability.

1.2 Related Work

Edge Inference and Model Compression

Deploying deep neural networks (DNNs) on edge devices relies on lightweight architectures [1–3] and compression techniques including pruning [4] and quantization [5]. These pure inference methods assume the deployment data matches the training distribution. In practice, domain shifts caused by hardware degradation or environmental changes reduce model accuracy [6], which is a problem that unsupervised test-time adaptation [7] cannot fully resolve. Therefore, continuous on-device fine-tuning is required to restore performance.

Memory-Efficient Fine-Tuning

While edge-cloud collaborative frameworks [8] alleviate memory constraints by offloading computations, purely on-device training is essential to ensure data privacy and minimize communication latency. However, end-to-end local training imposes a substantial memory footprint, dominated by intermediate activation caching rather than model parameters [9, 10].

Existing methods exhibit inherent trade-offs between memory bottleneck with edge hardware constraints. Linear probing [11] is memory-efficient but lacks the representational capacity for severe domain shifts. Gradient checkpointing [12] reduces peak memory but imposes impractical recomputation overhead for energy-constrained devices. Similarly, parameter-efficient fine-tuning methods [13–15] reduce trainable weights, yet still require caching high-resolution feature maps from shallow layers during backpropagation. To address this bottleneck, our work adopts a partial fine-tuning strategy by freezing the initial network layers [16, 17], which avoids the memory-intensive caching of high-resolution feature maps.

Low-Precision and Quantized Training

Conventional Quantization-Aware Training (QAT) [18, 19] targets the forward pass, but high-precision backward passes exceed the resource capacities of edge FPGAs. Our QAT paradigm applies simulated fixed-point truncation to both forward activations and backward error gradients. Maintaining FP32 for SGD parameter updates ensures convergence while modeling hardware bit-width constraints.

Fixed-point truncation of error gradients introduces underflow, mapping low-magnitude values to zero and causing training divergence. Stochastic rounding mitigates this by using probabilistic mapping to preserve the expected value of small gradients [20–22]. Our pipeline incorporates stochastic rounding to stabilize convergence under these simulated fixed-point constraints.

Sparse Training and Gradient Pruning

To alleviate the dense computational burden of backpropagation, extensive research has leveraged the inherent over-parameterization of DNNs through sparse training [23–25]. These works demonstrate that discarding low-magnitude weights or gradients during training maintains high structural sparsity without sacrificing representational capacity, ensuring stable convergence [26–28].

To accelerate the backward pass, prior works aggressively prune small gradients [29, 30]. However, their reliance on $\mathcal{O}(N \log N)$ gradient sorting is incompatible with the resource constraints of edge FPGAs. To enable efficient on-device sparse training, we introduce Stochastic Approximate Pruning (SAP). By utilizing a lightweight probabilistic mechanism to discard insignificant gradients, SAP avoids exact sorting and directly bypasses redundant on-chip MAC operations and memory accesses.

Spatial Architectures and Dataflow

Spatial architectures comprising 2D PE arrays address the energy and latency constraints of deep learning [31]. While hardware frameworks like ScaleDeep [32] support neural network learning, maximizing on-chip data reuse dictates overall energy efficiency. The Row-Stationary (RS) dataflow [33, 34] minimizes off-chip memory accesses via local tensor residency and spatial multicasting. Our work extends this RS paradigm to the entire fine-tuning cycle, proposing a unified mapping abstraction that accommodates the geometric and spatio-temporal requirements of both forward and backward passes.

1.3 Thesis Outline

The remainder of this thesis is organized as follows.

- **Chapter 2: Background** introduces the foundational concepts of CNNs, reviews the mathematical principles of backpropagation, and delves into spatial architecture and dataflow mapping, analyzing convolution as nested loops.
- **Chapter 3: Design Methodology** details the algorithmic framework. It formally defines the partial fine-tuning task, establishes the unified 7-dimensional training computation model, and introduces the HA-QAT pipeline alongside the stochastic rounding and SAP mechanisms. The chapter concludes by formulating the theoretical dataflow and tensor reuse limits.
- **Chapter 4: Hardware Architecture Design** translates the proposed methodology into an RTL microarchitecture. It details the design of the top-level controllers, dynamic routing clusters, and the configurable PE array. The chapter further describes specialized functional modules, including the hardware-approximated Softmax unit, LFSR generator, and residual-based parameter update logic.

- **Chapter 5: System Integration and Evaluation** presents the system-level integration of the accelerator on the ZCU104 development board. It describes the AXI-based communication between the PS and PL, and provides a comprehensive analysis of hardware resource utilization, timing performance, power distribution, and on-chip training performance metrics.
- **Chapter 6: Conclusion and Future Work** summarizes the main contributions of this thesis and discusses potential directions for future architectural optimization and algorithmic extension.

Background

2.1 Fundamentals of Convolutional Neural Networks

DNNs are multi-layered computational models designed to extract hierarchical representations from high-dimensional data, typically involving vast quantities of *multiply-and-accumulate* (MAC) operations and frequent memory accesses [31]. Structurally, these networks comprise interconnected layers of processing nodes, where each connection is parameterized by a trainable weight. The forward propagation of data through these dense weight matrices fundamentally dictates the arithmetic intensity and memory footprint of the model.

As a specialized architectural variant of DNNs, *Convolutional Neural Networks* (CNNs) serve as an effective and widely deployed model that powers various artificial intelligence (AI) applications, including image classification [35], natural language processing [36], and robotic systems [37]. In contrast to traditional fully-connected networks, CNNs are specifically designed to process grid-structured data, most notably two-dimensional (2D) images. By leveraging local receptive fields and weight sharing, CNNs inherently achieve translation invariance while significantly reducing model complexity and parameter overhead [38]. Macroscopically, a standard CNN operates as an end-to-end feature extraction and classification pipeline, obviating the reliance on manual feature engineering, following the structural organization depicted in Fig. 2.1. As input data propagates forward through the pipeline, low-level pixel features are hierarchically abstracted into high-level semantic representations [39]. In practice, this processing flow is constructed by sequentially arranging the following functional layers:

2.1.1 Convolutional Layer

The convolutional (Conv) layer is the fundamental building block of CNNs, which is explicitly designed to leverage the intrinsic spatial structure of grid-structured multi-dimensional data, such as 2D images. Functionally, it acts as an automated end-to-end feature extractor that eliminates manual feature engineering by learn-

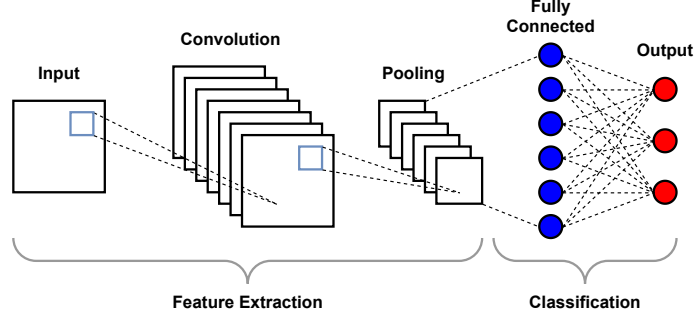


Figure 2.1: Schematic diagram of a basic CNN architecture.

ing hierarchical *feature maps* (fmaps) progressively, from low-level geometric primitives (*e.g.*, edges and textures) in shallow layers to complex semantic patterns in deep network layers [40]. Unlike affine transformations applied to flattened inputs in fully-connected layers, which treat all input dimensions identically and discard topological information, the discrete convolution operation in Conv layers functions as a linear transformation that preserves the inherent spatial order and local structural patterns of the data [41].

Table 2.1: Convolutional parameters

Shape Parameter	Description
N	Batch size of 3-D fmaps
M	# of 3-D filters / # of ofmap channels
C	# of ifmap/filter channels
H/W	ifmap plane height/width
R/S	filter plane height/width
E/F	ofmap plane height/width

For intuitive understanding, the convolution operation is defined by several geometric parameters summarized in Table 2.1. In a 2D context, given an *input feature map* (ifmap) $\mathbf{I}$ of spatial size $H \times W$ and a kernel $\mathbf{K}$ of spatial size $R \times S$, the kernel slides across the input grid to perform a series of localized computations. At each overlapping position (i, j) , the output value is derived from the sum of element-wise products between the kernel and the corresponding input pixels, with an additional constant bias term b . This process can be formally expressed

as:

$$(\mathbf{I} * \mathbf{K})_{ij} = \sum_{m=0}^{R-1} \sum_{n=0}^{S-1} K_{m,n} \cdot I_{i+m,j+n} + b,$$

where m and n are dummy indices for local summation over the spatial extent of the kernel, traversing its height R and width S , respectively; i and j denote the spatial position of the *output feature map* (ofmap), with valid ranges defined as $0 \leq i \leq H - R$ and $0 \leq j \leq W - S$ under unit stride and no padding.

In this 2D context, the output dimensions (E, F) are determined by the interaction between the input size (H, W) , kernel size (R, S) , stride (s) , and padding (p) . This mechanism ensures that each output unit only perceives a constrained local receptive field of the input, inherently inducing sparse connectivity. This spatial relationship is numerically demonstrated in Fig. 2.2, which illustrates a 3×3 kernel applied to a 5×5 input with 1×1 zero padding and 2×2 strides, providing a concrete visualization of the local summation logic and grid boundaries.

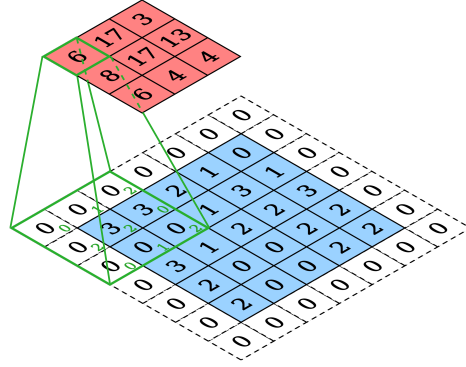


Figure 2.2: Numerical schematic of 2D convolution operation for $H = W = 5$, $R = S = 3$, $s = 2$, and $p = 1$.

While the 2D single-channel formulation provides an intuitive simplification, real-world visual tasks typically require multi-channel processing. In practice, input tensors consist of multiple fmaps, exemplified by the Red, Green, and Blue (RGB) channels of raw input images or the learned semantic feature channels in deeper network layers. This operation can be systematically extended to generate M (output channels) ofmaps from a batch of N ifmaps, as illustrated in Fig. 2.3. For multi-channel input tensors, each output channel is synthesized by applying an individual filter (a cuboid of C kernels) of size $R \times S \times C$, where C denotes the number of input channels. Equivalently, each ifmap is convolved with a dedicated 2D kernel, and the resulting partial spatial activations are accumulated element-wise along the channel dimension to synthesize a single unified output channel.

From a computational perspective, the Conv layer is the most resource-intensive component in standard CNNs. Driven by aggressive weight sharing and spatial data reuse, it exhibits exceptionally high operational intensity. Under the *Roofline* model framework [42], this property categorizes Conv layers as compute-bound,

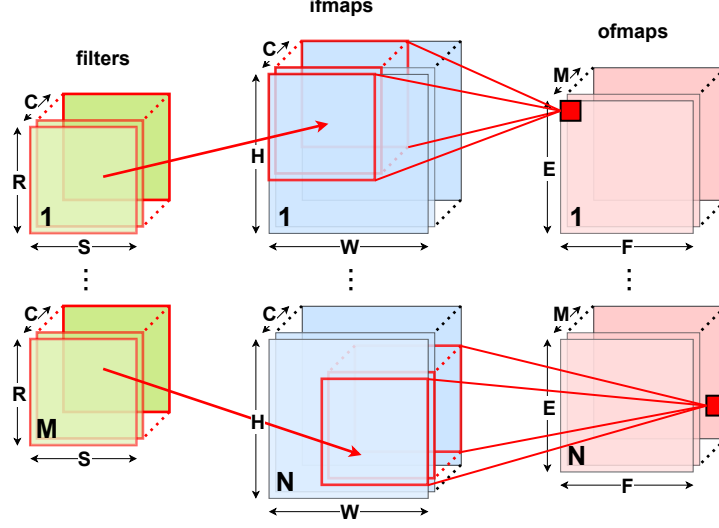


Figure 2.3: Schematic of multi-channel 2D convolution operation.
Redrawn from [31].

shifting the performance bottleneck from memory bandwidth to peak hardware throughput.

2.1.2 Normalization Layer

In deep neural networks, iterative updates to preceding layer parameters induce *Internal Covariate Shift* (ICS), perturbing the input distributions of downstream layers and impeding convergence. To mitigate this instability, *Batch Normalization* (BN) is extensively adopted after Conv layers to normalize intermediate activations. As specified in Algorithm 1, BN dynamically normalizes each mini-batch to zero mean and unit variance during training [43]. Crucially, to preserve the hierarchical features learned by the network, BN leverages channel-wise learnable affine transformations with a scale parameter γ and a shift parameter β . This adaptive mechanism allows the network to retain its representational capacity while capitalizing on the stability afforded by normalized activations.

A defining characteristic of BN is its operational asymmetry between the training and inference phases, as illustrated in Fig. 2.4. While training relies on stochastic mini-batch statistics to normalize features dynamically, inference requires strictly deterministic normalization. To ensure consistent predictions, global population statistics ($\mu_{\text{running}}, \sigma_{\text{running}}^2$) are accumulated through the *exponential moving average* (EMA) throughout training and frozen for deployment. This deterministic formulation enables the normalization transforms and affine parameters to be mathematically fused into a unified efficient linear transformation. Architecturally, by smoothing the non-convex optimization landscape, BN permits substantially

larger learning rates, mitigates sensitivity to weight initialization, and acts as a mild stochastic regularizer, thereby reducing reliance on conventional regularization techniques such as Dropout.

Algorithm 1 Forward propagation of BN during training.

Input: A mini-batch of spatial activations: $\mathcal{B} = \{x_1, \dots, x_m\}$;

Trainable affine parameters: γ, β

Output: Normalized and scaled activations: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\begin{aligned} \mu_{\mathcal{B}} &\leftarrow \frac{1}{m} \sum_{i=1}^m x_i && \triangleright \text{Compute mini-batch mean} \\ \sigma_{\mathcal{B}}^2 &\leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 && \triangleright \text{Compute mini-batch variance} \\ \hat{x}_i &\leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} && \triangleright \text{Normalize} \\ y_i &\leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) && \triangleright \text{Scale and shift} \end{aligned}$$

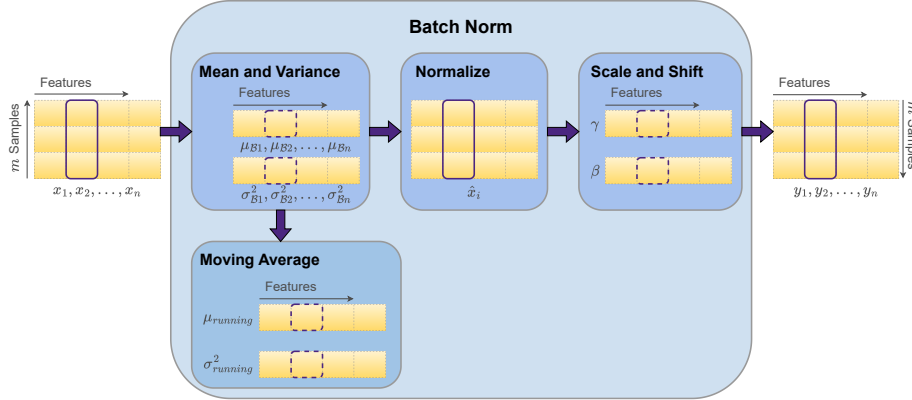


Figure 2.4: Computational pipeline of the BN layer. The diagram contrasts stochastic training normalization using mini-batch statistics ($\mu_{\mathcal{B}}, \sigma_{\mathcal{B}}^2$) with deterministic inference normalization using frozen running statistics ($\mu_{\text{running}}, \sigma_{\text{running}}^2$) and learned affine parameters (γ, β). Adapted from [44].

From a hardware implementation perspective, while BN is critical for training stability in algorithm design, its deterministic behavior in inference phase allows it to be mathematically folded into the weights and biases of the prior CONV layer. Such Conv-BN Fusion merges two sequential operations into one unified linear transformation, thereby eliminating the requirement for dedicated normalization hardware and significantly reducing the memory access overhead of intermediate

activations. Accordingly, the proposed hardware accelerator adopts a software preprocessing scheme for BN, instead of deploying independent physical hardware modules for normalization.

2.1.3 Activation Layer

The activation layer introduces nonlinearity into deep neural networks, enabling the modeling of complex real-world mappings. Without nonlinear activation, stacked linear layers degenerate into a single equivalent linear transformation, depriving deep models of their universal approximation capability [45]. Guided by this principle, a variety of representative activation functions have been developed to map the extracted spatial features into complex nonlinear manifolds.

Sigmoid and Tanh

Historically, continuous non-linear activations like the Sigmoid and Hyperbolic Tangent (Tanh) functions were widely used to map inputs to bounded intervals (as illustrated in Figs. 2.5(a) and (b)):

$$f(x) = \frac{1}{1 + e^{-x}},$$

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

However, they suffer from severe gradient vanishing in saturation regions [46, 47]. More critically for spatial hardware architectures, the exponential and division operations required by these functions impose a substantial arithmetic burden. Consequently, they are rarely adopted in the hidden layers of modern CNNs.

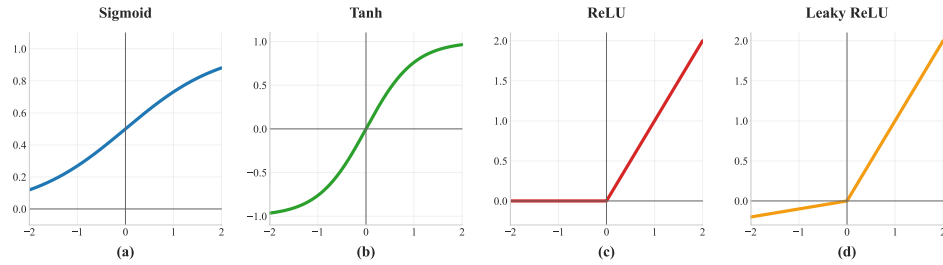


Figure 2.5: Activation Functions

ReLU

The Rectified Linear Unit (ReLU) has become the standard activation function due to its algorithmic and architectural efficiency. Defined mathematically as:

$$f(x) = \max(0, x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0. \end{cases}$$

ReLU exhibits an asymmetric piecewise linear response with a sharp threshold at the origin (Fig. 2.5(c)). By maintaining a constant unit gradient for positive inputs, it effectively mitigates the vanishing gradient problem [48], a phenomenon where error signals exponentially decay during backpropagation, preventing shallow layers from converging. From a hardware perspective, this piecewise formulation is highly optimal: it simplifies the nonlinear transformation into basic sign-bit verification, bypassing complex arithmetic logic. Despite its efficiency, the zero-truncation mechanism introduces the *dying ReLU* problem. Substantial negative gradient updates can force neurons into permanent inactivity, blocking gradient flow and ultimately limiting the model’s representational capacity.

Leaky ReLU and PReLU

To prevent permanent neuron deactivation, variants such as Leaky Rectified Linear Unit (Leaky ReLU) and Parametric Rectified Linear Unit (PReLU) introduce a small slope for negative inputs:

$$f(x) = \max(\alpha x, x) = \begin{cases} x, & x \geq 0 \\ \alpha x, & x < 0. \end{cases}$$

By guaranteeing a non-zero derivative across the entire input range (Fig. 2.5(d)), this design sustains continuous gradient flow during backpropagation [49]. In Leaky ReLU, α is a predefined static hyperparameter (e.g., 0.01). PReLU overcomes this static constraint by treating α as a learnable parameter, allowing the network to dynamically adapt its activation profile to specific data distributions [50]. Architecturally, PReLU maximizes representational capacity with minimal computational overhead during the forward pass, requiring only marginal additional on-chip memory to store the learned, channel-wise coefficients.

2.1.4 Pooling Layer

The pooling layer is an indispensable component in CNNs, typically positioned after Conv and activation layers to perform spatial downsampling and feature aggregation. Specifically, by aggregating local feature responses within predefined receptive fields, it yields compact low-resolution feature representations that preserve task-critical semantics while discarding redundant spatial details [51]. This structural mechanism fulfills dual critical objectives. First, it enhances computational efficiency by reducing arithmetic intensity, memory bandwidth demands, and the parameter volume of subsequent dense layers, which also effectively mitigates overfitting. Second, it improves representational robustness by conferring spatial translation invariance against minor positional shifts, geometric deformations, and input noise. Ultimately, pooling facilitates hierarchical feature abstraction from low-level structural patterns to high-level semantic concepts, enabling deeper layers to integrate information over progressively broader receptive fields. This section presents the typical pooling operations widely adopted in modern CNN architectures.

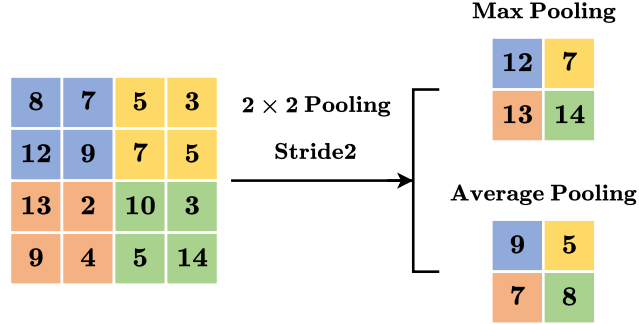


Figure 2.6: Numerical schematic of 2×2 max pooling and average pooling operations applied to a 4×4 ifmap with a stride of 2.

Max Pooling

Max pooling is a non-linear downsampling operation that extracts the maximum activation value within a predefined local neighborhood [52]. Mathematically, for an ifmap X and a spatial pooling window $\mathcal{R}_{i,j}$, it is formulated as:

$$y_{i,j} = \max_{(m,n) \in \mathcal{R}_{i,j}} x_{m,n}.$$

As illustrated in the upper section of Fig. 2.6, this mechanism propagates only the most salient feature responses. Consequently, it delivers strong local translation invariance and stands as the default spatial downsampling strategy in modern CNN architectures.

Average Pooling

In contrast to max pooling, this linear alternative performs downsampling by computing the arithmetic mean over the local window $\mathcal{R}_{i,j}$:

$$y_{i,j} = \frac{1}{|\mathcal{R}_{i,j}|} \sum_{(m,n) \in \mathcal{R}_{i,j}} x_{m,n},$$

where $|\mathcal{R}_{i,j}|$ denotes the size of the pooling window. As shown in Fig. 2.6 (bottom), this mechanism smooths spatial variations to preserve background distributions, at the expense of attenuating salient features.

Global Pooling

Global pooling represents an extreme spatial downsampling scheme that aggregates each channel of the fmap into a single scalar. Functionally, this is equivalent to applying a pooling window with spatial dimensions matching the full extent of the fmap. By integrating the global spatial context, it yields a highly abstracted representation that forces the network to capture holistic spatial responses. It comprises two primary variants: *Global Average Pooling* (GAP), which computes

the arithmetic mean of all spatial activations per channel, and *Global Max Pooling* (GMP), which extracts the maximum activation across the entire spatial grid.

2.1.5 Fully Connected Layer

Positioned after the hierarchical feature extraction stages, the Fully Connected (FC) layer serves as the final decision-making component of the CNN. Implemented as a *Multilayer Perceptron* (MLP), illustrated in Fig. 2.7(a), it employs dense connectivity to aggregate spatially distributed local features into high-level global representations [53]. Through this mechanism, the FC layer enables the critical transition from localized spatial processing to global semantic inference, mapping latent features directly into the target output space.

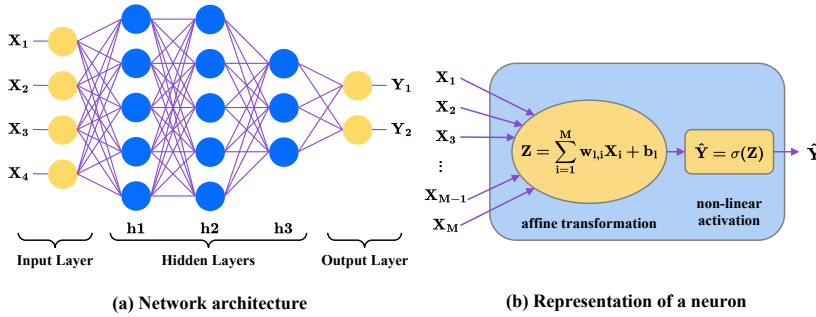


Figure 2.7: Schematic of a FC layer. (a) Architecture of an MLP illustrating dense connectivity between consecutive layers. (b) Internal computational mechanism of an individual neuron, consisting of an affine transformation and a subsequent non-linear activation. Adapted from [53].

The transition from convolutional feature tensors to FC inputs typically follows two primary strategies: traditional spatial flattening and modern GAP. While flattening preserves all spatial activation values via vectorization, the resulting high-dimensional vector fundamentally dictates a massive weight matrix in the subsequent dense layer, incurring a substantial parameter overhead and an elevated overfitting risk. In contrast, leveraging GAP as an architectural alternative bypasses the flattening operation entirely. By mapping each fmap directly to a per-channel scalar before the classification head, GAP inherently regularizes the network structure. This yields a highly compact representation that maintains semantic integrity while significantly improving parameter efficiency and mitigating overfitting [54].

As detailed in Fig. 2.7(b), each FC neuron fundamentally executes an affine transformation followed by a nonlinear activation, formulated as

$$\hat{Y} = \sigma\left(\sum_{i=1}^M w_{i,1} X_i + b_1\right).$$

While highly effective for global feature integration, such dense connectivity introduces considerable model complexity and elevates overfitting risk. To mitigate this issue, *Dropout* is employed as a regularization technique during training. By randomly deactivating a fraction of neurons, Dropout prevents complex co-adaptations, encouraging the learning of disentangled, robust representations that strengthen generalization [55].

2.1.6 Softmax Layer

Succeeding the high-level feature aggregation in the preceding FC layers, the Softmax layer operates as the terminal classification head of the CNN architecture, projecting unnormalized output scores (logits) onto a normalized probability distribution over predefined classes. Rather than executing further feature extraction, it provides a probabilistic interpretation of the network outputs, facilitating a direct translation from learned representations to categorical decision outcomes.

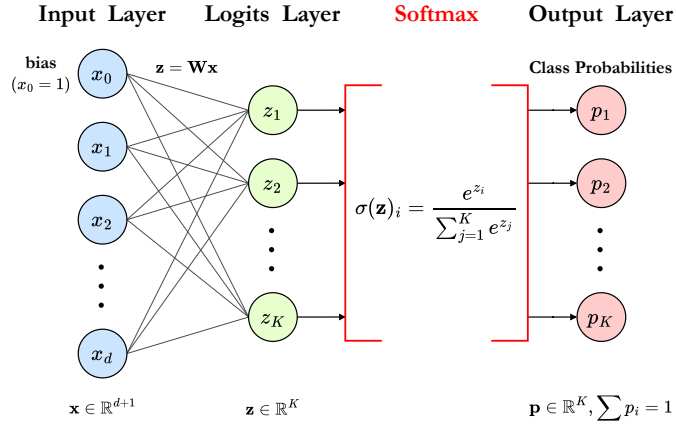


Figure 2.8: Computational pipeline of the Softmax layer.

As illustrated in Fig. 2.8, the Softmax layer processes a vector of unnormalized scores (logits), denoted as $\mathbf{z} \in \mathbb{R}^K$, where K designates the total number of target classes. It maps these logits onto a normalized probability distribution $\mathbf{p} \in \mathbb{R}^K$ by exponentiating each component and applying a global normalization factor:

$$p_k = \sigma(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{j=1}^K e^{z_j}}, \quad k = 1, \dots, K.$$

This formulation guarantees that each output is bounded within $p_k \in (0, 1)$ and that the elements collectively lie on a probability simplex, i.e., $\sum_{k=1}^K p_k = 1$. Consequently, the Softmax layer yields a valid categorical distribution, encoding the model's relative confidence across all competing classes. During inference, the

definitive prediction is typically obtained via the *maximum a posteriori* (MAP) decision rule, evaluated as $\arg \max_k p_k$.

During training, the Softmax layer is algorithmically fused with the *Cross-Entropy* (CE) loss function to construct a unified optimization objective for multi-class classification [56]. For a given one-hot encoded ground-truth vector $\mathbf{y}$, the empirical loss is formulated as:

$$L = - \sum_{k=1}^K y_k \log p_k.$$

This objective explicitly penalizes the divergence between the predicted probability distribution and the target distribution, yielding highly stable gradients that accelerate convergence. Notably, the Softmax-CE fusion admits a simplified gradient expression with respect to the input logits (i.e., $\frac{\partial L}{\partial z_k} = p_k - y_k$), thereby circumventing explicit Jacobian computations and mitigating numerical instability during backpropagation.

From both computational and architectural perspectives, the Softmax layer constitutes a critical numerical bottleneck rather than a parameter-intensive module. Although it operates on a vector whose dimensionality equals the output class count and thus incurs a negligible memory footprint, its execution introduces nontrivial overhead due to exponentiation, division, and a global reduction step. These operations due to exponentiation, division, and a global reduction step increase arithmetic intensity relative to preceding MAC-dominated layers, while also imposing strict data dependency and synchronization requirements. Particularly in resource-constrained or low-precision on-device training scenarios, such characteristics can limit throughput and energy efficiency. Consequently, despite its small parameter footprint and limited contribution to overall compute volume, the Softmax layer plays a critical role in determining numerical stability, execution efficiency, and convergence behavior of the training pipeline.

2.2 Training versus Inference

The operational lifecycle of DNNs typically consists of two distinct phases: training and inference, as illustrated in Fig. 2.9. Within the context of supervised learning, *Training* is a computationally intensive, iterative optimization procedure that utilizes large-scale labeled datasets to minimize a specific loss function. This is achieved through a continuous cycle of forward propagation, error back-propagation, and gradient-based parameter updates [57]. In contrast, *inference* corresponds to the model deployment stage, executing a unidirectional forward pass to yield deterministic predictions on unseen data using frozen model weights. Although both stages share an identical network topology and rely on fundamental computational operations, their divergent algorithmic flows impose asymmetric demands on memory footprint and hardware resources, thus requiring distinct architectural optimizations for efficient execution.

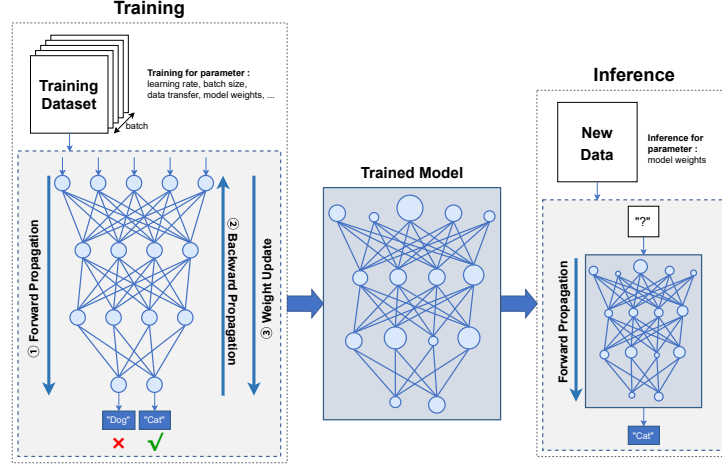


Figure 2.9: Conceptual framework of training and inference phases. Adapted from [57].

2.2.1 Inference

Inference represents a deterministic execution phase with simplified dataflow and efficient memory access patterns, as shown in Fig. 2.10(a). During model deployment, input activations propagate sequentially through the network layers without the computational burden of error backpropagation and parameter updates. BN employs frozen running statistics that can be fused into the preceding convolution, inherently eliminating the need for independent BN computations and associated intermediate memory allocations. Crucially, without the need to backpropagate gradients, intermediate activations no longer need to be persistently retained within the memory hierarchy. Instead, memory accesses are limited to fetching static parameters (W_{cnn} , W_{bn}) and input features, while output activations can be dynamically overwritten via in-place operations once consumed by the subsequent layer. This unidirectional execution flow significantly reduces memory bandwidth overhead and memory footprint.

2.2.2 Training

2.2.2.1 Memory Footprint and Data Access Patterns

Training imposes intensive computational workloads and intricate memory access patterns, structured into three interdependent stages, as illustrated in Fig. 2.10(b) [58]. During forward propagation, the network generates layer-specific intermediate activations (X_{cnn} , X_{bn} , and X_{act}) that must be retained in the memory hierarchy. Critically, batch normalization (BN) layers require the on-the-fly computation of the mini-batch mean (μ) and variance (σ^2), preserved for subsequent

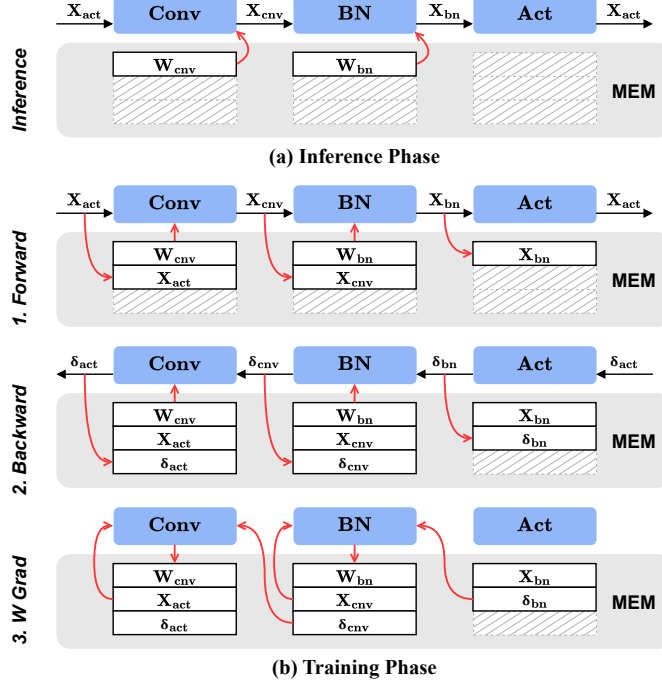


Figure 2.10: Dynamic memory footprint and data access patterns. (a) The inference phase. (b) The training phase (Forward, Backward, and Weight Gradient). Adapted from [58].

gradient derivations. Backward propagation relays error gradients (δ_{act} , δ_{bn} , and δ_{cnn}) from deep to shallow layers, relying on activation function derivatives computed from the cached activations to ensure precise error backflow. Finally, weight gradient computation leverages both forward activations and backward gradients to derive updates for trainable parameters (W_{cnn} and W_{bn}), closing one iterative training loop. Driven by repeated read and write operations of activations, gradients, and BN statistics, this bidirectional data flow imposes substantial memory bandwidth and footprint bottlenecks that distinguish training from inference.

2.2.2.2 Algorithmic Workflow of Training

The algorithmic workflow of the training phase is driven by an iterative optimization loop designed to minimize a specific loss function, as illustrated in the computational graph of Fig. 2.11. This process comprises three sequential operational phases: the forward pass, the backward pass, and the parameter update.

- **Forward Propagation and Loss Evaluation:** In a supervised setting,

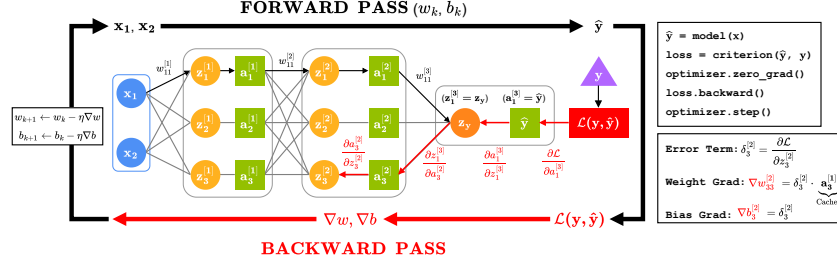


Figure 2.11: Computational graph of the deep neural network training cycle, illustrating the execution flow, contrasting the forward evaluation of the loss $\mathcal{L}$ with the backward derivation of parameter gradients ($\nabla w, \nabla b$).

the training cycle initiates with the forward pass given a mini-batch of input samples and their corresponding ground-truth labels. As data propagate through the hierarchical architecture, intermediate activations are computed via successive affine transformations and non-linear activations, ultimately yielding the network predictions ($\hat{y}$). A predefined loss function, denoted $\mathcal{L}(y, \hat{y})$, subsequently evaluates these predictions against the targets to derive a scalar penalty, thereby explicitly quantifying the optimization error.

- **Error Backpropagation:** To minimize this quantified error, the backward pass is executed. Governed by the chain rule of differential calculus, the error signal is propagated in reverse topological order throughout the computational graph. This recursive procedure yields layer-specific error terms (δ), which serve as the fundamental basis for computing parameter gradients.
- **Weight Gradient Derivation and Optimization:** The exact gradients of the loss with respect to the trainable parameters (∇w and ∇b) are derived by integrating the cached forward activations with the backpropagated error signals. An optimization algorithm then applies these gradients to update the model parameters, thereby closing the current training iteration and driving the systematic convergence of the loss function.

Beyond the algorithmic workflow, the selection of loss functions introduces distinct computational constraints on target hardware architectures. Representative examples include **Cross-Entropy** (CE) for classification and **Mean Squared Error** (MSE) for regression. CE, typically coupled with Softmax activations, requires transcendental operations such as exponentials and logarithms, leading to increased arithmetic intensity and potential numerical instability under low-precision formats. In contrast, MSE relies on simple algebraic operations, resulting in significantly lower computational and hardware overhead. This arithmetic disparity propagates into the backward pass, affecting both gradient computation complexity and the scheduling of backward dataflow.

The choice of optimization algorithm dictates the computational complexity and

memory footprint during model training. **Stochastic Gradient Descent** (SGD) ensures hardware compatibility by updating parameters using only instantaneous gradients and a fixed learning rate, incurring minimal storage overhead. In comparison, adaptive optimizers like **Adam** maintain first- and second-order moment estimates for each parameter. Storing these auxiliary state variables imposes a dominant memory overhead that exceeds the footprint of the model weights alone [59]. This expanded memory requirement elevates bandwidth demands and arithmetic intensity, thereby restricting the viability of adaptive optimizers for resource-constrained on-device learning.

2.3 Spatial Architecture and Dataflow Mapping

2.3.1 Convolution as Nested Loops and Data Reuse Opportunities

From a computational perspective, the multi-channel 2D convolution is formally characterized as a 7-nested loop execution [60]. As detailed in Algorithm 2, the procedure systematically traverses seven orthogonal dimensions: the batch size (N), the number of filters/output channels (M), the spatial dimensions of the ofmap (E, F), the input channel depth (C), and the kernel spatial extent (R, S). The core primitive within this nested structure is a MAC operation, which aggregates partial products to produce each element of the ofmap. The geometric correspondence between these loop indices and the three core tensors is visualized in Fig. 2.12.

Algorithm 2 The 7-nested loop of Convolution

```

Input:  $I[N, C, H, W]$ ,  $W[M, C, R, S]$ 
Output:  $O[N, M, E, F]$ 
for  $n = 0; n < N; n++$  ▷ batch size
  for  $m = 0; m < M; m++$  ▷ output channel / filter
    for  $e = 0; e < E; e++$  ▷ output height
      for  $f = 0; f < F; f++$  ▷ output width
        for  $c = 0; c < C; c++$  ▷ input channel
          for  $r = 0; r < R; r++$  ▷ kernel height
            for  $s = 0; s < S; s++$  ▷ kernel width
               $O[n, m, e, f] += W[m, c, r, s] \cdot I[n, c, e + r, f + s]$ 

```

This nested formulation reveals inherent mathematical data redundancy, presenting three fundamental data reuse opportunities:

- **Input Reuse (IR):** A loaded ifmap element is shared across M distinct filters to generate multiple output channels.
- **Output Reuse (OR):** This strategy focuses on the local accumulation of psums derived from multiple input channels and kernel elements (across the

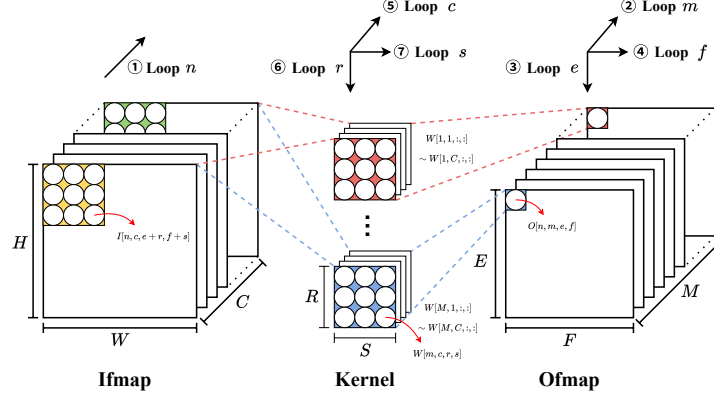


Figure 2.12: Geometric mapping of the 7-nested loop indices over the spatial coordinates of ifmaps, weight kernels, and ofmaps.

C , R , and S dimensions).

- **Weight Reuse (WR):** A specific weight parameter is reused across the spatial dimensions (E, F) of the ifmap as the window slides, or across multiple samples in a batch (N).

Identifying these underlying reuse patterns establishes the theoretical foundation for designing specialized dataflows that effectively close the gap between peak hardware throughput and stringent memory bandwidth constraints.

2.3.2 The Memory Wall: From Temporal to Spatial Architectures

The primary bottleneck of modern CNN acceleration is the *Memory Wall*, characterized by the severe energy and latency disparity between hierarchical data movement and arithmetic computation. As quantified in Fig. 2.13, while a standard MAC operation is normalized to a baseline of $1\times$, fetching operands from off-chip DRAM requires approximately $200\times$ more energy. Even intermediate on-chip buffer accesses incur a significant $6\times$ energy overhead relative to computation [33]. This steep hierarchical energy gradient dictates the core design philosophy for efficient accelerators: minimizing global memory fetches by maximizing on-chip data reuse and locality.

General-purpose processors, such as CPUs and GPUs, typically utilize **temporal architectures** based on the *Single Instruction, Multiple Data* (SIMD) or *Single Instruction, Multiple Threads* (SIMT) paradigms, as illustrated in Fig. 2.14 (left). These architectures rely on a centralized control unit to broadcast a single instruction across multiple parallel execution lanes. As indicated by the black data paths, each ALU must frequently fetch operands from and return results to a shared memory hierarchy and centralized register file. This centralized approach necessitates long-distance data movement across the chip. Considering the high energy costs

associated with such movement shown in Fig.2.13, this paradigm inevitably leads to suboptimal efficiency for data-intensive CNN workloads.

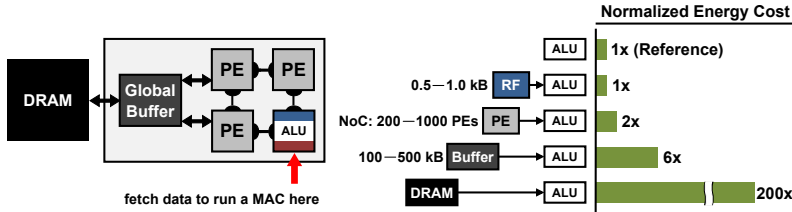


Figure 2.13: Normalized energy cost of data movement across different levels of the memory hierarchy relative to a single MAC operation (1×). Adapted from [31].

To mitigate the Memory Wall, **spatial architectures** (dataflow processing) have emerged as the dominant paradigm for modern deep learning accelerators. As illustrated in Fig. 2.14 (right), a spatial architecture replaces the centralized control of temporal designs with a distributed array of PEs. Each PE functions as an autonomous unit that integrates a local ALU, dedicated control logic, and a private Register File (RF). This organization enables localized dataflow processing, where data are streamed directly between adjacent units via point-to-point interconnects (PE-to-PE communication). By maintaining data stationarity or moving operands only over short distances, spatial architectures drastically reduce the reliance on energy-intensive global memory accesses, thereby establishing the essential hardware substrate for advanced dataflow mapping and optimizations.

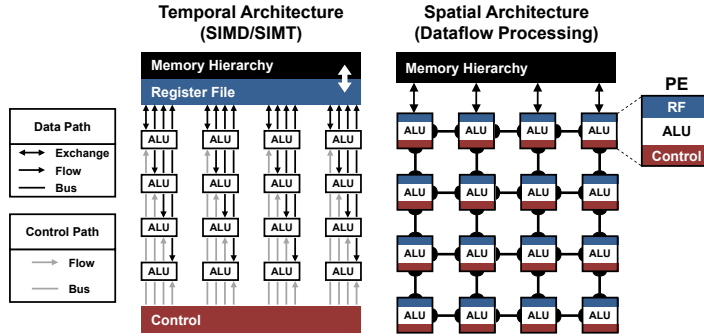


Figure 2.14: Comparison of parallel computing paradigms. (Left) Temporal architecture with centralized control and shared memory hierarchy. (Right) Spatial architecture featuring a distributed PE array. Adapted from [31].

2.3.3 Loop Mapping: Tiling, Unrolling, and Interchange

Mapping the 7-nested convolutional iteration space onto hardware requires reconciling algorithmic computation with stringent on-chip resource constraints. Their massive data footprint and intrinsic arithmetic intensity exceed the inherent limits of on-chip memory bandwidth and computational throughput. To alleviate these bottlenecks, loop transformation methods such as *tiling*, *unrolling*, and *interchange* are employed to orchestrate the spatial and temporal scheduling of the convolutional workload [61].

Table 2.2: Loop Optimization Parameters for Hardware Mapping

Loop Dimension	Loop Index (L_i)	Unroll Factor (P_i)	Tiling Factor (T_i)
Batch Size (N)	L_N	P_N	T_N
Output Channel Count (M)	L_M	P_M	T_M
Output Spatial Height (E)	L_E	P_E	T_E
Output Spatial Width (F)	L_F	P_F	T_F
Input Channel Count (C)	L_C	P_C	T_C
Kernel Height (R)	L_R	P_R	T_R
Kernel Width (S)	L_S	P_S	T_S

Loop Tiling

Loop Tiling mitigates on-chip buffer capacity limitations by partitioning global tensors into manageable sub-tensors. Governed by dimension-specific tiling factors T_i listed in Table 2.2, such sub-tensors are configured to adapt precisely to the on-chip memory hierarchy. This approach extends on-chip data locality and reduces energy-intensive data transfers between the off-chip DRAM and computational datapaths, as demonstrated by the tiled sub-tensor partitioning in Fig. 2.15.

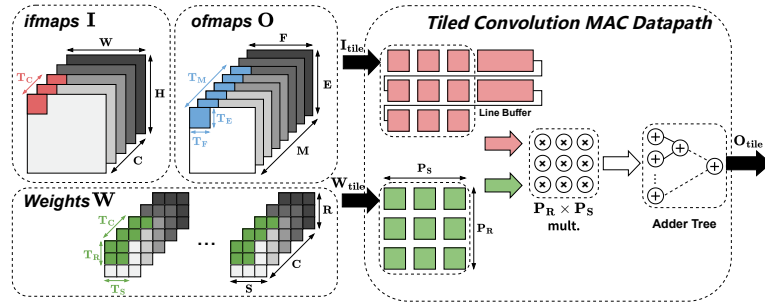


Figure 2.15: Hardware mapping of the 7-nested convolution loops via temporal tiling and spatial unrolling. Adapted from [61].

Loop Unrolling

Loop unrolling exploits hardware spatial parallelism by mapping iterations onto an array of PEs for concurrent execution. The unroll factors P_i define the degree of hardware parallelism, dictating the number of active computational units within the datapath. Specifically, unrolling the kernel spatial dimensions facilitates the simultaneous computation of localized inner sums. As illustrated in Fig. 2.15, continuous data feeding is sustained via on-chip line buffers, while parallel computation outputs are aggregated through a binary adder tree for efficient *partial sum* (psum) reduction.

Loop Interchange

Loop interchange reconfigures the nesting hierarchy to reorder the temporal priority of data access. Whereas tiling characterizes computational granularity, and unrolling regulates spatial parallelism, iterative sequence determines the behavior of on-chip data reuse. Such iterative rearrangement enables specialized dataflow configurations [62], such as input or output reuse modes, providing the architectural flexibility necessary to adapt to diverse CNN workloads.

As illustrated in Fig. 2.16, loop interchange translates the theoretical reuse opportunities into physical hardware behavior. The nesting order of the outer loops determines which data tensor remains stationary within the on-chip buffers. In the Input Reuse scheme (Fig. 2.16a), the spatial loops (E, F) are embedded within the channel loops (M, C), enabling ifmaps to be cached and reused across multiple output channels to update psums. In contrast, Output Reuse repositions spatial loops to the outermost hierarchy (Fig. 2.16b), facilitating the local accumulation of psums to finalize the 3D convolution on-chip before committing final ofmaps to memory. Weight Reuse ensures the persistence of kernels across the entire spatial extent of the ifmap (Fig. 2.16c), thereby minimizing weight-reloading penalties by traversing the full spatial dimensions before switching filters. This inherent adaptability allows the hardware to maximize energy efficiency by mitigating high-cost data movement tailored to specific CNN layer dimensions.

2.3.4 Taxonomy of Dataflows: Input, Output, and Weight Stationary

Dataflow mapping fundamentally determines the hardware efficiency of CNN spatial architectures by orchestrating data movement, reuse patterns, and loop execution order [63]. By pinning specific nested loops to the innermost execution level, a dataflow statically maximizes the on-chip retention of a target tensor (ifmaps, weights, or psums), thereby significantly reducing energy-intensive off-chip DRAM accesses. Based on this principle, traditional systolic arrays typically adopt one of three foundational paradigms: *Input Stationary* (IS), *Output Stationary* (OS), or *Weight Stationary* (WS), as illustrated in Fig. 2.17. To facilitate a clear comparison, these dataflows are depicted using a unified loop nesting hierarchy: Loop 1 (N , batch, outermost) $\supset$ Loop 2 (M , output channels, middle) $\supset$ Loop 3 (C , input channels, innermost), with spatial and temporal unrolling denoted by light

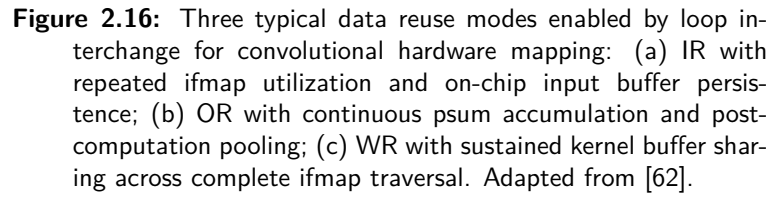


Figure 2.17: Hardware mapping of three foundational 1D dataflow paradigms onto a 2D systolic array. Redrawn from [63].

Input Stationary (IS) In the IS dataflow (Fig. 2.17a), ifmap activations (I) are held in local PE registers to minimize external memory fetches. Concurrently, weight kernels (W) stream horizontally across the array while output psums (O) accumulate vertically. This configuration is achieved by spatially unrolling the batch (N) and input channel (C) dimensions, paired with the temporal unrolling of the output channels (M). Sustaining input stationarity across all M iterations maximizes IR by sharing cached activations among multiple filters. Consequently, IS excels in network layers with massive spatial dimensions and shallow channel depths, though it compromises the reuse of weights and psums.

Output Stationary (OS) The OS dataflow (Fig. 2.17b) tackles the dominant memory bandwidth bottleneck: frequent reads and writes of intermediate psums. By locking output psums (O) within local registers, OS bypasses costly write-back cycles to accumulate results on-site. Simultaneously, inputs (I) and weights (W) are continuously broadcast across the grid. This behavior is realized by spatially unrolling the batch (N) and output channel (M) dimensions, coupled with the temporal unrolling of the input channels (C). Preserving these sums until the channel-wise reduction completes maximizes OR. As a result, OS is highly effective for deep convolutional layers with large channel depths, although it inherently limits the data locality of inputs and weights.

Weight Stationary (WS) The WS dataflow (Fig. 2.17c) addresses the severe memory overhead of parameter fetching. By anchoring convolution kernels (W) within the PE array, WS prevents redundant weight reloading. While these parameters remain fixed, input features (I) and psums (O) sequentially stream through the array. This execution is organized by spatially unrolling the output channel (M) and input channel (C) dimensions, alongside the temporal unrolling of the batch (N) loop. Fixing these kernels across all N iterations maximizes WR. This property positions WS as the preferred strategy for weight-intensive workloads, including fully-connected layers and convolutions with large filter sizes. However, it inevitably diminishes the reuse efficiency of input and output tensors.

2.3.5 Row-Stationary Dataflow for On-Device Training

While conventional dataflows (such as IS, OS, and WS) are architecturally constrained to retain only a single tensor on-chip, they inherently compromise the reuse potential of the other two data types, leading to suboptimal system-level energy efficiency. To overcome this limitation, the **RS** dataflow is adopted to maximize the local reuse of all data types, including weights, ifmaps, and psums, within each PE [64]. As depicted in Fig. 2.18(a), the fundamental computational primitive of the RS paradigm is the 1D row-wise convolution. These 1D primitives are spatially mapped onto the 2D PE array to fully exploit hardware parallelism. Specifically, weight rows are systolically forwarded horizontally across the PE rows, ifmap rows stream diagonally through the array, and psums are accumulated vertically along the PE columns, as shown in Fig. 2.18(b). This specialized spatial

routing minimizes energy-intensive off-chip DRAM accesses by extending data retention within the interconnected PE array.

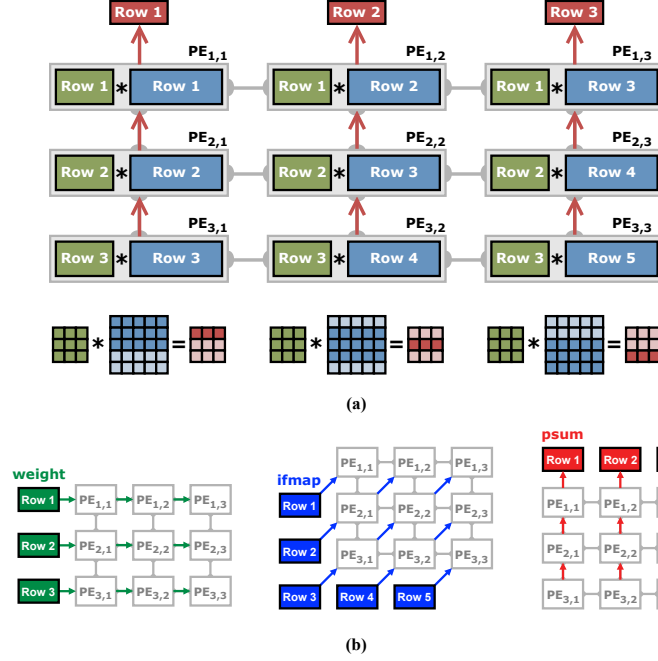


Figure 2.18: Execution of the RS dataflow on a 2D PE array.
 (a) Logical mapping of row-wise 1D convolution primitives.
 (b) Spatial data routing illustrating horizontal weight propagation, diagonal ifmap streaming, and vertical psum accumulation. Adapted from [64, 65].

At the microarchitectural level, each PE incorporates a local register file comprising dedicated *scratchpad memories* (spads) for weights, ifmaps, and psums, as illustrated in Fig. 2.19. This physical separation of storage allows operands to be buffered and retrieved locally without repeated accesses to the global buffer (GLB) or off-chip DRAM, thereby enabling independent scheduling of data movement and accumulation [66]. The core computational engine is a MAC unit, typically implemented using a Digital Signal Processing (DSP) block, which consumes one resident weight and one ifmap element per cycle. It multiplies the operands, accumulates the result with the incoming psum from the adjacent lower PE, and forwards the updated psum upward along the column to support vertical accumulation. Collectively, this tightly coupled design of local storage, computation, and inter-PE communication enables each PE to support fine-grained data reuse and efficient data propagation, thereby sustaining high utilization and improving energy efficiency in the deeply pipelined PE array.

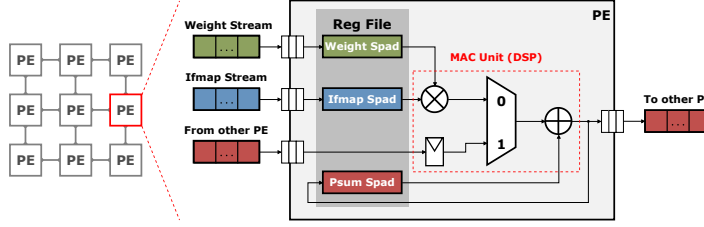


Figure 2.19: Microarchitecture of a single PE within the RS spatial array.

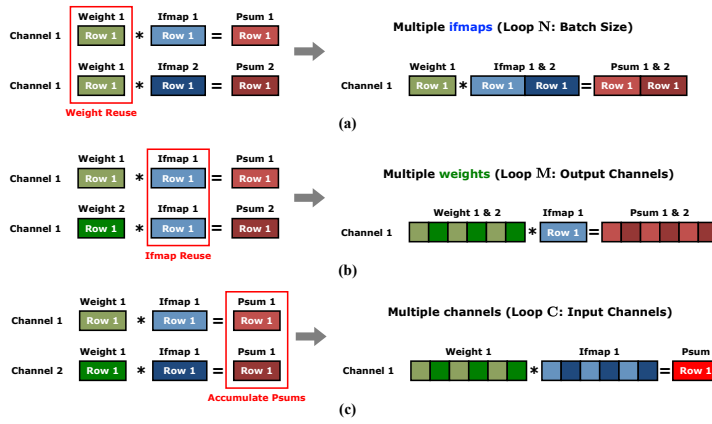


Figure 2.20: Temporal multiplexing and accumulation strategies within a single PE for handling high-dimensional convolutions. (a) Weight reuse across multiple ifmaps. (b) Ifmap reuse across multiple weights. (c) Psum accumulation across multiple input channels. Redrawn from [33].

Beyond spatial mapping, the RS dataflow exploits temporal multiplexing within each PE to accommodate high-dimensional convolution loops that scale beyond the physical array dimensions, as illustrated in Fig. 2.20. Depending on the mapped loop dimension, the PE dynamically configures its stationary operands: a resident weight row is reused across streaming ifmaps to process larger batch sizes (Loop N), whereas a buffered ifmap row is held stationary and multiplied against sequentially fetched weights to compute multiple output channels (Loop M). Furthermore, intermediate psums spanning different input channels (Loop C) are iteratively accumulated within the local spad, ensuring that only fully resolved output activations are evicted from the PE. By synergizing intra-PE temporal reuse with inter-PE spatial reuse, the RS dataflow delivers a balanced mapping strategy that minimizes data movement and sustains high utilization under the stringent energy and performance constraints of on-device training.

Design Methodology

3.1 Task Modeling and Network Configuration

To lay a solid theoretical foundation for the subsequent computational abstraction and hardware microarchitecture design, this section formally defines the target on-device learning task, specifies the optimization constraints, and details the structural parameters of the CNN adopted in this work.

3.1.1 Edge Fine-Tuning Task

While pre-trained models generalize well on standard datasets, deployment in edge environments often suffers accuracy degradation due to domain shift induced by unpredictable sensory and environmental variations [31, 67]. To emulate realistic edge deployment scenarios, such as aging and misaligned surveillance cameras, this study constructs a visual distortion pipeline with deterministic perturbation settings. Specifically, the process employs fixed parameters: 100% horizontal flip for camera mirroring, a 15° fixed rotation for installation tilt, and aggressive color jittering with $1.2\times$ brightness and $0.8\times$ contrast to simulate sensor drift.

It is imperative to clarify that these deterministic perturbations are not used as stochastic data augmentation techniques for mitigating overfitting during training. Instead, they are deliberately constructed to emulate realistic edge deployment degradations. This pipeline is therefore intended to rigorously evaluate the model's ability to recover accuracy under prescribed domain shifts, rather than to artificially improve baseline generalization and mask the necessity of on-device fine-tuning. Consequently, this work adopts a dual-track evaluation paradigm to benchmark model performance across normal and distorted test sets, measuring accuracy recovery while capturing generalization on the original domain.

To restore accuracy under the stringent resource constraints of an FPGA platform, an on-device partial fine-tuning strategy is formulated. Performing full-network backpropagation requires buffering extensive intermediate activations throughout the training cycle. For shallow layers with high spatial resolutions (e.g., Conv1 and

Conv2 at 32×32), this caching requirement would rapidly exhaust the available on-chip memory budget. Consequently, the backpropagation path is structurally truncated: gradient computation and weight updates are restricted to the deeper, task-specific modules (Conv3 through Conv6, and the FC layer), while the shallow layers remain frozen as generic feature extractors.

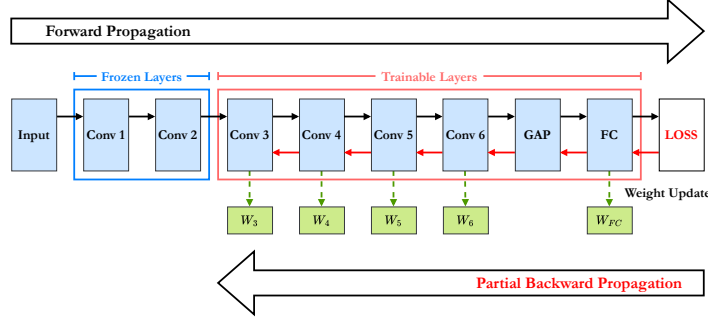


Figure 3.1: Algorithmic workflow of the hardware-aware partial fine-tuning strategy.

As illustrated in Fig. 3.1, this partial fine-tuning strategy is enforced by the hardware control logic across all three training phases.

- During forward propagation, all layers execute sequentially. The shallow frozen layers (Conv1 and Conv2) still write their output feature maps to the GLB to feed subsequent layers, but they explicitly skip the dedicated backprop activation and gradient storage.
- The backward propagation establishes a hard truncation boundary at Conv3. To maximize hardware efficiency, the controller explicitly skips the computation of the input data gradient for Conv3 (i.e., δA_2 is not calculated), directly reducing the computational overhead and memory footprint associated with intermediate backward gradients.
- Finally, the weight update phase strictly restricts parameter modifications to the designated GLB address space spanning from Conv3 to the FC layer, ensuring the shallow layers remain physically frozen. This hardware-aware algorithmic design effectively alleviates the primary memory capacity bottleneck, preserving the feasibility of on-device fine-tuning.

3.1.2 Target CNN Architecture

The target model employed for this fine-tuning task is a lightweight VGG-style architecture [68] tailored for CIFAR-10 classification [69]. As outlined in Table 3.1, the network consists of six 3×3 convolutional layers, three 2×2 max-pooling layers, a GAP layer, and a final FC classification head. To introduce

necessary nonlinearity, ReLU activation is systematically applied after each convolutional layer. The spatial resolution is progressively downsampled from 32×32 to 4×4 while the channel depth increases from 3 to 128. By replacing the conventional large dense classifier with a GAP-based design, the model complexity is substantially reduced, yielding an overall parameter count of approximately 288K. The parameter count for each convolutional layer is analytically computed as $\text{Params} = C_{out} \times (C_{in} \times K^2 + 1)$, where K denotes the kernel size and the “+1” term accounts for the bias.

Table 3.1: Target CNN configuration with Partial Fine-tuning segmentation.

Layer (In→Out)	Output Size	Kernel	Status	Weights	Biases	Params
Conv1 (3 → 32)	32×32	3×3	Frozen	864	32	896
Conv2 (32 → 32)	16×16	3×3	Frozen	9,216	32	9,248
Conv3 (32 → 64)	16×16	3×3	Trainable	18,432	64	18,496
Conv4 (64 → 64)	8×8	3×3	Trainable	36,864	64	36,928
Conv5 (64 → 128)	8×8	3×3	Trainable	73,728	128	73,856
Conv6 (128 → 128)	4×4	3×3	Trainable	147,456	128	147,584
GAP	1×1	-	Trainable	0	0	0
FC (128 → 10)	10	-	Trainable	1,280	10	1,290
Total				287,840	458	288,298

Note: Total parameters correspond to the hardware-imported values after BN fusion.

The structural dimensions provided in Table 3.1 are directly linked to the physical hardware constraints. Based on the partial fine-tuning strategy defined in Section 3.1.1, the network is logically divided into two segments: (i) frozen layers (Conv1 and Conv2), which operate solely in forward inference mode, and (ii) trainable layers (Conv3 through Conv6, GAP, and FC), which participate in the full forward, backward, and weight-update cycle. This algorithmic truncation is rigorously enforced in the hardware architecture, which omits the backward computational datapaths for Conv1 and Conv2 to reduce on-chip resource overhead. Consequently, the specific layer parameters and feature map resolutions defined in this section establish the absolute baseline constraints for sizing the GLB, configuring the PE array parallelism, and physically implementing the RS dataflow architecture detailed in Chapter 4.

3.2 Unified Training Computation Model

To map partial fine-tuning efficiently onto spatial hardware architectures, a unified mathematical framework is therefore indispensable. Different from the unidirectional feed-forward nature of model inference, training consists of three distinct operational phases: Forward Propagation (FW), Backward Propagation (BW) for error backpropagation, and Weight Gradient (WG) computation for parameter

updates. This section reveals that, despite disparate mathematical expressions, the three stages share inherent structural symmetry and can be generalized into a unified 7-dimensional nested loop model.

3.2.1 Training Computation Primitives

3.2.1.1 Forward Propagation

Forward propagation in a convolutional layer is fundamentally a MAC operation. Following the notation introduced in Chapter 2, we briefly restate the relevant tensor dimensions and loop indices: N denotes the batch size; C and K represent the input and output channel depths, respectively; H and W denote the spatial height and width of the input feature map; R and S correspond to the spatial dimensions of the convolutional kernel; and E and F represent the spatial height and width of the output feature map.

The standard forward convolution generating the output tensor O from the input tensor I and weight tensor W is expressed as:

$$O(n, k, e, f) = \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} I(n, c, e+r, f+s) \times W(k, c, r, s).$$

From a hardware perspective, pure inference allows the input tensor I to be overwritten or discarded from the on-chip memory immediately after O is computed. However, during training, I must be preserved and cached in the memory hierarchy, as it is a mandatory operand for subsequent WG computation. This mechanism, referred to as activation caching, constitutes the primary memory bottleneck in on-device training. The spatial dimensions (E, F) of shallow layers are excessively large, leading to an exponential explosion in caching requirements. This constraint directly motivates the design choice described in Section 3.1.1: fine-tuning should be restricted to deeper layers, such as Conv5 and Conv6, where spatial dimensions have been sufficiently reduced to fit within the limited on-chip block RAM resources of the FPGA.

3.2.1.2 Backward Propagation

The first step of the backward pass is to compute the data gradient, which routes the error signal from the current layer back to the preceding layer. Let δO denote the incoming error gradient from the subsequent layer, and δI denote the outgoing gradient propagated to the previous layer. Assuming a convolutional stride of 1, the computation is formulated as:

$$\delta I(n, c, h, w) = \sum_{k=0}^{K-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} \delta O(n, k, h-r, w-s) \times W(k, c, r, s).$$

(Note: To maintain valid indices for $(h-r)$ and $(w-s)$ at spatial boundaries, standard zero-padding is implicitly applied to δO .)

A closer examination reveals a profound Geometric Duality between the BW and FW phases. In the BW formulation, the output channel K acts as the reduction loop, while the input channel C serves as the parallel output dimension. Spatially, the indices traverse $(h - r)$ and $(w - s)$, which is mathematically equivalent to rotating the weight kernel W by 180° and applying zero-padding to δO . Consequently, data gradient computation is isomorphic to a transposed convolution. This mathematical symmetry guarantees that the BW phase can seamlessly reuse the identical hardware MAC arrays designed for the FW phase. Rather than physically rearranging data in memory, this hardware-level equivalence is established through dedicated address transformation logic, the specific mechanisms of which are elaborated in the subsequent architectural implementation sections.

3.2.1.3 Weight Gradient Computation

The computation of the weight gradient δW is unique to the training cycle and determines the parameter update. It is defined as the cross-correlation between the cached input activations I and the incoming error gradients δO :

$$\delta W(k, c, r, s) = \sum_{n=0}^{N-1} \sum_{e=0}^{E-1} \sum_{f=0}^{F-1} I(n, c, e + r, f + s) \times \delta O(n, k, e, f).$$

A fundamental shift in computational roles occurs in this phase. In FW and BW, the batch size N and the spatial dimensions E and F operate as parallel output dimensions. In WG, however, N , E , and F orthogonally transform into reduction (accumulation) dimensions. To synthesize a hardware architecture capable of supporting all three phases without instantiating redundant compute engines, these disparate operations must be abstracted into a single, unified 7-dimensional nested loop, as formulated in Algorithm 3.

This unified abstraction proves that the identical 7 indices govern all training stages, but its viability relies on dynamic spatial bounding. While the nested loop structure remains rigid to match the physical hardware iterators, the spatial traversal bounds ($Bound_y$ and $Bound_x$) dynamically adapt based on the execution phase. During the FW and WG phases, the bounds correspond to the output feature map dimensions (E and F). Conversely, during the BW phase, the bounds expand to the input feature map dimensions (H and W) to accommodate the transposed convolution mapping. This parameterized bounding ensures that a single stationary control finite state machine (FSM) can orchestrate all three training stages, establishing the theoretical foundation for the optimized RS dataflow mapping.

3.2.2 Forward Softmax Approximation

The final operation in the forward pass is the classification head, which relies on the Softmax function to output a normalized probability distribution. For an input

Algorithm 3 Unified 7-Dimensional Nested Loop for CNN Training

Require: Input activations I , Weights W , Output gradients δO , Current Phase

Ensure: Output activations O (FW), Input gradients δI (BW),
or Weight gradients δW (WG)

```

1: Set spatial loop bounds  $Bound_Y$  and  $Bound_X$  based on Phase
   (FW/WG:  $E, F$ ; BW:  $H, W$ )
2: for  $n = 0$  to  $N - 1$  do                                ▷ Batch dimension
3:   for  $k = 0$  to  $K - 1$  do                                ▷ Output channel dimension
4:     for  $c = 0$  to  $C - 1$  do                                ▷ Input channel dimension
5:       for  $i = 0$  to  $Bound_Y - 1$  do                        ▷ Generic spatial height
6:         for  $j = 0$  to  $Bound_X - 1$  do                        ▷ Generic spatial width
7:           for  $r = 0$  to  $R - 1$  do                            ▷ Kernel height
8:             for  $s = 0$  to  $S - 1$  do                            ▷ Kernel width
               ▷ — Compute operations based on phase —
9:             if Phase == FW then
10:               Let  $e = i, f = j$ 
11:                $O[n, k, e, f] += I[n, c, e + r, f + s] \times W[k, c, r, s]$ 

12:             else if Phase == BW then
13:               Let  $h = i, w = j$ 
14:                $\delta I[n, k, h, w] += \delta O[n, k, h - r, w - s] \times W[k, c, r, s]$ 

15:             else if Phase == WG then
16:               Let  $e = i, f = j$ 
17:                $\delta W[k, c, r, s] += I[n, c, e + r, f + s] \times \delta O[n, k, e, f]$ 

```

vector $\mathbf{x}$ of length N , the standard Softmax is mathematically defined as:

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^N e^{x_j}}.$$

Directly evaluating this formula requires floating-point exponentiation and dynamic division, both of which are computationally expensive and fundamentally incompatible with the MAC-centric, fixed-point spatial architecture. To unify the classification head with the rest of the integer datapath, the Softmax computation is algebraically reconstructed into a four-step hardware-aware sequence [70]:

1. **Numerical Stabilization:** To prevent severe numerical overflow within finite-width hardware registers, the maximum value x_{max} is first extracted from the input vector. All elements are then shifted into the non-positive domain: $x'_i = x_i - x_{max}$. This normalization ensures that all subsequent exponentiations are bounded within the range $(0, 1]$, effectively eliminating the risk of overflow during intermediate calculations.
2. **Exponentiation Approximation:** Instead of evaluating the natural exponential $e^{x'_i}$ directly, the computation is converted to a base-2 representation using the identity $e^{x'} = 2^{x' \cdot \log_2(e)}$. This base-2 exponentiation is evaluated utilizing the Schraudolph approximation [71] (a bit-level estimation method), augmented by a piecewise polynomial correction for the fractional component. This approach replaces complex exponentiation with efficient bit-shifts and standard MAC operations.
3. **Reciprocal Approximation:** To avoid the area and latency overhead of a hardware divider, the normalization step is reformulated as a multiplication by the reciprocal of the total sum, $1/Sum$, where $Sum = \sum e^{x'_j}$. The reciprocal is estimated by first normalizing the sum into a representable range using a Leading Zero Counter (LZC), followed by a fixed-point iterative reciprocal estimation. This transformation reduces the dynamic division to a deterministic sequence of shifts and multiplications.
4. **Final Probability Output:** The final Softmax probabilities are obtained by multiplying the cached exponential results from Step 2 with the global reciprocal derived in Step 3:

$$Softmax(x_i) \approx e^{x'_i} \times \frac{1}{Sum}.$$

To mitigate quantization bias during truncation, a fixed rounding bias is incorporated into this final output stage.

Through this algorithmic restructuring, the entire Softmax computation is downgraded to basic MAC and shift primitives, ensuring that the classification head can be seamlessly integrated into the unified computational fabric.

3.2.3 Layer-wise Backpropagation Rules

Beyond convolutional layers, end-to-end training requires backpropagation through nonlinear activations, pooling layers, and the classification head. Before detailing the computational rules, it is necessary to address the BN layers present in the pre-trained model. Dynamic calculation of BN statistics during on-device fine-tuning under stringent batch size constraints often induces severe training instability. To circumvent this, the BN running statistics are assumed to be frozen during the edge fine-tuning phase, reducing the BN operation to a deterministic linear transformation. Following the standard parameter folding paradigms used in integer-arithmetic inference frameworks [5], the BN running statistics [43] are algebraically fused into the weights and biases of upstream convolutional layers *offline*, prior

to hardware deployment. This offline fusion allows the proposed hardware architecture to completely omit dedicated BN modules, and execute training directly on the fused convolutional parameters without altering the mathematical equivalence of the forward pass. For efficient hardware implementation, the gradient derivation rules for the remaining active layers are summarized below.

Softmax-CrossEntropy Fusion

The computation of the error gradient begins with the classification head. Calculating the derivatives of the Softmax function and the Cross-Entropy loss independently introduces complex Jacobian matrices and division operations, which is unfavorable for fixed-point hardware implementation. Fusing these two operations yields an algebraically simplified gradient. Let P represent the predicted probability distribution vector and Y represent the ground-truth one-hot label vector. The initial error gradient is given by

$$\delta I_i = P_i - Y_i.$$

This initial gradient δI naturally serves as the incoming error gradient δO for the preceding fully connected layer. This fusion removes the need for explicit exponentiation and division units, and simplifies the backpropagation initialization to a lightweight vector subtraction. From a hardware standpoint, this design is particularly advantageous, as it permits the classification head to be implemented using only a compact subtractor array.

FC Layer

Upon receiving gradients propagated from the loss function, the FC layer computes its input and weight gradients simultaneously. The operations are formulated as

$$\delta I = \delta O \cdot W^T,$$

$$\delta W = I^T \cdot \delta O.$$

Unlike convolutions, the FC backward pass is a bandwidth-intensive General Matrix Multiply (GEMM) with limited data reuse. We therefore handle it in a dedicated FC module for scheduling and bandwidth control.

Global Average Pooling

Unlike flattening operations, the GAP layer compresses the spatial dimensions into a 1×1 vector per channel. During forward pass, it computes the spatial mean; thus, during the backward pass, the incoming error must be uniformly distributed across the spatial window. For a feature map with spatial dimensions $H \times W$, the gradient is

$$\delta I(c, h, w) = \frac{\delta O(c)}{H \times W}.$$

Here, $\delta O(c)$ denotes the gradient corresponding to the channel c . From a hardware implementation perspective, GAP delivers superior memory efficiency. While the input feature maps are naturally buffered by the preceding convolutional layer,

the GAP operation itself eliminates the need for any dedicated activation or index caching during the forward pass. Its backward computation simplifies to scalar multiplication with the constant $1/(H \times W)$, followed by spatial broadcasting. Therefore, GAP incurs negligible computational and storage overhead.

Max Pooling

In the forward pass, only the maximum value within each pooling window is retained, and the corresponding positional indices are recorded. In the backward pass, the error gradient is solely propagated to the spatial position yielding the maximum value, while all remaining positions are assigned zero gradients:

$$\delta I(c, h, w) = \begin{cases} \delta O(c, h', w'), & \text{if } (h, w) \text{ is the argmax index} \\ 0, & \text{otherwise} \end{cases}.$$

This operator introduces a distinct hardware constraint: the spatial indices of the maximum values must be encoded and cached as a binary mask in the on-chip memory during the forward phase. During the backward phase, the hardware control logic fetches this mask to explicitly route the incoming gradients. Compared with dense arithmetic operators, max pooling is therefore dominated by indexing and routing logic rather than numerical computation.

ReLU

The ReLU activation function performs piecewise linear thresholding and is therefore particularly amenable to hardware implementation. Its backward propagation depends solely on the sign of the cached forward activation

$$\delta I(c, h, w) = \begin{cases} \delta O(c, h, w), & \text{if cached } I(c, h, w) > 0 \\ 0, & \text{otherwise} \end{cases}.$$

At the circuit level, this operation requires no multiplication or nonlinear transformation, which avoids complex arithmetic. The backward rule is realized by evaluating the sign bit of the cached activation I to drive a multiplexer (MUX), which either passes the gradient δO unmodified or grounds the output signal to zero.

Convolutional Layer

After passing through pooling and activation layers, the propagated error gradient flows back into the convolutional backbone, where the major computational workload of training is concentrated. According to the backward propagation and weight-gradient formulations, the data gradient is calculated via transposed convolution between δO and rotated weights W , while the weight gradient is derived from cross-correlation between cached input activations I and δO . Both computation phases can be inherently mapped to the proposed unified 7-dimensional nested loop model, enabling seamless integration of dataflow scheduling.

3.2.4 Weight Update

The training iteration ends with the Weight Update (WU) phase, which updates the trainable parameters using the gradients computed in the WG phase. To accommodate the fixed-point hardware constraints, this framework employs a shift-based Vanilla SGD algorithm (without momentum) equipped with residual error feedback.

Specifically, the learning rate is encoded as a power-of-two shift. Gradient scaling is implemented via two arithmetic right shifts: the first realizes the effective learning rate (incorporating batch averaging factors), and the second aligns the fractional bit formats between the gradients and parameters. The pre-shifted gradient is then accumulated with a state from a per-parameter residual memory to ensure that microscopic updates are not lost across iterations. To prevent sign-dependent offsets, all shift operations enforce a sign-aware truncation toward zero.

Finally, to prevent numerical overflow, the updated weight is subjected to a signed saturation function, which dynamically clamps the value to the representable fixed-point range.

Table 3.2: Mathematical Notations for the Unified Training Computation Model

Symbol	Description
$\mathbf{A}_l$	Activation output of layer l
$\mathbf{Z}_l$	Pre-activation (linear output) of convolution or fully connected layer l
$\mathbf{W}_l, \mathbf{b}_l$	Weights and biases of layer l
$\delta \mathbf{A}_l, \delta \mathbf{Z}_l$	Error gradients with respect to $\mathbf{A}_l$ and $\mathbf{Z}_l$
$\star$	Convolution
$\star^T$	Transposed convolution (used for backward data gradient computation)
$\circ$	Hadamard product (element-wise multiplication)
$g(\cdot)$	ReLU activation function
$g'(\cdot)$	Derivative of ReLU (1 if > 0 , else 0)
$\text{unpool}(\cdot)$	Argmax routing (routes pooled gradients back to maximum index positions)

3.2.5 End-to-End Computational Mapping

To consolidate the layer-wise propagation rules discussed throughout this section and explicitly map them onto the target lightweight CNN architecture, an end-to-end computational summary is established. First, Table 3.2 formalizes the mathematical notations adopted for the unified training computation model. Based on these definitions, Table 3.3 illustrates the complete forward and backward dataflow of the proposed partial fine-tuning strategy. This comprehensive mapping highlights the algorithmic integration of the Softmax-Cross Entropy fusion at the classification head, the offline algebraic fusion of Batch Normalization parameters, and, most crucially, the structural truncation of backpropagation at Conv3. By explicitly configuring the hardware to skip the data gradient computation for Conv3 (halting the gradient flow so that δA_2 is intentionally truncated), the algorithmic formulation directly aligns with the memory constraints of the spatial architecture, thereby validating the systemic feasibility of the on-device fine-tuning paradigm.

Table 3.3: Forward and Backward Dataflow for the Target Partial Fine-Tuning CNN

Layer	Forward Pass	Shape (Output)	Backward Pass
Input Data	$\mathbf{A}_0 = \text{Input Images}$	$\mathbb{R}^{N \times 3 \times 32 \times 32}$	N/A (End of backpropagation)
Conv1 (Frozen)	$\mathbf{Z}_1 = \mathbf{A}_0 \star \mathbf{W}_1 + \mathbf{b}_1$ $\mathbf{A}_1 = g(\mathbf{Z}_1)$	$\mathbb{R}^{N \times 32 \times 32 \times 32}$	Algorithmic Truncation (Frozen Layers)
Conv2 (Frozen)	$\mathbf{Z}_2 = \mathbf{A}_1 \star \mathbf{W}_2 + \mathbf{b}_2$ $\mathbf{A}_{2r} = g(\mathbf{Z}_2)$ $\mathbf{A}_2 = \text{maxpool}(\mathbf{A}_{2r})$	$\mathbb{R}^{N \times 32 \times 16 \times 16}$	
Conv3 (Trainable)	$\mathbf{Z}_3 = \mathbf{A}_2 \star \mathbf{W}_3 + \mathbf{b}_3$ $\mathbf{A}_3 = g(\mathbf{Z}_3)$	$\mathbb{R}^{N \times 64 \times 16 \times 16}$	
Conv4 (Trainable)	$\mathbf{Z}_4 = \mathbf{A}_3 \star \mathbf{W}_4 + \mathbf{b}_4$ $\mathbf{A}_{4r} = g(\mathbf{Z}_4)$ $\mathbf{A}_4 = \text{maxpool}(\mathbf{A}_{4r})$	$\mathbb{R}^{N \times 64 \times 8 \times 8}$	$\delta \mathbf{Z}_3 = \delta \mathbf{A}_3 \circ g'(\mathbf{Z}_3)$ $\delta \mathbf{W}_3 = \mathbf{A}_2 \star \delta \mathbf{Z}_3$ $\delta \mathbf{b}_3 = \sum \delta \mathbf{Z}_3$ $\delta \mathbf{A}_2 \rightarrow \text{Truncated}$
Conv5 (Trainable)	$\mathbf{Z}_5 = \mathbf{A}_4 \star \mathbf{W}_5 + \mathbf{b}_5$ $\mathbf{A}_5 = g(\mathbf{Z}_5)$	$\mathbb{R}^{N \times 128 \times 8 \times 8}$	$\delta \mathbf{A}_{4r} = \text{unpool}(\delta \mathbf{A}_4, \text{idx}_4)$ $\delta \mathbf{Z}_4 = \delta \mathbf{A}_{4r} \circ g'(\mathbf{Z}_4)$ $\delta \mathbf{W}_4 = \mathbf{A}_3 \star \delta \mathbf{Z}_4$ $\delta \mathbf{A}_3 = \delta \mathbf{Z}_4 \star^T \mathbf{W}_4$
Conv6 (Trainable)	$\mathbf{Z}_6 = \mathbf{A}_5 \star \mathbf{W}_6 + \mathbf{b}_6$ $\mathbf{A}_{6r} = g(\mathbf{Z}_6)$ $\mathbf{A}_6 = \text{maxpool}(\mathbf{A}_{6r})$	$\mathbb{R}^{N \times 128 \times 4 \times 4}$	$\delta \mathbf{Z}_5 = \delta \mathbf{A}_5 \circ g'(\mathbf{Z}_5)$ $\delta \mathbf{W}_5 = \mathbf{A}_4 \star \delta \mathbf{Z}_5$ $\delta \mathbf{A}_4 = \delta \mathbf{Z}_5 \star^T \mathbf{W}_5$
GAP (Trainable)	$\mathbf{A}_7 = \frac{1}{H \times W} \sum_{h,w} \mathbf{A}_6$	$\mathbb{R}^{N \times 128}$	$\delta \mathbf{A}_{6r} = \text{unpool}(\delta \mathbf{A}_6, \text{idx}_6)$ $\delta \mathbf{Z}_6 = \delta \mathbf{A}_{6r} \circ g'(\mathbf{Z}_6)$ $\delta \mathbf{W}_6 = \mathbf{A}_5 \star \delta \mathbf{Z}_6$ $\delta \mathbf{A}_5 = \delta \mathbf{Z}_6 \star^T \mathbf{W}_6$
FC (Trainable)	$\mathbf{Z}_8 = \mathbf{A}_7 \mathbf{W}_{fc} + \mathbf{b}_{fc}$	$\mathbb{R}^{N \times 10}$	$\delta \mathbf{A}_6(h, w) = \frac{1}{H \times W} \delta \mathbf{A}_7$ (Broadcast)
Softmax + CE Loss	$\mathbf{P} = \text{softmax}(\mathbf{Z}_8)$	$\mathbb{R}^{N \times 10}$	$\delta \mathbf{W}_{fc} = \mathbf{A}_7^T \delta \mathbf{Z}_8$ $\delta \mathbf{A}_7 = \delta \mathbf{Z}_8 \mathbf{W}_{fc}^T$
Weight Update (WU)	$\mathbf{W}_l \leftarrow \text{sat}(\mathbf{W}_l - \Delta \mathbf{W}_l)$ for trainable layers $l \in \{\text{Conv3, Conv4, Conv5, Conv6, FC}\}$		

3.3 Hardware-Aware Quantization Strategy

Translating the continuous computational graph into a resource-constrained hardware implementation requires strict numerical discretization and complexity simplification. To bridge the gap between the algorithmic model and the physical hardware, this section introduces a hardware-aware quantization and sparsification framework. In the context of low-precision on-device training, the representational format of each tensor determines the validity of fixed-point arithmetic, the feasibility of shift-based rescaling, and the dynamic range of intermediate values that can be propagated without overflow. Accordingly, it first establishes the precision baseline and the corresponding tensor mapping rules, which collectively specify the canonical bit-width assignments for weights, activations, and gradients. These definitions not only remove ambiguity from subsequent arithmetic derivations, but also provide the numerical boundary conditions required for the fully quantized training and sparsification techniques detailed in the following subsections.

3.3.1 Fixed-Point Precision Baseline

To ensure bit-accurate consistency between algorithmic simulation and hardware implementation, this work employs a unified fixed-point representation framework throughout the training pipeline. Fixed-point formats are denoted as $Q_{m,n}$, where the data word operates under a standard two's complement signed representation, consisting of 1 sign bit, m integer bits, and n fractional bits, resulting in a total width of $1 + m + n$ bits. Under this notation, the software model establishes a unified precision baseline, while the RTL implementation adopts hardware-specific refinements derived from this baseline, ensuring numerical consistency without enforcing identical bit-width configurations.

While the software baseline defines a uniform mathematical boundary, the hardware implementation introduces targeted bit-width optimizations to maximize area and memory efficiency. As listed in Table 3.4, the global allocation strategy is specified below.

- **Forward Parameters:** Both W and B are quantized to $Q_{1,7}$ (9-bit total), which physically matches the predefined BRAM port width configuration, ensuring maximum on-chip storage density.
- **Forward Feature Maps:** Both A and O are mapped to $Q_{5,12}$ (18-bit total). The allocation of 5 integer bits provides a sufficient dynamic range to prevent overflow during inter-layer variance accumulation, while the 18-bit total width aligns with standard DSP data paths.
- **Parameter Gradients ($\delta W, \delta B$):** A deliberate split is introduced between local computation and global storage. Within the MAC arrays, the high-precision accumulated gradient sums are truncated to a compact $Q_{0,11}$ (12-bit) format for local buffering, significantly reducing the local register area footprint. However, when these gradients are transferred back to

Table 3.4: Software Baseline vs. Optimized Hardware Precision Allocation ($Q_{m.n}$)

Definition: $Q_{m.n}$ consists of 1 sign bit + m integer bits + n fractional bits.

Tensor / Signal	Format ($Q_{m.n}$)		HW Allocation	
	SW	HW	Bit Breakdown	Total Width
Weights (W)	$Q_{1.7}$	$Q_{1.7}$	1 Sign, 1 Int, 7 Frac	9-bit
Biases (B)	$Q_{1.7}$	$Q_{1.7}$	1 Sign, 1 Int, 7 Frac	9-bit
Activations (A)	$Q_{5.12}$	$Q_{5.12}$	1 Sign, 5 Int, 12 Frac	18-bit
OFMAPs (O)	$Q_{5.12}$	$Q_{5.12}$	1 Sign, 5 Int, 12 Frac	18-bit
Param Grads ($\delta W, \delta B$) - <i>Local</i>	$Q_{6.11}$	$Q_{0.11}$	1 Sign, 0 Int, 11 Frac	12-bit
Param Grads ($\delta W, \delta B$) - <i>GLB</i>	$Q_{6.11}$	$Q_{6.11}$	1 Sign, 6 Int, 11 Frac	18-bit

Note: SW uses a single backward quantizer; Local/GLB split is HW-specific.

the GLB, they are mapped to the $Q_{6.11}$ (18-bit) format via hardware sign-extension. This ensures memory interface alignment and reuses the existing 18-bit physical data buses without structural overhead.

Overall, this precision specification establishes a bit-accurate correspondence between software simulation and hardware execution. Table 3.4 is the specification reference for all subsequent steps, including shift-based alignment, saturation, gradient quantization, and pruning experiments. Most importantly, it fixes the numerical boundary conditions under which the proposed low-precision training framework is evaluated, ensuring that the reported behaviors are directly transferable to the RTL design.

3.3.2 Hardware-Aware Quantization-Aware Training

While Post-Training Quantization (PTQ) enables efficient static inference on edge devices, it fundamentally lacks the capacity to recover accuracy degradation caused by post-deployment domain shifts, such as sensor aging or environmental illumination variations. Standard QAT supports model fine-tuning; however, it typically follows a "pseudo-quantized forward, full-precision backward" paradigm. Performing error backpropagation and parameter updates entirely in 32-bit floating-point (FP32) substantially exceeds the stringent memory capacity and multiplier logic constraints of edge FPGAs.

To achieve on-device fine-tuning, this section introduces a HA-QAT pipeline. Unlike standard QAT, which typically only restricts forward activations and weights, our approach strictly quantizes the backward gradient flow to simulate real RTL constraints. For brevity and alignment with algorithmic conventions, we will refer to this customized pipeline simply as our QAT baseline in the following sections. By enforcing strict fixed-point constraints during forward propagation and error backpropagation, and utilizing continuous shadow parameters for stable update

accumulation, it establishes a closed-loop algorithmic simulation of the underlying hardware dataflow.

During the forward pass, the QAT pipeline simulates the fixed-point arithmetic of the spatial hardware architecture. As defined by the unified precision baseline, floating-point weights and biases are dynamically quantized to fixed-point integers and clipped to the representable $Q_{1.7}$ range, with stochastic rounding enabled specifically during the training phase. To address the inherent non-differentiability of this step, a Straight-Through Estimator (STE) [72] is employed to explicitly define the backward gradients for the quantization operators. Following the simulated MAC operations, the computation mimics hardware behavior through right-shift alignment and saturation, truncating the high-precision accumulator outputs to align with the $Q_{5.12}$ activation bit-width. To simulate these hardware-level non-linear truncations without breaking the computational graph, the pipeline utilizes a gradient-detachment operation, effectively treating the truncation as an identity mapping in the backward pass.

The backward propagation phase extends quantization to the gradient domain, distinguishing our approach from standard QAT. The primary challenge here lies in quantization underflow. Error gradients are typically of microscopic magnitude; if directly hard-quantized, a substantial fraction of these gradients could underflow to zero, degrading the learning process. In our exploratory algorithmic simulation, this is mitigated by Quantization-Aware Scaling (QAS), a dynamic scaling mechanism that temporarily amplifies gradients before quantization, coupled with stochastic rounding to prevent directional bias. However, QAS is confined to serving as an algorithmic baseline to establish a theoretical upper bound for preventing underflow in software, rather than a deployed hardware mechanism. This distinction directly motivates the adoption of the more hardware-friendly sparsification strategies for RTL implementation.

Finally, the parameter update stage enforces physical hardware boundaries to close the training loop. During the algorithmic simulation, parameter updates are executed using standard Stochastic Gradient Descent (SGD) on continuous full-precision (FP32) parameters. While maintaining these parameters in FP32 allows the optimization process to smoothly accumulate microscopic gradient steps, updating them without physical constraints under the STE can lead to weight drift. Therefore, after each SGD step, these continuous parameters are explicitly projected back into their representable fixed-point bounds through a clamping operation. This critical projection preserves the training stability under quantization, ensuring that the continuous optimization trajectory remains consistent with the finite representational capacity of the physical hardware.

3.3.3 Fixed-Point Arithmetic and Saturation Rules

Given the QAT pipeline, it is necessary to formalize the numerical operators that map real-valued tensors onto the fixed-point domain. This section defines the foundational algebraic rules governing radix alignment and boundary saturation to establish a deterministic fixed-point execution model. By ensuring that all

rescaling factors are derived from fractional-bit mismatches, and that all intermediate states are projected into representable ranges, this mathematical framework provides a bridge between abstract algorithmic training and hardware-constrained implementation.

3.3.3.1 Derivation of Fixed-Point Alignment Parameters

For convolutional and fully connected layers, the forward data path operates exclusively in the integer domain. Let f_{in} , f_w , and f_b denote the fractional bits of the input activations, weights, and biases, respectively. Since the hardware accumulator aggregates the product of two fixed-point operands, the effective fractional precision of the accumulator naturally expands to

$$f_{acc} = f_{in} + f_w.$$

To normalize the accumulator output back to the target output precision f_{out} , the required arithmetic right-shift amount is deterministically derived as

$$s_{out} = f_{acc} - f_{out}.$$

Similarly, injecting the bias into the accumulator requires strict radix alignment. Because the bias typically possesses fewer fractional bits than the expanded accumulator, this dictates a relative left-shift amount of

$$s_b = f_{acc} - f_b,$$

ensuring the bias is correctly scaled before accumulation. These alignment parameters are not empirically tuned hyperparameters; they are analytically derived from the tensor quantization formats, thus determining the shift-based arithmetic during forward propagation.

3.3.3.2 Shift-Based Output Alignment for Conv/FC

Based on the above definitions, the fixed-point forward computation can be summarized as

$$\tilde{O} = \mathcal{S}_{s_{out}}(\mathcal{A}(I, W, B)),$$

where $\mathcal{A}(\cdot)$ denotes the integer MAC operation with bias injection, and $\mathcal{S}_{s_{out}}(\cdot)$ represents a power-of-two rescaling implemented via an arithmetic right shift. To avoid significant truncation bias, this rescaling explicitly executes a round-to-nearest with offset mechanism. Mathematically, it is formulated as:

$$\mathcal{S}_s(x) = (x + 2^{s-1}) \gg s,$$

where $\gg$ denotes the bitwise arithmetic right shift. This unified arithmetic formulation applies identically to both layer types, as the FC layer is mathematically equivalent to a 1×1 convolution over a flattened tensor.

3.3.3.3 Integer Division in Global Average Pooling

GAP follows the same fixed-point principle, but executes spatial reduction rather than MAC accumulation. For a feature map of spatial dimensions $H \times W$, the output for channel c is computed as

$$O(c) = \left\lfloor \frac{1}{H \times W} \sum_{h=0}^{H-1} \sum_{w=0}^{W-1} I(c, h, w) \right\rfloor,$$

where the division is implemented as integer normalization with rounding. Crucially, for the specific target network deployed in this work, the GAP layer operates exclusively on the 4×4 output of Conv6. Consequently, the pooling area is perfectly aligned to a power of two ($H \times W = 16 = 2^4$). This architectural constraint allows the hardware to bypass complex division logic or multiplier-shift approximations, replacing the denominator entirely with a deterministic 4-bit arithmetic right shift.

3.3.3.4 Saturation and Projection Constraints

Because all intermediate values reside in finite-width hardware registers, strict saturation policies are mandatory to prevent overflow. Let x_{min} and x_{max} define the representable bounds of a given quantization format. The clipping operator is defined as

$$\text{clip}(x) = \min(\max(x, x_{min}), x_{max}).$$

This bounding constraint is applied at three distinct stages: (i) quantizer clipping during the initial floating-point to fixed-point conversion; (ii) activation saturation applied to the outputs of Conv, FC, and GAP layers after right-shift alignment to prevent inter-layer overflow; and (iii) post-update parameter projection. To prevent unbounded weight drift caused by STE during backpropagation, the trainable parameters θ are explicitly projected back into their allowable bounds $Q_{1.7}$ after every optimization step:

$$\theta_{t+1} = \text{clip}(\theta_t - lr \cdot \nabla_{\theta_t} L)$$

This closed-loop projection guarantees bit-accurate consistency between the algorithmic training trajectory and the bounded RTL execution model.

3.3.4 Quantization and Sparsification in Training

Motivated by the QAT baseline, this section evaluates the impact of three algorithmic enhancements on numerical stability and hardware implementation overhead under edge distortion simulation through systematic ablation studies: stochastic rounding, quantization-aware scaling (QAS), and SAP. These techniques are not viewed as isolated training heuristics, but rather as hardware-relevant design choices from perspectives of unbiased quantization, gradient trainability, and backward sparsification.

3.3.4.1 Stochastic Rounding vs. Standard Rounding

In low-precision backpropagation, standard round-to-nearest quantization deterministically truncates microscopic, consistently signed gradients to zero, leading to long-term drift and biased weight updates. Drawing upon the principles of limited numerical precision training [20], stochastic rounding replaces this deterministic rule with a probabilistic mapping, where values are rounded up or down based on their proportional distance to the nearest representable integers, effectively suppressing directional bias, and preserving an approximately zero-mean error in expectation.

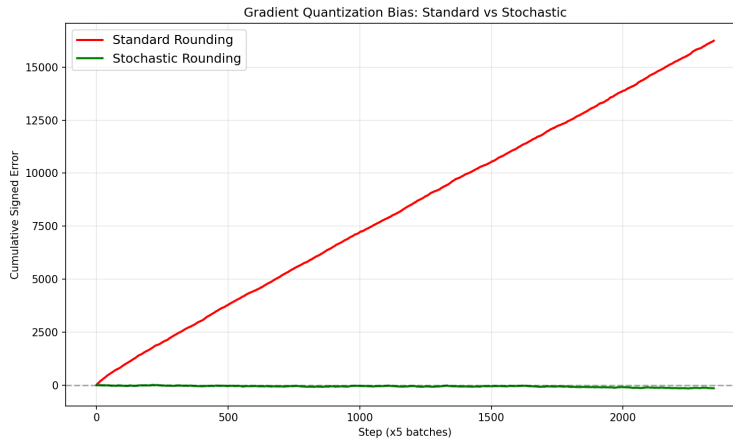


Figure 3.2: Cumulative signed error of gradient quantization over training steps.

As shown in Fig. 3.2, the ablation study over 30 epochs validates this theoretical advantage: standard rounding accumulates a large positive signed error, while stochastic rounding stays centered near zero. Table 3.5 further summarizes the quantization step size and the final accumulated error after training.

Table 3.5: Cumulative Signed Error of Gradient Quantization After 30 Epochs

Rounding Strategy	Cumulative Signed Error
Standard Rounding	+16248.6563
Stochastic Rounding	-142.4609

Note: Evaluated under $Q_{6.11}$ gradient precision (Resolution: 0.000488)

From a hardware perspective, generating probabilistic noise can be efficiently achieved by instantiating lightweight Pseudo-Random Number Generators (PRNGs), which are proposed to be realized via LFSRs alongside a compact comparison circuit. The projected logic overhead for such modules is minimal and highly

controllable within an FPGA fabric. Given its manageable hardware cost and indispensable role in preventing systemic training divergence, stochastic rounding is explicitly retained as the default quantization strategy for the on-device weight update phase.

3.3.4.2 Quantization-Aware Scaling

Inspired by the loss scaling techniques widely adopted in mixed-precision training [73], Quantization-aware scaling (QAS) aims to mitigate gradient underflow during backpropagation. Building upon dynamic fixed-point paradigms [74], it operates by dynamically extracting the maximum absolute value within a gradient tensor and applying an adaptive scaling factor prior to fixed-point truncation, artificially amplifying small gradients to fully utilize the available representational bandwidth. As shown in Fig. 3.3, the training and validation curves of QAS ON and QAS OFF remain close throughout the full training schedule, indicating that the method maintains optimization stability, but provides only a limited accuracy gain.

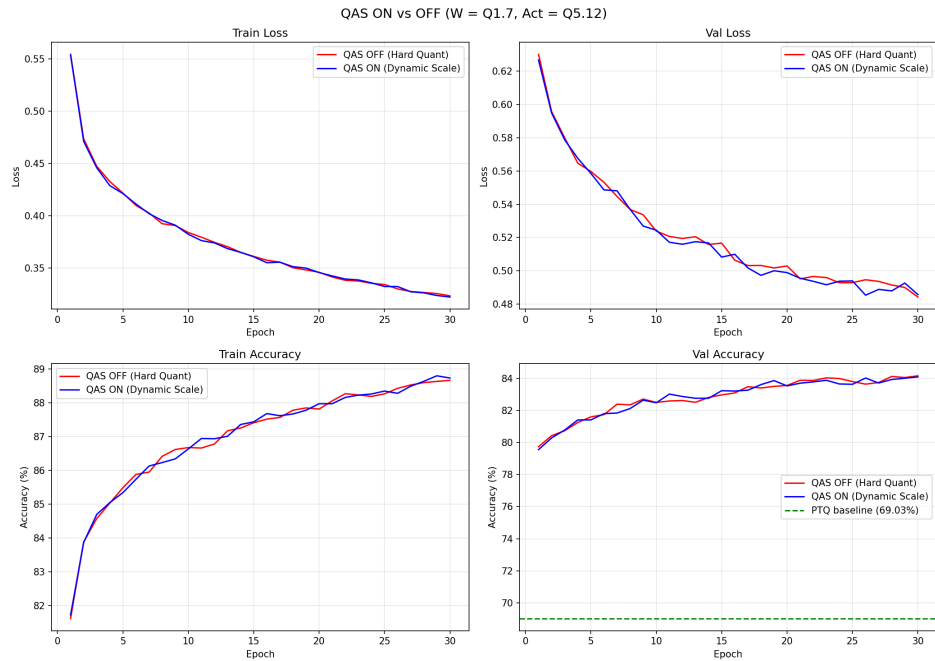


Figure 3.3: Training and validation trajectories comparing hard quantization (QAS OFF, red) and dynamic scaling (QAS ON, blue).

Table 3.6 reports the corresponding recovery results, while Table 3.7 summarizes the observed scaling statistics. Software telemetry indicates aggressive intervention, with QAS deploying an average scaling factor of $217.44\times$. However, the accuracy recovery logs reveal diminishing marginal returns: directly applying hard

quantization (QAS OFF) achieved an 84.15% validation accuracy, while the computationally intensive dynamic scaling (QAS ON) yielded an almost negligible improvement, peaking at 84.08%, with a noticeable runtime increase.

Table 3.6: Performance and Overhead Comparison: QAS ON vs. OFF

Mode	PTQ	Best	Recovery	Time
QAS OFF (Hard Quant)	69.03%	84.15%	+15.12%	1159s
QAS ON (Dynamic Scale)	68.84%	84.08%	+15.24%	1217s

Note: Evaluated under $Q_{1.7}$ weights/biases and $Q_{5.12}$ activations.

Table 3.7: QAS Dynamic Scaling Telemetry (Averaged Over Batches)

Metric	Value
Max Scale Factor Observed	2527.91×
Average Scale Factor (Mean)	217.44×
Median Scale Factor (P50)	11.47×
Min Scale Factor Observed	1.00×
Average Gradient Max	0.207385
Average Gradient Mean	0.040264

Mapping QAS to physical hardware introduces severe architectural bottlenecks, requiring a global extreme reduction tree across spatial arrays, followed by complex dynamic shifters or dedicated hardware multipliers/dividers. The prohibitive silicon footprint and latency costs completely outweigh the marginal accuracy gain. Consequently, while QAS serves as a theoretical reference in software, it is deprecated and excluded from the default RTL pipeline.

3.3.4.3 Stochastic Approximate Pruning

Motivated by foundational sparse backpropagation methodologies [75], Stochastic approximate pruning (SAP) addresses the dense computational burden of error backpropagation by structurally reshaping the gradient distribution. By applying a threshold-driven stochastic sparsification rule to small gradients, SAP maps a large fraction of low-magnitude updates to zero, while probabilistically boosting the most informative boundary gradients to the threshold to compensate for the truncated energy.

Algorithmically, SAP is embedded directly within the backward path of the STE, operating prior to QAS and final gradient quantization. To evaluate its theoretical efficacy under edge fine-tuning scenarios, this algorithmic baseline applies SAP in the continuous (floating-point) domain. Accordingly, the pruning operation is

confined to the backward convolution dataflow, exclusively targeting the propagated activation gradients while leaving the forward pass computations entirely dense and unaffected. To circumvent the prohibitive $\mathcal{O}(N \log N)$ computational overhead required by exact gradient sorting, the implementation utilizes a highly efficient $\mathcal{O}(N)$ threshold estimator. Under the assumption of a zero-mean Gaussian distribution, the standard deviation of the gradient tensor is approximated from the Mean Absolute Error (MAE):

$$\hat{\sigma} = \sqrt{\frac{\pi}{2}} \cdot \frac{1}{N} \sum |g|.$$

The global pruning threshold τ is then analytically derived from the target sparsity via the inverse error function. Once τ is established, each continuous gradient g is independently transformed using a stochastic routing rule governed by a uniform random variable $r \sim \mathcal{U}(0, 1)$:

$$g_{\text{SAP}} = \begin{cases} g, & \text{if } |g| \geq \tau \\ \text{sgn}(g) \cdot \tau, & \text{if } |g| < \tau \text{ and } r < \frac{|g|}{\tau} \\ 0, & \text{otherwise} \end{cases}.$$

Following the principles of unbiased gradient sparsification established in distributed optimization [76], this stochastic compensation mechanism preserves the unbiased expectation of the gradient (i.e., $\mathbb{E}[g_{\text{SAP}}] = g$). This mathematical property ensures that the optimization trajectory remains directionally accurate, enabling stable convergence even when extreme structural sparsification is enforced.

Empirical analysis of the gradient distribution (Fig. 3.4 and Table 3.8) reveals a profound structural shift under SAP: at a target sparsity threshold of 0.99, the proportion of exact zero-gradients surges from a baseline of 0.52% to 70.4%.

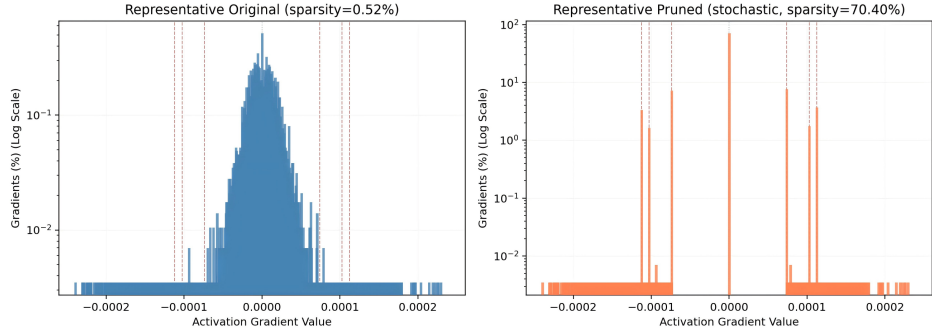


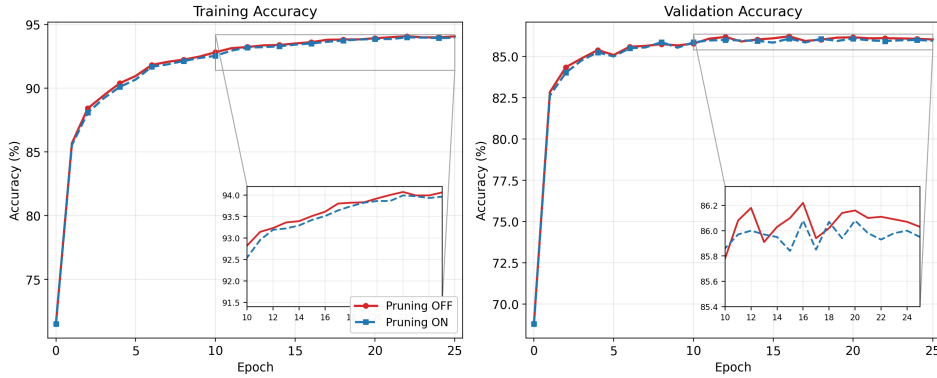
Figure 3.4: Gradient distribution before and after applying SAP. The peaks in the pruned distribution layer correspond to the threshold values $\pm\tau$ of each convolution layer.

Remarkably, despite this aggressive stripping of the backward update workload, the model’s recovery trajectory remains highly stable (Fig. 3.5). As detailed in Table 3.9, the final validation accuracy incurs a negligible penalty of less than

Table 3.8: Gradient Distribution and STE Internal Statistics (1 sample, Target Sparsity=0.99)

Metric	Pruning OFF	Pruning ON
Total elements	72,704	72,704
Zero gradients	378	51,183
Sparsity	0.52%	70.4%
Non-zero count	72,326	21,521
<i>STE Internal Stats (Pruning ON Breakdown)</i>		
$ g \geq \text{thr}$ (kept)	—	2,815 (3.87%)
$ g < \text{thr} \rightarrow \pm \text{thr}$ (boosted)	—	18,706 (25.73%)
$ g < \text{thr} \rightarrow 0$ (pruned)	—	51,183 (70.4%)

0.2% (+17.41% vs. +17.27% recovery). This recovery behavior is measured under a partial fine-tuning setting, where only the last four convolutional layers and the fully connected classifier are updated. This demonstrates that SAP can unlock massive computational sparsity for hardware acceleration with virtually no degradation in edge fine-tuning performance.

**Figure 3.5:** Training and validation accuracy trajectories comparing dense backpropagation (Pruning OFF, red) with highly sparse backpropagation (Pruning ON, blue).

From a hardware standpoint, the high zero-gradient ratio exposed by SAP directly translates into opportunities for zero-skipping, MAC bypass, and reduced memory traffic. Unlike QAS, which mainly improves numerical robustness but was deprecated due to overhead, SAP provides a concrete path toward compute and bandwidth reduction. Delivering computational compression with negligible accuracy degradation, SAP is retained and prioritized as a core mechanism for architectural energy-efficiency optimization.

Table 3.9: SAP Pruning Accuracy and Edge Recovery Comparison

Metric	Pruning OFF	Pruning ON
PTQ Baseline Accuracy	68.81%	68.81%
Best Validation Accuracy	86.22%	86.08%
Recovery Gain	+17.41%	+17.27%

Note: Evaluated under a target SAP threshold of 0.99.

3.4 Dataflow and Tensor Reuse Analysis

With the fixed-point training computations numerically defined, the next challenge is to efficiently schedule the unified 7-dimensional nested loops onto a two-dimensional spatial architecture. At this stage, the design focus shifts from numerical representation to the structural regularities of computation, including data movement, loop transformations, and spatio-temporal tensor reuse. This section investigates the dataflow mapping strategies required to maximize on-chip data reuse and minimize energy-intensive off-chip memory accesses. To support a coherent mapping between algorithmic behavior and hardware resources, it first introduces the fundamental abstraction of the target computation, from which the subsequent dataflow transformations and resource-mapping constraints can be systematically derived.

3.4.1 Loop Transformations

As established in the unified 7D nested loop model, on-device training requires mapping high-dimensional tensor computations onto a resource-constrained 2D PE array. To preserve utilization while avoiding memory stalls, a hardware-aware loop transformation methodology is proposed. This section abstracts the underlying hardware execution into three core dataflow mapping strategies: adaptive spatial unrolling, block-wise iteration space reordering, and compute-in-read index mapping.

3.4.1.1 Adaptive Spatial and Channel Unrolling

The structural dimensions of feature maps vary significantly across different network layers. A rigid loop unrolling strategy would inevitably lead to severe PE underutilization. Therefore, an adaptive unrolling mechanism is formulated to dynamically partition the computation workload.

As illustrated in Fig. 3.6, for shallow layers with large spatial dimensions and shallow channels (e.g., $32 \times 32 \times C_{in}$), the iteration space is aggressively unrolled along the spatial dimensions (4-way unrolling). Distinct horizontal tiles are dis-

tributed across all four PE clusters to process the same input channel concurrently. Conversely, as the network goes deeper, spatial resolution decreases while channel depth increases. The mapping transitions to a hybrid mode (2-way unrolling) and eventually to pure channel-level parallelism (1-way spatial unrolling) for deeper 8×8 and 4×4 feature maps, where each PE cluster operates on an independent channel. This adaptive spatial-to-channel transition ensures optimal array utilization across the entire network architecture.

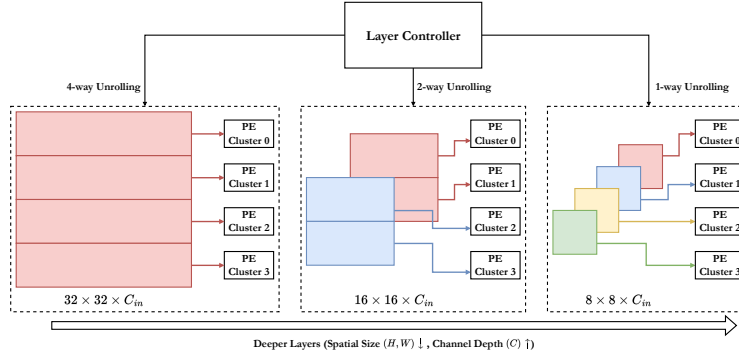


Figure 3.6: Adaptive spatial and channel unrolling strategy driven by the Layer Controller. As the network goes deeper, the mapping smoothly transitions from pure spatial parallelism (4-way) to hybrid (2-way) and channel parallelism (1-way) to maintain maximum utilization of the 4 PE Clusters.

3.4.1.2 Iteration Space Reordering and Halo Dependency Resolution

Tiling the spatial dimensions breaks the continuous memory access patterns, introducing isolated sub-iteration spaces. A critical methodological challenge in this block-wise iteration is the cross-tile data dependency caused by the overlapping receptive fields of convolutional kernels. As shown in Fig. 3.7, computing a 3×3 kernel at the upper boundary of Tile 1 requires access to the bottom rows logically belonging to Tile 0. Instead of redundantly fetching these overlapping input rows from the global memory, the iteration space is reordered to support a halo dataflow mechanism. The overlapping input boundary rows buffered during the processing of Tile 0 are intercepted and streamed directly to Tile 1 via an on-chip Halo FIFO, while the absolute outer boundaries of the global tensor are dynamically padded. This methodology truncates redundant global memory access, localizing data movement to adjacent computational clusters.

3.4.1.3 Compute-in-Read Index Transformation for Geometric Duality

During backpropagation, computing the data gradient mathematically equates to a transposed convolution, which requires a 180° geometric rotation of the weight

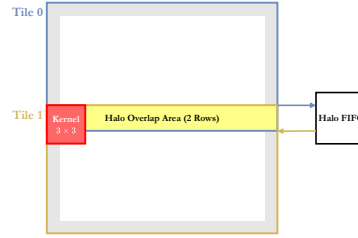


Figure 3.7: Halo dataflow mechanism and cross-tile dependency resolution.

kernel and a transposition of the input and output channels. Performing these transformations physically in memory incurs prohibitive latency and energy overheads. To address this, a "compute-in-read" index transformation strategy is introduced. As depicted in Fig. 3.8, the hardware physically preserves the forward weight memory layout. Assuming the 2D weight kernel is flattened into a contiguous 1D memory space, instead of moving data, the 180° rotation is equivalently achieved by manipulating the read pointer of the address generator: shifting the start address to the end of the kernel and toggling the stepping direction from an increment (+1) to a decrement (-1). Coupled with an algorithmic loop exchange between input and output channels, this zero-cost affine index transformation resolves the geometric duality, allowing the backward pass to seamlessly reuse the forward MAC arrays without additional structural overhead.

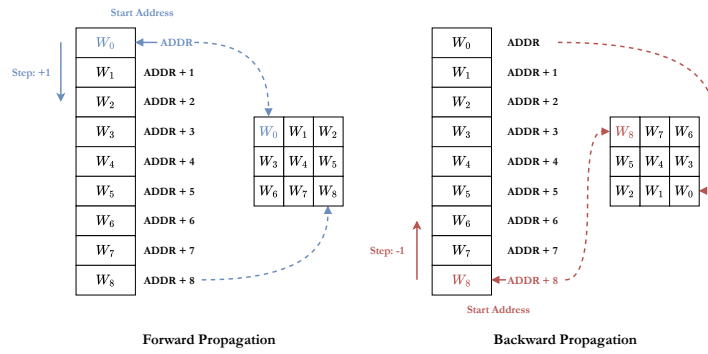


Figure 3.8: Compute-in-read index transformation for geometric duality.

3.4.2 Tensor Reuse Formulation

In resource-constrained edge environments, the energy cost of memory access inherently dominates the total training overhead. To lay a mathematical foundation for the subsequent hardware mapping, it is imperative to formalize the theoretical

limits of data reuse. The tensor reuse factor (R) is fundamentally defined as the ratio of the total number of MAC operations utilizing a specific tensor element to the memory access cost required to fetch it from the highest-cost memory tier (e.g., off-chip DRAM or the on-chip GLB). Based on the unified training computation model, the data flow can be categorized into three theoretical bounds: weight reuse (R_W), activation reuse (R_A), and gradient reuse (R_G).

To map these theoretical limits onto a spatial architecture, we establish a two-tier spatio-temporal abstraction space. This space comprises a logical local domain and a shared distribution domain. Temporal reuse is executed within the logical local domain, where data is iteratively accessed over time without additional global memory transactions. In contrast, spatial reuse is realized within the shared distribution domain, where data elements are logically replicated and broadcast across multiple parallel computation nodes in the spatial dimensions.

Let a computational tile feature an input activation spatial dimension of $H_t \times W_t$ and a convolutional kernel of size $K \times K$. Within a single computational block, the theoretical upper bounds for intra-tile reuse are analytically formulated as follows. For the weight tensors, since they are shared across the entire spatial map of the activation tile, the reuse factor is determined by the output dimensions:

$$R_W \approx (H_t - K + 1)(W_t - K + 1)$$

For the activation tensors, under the assumption of a unit stride ($stride = 1$), the dense overlapping of sliding windows yields a theoretical reuse limit of:

$$R_A \approx K^2$$

Similarly, during the weight gradient computation phase, the incoming error gradients (δO) participate in cross-correlation operations, exhibiting an isomorphic reuse property:

$$R_G \approx K^2$$

Furthermore, by introducing spatial unrolling dimensions, denoted as X_{dim} and Y_{dim} , to represent the algorithmic parallelism mapped onto a logic array, the overall global reuse achieves a multiplicative amplification effect. The specific scaling factors depend dynamically on the orthogonal broadcasting axes assigned by the spatial dataflow scheduling. Consequently, the global reuse factors scale as:

$$R'_W = R_W \cdot D_W, \quad R'_A = R_A \cdot D_A, \quad R'_G = R_G \cdot D_G,$$

where D_W, D_A , and $D_G \in \{X_{dim}, Y_{dim}\}$ represent the corresponding spatial broadcast dimensions for each tensor.

Because global memory accesses dominate the overall energy profile, let E_{mem} denote the base unit energy cost of such a memory transaction. The total memory access energy E_{total} exhibits a proportional relationship with the inverse of the global reuse factors, approximated as:

$$E_{total} \propto \text{MACs} \cdot E_{mem} \cdot \left(\frac{1}{R'_W} + \frac{1}{R'_A} + \frac{1}{R'_G} \right)$$

From this analytical model, it becomes evident that maximizing spatio-temporal reuse to approach the theoretical limits of R'_W , R'_A , and R'_G is the fundamental pathway to minimizing on-device training energy consumption. This observation directly determines the spatial scheduling logic, motivating the adoption of a specialized dataflow paradigm, which will be formalized in the next section as the RS dataflow for training.

3.4.3 Row-Stationary Dataflow for Training

To approach the theoretical reuse limits (R'_W , R'_A , and R'_G), the multidimensional training loops must be structurally mapped onto a 2D spatial computing array. The Row-Stationary (RS) scheme, originally formalized for spatial energy-efficient acceleration [77], is adopted here to enforce the local residency of a specific tensor while streaming the remaining tensors along designated array axes, thereby converting repetitive memory accesses into natural spatio-temporal reuse. It is imperative to note that this “residency” refers to the logical mapping and extended lifespan of data within the existing local registers of the Processing Elements (PEs), and does not imply any physical replication of hardware resources or redundant global storage. This schedule is intentionally training-aware and supports forward propagation, backward data gradients, and weight gradient formation within a unified abstraction.

3.4.3.1 Forward Propagation: Maximizing Weight and Activation Reuse

During forward propagation, as illustrated in Fig. 3.9(a), the standard 2D convolution is spatially decomposed into 1D primitives. The dataflow maximizes efficiency through three distinct mapping strategies:

- **Weight Stationary:** The convolutional kernel is split by row. Each kernel row is logically pinned within a PE, maintaining absolute residency in the local register while the input feature map elements are progressively streaming through it. This row-wise sharing across the horizontal array axis inherently maximizes temporal weight reuse.
- **Activation Reuse:** Due to the overlapping nature of sliding windows, identical activation rows are repeatedly consumed by adjacent output positions. In the 2D PE array, this sharing is geometrically routed via a diagonal multicast strategy. By streaming data along the anti-diagonals, a single activation row simultaneously serves multiple parallel PEs, fully exploiting spatial reuse.
- **Partial Sum (psum) Reuse:** To alleviate the output bandwidth pressure, the intermediate 1D results generated by different kernel rows are aggregated vertically through spatial reduction. Furthermore, for deep layers with massive channel dimensions, cross-channel psums are temporally accumulated within the PEs, effectively preventing intermediate read-modify-write traffic from accessing the global memory.

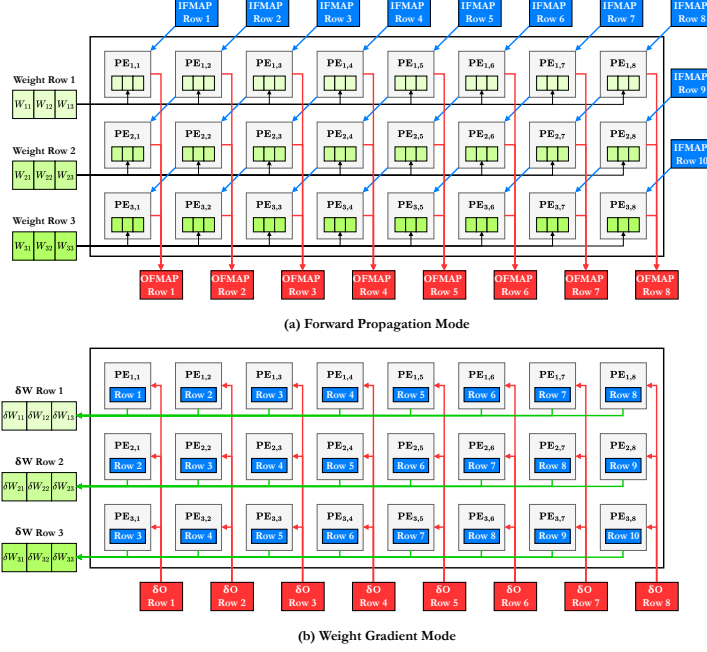


Figure 3.9: Spatio-temporal mapping of the Row-Stationary dataflow onto a 2D logical array.

3.4.3.2 Training Phase Reconfiguration: Maximizing Gradient Reuse

The most critical requirement during training lies in generating weight gradients without duplicating the compute fabric:

$$\delta W = I \star \delta O$$

As depicted in Fig. 3.9(b), this is resolved through a dynamic orthogonal remapping of the dataflow topology.

In this phase, the computational roles invert: the input activations resident in the local domain replace the weights as the stationary elements within the PEs (Activation Stationary). Concurrently, the error gradients (δO) act as the streaming tensors. Crucially, the spatial distribution network dynamically pivots by 90° , transitioning from a horizontal weight broadcast to a vertical gradient broadcast. This orthogonal reconfiguration ensures that identical δO slices are spatially multicasted to multiple parallel PEs along the array columns. By multiplying these vertically broadcast gradients with their respective cached activation histories, each PE computes a local weight gradient fragment. These fragments are subsequently aggregated via horizontal spatial reduction across the PE rows to produce the final weight gradient components (δW). This mechanism seamlessly reuses the forward datapath while maximizing spatial gradient reuse.

3.4.3.3 Spatio-Temporal Folding and Phase Unification

While the RS dataflow theoretically maximizes reuse, the practical scale is constrained by the physical dimensions of the PE array (e.g., 8×3 in this design). When tensor dimensions exceed the available hardware span, such as high channel depths or large feature maps, the dataflow scheduler employs tile-based folding. High-dimensional tensors are algorithmically partitioned into optimal tiles and executed iteratively over multiple temporal rounds. This spatio-temporal folding preserves the underlying RS principles while serializing the theoretical reuse across discrete temporal phases to accommodate hardware limitations.

Taken together, the RS dataflow establishes a unified scheduling framework that aligns the distinct training phases under a singular architectural abstraction. Forward propagation leverages weight stationary and activation diagonal multicast; backward propagation for data gradients is facilitated by index-driven geometric weight rotation; and weight-gradient computation is defined by activation residency and orthogonal gradient broadcast. By dynamically alternating stationary operands and streaming axes, the dataflow effectively unifies the execution of the complete training cycle.

3.5 Chapter Summary

This chapter has established the methodological foundation for the proposed on-device CNN fine-tuning framework. Beginning with the task definition and target network configuration, it delineated a hardware-aware partial fine-tuning strategy tailored to stringent FPGA resource constraints. The training computation was subsequently abstracted into a unified 7-dimensional nested-loop model, seamlessly integrating forward propagation, data gradient backpropagation, and weight gradient computation within a singular mathematical abstraction, which is then finalized by a hardware-aware weight update phase. Building upon this model, the chapter formalized a HA-QAT pipeline, defining fixed-point representations and deterministic arithmetic rules for shift-based rescaling and boundary saturation. In addition, several training-oriented numerical enhancement techniques were evaluated, including stochastic rounding, quantization-aware scaling, and SAP, with emphasis placed on their algorithmic roles and hardware implications.

Finally, by analyzing theoretical tensor reuse bounds and formulating hardware-aware loop transformations, this chapter bridged the semantic gap between algorithmic dataflow and spatial hardware mapping. Collectively, these methodological design choices define the numerical, computational, and structural constraints that guide the hardware realization in the next chapter. Chapter 4 will therefore focus on translating these algorithmic abstractions into an efficient RTL implementation, detailing the PE organization, memory hierarchy, and RS dataflow scheduling.

Hardware Architecture Design

4.1 Overall Architecture

This design is implemented on a Zynq-series FPGA platform that integrates a PS and PL. In this work, the PS is used for functional verification and system management: it transfers input images and configuration parameters to the PL, while the accelerator in the PL executes the main computation and on-chip training. The detailed interaction between the PS and PL will be discussed in Chapter 5.

At the system boundary, the **TOP_wrapper** provides the hardware–software interface on the PL side. It receives PS-issued control and configuration signals, maintains batch-level execution status, and dispatches synchronized strobes to the internal controller and datapath. In particular, it separates the triggers for single-sample execution (**i_start**) and parameter update (**i_start_update**), and supports mode selection between inference and training via **i_mode**. This separation allows the PS to define batch boundaries explicitly, while the PL side manages per-sample execution locally.

During training, the wrapper tracks a sample-active state and increments an internal batch counter when a sample completion pulse (**o_done**) is observed. The first-sample flag derived from this counter is forwarded to the datapath to select the gradient strategy (overwrite vs. accumulation). The wrapper itself performs no gradient computation; it only provides boundary-aligned control signals. In inference mode or upon update triggers, batch-related state is synchronously cleared to ensure safe mode transitions.

Below the wrapper, the overall architecture of the accelerator is organized into five functional modules (Fig. 4.1): the control module, data buffer module, convolution module, data post-processing module, and functional operation module. Their primary roles are summarized as follows:

a. Control module: The control module is responsible for the overall scheduling of the accelerator. It receives control information from the PS, configures the working mode and layer parameters, and coordinates the execution order of different computation and data-transfer stages.

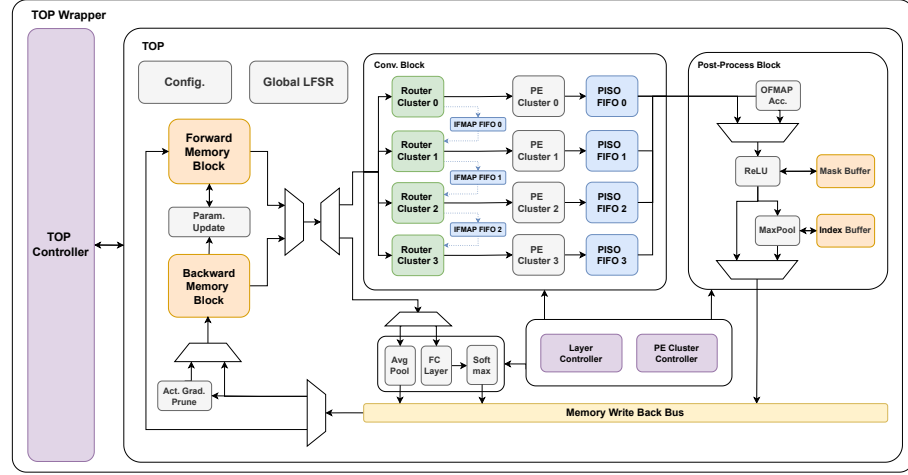


Figure 4.1: Accelerator overall architecture.

b. Data buffer module: The data buffer module stores activations, weights, biases, intermediate feature maps, gradients, and additional data such as pooling indices and ReLU masks. It provides on-chip data reuse and supplies data to the computation modules during both forward and backward propagation.

c. Convolution module: The convolution module is the main computation unit of the accelerator. It is composed of PE clusters, data routers, and Parallel-In Serial-Out First-In First-Out (PISO FIFO) output buffers. The PE clusters perform the main convolution operations, while the routers and FIFOs manage the data movement between GLBs, PE outputs, and local buffers. Under different control configurations, this module is reused for both forward and backward propagation.

d. Data post-processing module: The data post-processing module collects and rearranges the outputs generated by the PE clusters. It includes output feature-map gathering, partial-sum accumulation in the gather stage. The processed feature maps or gradients are then written back to the corresponding on-chip buffers.

e. Functional operation module: The functional operation module implements non-convolutional operations in the network, including LFSR, ReLU, pooling, fully-connected computation, softmax, activation gradient pruning, and parameter update. These modules work together with the convolution module to complete the full on-chip training flow.

4.2 Controller Module

The computation flow of the accelerator is managed by three main controllers: the top controller, the layer controller, and the PE cluster controller. These controllers are responsible for the internal execution schedule of the accelerator, including layer switching, parameter configuration, and PE-level computation control.

The top controller is the main finite state machine of the accelerator, as shown in Fig. 4.2. It determines the overall execution order of the network according to the current working mode. In inference mode, the controller schedules the forward path from convolution layer to average pooling and the fully-connected layer. In the forward path, the ReLU and max-pooling operations are included in the corresponding convolution-layer states rather than being scheduled as separate top-level states.

In training mode, after the forward path is completed, the controller continues to schedule the backward stage, including the fully-connected layer, average pooling, max pooling, ReLU, and convolution. During execution, it generates the start signals for different computation stages and waits for the corresponding done signals before moving to the next state. In this way, the top-level controller provides a unified control flow for both inference and training.

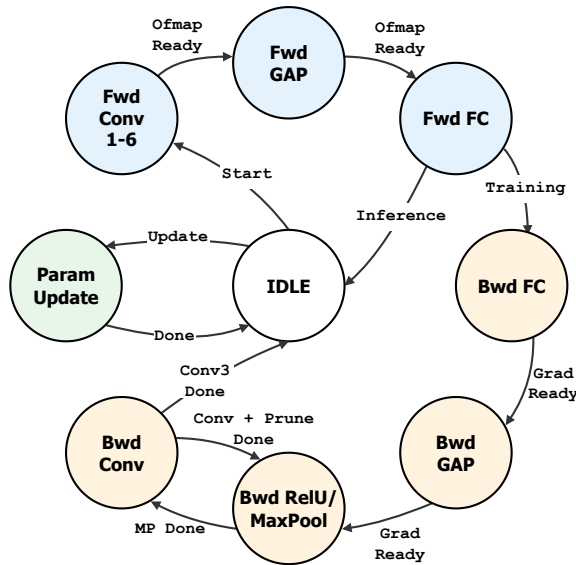


Figure 4.2: Top controller state transition diagram.

The layer controller converts the current layer state into hardware configuration signals. For each layer, it provides the base addresses of feature maps, weights, and biases. It also configures the feature-map size and convolution unroll mode.

During forward propagation, it follows the layer order from shallow layers to deep layers. During backward propagation, it traverses the trainable convolution layers in reverse order and provides the required parameters for gradient computation. Therefore, the layer controller connects the network-level execution flow with the datapath configuration.

The PE cluster controller manages the fine-grained execution of convolution computation. It controls weight and activation loading, starts the PE computation, and updates the input-channel and output-channel. It also generates control signals such as buffer switching and first/last input-channel flags to support correct partial-sum accumulation. The same controller is reused for forward and backward propagation.

4.3 Global Buffer Module

This section details a customized on-chip storage architecture designed to alleviate the massive memory-bandwidth bottleneck of backward propagation under the strict resource constraints of edge FPGA platforms. Guided by the dataflow abstraction established in Chapter 3, this architecture implements a highly parameterized global memory hierarchy to efficiently buffer forward tensors, error gradients, and auxiliary training control states.

4.3.1 On-Chip Storage Mapping

To mitigate the energy cost of off-chip memory traffic during edge training, the accelerator adopts a full on-chip data residence strategy. Within each forward-backward iteration (or a single inference pass), all critical tensors, including weights, biases, intermediate activations, and backpropagated gradients, remain inside the PL domain utilizing Block RAM (BRAM) and UltraRAM (URAM) resources. The PS and external DDR memory are therefore outside the real-time compute loop; they participate only during network initialization (loading inputs and pre-trained weights) and at epoch boundaries (retrieving metrics, loss values, and updated parameters).

All on-chip GLB dimensions and local storage parameters are centralized in the `config_pkg` package to ensure deterministic routing and scalable synthesis. The detailed physical buffer mapping is summarized in Table 4.1.

Rather than relying on static hard-coded constants, all buffer address widths are dynamically inferred at compile time via:

$$\text{Addr_Width} = \lceil \log_2(\text{Depth}) \rceil = \text{\$clog2}(\text{Depth}).$$

This parameterization minimizes the physical wire count and fan-out of inter-cluster address buses. For non power-of-two depths, it ensures a compact addressing space with a small number of unused addresses, avoiding over-provisioned multiplexer (MUX) structures and easing FPGA routing congestion.

Table 4.1: On-Chip Buffer Configurations and Address Quantization Summary

Buffer Name	Associated Tensor / Signal	Data Width	Depth	Addr Width	Total Bits
Weight GLB	Forward Weights (W , Conv & FC)	9-bit	287,840	19-bit	2,590,560
Activation GLB	Input Activations (A , iact)	18-bit	3,073	12-bit	55,314
Bias GLB	Forward Biases (b)	9-bit	458	9-bit	4,122
OFMAP GLB	Output Feature Maps (O , ofmap)	18-bit	71,818	17-bit	1,292,724
Weight Grad GLB	Weight Gradients (δW)	18-bit	277,760	19-bit	4,999,680
Bias Grad GLB	Bias Gradients (δB)	18-bit	394	9-bit	7,092
OFMAP Grad GLB	Activation Gradients (δO)	18-bit	32,768	15-bit	589,824
ReLU Mask Buffer	ReLU Activation States	1-bit	114,688	17-bit	114,688
MaxPool Index Buffer	MaxPool Quadrant Indices	2-bit	14,336	14-bit	28,672
OFMAP Buffer	PE Local Partial Sums	44-bit	1,024	10-bit	45,056

4.3.2 Auxiliary Training Buffers

A prominent architectural difference between on-device training and inference-only engines lies in the mandatory preservation of forward intermediate states to resolve mathematical dependencies during the backward propagation (BP) phase. Caching full-precision activations (e.g., 18-bit feature maps) across all trainable layers would impose a prohibitive on-chip memory footprint for edge FPGAs. To mitigate this, the design introduces two compact auxiliary state buffers: the ReLU Mask Buffer and the MaxPool Index Buffer.

For ReLU layers, the backward path depends only on the sign of the cached forward activations. Therefore, instead of storing the complete 18-bit activation stream (**A_DATA_WIDTH** = 18), the hardware records a 1-bit boolean mask (**RELU_MASK_WIDTH** = 1) indicating whether each element is positive (high level) or zero (low level). Configured with a total depth of 114,688 words (**RELU_MASK_DEPTH** = 114688), this dedicated mask scheme requires a 114,688 bits of storage. Evaluated against the mathematical baseline of caching the full 18-bit activation stream, which corresponds to a storage retention of only $\frac{1}{18} \approx 5.56\%$, this auxiliary tracking technique minimizes the local on-chip buffer footprint for backward ReLU gradient derivation by 94.44%. During backpropagation, the layer control logic fetches this 1-bit mask to directly drive a bank of MUXes, either passing the incoming activation gradient unmodified or zeroing it, without extra arithmetic.

For max-pooling, the backward path routes gradients back to the maximum-value element selected in the forward pass within each 2×2 window. The pooling module therefore emits a compact 2-bit index (**MAXPOOL_IDX_WIDTH** = 2) that encodes the quadrant of the maximum value and stores it in a dedicated buffer of depth **MAXPOOL_IDX_DEPTH** = 14336. When backpropagation begins, the controller retrieves this index to drive routing MUXes, sending the gradient to the selected maximum position and forcing the other three positions to zero. This mechanism eliminates recalculation and sorting operations in the backward pooling computa-

tion flow.

Importantly, the logging and tracking of these auxiliary control states occur in parallel with the forward streaming datapath. Since the mask and index extractions are embedded as lightweight monitor logic attached to the primary computation streaming pipelines, they do not stall or degrade forward throughput. This approach replaces dense tensor caching with low-bitwidth control metadata, enabling training-capable behavior under tight on-chip memory budgets.

4.3.3 Automated Bit-Width Derivation

To prevent overflow and precision loss in multi-stage fixed-point accumulation, the RTL adopts compile-time bit-width derivation in `config_pkg` rather than hard-coded empirical widths. Internal datapath widths are computed from structural parameters so that arithmetic growth is accommodated deterministically.

Within the PE MAC pipeline, multiplying an 18-bit activation by a 9-bit weight produces a 27-bit product. To support inner-product accumulation across channels without truncation, the RTL defines the local accumulation width as:

$$P_DATA_WIDTH = 2 \times A_DATA_WIDTH + \$clog2(32).$$

Given the baseline hardware configuration of $A_DATA_WIDTH = 18$, the internal datapath P_DATA_WIDTH is automatically derived as $2 \times 18 + 5 = 41$ -bit.

When partial sums are recombined across the spatial array in the horizontal dimension (X_{dim}) during gather/reduction, the bus width is further extended by:

$$SUM_WIDTH = P_DATA_WIDTH + \$clog2(X_{dim})$$

Substituting the physical structural parameter $X_DIM = 8$ ($\$clog2(8) = 3$), the final accumulation width becomes $41 + 3 = 44$ -bit, which directly defines the local `OFMAP Buffer` width and the gather datapath.

This parameterized derivation allows top-level precision choices to propagate automatically into internal add-tree and buffer widths, reducing manual retuning while keeping fixed-point behavior consistent between RTL and software models, and ensuring maximum utilization efficiency of FPGA lookup tables (LUTs) and internal slice registers.

4.4 Convolution Module

4.4.1 Router Cluster

The router cluster is the data-distribution interface between the GLBs and the PE cluster. The dataflow of this module, including unrolling, halo data exchange, and weight addressing, have been discussed in Section 3.4. Therefore, this subsection focuses on how these mechanisms are implemented in the RTL

structure, both at the macroscopic routing level and the microscopic addressing level. As shown in Fig. 4.3, the static physical topology consists of a memory level (GLB_cluster and GLB_grad_cluster), the routing subsystem, and the computation array (PE_cluster). The router cluster is subdivided into the IACT Router, WGHT Router, and OFMAP Router. This fixed physical interconnect allows the same PE cluster datapath to be reused across the complete training process by simply altering the routing behavior.

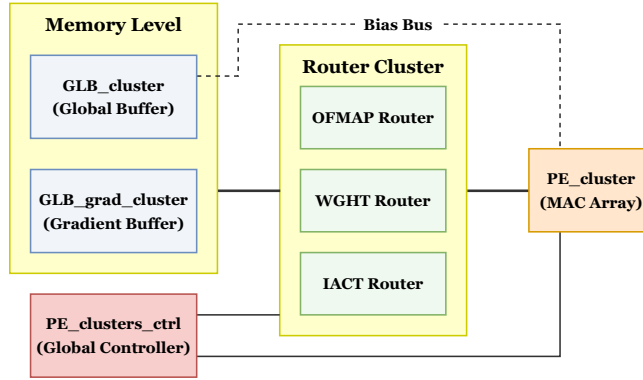


Figure 4.3: Overall Physical Topology.

In the forward propagation (FP) path (Fig. 4.4a), the IACT Router reads input activations from the GLB and writes them into the local activation buffer of the PE cluster. It generates the corresponding read addresses according to the current unroll mode and feature-map position, inserting padding when the sliding window reaches the boundary. For spatially partitioned feature maps, the IACT router also interfaces with the halo buffer, allowing neighboring rows to be passed between adjacent computation regions without additional GLB access. Simultaneously, the WGHT Router fetches the forward weights, while the OFMAP Router collects the computed results and writes them back to the GLB.

The two backward propagation dataflows are illustrated in Fig. 4.4b and Fig. 4.4c. During the activation-gradient calculation phase (BP, Fig. 4.4b), the IACT Router is multiplexed to read the error gradients (dZ) from the next layer via the gradient buffer (GLB_grad_cluster). Concurrently, the WGHT Router fetches the stored weights (W). The PE cluster computes the input gradients (dX), which are subsequently written back to the gradient buffer by the OFMAP Router. Conversely, during the weight-gradient computation phase (WG, Fig. 4.4c), the IACT Router reloads the stored forward activations (X). The WGHT Router is repurposed to stream the error gradients (dZ) into the PE array. To avoid external read-modify-write overhead, the weight gradients (dW) are accumulated directly within the on-chip SRAM of the PE cluster. Finally, the OFMAP Router routes these accumulated weight gradients back to the gradient buffer.

Beyond macroscopic routing, the router cluster must also handle the specific algorithmic data mapping required by different training phases. The two backward

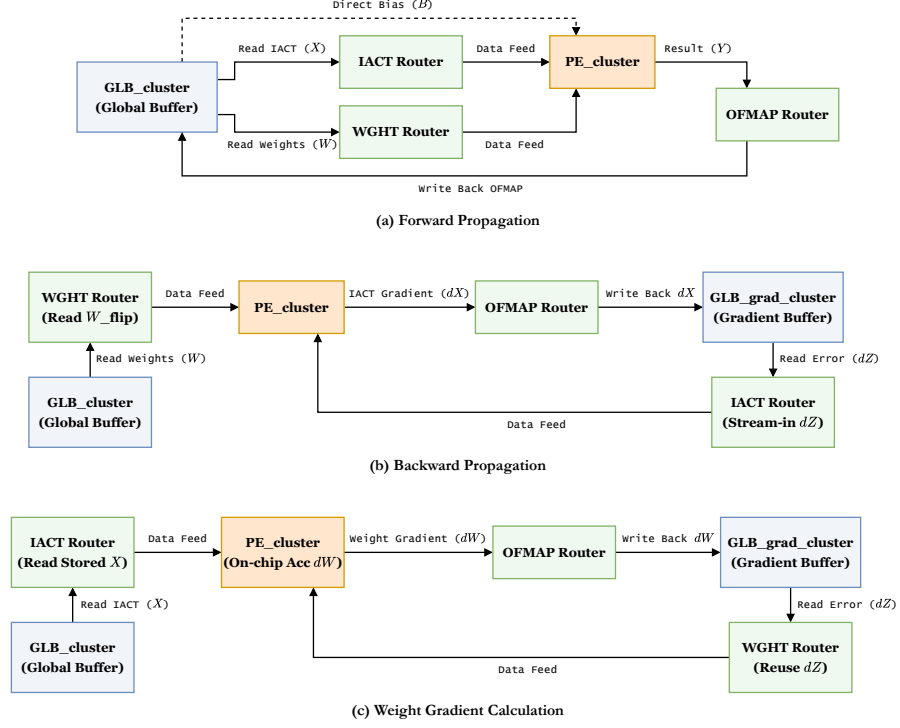


Figure 4.4: Dynamic Routing Configurations

data-reading patterns of the sixth convolution layer are shown in Fig. 4.5 and Fig. 4.6. In this layer, both the input and output feature maps have a spatial size of 8×8 . For activation-gradient calculation, the activation router reads the gradient feature map from the next layer, while the weight router reads the stored weights in reversed kernel order. This reversed read order realizes the 180° kernel rotation through address generation, rather than physically rearranging the weights in memory, as discussed in Section 3.4. For weight-gradient calculation, the activation router reloads the stored forward activations, while the weight router provides the activation gradients from the next layer as the streaming operand for gradient accumulation. Therefore, the same router cluster can support different backward stages by changing the data source and address generation pattern.

The output router collects the PE cluster outputs and writes them into the PISO buffer. It preserves the PE output format for the following gather and post-processing stages. In this way, the router cluster hides the difference between the global-buffer memory layout and the local PE computation format, allowing the same PE cluster datapath to be reused across the complete training process.

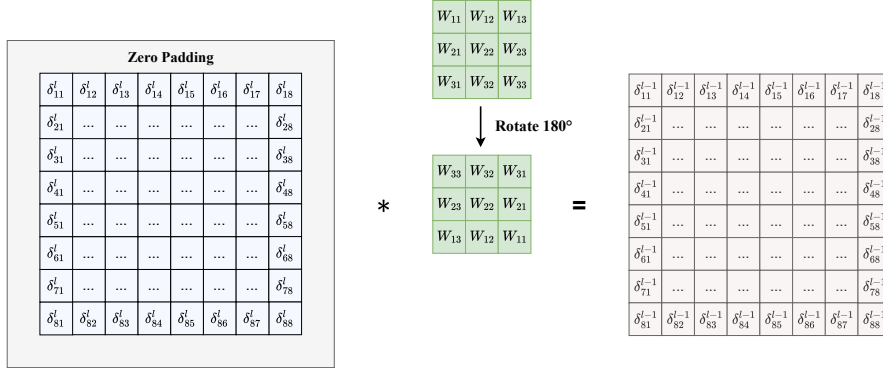


Figure 4.5: Data reading pattern for activation-gradient calculation in the sixth convolution layer.

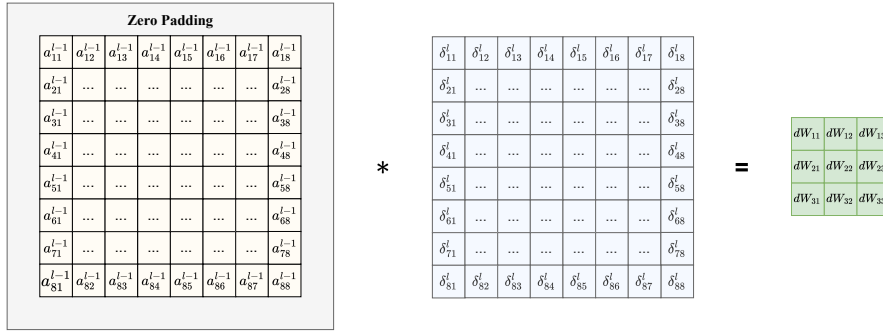


Figure 4.6: Data reading pattern for weight-gradient calculation in the sixth convolution layer.

4.4.2 Processing Element Cluster

The PE cluster is the core computation block of the convolution module. In this design, the convolution module contains four PE clusters working in parallel. As shown in Fig. 4.7, each cluster consists of an 8×3 PE array, ten activation row banks, a weight distribution path, and an output summation path. The activation row banks are shared at the cluster level, while the weight values are distributed to the local weight banks inside each PE. This organization decouples PE computation from global-buffer access and provides short-distance data reuse inside the cluster.

A key reason for introducing the local activation row banks is to exploit input feature-map reuse along the output-channel dimension. As indicated by the unified convolution loop in Algorithm 2, for a fixed input channel and spatial window, the same input activation tile is reused when the output-channel loop traverses different convolution kernels. These kernels generate different output channels

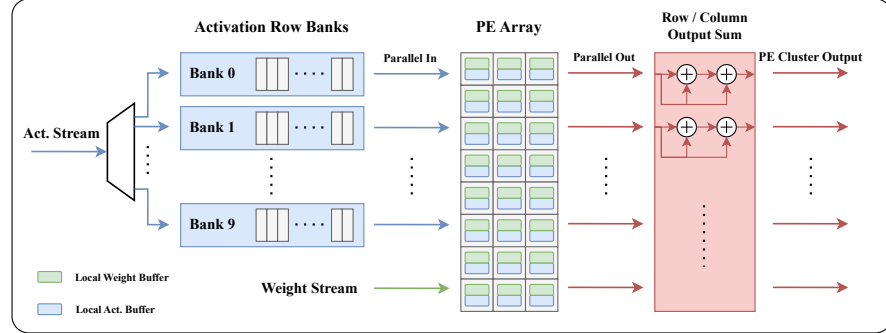


Figure 4.7: Structure of single PE cluster.

from the same local input region. Therefore, the activation router first loads the required input tile into the activation row banks. The PE cluster can then keep this tile locally and reuse it while the controller switches the output-channel index and loads a new group of weights into the local PE weight banks.

This reuse process is scheduled by the PE cluster controller, as shown in Fig. 4.8. The controller first loads the required weights and activation rows into the local buffers, and then starts the PE computation. During the traversal of output channels, it determines whether the current activation tile can be reused or whether a new weight or activation tile must be loaded.

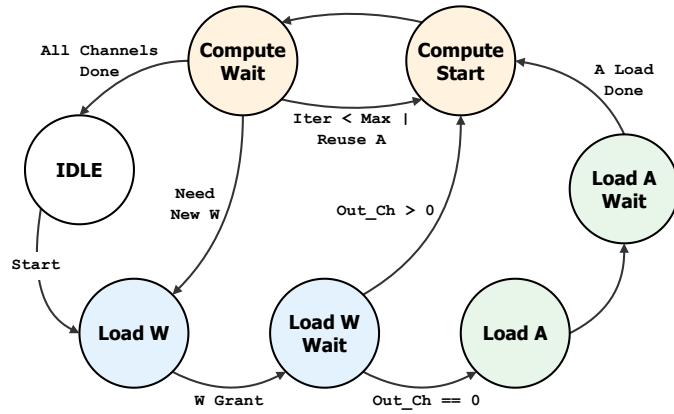


Figure 4.8: Control flow of the PE cluster controller.

The PE cluster size is selected according to the spatial size of the target dataset and the four-cluster parallel structure of the accelerator. As discussed in Chapter 3, spatial unrolling should be matched to the feature-map shape to avoid idle PEs and maintain high array utilization. This principle is also consistent with the general loop-unrolling analysis in CNN accelerators, where the unrolling factors determine

how many loop iterations can be executed concurrently by the PE array.

For the target CIFAR-10 input, the largest feature map is 32×32 . In the 4-way spatial unrolling mode, as illustrated in Fig. 9, the 32 output rows are evenly assigned to four PE clusters, so each cluster computes eight output rows. Therefore, the PE array uses $X_{dim} = 8$ to match the eight output positions processed by one cluster. Since all convolution layers use a 3×3 kernel, the vertical dimension is set to $Y_{dim} = 3$, allowing the three kernel rows to be mapped naturally onto the three PE rows.

The activation buffer is then organized according to this local computation tile. For a stride-1 convolution, an output tile of eight rows requires

$$T_{in} = T_{out} + K - 1 = 8 + 3 - 1 = 10$$

input feature-map rows. Therefore, each PE cluster contains ten activation row banks. These banks store the complete local input tile required by the cluster, allowing the sliding-window computation to reuse overlapping rows locally while the PE array remains fully occupied in the main spatial-unrolling case.

After the PE computations, the partial results are accumulated inside the cluster according to the current computation phase. In forward convolution and activation-gradient calculation, the products are accumulated along the vertical direction to produce each output feature-map value. In weight-gradient calculation, the accumulation direction is changed to match the gradient-generation dataflow, and the PE outputs are summed along the corresponding row or column direction to form the final weight-gradient values. Through this phase-dependent reduction, the same 8×3 PE array can generate the final outputs required by both propagation.

4.4.3 Processing Element

The PE is the basic arithmetic unit inside the PE cluster. It performs local multiplication, addition, partial-sum accumulation, and result output. Instead of being implemented as a single fixed MAC unit, the PE is designed as a small configurable computation block. Under different execution phases, it keeps the same external interface, while the input source, buffer read order, and accumulation path are controlled by the PE cluster controller. The datapath of single PE is shown in Fig. 4.9.

For forward convolution and activation-gradient calculation, the PE uses a three-multiplier datapath. One activation value is read from the selected activation row buffer in each cycle and shifted into the activation shift register. The register then provides three adjacent activation values to the multiplier lanes. The weights are read from the selected weight row buffer and supplied as the corresponding kernel taps. As shown in Fig. 4.9, the fourth multiplier lane is not used in this mode. Both the activation buffer and the weight buffer adopt a ping-pong structure, so that one row buffer can support the current computation while the other row buffer is prepared for the next computation row.

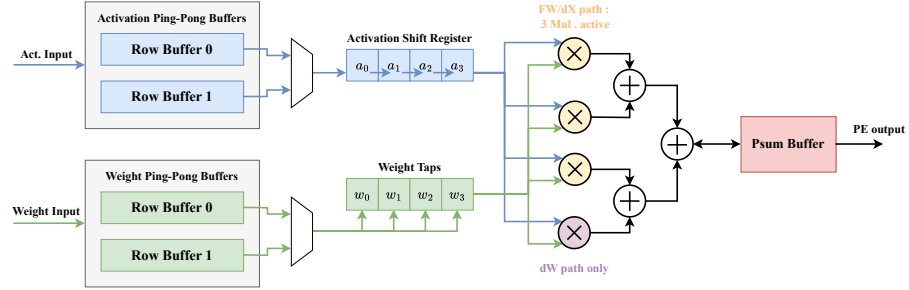


Figure 4.9: Datapath of single Processing Element. The first three multiplier lanes are used for forward convolution and activation-gradient calculation, while the fourth lane is additionally enabled during weight-gradient calculation.

For weight-gradient calculation, the PE enables the additional multiplier lane. In this stage, the computation is based on the correlation between the stored forward activations and the activation gradients δ . Since the feature-map width and height of each convolution layer in the target network are multiples of 4, the δ stream can be grouped into four adjacent values. Therefore, the weight-gradient datapath uses four parallel multipliers, allowing four pairs of stored activations and activation gradients to be multiplied in one cycle.

The PE also contains local accumulation logic. For forward convolution and activation-gradient calculation, partial sums are accumulated across the required input-channel computations and kept inside the PE. The PE only sends the result to the cluster-level output path when the current accumulation group is completed. For weight-gradient calculation, the products from the four multiplier lanes are accumulated into the local gradient accumulator before being sent to the cluster output path. This local accumulation reduces unnecessary intermediate write-back traffic and keeps the main accumulation close to the arithmetic datapath.

With this organization, the PE array can be driven by a unified cluster-level control interface, while the internal arithmetic path is reconfigured according to the current computation phase.

4.4.4 PISO FIFO

The PISO FIFO is placed at the output side of each PE cluster. The PE cluster generates multiple output lanes in parallel, whereas the global-buffer write-back logic accepts data according to its own address order and timing. Directly coupling these two sides would force the PE cluster to stall whenever the write-back logic is busy. Therefore, the PISO output FIFO stores each group of PE outputs as a packet and releases the values lane by lane, preventing write-back latency from directly back-pressuring the PE computation. The structure of the PISO FIFO is

shown in Fig. 4.10.

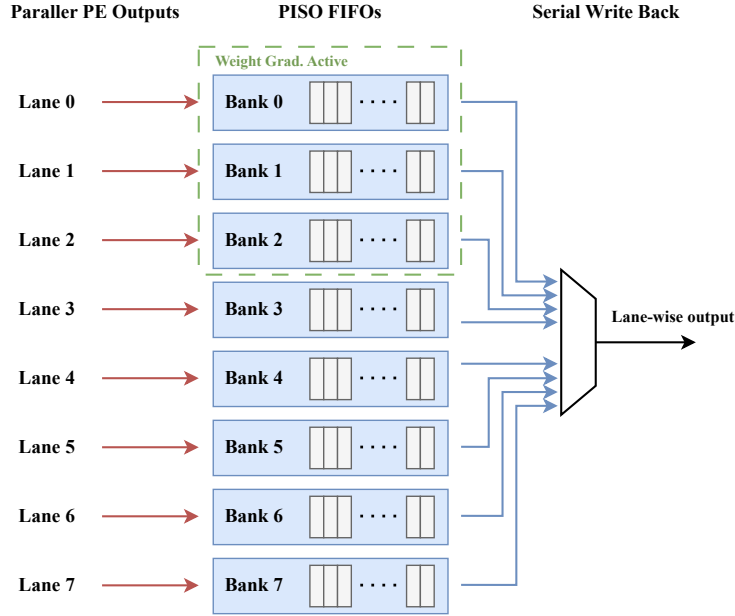


Figure 4.10: Structure of the PISO FIFO used at the output side of each PE cluster.

In this design, the FIFO is organized as $X_{dim} = 8$ lane banks. The eight PE output lanes are written into Bank 0 to Bank 7 in parallel. During a write operation, these eight values share the same packet address, so one write step stores one complete output packet. During a read operation, a lane counter selects the bank to be read from the current packet. After all valid lanes in the current packet have been read, the read pointer moves to the next packet. The packet counter is used to generate the full and empty status signals, which provide flow control between the PE cluster and the following write-back logic.

The valid lane number depends on the current computation phase. In forward convolution and activation-gradient calculation, all $X_{dim} = 8$ lanes are used, and the FIFO serializes the eight cluster outputs into a lane-wise stream. In weight-gradient calculation, the valid lane number is reduced to $Y_{dim} = 3$, matching the three-row structure of the PE array. Therefore, only the first three lane banks are active in this mode, as indicated in Fig. 4.10. This organization allows the FIFO to serve both the normal 8-lane output pattern and the 3-lane weight-gradient output pattern with the same storage structure.

4.5 Data Post-processing Module

The data post-processing module is placed between the PISO FIFOs and the GLBs. Its basic function is to collect the local results produced by the four PE clusters, reconstruct their global tensor order, and write them back to the corresponding GLB. Since each PE cluster only computes a part of the feature map, the cluster outputs are naturally distributed in several local streams. The data post-processing module converts these local streams into a unified global memory layout.

The control flow of this module is shown in Fig. 4.11. In the *Wait Cluster* state, the module waits until one of the cluster-side PISO FIFOs contains valid data. It then enters the *Read Buffer* state and sends a read request to the selected FIFO. After the data is returned, the module moves to the *Prepare* state, where the output address is calculated according to the cluster index, spatial counters, and unroll mode. If the current unroll mode requires results from multiple clusters to be merged, the intermediate values are temporarily stored and accumulated inside the module. Finally, in the *Write GLB* state, the valid result is written back to the GLB. When all required output chunks of the current layer or training phase have been written, the module enters the *Finished* state and sends a completion signal to the upper-level controller.

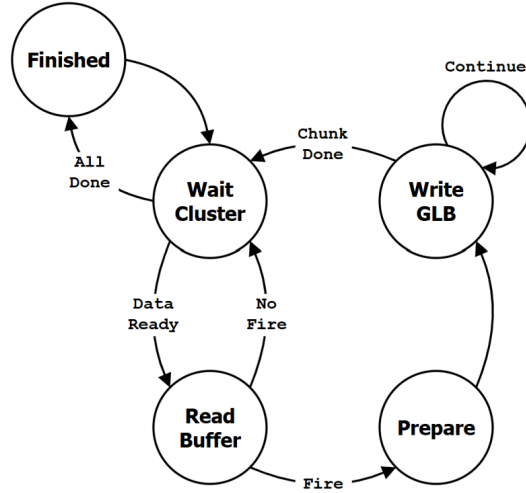


Figure 4.11: State transition diagram of the data post-processing module.

Besides the state control, the data post-processing module also handles the accumulation of partial sums generated under different spatial unrolling modes. This mapping is illustrated in Fig. 4.12. In the 4-way spatial unrolling mode, the four PE clusters are assigned to different spatial tiles, so their outputs correspond to different global feature-map regions and can be written back directly. When the unroll factor is reduced to 2-way or 1-way, some PE clusters are instead assigned

to different input-channel groups of the same output feature-map region. These clusters produce partial sums for the same output positions, and the data post-processing module accumulates them along the input-channel dimension before writing the final results back to the GLB.

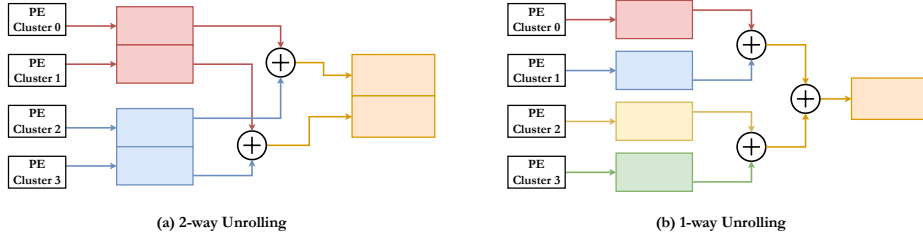


Figure 4.12: Partial-sum recombination in 2-way and 1-way spatial unrolling modes.

For forward convolution and activation-gradient calculation, the module reads the serialized outputs from the PISO FIFOs and maps them back to the global feature-map address space. The cluster index and local spatial counters are used to generate the global write address. When several clusters correspond to different input-channel groups of the same output region, the internal accumulation buffer is used to combine their partial sums before write-back.

For weight-gradient calculation, the module writes the collected results into the weight-gradient buffer instead of the feature-map buffer. The module calculates the corresponding gradient addresses and writes the valid gradient values to the global gradient buffer. During batch accumulation, if the current sample is not the first sample of the batch, the previous gradient value is read first and then accumulated with the newly generated gradient before being written back.

In addition to the basic gather and write-back function, this module also provides the interface for later functional operations, such as ReLU and pooling related processing. These functions will be introduced separately in the following sections.

4.6 Functional Operation Module

4.6.1 Max Pooling

The max pooling is used to downsample the convolution output feature map. In this design, the pooling window size is 2×2 and the stride is 2 and the data are provided as a row-wise stream.

For a 2×2 pooling window, four input values from two adjacent rows and two adjacent columns must be compared. Since the input data arrive row by row, the two adjacent values in the same row can be compared directly, while the corresponding result from the previous row must be temporarily stored. Therefore,

a FIFO line buffer is used to keep the horizontal maximum values of the previous row.

The streaming dataflow is shown in Fig. 4.13. For each input row, two adjacent values are first compared along the horizontal direction. The pairwise results of the first row in the pooling window are written into the FIFO. When the next row arrives, the module performs the same comparison on the current row and reads the corresponding stored values from the FIFO. These two values are then compared vertically to produce the final result.

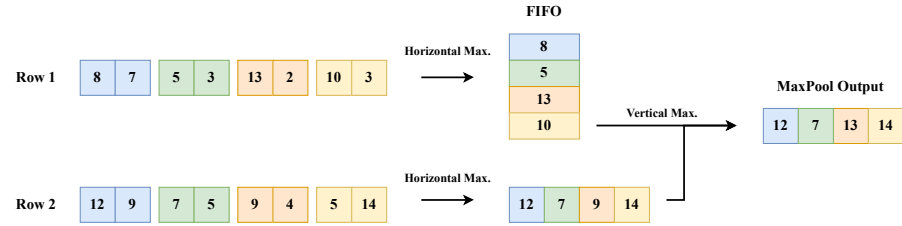


Figure 4.13: Streaming dataflow of the 2×2 max-pooling with a FIFO line buffer.

This structure avoids storing a complete two-row input window. The FIFO only stores the pairwise maxima from the previous row, so its depth is only half of the input feature-map width. As a result, max-pooling can be performed in the streaming path with low buffer cost.

During training, the backward pass of max-pooling requires the position of the maximum value selected during the forward pass. Therefore, the module also generates a 2-bit index for each 2×2 pooling window. This index records the selected position in the pooling window and is stored together with the forward execution information. During backward propagation, it is used to route the incoming gradient back to the corresponding input position.

4.6.2 Global Average Pooling

GAP is applied after the last convolutional layer to compress each spatial feature map into a single channel-wise value. In the target network, the input feature map of this module has a spatial size of 4×4 , so one output value is produced for each channel. The forward and backward dataflow of this operation is illustrated in Fig. 4.14.

In the forward path, the module accumulates the 16 spatial values of the current channel using a local accumulator. The accumulated sum is then scaled by the constant factor $1/16$ to generate the GAP output. Since the pooling window size is fixed, this scaling is implemented as a right shift by 4 bits, avoiding the use of a hardware divider.

During backward propagation, the gradient from the following fully-connected

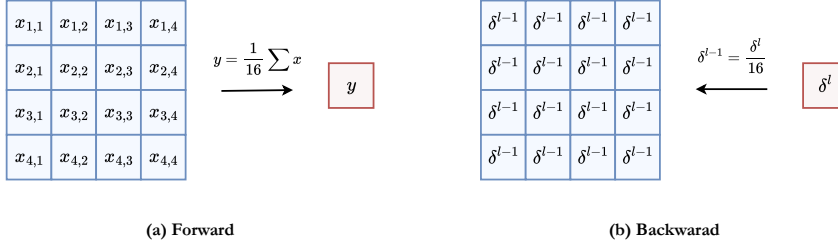


Figure 4.14: Forward and backward dataflow of global average pooling over a 4×4 feature map.

layer is evenly distributed to all 16 spatial positions of the corresponding channel, as shown in Fig. 4.14(b). This operation also uses the same constant scaling factor of $1/16$, which can be implemented by a right shift. Therefore, both the forward and backward average-pooling operations can be realized with simple accumulation and shift logic.

This implementation keeps the average-pooling module lightweight while providing the spatial compression required before the fully-connected layer.

4.6.3 Fully-Connected Layer

The fully-connected layer is placed after the global average pooling module and serves as the final classification of the network. It has 128 input channels and produces 10 output logits for the following Softmax module. Since the computation scale of this layer is only 128×10 , it is much smaller than that of the convolution layers. Therefore, the RTL implementation does not reuse the PE cluster structure. Instead, a compact sequential MAC datapath is used to read the input features, weights, and biases, and then generate the accumulated output with fixed-point scaling and saturation.

During training, the same module also supports the backward computation of the fully-connected layer. It receives the Softmax output and label information to form the classification gradient, and then computes the input gradient passed to the average-pooling backward stage. At the same time, the weight gradients and bias gradients of the fully-connected layer are generated and accumulated in the corresponding gradient buffers.

4.6.4 SoftMax

The Softmax module is placed after the fully-connected layer to convert the class logits into a probability distribution. In the target network, the number of classes is 10, so the module receives ten fixed-point logits and outputs ten normalized probability values. During training, these probability values are also used by the

loss function and fully-connected backward computation.

Although the number of classes is small, a direct hardware implementation of Softmax is still expensive because it involves exponential evaluation and division. Therefore, the Softmax computation is divided into several hardware stages: normalization, exponential approximation, reciprocal approximation, and final probability output. This organization reuses one exponential unit and one reciprocal unit, avoiding multiple expensive function units.

As discussed in Chapter 3, the Softmax computation is first rewritten into a max-shifted form to improve numerical stability:

$$p_i = \frac{e^{z_i - z_{\max}}}{\sum_{j=0}^{C-1} e^{z_j - z_{\max}}}, \quad z_{\max} = \max(z_0, z_1, \dots, z_{C-1}).$$

After this normalization, the inputs to the exponential unit are non-positive, which reduces the dynamic range required by the fixed-point implementation.

The exponential unit further converts the natural exponential into a base-2 form:

$$e^x = 2^{x/\ln 2}.$$

The converted value is decomposed as

$$x' = x \cdot \frac{1}{\ln 2} = q + f, \quad q = \lfloor x' \rfloor, \quad f \in [0, 1).$$

In hardware, the integer part q is handled by shift operations, while the fractional part f is approximated by a two-segment quadratic function.

The fractional approximation adopts the following form:

$$P(f) = a + b(f - 0.5) + c((f - d)^2 - d^2).$$

For $f \in [0, 0.5)$, the coefficients are selected as

$$P_1(f) = 1.4210526 + 0.8411092(f - 0.5) + 0.2860875((f - 0.2757789)^2 - 0.2757789^2).$$

For $f \in [0.5, 1)$, the coefficients are selected as

$$P_2(f) = 1.4963002 + 0.9269232(f - 0.5) + 0.4045117((f - 0.4512034)^2 - 0.4512034^2).$$

All coefficients are quantized as fixed-point constants in the RTL implementation, and the polynomial is evaluated using fixed-point multiplication, addition, and arithmetic shifting. The output of the exponential unit is finally saturated to the unsigned exponent format.

After all exponential values are generated, they are accumulated to form the denominator:

$$S = \sum_{j=0}^{C-1} E_j.$$

The reciprocal module then computes an approximation of $1/S$. To avoid using a general divider, the reciprocal unit first normalizes the input with a leading-zero counter, estimates an initial reciprocal, and applies a multiplicative refinement step. The final probability is then generated by

$$p_i = E_i \cdot S^{-1}.$$

The accuracy of the exponential approximation was evaluated using a Python fixed-point model consistent with the RTL implementation. The tested range is $x \in [-10, 0]$, which matches the effective input range after max-shift normalization. As shown in Fig. 4.15, the approximated curve almost overlaps with the floating-point reference e^x , and the relative error remains within a small range. This indicates that the two-segment quadratic approximation provides sufficient accuracy for the Softmax computation while keeping the hardware implementation compact.

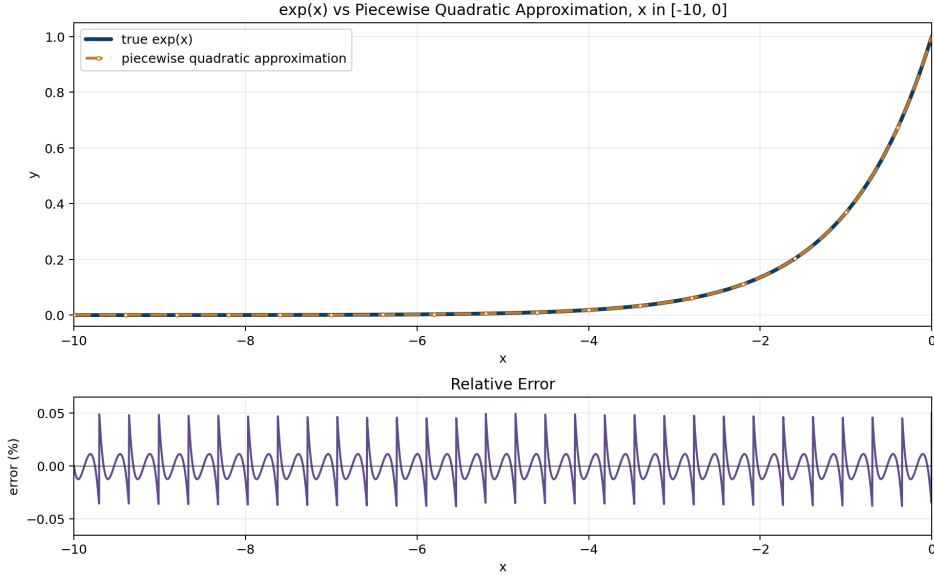


Figure 4.15: Accuracy evaluation of the fixed-point exponential approximation over $x \in [-10, 0]$.

4.6.5 Stochastic Rounding & LFSR

As discussed in Chapter 3, stochastic rounding is introduced to reduce the bias caused by deterministic fixed-point truncation. This subsection focuses on its hardware implementation. In the proposed accelerator, an 18-bit LFSR is used as the pseudo-random source for stochastic rounding. The LFSR is updated every clock cycle and its output is distributed to the datapath stages that require random rounding.

The structure of the LFSR is shown in Fig. 4.16. It is implemented as a Fibonacci LFSR with a non-zero initial seed. In each cycle, bit 10 and bit 17 are XORed and fed back to the lowest bit, while the remaining bits are shifted by one position. The feedback operation can be written as

$$LFSR[0] \leftarrow LFSR[10] \oplus LFSR[17],$$

and the corresponding feedback polynomial is

$$f(x) = x^{18} + x^{11} + 1.$$

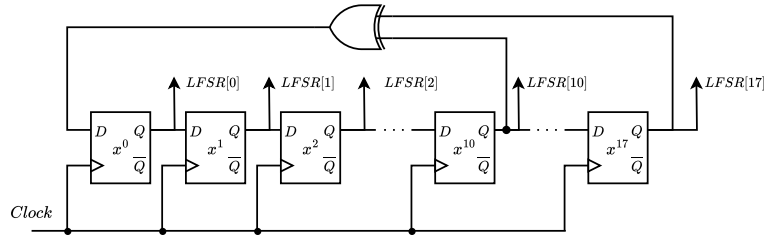


Figure 4.16: 18-bit Fibonacci LFSR used as the pseudo-random source.

Since this polynomial is primitive, the LFSR can traverse $2^{18} - 1 = 262143$ non-zero states before repeating, while the all-zero state is avoided by using a non-zero initial seed.

In the RTL implementation, stochastic rounding is mainly applied when high-precision multiplication or accumulation results are rescaled back to a lower-precision fixed-point format. These rescaling operations are commonly implemented by arithmetic right shifts. Instead of directly discarding the lower bits, a pseudo-random value generated from the LFSR is added before the shift:

$$y = \left\lfloor \frac{x + r}{2^s} \right\rfloor,$$

where x is the high-precision intermediate result, s is the number of discarded fractional bits, and r is taken from the lower bits of the LFSR. In this way, the discarded fractional part is converted into a probabilistic carry, which reduces the long-term bias caused by deterministic truncation.

This mechanism is inserted into several fixed-point rescaling paths after multiplication or accumulation, such as the fully-connected computation, convolution output accumulation, gradient accumulation, and pooling-related scaling. Therefore, stochastic rounding is not implemented as an independent computation stage, but as a lightweight rounding logic attached to existing shift-based rescaling operations.

The same LFSR is also reused by the stochastic activation-pruning module to generate random comparison values. As a result, stochastic rounding and stochastic

pruning share a single pseudo-random source, avoiding additional random-number generators.

To evaluate its effect, the RTL results with and without stochastic rounding are compared against the floating-point reference at different layers, as shown in Fig. 4.17. The comparison demonstrates the influence of stochastic rounding on the fixed-point error distribution. Instead of producing a deterministic truncation error, the LFSR-based implementation introduces a less biased rounding behavior.



Figure 4.17: Layer-wise activation error comparison between RTL results and floating-point reference with and without stochastic rounding.

4.6.6 Stochastic Approximate Pruning

In the proposed accelerator, the SAP module is placed beside the activation-gradient buffer. It does not affect the forward path or the main PE computation datapath. Instead, it processes the activation gradients generated during back-propagation.

The hardware implementation follows a two-phase flow. In the collection phase, the activation-gradient stream is monitored while it is written into the gradient buffer. At the same time, the module accumulates the absolute values of the incoming gradients. After all gradients of the current tensor have been collected, the pruning controller starts the pruning phase.

The pruning threshold is generated using a layer-dependent fixed-point coefficient. Since different backward layers have different tensor sizes, the corresponding coefficient is pre-computed and stored as a constant in the RTL. The threshold is

calculated as

$$T_l = \left(\sum_i |g_i| \right) \cdot \alpha_l,$$

where g_i denotes the activation gradient and α_l is the coefficient selected for layer l . This formulation avoids gradient sorting in hardware and allows the threshold to be obtained using only an accumulator, a fixed-point multiplier, and a shift operation.

In the pruning phase, the stored gradients are read back from the gradient buffer and processed according to the SAP rule described in Chapter 3. The hardware compares the gradient magnitude with the generated threshold and uses the LFSR output to decide whether a small gradient is dropped or mapped to the signed threshold value. Therefore, the pruning datapath mainly consists of an absolute-value unit, an accumulator, a threshold multiplier, magnitude comparators, random comparison logic, and a small output selection path.

Fig. 4.18 shows the overall activation-gradient distribution before and after RTL pruning. The original activation gradients are densely distributed around zero, with only 0.89% sparsity. After pruning, the sparsity increases to 70.25%. This greatly reduces the amount of non-zero gradient data that needs to be propagated and computed in the backward path.

To further examine the pruning behavior across different layers, Fig. 4.19 shows the activation-gradient distributions of selected convolution layers before and after RTL pruning. For all shown layers, the original gradients are concentrated around zero, while the pruned distributions contain a large zero peak and two side peaks around the positive and negative pruning thresholds. This matches the stochastic pruning rule described in Chapter 3, where small gradients are either removed or mapped to the signed threshold value. The sparsity of Conv6, Conv5, and Conv4 increases to 70.79%, 70.78%, and 69.85%, respectively, indicating that the RTL pruning module provides consistent sparsification across different backward layers.

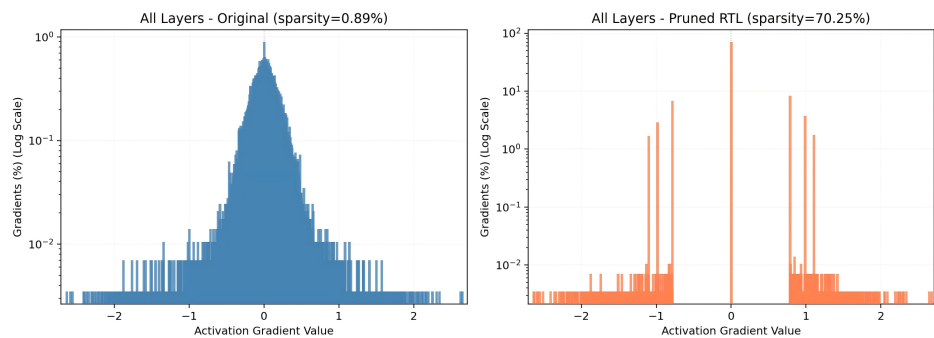


Figure 4.18: Activation-gradient distribution before and after RTL stochastic pruning.

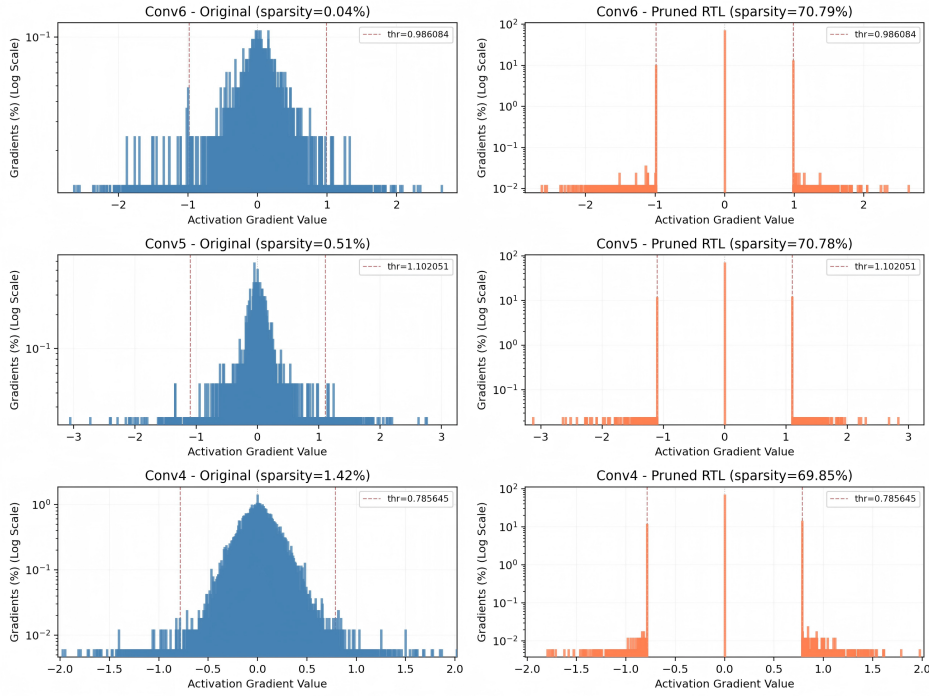


Figure 4.19: Layer-wise activation-gradient distributions before and after RTL stochastic pruning.

4.6.7 Parameter Update

After the gradients of one batch have been accumulated, the parameter update module is triggered to update the trainable weights and biases. This module reads the current parameters from the parameter GLB and reads the accumulated gradients from the gradient GLB. The updated parameters are then written back to the original parameter buffer.

The overall datapath of the residual-based update module is illustrated in Fig. 4.20. Besides the parameter and gradient buffers, an additional residual memory is introduced to store the fractional update components that cannot be represented in the parameter format during the current batch update.

In this design, the weight update and bias update share the same update core. The two update modules only differ in their address ranges and residual memory depths. This shared structure reduces duplicated control logic and keeps the update behavior consistent for both weights and biases.

The update module adopts a shift-based update scheme. Instead of using a multiplier to apply the learning rate, the accumulated gradient is first scaled by a configurable shift:

$$s_{\text{pre}} = s_{\text{lr}} + s_{\text{batch}},$$

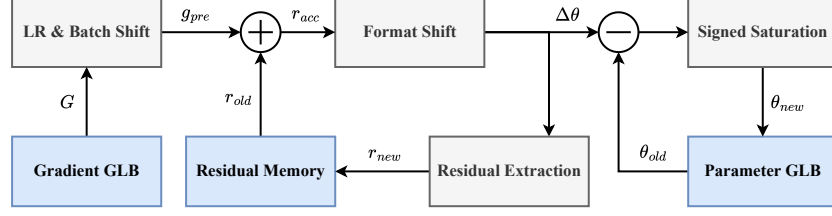


Figure 4.20: Residual-based parameter update datapath.

where s_{lr} is the learning-rate shift configured by the PS, and s_{batch} is the batch-average shift determined by the batch size. Since the batch size is selected as a power of two, batch averaging can be implemented as a right shift. The pre-scaled gradient is therefore obtained as

$$g_{pre} = G \gg s_{pre},$$

where G is the accumulated gradient read from the gradient buffer.

A residual accumulation mechanism is introduced to reduce the loss caused by fixed-point format conversion. Since the gradient format and the parameter format use different fractional widths, directly shifting the gradient to the parameter format would discard small update components. To avoid permanently losing these small values, each trainable parameter is associated with a residual value stored in the residual memory. During each batch update, the pre-scaled gradient is added to the previous residual:

$$r_{acc} = g_{pre} + r_{old}.$$

The accumulated value is then shifted to the parameter fractional format:

$$\Delta\theta = r_{acc} \gg s_{diff},$$

where s_{diff} is determined by the fractional-width difference between the gradient and the parameter. In this design, the accumulated weight and bias gradients use 11 fractional bits, while the weights and biases use 7 fractional bits. Therefore, $s_{diff} = 11 - 7 = 4$. The remaining part that cannot be represented in the parameter format is written back as the new residual:

$$r_{new} = r_{acc} - (\Delta\theta \ll s_{diff}).$$

In this way, update components smaller than one parameter LSB are not discarded immediately. Instead, they are accumulated across batch updates and can contribute to the parameter value once their magnitude becomes large enough.

The final parameter update is performed as

$$\theta_{new} = \text{sat}(\theta_{old} - \Delta\theta),$$

where θ_{old} is the original parameter value and $\text{sat}(\cdot)$ denotes signed saturation to the target parameter width. The saturation logic prevents overflow when the updated value exceeds the representable fixed-point range.

System Integration and Evaluation

This chapter focuses on the accelerator proposed in Chapter 4 and presents the development of a complete FPGA-based SoC system, as well as its design and verification. By deploying the CIFAR-10 network, various accelerator performance metrics are analyzed and compared with those reported in related studies.

5.1 FPGA Platform Overview

The FPGA-based prototype validation in this work is performed on the AMD Xilinx ZCU104 development board from the Zynq UltraScale+ MPSoC family. This platform integrates a PS based on ARM cores with PL resources, making it suitable for embedded acceleration scenarios that require high computing capability, compact system integration, and efficient power consumption.

The ZCU104 board is built on the Xilinx UltraScale+ architecture and provides abundant hardware resources for customized accelerator implementation. Its PS integrates a quad-core Arm Cortex-A53 processor and a dual-core Arm Cortex-R5 real-time processor. Meanwhile, the PL fabric offers 44.2 Mb of on-chip storage resources, including 6.2 Mb of distributed RAM, 312 BRAM blocks, and 96 URAM blocks. In addition, the PL region contains 1,728 DSP slices, 230,400 LUTs, and 460,800 flip-flops, providing the computational and storage resources required by the proposed accelerator.

The development environment is based on Vivado and Vitis 2024.1. Vivado is used for the implementation of the on-chip CNN training accelerator in the PL, while Vitis is employed to develop the control program executed on the PS. During system integration, the PS and the PL communicate through AXI-Lite interfaces for control register access and data transfer between the PS and PL sides.

5.2 Verification System Design and Workflow

The architecture of the FPGA-based CNN accelerator system is shown in Fig. 5.1, which consists of two main parts: the PS and the PL. The PS acts as the system host, responsible for initialization, data scheduling, parameter configuration, and result collection, while the PL implements the CNN accelerator and performs the computation.

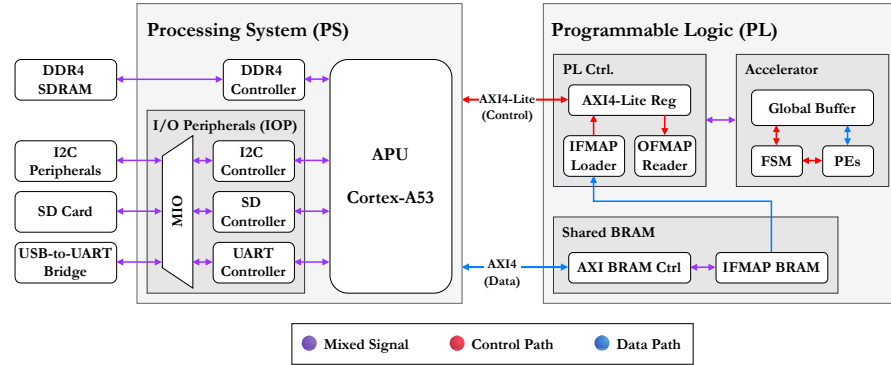


Figure 5.1: Architecture of the FPGA prototype verification system.

On the PL side, additional control and interconnection modules are integrated on top of the accelerator core to support data transfer and execution control between the PS and the accelerator. An on-chip BRAM buffer is used for input images, allowing the PS to write sample data before the accelerator starts.

The PS connects to the PL Controller via an AXI4-Lite control bus, while the on-chip shared BRAM is accessed through an AXI BRAM Controller on a separate AXI4 data interface. The AXI Interconnect IP performs address decoding and routing for these master/slave links. Through this fabric, the AXI BRAM controller exposes the on-chip buffer to the PS, and the PL Controller receives configuration parameters and control signals to drive accelerator execution. The main configuration and control registers are listed in Table 5.1.

It is important to note that these register-level control signals are not fed directly into the compute core. Inside the PL, they are first decoded by the `TOP_wrapper`, which maintains batch-level state and generates internal control qualifiers (e.g., the first-sample flag for gradient accumulation). This wrapper layer bridges the PS/PL protocol boundary and ensures that the accelerator core receives consistent, phase-aligned control signals during training.

During execution, the dataset is stored on the SD card. The PS loads input images into DDR memory and then writes them into the PL IFMAP BRAM buffer. After `start_sample` is issued, the PL Controller triggers the accelerator to perform forward inference or forward-backward training. In training mode, the batch size configured by `batch_size` is used by the PL-side wrapper logic to track batch progress and to distinguish the first sample of each batch for correct

Table 5.1: AXI4-Lite register map of the PL control interface.

Offset	Access	Field	Description
0x00	W	<code>start_load</code>	Start loading one sample from BRAM.
0x04	W	<code>start_sample</code>	Start sample computation.
0x08	W	<code>start_update</code>	Start parameter update.
0x0C	W	<code>mode</code>	Select inference or training mode.
0x10	W	<code>batch_size</code>	Set the training batch size.
0x14	W	<code>learning_rate</code>	Set the learning rate.
0x18	R	<code>load_done</code>	Indicate that input loading is complete.
0x18	R	<code>top_done</code>	Indicate that computation or update is complete.
0x1C-0x40	R	<code>results</code>	Read ten classification outputs.
0x44	R	<code>top_state</code>	Read the current top-level controller state.

gradient accumulation.

When computation completes, results and status signals are returned to the PS through the PL Controller. The PS aggregates the outputs, computes accuracy and loss statistics, and reports the results to the PC via UART. In addition, the PS I²C interface is enabled for board-level initialization on the ZCU104 platform, allowing the First Stage Boot Loader (FSBL) to configure clocks, power, and I/O peripherals before the accelerator application starts.

In this design, the PS manages data movement and system control, while the PL executes the computation-intensive CNN workload. Their collaboration through AXI-based communication and on-chip memory forms a complete FPGA prototype verification platform.

5.3 Vivado Verification

The Block Design view of the system is shown in Fig. 5.2. In this design, the `TOP_wrapper` module serves as the accelerator, while `PL_Ctrl` acts as the accelerator controller. Both modules are packaged as custom IP cores, with their external interfaces preserved for communication with other system modules. The remaining components are implemented using official Xilinx IP cores. After implementation, the resource utilization report of the system is obtained from Vivado, as summarized in Fig. 5.3.

During accelerator execution, the weights, input feature maps, output feature maps, and gradients of each layer are stored on chip using BRAM and URAM resources. This storage organization supports data reuse and reduces frequent off-chip memory access during training and inference.

For DSP resources, the main consumption comes from the PE clusters in the accelerator. The design contains four PE clusters, and each cluster consists of 8×3 PE units. Each PE unit internally includes four multipliers, resulting in 384 DSP slices used by the PE clusters. In addition, other modules, such as the

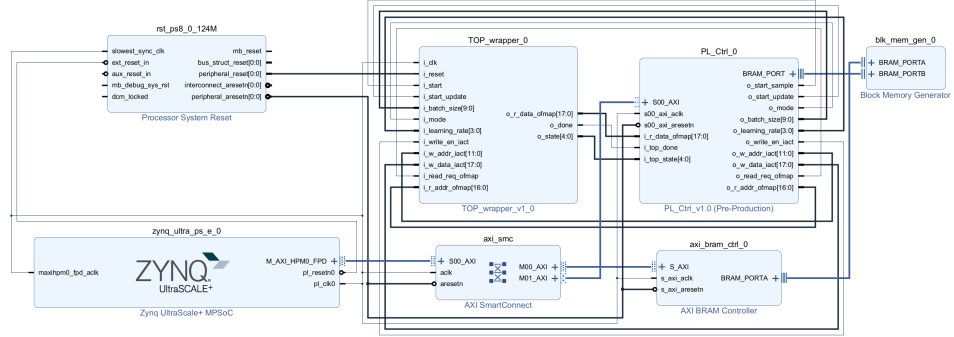


Figure 5.2: Vivado block design.

fully connected layer and Softmax-related computation, also involve multiplication operations and require 24 DSP slices. Therefore, the accelerator uses 408 DSP slices in total.

Summary

Resource	Utilization	Available	Utilization %
LUT	156880	230400	68.09
LUTRAM	8570	101760	8.42
FF	219851	460800	47.71
BRAM	183.50	312	58.81
URAM	62	96	64.58
DSP	408	1728	23.61

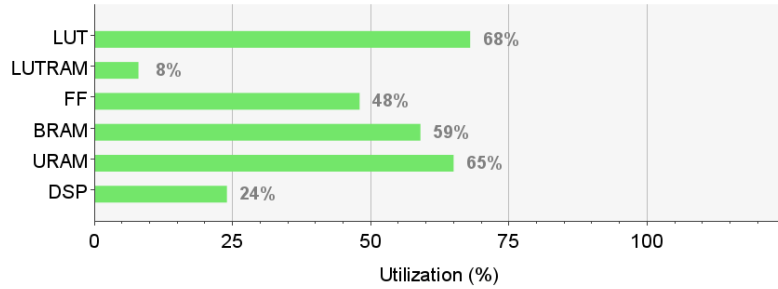


Figure 5.3: Utilization report.

The operating frequency of the accelerator is set to 166.67 MHz, corresponding to a clock period of 6 ns. After implementation in Vivado, timing analysis was performed, and the generated timing report is shown in Fig. 5.4. According to the report, the accelerator satisfies the timing requirements at the target clock frequency. During timing optimization, several techniques were adopted to im-

prove timing closure, including register insertion on critical paths and duplication of high-fanout sources. After optimization, both the setup slack and hold slack remain positive, indicating that no timing violations occur in the implemented design.

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0.030 ns	Worst Hold Slack (WHS): 0.010 ns	Worst Pulse Width Slack (WPWS): 1.500 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 768597	Total Number of Endpoints: 768597	Total Number of Endpoints: 238836

All user specified timing constraints are met.

Figure 5.4: Timing report.

After implementation, post-implementation simulation was performed to obtain the switching activity of the implemented netlist. The switching activity was dumped into SAIF files and then back-annotated into Vivado for power analysis. The generated power reports before and after enabling stochastic pruning are shown in Fig. 5.5 and Fig. 5.6, respectively.

Before pruning, the total on-chip power is 4.829 W, including 4.121 W of dynamic power and 0.709 W of static power. After pruning is enabled, the total on-chip power decreases to 4.815 W, including 4.106 W of dynamic power and 0.709 W of static power. The reduction mainly comes from the decrease in dynamic power, while the static power remains unchanged because it is primarily determined by the device leakage and implemented resources.

Since the PS is only used for system verification and does not participate in accelerator computation, its power consumption is excluded when evaluating the accelerator itself. In both reports, the PS consumes 2.630 W of dynamic power and 0.100 W of static power. Therefore, the PL accelerator power is 2.099 W before pruning and 2.085 W after pruning, corresponding to a reduction of 0.014 W.

The power reduction is relatively small because the current implementation only applies pruning to the activation-gradient data, while the PE array has not yet implemented zero-skipping for pruned gradients. Therefore, the main benefit of pruning in this work is the reduction of non-zero gradient data in the backward path. Further power reduction can be expected if the pruning result is combined with zero-skipping computation and sparse memory access.

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power: 4.829 W
Design Power Budget: Not Specified
Process: typical
Power Budget Margin: N/A
Junction Temperature: 29.7°C
 Thermal Margin: 70.3°C (70.4 W)
 Ambient Temperature: 25.0 °C
 Effective θ_{JA} : 1.0°C/W
 Power supplied to off-chip devices: 0 W
 Confidence level: High
[Launch Power Constraint Advisor](#) to find and fix invalid switching activity

On-Chip Power

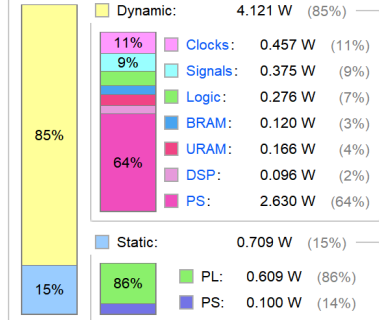


Figure 5.5: Post-implementation power report before enabling stochastic pruning.

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power: 4.815 W
Design Power Budget: Not Specified
Process: typical
Power Budget Margin: N/A
Junction Temperature: 29.7°C
 Thermal Margin: 70.3°C (70.4 W)
 Ambient Temperature: 25.0 °C
 Effective θ_{JA} : 1.0°C/W
 Power supplied to off-chip devices: 0 W
 Confidence level: High
[Launch Power Constraint Advisor](#) to find and fix invalid switching activity

On-Chip Power

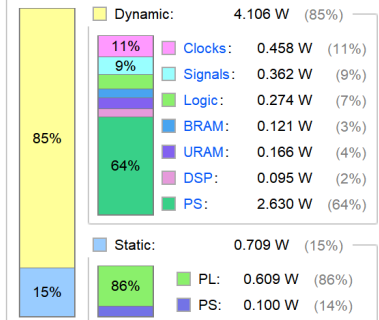


Figure 5.6: Post-implementation power report after enabling stochastic pruning.

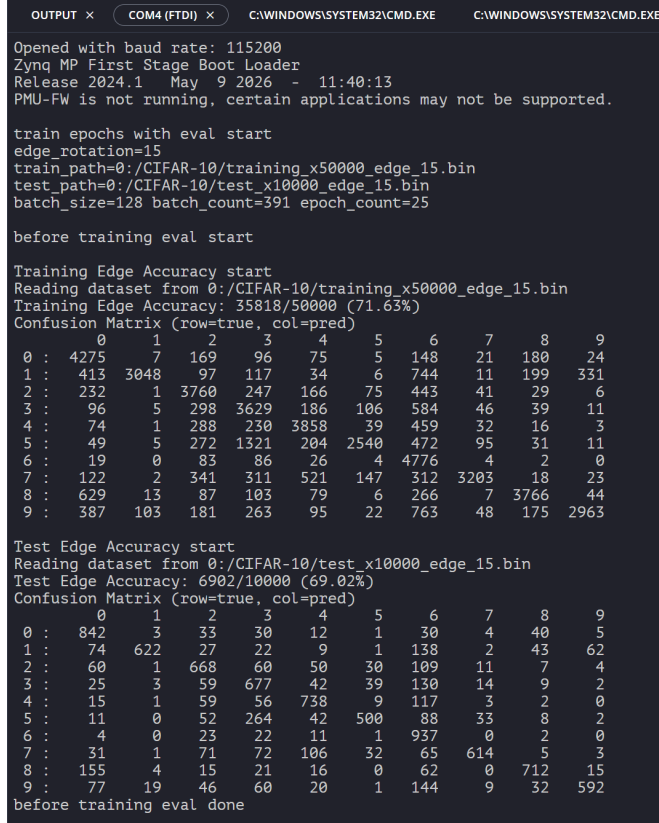
5.4 Functional Verification

To verify the complete operation of the proposed accelerator on the FPGA platform, both inference mode and training mode are tested on the transformed CIFAR-10 dataset. The transformed dataset contains 50,000 training images and 10,000 test images, and the batch size is set to 128 during training. Therefore, each epoch contains 391 batches, and the on-board training experiment is conducted for 30 epochs in total.

As discussed in the previous section, the dataset is stored on the SD card and transferred to the PL-side input buffer through the PS. After the PL completes inference or training, the loss, accuracy, and confusion matrix are printed through

the UART interface.

Fig. 5.7 shows an example UART output in inference mode. According to the UART output, the pretrained parameters achieve a training accuracy of 71.63% and a test accuracy of 69.02% on the transformed dataset. This result is used as the initial baseline before on-chip training.



```

OUTPUT x COM4 (FTDI) x C:\WINDOWS\SYSTEM32\CMD.EXE C:\WINDOWS\SYSTEM32\CMD.EXE
Opened with baud rate: 115200
Zynq MP First Stage Boot Loader
Release 2024.1 May 9 2026 - 11:40:13
PMU-FW is not running, certain applications may not be supported.

train epochs with eval start
edge_rotation=15
train_path=0:/CIFAR-10/training_x50000_edge_15.bin
test_path=0:/CIFAR-10/test_x10000_edge_15.bin
batch_size=128 batch_count=391 epoch_count=25

before training eval start

Training Edge Accuracy start
Reading dataset from 0:/CIFAR-10/training_x50000_edge_15.bin
Training Edge Accuracy: 35818/50000 (71.63%)
Confusion Matrix (row=true, col=pred)
  0 1 2 3 4 5 6 7 8 9
0 : 4275 7 169 96 75 5 148 21 180 24
1 : 413 3048 97 117 34 6 744 11 199 331
2 : 232 1 3760 247 166 75 443 41 29 6
3 : 96 5 298 3629 186 106 584 46 39 11
4 : 74 1 288 230 3858 39 459 32 16 3
5 : 49 5 272 1321 204 2540 472 95 31 11
6 : 19 0 83 86 26 4 4776 4 2 0
7 : 122 2 341 311 521 147 312 3203 18 23
8 : 629 13 87 103 79 6 266 7 3766 44
9 : 387 103 181 263 95 22 763 48 175 2963

Test Edge Accuracy start
Reading dataset from 0:/CIFAR-10/test_x10000_edge_15.bin
Test Edge Accuracy: 6902/10000 (69.02%)
Confusion Matrix (row=true, col=pred)
  0 1 2 3 4 5 6 7 8 9
0 : 842 3 33 30 12 1 30 4 40 5
1 : 74 622 27 22 9 1 138 2 43 62
2 : 60 1 668 60 50 30 109 11 7 4
3 : 25 3 59 677 42 39 130 14 9 2
4 : 15 1 59 56 738 9 117 3 2 0
5 : 11 0 52 264 42 500 88 33 8 2
6 : 4 0 23 22 11 1 937 0 2 0
7 : 31 1 71 72 106 32 65 614 5 3
8 : 155 4 15 21 16 0 62 0 712 15
9 : 77 19 46 60 20 1 144 9 32 592

before training eval done

```

Figure 5.7: UART output example in inference mode before on-chip training.

Fig. 5.8 shows an example UART output during training mode. The accelerator performs multi-epoch training and evaluates the training and test accuracy after each epoch. This demonstrates that the complete on-chip training flow can run correctly on the FPGA platform.

To further evaluate the effect of stochastic pruning in the RTL implementation, the training process is tested with pruning disabled and enabled. The corresponding loss curves are shown in Fig. 5.9. In both cases, the training loss decreases during the first several epochs, which indicates that the RTL training datapath is able to update the model parameters effectively. When pruning is enabled, the training loss still converges, but the test loss becomes higher in the later epochs. This suggests that pruning reduces part of the small activation-gradient information

```

COM4 (FTDI) x C:\WINDOWS\SYSTEM32\CMD.EXE C:\WINDOWS\SYSTEM32\CMD.EXE C:\WI
Test Edge Loss start
Reading dataset from 0:/CIFAR-10/test_x10000_edge_15.bin
Test Edge Loss: loss=0.502 samples=10000
after epoch eval done

one epoch training start
epoch=13/30 learning_rate=5 train samples=50000
epoch 13 average training loss=0.238 samples=50000
one epoch training done
epoch 13/30 done

after epoch eval start

Training Edge Accuracy start
Reading dataset from 0:/CIFAR-10/training_x50000_edge_15.bin
Training Edge Accuracy: 46140/50000 (92.28%)
Confusion Matrix (row=true, col=pred)
  0   1   2   3   4   5   6   7   8   9
0 : 4676 13  92  48  23  4  11  32  73  28
1 :  20 4775  8  9  1  8  7  9  39 124
2 :  88  4 4575 105  69 52 47 45 12  3
3 :  18  4  95 4417  74 234 56 76 14 12
4 :  23  2  83  104 4530  59 34 149 12  4
5 :  9  3  71  434  70 4226 27 151  3  6
6 :  13  6  96  124  66 29 4628 23  6  9
7 :  20  0  26  48  43 44  8 4798  4  9
8 :  86 26  14  34  12  5  4  8 4768 43
9 :  41 72  18  43  4  4 10 31 30 4747

Test Edge Accuracy start
Reading dataset from 0:/CIFAR-10/test_x10000_edge_15.bin
Test Edge Accuracy: 8482/10000 (84.82%)
Confusion Matrix (row=true, col=pred)
  0   1   2   3   4   5   6   7   8   9
0 :  844 10  34  19 12  2  2 18 40 19
1 :  11 901  7  10  0  0  3  4 11 53
2 :  29  0 810  38 40 32 24 19  6  2
3 :  12  5  36 748 26 93 29 30 11 10
4 :  10  1  24  39 848 18 12 45  3  0
5 :  9  1  30 117 23 759  8 49  3  1
6 :  7  3  24  33 25 10 887  6  2  3
7 :  7  0 18  14 17 31  3 901  1  8
8 : 46 12  8  6  1  1  4  5 901 16
9 : 26 42  7  9  1  1  5 12 14 883

Test Edge Loss start
Reading dataset from 0:/CIFAR-10/test_x10000_edge_15.bin
Test Edge Loss: loss=0.508 samples=10000
after epoch eval done

one epoch training start

```

Figure 5.8: UART output example during on-chip training.

and slightly weakens the fine-grained adjustment ability of the model.

The corresponding accuracy curves are shown in Fig. 5.10. Starting from the initial test accuracy of 69.02%, both configurations recover the model accuracy to above 84%. Without pruning, the highest test accuracy reaches 85.15%. With pruning enabled, the highest test accuracy reaches 84.85%. Therefore, stochastic pruning only introduces a 0.30% accuracy drop in the RTL on-board training result.

The RTL results are also compared with the floating-point PyTorch training results discussed in Chapter 3. The peak accuracy comparison is summarized in Table 5.2. In floating-point training, pruning reduces the best test accuracy from 86.22% to 86.08%, corresponding to a 0.14% drop. In RTL training, the corresponding drop is 0.30%. Although the accuracy degradation is slightly larger in RTL, the overall

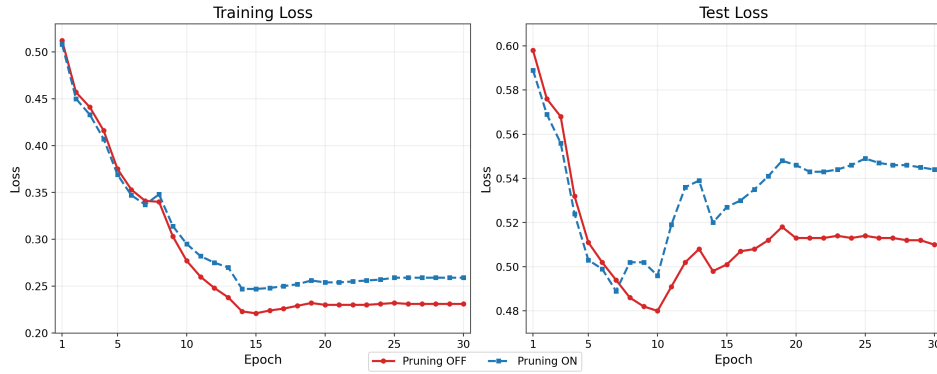


Figure 5.9: Training and test loss comparison of RTL on-chip training with pruning disabled and enabled.

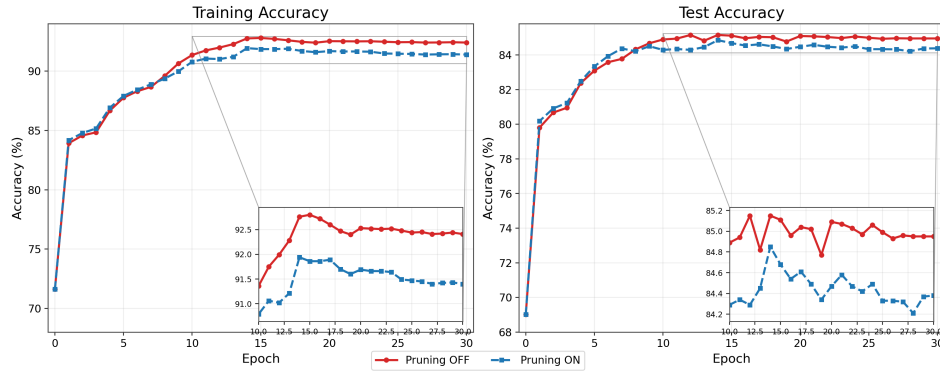


Figure 5.10: Training and test accuracy comparison of RTL on-chip training with pruning disabled and enabled.

impact remains small.

Table 5.2: Peak test accuracy comparison between floating-point training and RTL on-board training

Training Type	Before Training	After Training		Pruning Accuracy Drop
		Pruning OFF	Pruning ON	
PyTorch	68.81%	86.22%	86.08%	0.14%
FPGA	69.02%	85.15%	84.85%	0.30%

Compared with floating-point training, the RTL implementation has a lower peak accuracy. Without pruning, the RTL result is 1.07% lower than the floating-point result. With pruning enabled, the RTL result is 1.23% lower than the floating-point result. This difference mainly comes from fixed-point quantization and limited-

precision parameter update.

Overall, the on-board results show that the proposed accelerator can correctly execute both inference and training on the transformed CIFAR-10 dataset. The on-chip training process improves the test accuracy from 69.02% to 85.15% without pruning, and still reaches 84.85% when stochastic pruning is enabled. This confirms that the pruning mechanism can reduce the activation-gradient workload while maintaining stable training behavior and only a minor loss in final accuracy.

5.5 Performance Analysis

The overall performance of the proposed training framework is further evaluated. First, Table 5.3 lists the number of MAC operations required by each layer of the custom network during one inference-training process. In this work, one MAC is counted as two operations. Based on this calculation, the total computational workload of one inference-training pass is approximately 181.08 MOPs.

Table 5.3: MAC count of the CNN training workload.

Layer	OFMAP Size	Kernel	Cin	Cout	MACs
conv1	32×32	3×3	3	32	884,736
conv2	32×32	3×3	32	32	9,437,184
conv3	16×16	3×3	32	64	4,718,592
conv4	16×16	3×3	64	64	9,437,184
conv5	8×8	3×3	64	128	4,718,592
conv6	8×8	3×3	128	128	9,437,184
FC	10	128	128	10	1,280
Forward total					38,634,752
FC dW	10×128	1	128	10	1,280
FC dX	128	10	10	128	1,280
conv6 dW	3×3	8×8	128	128	9,437,184
conv6 dX	8×8	3×3	128	128	9,437,184
conv5 dW	3×3	8×8	64	128	4,718,592
conv5 dX	8×8	3×3	64	128	4,718,592
conv4 dW	3×3	16×16	64	64	9,437,184
conv4 dX	16×16	3×3	64	64	9,437,184
conv3 dW	3×3	16×16	32	64	4,718,592
Backward total					51,907,072
Training total					90,541,824

MACs are computed as $H_{out}W_{out}C_{in}C_{out}K_hK_w$.

The execution latency of the complete inference-training flow is obtained through simulation. The complete inference-training process requires 1,376,396 clock cy-

cles, including 475,513 cycles for the forward pass and 900,883 cycles for the backward pass. At the target frequency of 166.67 MHz, these correspond to approximately 2.85 ms and 5.41 ms, respectively, and the total execution latency is about 8.26 ms. It should be noted that the simulated latency includes not only the layers involving MAC operations, but also other layers such as max-pooling, ReLU, average-pooling and pruning, as well as the time spent on memory access. Since these non-MAC layers and memory access do not contribute to the MAC-based operation count, their execution time is included in the total latency but is not reflected in the total number of operations.

Finally, the verification system implemented on the ZCU104 development board achieves an average throughput of 21.93 Giga Operations Per Second (GOPS), as summarized in Table 5.4. Based on the post-implementation power analysis, the accelerator power is 2.099 W before pruning and 2.085 W after pruning. Therefore, the corresponding energy efficiencies are 10.45 GOPS/W and 10.52 GOPS/W, respectively.

Table 5.4: Effective throughput of the CNN training workload.

Metric	Forward	Backward	Total
MACs	38,634,752	51,907,072	90,541,824
Operations	77,269,504	103,814,144	181,083,648
Cycles	475,513	900,883	1,376,396
Latency	2.85 ms	5.41 ms	8.26 ms
Throughput	27.08 GOPS	19.21 GOPS	21.93 GOPS

One MAC is counted as two operations.

As shown in Table 5.5, compared with existing designs, this work achieves competitive energy efficiency through a targeted data reuse strategy and lower DSP resource requirements, which reduce the power consumption caused by data movement. Since the proposed design uses fewer DSP resources, its absolute throughput is lower than designs with larger computing arrays. Nevertheless, the energy efficiency of the proposed accelerator reaches 10.52 GOPS/W, which is higher than most related works and is only lower than that reported in DATE 2023 [78].

To further evaluate the efficiency under resource constraints, the DSP-normalized energy efficiency is also calculated. This metric divides the energy efficiency by the number of DSPs in thousands, indicating the energy-efficient throughput achieved per unit DSP resource. Under this metric, the proposed design achieves 25.78 GOPS/(W·kDSP), which is the highest among the compared designs. This result shows that the accelerator is especially suitable for low-power and resource-limited on-device training scenarios.

Table 5.5: Comparison with FPGA-based CNN training accelerators.

Work	ICFPT 2017 [79]	DATE 2023 [78]	FPL 2019 [80]	TODAES 2022 [81]	Ours
Platform	ZU19EG	XCVU9P	Stratix 10 GX	ZCU102	ZCU104
Dataset	CIFAR-10	Tiny ImageNet	CIFAR-10	ImageNet	CIFAR-10
Precision	FP 32	FP 16/32	FXP 16	FP 32	FXP 9/18
Network	LeNet-10	ResNet-18	Custom	VGG-16	VGG-like
Freq. (MHz)	200	180	240	100	166.67
DSP	1,500	1,216	1,699	1,508	408
Throughput (GOPS)	86.12	299.86	163	46.99	21.93
DSP Eff. (GOPS/kDSP)	57.41	246.6	95.94	31.16	53.75
Power (W)	14.24	17.94	10.28	7.712	2.085
Energy Eff. (GOPS/W)	6.05	16.71	7.90	6.09	10.52
Energy Eff. /kDSP	4.03	13.74	4.65	4.04	25.78

Conclusions and Future Works

6.1 Conclusions

This thesis presents an FPGA-based on-device fine-tuning accelerator for edge CNN adaptation. The main objective is to enable a deployed CNN model to recover its accuracy under domain shift without relying on cloud-side retraining. To achieve this goal, the proposed design combines a hardware-friendly fixed-point training datapath with a spatial accelerator architecture implemented and verified on the ZCU104 FPGA platform.

The accelerator adopts a row-stationary dataflow and adaptive spatial unrolling to improve on-chip data reuse and maintain PE utilization across different convolution layers. The same computation fabric is reused for forward propagation, activation-gradient calculation, and weight-gradient calculation. This organization reduces unnecessary data movement and allows the training datapath to fit within the resource constraints of the target FPGA platform.

To improve the robustness and efficiency of low-precision training, stochastic rounding and stochastic approximate pruning are introduced into the hardware datapath. Stochastic rounding is used to reduce quantization bias during fixed-point scaling, while stochastic pruning reduces the amount of non-zero activation-gradient data in the backward path. The RTL pruning result shows that the activation-gradient sparsity increases to about 70.25%, indicating that the pruning mechanism can effectively expose gradient sparsity in hardware.

The complete system is verified on the transformed CIFAR-10 dataset. The accelerator successfully supports both inference and on-board training. Without pruning, the highest test accuracy after on-board training reaches 85.15%. When stochastic pruning is enabled, the highest test accuracy reaches 84.85%, with only a 0.30% accuracy drop. This result shows that the pruning mechanism can reduce the backward gradient workload while maintaining stable training behavior.

In terms of performance, the implemented accelerator achieves an average throughput of 21.93 GOPS. After enabling stochastic pruning, the accelerator power is 2.085 W, corresponding to an energy efficiency of 10.52 GOPS/W. Compared with

related FPGA-based CNN training accelerators, the proposed design achieves competitive energy efficiency with relatively low DSP usage and power consumption.

Overall, this work demonstrates the feasibility of performing CNN fine-tuning directly on an FPGA-based edge platform. The results confirm that the proposed accelerator can recover model accuracy under deployment distortion while keeping the computation, resource usage, and power consumption within practical embedded-system constraints.

6.2 Future Works

Although the proposed accelerator has been functionally verified on the FPGA platform, there are still several directions for further improvement.

- **Zero-Skip Computation.** In the current design, stochastic pruning can produce a large number of zero activation gradients, but the PE array still follows a dense computation schedule. Future work can introduce zero-skip computation into the training flow. When a pruned gradient becomes zero, the corresponding MAC operation can be skipped, reducing unnecessary computation and improving throughput.
- **Sparse Gradient Storage.** At present, zero gradients are still stored and transferred in a dense format. A sparse gradient buffer or compressed write-back format can be introduced to avoid storing and reading these zero values. This would reduce the usage of gradient-buffer, lower the memory traffic on the chip, and further decrease power consumption during training on the device.
- **Flexible PE Mapping.** The current PE cluster size and spatial unrolling strategy are mainly optimized for the CIFAR-10 input size and the target VGG-like network. A more flexible mapping mechanism could adapt the PE array to different image resolutions, layer shapes, and channel configurations. This would improve the scalability of the accelerator for a wider range.

References

- [1] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *arXiv preprint arXiv:1704.04861*, 2017.
- [2] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, “Squeezenet: Alexnet-level accuracy with 50x fewer parameters and < 0.5 mb model size,” *arXiv preprint arXiv:1602.07360*, 2016.
- [3] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, “Shufflenet v2: Practical guidelines for efficient cnn architecture design,” in *Proceedings of the European conference on computer vision (ECCV)*, pp. 116–131, 2018.
- [4] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” in *International Conference on Learning Representations (ICLR)*, 2016.
- [5] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, pp. 2704–2713, 2018.
- [6] D. Hendrycks and T. Dietterich, “Benchmarking neural network robustness to common corruptions and perturbations,” in *International Conference on Learning Representations (ICLR)*, 2019.
- [7] D. Wang, E. Shelhamer, S. Liu, B. Olshausen, and T. Darrell, “Tent: Fully test-time adaptation by entropy minimization,” in *International Conference on Learning Representations (ICLR)*, 2021.
- [8] S. Shi, Q. Yang, Y. Xiang, S. Qi, and X. Wang, “An efficient split fine-tuning framework for edge and cloud collaborative learning,” *arXiv preprint arXiv:2203.15332*, 2022.
- [9] S. Zhu, T. Voigt, F. Rahimian, and J. Ko, “On-device training,” *ACM Transactions on Sensor Networks*, vol. 20, no. 6, 2024.

- [10] D. Vucetic, M. Tayaranian, M. Ziaeeefard, J. J. Clark, B. H. Meyer, and W. J. Gross, “Efficient fine-tuning of bert models on the edge,” in *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1838–1842, 2022.
- [11] Y. Guo, N. C. Codella, L. Karlinsky, J. V. Codella, J. R. Smith, K. Saenko, T. Rosing, and R. Feris, “A broader study of cross-domain few-shot learning,” in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2019.
- [12] T. Chen, B. Xu, C. Zhang, and C. Guestrin, “Training deep nets with sub-linear memory cost,” *arXiv preprint arXiv:1604.06174*, 2016.
- [13] H. Cai, C. Gan, L. Zhu, and S. Han, “Tinytl: Reduce memory, not parameters for efficient on-device learning,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.
- [14] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang, C. Gan, and S. Han, “On-device training under 256kb memory,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.
- [15] S. Sarkar, S. Kundu, K. Zheng, and P. A. Beerel, “Block selective reprogramming for on-device training of vision transformers,” in *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 8094–8103, 2024.
- [16] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, “How transferable are features in deep neural networks?,” in *Advances in Neural Information Processing Systems*, vol. 27, pp. 3320–3328, 2014.
- [17] A. Brock, T. Lim, J. Ritchie, and N. Weston, “Freezeout: Accelerate training by progressively freezing layers,” *arXiv preprint arXiv:1706.04983*, 2017.
- [18] S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy, and D. S. Modha, “Learned step size quantization,” in *International Conference on Learning Representations (ICLR)*, 2020.
- [19] O. Kuchaiev, B. Ginsburg, I. Gitman, V. Lavrukhin, J. Li, H. Nguyen, C. Case, and P. Micikevicius, “Mixed-precision training for nlp and speech recognition with openseq2seq,” *arXiv preprint arXiv:1805.10387*, 2018.
- [20] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep learning with limited numerical precision,” in *International Conference on Machine Learning (ICML)*, pp. 1737–1746, 2015.
- [21] L. Xia, M. Anthonissen, M. Hochstenbach, and B. Koren, “A simple and efficient stochastic rounding method for training neural networks in low precision,” *arXiv preprint arXiv:2103.02438*, 2021.
- [22] S. B. Ali, S.-I. Filip, and O. Sentieys, “A stochastic rounding-enabled low-precision floating-point mac for dnn training,” in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, IEEE, 2024.

- [23] S. Srinivas, A. Subramanya, and R. V. Babu, "Training sparse neural networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 455–462, 2017.
- [24] C. Louizos, M. Welling, and D. P. Kingma, "Learning sparse neural networks through l_0 regularization," *arXiv preprint arXiv:1712.01312*, 2017.
- [25] J. W. Siegel, J. Chen, P. Zhang, and J. Xu, "Training sparse neural networks using compressed sensing," *arXiv preprint arXiv:2008.10300*, 2020.
- [26] H. Mostafa and X. Wang, "Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization," *arXiv preprint arXiv:1902.05967*, 2019.
- [27] U. Evci, T. Gale, J. Menick, P. S. Castro, and E. Elsen, "Rigging the lottery: Making all tickets winners," *arXiv preprint arXiv:1911.11134*, 2019.
- [28] A. I. Nowak, B. Grooten, D. C. Mocanu, and J. Tabor, "Fantastic weights and how to find them: Where to prune in dynamic sparse training," *arXiv preprint arXiv:2311.14487*, 2023.
- [29] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," *arXiv preprint arXiv:1712.01887*, 2017.
- [30] B. McDanel, H. Dinh, and J. Magallanes, "Accelerating dnn training with structured data gradient pruning," in *2022 26th International Conference on Pattern Recognition (ICPR)*, pp. 2293–2299, 2022.
- [31] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.
- [32] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, and P. Dubey, "Scaleddeep: A scalable compute architecture for learning and evaluating deep networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 13–26, 2017.
- [33] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "14.5 eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 262–263, IEEE, 2016.
- [34] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks," *arXiv preprint arXiv:1807.07928*, 2018.
- [35] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

- [36] T. N. Sainath, A.-r. Mohamed, B. Kingsbury, and B. Ramabhadran, “Deep convolutional neural networks for LVCSR,” in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 8614–8618, IEEE, 2013.
- [37] S. Levine, C. Finn, T. Darrell, and P. Abbeel, “End-to-end training of deep visuomotor policies,” *arXiv preprint arXiv:1504.00702*, 2015.
- [38] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
- [39] Y. LeCun and G. Hinton, “Deep learning,” *Nature*, vol. 521, pp. 436–44, 05 2015.
- [40] S. Krig, *Computer Vision Metrics: Survey, Taxonomy and Analysis of Computer Vision, Visual Neuroscience, and Deep Learning*. Springer, textbook ed., 2016.
- [41] V. Dumoulin and F. Visin, “A guide to convolution arithmetic for deep learning,” *arXiv preprint arXiv:1603.07285*, 2016.
- [42] S. Williams, A. Waterman, and D. Patterson, “Roofline,” *Communications of the ACM*, vol. 52, no. 4, p. 65, 2009.
- [43] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *International conference on machine learning*, pp. 448–456, PMLR, 2015.
- [44] B. Byers and A. Sheta, “Design of convolutional neural networks for fish recognition and tracking,” vol. 22, pp. 1–9, 2023.
- [45] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.
- [46] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feed-forward neural networks,” in *International Conference on Artificial Intelligence and Statistics*, 2010.
- [47] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri, “Activation functions in deep learning: A comprehensive survey and benchmark,” *Neurocomputing*, vol. 503, pp. 92–108, 2022.
- [48] X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier neural networks,” in *International Conference on Artificial Intelligence and Statistics*, 2011.
- [49] A. L. Maas, A. Y. Hannun, and A. Y. Ng, “Rectifier nonlinearities improve neural network acoustic models,” in *Proceedings of the 30th International Conference on Machine Learning*, vol. 28, 2013.
- [50] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,” in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pp. 1026–1034, 2015.

- [51] D. Scherer, A. Müller, and S. Behnke, “Evaluation of pooling operations in convolutional architectures for object recognition,” in *Artificial Neural Networks – ICANN 2010*, pp. 92–101, Springer, 2010.
- [52] Y.-L. Boureau, J. Ponce, and Y. LeCun, “A theoretical analysis of feature pooling in visual recognition,” in *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pp. 111–118, 2010.
- [53] M. Ihme, W. T. Chung, and A. A. Mishra, “Combustion machine learning: Principles, progress and prospects,” *Progress in Energy and Combustion Science*, vol. 91, p. 101010, 2022.
- [54] M. Lin, Q. Chen, and S. Yan, “Network in network,” 2013.
- [55] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.
- [56] C. M. Bishop, *Pattern recognition and machine learning*. Springer, 2006.
- [57] D. Ha, M. Kim, K. Moon, and C. Y. Jeong, “Accelerating on-device learning with layer-wise processor selection method on unified memory,” *Sensors*, vol. 21, no. 7, p. 2364, 2021.
- [58] J. Lee and H.-J. Yoo, “An overview of energy-efficient hardware accelerators for on-device deep-neural-network training,” *IEEE Open Journal of the Solid-State Circuits Society*, vol. 1, pp. 115–128, 2021.
- [59] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, “ZeRO: Memory optimizations toward training trillion parameter models,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16, 2020.
- [60] A. Parashar, P. Raina, Y. Shao, Y.-H. Chen, V. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. Keckler, and J. Emer, “Timeloop: A systematic approach to DNN accelerator evaluation,” in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 304–315, 2019.
- [61] M. Hassaballah and A. I. Awad, *Deep Learning in Computer Vision: Principles and Applications*. CRC Press, 3 2020.
- [62] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei, “Deep Convolutional Neural Network Architecture With Reconfigurable Computation Patterns,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 8, pp. 2220–2233, 2017.
- [63] R. Xu, S. Ma, Y. Guo, and D. Li, “A survey of design and optimization for systolic array-based dnn accelerators,” *ACM Computing Surveys (CSUR)*, vol. 56, no. 1, 2023.
- [64] Y.-H. Chen, J. Emer, and V. Sze, “Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks,” in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 367–379, IEEE, 2016.

- [65] Y.-H. Chen, J. Emer, and V. Sze, “Using dataflow to optimize energy efficiency of deep neural network accelerators,” *IEEE Micro*, vol. 37, no. 3, pp. 12–21, 2017.
- [66] C. Park, S. Park, and C. S. Park, “Roofline-model-based design space exploration for dataflow techniques of CNN accelerators,” *IEEE Access*, vol. 8, pp. 172509–172523, 2020.
- [67] M. Wang and W. Deng, “Deep visual domain adaptation,” *Neurocomputing*, vol. 312, p. 135, 2018.
- [68] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.
- [69] A. Krizhevsky, “Learning multiple layers of features from tiny images,” 2009.
- [70] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer, “I-bert: Integer-only bert quantization,” in *International Conference on Machine Learning (ICML)*, pp. 5506–5518, PMLR, 2021.
- [71] N. N. Schraudolph, “A fast, compact approximation of the exponential function,” *Neural Computation*, vol. 11, no. 4, pp. 853–862, 1999.
- [72] Y. Bengio, N. Léonard, and A. Courville, “Estimating or propagating gradients through stochastic neurons for conditional computation,” *arXiv preprint arXiv:1308.3432*, 2013.
- [73] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, “Mixed precision training,” *arXiv preprint arXiv:1710.03740*, 2017.
- [74] M. Courbariaux, Y. Bengio, and J.-P. David, “Training deep neural networks with low precision multiplications,” *arXiv preprint arXiv:1412.7024*, 2014.
- [75] X. Sun, X. Ren, S. Ma, and H. Wang, “meprop: Sparsified back propagation for accelerated deep learning with reduced overfitting,” in *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pp. 3299–3308, PMLR, 2017.
- [76] J. Wangni, J. Wang, J. Liu, and T. Zhang, “Gradient sparsification for communication-efficient distributed optimization,” *arXiv preprint arXiv:1710.09854*, 2017.
- [77] Y. Chen, T. Krishna, J. S. Emer, and V. Sze, “Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks,” *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, p. 127, 2017.
- [78] C. Fang, W. Sun, A. Zhou, and Z. Wang, “Cest: Computation-efficient n:m sparse training for deep neural networks,” in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–2, 2023.
- [79] Z. Liu, Y. Dou, J. Jiang, Q. Wang, and P. Chow, “An fpga-based processor for training convolutional neural networks,” in *2017 International Conference on Field Programmable Technology (ICFPT)*, pp. 207–210, 2017.

-
- [80] S. Kolala Venkataramanaiah, Y. Ma, S. Yin, E. Nurvithadhi, A. Dasu, Y. Cao, and J.-S. Seo, “Automatic compiler based fpga accelerator for cnn training,” in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 166–172, 2019.
 - [81] Y. Tang, X. Zhang, P. Zhou, and J. Hu, “Ef-train: Enable efficient on-device cnn training on fpga through data reshaping for online adaptation or personalization,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 27, June 2022.



LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2026-1140
<http://www.eit.lth.se>