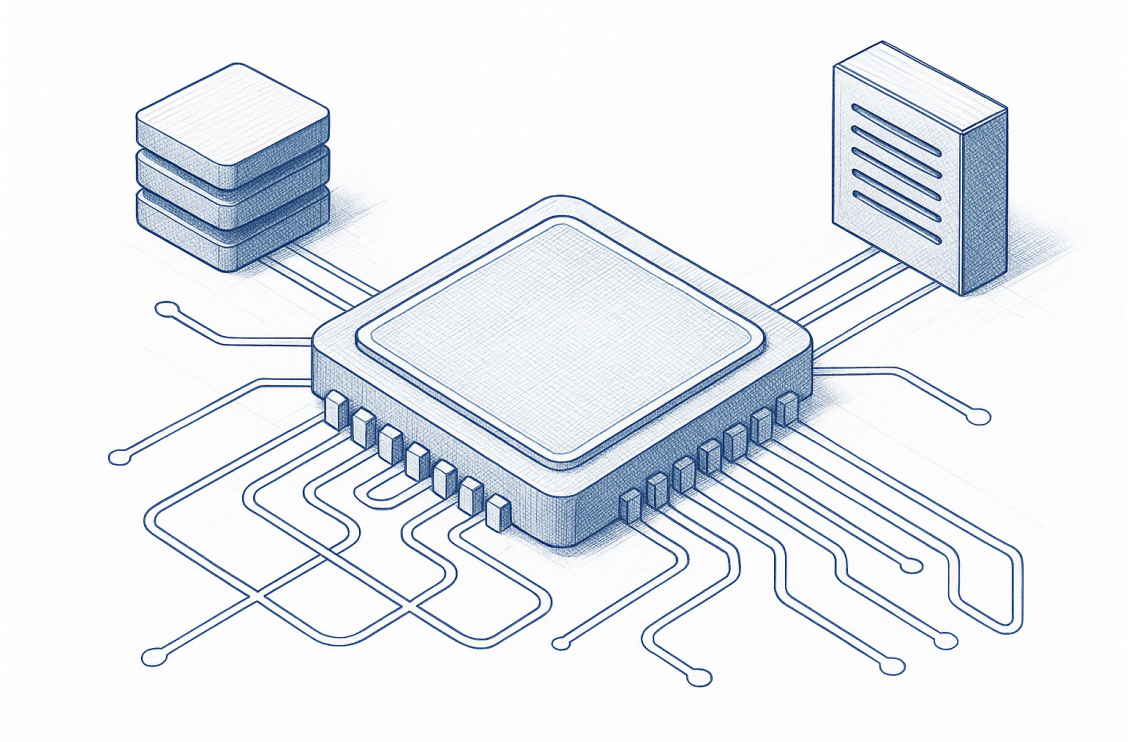


Exploring the Benefits of Compression to Improve Data-intensive Neural Network Inference

Master's Thesis in Computer Systems and Networks

Max Andreas Peter Meidinger

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY

Gothenburg, Sweden 2026

www.chalmers.se

Master's Thesis 2026

Exploring the Benefits of Compression to Improve Data-intensive Neural Network Inference

Max Andreas Peter Meidinger

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Computer and Network Systems
Chalmers University of Technology
Gothenburg, Sweden 2026

Exploring the Benefits of Compression to Improve Data-intensive Neural Network Inference
MAX ANDREAS PETER MEIDINGER

© MAX ANDREAS PETER MEIDINGER, 2026.

Supervisor: Ioannis Sourdis, Computer and Network Systems
Examiner: Ioannis Sourdis, Computer and Network Systems

Master's Thesis 2026
Department of Computer Science and Engineering
Division of Computer and Network Systems
Chalmers University of Technology
SE-415 27 Gothenburg
Telephone +49 1578 4657515

Typeset in LaTeX
Version: commit `2ad9eb15`, built Thursday 29th October, 2026
Gothenburg, Sweden 2026

Abstract

Convolutional Neural Networks (CNNs) have become essential for many computer vision tasks such as image classification, object detection and semantic segmentation. State-of-the-art models come with high computational demands that require efficient acceleration. FPGAs are a popular choice, as they can host energy-efficient, high-throughput inference designs. Streaming dataflow designs are particularly efficient, as they create custom per-layer pipelines which confine most of the data movement on chip. As these accelerators also prefer to keep model parameters on chip, on-chip buffer space becomes a limiting factor. Long skip connections, a ubiquitous structure of today's networks, aggravate this problem further, as they require large amounts of buffer space to synchronize the diverging branches of the model's architecture. Off-chip eviction instead moves the activations of skip connections temporarily to off-chip memory, but this puts additional load on the memory bus, which can lead to stalls in the compute pipeline if the processing engines cannot be fed. This problem can be alleviated using lossless compression, but lossless compression is fundamentally limited.

This thesis explores lossy compression of the evicted activations instead, trading a bounded loss of inference accuracy for a higher compression ratio, without retraining. Starting from SZ1, a prediction-based compressor whose model fits the streaming paradigm, we develop ASZ (Activation SZ) in a design space exploration, which adds block-level error thresholds, new prediction functions, per-channel selection of the prediction functions, and Huffman coding and downsampling stages. We evaluate ASZ against lossless (Huffman coding, RLE) and lossy baselines (quantization with Huffman coding (QH), Lorenzo prediction, SZ1, downsampling, SVD and JPEG) on the 74 skip connections of eight CNN classifiers on ImageNet, CIFAR-10 and CIFAR-100, and estimate the hardware cost of every method.

Lossy compression goes well beyond lossless encoding: on every connection, a lossy method exceeds the best lossless one, on 71 of them at a cost of at most 0.1 pp of top-1 accuracy. Within a budget of 1 pp, the streaming-compatible compressors QH, Lorenzo and ASZ reach geometric-mean compression ratios of 3.9, 3.8 and 3.5, 2.3 to 2.6 times the 1.5 of Huffman coding, while RLE does not compress on average. ASZ improves on SZ1 (2.7), but trails the quantizers on most connections, since SZ1-type compressors spend their bits on the choice of a prediction function rather than on the values or their residuals. JPEG reaches 6.6, but buffers eight rows per channel and needs multipliers on the data path. When several connections are compressed, their errors add up and the ratio within the budget falls, least for the error-bounded Lorenzo and SZ1; with three connections, the best lossy method still stays 1.5 to 4.3 times above Huffman coding. Lossy compression can thus substantially reduce the off-chip traffic of evicted skip connections, and QH, the cheapest lossy compressor and

the best streaming-compatible one in ratio, offers the best trade-off between compression ratio and hardware cost.

Contents

CHAPTER 1 **Introduction 1**

1.1 Motivation 1
1.2 Problem Statement 2
1.3 Approach and Objectives 4
1.4 Contributions 4
1.5 Outline 5

CHAPTER 2 **Background and Related Work 7**

2.1 Architectures for CNN Acceleration 7
2.1.1 CPUs 7
2.1.2 GPUs 7
2.1.3 ASICs 8
2.1.4 FPGAs 8

2.2 CNN Architectures 9
2.3 Mapping CNNs to FPGAs 9
2.3.1 Instruction-based Accelerators 10
2.3.2 Streaming Accelerators 10

2.4 Fitting modern CNN on Accelerators 12
2.4.1 Model Compression 13
2.4.1.1 Pruning 13
2.4.1.2 Quantization 13
2.4.1.3 Distillation 14
2.4.1.4 Low-Rank Approximation 14
2.4.1.5 Model Compression on FPGAs 14
2.4.2 Scaling Accelerators: Off-chip Streaming 15

2.5 Compression Fundamentals 17
2.5.1 Lossless Compression Techniques 17
2.5.1.1 Run-Length Encoding (RLE) 18
2.5.1.2 PNG Filters 18
2.5.1.3 Discrete Cosine Transform (DCT) 19
2.5.1.4 Huffman Coding 19
2.5.2 Lossy Compression Techniques 20
2.5.2.1 Quantization 20
2.5.2.2 Downsampling 21
2.5.2.3 Singular Value Decomposition (SVD) 22
2.5.2.4 Curve-fitting Prediction 23
2.5.3 Compression Pipelines 24
2.5.3.1 Quantization + Huffman 24
2.5.3.2 JPEG 25
2.5.3.3 SZ1 25
2.5.3.4 SZ-1.4 27

2.6 Related Work 28
2.6.1 Compressing Neural-Network Data 28
2.6.1.1 Weights 28
2.6.1.2 Activations during Training 28
2.6.1.3 Activations during Inference: Lossless 29
2.6.1.4 Activations during Inference: Lossy 29
2.6.2 Error-bounded Compression 29
2.6.2.1 Prediction-based SOTA Compressors 29
2.6.2.2 Hardware Implementations 30
2.6.2.3 Error-bounded Compression for Neural Networks 31
2.6.3 Summary 31

CHAPTER 3 **Design Space Exploration 33**

3.1 Methodology 33
3.1.1 Problem Framework 34
3.1.2 Goal 36
3.1.3 Relevant Metrics 36
3.1.4 Measured Metrics 36
3.1.5 Estimated Metrics 37
3.1.6 Metrics out of Scope 38
3.1.7 Experimental Setup 38

3.2 Standard SZ 40
3.3 Block Error Thresholding 44
3.3.1 Definition 45
3.3.2 Parameter Space 46
3.3.3 Block Types 47
3.3.3.1 Linear Blocks 48
3.3.3.2 Spatial Blocks 48
3.3.3.3 Sliding Window 49
3.3.3.4 Comparison of Block Types 49
3.3.4 Hardware Cost 50
3.3.5 Summary 51

3.4 Extended Predictor Set 52
3.4.1 Notation 53
3.4.2 Spatial predictors 54
3.4.2.1 PNG Filters as Predictors 55
3.4.2.2 Spatial variants of standard prediction functions 56
3.4.2.3 Linear Regression Prediction 56
3.4.2.4 Lorenzo Predictor 57
3.4.3 Stochastic Encoding 58
3.4.4 Selected Prediction Function Combinations 59
3.4.5 Hardware Cost 63

3.5 Dynamic function selection 66
3.5.1 Selection Algorithm Description 66
3.5.2 Selection Results 69

3.6 Lossless and Pre-processing Stages 71
3.6.1 Downsampling 72
3.6.2 Huffman 74

3.7 Parameter Optimization 78
3.7.1 Initial Parameter Selection 79
3.7.1.1 Block Error Threshold 79
3.7.1.2 Value Error Threshold 79
3.7.1.3 Block Size 80

3.7.2 Search Procedure 82
3.7.3 Generalization to Unseen Data 85

3.8 Conclusion 87
3.8.1 Final Design Points 89
3.8.2 Hardware Cost 90
3.8.3 Limitations 91

CHAPTER 4 **Evaluation 93**

4.1 Evaluation Methodology 93
4.1.1 Datasets and Models 94
4.1.2 DSE Design Points 94
4.1.3 Baselines 95
4.1.3.1 Huffman 96
4.1.3.2 RLE 96
4.1.3.3 Downsampling 97
4.1.3.4 SVD 98
4.1.3.5 JPEG 98
4.1.3.6 Quantization + Huffman 98
4.1.3.7 Lorenzo + Huffman 98
4.1.3.8 SZ1 + Huffman 99
4.1.4 Evaluation Setup and Presentation of Results 99

4.2 Individual Layer Compression 100
4.2.1 ImageNet 101
4.2.1.1 RegNetX-400MF 101
4.2.1.2 ResNet-18 104
4.2.1.3 MobileNetV2 107
4.2.1.4 MobileNetV3-Small 109
4.2.2 Cifar100 111
4.2.2.1 MobileNetV2 111
4.2.2.2 ResNet-20 113
4.2.3 Cifar10 115
4.2.3.1 MobileNetV2 115
4.2.3.2 ResNet-20 117
4.2.4 All Networks 118

4.3 Use Cases: Compression of Multiple Layers 121
4.4 Hardware Cost Estimates 126
4.4.1 Suitability for Streaming 131

4.5 Summary and Discussion 132

CHAPTER 5 **Conclusion 137**

5.1 Overview of work 137
5.2 Findings 138
5.2.1 Characterization of Skip-Connection Activations 138
5.2.2 Adaptation of SZ1 138
5.2.3 Comparison 140
5.2.4 The Requirements Revisited 140

5.3 Reflections 141
5.4 Limitations and Future Work 142

APPENDIX A **Further Experiments and Evaluations 145**

A.1 Profiling 145
A.2 Linear Blocks vs. Sliding Window 146

A.3 Frontloading vs. backloading 148
A.3.1 Block Size and Block Error Interaction 148
A.3.2 Prediction sensitivity to error distribution across feature maps 148
A.3.2.1 Experiment Setup 149
A.3.3 Error Concentration and Block Size 150

A.4 Outlier Preconditions 151
A.5 Stochastic Encoding 153
A.5.1 Fit of the Exponential Distribution 154

A.6 Trajectory Search vs. Grid Sweep 157
A.7 Huffman gain after different lossy stages 158
A.8 Prediction Function Selection by Context Size 162
A.9 Layer Sensitivity 162
A.9.1 Impact of cutting connections 162
A.9.2 Impact of quantizing connections 164
A.9.3 Sensitivity of the evaluated connections 168

A.10 Per-Channel Thresholding 170
A.11 Run-Length Encoding vs. Zero Run-Length Encoding 172
A.11.1 Run Structure of the Evaluated Connections 173

A.12 Number of Lorenzo Layers 176
A.13 Downsampling in Isolation 177
A.14 Best Compression Ratio per Connection 179

APPENDIX B **Model Graphs 181**

References 185

1 Introduction

1.1 Motivation

Since AlexNet's [46] success in the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC), Convolutional Neural Networks (CNNs) have become the de-facto standard for many computer vision tasks, such as classification, object detection and segmentation [68]. To push accuracy further, networks have been growing both deeper and wider: within just three years, the state of the art went from AlexNet's eight layers to the 152 layers of ResNet [32], and modern models such as ConvNeXt V2 [87] scale to hundreds of millions of parameters.

The resulting computational cost has motivated a large body of work on accelerating CNN inference in hardware [12]. FPGAs are a particularly attractive platform, as they can achieve high throughput and energy efficiency while allowing the hardware to be tailored to a specific model [82]. FPGA-based CNN accelerators broadly follow one of two approaches [54]: *Instruction-based accelerators* are "one-size-fits-all" designs that compile a network description into an instruction sequence for a time-multiplexed processing engine, optimized for typical CNN operations [25, 29]. *Streaming accelerators* are instead generated by toolflows such as fpgaConvNet [82] and FINN [81], which build a dedicated compute engine for every layer and connect them into a deep pipeline, resulting in highly parallel designs customized towards a specific platform and model. Streaming accelerators are generally considered more efficient [83] and are the focus of this work.

This specialization comes at the cost of scalability. Weights are preferably preloaded into on-chip Block RAM (BRAM) [92], and every layer occupies its own compute engine, so the on-chip resources of the target device are exhausted quickly; large networks may not fit at all [80]. *Skip connections* aggravate the problem. Introduced with ResNet [32] to counter the *vanishing gradient* problem, they create short-cuts through the network along which the gradient can reliably travel back to the earliest layers, enabling the training of very deep networks. Since then, they have become a defining feature of modern CNNs [68]. In a streaming pipeline, however, the activations travelling along a skip connection have to stay synchronized with the parallel branch, so they must be buffered for the entire depth of the layers they bypass. This is on-chip memory spent purely on transporting data, without any computation taking place, and for long skip connections these buffers alone can exceed the available BRAM [55, 80].

Traditionally, streaming toolflows confine almost all data movement to the chip and use off-chip memory only for the network's inputs and outputs [80]. SMOF [80] breaks with this convention and opens up a trade-off between on-chip memory and off-chip memory bandwidth. Among other techniques, it allows

the activations of long skip connections to be *evicted*: they are written to off-chip memory at the start of the connection and read back where they are needed again. Eviction frees buffer space that the toolflow's optimizer can spend elsewhere, for example on additional parallelism, but it moves the cost to the memory bus, which is shared by all evicted connections, off-loaded weights, and the network's input and output streams. Once the bus saturates, compute engines stall and the pipeline loses its throughput advantage [80]. To reduce this pressure, SMOF encodes evicted data with lightweight lossless schemes, Huffman coding and run-length encoding (RLE), and shows that this yields significant system-level performance gains: a higher compression ratio lowers the bandwidth demand of each evicted connection, and the optimizer can use the resulting slack for further optimizations of the design.

SMOF thereby applies to activations what has long been standard practice for the model itself. To fit the memory and compute budgets of constrained hardware, networks are routinely pruned and quantized before deployment [8, 60]. In their seminal work, Han et al. [31] combine pruning, quantization and Huffman coding to reduce the size of AlexNet and VGG-16 by 35× and 49× without loss of accuracy. Quantization in particular not only shrinks buffers and memory bandwidth requirements, but also replaces expensive 32-bit floating-point arithmetic with cheaper low-precision fixed-point arithmetic [60]. Results like these, however, typically rely on retraining: Han et al. fine-tune the network after pruning and quantization, and quantization-aware training simulates quantization during training so that the network learns to compensate for it [60]. Where training data is unavailable or retraining is impractical, post-training quantization is the alternative, but at low bit widths it can cost considerable accuracy [60].

Error-bounded lossy compression offers a different route. DeepSZ [41] applies SZ, a compressor from scientific computing, to the weights of a network and achieves high compression ratios with little loss of accuracy by bounding the error introduced into each layer. Modern SZ variants, however, rely on elaborate prediction and coding stages whose arithmetic is costly in hardware, and FPGA implementations of such compressors have had to adapt the algorithms considerably to make them hardware-friendly [74, 88]. The simpler prediction model of the original SZ1 [13], on the other hand, lends itself to lightweight hardware implementations [18] and fits the streaming data-flow architectures of accelerators by toolflows such as FPGAConvNet [82] well.

1.2 Problem Statement

The benefit of eviction is bounded by how far the evicted activations can be compressed, and lossless compression, while having significant impact, is fundamentally limited. On the image classifiers studied in this thesis, Huffman coding achieves median compression ratios between roughly 1.2 and 1.8 per model, while RLE frequently *expands* the data instead (chapter 4). Moreover, the ratio achieved by a lossless code depends on the input, so the bandwidth an evicted connection will require cannot be determined exactly at compile time [80].

Lossy compression can go considerably further, as it allows the reconstructed data to deviate from the original in exchange for a more compact representation. Neural networks are known to tolerate such approximation well, as they are inherently highly redundant [41]; the model compression techniques discussed above exploit exactly this redundancy. Our own preliminary experiments (section A.9.2) indicate that the activations of skip connections, too, can

be remarkably robust to approximation. The price is that errors introduced into the activations propagate through the remainder of the network and degrade inference accuracy. Lossy compression of evicted activations therefore turns accuracy into a further resource the toolflow can trade, in this case against memory bandwidth.

The eviction scenario, however, places requirements on a lossy compressor that rule out many established approaches:

No retraining. The compressor is inserted into a model that has already been trained, and typically quantized and pruned, before it is handed to the toolflow. Retraining to recover lost accuracy is not an option: it is costly, and the labeled data required for fine-tuning may not be available [60]. Compression must therefore be applied post hoc and be complementary to any compression the model has already undergone.

Controllable error. For accuracy to be usable as a design parameter, the compressor needs a parameter with a predictable effect on the introduced error, so that the optimizer can stay within a user-specified accuracy budget. Bounding the error alone is, however, not enough, as not only the magnitude of error impacts accuracy, but also its distribution across featuremaps.

Streaming operation. The compressor sits directly in the pipeline, receiving one value per channel every cycle in raster order. Data access patterns for compression should follow this order as much as possible, as accessing data of the feature map in other orders costs buffer space on chip (see section 3.1.1) — the resource that is actually meant to be saved.

Low hardware cost. Compression logic is replicated across the parallel channels of every evicted connection [80]. Resources spent on it are no longer available to the rest of the accelerator, so a compressor that is too expensive offsets the gains it enables.

Transform-based codecs such as JPEG need to buffer whole blocks before they can encode them [21] and are thus not very well-suited for compression within an accelerator. Dictionary coders such as LZW are costly to implement in hardware [44], and most work on compressing neural networks targets their weights or relies on retraining. Lossy compression of evicted activations in streaming accelerators has, to the best of our knowledge, not been explored.

This thesis therefore addresses the following question: *How far can the compression ratio of evicted skip-connection activations in streaming CNN accelerators be increased beyond lossless encoding by trading a bounded loss of inference accuracy, using compressors that meet the requirements above?*

Scope We focus on CNN classifiers whose skip connections merge through element-wise addition, covering both residual [32] and inverted residual [67] connections. All models are pre-trained, quantized to 8 bits and treated as fixed. Since we do not provide hardware implementations, compressors are evaluated in a software simulation of the eviction process, using the compression ratio as a proxy for the reduction in bandwidth demand, complemented by estimates of their hardware cost. The detailed assumptions are given in section 3.1.1.

1.3 Approach and Objectives

Error-bounded lossy compression is well established in scientific computing, where the user specifies a maximum error and the compressor exploits all of this tolerance to increase the compression ratio [14]. The SZ family of compressors has repeatedly set the state of the art in this field [13, 14, 50, 76, 96]. Its first version, SZ1 [13], uses a *prediction-based* model: every value is predicted from its already reconstructed predecessors by one of a small set of prediction functions, and if a prediction lies within the error bound, only a short code identifying the function is stored instead of the value itself. The resulting code stream is then compressed further with a dictionary-based coder.

This model fits the requirements of the eviction scenario remarkably well. Predictions depend only on values that have already arrived, which matches the order in which a streaming pipeline delivers data, and the per-value error bound offers direct control over the introduced error. MemSZ [18] has furthermore shown that SZ-style compression can be implemented in hardware with low latency, high throughput and minimal buffering. Error-bounded compression has been applied to network weights [41] and to activations stored during training [42], but, to the best of our knowledge, not to activations during inference in streaming accelerators (section 2.6.3). We therefore choose it as the focus of this thesis, with the following objectives:

1. **Characterize** the activations of skip connections in CNN classifiers with respect to their compressibility and their tolerance to approximation.

2. **Adapt** the SZ1 compression model to activations in a streaming accelerator and explore its design space, covering error bounding, the set of prediction functions, their selection, and combinations with further compression stages. Since data distributions and error tolerance vary across a pre-trained network [24], compressors are configured per connection.

3. **Compare** the resulting compressor against lossless baselines and alternative lossy schemes in terms of their trade-off between compression ratio and inference accuracy, as well as their estimated hardware cost.

1.4 Contributions

The contributions of this thesis are:

- A first exploration of the trade-off between compression ratio and inference accuracy when applying lossy compression to evicted skip-connection activations in streaming dataflow accelerators.

- ASZ (Activation SZ), a prediction-based compressor derived from SZ1 [13] through an incremental design space exploration, which extends the original with block-level error thresholds, spatial prediction functions, dynamic function selection, and downsampling and Huffman coding stages.

- An evaluation of ASZ against lossless (Huffman, RLE [80]) and lossy (quantization with Huffman coding [31], downsampling [19], SVD [92], JPEG [21], SZ1 [13] and modern SZ derivatives [76]) compression schemes on the skip connections of a range of CNN classifiers and datasets.

1.5 Outline

The remainder of this thesis is structured as follows. Chapter 2 introduces hardware architectures for CNN acceleration with a focus on streaming accelerators and their scaling limitations, and reviews the compression techniques relevant to this work. Chapter 3 defines the problem framework and evaluation methodology, and then develops ASZ step by step, starting from the original SZ1 algorithm. Chapter 4 compares ASZ to lossless and lossy alternatives across models and datasets. Chapter 5 summarizes our findings and discusses the limitations of this work, and closes with directions for future research (section 5.4).

2 Background and Related Work

This thesis lies in the intersection of three fields of study: Hardware acceleration, neural networks and data compression. In this chapter, we will introduce some fundamentals, starting with a assessment of different acceleration platforms for deep neural networks. This will serve as a motivation for our choice of platform, acceleration architecture and finally also compression schemes.

2.1 Architectures for CNN Acceleration

2.1.1 CPUs

Central Processing Units (CPUs) are widely available, supported by all popular machine learning frameworks such as PyTorch and TensorFlow, and backed by optimized linear algebra libraries such as Intel MKL and OpenBLAS [12]. Their large caches and extensive branch control logic make them well suited for control-heavy tasks such as model setup, data preprocessing and orchestrating the other components of a system. Their limited number of cores, however, restricts how much data they can process in parallel, so they struggle with the high-volume multiply-accumulate operations that dominate convolutional layers [12]. While CPUs still dominate DNN inference on low-power edge and IoT devices, they struggle to execute complex DNNs efficiently, which motivates specialized hardware [12]. In addition, CPUs lack the memory bandwidth required to feed CNN workloads [36]: data-center GPUs such as the NVIDIA V100 and A100 reach 900 to 1,555 GB/s [36] through High-Bandwidth Memory (HBM), far beyond what the DDR memory of a typical CPU system provides.

2.1.2 GPUs

Graphics Processing Units (GPUs) are the most mature and most widely used platform for both training and inference of Deep Neural Networks [12, 36]. Their thousands of cores exploit data-level parallelism through the Single Instruction Multiple Thread (SIMT) execution model, which is a good match for the highly parallel Multiply-Accumulate (MAC) operations of matrix multiplications, the central operation in neural networks. A mature software ecosystem, from libraries such as cuBLAS and cuDNN up to the major machine learning frameworks, makes accelerating machine learning workloads on GPUs very accessible [12, 36]; cuBLAS alone is reported to be 6 to 17 times faster than the CPU-based MKL BLAS [36]. Among programmable platforms, GPUs offer the highest computational capacity and memory bandwidth, but this comes at the price of high power consumption: FPGAs and ASICs have fewer computational

and memory resources, but reach moderate performance while using considerably less energy [12, 36]. GPUs are also comparatively inflexible: their data types and structure are fixed, so they cannot adapt to models with changing requirements [36].

For this thesis we relied heavily on the compute power of GPUs.

2.1.3 ASICs

Application-Specific Integrated Circuits (ASICs) are generally the most efficient but least flexible platforms for hardware acceleration. Being customized for a specific application, they offer the best computation speed and energy efficiency of all platforms, at the cost of reconfigurability and scalability [12, 36]. While the high non-recurring engineering (NRE) cost makes ASICs expensive to develop, the cost per unit drops substantially at high production volumes [72]. Their main drawbacks are this inflexibility and their complex and long design process [12, 36]. An FPGA design is mapped onto a fixed, prefabricated fabric; an ASIC design additionally requires custom circuit-level and physical design, followed by fabrication. This lengthens the time to market and adds to the NRE cost.

In the rapidly moving space of Deep Learning, with new models and architectures being released continuously, it does not seem feasible to give up any amount of flexibility, keeping the lengthy design process in mind. Once an ASIC for a specific architecture has been designed and starts production, the state of the art could be far beyond the finished product. Reconfigurable hardware, in contrast, can follow the constantly iterating models without the need to repurchase hardware [36].

Despite this, companies like Taalas doubled down on creating highly specialized chips for one specific model, effectively hardcoding the entire structure into the silicon. Their focus is on a streamlined design process that allows them to achieve a much quicker time-to-market than usual ASIC designs. So far however, their designs focus on the popular transformer architecture and have not been applied to CNNs [75].

2.1.4 FPGAs

While ASICs design results in a fixed architecture that can not be changed after a certain point in the design process, Field-Programmable-Gate-Arrays allow for reconfiguration throughout their lifetime. They are not only adaptable during development, but once deployed can still be reconfigured to fit new scenarios, changed constraints or updated requirements. As a deployment platform for deep learning systems, they could provide tunable trade-offs between critical system parameters, such as performance, power-consumption and cost [82]. This flexibility also exposes more opportunities for optimization to the programmer and allows mapping alternative compute models that more closely fit the problem domain directly to hardware.

This flexibility comes at a cost: Compared to commodity CPUs and GPUs, which only require software to be written and compiled for them, FPGAs require — on top of the initial hardware design — multiple additional steps to realize the mapping from design to platform: logic synthesis, technology mapping, placement, routing and bitstream generation [72]. High-level Synthesis (HLS) languages and automated tool-flows ease the process, but the time required to get to a bitstream is still much higher, and no tooling ecosystem comparable to CUDA exists for FPGA-based deep learning [36].

Compared to ASICs, however, FPGAs are cheaper to develop, have a simpler design flow and a shorter time to market [12].

FPGAs generally fall behind GPUs in terms of raw throughput, but consume considerably less power and area [12, 36], making them a suitable choice for embedded and mobile computing scenarios where resources are limited.

ASICs, in turn, are more energy-efficient than FPGAs, because FPGAs dedicate a significant share of their chip area to reconfigurability [12]. As much as 90% of the area can be attributed to the programmable interconnect alone, and the interconnect accounts for 65% of the power consumption as well [72].

2.2 CNN Architectures

The seminal work in the area of CNNs, the Neocognitron, deviced by Fukushima [23], has a feed-forward architecture inspired by the mammalian visual cortex. Modern topologies still follow the same idea: More complex features are build and recognized from simpler features with increasing depth of the network.

While neither data nor compute was available for large-scale training of CNNs when Neocognitron was released, Krizhevsky et al. [46] showed in the 2012 ILSVRC (ImageNet Large-Scale Visual Recognition Challenge) that CNNs can outperform traditional computer vision algorithms for classification. Since then, ILSVRC was dominated by CNN classifiers with a trend towards ever-deeper models with an increasing number of parameters [41].

With deeper networks, the vanishing gradient problem emerged, which made it harder to train networks as the gradient from later layers could not be properly propagated all the way to early layers.

Network architectures — such as used by the ResNet family of networks [32] — omitted that problem, by furnishing the feed-forward architectures with skip (or residual) connections. These connections skip a number of layers to accommodate the propagation of learning gradient to earlier layers. Skip connections are ubiquitous in modern CNNs such as ConvNext [79] and RegNet [64]. Figure 2.1 shows a small network with a single residual block.

The increasing size, as well as structures such as skip-connections in models present challenges for the design of efficient acceleration architectures for CNNs. The following sections will elaborate on that issue in the context of FPGA acceleration.

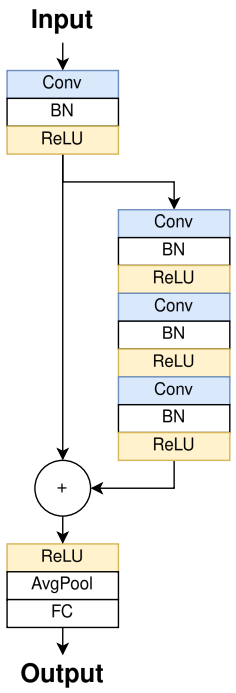


Figure 2.1: A small CNN with a single residual block. After an initial Conv-BN-ReLU stage, the data flow branches: the main branch passes through three Conv-BN-ReLU stages, while the skip connection carries the same activations unchanged to the element-wise addition (+). A final ReLU, average pooling and a fully connected (FC) layer produce the output.

2.3 Mapping CNNs to FPGAs

In the space of mapping CNNs to FPGAs, there are two central architectures: The "one-size-fits-all" unified, instruction-based accelerator architecture, and specialized streaming architectures. These two are examined in the following.

Streaming architectures are generally considered superior to instruction-based accelerators [57]. The key differences between the two approaches are summarised below before each is examined in detail.

Instruction-based accelerators are typically implemented as systolic array hardware designs. They use micro-instructions to execute the layers of a CNN one at a time, and are limited by inefficient processor-like control mechanisms. Examples include Snowflake [25], Angel-Eye [29], OPU [91], Brainwave [22] and VTA [58].

Streaming accelerators rely on deeply pipelined hardware designs in which every layer of the CNN is mapped to a dedicated computation engine. Toolflows such as FINN [81], FPGAConvNet [82] and HLS4ML automate the generation of these designs and often achieve high-performance results, because the design is tailored to the specific hardware platform and model.

2.3.1 Instruction-based Accelerators

Unified accelerator architectures such as those generated by the Snowflake [25] and Angel-Eye [29] toolflows are fixed, general-purpose compute engines. Architecturally, they are similar to GPUs or Tensor Processing Units (TPUs), combining scalar control logic with vector / tensor units for heavy math operations. To run a CNN on them, the model is first converted into a stream of instructions that is then fed to the accelerator. Inputs, outputs, and intermediate results are streamed to and from off-chip DRAM between instructions. Their main benefit is flexibility: No reconfiguration or generation of a new bitstream is required for running a different model on the FPGA. Only the instruction stream has to be regenerated for the new model, with no change to the underlying hardware. This makes the unified accelerator architecture particularly interesting in scenarios where multiple models are used in quick succession. Additionally, tensor and vector units can be scaled up or down depending on the available resources of the target FPGA, achieving high compute utilization. However, their flexibility comes at a cost: Requiring intermediate results to be written to and read from DRAM between instructions places significant pressure on the memory bus. The fixed datapath also forfeits per-layer optimization opportunities; irregularly shaped layers cannot efficiently be matched to the engine geometry.

2.3.2 Streaming Accelerators

Streaming dataflow (SDF) accelerators are created by customizing a layer-wise pipelined design for each specific workload; every layer gets its own dedicated compute module on the chip. Dataflow is confined to mostly on-chip; data flows from layer to layer and only the input and output stages need to go through the off-chip memory bus. Weights are also stored statically on-chip in Block RAM (BRAM) near the compute units, rather than having to be pulled in for the computation of each layer [57, 80, 83].

Since manual design of a custom pipeline for a particular network is expensive and would have to be redone for new networks, design of streaming architectures is done via automated toolflows [30, 81, 82], which — given a network and a platform — try to find the best mapping of that network to the particular FPGA. A lot of the work is focused on introducing novel design space dimensions or optimizing aspects of the system to allow the optimizer to explore novel configurations and trade-offs [57, 80, 92].

In the streaming compute model, the network is represented as a Directed Acyclic Graph (DAG) whose vertices are operations, such as convolutions and non-linear activation functions, and whose edges represent the data flow between them [80]. To smooth out data-rate mismatches between two connected

nodes, buffers are inserted between them. These buffers become especially important where the DAG branches: if the parallel branches differ in depth, the shorter branch must buffer its data to stay synchronized with the longer one and to avoid deadlocks. This can result in large on-chip memory demands for a branch in which no computation takes place at all. Such branches of unequal depth are referred to as long-skip connections [55, 80]. Figure 2.2 shows the network of Fig. 2.1 mapped in this way, with a branch synchronization buffer on its long-skip connection.

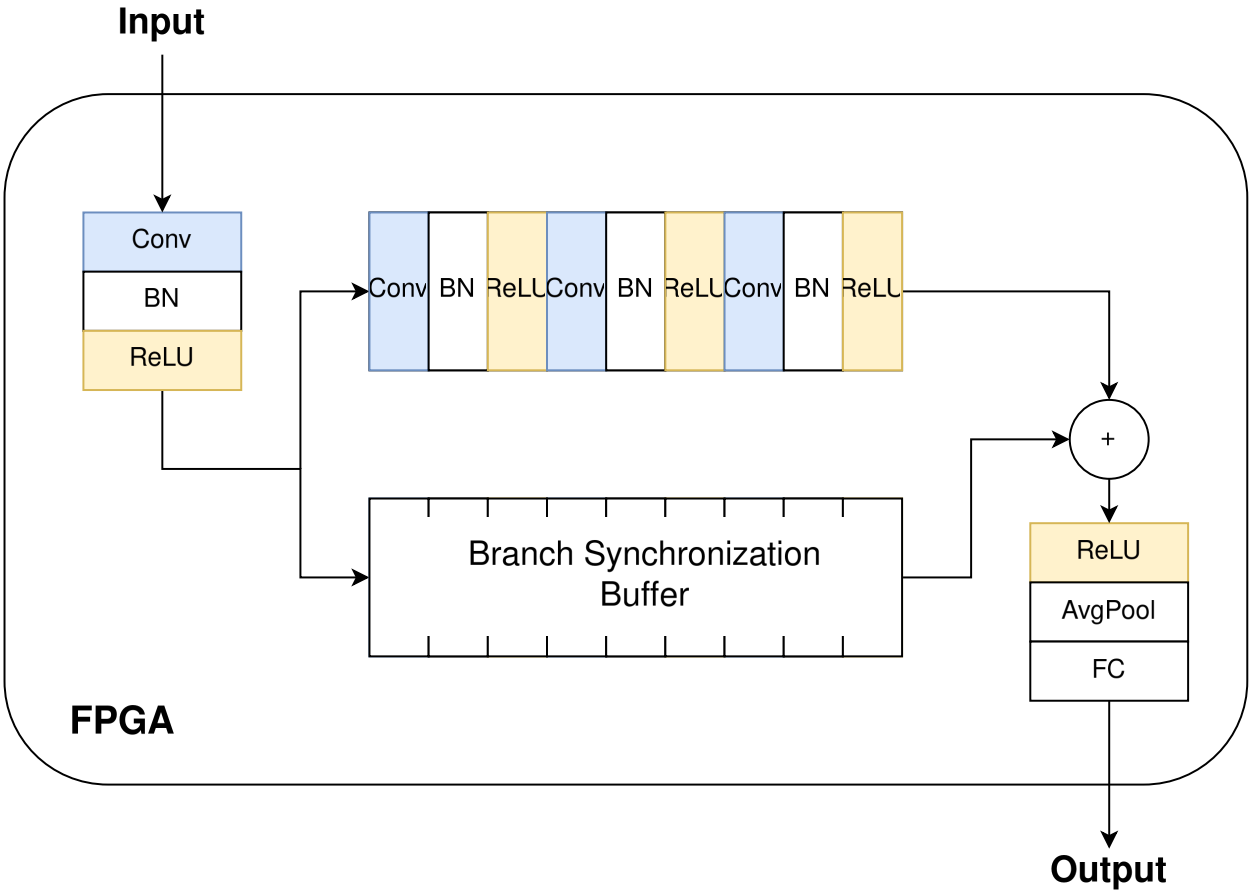


Figure 2.2: The network of Fig. 2.1 mapped to an FPGA by a streaming accelerator toolflow. Every layer (convolution, batch normalization (BN), ReLU, average pooling, fully connected (FC)) is implemented by its own compute module, and data flows from module to module on-chip; only the network input and output cross the FPGA boundary. After the first Conv-BN-ReLU stage the data flow branches: the main branch passes through three further Conv-BN-ReLU stages, while the long-skip connection holds the same activations in a branch synchronization buffer until the main branch's output reaches the element-wise addition.

Strengths Per-layer customization allows creating highly pipelined, high-throughput designs. Because every layer gets its own tailored compute module, the architecture is a much better fit for irregularly shaped layers than the generalized instruction-based approach, which may suffer poor utilization when layer dimensions do not match the engine geometry. This tailored mapping generally results in better resource utilization and improved energy efficiency [83].

Their flexibility also allows them to efficiently handle non-uniform network and layer structures, such as those produced by heavy pruning. Because each layer is assigned its own compute block, irregular weight distributions introduced by pruning can be projected efficiently onto the hardware without the overhead of a general-purpose datapath [57].

Compared to unified accelerators, the absence of an instruction stream and the mostly on-chip data movement significantly reduce pressure on the off-chip memory bus. Data flows through the computational graph from one engine to the next; the memory bus is primarily used for streaming new inputs and final outputs [80].

Weaknesses The primary weakness of SDF accelerators is their high demand for on-chip memory. Modern networks have huge number of parameters that are preferably pre-loaded in streaming architectures to maximize performance [92]. Additionally, to assure continuous data flow between pipeline stages, FIFO buffers are inserted between layers to absorb any rate mismatches. These buffers are particularly costly for long-skip connections, where an activation tensor produced early in the network must be buffered until a much later layer consumes it. For modern CNN topologies with deep and wide skip connections, the on-chip memory required for these FIFOs alone can exceed the available BRAM of the target device [55, 80].

A further weakness is inflexibility: switching to a different network requires a full re-run of the toolflow, including logic synthesis, placement, routing and bitstream generation. This makes SDF accelerators poorly suited to scenarios where multiple models must be served in quick succession [54].

Finally, the entire network — or at least a self-contained subgraph of it — must fit onto the FPGA simultaneously. This is especially problematic for modern CNNs, which combine high parameter counts with long skip connections, placing a large joint demand on both on-chip compute resources and on-chip memory [80].

2.4 Fitting modern CNN on Accelerators

In this section we address the question how modern CNNs can be accommodated on accelerators. This problem is approached from two sides: (1) reduction of network size and complexity; and (2) introducing new trade-offs into the accelerators design space to allow the toolflows optimizer to explore more design options.

In the following section, we will start with (1), a general discussion of model compression techniques that are applied to networks to reduce their resource demands before being mapped to an accelerator. These techniques are usual orthogonal to further compression happening on-chip during inference and thus produce such networks as we expect them to be passed to toolflows.

For the accelerator side (2), there are different ways of how big CNNs can be accommodated by restricted FPGA platforms. Some work distributes workloads across multiple devices [3, 38]; others time-multiplexed the device by reconfiguration [5, 80]. An additional way is to trade memory bandwidth with on-chip buffer space by evicting intermediate data[55, 80].

The improvement of the latter scenario is focus of this work and will be examined in more detail after a general discussion on structural model compression techniques.

2.4.1 Model Compression

Here we provide a short overview of the most important structural compression techniques for CNNs and neural networks in general.

Most approaches are lossy and are often combined with retraining and/or finetuning to recover some, if not all, of the accuracy lost during compression. Retraining may not always be possible as labeled data required for retraining is not always available.

2.4.1.1 Pruning

Pruning in neural networks has a long history, originating in works such as "Skeletonization" by Mozer and Smolensky [59] and "Optimal Brain Damage" by LeCun et al. [48]. According to a recent survey by Cheng et al. [8], more then half of the papers on neural network compression are on pruning and pruning-related approaches.

Pruning is, in a nutshell, the process of removing unimportant weights and structures from a neural network to reduce its size. Pruning does not only reduce the size of the stored model, but — by setting weights to zero — also reduces the computational cost and energy cost during runtime as fewer computations need to be performed [8].

Pruning can target every element of a neural net: weights, neurons, channels, filters and whole connections. It can be done in a *unstructured* way, selecting simply those weights, that fall below a certain threshold. This results in high sparsity, which is randomly distributed across weight matrices, resulting in irregular memory accesses [8].

structured pruning, on the other hand, usually works on bigger units, geometrically reshaping matrices or cutting out complete parts to produce a smaller denser network [8].

There is a plethora of further categorizations one could organize pruning approaches in. But if applied properly, pruning generally reduces both network size, as sparser networks can be more easily compressed by other lossless compression methods and encoding formats such as Compressed Sparse Row (CSR) [31]; and can also increase performance since zero multiplications can be skipped [8].

2.4.1.2 Quantization

Neural networks are most commonly trained on floating point formats such as IEEE-754 [39]. This is not only a quite wide format but also requires complicated computational unit to be used in hardware. To reduce both storage and computational requirements, networks are quantized to lower-bitwidth fixed-point formats [60, 85].

Full-size values are mapped to their corresponding reduced-precision representation $\hat{x} = s \cdot \text{clamp}(\text{round}(x/s) + z,\ 0,\ 2^b - 1) - sz$ with scale s, zero point z, bitwidth b [60]. As b shrinks, the storage requirements and computational costs shrink, but quantization noise increases, reducing accuracy [60].

This is a problem in PTQ (Post-training Quantization), where quantization is applied to weights after training has finished. Aggressive quantization, or even binarization, can result in unacceptable accuracy loss [85].

To reduce the accuracy loss during training, an approach called Quantization-aware Training (QAT) can be followed, which fake-quantizes weights during training while gradients are still calculated in full precision. This allows aggressive quantization with — in many cases — no loss of accuracy [60, 85].

Mixed precision approaches apply different quantization-levels to different parts of the networks, assigning more budget to more important structures of the networks [16].

In our work we generally assume a network to be quantized to 8 bits with per-channel asymmetric (affine) linear quantization: every channel has its own scale s and zero point z, profiled from its minimum and maximum.

2.4.1.3 Distillation

Distillation is the process of training a small "student" network to mimic a larger "teacher" network: The student learns from the teacher's softened output distribution in addition to the true labels of training data. This technique is often combined with other techniques, such as quantization and pruning: The uncompressed model can be used to recover the accuracy lost after these lossy techniques were applied [35].

Distillation is in comparison with the prior two techniques quite costly, as it requires to go through the whole training process for the student model.

2.4.1.4 Low-Rank Approximation

Low-rank approximation exploits redundancy within the weight tensors of a trained network: A layer's weight matrix is approximated by a product of two smaller matrices, e. g. obtained by a truncated SVD (section 2.5.2.3), which reduces both parameters and computations [11, 40]. Convolution kernels can be decomposed analogously using tensor decompositions [43, 47].

The rank trades size against accuracy, and decomposed networks are usually fine-tuned afterwards to recover accuracy [43, 47].

We also evaluate truncated SVD as an option for compressing activations.

2.4.1.5 Model Compression on FPGAs

The above mentioned techniques are also taken advantage of in hardware implementations; either directly, by applying the techniques to the networks as part of the hardware design itself or indirectly, by leveraging changed properties of weights and activations after compression.

Pruning can leave networks with sparse weight matrices, which can be compressed down further. HPIPE [30] uses a sparse encoding scheme that stores position of non-zero weights relative to each other in a pruned ResNet50 network with a sparsity of 85%. This allows storing all of the weights on chip, rather than streaming them in during runtime and additionally reduces compute requirements as multiplications involving zero-weights can be skipped.

Quantization is a particularly interesting method when paired with FPGA acceleration as the customizability of an FPGA allows creating compute units custom-fit for specific precisions. Mixed-precision quantization methods, such as HAWQ [16], allow applying different quantization level at parts of different granularity of the network. While HAWQ only explored assigning tailored precisions per layer, newer approaches, like the work by Sun et al. [73], explores even per-weight custom quantization levels.

With Mixed-TD, Yu and Bouganis [92] propose a tensor decomposition approach which decides — per convolutional layer — whether the layer should be decomposed using Singular Value Decomposition (SVD) or Canonical Polyadic Decomposition (CPD). Whichever is better for accuracy is chosen, as they note that the error tolerance and suitability of the two decomposition approaches varies for different parts of the network. They use additional fine-tuning to reduce accuracy impact of the decomposition. The result is a reduced model size which allows loading models into their streaming accelerator which previously could not fit on accelerators with limited buffer space.

PASS [57] leverages sparsity present in activations to increase performance. Activations sparsity is only seen at runtime, thus they propose a dynamic zero-skipping mechanism and compile-time workload balancing design which is determined by a sparsity-aware DSE (Design Space Exploration). They avoid formats such as CSR (Compressed Sparse Row) for activations as they pay off only for extremely sparse data. They report a almost two-fold increase in performance density over instruction-based sparse accelerators.

2.4.2 Scaling Accelerators: Off-chip Streaming

Off-chip streaming is the category this thesis builds on, so we discuss it in the most detail. It moves the bottleneck from on-chip memory to the memory bus, which raises the question of how the data sent across the bus can be reduced. The compression techniques that can answer this question are introduced in section 2.5, and the works applying them are discussed in section 2.6. Apart from streaming network inputs and outputs, streaming accelerators leave the off-chip memory largely unused, in contrast to instruction-based accelerators, which depend heavily on it. SMOF [80] puts this spare bandwidth to use by moving data that would otherwise occupy on-chip memory to DRAM, explicitly trading on-chip memory for off-chip bandwidth. It introduces two complementary techniques.

Weight fragmentation splits the weights of a layer into a part that stays on-chip and a part that is streamed in from DRAM during computation. Since the weight access pattern is fully known at compile time, these transfers can be scheduled statically. Weights can also be compressed offline; because their compressed size is known in advance, the resulting bandwidth demand stays predictable.

Activation eviction targets the FIFO buffers of long-skip connections. Instead of holding the skipped activation tensor on-chip until the consuming layer is reached, it is written to off-chip memory and read back once needed, freeing on-chip memory for the compute modules. To reduce the bandwidth cost of eviction, SMOF encodes evicted activations with lightweight lossless schemes. Unlike for weights, however, the compressed size of activations depends on the input and is unknown at compile time, which complicates static bandwidth analysis. Figure 2.3 shows the network of Fig. 2.2 with its long-skip connection evicted and compressed: the branch synchronization buffer is no longer needed, and the on-chip memory it occupied is freed.

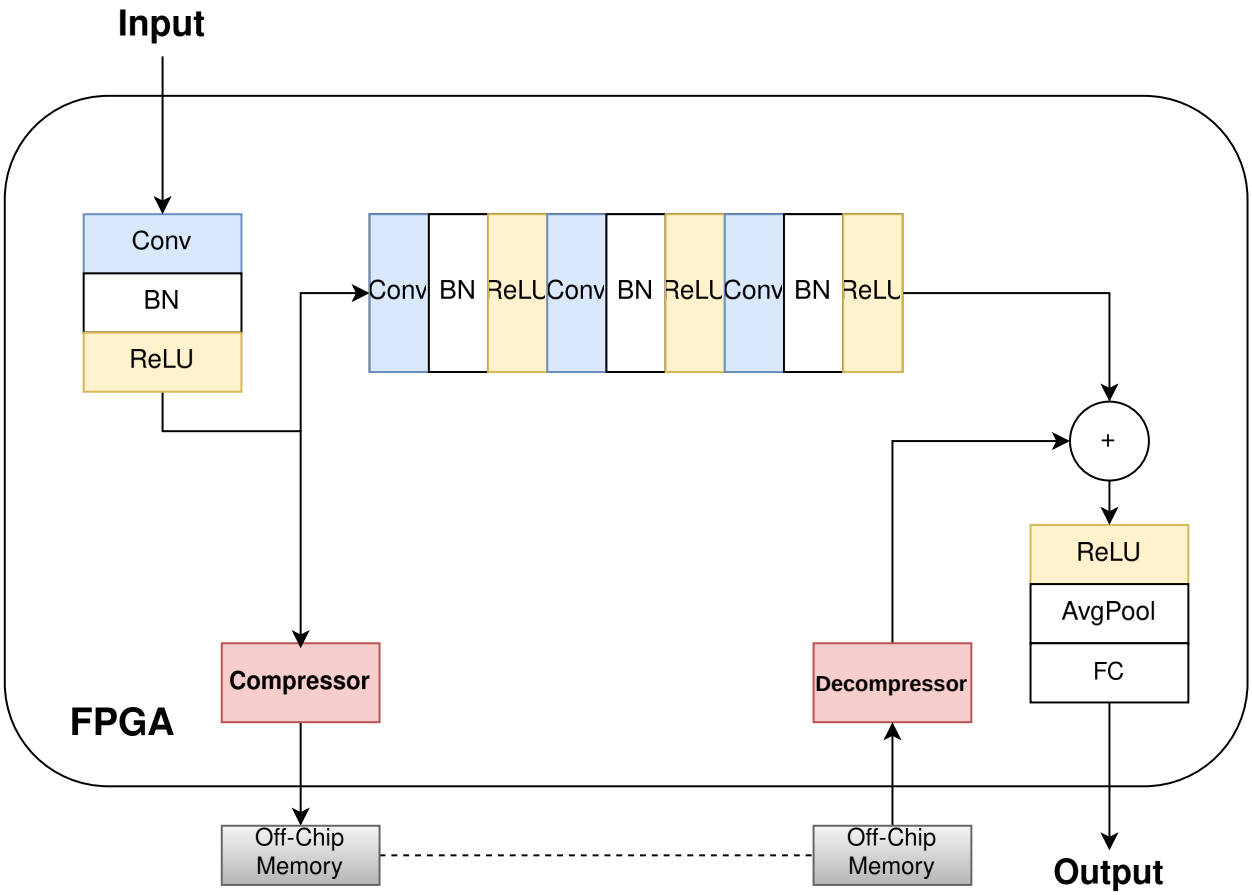


Figure 2.3: The network of Fig. 2.2 with its long-skip connection evicted. The branch synchronization buffer, and the on-chip memory it occupied, is replaced by an on-chip compressor, which writes the activations to off-chip memory, and an on-chip decompressor, which reads them back and passes the reconstructed activations to the element-wise addition. Both off-chip memory boxes denote the same memory (dashed line); only the compressed representation crosses the memory bus.

Both techniques shift pressure from on-chip memory to the memory bus, which is shared by all fragmented layers, all evicted connections, and the input and output streams. Once the bus saturates, compute modules stall while waiting for weights or activations, and the pipeline loses its throughput advantage. Long-skip activations are particularly demanding. The available bandwidth therefore limits how much can be evicted, and any reduction in the volume of evicted data translates directly to more freed on-chip resources, which can in turn be used by the optimizer to improve other parts of the design [55].

To further increase the efficacy of eviction, SMOF uses lossless Run Length Encoding (RLE) and Huffman encoding. They show that this can significantly improve system performance: A decrease in memory bandwidth requirements by means of compressing activations is directly tied to overall performance improvements. Higher compression ratios proof to be a good proxy for increased performance.

Lossless encoding, however, only achieves limited compression ratio on activations, as compression ratio is fundamentally limited by the entropy of the compressed data [69]. Since lossless encoding achieves only limited compression ratios on activations, this thesis investigates trading inference accuracy for higher compression ratios, using the lossy techniques introduced in Section 2.5.

2.5 Compression Fundamentals

The setting established in section 2.4.2 narrows down which compression techniques are worth considering. The requirements stated in section 1.2 serve as selection criteria: they steer our selection of techniques in this section, of related work in section 2.6 and of the baselines we compare against in chapter 4. We briefly restate them here; section 3.1.1 details the assumptions behind them.

- **No retraining:** the network is given and already compressed; neither weights nor structure are changed.
- **Controllable error:** the error introduced by lossy compression has to be configurable, so that it can be traded against inference accuracy.
- **Streaming operation:** values arrive one per cycle in a fixed order and can only be buffered in small amounts.
- **Low hardware cost:** a compressor is instantiated for every evicted connection, so its cost per channel has to be small.

We first introduce the individual building blocks, starting with lossless techniques such as entropy coding and run-length encoding, followed by lossy techniques: requantization, decimation, bit manipulation, transformation, low-rank approximation and prediction. The prediction-based model of SZ [13] is the starting point of our own design.

Afterwards, we describe some hand-picked compression pipelines, that will also serve as baselines for our evaluation.

2.5.1 Lossless Compression Techniques

Di et al. [14] organize lossless encoding techniques into the following categories:

- **Entropy Encoding:** assigns variable-length codewords based on symbol probabilities, so that frequent symbols get shorter codewords.
- **Reduce Repeated Symbols:** replaces repeated occurrences of a symbol, such as runs of identical values, by a more compact description of the repetition.
- **Predictive Delta Encoding:** stores the difference between a value and a prediction from preceding values, which concentrates values around zero.
- **Transform Encoding:** applies a reversible transformation that rearranges the data into a more compressible form.
- **Dictionary Encoding:** replaces recurring sequences by references to earlier occurrences or to entries of a dictionary.

Ko et al. [44] has described dictionary-based encoding as being too expensive for effective hardware implementations and since our focus is on light-weight compression, we decided to not consider it in this thesis.

In the following, we discuss the concrete lossless techniques that are relevant for the remainder of this thesis, and state the category each of them belongs to.

2.5.1.1 Run-Length Encoding (RLE)

Run-Length Encoding (RLE) belongs to the category of reducing repeated symbols. Instead of storing successive identical symbols individually, it stores a record consisting of the symbol and a counter holding the length of the run. With d-bit symbols and a c-bit counter, every record occupies $d + c$ bits and can describe a run of up to 2^c symbols. A run of length L thus requires $\lceil L/2^c \rceil$ records, and its compression ratio is

$$\mathrm{CR} = \frac{L \cdot d}{\lceil L/2^c \rceil \, (d + c)}.$$

RLE only compresses data that actually contains runs: an isolated symbol ($L = 1$) costs $d + c$ instead of d bits, so data without long runs is expanded rather than compressed.

Zero Run-Length Encoding (ZRLE) is a variant tailored to sparse data, in which most values are zero and non-zero values rarely occur. Only runs of zeros are run-length encoded: each record consists of the number of zeros preceding a non-zero value and that value itself. A zero run and the non-zero value ending it thus share one record of $d + c$ bits, where plain RLE would need two records, and the zero symbol itself never has to be stored. This makes ZRLE effective for activations after ReLU functions, which contain many zeros [95]. Appendix A.11 compares both variants on the skip connections compressed in this thesis.

The counter width c is the central parameter and has to be matched to the data. A counter that is too narrow forces the encoder to split long runs into multiple records, while a counter that is too wide adds unused bits to every single record. If sample data is available, c can be chosen based on the distribution of run lengths measured on it. Since RLE only exploits runs along the order in which the data is traversed, the traversal order matters as well; JPEG, for instance, reads the quantized coefficients in a zigzag order so that the zero coefficients form long runs (section 2.5.3.2).

In hardware, RLE requires only a comparator, a counter and a register for the current symbol, and processes one symbol per cycle without any buffering or tables. SMOF [80] uses it to encode evicted activations, which makes it one of the lossless baselines of this thesis.

2.5.1.2 PNG Filters

The filters of the PNG image format [84] are a predictive delta encoding technique. Before compression, each byte x of an image row is replaced by the difference to a prediction $\hat{x}$ computed from neighbouring, already processed bytes,

$$r = (x - \hat{x}) \bmod 256.$$

Neighbouring pixels of natural images are strongly correlated, so most residuals r are close to zero and their distribution is much more skewed than that of the raw values, which makes them easier to compress for the subsequent encoder. Since the prediction only uses values that the decoder has already reconstructed, the filtering is exactly reversible.

With a denoting the byte to the left, b the byte above and c the byte to the upper left of x, PNG defines five filter types [84]:

- **None:** $\hat{x} = 0$, the row is stored unchanged.

- **Sub:** $\hat{x} = a$, the difference to the left neighbour.
- **Up:** $\hat{x} = b$, the difference to the neighbour above.
- **Average:** $\hat{x} = \lfloor (a + b)/2 \rfloor$.
- **Paeth:** computes the linear estimate $p = a + b - c$ and predicts with whichever of a, b and c is closest to p.

The filter type is chosen per row and stored as one additional byte in front of the row, so the encoder can pick the best-fitting predictor for each part of the image. The filtered rows are then compressed with DEFLATE, i. e. LZ77 followed by Huffman coding.

We introduce the PNG Average and Paeth filter as additional prediction functions for our prediction-based compressor, that we develop in chapter 3.

2.5.1.3 Discrete Cosine Transform (DCT)

The Discrete Cosine Transform (DCT) [2] is a transform encoding technique. It represents a block of N values $x_0, \ldots, x_{N-1}$ as a weighted sum of cosine functions of increasing frequency,

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n + \frac{1}{2}\right)k\right], \qquad k = 0, \ldots, N-1.$$

For correlated data, most of the energy is concentrated in the few low-frequency coefficients, while the remaining coefficients are close to zero and can be coded compactly. The transform itself is invertible; compression only becomes lossy when the coefficients are quantized, as in JPEG (see section 2.5.3.2). Images are transformed in two dimensions, applying the DCT to the rows and columns of a block, e. g. of 8×8 pixels.

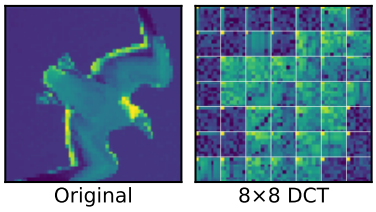


Figure 2.4: Feature map (left) and its 8x8 block DCT (right), each block's coefficient magnitudes drawn in place of its pixels on a logarithmic colour scale

2.5.1.4 Huffman Coding

Huffman coding is an entropy encoding technique. Entropy coding compresses data by assigning shorter codewords to symbols that occur more frequently and longer codewords to symbols that occur less frequently. Huffman [37] proposed a simple method for constructing an optimal prefix-free code from the probabilities p_i of the source symbols. The compressibility achievable this way is bounded by the entropy of the source distribution [69]: no memoryless code can reduce the average codeword length ℓ below the entropy $H(X)$, and a Huffman code comes within one bit of it,

$$H(X) \le \ell < H(X) + 1, \qquad H(X) = -\sum_i p_i \log_2 p_i.$$

The gap stems from the integer codeword lengths: up to one bit per symbol can be lost, which is severe for small alphabets or highly skewed distributions. Since every codeword is at least one bit long, the compression ratio for d-bit symbols can never exceed d.

Both limitations can be relaxed with **block codes**, which encode n successive symbols as a single symbol of the extended alphabet $\mathcal{S}^n$:

$$\frac{H(X_1, \ldots, X_n)}{n} \le \frac{\ell_n}{n} < \frac{H(X_1, \ldots, X_n)}{n} + \frac{1}{n}.$$

This yields two gains. The rounding loss per symbol shrinks to $1/n$, and, if the block histogram is counted on the data, correlation between neighbouring symbols is exploited, since $H(X_1, \ldots, X_n) \leq n\,H(X)$. The cost is a code book of size $|\mathcal{S}|^n$, which grows exponentially with n: for 8-bit values and $n = 2$, it already holds $256^2 = 65\,536$ entries, whereas a 4-symbol alphabet needs only $4^2 = 16$. Larger code books require more table storage and more calibration data to estimate the block probabilities. Block codes can therefore pay off for small alphabets, such as predictor codes, but not for raw 8-bit values.

The benefits of using block codes by combining n symbols together is, however, still fundamentally bounded by the probability distribution of symbols. A source alphabet with approximately equal distributions of its symbols and without meaningful correlations between symbols can not gain much from block coding, no matter how big the blocks. The fundamental limit to compressibility remains the entropy.

2.5.2 Lossy Compression Techniques

To focus our selection of compression techniques to use and compare against, we took guidance from a survey by Di et al. [14], which not only gives an overview of state-of-the-art (SOTA) compression techniques for scientific data sets, but also provides a novel taxonomy and unified pipeline model for compression methods. We consider the following categories of lossy compression techniques:

- **Quantization:** maps values to a coarser set of representable values, e. g. fewer bits per value.
- **Decimation-/filtering-based Compression:** reduces the number of values by sampling or filtering the data in time or space; the omitted values are reconstructed, e. g. by interpolation, on decompression.
- **Bit-manipulation-based Compression:** exploits properties of the binary representation of the data, e. g. by truncating insignificant bits, and restricts itself to lightweight operations such as additions, subtractions and bitwise operations.
- **Transformation-based Compression:** transforms the data into a coefficient domain in which the energy is concentrated in few coefficients, and quantizes the coefficients.
- **Dimension Reduction:** approximates the data by a representation of lower dimension or rank, e. g. by projecting it onto a few basis vectors.
- **Prediction-based Compression:** predicts each value from previously reconstructed values and encodes only how the actual value deviates from the prediction.

In the following, we discuss the concrete lossy techniques relevant for this thesis, and state the category each of them belongs to.

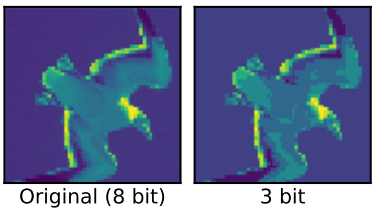


Figure 2.5: Feature map (left) and its reconstruction after uniform quantization to 3 bits (right)

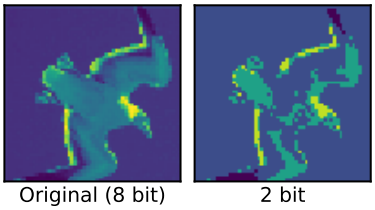


Figure 2.6: Feature map (left) and its reconstruction after uniform quantization to 2 bits (right)

2.5.2.1 Quantization

Quantization, already introduced as a model compression technique in section 2.4.1, maps a range of values to a smaller set of representable values, and thus to a smaller bitwidth. Applied to activations that are already quantized, e. g. to 8 bits, it amounts to requantizing them to fewer bits (Figs. 2.5 and 2.6).

For a uniform quantizer with step size Δ, every value is rounded to the nearest level, so the error is at most $\Delta/2$; assuming the error is uniformly distributed, the mean squared error is $\Delta^2/12$ [28]. Removing one bit doubles Δ and thus quadruples the mean squared error.

Besides uniform (linear-scale) quantization, non-uniform schemes such as log-scale or vector quantization place more levels in dense regions of the value distribution [14].

Quantization is commonly combined with other techniques: Deep Compression [31] Huffman-codes the quantized values (section 2.5.3.1), and mixed-precision approaches such as HAWQ [16] assign each layer its own bitwidth according to its sensitivity.

2.5.2.2 Downsampling

Downsampling is a decimation-based compression technique. Di et al. [14] describe decimation techniques to be able to achieve high data compression performance while posing potentially high computational costs for the data reconstruction step, as numeric methods may be necessary for reconstruction. We investigate a simple downsampling approach as both a standalone compression technique and as a preprocessing step for advanced compression pipelines.

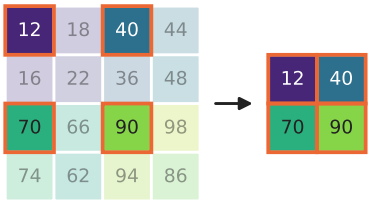


Figure 2.7: Strided-Subsampling Downsampling: Every f_{down}^2-th value of the featuremap is retained

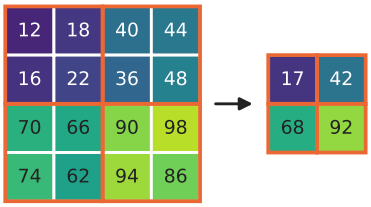


Figure 2.8: Average-Pooling Downsampling: At downsampling facto $f_{down} = 2$ f_{down}^2 values are averaged together to produce a value in the output map

Downsampling is a simple reduction of the spatial dimensions of a — in our case — two-dimensional feature map. We evaluate a simple **strided subsampling** approach, where only every f_{down}-th value along each spatial dimension of the feature map is retained, resulting in $\frac{1}{f_{down}^2}$ of the original values being kept. For a subsampling factor of $f_{down} = 2$, for example, one in four values is retained by keeping only every second value in both the horizontal and vertical dimensions. For a feature map $X \in \mathbb{R}^{H \times W}$ and a factor f_{down} that divides H and W, the subsampled map $Y \in \mathbb{R}^{\frac{H}{f_{down}} \times \frac{W}{f_{down}}}$ is

$$Y_{i,j} := X_{f_{down}i, f_{down}j}, \qquad 0 \leq i < \frac{H}{f_{down}},\ 0 \leq j < \frac{W}{f_{down}}.$$

A downsampling method aimed at additionally preserving information of neighboring pixels is **average pooling**, which is equivalent to processing the feature map with a convolution first, where the filter assigns equal weight to all input pixels. The spatial reduction is dependent on the filter. E. g. for a 2x2 filter with a stride of 2, the size of the feature map would be reduced 4×. Which pixels of the input contribute to the values of the decimated output depend on the filter size. For average pooling,filters are chosen such that all pixels of the input feature map have equal impact on the pixels of the output map. With a filter of size $f_{down} \times f_{down}$ and stride f_{down}, every non-overlapping $f_{down} \times f_{down}$ window is replaced by its mean, rounded back onto the integer grid of the quantized feature map:

$$Y_{i,j} := \text{round}\left(\frac{1}{f_{down}^2} \sum_{m=0}^{f_{down}-1} \sum_{n=0}^{f_{down}-1} X_{f_{down}i+m, f_{down}j+n}\right).$$

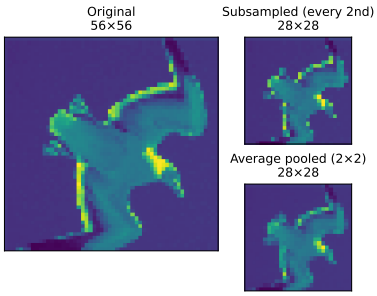


Figure 2.9: Downsampling using strided subsampling and average pooling

Reconstruction The decimated feature maps have to be restored to their original sizes in order to be used in their original scenario again. Here we also choose two deliberately cheap options, **nearest neighbor upsampling** and **bilinear interpolation**.

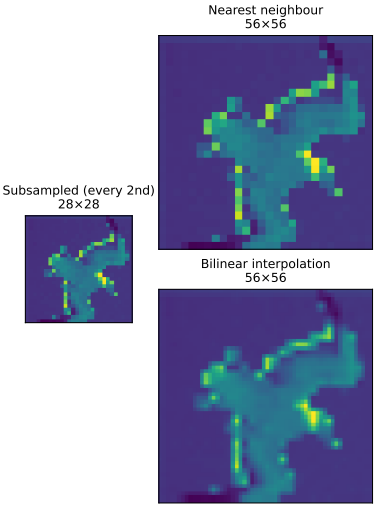


Figure 2.10: Upsampling a factor-2 subsampled feature map back to full size using nearest neighbor upsampling and bilinear interpolation

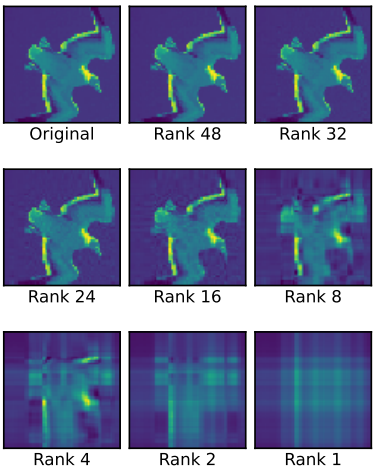


Figure 2.11: Feature map (top left) and its truncated SVD reconstructions at ranks 48, 32, 24, 16, 8, 4, 2 and 1

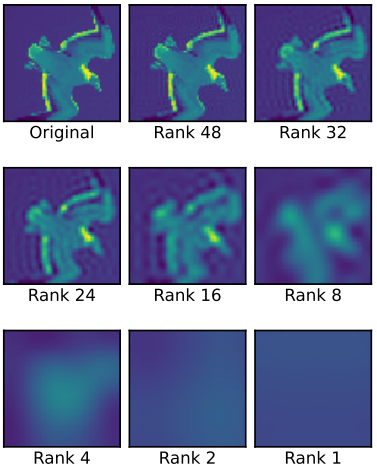


Figure 2.12: Truncated SVD with shared bases calculated from the ImageNet 5000 sample validation dataset; the fidelity of the reconstruction is — in comparison to Fig. 2.11 — noticeably reduced

Nearest neighbor simply takes the closest neighbor for the new pixels while bilinear interpolation uses a linear fit between neighboring pixels in the reduced map to calculated new values. Both of these approaches are computationally cheap, but are costly in terms of error. Nearest neighbor upsampling copies every stored value into the $f_{down} \times f_{down}$ window it represents:

$$\hat{X}_{i,j} := Y_{\lfloor i/f_{down} \rfloor, \lfloor j/f_{down} \rfloor}.$$

Bilinear interpolation, the two-dimensional extension of linear interpolation, maps each output pixel back to a fractional position in the reduced map, treating values as pixel centres and clamping at the border,

$$u := \mathrm{clamp}\Big(\tfrac{i+1/2}{f_{down}} - \tfrac{1}{2},\ 0,\ \tfrac{H}{f_{down}} - 1\Big), \qquad v := \mathrm{clamp}\Big(\tfrac{j+1/2}{f_{down}} - \tfrac{1}{2},\ 0,\ \tfrac{W}{f_{down}} - 1\Big),$$

and weights the four surrounding stored values by their distance. With $i_0 := \lfloor u \rfloor$, $i_1 := \min(i_0 + 1, \tfrac{H}{f_{down}} - 1)$, $\alpha := u - i_0$, and j_0, j_1, β defined analogously from v:

$$\hat{X}_{i,j} := (1-\alpha)(1-\beta)\, Y_{i_0,j_0} + (1-\alpha)\,\beta\, Y_{i_0,j_1} + \alpha\,(1-\beta)\, Y_{i_1,j_0} + \alpha\,\beta\, Y_{i_1,j_1}.$$

2.5.2.3 Singular Value Decomposition (SVD)

Low-rank approximation by the SVD is a dimension reduction technique. By interpreting a matrix of data as a transformation matrix, SVD guarantees that a factorization of the matrix into a rotation, a scaling, and another rotation operation is possible.

Definition Every real matrix $A \in \mathbb{R}^{m \times n}$ can be factorised as $A = U\Sigma V^\top$, where $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are orthogonal and $\Sigma \in \mathbb{R}^{m \times n}$ is diagonal [27]. The diagonal entries $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$ of Σ are the **singular values**, and the columns u_i of U and v_i of V are the corresponding **left** and **right singular vectors**. Equivalently, A is a sum of rank-one matrices, ordered by their singular values:

$$A = \sum_{i=1}^{r} \sigma_i\, u_i v_i^\top,$$

where r, the number of non-zero singular values, is the rank of A. Each term adds one "layer" of structure to the matrix, and the singular value states how much that layer contributes. Computing the SVD costs $O(mn\,\min(m,n))$ operations, which is affordable offline but not per input in hardware.

Truncated SVD In practice, the singular values of real-world data often decay quickly: a few terms capture the coarse structure of the matrix, while the remaining terms only add fine detail and noise. The truncated SVD exploits this by keeping only the k largest terms,

$$A_k = \sum_{i=1}^{k} \sigma_i\, u_i v_i^\top = U_k \Sigma_k V_k^\top.$$

By the Eckart–Young–Mirsky theorem, A_k is the best possible approximation of A with rank k, and its error is determined entirely by the discarded singular values [17]. The rank k is thus a single knob trading accuracy for size: instead of mn values, only the $k(m+n+1)$ values of the factors have to be stored, which pays off whenever k is small compared to m and n. Figure 2.11 shows a feature map reconstructed from decreasing numbers of terms: the coarse structure survives down to low ranks, while fine detail is lost first.

Shared bases The factors U_k and V_k of a truncated SVD are specific to one matrix and have to be stored alongside it, and computing them requires an SVD of every matrix. When many matrices of the same shape and with similar content have to be compressed, such as the feature maps of one channel across different inputs, a single pair of bases can instead be learned offline from sample data and shared by all of them. Each matrix A_i is then compressed by projecting it onto these bases, which leaves only a small $k \times k$ core matrix to be stored, and reconstructed by projecting back:

$$C_i = U_k^\top A_i V_k, \qquad \hat{A}_i = U_k C_i V_k^\top.$$

Compression and decompression thus reduce to multiplications with constant matrices, and the compression ratio rises to $\frac{mn}{k^2}$, since the bases are not stored per matrix. The bases can be obtained from the SVD of the stacked sample matrices (2D-SVD [15]) or refined iteratively (GLRAM [89]). The price is that they are only good for data resembling the samples they were learned on; content outside the learned subspace is lost as reconstruction error.

Low-rank methods in neural network compression Most applications of low-rank approximation in neural networks target the **weights**: fully-connected and convolutional weight tensors are factorised into two or more thinner layers, reducing both the number of parameters and the number of multiply-accumulate operations (section 2.4.1). Yu and Bouganis [92], for instance, combine SVD with the Canonical Polyadic Decomposition and choose a decomposition per layer for streaming FPGA accelerators. Applying the same idea to **activations** differs in one important way: weights are fixed and can be decomposed offline, whereas activations change with every input. Decomposing each activation at runtime would require computing an SVD per feature map in hardware. Shared bases learned offline avoid this and reduce the runtime work to projections with constant matrices. We apply this idea to activations and evaluate it against other baselines in chapter 4.

2.5.2.4 Curve-fitting Prediction

Curve-fitting prediction, as introduced by SZ1 [13], is a prediction-based compression technique. The data is traversed in a fixed order, and each value x_i is predicted by extrapolating a curve through its immediate predecessors. SZ1 uses three such curves:

$$\begin{aligned} \text{Preceding Neighbor Fitting (PNF):} \quad & \hat{x}_i = x'_{i-1}, \\ \text{Linear Curve Fitting (LCF):} \quad & \hat{x}_i = 2x'_{i-1} - x'_{i-2}, \\ \text{Quadratic Curve Fitting (QCF):} \quad & \hat{x}_i = 3x'_{i-1} - 3x'_{i-2} + x'_{i-3}. \end{aligned}$$

For every value, all three predictions are evaluated and the best-fitting one is selected. If its error lies within the user-specified error bound, only a short code identifying the chosen function is stored instead of the value; otherwise, the value is marked as *unpredictable* and stored separately as an outlier.

The predictions are computed from the reconstructed values x' rather than the original values x. Di et al. [14] call this the *reconstructed-data-driven policy*: the decompressor only has access to reconstructed values, so the compressor has to predict from the same values to keep both sides in sync and to prevent errors from accumulating beyond the error bound. As a consequence, every prediction depends on the reconstruction of the previous values.

Since a prediction only needs the last three reconstructed values, curve-fitting prediction requires almost no buffering and maps well to a stream of values arriving one at a time. The price is that it only exploits correlation along the traversal order: when multidimensional data is linearized row by row, spatial locality across rows is lost, and predictions break down at row changes [13]. How the code and outlier streams are encoded is part of the SZ1 pipeline (section 2.5.3.3).

The fact that this compression scheme is lightweight, error-bounded and fits the streaming paradigm pushed us to focus on a development of this compression model in this thesis.

2.5.3 Compression Pipelines

A compression technique is rarely used on its own; usually, more capable compression pipelines are built by combining multiple compression modules together. Di et al. [14] suggests viewing a compression pipeline as consisting of three parts: Preprocessing, Decorrelation and Encoding.

There is many different concrete compression methods in each of those categories, but those can not naively be combined to create a strong compression pipeline. The data properties of the input, as well as after each of the stages is decisive for the performance of the compression methods used further down the line.

In the following, we present compression pipelines that we use as comparison baselines for the evaluation and also shed some light on the rationale behind certain combinations of compression modules.

2.5.3.1 Quantization + Huffman

The simplest pipeline combines quantization (section 2.5.2.1) with Huffman coding (section 2.5.1.4) and skips decorrelation entirely. Quantization maps each value to one of few levels, and the error is bounded by the step size. The resulting indices are usually far from uniformly distributed, which Huffman coding exploits by assigning short codes to frequent levels.

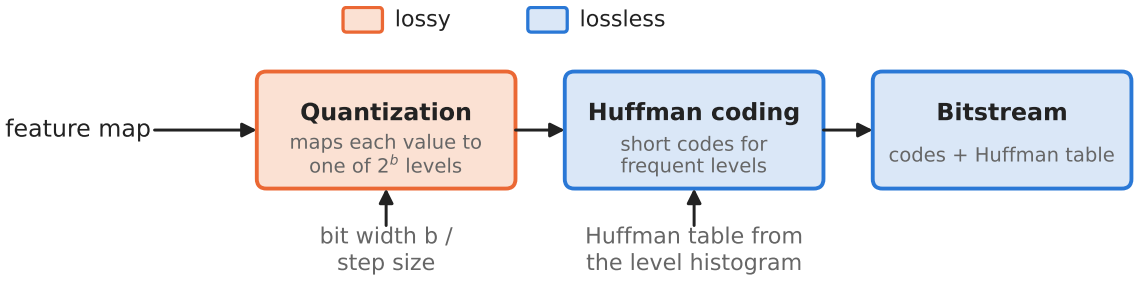


Figure 2.13: A simple compression pipeline consisting of Quantization and Huffman coding

Han et al. [31] use this pipeline in Deep Compression for network weights: after pruning, weights are clustered into a small codebook of shared values, and the codebook indices are Huffman-coded, which saves a further 20–30 % of storage. Together, the three stages compress AlexNet and VGG-16 by 35× and 49× without loss of accuracy. Since no decorrelation takes place, the compression ratio depends only on the value distribution, not on the correlation between neighbouring values.

2.5.3.2 JPEG

One of the most-widely used digital image formats, JPEG, uses a discrete cosine transform (DCT) on 8x8 pixel matrices and uses quantization and run-length encoding (RLE) in the compression pipeline. These approaches can generally achieve a high rate-distortion, but do generally not allow for a fine-grained selection of the error bound. The error can however be controlled to some degree by the size of divisors in the quantization table — the lossy step in the pipeline.

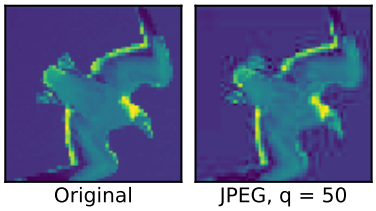


Figure 2.14: Feature map (left) and its reconstruction after JPEG compression at quality 50 (right)

JPEG is very effective for natural images where low-frequency information usually dominates. The transformation into the frequency domain encodes the block in terms of overlapping frequencies. The transformation results in 64 factors for waves of different frequencies. Low-frequency information is stored in the lower indexes, as Fig. 2.16 shows. The lossy step in JPEG is the division of the factors using a quantization table. This will result in many zeros in the high frequency domain — information that is usually not as important for the human periphery. The quantized coefficients are then traversed by the so-called Zigzag traversal scheme, which leads to long rows of zeros that can then be compressed with run-length coding. The final stream will then be encoded with a typically precomputed Huffman table to create the final bitstream.

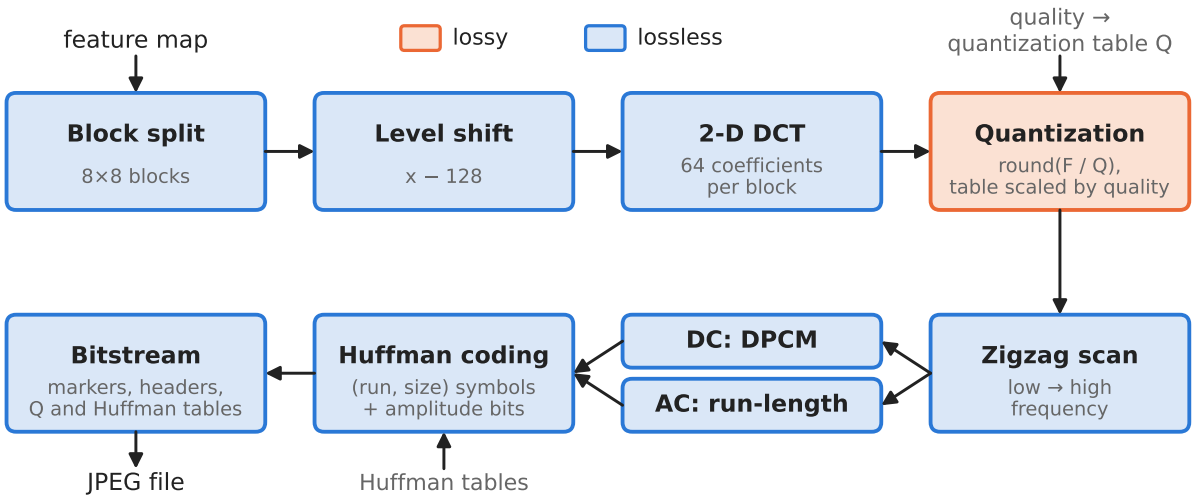


Figure 2.15: JPEG compression pipeline: Multiple lossless transformations and a lossy quantization step result in a very effective compression scheme for images

JPEG poses a number of problems for streaming accelerators: Preparing for the 2-D DCT requires buffering 8x8 blocks of data, and since data comes in row-major, we have to buffer up to $Buf_{JPEG} = 8w$ values, where w is the width of the feature map that is being compressed; the Zigzag scan traverses the DCT coefficients from low to high frequencies, rather than in row major order, requiring additional buffering. This traversal scheme is, however, essential to create a long stream of consecutive zero-values, that can be efficiently compressed with the run-length encoding step.

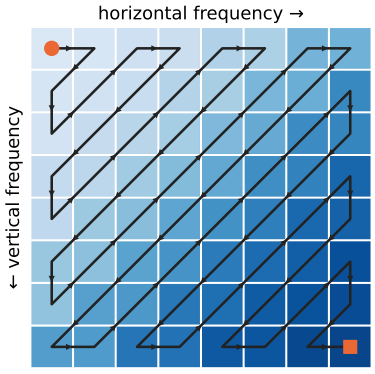


Figure 2.16: Zigzag-traversal of a 8x8 DCT factor matrix

2.5.3.3 SZ1

SZ1 [13] combines the curve-fitting prediction of section 2.5.2.4 with an error-bounded representation of unpredictable values and a final dictionary coding

stage. The output of the compressor logically consists of two streams: The stream of codes and the stream of outliers.

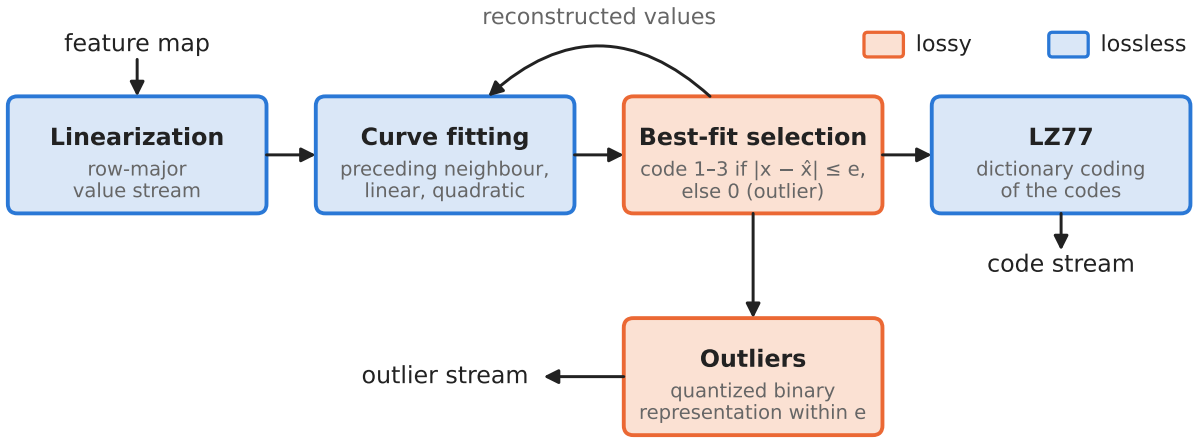


Figure 2.17: SZ1 uses best-fit curve-fitting and decorrelates the original stream from the output stream by encoding the choice of compression function. Di and Cappello [13] showed that this decorrelated stream of codes can be efficiently encoded using a dictionary based compressor such as LZ77

The stream of codes identifies which prediction function was selected at a particular position of the stream. This may be a 0 for an outlier or 1-3 for one of the actual prediction functions. The stream of outliers contains an outlier value for each of occurrence of the outlier-code in the code stream. Typically, the first few values are outliers, as the prediction-chain needs some values to get started. The outliers are not stored verbatim, however, but are quantized to a lower bit-width as much as the user-defined error bound allows it.

Outlier and code stream are enough for the decompressor to restore an approximated version of the original stream. The effectiveness of the compression is — apart from the user-specified error-bound — primarily dependent on how well the prediction functions can represent the input stream, i. d. how many outliers will end up in the outlier stream.

Assuming the input stream values are represented by 32 bit floats, a data set that can be perfectly predicted by the compressor approaches a compression ratio of 16:1; only the code stream — with 2 bits per data point — and the outlier stream — containing at least one outlier for seeding — need to be stored. The compression ratio is fundamentally limited by both the bit width of input values and the width of the prediction function codes.

In the worst case, no value can be effectively compressed using the prediction functions and the outliers stream contains as many values as the original stream in addition to the code stream. However, since the outliers are still quantized according to the user-defined bound, positive compression ratios are still possible, as long as they reduce each outlier value by at least two bit as compared to the original value.

Encoding the original stream using prediction functions completely decouples the distribution of values from the encoded stream. Instead of encoding raw values, or a transformation of those, the resulting stream encodes values in terms

of "which prediction works best". This resulting code stream contains its own properties, which may benefit from further lossless compression techniques. Di and Cappello [13] use the LZ77 dictionary encoding scheme to further compress the code stream. They base this on the observation that the resulting code stream has many repeating patterns that can be leveraged by such a dictionary encoder.

This enables the compression scheme to push far beyond the constrained compression ratios of the prediction module alone.

While we evaluate the SZ1 compression scheme as a baseline, we do so without the dictionary coding. We base this decision on Ko et al. [44], who stated that dictionary encoding is usually too expensive on FPGAs. Instead we pair the compressor with a Huffman stage, to make the comparison with other pipelines fair.

2.5.3.4 SZ-1.4

SZ-1.4 [76] introduced the pipeline that later SZ versions build on. A multidimensional Lorenzo predictor predicts each value from its already reconstructed neighbours, in two dimensions as $\hat{x}_{i,j} = x_{i-1,j} + x_{i,j-1} - x_{i-1,j-1}$. The prediction error is quantized into intervals of twice the error bound (error-controlled quantization), and the resulting quantization codes are Huffman-coded. Unlike SZ1, SZ-1.4 thus moves the error-bounding responsibility completely to the quantization side: after prediction, the difference between predicted and original values is stored and quantized according to the error bound. Values that are completely unpredictable are still stored verbatim. This changed the data properties for subsequent stages of the compression pipeline: instead of the decorrelated function-code stream, the outgoing stream contains the quantized differences between the predicted and the original stream.

Starting with SZ2 [49], later members of the SZ family [50, 96] have moved from choosing a best-fit function per value to choosing a best-fit function per block of data, using more elaborate functions than SZ1 [13] did. These functions, primarily the Lorenzo predictor, are modeled to work well on the smooth progressions often present in multi-dimensional, scientific data sets.

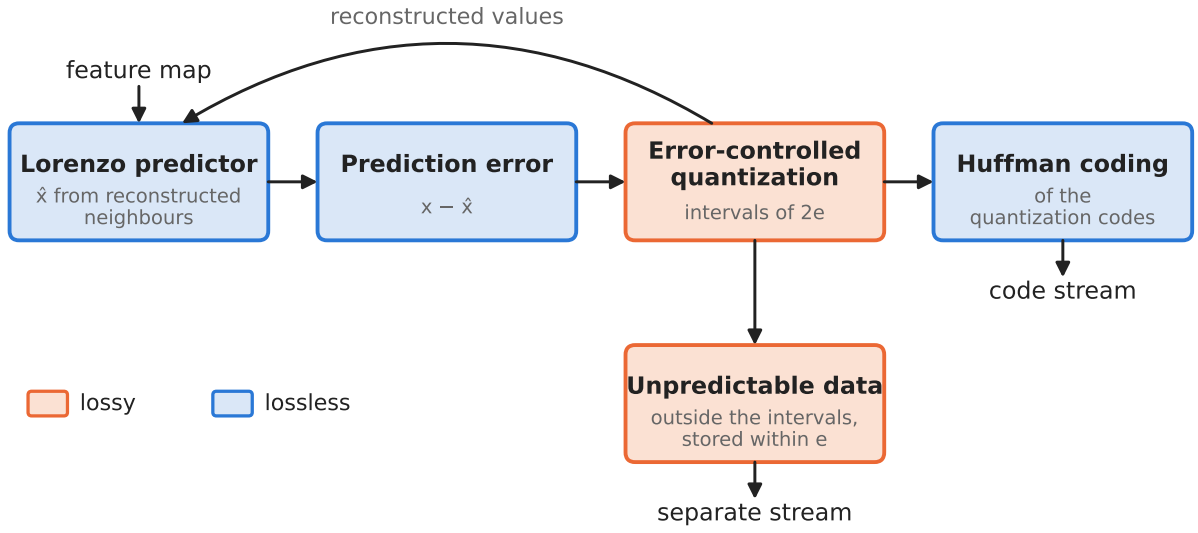


Figure 2.18

SZ-1.4 [76] applies the Lorenzo predictor to all values; only SZ2 [49] evaluates multiple prediction functions for each block and chooses the one that compresses the data best. For our comparison pipeline, we follow SZ-1.4 and use the one-layer Lorenzo predictor as a fixed choice. The rest of the pipeline works similarly.

2.6 Related Work

Two strands of related work meet in this thesis. The first compresses neural-network data — weights and activations, during training and inference — and is predominantly lossless where it targets inference activations (section 2.6.1). The second is error-bounded lossy compression, developed for scientific data, from which our design derives (section 2.6.2). We assess both against the selection criteria of section 2.5 and close with a summary (section 2.6.3).

2.6.1 Compressing Neural-Network Data

2.6.1.1 Weights

Weights are the most widely studied target of neural network compression. They are known before deployment, can be compressed offline, and their compressed size is fixed at compile time; retraining to recover accuracy is common.

Deep Compression [31] prunes, quantizes and Huffman-codes weights (section 2.5.3.1), and Weightless [65] encodes them lossily with Bloomier filters, retraining the network around the introduced errors.

Several works apply lossless entropy coding to weights for model distribution and serving: ZipNN [34] Huffman-codes the skewed exponents of floating-point weights, Heilper and Singer [33] extend this to low-precision formats and K/V caches, and Agrawal et al. [1] reuse an averaged Huffman codebook across shards instead of building one per batch. Silfa et al. [71] Huffman-code groups of binary weights in BNNs.

Closest to accelerator hardware, Metz et al. [53] combine mixed-precision quantization with entropy coding and decode weights with an ANS decoder between DRAM and accelerator just before computation.

2.6.1.2 Activations during Training

During training, activations have to be kept for back-propagation and dominate the memory footprint on GPUs. JPEG-ACT [21] exploits the spatial correlation in CNN feature maps and compresses them with a JPEG-like pipeline of DCT, quantization, run-length and Huffman coding. AC-GC [20] bounds the compression error and adapts the bound dynamically so that convergence is guaranteed, and COMET [42] applies an SZ-based error-bounded compressor to training activations (section 2.6.2.3). These works compress whole tensors on GPUs, and the error affects convergence rather than inference accuracy. They show, however, that feature maps contain spatial correlation that a compressor can exploit.

2.6.1.3 Activations during Inference: Lossless

Lossless compression of activations at inference time mostly exploits their sparsity and low entropy. Weng et al. [86] propose a block-based format that exploits both sparsity and the similarity between channels, with encoder and decoder implemented in hardware. CAT [4] adds an entropy penalty to the training loss so that quantized activations become cheap to entropy-code, reducing bandwidth by 2–4×, but requires retraining. Lane Compression [44] splits values into bit lanes and selects a cheap coder per lane from offline profiling, matching Huffman coding at a much lower hardware cost. Its authors also measure the Shannon limit of quantized inference activations at only about 1.6–3.4×, which bounds what any lossless scheme can achieve.

DEF [56] is closest to our setting: it encodes streamed feature maps as differences to the previous value, which corresponds to SZ1's preceding-neighbour predictor, but aims at fewer bit transitions on the off-chip bus, i. e. lower power, rather than less data. In streaming accelerators, SMOF [80] evicts activations to off-chip memory (section 2.4) and optionally encodes them with RLE or Huffman coding, with gains that vary between networks. SMOF is the primary motivation for our work.

2.6.1.4 Activations during Inference: Lossy

Lossy compression of activations at inference time has received less attention. Nakahara et al. [61] compress the input images of a streaming ResNet accelerator with a simplified JPEG to relieve the host–FPGA link, at an accuracy loss of 0.3 points. Shao et al. [70] compress interlayer feature maps with a hardware 8×8 DCT, quantization and sparse coding, reducing them by 1.4–3.3× at less than 1 % accuracy loss, and Lim and Jun [51] compress weights and activations with an embedded variant of ZFP, achieving higher accuracy than 8-bit post-training quantization at the same bit budget. CompAct [95] uses a lossy run-length coding for on-chip activation buffers of systolic arrays, in which a run only ends when a value deviates from it by more than a per-layer threshold; the thresholds are chosen greedily under an accuracy budget.

Apart from Nakahara et al., who only compress the network input, these works target time-multiplexed accelerators. The transform-based methods need blocks of values and do not bound the error per value. CompAct comes closest to our approach, as its lossy run-length coding is effectively an error-bounded prediction from the preceding value, but it does not address the activations held back by skip connections in streaming accelerators.

2.6.2 Error-bounded Compression

2.6.2.1 Prediction-based SOTA Compressors

Since SZ was first proposed in 2016 [13], a large number of derivatives have appeared, tailoring the compressor to new data sets, quality metrics and platforms [14]. We do not attempt a complete overview, but highlight the members that mark conceptual changes relevant to our setting.

Our starting point is the original SZ compressor (SZ1) as described in section 2.5.2.4 and section 2.5.3.3: it decorrelates the data stream by encoding, for every value, *which* of three curve-fitting functions predicts it within the error bound, and stores values that cannot be predicted as outliers. The resulting stream of codes is then compressed using the LZ77 dictionary compressor.

The SZ authors have kept the general prediction mechanism, but stepped away from the per-value prediction function choice. Already SZ-1.4 [76] replaces the choice among curve-fitting functions by a single multidimensional Lorenzo predictor and introduces error-controlled quantization: the difference between the prediction and the original value is quantized into intervals of twice the error bound, and the resulting quantization codes are Huffman-coded. The encoded stream thus no longer describes the choice of predictor, but the quantized prediction residual. All later SZ versions retain this structure and refine the predictor instead: SZ2 chooses per block between the Lorenzo predictor and a linear regression [49], SZauto adds second-order predictors and selects parameters from a sample of the data [96], and QoZ uses multi-level interpolation and tunes its parameters towards a user-specified quality metric [52]. SZ3 finally generalizes the family into a modular framework of interchangeable preprocessing, prediction, quantization and encoding stages [50].

These refinements target smooth, multidimensional scientific data sets, which are stored as a whole before compression. Regression over blocks and interpolation from points on both sides require buffering large parts of the data, which conflicts with values arriving one at a time in a streaming accelerator. cuSZ [77] addresses a different bottleneck: its dual quantization removes the dependency of predictions on previously reconstructed values, enabling massively parallel compression on GPUs. Outside of SZ, LFZip [6] applies the same prediction–quantization–entropy coding structure to time series on constrained edge devices.

2.6.2.2 Hardware Implementations

GhostSZ [88] was the first FPGA implementation of SZ and uses SZ1's curve-fitting model, which suits values arriving one by one. Its main challenge is the dependency of predictions on reconstructed values, which it resolves with a staged pipeline; floating-point arithmetic and a Gzip stage, however, make it costly. The Gzip stage (LZ77 and Huffman coding) alone occupies 234 BRAM blocks and about as much logic as a complete prediction pipeline, which needs only a single BRAM block, and its BRAM consumption limits how many pipelines fit on the device [88]. Its successors waveSZ [78] and CEAZ [94] implement the Lorenzo-based model of later SZ versions instead, using a wavefront memory layout and dual quantization with offline Huffman tables, respectively. These designs target the throughput of HPC data on floating-point values and require line buffers for multidimensional prediction.

Outside the SZ family, ZFP-V [74] shows the same for the transform-based compressor ZFP: its algorithm had to be modified for hardware at a small cost in compression ratio, and as a block transform it has to buffer complete blocks before compressing them.

At the lightweight end, SZx [90] restricts an error-bounded compressor to additions, subtractions and bitwise operations, albeit on CPUs and GPUs. In memory compression, AVR [19] downsamples blocks of approximation-tolerant data and reconstructs them by bilinear interpolation, and its successor MemSZ [18] replaces downsampling with SZ1's prediction model, compressing blocks of 256 values in 16 cycles at compression ratios of up to 16:1. MemSZ shows that SZ1's model can be implemented cheaply in hardware, but it operates on blocks between cache and memory controller rather than on a raster-order stream. Our design starts from this lightweight model and adapts it to the one-value-per-cycle streams of a streaming accelerator.

2.6.2.3 Error-bounded Compression for Neural Networks

The authors of SZ have applied their error-bounded compressors to neural networks themselves. DeepSZ [41] compresses the pruned weights of fully-connected layers, which dominate the storage of networks such as AlexNet and VGG-16, with SZ, achieving compression ratios of 46× and 116×. It assesses the accuracy loss of each layer for different error bounds and then selects an error bound per layer under an overall accuracy constraint, assuming that the losses of individually compressed layers add up approximately linearly.

COMET [42] applies a modified cuSZ to the activations that are stored for back-propagation during CNN training on GPUs, adapting the error bound per layer to the training status, and reduces training memory by up to 13.5× with little or no loss of accuracy.

Both approaches are complementary to ours. DeepSZ reduces the size of a model for storage and transfer, and the weights are decompressed before inference; a model compressed with DeepSZ could still be deployed with our compressor. COMET compresses whole tensors on GPUs, and its error affects convergence rather than inference accuracy. Their per-layer selection of error bounds under an accuracy constraint, however, resembles our per-connection configuration, and we revisit the assumption of linearly adding losses when several connections share an accuracy budget.

That the tolerance to errors varies across a network is also shown outside of compression: Gambardella et al. [24] inject permanent stuck-at faults into individual output channels of FINN accelerators for quantized networks and observe accuracy drops of up to 10.81 % from a single faulty channel, depending strongly on both the layer and the channel. This supports configuring the compressor per connection rather than with a single global error bound.

2.6.3 Summary

Table 2.1 compares the closest related works against the requirements of section 1.2. Lossless schemes for inference activations fit streaming accelerators, but are limited by the entropy of the data. Lossy schemes for inference activations target time-multiplexed accelerators and mostly rely on transforms over blocks of values without a per-value error bound. Error-bounded compressors provide this bound and have lightweight hardware implementations, but have only been applied to weights, training activations and scientific data. No prior work applies error-bounded lossy compression to the activations evicted by a streaming accelerator.

Work	Data	Phase	Platform	Lossy	Error-bounded	No retraining	Streaming
SMOF [80]	A	Inference	FPGA (dataflow)	✗	–	✓	✓
DEF [56]	A	Inference	FPGA (memory bus)	✗	–	✓	✓
Lane [44]	W, A, G	Both	ASIC	✗	–	✓	✓
CAT [4]	A	Inference	Custom HW	✗	–	✗	✓
CompAct [95]	A	Inference	ASIC (systolic array)	✓	✓	✓	✓
Shao et al. [70]	A	Inference	FPGA/ASIC	✓	✗	✓	✗
Lim & Jun [51]	W, A	Inference	FPGA	✓	✗	✓	✓
DeepSZ [41]	W	Offline	CPU	✓	✓	(✓)	–
COMET [42]	A	Training	GPU	✓	✓	–	✗
GhostSZ [88]	Scientific	–	FPGA	✓	✓	–	✓
AVR [19]	Memory	–	Memory controller	✓	✓	–	✗
MemSZ [18]	Memory	–	Memory controller	✓	✓	–	✗
This thesis	A	Inference	FPGA (dataflow)	✓	✓	✓	✓

Table 2.1: Comparison of the closest related works. Data: W = weights, A = activations, G = gradients. Streaming: compresses values in arrival order without buffering blocks or whole tensors. – = not applicable, ? = not reported, (✓) = no retraining after compression, but pruning beforehand requires it.

3 Design Space Exploration

In the following, we explore a set of generalizations to the first version SZ compressor, as presented in Di and Cappello [13]. We focus our work on this prediction-based compression method, because previous work, especially Eldstål-Ahrens and Sourdis [18], has shown that very light-weight implementations with minimal amounts of buffering are possible. The compression model also seems to be a good fit for the streaming data-flow (SDF) architecture of streaming FPGA accelerators, per-value predictions rely on the availability of previous values.

We start off the rest of this chapter with a description of our methodology for the Design Space Exploration (DSE). This includes a detailed description of our problem framework, which informs the choice of algorithms, the estimation of hardware costs and the evaluation of trade-offs. The DSE itself will start with a baseline evaluation of the SZ1 [13] algorithm. This is followed by the modification of different aspects of the compression algorithm and concluded by a summary of the modifications.

The modifications fall into the following categories:

- **Error bounding:** a block-level error threshold in addition to the per-value bound of SZ1 (section 3.3).
- **Prediction functions:** spatial, regression-based and stochastic prediction functions in addition to the curve-fitting functions of SZ1 (section 3.4).
- **Function selection:** choosing the active prediction functions per channel, statically or adaptively at runtime (section 3.5).
- **Further pipeline stages:** downsampling before and lossless coding after the prediction stage (section 3.6).
- **Parameter optimization:** finding the parameters that maximize the compression ratio within a given accuracy budget (section 3.7).

3.1 Methodology

This section describes the methodology of our design space exploration. We first define the problem framework (section 3.1.1), i.e. the setting in which the compressor operates and the assumptions we derive from it. We then state the

goal of the exploration and the metrics by which we compare design points, distinguishing between metrics measured in simulation and hardware costs that we estimate. Finally, we describe the experimental setup used throughout this chapter.

3.1.1 Problem Framework

The compression schemes explored in this chapter are designed for the eviction scenario introduced by SMOF [80], in which the activations of long-skip connections are moved to off-chip memory to free on-chip buffer space. In the following, we describe the assumptions this scenario imposes on the compressor. They are not strict constraints: in principle, any algorithm could be inserted into the pipeline. However, they define which designs fit the streaming dataflow (SDF) compute model naturally and thereby guide both the choice of algorithms and our estimation of hardware costs. Figure 3.1 illustrates the scenario for a larger network than in Fig. 2.3, with two consecutive residual blocks whose long-skip connections are both evicted.

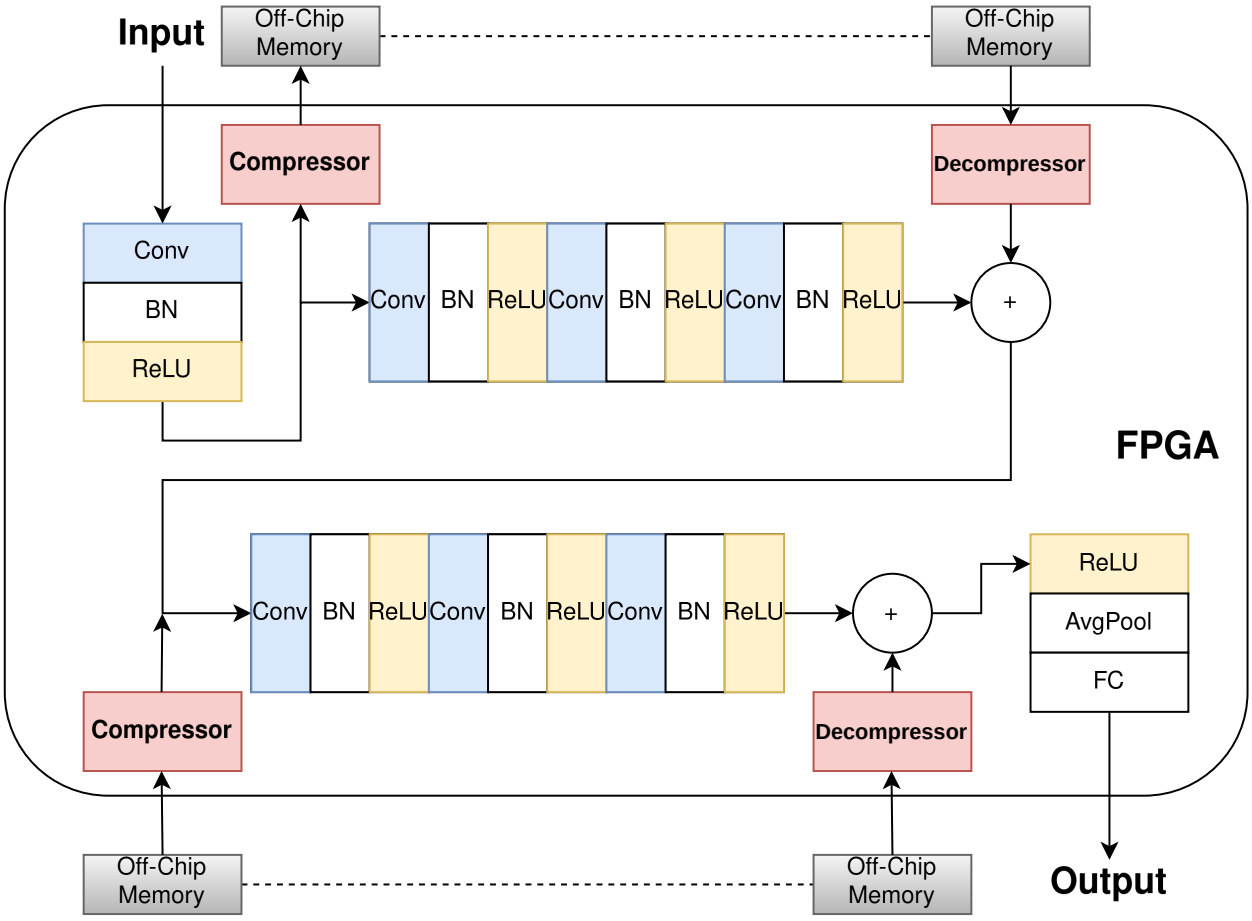


Figure 3.1: A larger network than in Fig. 2.3: two consecutive residual blocks mapped to an FPGA, both with an evicted long-skip connection. Eviction frees the on-chip memory of both branch synchronization buffers, so that larger networks fit on the device, but every evicted connection adds its own stream to the memory bus and increases the pressure on its bandwidth. Each evicted connection has its own compressor at the branch point and its own decompressor before the element-wise addition, both on-chip. All four off-chip memory boxes denote the same off-chip memory (dashed lines), so the compressed streams of all evicted connections share one memory bus, together with the network input and output.

Compressor Placement A compressor is inserted at each evicted skip connection and compression hardware duplicated for each channel; matching decompressors lie before the activation requiring the evicted activations before decompression. Both modules reside on-chip: since the purpose of compression is to reduce the pressure on the memory bus, only the compressed representation may cross it. As a consequence, compressor and decompressor are located on the same device and can operate in lockstep. Any state the compressor derives from the data, such as the previously reconstructed values a prediction-based scheme relies on, can be reproduced deterministically by the decompressor and does not need to be transmitted.

Data Arrival Model We assume a fully unrolled design in which the compressor receives one new value per channel in every cycle. Let $X \in \mathbb{R}^{C \times H \times W}$ be the activation tensor of an evicted connection. Each channel c then arrives as a stream $X_{c,0}, X_{c,1}, \ldots, X_{c,n}$ with $n = HW - 1$, in row-major order, which is compressed, evicted, loaded back, and decoded into the reconstructed stream $\hat{X}_{c,0}, \hat{X}_{c,1}, \ldots, \hat{X}_{c,n}$. Throughout this chapter, $X_{c,i}$ denotes the raw value at stream position i of channel c and $\hat{X}_{c,i}$ its reconstruction. From the compressor's perspective, the data is therefore strictly one-dimensional: the spatial structure of the feature map is not exposed explicitly, but is still present implicitly in the order in which values arrive. Exploiting it requires buffering previously seen values, whose cost scales with the distance between correlated values in the stream (e.g. one row of W values for vertical neighbours).

Channel Parallelism Each channel is compressed independently by its own compressor instance, so the number of compressors equals the number of channels of the evicted connection. We deliberately leave hardware-level design optimizations out of scope. For example, a concrete hardware implementation may interleave several channels onto a shared compressor, trading logic for additional buffer space. Such optimizations concern how a compressor is mapped to hardware rather than the compression scheme itself; the designs presented here can be adapted to them, but we do not evaluate them. Treating channels independently also means we only exploit correlations within a single feature map and disregard correlations across channels, even though these are known to exist [86]. We discuss their potential in section 5.4. Channel-parallelism duplicates compression hardware per channel which means that resource overhead in terms of LUTs and FFs increases linearly with the number of channels.

Models and Connections We restrict our evaluation to CNN classifiers whose long-skip connections merge through element-wise addition, which covers both residual [32] and inverted residual [67] connections. Other merge operations, such as concatenation, may produce activations with different characteristics. We treat the models as fixed: they are pre-trained and are not retrained or fine-tuned for compression. All models are quantized to 8 bits with per-channel asymmetric (affine) linear quantization, in which all values of a channel share a common scale and zero point. Our compression is complementary to quantization or any prior compression method the model has been subjected to. It offers an additional trade-off for the toolflow: a small loss in accuracy in exchange for further headroom in memory bandwidth.

The DSE in this chapter is carried out on a single connection; section 3.1.7 describes our choice.

3.1.2 Goal

The goal of this exploration follows from the objectives stated in chapter 1: starting from SZ1 [13], we look for compressor designs that maximize the compression ratio for a given loss of inference accuracy, while remaining cheap enough to be replicated across the channels of an evicted connection. Since we do not provide hardware implementations, the compression ratio serves as a proxy for the reduction in memory bandwidth, and thereby for system performance; we complement it with estimates of the hardware cost of each design.

The tolerated loss of inference accuracy is specified by the user as an *accuracy budget* Δ: the largest drop in top-1 accuracy, in percentage points, that the user is willing to trade for compression. The budget is the parameter through which the user configures the compressor, and it is how we compare designs: for a given budget, the better design achieves the higher compression ratio. Throughout this chapter, we use budgets of 1 to 3 pp.

3.1.3 Relevant Metrics

To compare different points in the design space, we need metrics that capture both sides of the trade-off we explore: what lossy compression gains us and what it costs. The gain is a smaller representation of the evicted activations; the cost is an accuracy loss in the model and the hardware resources the compressor occupies. Our software setup lets us measure the first two directly, while the hardware cost can only be estimated. We therefore split the metrics into three groups: the *measured* metrics, which form the basis of all comparisons in this chapter; the *estimated* metrics, which give a first indication of the hardware cost of a design; and the metrics that are common in related work on FPGA acceleration but that we cannot provide without a hardware implementation.

3.1.4 Measured Metrics

These are the metrics most central to our evaluation and are directly measured in our software setup. We will use these metrics, and in particular the relationship between accuracy and compression ratio, to gauge the performance of our design modifications and baselines for the evaluations in the next chapter.

Accuracy Approximating activations introduces errors, which generally impact the inference accuracy negatively. For every input image, a classifier outputs a score for each class. The *top-1 accuracy* is the share of images, in percent, whose highest-scoring class is the correct one: $A = \frac{100}{|D|} \sum_{(x,y)\in D} \mathbb{1}\left[\arg\max_k f_k(x) = y\right]$ for a set D of images x with labels y and class scores $f_k(x)$. It is the standard measure for image classifiers and our main metric to gauge the performance degradation the model suffers from lossy compression.

Compression Ratio The compression ratio (CR) is the ratio between the size of the uncompressed and the compressed representation of a connection,

$$\mathrm{CR} := \frac{b_{\text{in}}\, N}{b_{\text{code}}\, N + b_{\text{in}}\, N_{\text{out}}}, \tag{3.1}$$

where $N = CHW$ is the number of values per sample, N_{out} the number of outliers among them, $b_{\text{in}} = 8$ the bit width of the quantized activations and $b_{\text{code}} = \lceil \log_2(|F| + 1) \rceil$ the width of a code for $|F|$ active prediction functions and the outlier code. Every value costs a code; an outlier additionally costs its verbatim value. CR is our main metric to gauge the positive impact of lossy

compression. It serves as a proxy for increased system performance in a potential FPGA implementation, as higher CRs reduce the bandwidth an evicted connection requires and thus allow a more efficient use of memory bandwidth and on-chip memory. Since we lack concrete hardware designs, CR is the closest proxy to system performance we can provide.

Equation (3.1) also shows a fundamental limit of the prediction-based compression model: even without a single outlier, every value costs b_{code} bits, so that

$$\text{CR} \leq \frac{b_{\text{in}}}{b_{\text{code}}}.$$

With the four codes of SZ1 (three prediction functions and the outlier code), $b_{\text{code}} = 2$ and CR is bounded by 4, regardless of how lax the error thresholds are. Larger function sets widen the code and lower this bound.

Data that is known statically after profiling, such as the channel magnitudes M_c, the quantization parameters or a fixed selection of prediction functions, is not counted towards the compressed size. It is available to compressor and decompressor alike and never has to cross the memory bus. We instead account for it as additional buffering (section 3.1.5). Data that is derived at runtime is counted, such as the per-batch selection of prediction functions under adaptive selection (section 3.5). A selection that is fixed beforehand counts as static data.

Error Channels differ widely in their magnitudes, so we do not measure errors in absolute terms but relative to the magnitude $M_c := \max_c - \min_c$ of the channel a value belongs to (section 3.2). The **pointwise relative error** of the reconstruction $\hat{X}_{c,i}$ of a value $X_{c,i}$ is

$$\varepsilon_{c,i} := \frac{\left|X_{c,i} - \hat{X}_{c,i}\right|}{M_c}, \tag{3.2}$$

and the **average relative error** over a set S of positions is its mean,

$$\varepsilon(S) := \frac{1}{|S|} \sum_{(c,i) \in S} \varepsilon_{c,i}. \tag{3.3}$$

The value error threshold τ_v bounds the pointwise error $\varepsilon_{c,i}$ of every value, the block error threshold τ_b bounds the average error ε over a block of a single channel (section 3.3). Taking S as the whole activation tensor, we use ε to report the error a compression method introduces into a connection.

3.1.5 Estimated Metrics

Lacking concrete hardware implementations, we cannot report the resource utilization of a design. We can, however, derive two quantities directly from the definition of a compression scheme that determine much of its hardware cost: the arithmetic it performs and the data it has to keep on-chip. Neither replaces a synthesis result, but together they allow a first assessment of the cost of a technique and a relative comparison between design points. Both are given per channel. Since every channel is compressed by its own compressor instance (section 3.1.1), the cost of an evicted connection is C times the per-channel cost.

Arithmetic We count the floating point operations (FLOPs) a compressor performs per incoming value. Every addition, subtraction, multiplication, division, absolute value and comparison counts as one FLOP; we do not weight

them by their implementation cost, although, for example, a multiplication typically maps to a DSP slice while a comparison is cheap in LUTs. In the fully unrolled design we assume, the compressor processes one value per cycle, so all operations that may be needed for a value have to be instantiated in hardware, whether or not their result is used. The FLOP count is therefore a proxy for the logic and DSP resources of a compressor, not for its latency or energy.

Buffer Space We count the number of values a compressor has to store on-chip to process the stream. This includes the reconstructed values that prediction functions read (section 3.4.5), as well as state such as the running error sum of the block error threshold (section 3.3) and per-channel parameters such as the channel magnitude M_c. Because data arrives as a one-dimensional stream, any access to values in previous rows of a featuremap requires buffering at least one full row of width W. Buffer space is particularly relevant in our setting: the purpose of eviction is to free on-chip memory [80], so every value the compressor buffers reduces the memory saved by evicting a connection.

We do not account for the control logic, the encoding of codes and outliers into the compressed stream, or the decompressor. The decompressor mirrors the prediction and reconstruction of the compressor and therefore has comparable arithmetic and buffering requirements, so the cost of a compression scheme roughly doubles once both sides are taken into account.

3.1.6 Metrics out of Scope

Since this thesis lacks hardware implementations, we are not able to supply metrics that would have made the work more readily comparable to related work in the space of FPGA CNN acceleration. Here we list metrics commonly found in related work to gauge the efficacy of designs, which we are unable to provide at this point in time:

Bandwidth requirements Toupas et al. [80] used lossless compression techniques to reduce the volume of off-chip evicted activations. The primary motivation for this is to relieve the pressure the evictions have on the memory bus. Smaller evictions allows for more evictions, which in turn frees up more BRAM/URAM to be used for increasing parallelism or for merging subgraphs. The compression ratios we provide for evaluated compression schemes are merely a proxy for a evicted connections bandwidth requirements. Bandwidth requirements of a connection can also be a proxy of the overall performance in MACs/second.

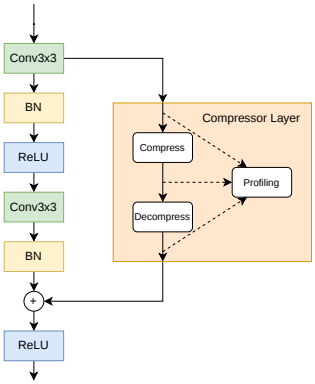


Figure 3.2: Compute graph of a residual block after insertion of our custom compressor layer. The activations entering the highlighted node are compressed and decompressed, simulating the compression-and-eviction process Toupas et al. [80] use to trade off memory bandwidth and on-chip buffering requirements.

3.1.7 Experimental Setup

Simulation All compressor designs are evaluated in software, using PyTorch 2.10 on a GPU. We insert a custom compressor layer into the compute graph of the model at the position of the evicted connection (Fig. 3.2). It compresses the activations passing through it and immediately decompresses them again, so that the rest of the network operates on the reconstructed values $\hat{X}$, just as it would after the activations have been evicted and loaded back.

The compressor layer follows the streaming model of section 3.1.1. Every channel is processed as a stream in row-major order, and every code depends only on values that precede it in the stream, so that the GPU implementation produces the same codes as a sequential compressor would. Within a batch, the featuremaps of a channel are concatenated into one stream. Predictions never reach across

featuremap boundaries: both the availability of rays (section 3.4.1) and the warm-up of section 3.3 are defined per featuremap. The only mechanisms that carry state across featuremaps are the sliding window of the block error threshold (section 3.3) and those that adapt explicitly between batches, namely the dynamic function selection (section 3.5) and the adaptive Huffman codec (section 3.6). The `random` predictor (eq. (3.22)) is seeded with the stream position, so all runs are deterministic.

Quantization We simulate the 8-bit per-channel quantization of section 3.1.1 by fake quantization: an additional layer after every computational layer of the network quantizes its output with the scale and zero point profiled for each channel, and dequantizes it again. The following layers therefore compute in floating point, but on values restricted to the 8-bit grid. The compressor operates on the quantized integer representation, shifted per channel so that the profiled channel minimum maps to zero; its output is dequantized again before it is passed on. The arithmetic of the compressor itself, in particular the evaluation of the prediction functions, is however carried out in floating point. A hardware implementation would more likely use integer or fixed-point arithmetic, which can produce slightly different predictions. We do not model this difference.

Data We split the ImageNet validation set into two disjoint sets: a *calibration set* of 5000 images, five per class, and a *test set* of the remaining 45000 images. All profiling is done on the calibration set. This covers the quantization parameters, the channel magnitudes M_c, the channel means μ_c of the `random` predictor, and every other parameter that we treat as statically known (section 3.1.5). Unless noted otherwise, all evaluations in this chapter run on the calibration set as well. Since our error thresholds are relative to profiled magnitudes, this is the most favourable setting for them. Section 3.7.3 tests how well compression ratio and accuracy hold up on the unseen images of the test set.

Reference accuracy We report top-1 accuracy. Accuracy drops are given relative to the 8-bit quantized model without compression, evaluated on the same set as the compressed model, and are stated in percentage points (pp).

Model and connection Approximating different connections has very different effects on inference accuracy, depending on factors such as layer depth, featuremap dimensions and layer type (see sections A.9.1 to A.9.3). We carry out the DSE on a single, particularly sensitive connection: the first skip connection of RegNetX-400MF [64] in its pre-trained torchvision version, `trunk_output.block1.block1-0`. The compressed tensor is the output of the block's projection, the operand the residual addition consumes, with $C \times H \times W = 32 \times 56 \times 56$. This connection is the one that is evicted and compressed; all other connections of the network remain uncompressed. In the eviction scenario of SMOF [80], several connections are evicted at the same time, and their errors add up; we turn to this joint setting in chapter 4. The compute graphs of RegNetX-400MF and of the other models of chapter 4, with the positions of their skip connections, are shown in chapter B.

A sensitive connection is where error control matters most: small changes to the compressor show up as measurable changes in accuracy, which makes it easier to tell design points apart. Less sensitive connections tolerate cruder approximation, and simpler methods, such as plain quantization or downsampling, can already compress them well; chapter 4 compares against them. The choice also carries a risk: design decisions tuned on one connection of one model need not transfer to others. Chapter 4 therefore evaluates the final design of this chapter across a wider range of models and connections.

3.2 Standard SZ

We have introduced the prediction-based compressor design of the SZ1 compressor, introduced by Di and Cappello [13], in sections 2.5.2.4 and 2.5.3.3. To establish a baseline for our modifications to the design we first evaluate the compression-accuracy trade-off of using the SZ1 compressor as it was original designed. Note that we will focus on the prediction-based compression module only: The original paper also increased the performance of the compressor further by using quantization on outliers and by additionally using a dictionary-based compression scheme on the code stream. We deliberately exclude these parts in the first part of the DSE to make it easier to quantify the direct impact our modifications have. In section 3.6, we will investigate how to push the performance of the SZ compression scheme further by using additional compression modules, such as downsampling and Huffman coding.

We now define our baseline setup of the SZ1 compressor, to form a rigorous foundation for our following modifications.

Value Error Threshold SZ1 allows modifying the per-value error bounding parameter. Users can choose between using relative errors, absolute errors, or either the minimum or the maximum of the two.

In appendix A.1 we observed that channels can have different magnitudes and value distributions. Motivated by this fact, we adopted a relative error definition. This allows us to define one relative value error threshold τ_v per layer, instead of setting an absolute bound per channel. A common absolute bound for all channels of one layer, could become overly lax for some channels and at the same time, needlessly strict for others.

τ_v is a unit-less, normalized bound, which is used with profiled, channel-specific magnitudes to arrive at an absolute, per-channel bound which can be used for deciding whether a prediction is within the acceptable error tolerance or not. We profile the channel range

$$R_c := [\ \min_c,\ \max_c\ ]$$

defined by channel minimum and maximum and calculate the channel magnitude

$$M_c := \max_c - \min_c\ .$$

Based on M_c and the user-defined, normalized and unitless value error threshold τ_v, the absolute, per-channel error tolerance is defined as

$$\Delta_c^v := M_c\ \tau_v.$$

Feature map traversal A deeply pipelined, streaming mapping of a classification model to a FPGA, such as FPGAConvNet [82] would produce it for a CNN, delivers a new value of the feature map to the compressor every cycle in row-major order. This is the same order Di and Cappello [13] used initially. They had practical reasons for doing it thus, as reordering their multi-dimensional data

to follow a different order brought additional costs in terms of compression time. In our SDF scenario, we could do the reordering on the fly, by buffering past rows. We will explore the trade-offs of different traversal schemes in section 3.4.2.2, but for now we will stick with the row-major traversal scheme.

Baseline Prediction Functions The three SZ1 functions we extend — Preceding Neighbor Fitting, Linear Curve Fitting and Quadratic Curve Fitting [13] — all calculate their predictions from the same stream of values — the stream of reconstructed, previous predictions $p(k) := \hat{X}_{c,i-k}$, $k \geq 1$, where k counts the steps back from the current position i

$$\hat{X}_{c,i}^{(\texttt{constant})} = p(1) \tag{3.4}$$

$$\hat{X}_{c,i}^{(\texttt{linear})} = \texttt{clamp}(\, 0, 2\,p(1) - p(2)\,) \tag{3.5}$$

$$\hat{X}_{c,i}^{(\texttt{polynomial})} = \texttt{clamp}(\, 0,\ 3\,p(1) - 3\,p(2) + p(3)\,) \tag{3.6}$$

Clamping in eq. (3.5) and eq. (3.6) is a first, minimal adaptation of the function set to our data: the compressor operates on the quantized values, shifted per channel so that the channel minimum maps to zero (section 3.1.7), so that no value it compresses is negative. Thus, a negative linear extrapolation and quadratic fit is known to be wrong before it is ever evaluated and is clamped to zero instead of being discarded. Figure 3.3 illustrates the three functions.

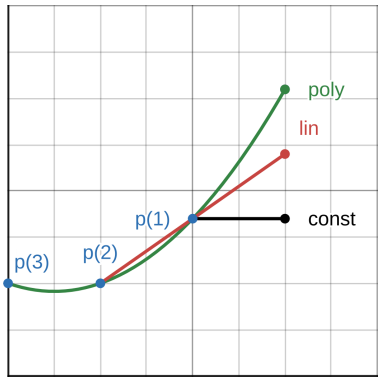


Figure 3.3: Example of constant prediction (black), linear prediction (red) and polynomial prediction (green) and how they are based on their predecessors

Encoding decision Every candidate prediction function f of the active predictor set produces a prediction $\hat{X}_{c,i}^{(f)}$ from the values reconstructed so far; section 3.4.1 defines the individual functions. The encoder always offers the prediction with the smallest absolute error,

$$f_{c,i}^{\star} := \arg\min_{f} \left| X_{c,i} - \hat{X}_{c,i}^{(f)} \right|, \qquad E_{c,i} := \left| X_{c,i} - \hat{X}_{c,i}^{(f^{\star})} \right|, \tag{3.7}$$

and accepts it if and only if it stays within the value error threshold

$$\text{VALUE-OK}_{c,i} \iff E_{c,i} \leq \Delta_c^{v}.$$

An accepted prediction is stored as the code $f_{c,i}^{\star} + 1$ (functions being numbered by their position in the active set) and reconstructed as $\hat{X}_{c,i} = \hat{X}_{c,i}^{(f^{\star})}$. Otherwise the code 0 marks an outlier, whose value is stored verbatim, so that $\hat{X}_{c,i} = X_{c,i}$ and the outlier carries no error at all. We call the resulting sequence of codes the *code stream*; with three prediction functions and the outlier marker, every code takes 2 bits. Since predictions are computed from $\hat{X}$ rather than from X, compressor and decompressor stay in lockstep without any side information being transmitted. Section 3.3 extends this decision rule with a second, block-level threshold. Figure 3.4 illustrates the encoding decision on a short stream.

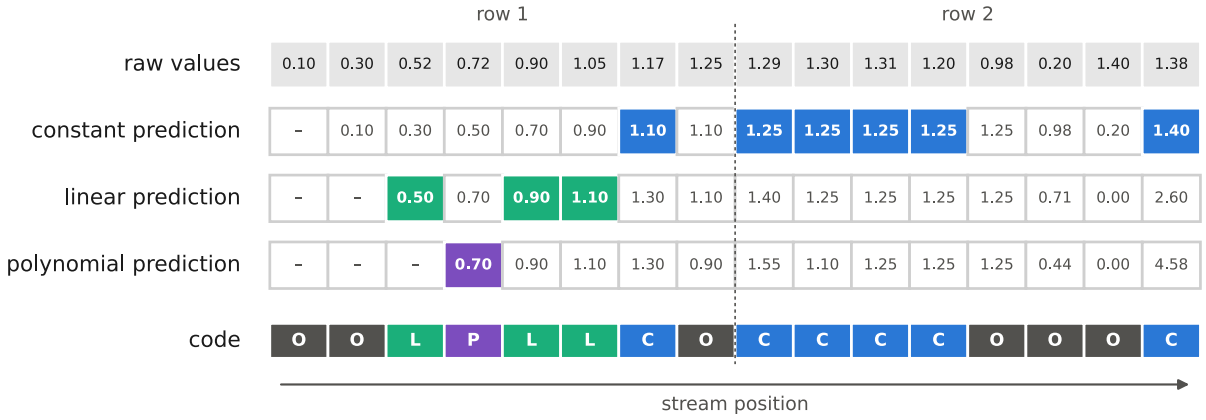


Figure 3.4: The encoding decision of SZ1 on two rows of a featuremap of width $W = 8$, in row-major order (illustrative values, absolute error threshold 0.1). Top: the raw values. Middle: the predictions of the constant, linear and polynomial function (eqs. (3.4) to (3.6)), computed from the reconstructed values; –: not enough preceding values. Bottom: the code stream. At every position, the prediction with the smallest error is accepted if it lies within the threshold (coloured, code C, L or P); otherwise the value is stored as an outlier (O). Since predictions continue from the reconstructed rather than the raw values, the constant prediction stays at 1.25 along the plateau of the second row, while the raw values drift by up to 0.06. Dashed: row boundary.

Baseline Performance To determine the baseline performance, we simply sweep across a range of τ_v and evaluate accuracy and compression ratio for each. Fig. 3.5 shows the compression ratio - accuracy Pareto front for the sweep. It shows that increasing compression will cost increasingly more accuracy. It also shows a central limitation of prediction based compression model: In the SZ setup with four different prediction functions, we need 2 bits to encode which function was used and since we are compressing connections in 8 bit quantized networks, a source symbol is 8 bits in size. Thus, our compression ratio is fundamentally limited at 4:1.

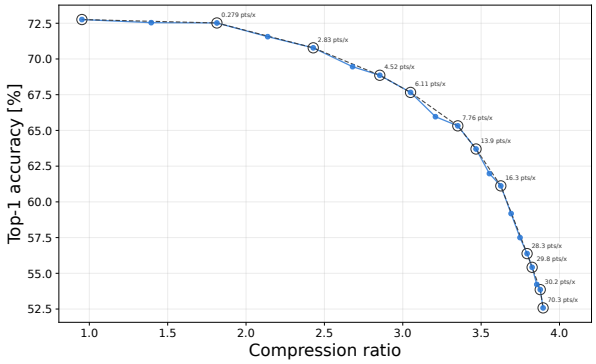


Figure 3.5: Standard SZ1 (constant, linear and polynomial predictor): top-1 accuracy against compression ratio for a sweep of τ_v. The dashed line is the efficient frontier, its edges labelled with the accuracy each further unit of compression ratio costs. The price rises steeply as the ratio approaches the 4:1 limit of the 2-bit code; this is a symptom of the central limitation of the prediction based compression mechanism

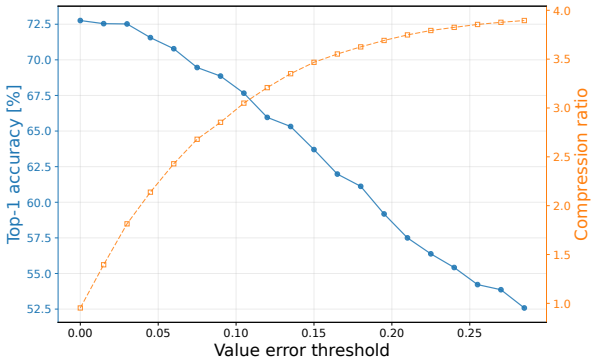


Figure 3.6: The sweep of Fig. 3.5 against τ_v: top-1 accuracy (blue, left axis) and compression ratio (orange, right axis). Accuracy falls roughly linearly with τ_v, while the compression ratio saturates towards 4:1.

Fig. 3.6 shows how increasing τ_v impacts compression ratio and accuracy. There seems to be a linear relationship between τ_v and accuracy; the increase in compression ratio decreases as it approaches the 4:1 limit.

We also looked at the distribution of codes, i. e. how often each prediction function and the outlier marker occur in the code stream. Fig. 3.7 shows us, that one single prediction function—constant prediction in this case—dominates for the majority of runs. This on its own could imply that the constant predictor just fits the data really well, but the steadily decreasing accuracy rather point towards the linear and polynomial prediction functions being ill-suited. Since every prediction costs the same 2-bit code, the compression ratio depends solely on the share of outliers: every outlier additionally stores its value verbatim.

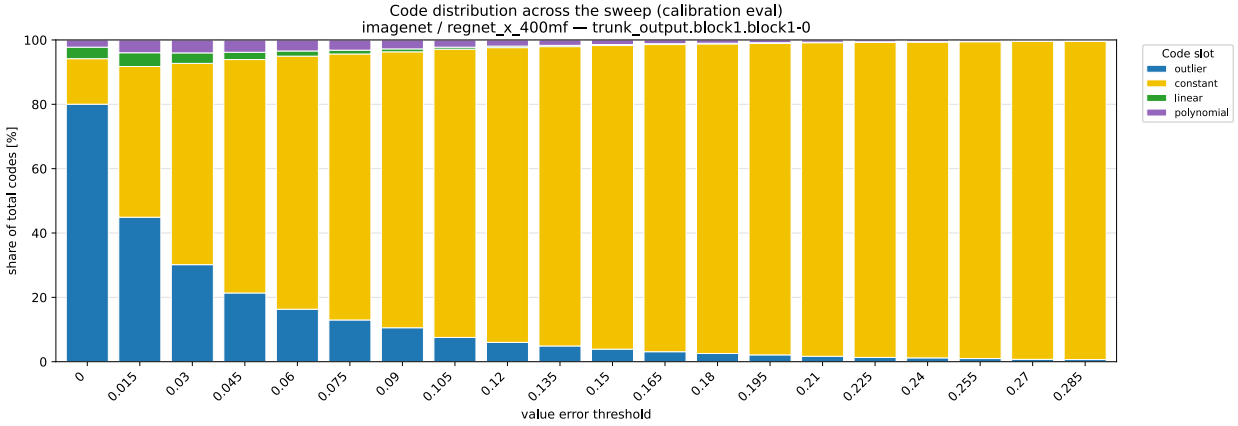


Figure 3.7: Each bar represents the distribution of the compressors codes for a specific τ_v. Along the horizontal axis, we increase τ_v. The compression ratio depends solely on the share of outliers in the overall distribution. Higher outlier share means that more additional values need to be stored to stay within τ_v. Notable in this particular distribution is the fact that the share of polynomial and linear predictions increases at a τ_v of 0.015, but decreases for higher τ_v as the share of constant predictions increase. This is a result of the interactions between prediction functions, which we will analyse in more detail in section 3.4

But what the graph also shows is that for the first step in τ_v sweep, the shares of constant and polynomial predictions actually increased, only to steadily decrease thereafter, while the number of constant prediction increases. Looking back to the definitions of the prediction functions, the reason for this behavior is clear: To guarantee correct reconstruction, the prediction functions need to be calculated from the reconstructed values of previous predictions, rather than the raw values—those are not available to the decompression module. Di et al. [14] called this the *Reconstructed-Data-Driven Policy*. With increasing τ_v values, the constant prediction can be applied more often. If there is one constant prediction preceding, then the linear prediction will actually reduce to a constant prediction as well, because now, its preceding values are the same. The same happens to polynomial predictions when there is two preceding constant predictions (Figs. 3.8 to 3.10). This interaction is visible where compression fails: in a sample of recorded outliers, most follow a flat run of constant predictions, on which the polynomial prediction coincides with the constant one and both miss together (section A.4).

The fact that its still only constant predictions showing up while they are equivalent to linear and polynomial predictions in many positions is merely an artifact of our implementation, which chooses the lower-index prediction function on ties.

Figure 3.11 shows what this compression does to a single featuremap, channel 27 of our connection at a compression ratio of 2.85:1 and an accuracy drop of about 4 pp. The reconstruction shows horizontal artifacts, which are a direct result of the row-major traversal order of the SDF compute model.

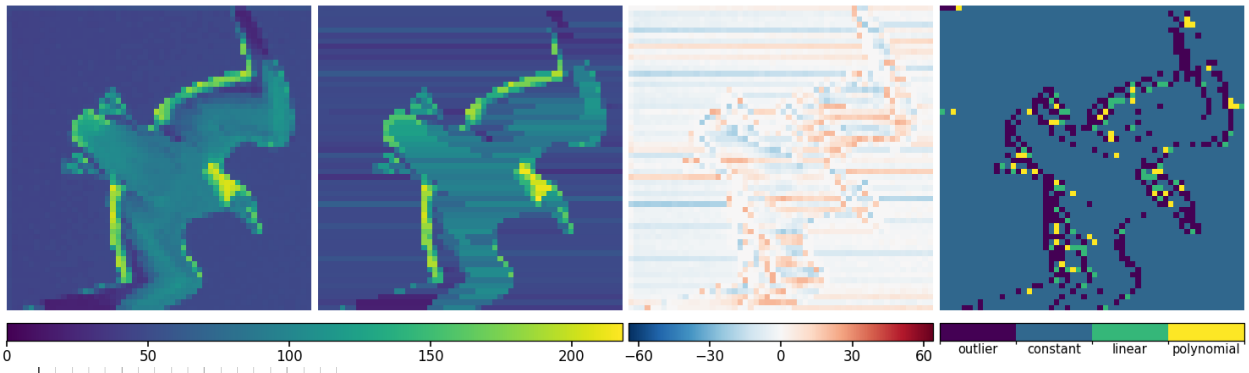


Figure 3.11: Heatmaps showing **(1)** the uncompressed feature map, **(2)** the feature map after reconstruction, **(3)** the error stream, and **(4)** the code stream; Notable here are the horizontal artifacts, which are a direct result of the row-major traversal order in the SDF compute model. This feature map was taken from channel 27 of the first skip-connection of ImageNet/RegnetX 400mf at a τ_v of 0.9. At a compression ratio of 2.85:1, the accuracy drops about 4 percentage points

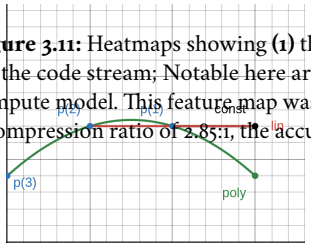


Figure 3.9: A previous constant prediction results in p(1) and p(2) being equal; the linear and constant prediction will produce the same value in this situation

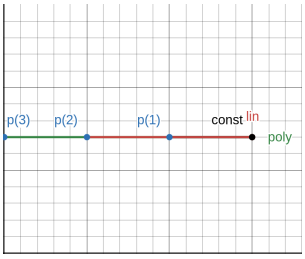


Figure 3.10: With two preceding constant predictions both the linear and the polynomial predictions are reduced to the constant prediction

3.3 Block Error Thresholding

SZ1 uses a point-wise error bounding mechanism only. Inspired by Eldstål-Ahrens and Sourdis [18], we extend it with a per-block error threshold τ_b. Instead of bounding the error per-value, τ_b assures that the average error of all predictions within a block of user-defined size stay below the user-defined error bound. When used together with τ_v, if $\tau_v \leq \tau_b$, there is no effect gained from τ_b. If $\tau_v > \tau_b$, τ_b allows single values to surpass τ_b (if it was applied per-value), while still being bound by τ_v. This requires previous predictions within the same block to fall bellow the average error threshold, such that additional error budget is accumulated for single value to spend it on a cheap prediction, rather than resorting to a expensive outlier. This mechanism is exemplified in Fig. 3.12

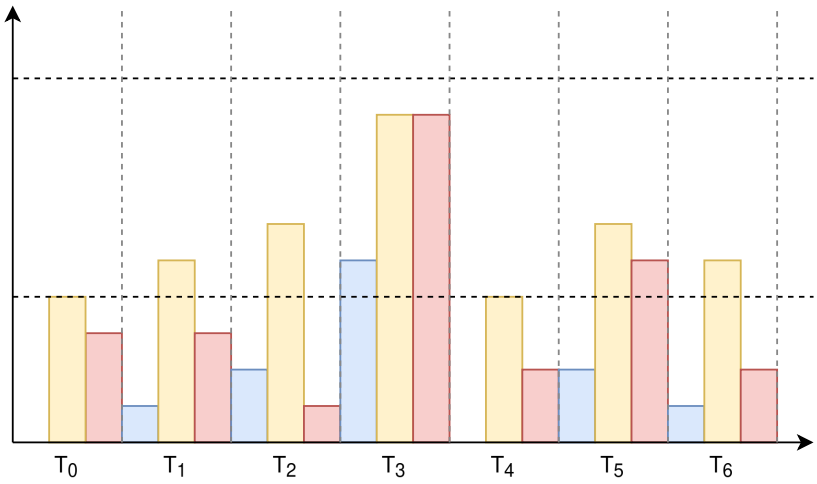


Figure 3.12: Error Budget Accumulation Each step of the horizontal axis represents a new prediction; the height of bars represents the error magnitude; the first horizontal dashed line is τ_b average error threshold and the second dashed line is the single-value τ_v; **Blue** bars represent the error budget made available by previous predictions falling below τ_b; **Yellow** bars represent the budget being available at step T; **Red** bars represent the used budget in that step. Every step the realized error budget falls below τ_b line, budget accumulates. This budget can then be used by hard-to-predict values, which require a one-off higher budget than τ_b would allow; all predictions are still bound by τ_v

The underlying hypothesis for the assumed utility of this bound is that the average error is more important for accuracy bounding than any single error. We have done some first tests (see section A.3.2) to verify this hypothesis, which indicate that the average error alone is not a sufficient proxy for accuracy loss in a network, and error distribution across a featuremap also plays an important role. However, as the rest of this section will show, block error bounding can nonetheless be used to push the Pareto front further and can—combined with

careful tuning of the block size—allow a more refined control of compression trade-offs.

3.3.1 Definition

The definition of τ_b is the same as that of τ_v: It describes the bound in terms of a deviation of a value relative to the channel magnitude. Analogously to τ_v, the block error threshold τ_b becomes an absolute, per-channel tolerance

$$\Delta_c^b := M_c \, \tau_b.$$

Let B be the block currently being filled and k the number of values already committed to this block instance. The block carries the running sum of the absolute reconstruction errors of those values,

$$S_B := \sum_{\substack{j \in B \\ j<i}} \left| X_{c,j} - \hat{X}_{c,j} \right|,$$

to which outliers contribute nothing, as they are stored verbatim. A candidate prediction is admissible with respect to the block budget if and only if the mean absolute error the block would carry after accepting it stays within tolerance:

$$\text{BLOCK-OK}_{c,i} \iff \frac{S_B + E_{c,i}}{k+1} \leq \Delta_c^b.$$

Combining both thresholds, the code emitted at stream position i of channel c is

$$\text{code}_{c,i} = \begin{cases} f_{c,i}^{\star} + 1, \text{if VALUE-OK}_{c,i} \ \wedge \ \text{BLOCK-OK}_{c,i} \ \wedge \ (i \bmod HW) \geq 3, \\ 0 \ \ (\text{outlier, store } X_{c,i} \text{ verbatim}), \text{otherwise,} \end{cases}$$

where H and W are the height and width of the featuremap. The last term is a forced-outlier warm-up: the first three positions of every featuremap are always stored verbatim, because the prediction functions of section 3.4.1 need up to three reconstructed values before they can produce a meaningful prediction.

Note that the two thresholds are not independent. A single value that exhausts the full τ_v consumes the block budget of

$$\text{round}\left(\frac{\tau_v}{\tau_b}\right)$$

stream positions, so a block of N values admits at most $N \, \tau_b/\tau_v$ maximal-error predictions, and the budget for each of them has to be accumulated by positions predicted more accurately first. This ratio, rather than either threshold on its own, is what the parameter heuristic of section 3.7.1 tunes.

Previous plots (see Fig. 3.5) show, that with increasing τ_v values, the prediction accuracy of the model keeps on falling. So the choice of the right τ_v is critical to achieve a good compression ratio/accuracy trade-off. Using τ_b shifts the role of τ_v: Instead of using it as the general error-limiting threshold by setting an upper limit on every predictions and thus, indirectly a upper limit for an entire block, it is now used to limit one-off predictions that surpass τ_b in a single point, while τ_b takes the role of explicitly restricting the average error across a block.

3.3.2 Parameter Space

Here we give a overview of the impact different choices for block size, τ_b and τ_v have.

Varying τ_v with a fixed τ_b Fig. 3.13 compares a sweep of different τ_v values in with and without block error thresholding being used: τ_b can be used to effectively limit both the increase in compression ratio and the accuracy drop. After hitting the point of diminishing returns, increasing τ_v will still push the compression ratio ever so slightly, as it allows more predictions that are farther off the real value to be accepted. This, however, also has a significant effect on the accuracy, which shows that limiting the average error across a block is not enough to limit accuracy drop. In section 3.7, we present a simple heuristic to choose τ_v based on τ_b, which aims to set τ_v optimally.

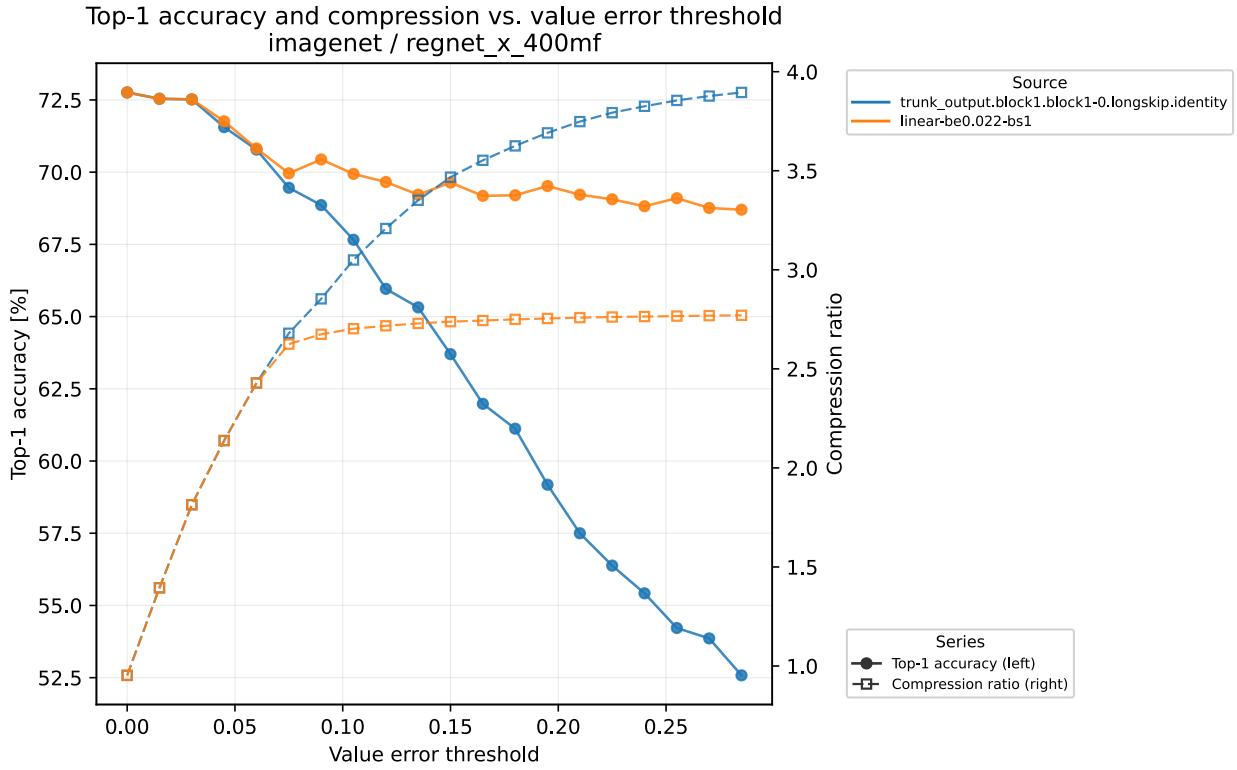


Figure 3.13: Using τ_v only vs. using $\tau_v + \tau_b$: Blue points show the unrestricted τ_v sweep while the orange sweep uses τ_b as an additional error limiting mechanism; this allows a soft bounding of both accuracy and compression ratio ; however, with increasing τ_v some high-error predictions can still be allowed, which still continue the downward-slope of the accuracy line, just massively decreased

Varying τ_b Figure 3.14 repeats the τ_v sweep for different values of τ_b. The choice of τ_b soft-caps the accuracy loss: each sweep levels off at a drop that depends on τ_b. Within each sweep, however, increasing τ_v still admits increasingly large one-off errors, which keep lowering the accuracy while the compression ratio changes only little. Not only τ_b, but also τ_v therefore has to be chosen carefully with respect to the accuracy budget.

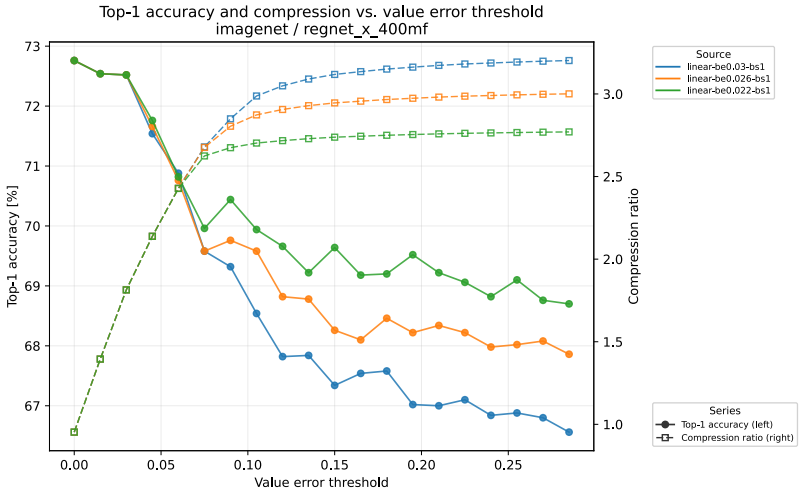


Figure 3.14: Different τ_b can be used to soft-cap the accuracy loss incurred by compression; however, with increasing τ_v increasingly high one-off errors impact accuracy while compression ratio changes only little; this points towards the necessity of choosing not only τ_b but also τ_v carefully towards the allowed error budget

Varying block sizes Figure 3.15 shows sweeps for an increasing number of featuremap subdivisions N_{Subdiv}, i. e. for decreasing block sizes. Smaller blocks can be used together with τ_b to limit the accuracy drop. While τ_b determines how much average error a block may carry, the block size determines how much budget can accumulate within a block, and thus how often a prediction can exceed τ_b. This matters because, as section A.3.2 has shown, a few large errors hurt the accuracy more than the same average error spread evenly over the featuremap.

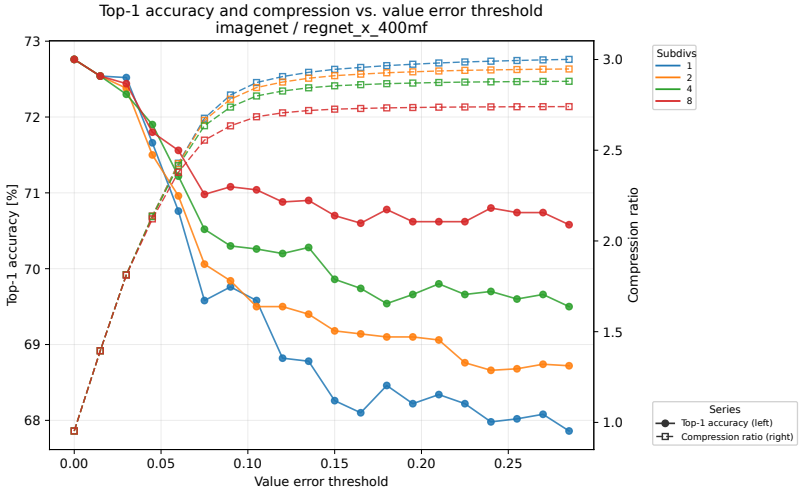


Figure 3.15: Increasing subdivision sweeps: Higher number of subdivisions — corresponding to smaller block sizes — can be used together with τ_b to limit accuracy drop; while the τ_b controls how much average error is allowed within one block, the number of featuremap subdivision controls how much error budget accumulates and thus how often predictions out of the usual τ_b can occur; section A.3.2 has shown that single but larger prediction errors can have an outsizes impact on accuracy while even error distribution does not hurt as much

3.3.3 Block Types

The natural way of accounting for block errors in our SDF problem framework is to treat N successive values as a block, where N is the size of the block. In fact, this is the way we have split feature maps into blocks in runs generating the results of prior graphs.

We want to explore two alternative block accounting mechanisms: **Spatial Blocks**, where the feature map is split into square blocks across the spatial dimensions of the feature map, rather than splitting it into row-major stripes, and using a **Sliding Window**, assuring the average error stays below the threshold

within the last N values, having the block move with current prediction rather than having it anchored on the feature map.

For all of these approaches, we are using a back-loading mechanism, i. e., the first prediction of each block only has the budget for that single value available; the second prediction of the block has both its own budget plus the residual budget of the previous prediction available; budget has to accumulate first before it can be used by predictions. We explored a front-loading mechanism as well, in which all of the blocks budget is immediately available. See section A.3 for more information.

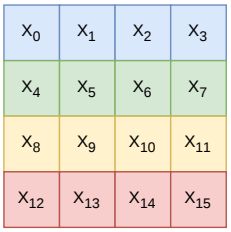


Figure 3.16: This block represents the value matrix of one channel that one sample produces as an output of a layer. The green arrows mark how the values are presented to the compressor, row-major order. This example illustrates the usage of linear blocks, where the block organization follows the natural order of how values enter the compressor

3.3.3.1 Linear Blocks

In deeply pipelined accelerator designs (Fig. 3.16), single compute engines, such as our per-channel compressor, get a new featuremap value each cycle in row-major order. The most natural way to account errors in this scenario is to stick with the row-major order and to keep a sum of those errors for the past values within the current block.

Rather than directly specifying a block size, we define it based on a so-called "featuremap-subdivision" N_{Subdiv}. This number simply specifies how often the feature map should be subdivided. Combined with the feature maps width and height, the block size can be calculated. We restrict N_{Subdiv} to those values that evenly subdivide a given feature map.

The idea behind defining the block size based on N_{Subdiv} rather than directly is that the block accounting should follow the natural boundaries of a two-dimensional feature map. I. e., defining a block to be bigger than a feature map would grant a feature map belonging to the next image in the batch additional error budget to be used. In an extreme scenario where a block spans two successive images, this could mean allowing the second image to be completely distorted, if there was only little error in the first one. Thus, a error accounting mechanism should stick to the spatial boundaries of the feature map. A similar argument can be made when defining blocks that cross over row boundaries. The last value of row one may have little spatial relation to the first value in row two.

In fact, we have observed that configurations with higher N_{Subdiv} (that is, smaller blocks) generally dominate the Pareto front (section 3.7.2). A deeper analysis whether this is caused by strict adherence to feature map boundaries or because smaller blocks limit the number of predictions breaking out of τ_b is out of scope for this work.

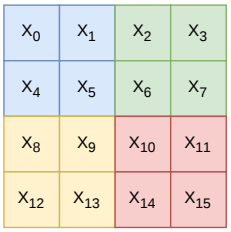


Figure 3.17: Organizing the channels' values into square blocks poses additional costs: State needs to be kept not only for the current block, but also for past blocks that span across multiple rows. This requires additional hardware, but may be a better fit to the nature of the data and distribute error more evenly across the feature map

3.3.3.2 Spatial Blocks

With their *struts-and-arms* architecture, Eldstål-Ahrens and Sourdis [18] automatically distribute the block error threshold across a 16x16 block of values. While this design is motivated by achieving higher rates of parallelism for compressing the block of memory rather than trying to retain spatial properties of the data, it nonetheless made us wonder about the impact of spatially-aware error accounting on inherently spatial data (such as feature maps in CNNs). Di and Cappello [13] also briefly touched upon a more spatially-aware method for linearizing multi-dimensional arrays before feeding them to the compressor, e. g. using space-filling curves. They did, however, not further investigate in the benefits of alternative traversal schemes, as the cost of reorganizing arrays was too high to justify it.

While alternative traversal schemes are possible in a per-channel approach, they would add additional buffering costs. How much additional buffering cost depends on both the traversal scheme and the number of values required per prediction. We will open up the space and allow more buffering in our exploration into what we call *Spatial Predictors* in section 3.4.2.

Rather than aligning blocks alongside rows, spatial blocks subdivide a feature map across both height and width. Fig. 3.17 shows how a 4x4 feature map would be split into 4 blocks. This accounting mechanism, in contrary to the linear one, does not directly follow the arrival pattern of new values to the compressor. This requires keeping additional state for blocks that have not been fully processed.

3.3.3.3 Sliding Window

The third block-error accounting mechanism we experimented with is to keep a sliding window with a size N_{SW}. This sliding window tracks the error of the past N_{SW} values and only passes a new prediction if its error plus the sum of error of the past predictions within the window would not surpass the average bound.

Fig. 3.18 shows how a window with $N_{SW} = 4$ traverses a 4x4 feature map. It follows the natural row-major order, keeping available budget when crossing row and feature map boundaries.

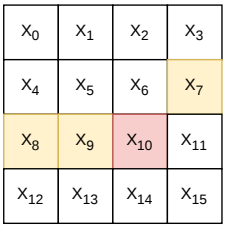


Figure 3.18: Using a sliding window, the average error is bound based on the last N_{SW} values, crossing feature map and row-boundaries. The **red** field is the current value for which a prediction is made; **yellow** fields mark positions over which the average error is calculated to arrive at a bound for the current prediction

3.3.3.4 Comparison of Block Types

Performance In Fig. 3.19, we compare the different block error accounting mechanism with each other based on their compression ratio vs. accuracy performance. It shows the linear blocks as generally the dominant approach. Square blocks dominate in a single point only, right at a drop of 3 pp from the baseline accuracy, which suggests that square blocks may become the dominant mechanism at larger accuracy drops. The sliding window is dominated in both respects: compared with linear blocks of similar size over the same τ_v sweep, it reaches a lower compression ratio and loses more accuracy at almost every τ_v (section A.2).

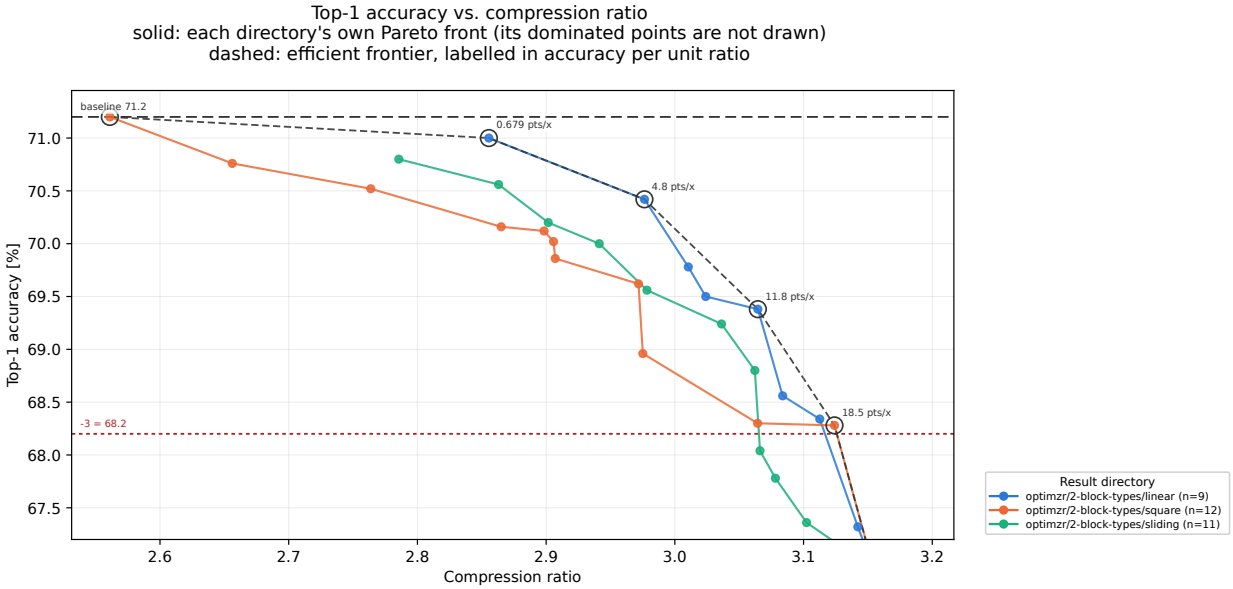


Figure 3.19: Orange points are results from an evaluation using square blocks; **Green** points stem from sliding window runs; and **Blue** points use a linear block error accounting mechanism. While the Pareto front is generally dominated by linear blocks, the square blocks dominate in just a single point right at the 3 percent point drop from the uncompressed, baseline accuracy. This points to the possibility that square error accounting starts being the dominant mechanism at higher accuracy drops

3.3.4 Hardware Cost

Value error threshold Bounding every value by τ_v is the cheapest part of the encoding decision. The absolute errors of the candidate predictions have to be computed anyway to choose the best prediction function, and the per-channel tolerance $\Delta_c^v = M_c\,\tau_v$ is a constant, fixed once the channel magnitude M_c has been profiled. VALUE-OK therefore costs a single comparison of $E_{c,i}$ per value.

Block error threshold Adding τ_b adds a second comparison per value, BLOCK-OK, against the per-channel constant Δ_c^b. On top of the comparison, it requires state: the running error sum S_B of the current block and the number k of values already committed to it, from which the mean error of the block is obtained by a division. The division can be avoided by comparing against the accumulated budget instead, $S_B + E_{c,i} \leq (k+1)\,\Delta_c^b$, whose right-hand side grows by Δ_c^b with every value and thus costs only an addition. Both thresholds are thus cheap; what differs between the block types is how much of this state has to be kept.

Linear blocks Linear blocks follow the arrival order of the values, so only a single block is open at any time. The accounting consists of one accumulator for S_B and one counter for k per channel, both reset at every block boundary. This makes linear blocks the cheapest of the three mechanisms.

Spatial blocks Square blocks span several rows, so a row of the featuremap passes through all blocks next to each other before any of them is complete. The accounting of the linear blocks is therefore needed once per subdivision of the horizontal axis, i. e. once per block that is processed in parallel, and the compressor has to switch between these accumulators as the stream crosses from one block into the next.

Sliding window An exact sliding window has to subtract the error of the value leaving the window at every step, so it has to remember the error of each of

the last N_{SW} values, i. e. keep a buffer of the size of the window per channel. This is how the sliding window was evaluated in Fig. 3.19. The buffer can, however, be avoided by an approximation: instead of subtracting the exact error that leaves the window, the accumulated error is reduced by the current average error of the window, $S \leftarrow S - S/N_{SW} + E_{c,i}$. This requires a single accumulator, as for linear blocks, plus a division by N_{SW}, which reduces to a shift for window sizes that are powers of two. The effect of this approximation on compression ratio and accuracy has not been evaluated.

3.3.5 Summary

The comparison of the block error types in Fig. 3.19 showed the linear block as the overall dominant accounting mechanism. For the remainder of this thesis, we will be using this one as the default block type for all further evaluations.

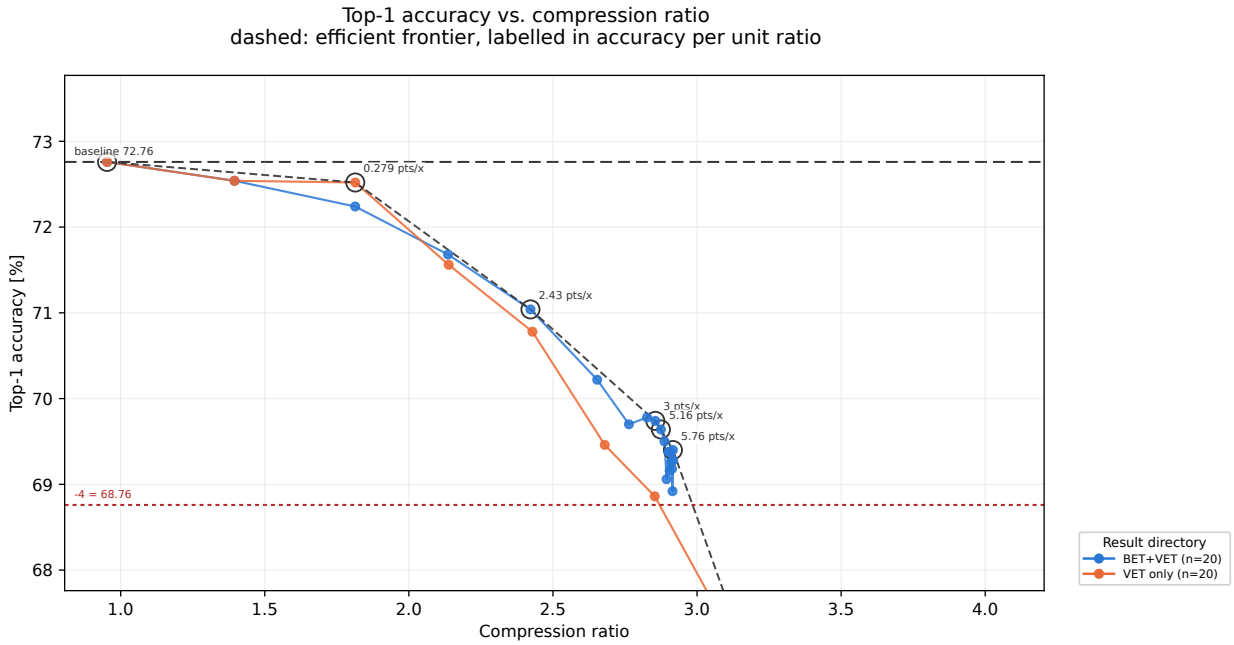


Figure 3.20: Top-1 accuracy against compression ratio for a τ_v sweep with block error thresholding (τ_b and τ_v, blue) and without it (τ_v only, orange). Dashed horizontal lines: baseline accuracy and a 4 pp budget; dashed diagonal: efficient frontier. With τ_b, increasing τ_v no longer drives the accuracy below the budget: the sweep settles at a compression ratio of about 2.9 and a drop of 3 to 3.7 pp.

Fig. 3.20 shows the compression ratio vs. accuracy Pareto front, which is lead by the design using a block error threshold, as opposed to one that only uses a per-value error bounding. This shows that the block error threshold can act as a soft accuracy drop limiter: with τ_b in place, increasing τ_v no longer drives the accuracy below the 4 pp budget, and the sweep settles at a compression ratio of about 2.9 and a drop of 3 to 3.7 pp. Tuning both τ_v and the block size allows to push the performance further.

Figure 3.21 shows the effect on the featuremap of Fig. 3.11. With linear blocks, the horizontal artifacts are reduced and the accuracy stays within 1 pp of the baseline, at a slightly lower compression ratio of 2.58:1. The average error is lower, since a prediction can only use its full τ_v once the block has accumulated enough budget. Limiting the average error within a featuremap thus appears to be an effective way to limit the accuracy loss, given an appropriate threshold.

It is, however, not a reliable one: the extreme cases tested in section A.3.2 lose different amounts of accuracy at the same average error.

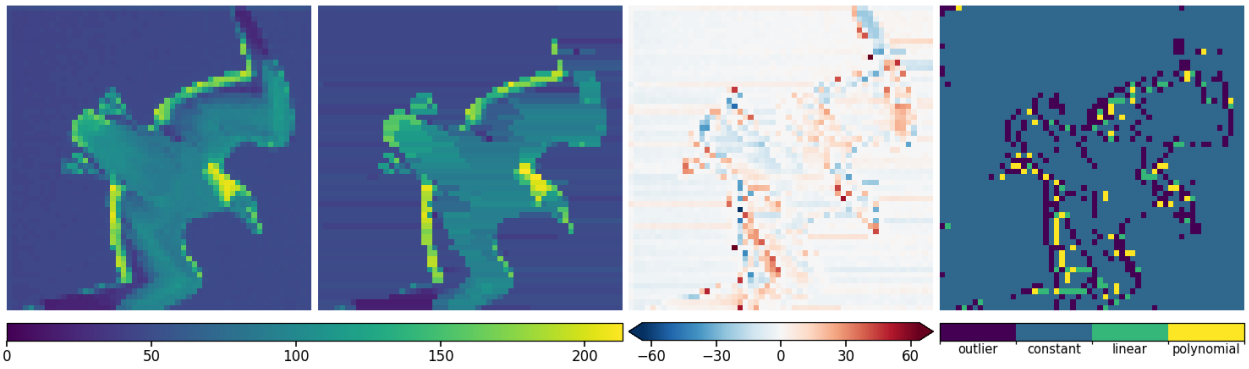


Figure 3.21: Compared to Fig. 3.11, the usage of a linear block budget for errors is able to decrease horizontal compression artifacts and improve the accuracy to be within one percent of the uncompressed baseline. Compression suffers slightly, dropping to 2.582:1. The average error is lower than in Fig. 3.11 since τ_b limits how often a prediction can actually get close to leveraging its full τ_v; budget has to accumulate first. Here it seems that limiting the average error within an approximated feature map can be an effective technique to limit the accuracy loss, one simply has to find the appropriate threshold first. However, testing the extreme cases in section A.3.2, we showed that limiting the average error is not a reliable way to limit the accuracy loss.

3.4 Extended Predictor Set

Fig. 3.7 has shown that the usage of prediction functions becomes increasingly one-sided with increasing τ_v values; while the decrease in the number of outliers is desirable, the uneven distribution among the other predictors points towards the predictors not working well for the data of activations.

Two obvious paths can be taken in this scenario. **(A)** We could double down on the one prediction function model and try to create one prediction function that works well in many different scenarios. This prediction function would likely be more sophisticated and computationally more expensive.

In scenario **(B)**, we would instead focus finding further prediction functions which are better suited for the particular data encountered in the activation, while trying to keep those function as computationally light as possible.

The SZ family of compressors, as well as further related work using prediction encoding [6] have adopted approach (A). We want to further explore scenario (B), as it has not been explored as extensively as (A); the prediction model with limited context requirements seems to be a good fit for streaming accelerators; and simpler prediction functions promise cheaper hardware implementations. A desirable property, since it leaves room for optimization of the rest of the accelerator.

The goal for this section is to introduce additional prediction functions, the first part focuses on functions that we call spatial predictors: They rely on a reconstructed two-dimensional view of the feature maps, which is realized through additional buffering of the one-dimensional stream in the compressor.

The second part explores a novel predictor design which leverages stochastic encoding, requiring no context at all, but rather only a prior profiling of the data distribution.

3.4.1 Notation

All prediction functions operate on the stream of a single channel c and produce the candidate $\hat{X}^{(f)}_{c,i}$ that enters the encoding decision of section 3.3. They read exclusively from the already reconstructed values $\hat{X}_{c,\cdot}$, so that compressor and decompressor compute the same candidates.

A channel's featuremap of height H and width W arrives in row-major order, one value per cycle. We write $u := i \bmod HW$ for the position within the current featuremap and

$$y := \lfloor u/W \rfloor, \qquad x := u \bmod W$$

for its row and column. The reconstructed neighbourhood of u is exposed as four causal *rays*, indexed by their distance $k \geq 1$ from the current position (Fig. 3.22):

$$r_W(k) := \hat{X}_{c,i-k} \quad \text{(west, the natural data flow)} \tag{3.8}$$

$$r_{NW}(k) := \hat{X}_{c,i-k(W+1)} \quad \text{(north-west diagonal)} \tag{3.9}$$

$$r_N(k) := \hat{X}_{c,i-kW} \quad \text{(north)} \tag{3.10}$$

$$r_{NE}(k) := \hat{X}_{c,i-k(W-1)} \quad \text{(north-east diagonal)} \tag{3.11}$$

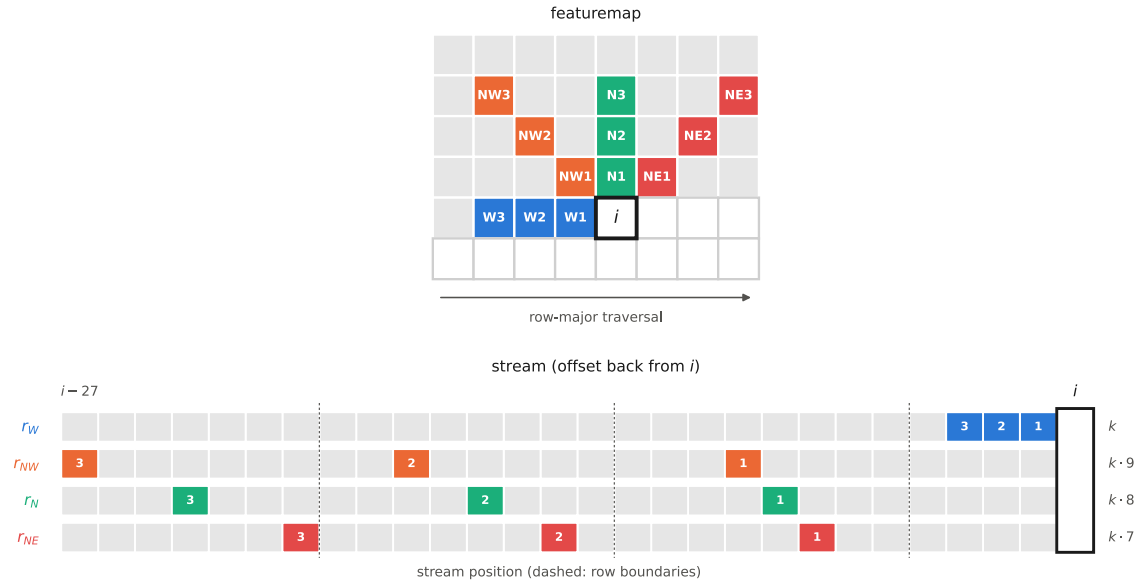


Figure 3.22: The four causal rays for a featuremap of width $W = 8$. Top: already reconstructed positions (grey), the current position i, and the first three values $k = 1, 2, 3$ of the west (W), north-west (NW), north (N) and north-east (NE) ray. Bottom: the same values in the one-dimensional stream, k, $k(W + 1)$, kW and $k(W - 1)$ positions before i; dashed lines mark row boundaries. All rays except W reach back at least one full row, which therefore has to be buffered.

The stream p of the baseline functions is the west ray, $p(k) = r_W(k)$. We write $D := (W, NW, N, NE)$ for this ordered tuple of directions and D_m for its m-th entry, counting from $D_0 = W$. All four rays are causal under the row-major traversal: r_W precedes the current position in its own row, the other three lie in rows that have been completed already.

The compressor buffers the last K rows of the featuremap, which gives the rays r_{NW}, r_N and r_{NE} one value per buffered row, and the last K_W values of the stream for r_W. We use $K_W = 3$, the depth the polynomial predictor of SZ1 requires.

These two bounds directly determine the buffering cost of a predictor. Apart from r_W, rays do not wrap around, so fewer values exist closer to the borders of the featuremap; we write a_d for the number of values ray d can supply at the current position:

$$a_W = \min(u,\ K_W), \qquad a_{NW} = \min(y,\ x,\ K), \tag{3.12}$$

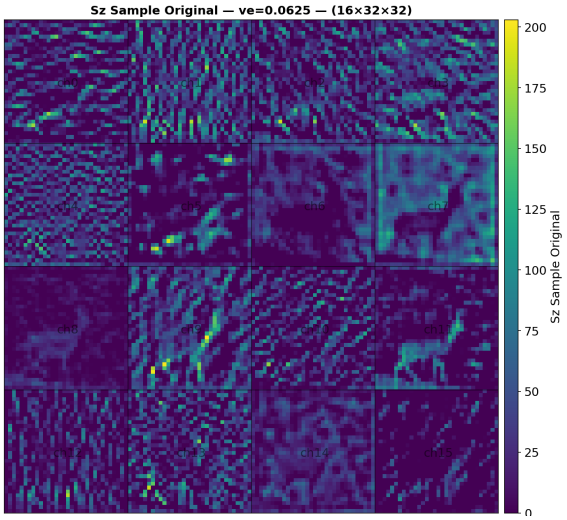
$$a_N = \min(y,\ K), \qquad a_{NE} = \min(y,\ W - 1 - x,\ K). \tag{3.13}$$

These availabilities are a function of the position only and never of the data, so encoder and decoder agree on which prediction functions are applicable at any position without transmitting side information. Note that r_W is read from the flattened stream rather than from the two-dimensional featuremap, so for $x < k$ it wraps around the row boundary into the tail of the preceding row — the same row-major linearisation SZ1 performs.

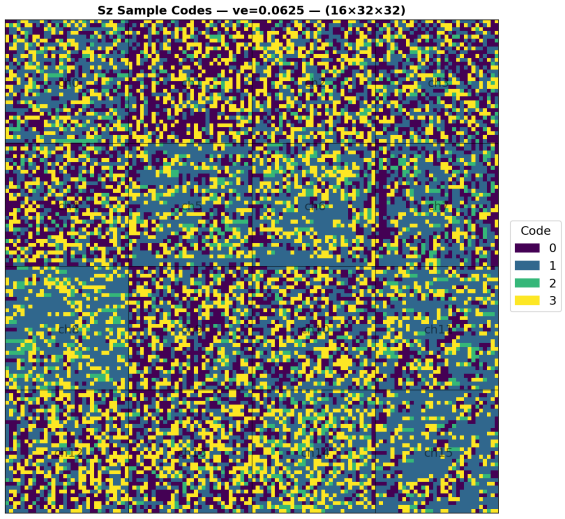
This expression of the two-dimensional featuremap in terms of the one-dimensional stream allows us to use two-dimensional prediction functions and express their requirements as a requirement vector $\nu = (\nu_0, \nu_1, \nu_2, \nu_3)$, given relative to D: slot j reads $r_{D_j}(1), \ldots, r_{D_j}(\nu_j)$, and the function is applicable if and only if $\nu_j \leq a_{D_j}$ for all j. Section 3.4.2.2 rotates this assignment of slots to directions to obtain the spatial variants of a function.

3.4.2 Spatial predictors

The motivation to use prediction functions that make use of the spatial dimensions of feature maps, rather than only properties of the one-dimensional value stream, is that feature maps are fundamentally two-dimensional [21] and treating them merely as a one-dimensional stream throws away correlations present between rows that could be used for more accurate predictions.



(a) This sample taken out of a skip connection shows the different channels laid out. Many of the different channels show small-scale gradients in different directions: The first channel shows horizontal gradients, the second vertical gradients, the third vertical gradients from top-left to bottom-right



(b) Codes choose for compression of (a): Channels with non-smooth progressions or many erratic changes of values along the horizontal axis have been dealt a lot more outliers (dark blue) in comparison to the other channels

The original SZ predictors are inherently one-dimensional and are intended to model smooth progressions of data often found in large scientific data sets. They are, however, decidedly bad at dealing with edges within a image and would typically make the compressor resort to an outlier instead.

Fig. 3.23a shows a cross section of a featuremap with different directions of gradients in their channels. While a gradient like the one in the first layer might be modeled relatively accurately by the default predictors, they fail in channels with more erratic changes across the linearized stream. Figure 3.23b shows the codes chosen for these channels: channels with non-smooth progressions or erratic changes along the horizontal axis receive considerably more outliers than the others. This hurts compression ratio.

We tackle this problem by firstly resorting back to PNG filters, which are widely used for more efficient storage of images in PNG and other formats. Then we present a way to apply to original SZ predictors along different spatial directors using our previously introduced notion of rays introduced in eq. (3.8).

Both of these - Observation: Directional Gradients in Featuremaps -> Show all-channels heatmap -> there seems to be a smooth transition the vertical axis rather than the horizontal axis in some of the featuremaps
- Default SZ predictors are very good at predicting smooth regions (See examples of sz works), but always default back to outliers if there is a clear break in the data stream

3.4.2.1 PNG Filters as Predictors

PNG [84] uses a number of filters to pre-process data per row (section 2.5.1.2). Filters are defined based on neighbors of a pixel in the two-dimensional image, that would—when doing row-major traversal—already been processed. Similar to how predictions work here, just with the difference, that the difference between predicted values and raw values is stored, such that the original image can be restored perfectly from the difference stream.

The filters we have picked from PNG are a 2 and 3-input average filter and the Paeth filter. PNG also offers a Sub and a Up filter, but these are equivalent to constant predictions from a different ray and are covered by the successive section already [84].

Definition Let $a := r_W(1)$, $b := r_N(1)$ and $d := r_{NW}(1)$ be the immediate left, top and top-left neighbours of the current position. The PNG filters then read as prediction functions as follows.

$$\hat{X}^{(\mathsf{avg2})}_{c,i} = \frac{a+b}{2}, \qquad \nu = (1,0,1,0) \tag{3.14}$$

$$\hat{X}^{(\mathsf{avg3})}_{c,i} = a + b - d, \qquad \nu = (1,1,1,0) \tag{3.15}$$

Equation (3.14) is the PNG *Average* filter, without the truncation PNG applies when averaging two bytes. Equation (3.15) is the gradient or *Paeth base* predictor; it coincides with the one-layer Lorenzo predictor of eq. (3.20), which is why we list it here rather than duplicating it as a third multilayer instance. Note that `avg2` skips the north-west ray entirely, so it remains applicable in the first column where `avg3` is already gated off.

The *Paeth* filter itself selects, among the three neighbours, the one closest to the gradient base $g := a + b - d$, with ties broken in the order a, b, d:

$$\hat{X}_{c,i}^{(\texttt{paeth})} = \begin{cases} a, & |g-a| \le |g-b| \ \wedge \ |g-a| \le |g-d|, \\ b, & \text{otherwise, if } |g-b| \le |g-d|, \\ d, & \text{otherwise,} \end{cases} \qquad \nu = (1,1,1,0) \tag{3.16}$$

Unlike every other function in our set, eq. (3.16) never extrapolates: its output is always one of the reconstructed neighbours it reads, which bounds its prediction to the range already present in the local neighbourhood. It is also cheap to implement, needing only additions, subtractions, absolute values and comparisons, but no multiplications.

3.4.2.2 Spatial variants of standard prediction functions

Rather than defining a separate function per direction, we keep the definitions of eqs. (3.4) to (3.6) and (3.14) to (3.16) and rotate the assignment of their slots to the directional rays. A *variant* is a rotation $\sigma \in \{0,1,2,3\}$, which maps slot j of a function to the direction

$$\delta_\sigma(j) := (\sigma + j) \bmod 4, \tag{3.17}$$

so that slot j reads $r_{D_{\delta_\sigma(j)}}$ instead of r_{D_j}. The applicability condition of section 3.4.1 rotates with it: the variant fits at the current position if and only if $\nu_j \le a_{D_{\delta_\sigma(j)}}$ for all j. The identity variant $\sigma = 0$ reproduces the definitions as given above.

Which variants are meaningful depends on the geometry a function assumes. The single-ray functions accept all four rotations, since the ray they extrapolate along can be any of the four directions — eq. (3.4) at $\sigma = 2$, for instance, is exactly the PNG *Up* filter, and eq. (3.5) at $\sigma = 2$ is a vertical linear extrapolation $2\,r_N(1) - r_N(2)$. The three-ray functions eqs. (3.14) to (3.16) assume a corner rather than a ray and therefore only admit the two rotations whose three consecutive directions are all causal, $\sigma \in \{0,1\}$: the variant $\sigma = 0$ anchors the corner at (W, NW, N), the variant $\sigma = 1$ at (NW, N, NE), which reflects the function about the vertical axis of the featuremap. The multilayer functions of eq. (3.19) and the stochastic function of eq. (3.22) admit the identity variant only.

3.4.2.3 Linear Regression Prediction

Motivated by Zhao et al. [96], we introduce several variants of regression prediction function. The functions $\texttt{linreg}J$, $J \in \{1,2,3,4\}$, extrapolate along J adjacent rays at once. Unlike the functions above they have no fixed requirement: every ray contributes as many values as it can supply at the current position. Let

$$d_j := D_{\delta_\sigma(j)}, \qquad m_j := a_{d_j}, \qquad j = 0, \dots, J-1,$$

be the ray read by slot j and its depth, and let ℓ_j be the least-squares line through the points $\big(-k,\ r_{d_j}(k)\big)$, $k = 1, \dots, m_j$, evaluated at the current position. For $m_j = 1$ the line degenerates to the constant $r_{d_j}(1)$, for $m_j = 2$ it is the linear extrapolation $2\,r_{d_j}(1) - r_{d_j}(2)$ (without the clamping of eq. (3.5)). The rays are combined weighted by their depth,

$$\hat{X}_{c,i}^{(\texttt{linreg}J)} = \frac{\sum_{j=0}^{J-1} m_j\, \ell_j}{\sum_{j=0}^{J-1} m_j}, \qquad \nu = (0,0,0,0), \tag{3.18}$$

so that a ray without values ($m_j = 0$) drops out, and the prediction is 0 if none of the J rays can supply a value. Since $\nu = (0, 0, 0, 0)$, the function is applicable at every position; its buffering cost is nonetheless that of reading up to K values from each of its rays, or K_W from r_W.

The variants are the rotations $\sigma \in \{0, \ldots, 4 - J\}$, for which the J rays are consecutive entries of D and do not wrap from NE back to W: four single rays for $J = 1$, the pairs (W, NW), (NW, N), (N, NE) for $J = 2$, the two corners of section 3.4.2.2 for $J = 3$ and all four rays for $J = 4$.

3.4.2.4 Lorenzo Predictor

We include the multilayer generalisation of the Lorenzo predictor introduced with SZ-1.4 [76]. It reads the complete $n \times n$ block above and to the left of the current position rather than individual rays, so we write

$$v(k_1, k_2) := \hat{X}_{c,\, i - k_1 W - k_2}$$

for the reconstructed value k_1 rows up and k_2 columns to the left; the rays are its special cases $r_W(k) = v(0, k)$, $r_N(k) = v(k, 0)$ and $r_{NW}(k) = v(k, k)$. The n-layer predictor is

$$\hat{X}_{c,i}^{(\texttt{layer}n)} = \sum_{\substack{0 \le k_1, k_2 \le n \\ (k_1, k_2) \neq (0,0)}} (-1)^{k_1 + k_2 + 1} \binom{n}{k_1} \binom{n}{k_2} v(k_1, k_2), \qquad \nu = (n, n, n, 0), \tag{3.19}$$

which we offer for $n \in \{1, 2, 3, 4\}$. The requirement $a_{NW} \ge n$ places the whole block inside the featuremap ($x \ge n$, $y \ge n$), so `layer`n needs $K \ge n$ buffered rows. For $n = 1$, eq. (3.19) is the original Lorenzo predictor,

$$\hat{X}_{c,i}^{(\texttt{layer1})} = v(0, 1) + v(1, 0) - v(1, 1) = a + b - d, \tag{3.20}$$

identical to eq. (3.15).

Close to the top and left borders, where `layer`n is not applicable, `layer_adaptive` uses as many layers as the position allows:

$$\hat{X}_{c,i}^{(\texttt{layer_adaptive})} = \hat{X}_{c,i}^{(\texttt{layer}n^\star)}, \qquad n^\star := \min(4,\, a_W,\, a_{NW},\, a_N), \qquad \nu = (0, 0, 0, 0). \tag{3.21}$$

For $n^\star = 0$ no row above is usable and it falls back to the west ray, using the first of eqs. (3.4) to (3.6) that a_W admits, and to 0 at the first position of a featuremap.

3.4.3 Stochastic Encoding

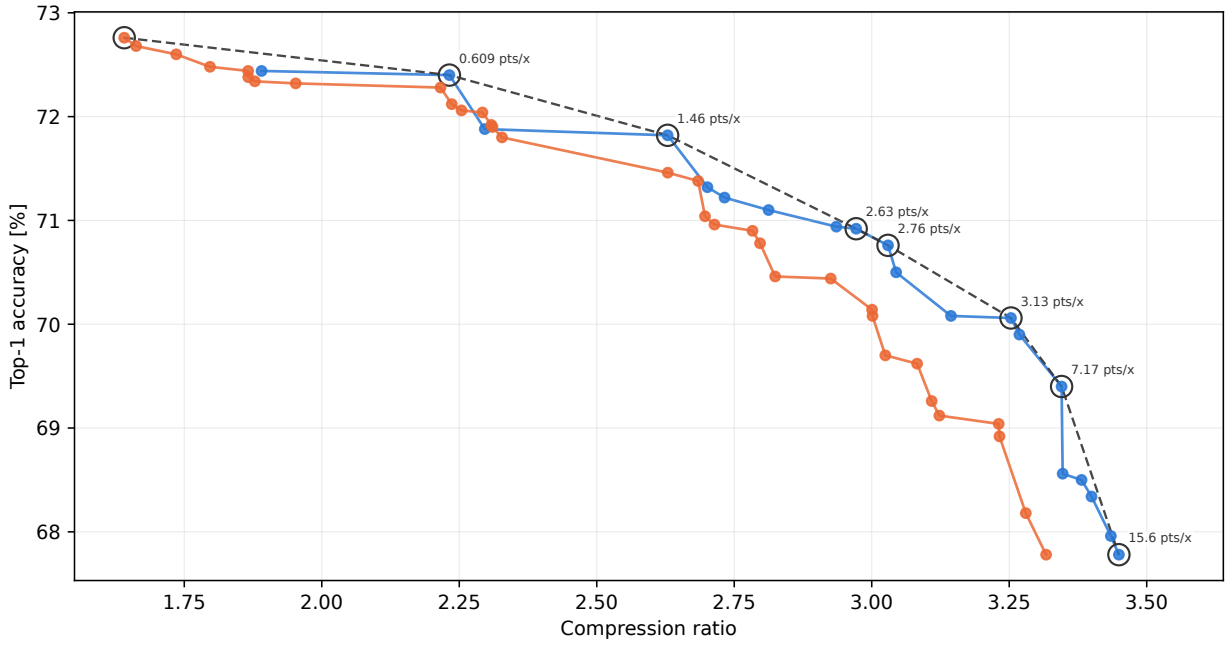


Figure 3.24: Pareto fronts of the parameter search with the predictor set constant, random, polynomial (blue) and of an exhaustive parameter sweep (orange); top-1 accuracy against compression ratio, dashed: common efficient frontier. Using stochastic encoding as a prediction function clearly dominates

Here we explore using a pseudo-random number generator (PRNG) as a prediction function. The idea is simple: We profile the distribution of a featuremap (or a channel) and create a skewed pseudo random number generator based on the profiled distribution. Both compressor and decompressor will then have a own instance of that number generator which produces the same sequence of random number in lockstep; once the value compressed using a PRNG predictor reaches the decompressor, the decompressor reconstruction machinery will have the same random number available. This requires the same seeding in both compressor and decompressor and that the sequence of random numbers generated based on that seed is deterministic.

The choice of the distribution goes back to an early experiment on the output of a post-ReLU block of ResNet-20 on CIFAR-10 (section A.5.1). There, the activations are dominated by zeros and small values and decay roughly exponentially towards large ones, and replacing the polynomial function of the SZ1 set by a predictor drawing from an exponential distribution kept the accuracy within about 2 pp of the baseline up to a ratio of 3.4, while the SZ1 set fell by more than 5 pp before reaching 2.9. This distribution is, however, a property of post-ReLU activations and does not hold for every connection. Concretely, we profile the per-channel mean μ_c of the activations and draw the prediction from an exponential distribution matched to it:

$$\hat{X}_{c,i}^{(\mathsf{random})} \sim \mathrm{Exp}(\lambda_c), \qquad \lambda_c := \frac{1}{\mu_c}, \qquad \nu = (0,0,0,0) \tag{3.22}$$

so that the drawn values have the profiled channel mean in expectation, $\mathbb{E}\left[\hat{X}_{c,i}^{(\mathsf{random})}\right] = \mu_c$. The draw is taken from a pseudo-random generator seeded deterministically with the stream position i, which is what keeps the decompressor in lockstep: it re-draws the identical value from the identical seed, so a code pointing at

this function is all that has to be stored. A degenerate channel with $\mu_c \leq 0$ has no valid rate; there the function falls back to predicting the constant μ_c, which is deterministic and therefore equally reproducible. Figure 3.25 illustrates the stream of random predictions next to the stream of raw values.

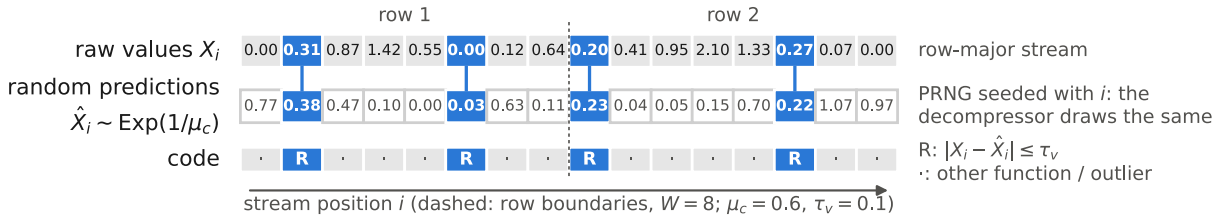


Figure 3.25: The random predictor on two rows of a featuremap of width $W = 8$, in row-major order (illustrative values, $\mu_c = 0.6$, $\tau_v = 0.1$). Top: the raw values X_i. Middle: the random predictions, one draw per position from $\mathrm{Exp}(1/\mu_c)$ by a pseudo-random generator seeded with the position i, so that the decompressor draws the same values. Bottom: the code of each position. Where a draw happens to lie within τ_v of the raw value (blue), the position is coded with the random predictor (R) at no further cost; all other positions (·) are left to the other prediction functions or become outliers. Dashed: row boundary.

Because $\nu = (0, 0, 0, 0)$, eq. (3.22) reads no reconstructed values at all. It is the only function in our set that needs neither buffering nor arithmetic on the data path, it is applicable at every position at which the warm-up of section 3.3 permits a prediction at all, and it is the only one whose prediction is not a function of the local neighbourhood — which is precisely what makes it a candidate for the positions at which the deterministic functions have collapsed into each other.

Figure 3.24 compares the Pareto front of the parameter search with the set constant, random, polynomial against the front of an exhaustive parameter sweep. The set containing the random predictor lies on or above the other front over the whole range.

The early trial on ResNet-20 showed the promise of an exponential distribution most clearly (section A.5.1). Whether the random predictor replaced the linear or the polynomial function, it took about a quarter of all codes and lowered the outlier share from about 10 to about 4 %, and the remaining deterministic function was used several times as often as in the SZ1 set. On the connection of this chapter, whose activations follow a different distribution, the gain is smaller; the appendix discusses both cases and why the random predictor helps at all.

3.4.4 Selected Prediction Function Combinations

Table 3.1 summarizes the extended predictor set. A concrete configuration selects a small number of (function, variant) pairs per channel, since the number of selectable pairs determines the width of the code alphabet and thus the size of the compressed stream.

Function	Definition	ν	Variants
`constant`	eq. (3.4)	$(1, 0, 0, 0)$	$\sigma \in \{0, 1, 2, 3\}$
`linear`	eq. (3.5)	$(2, 0, 0, 0)$	$\sigma \in \{0, 1, 2, 3\}$
`polynomial`	eq. (3.6)	$(3, 0, 0, 0)$	$\sigma \in \{0, 1, 2, 3\}$
`avg2`	eq. (3.14)	$(1, 0, 1, 0)$	$\sigma \in \{0, 1\}$
`avg3`	eq. (3.15)	$(1, 1, 1, 0)$	$\sigma \in \{0, 1\}$
`paeth`	eq. (3.16)	$(1, 1, 1, 0)$	$\sigma \in \{0, 1\}$
`linreg`J	eq. (3.18)	$(0, 0, 0, 0)$	$\sigma \in \{0, \ldots, 4 - J\}$
`layer`n	eq. (3.19)	$(n, n, n, 0)$	identity only
`layer_adaptive`	eq. (3.21)	$(0, 0, 0, 0)$	identity only
`random`	eq. (3.22)	$(0, 0, 0, 0)$	identity only

Table 3.1: The extended predictor set. ν gives the number of reconstructed values the function requires from each of the four directional rays of section 3.4.1, and the last column the spatial variants under which it is offered. `linreg`J and `layer_adaptive` require nothing and read as many values as are available. `avg3` and `layer1` define the same function.

To see how individual functions contribute, we replace one function of the SZ1 set at a time and repeat the τ_ν sweep of section 3.2.

- **`avg2` instead of `polynomial`.** The set keeps about 68 % top-1 at a ratio of 3.6, where the SZ1 set has dropped to about 61 % (Fig. 3.26). `avg2` takes about a quarter to a third of the codes, the linear predictor, which the SZ1 set barely uses, about a sixth, and the outlier share drops faster than with the SZ1 set (Fig. 3.30).

- **Lorenzo instead of linear.** The set gains about one to two points of accuracy at equal ratio (Fig. 3.27), although the Lorenzo predictor is chosen for only a few percent of the codes; for larger τ_ν, both sets are dominated by the constant predictor (Fig. 3.31).

- **Random instead of polynomial.** Up to a ratio of about 2.5, both sets coincide; beyond, the set with the random predictor loses far less accuracy and keeps about 63 % top-1 near the 4:1 limit (Fig. 3.28). The random predictor takes around 8 % of the codes from $\tau_\nu \approx 0.1$ on, while the outlier shares of both sets stay almost identical (Fig. 3.32). At equal outlier shares, the compression ratios at equal τ_ν are the same, so the gain lies entirely in accuracy.

- **Random instead of linear.** The set stays above the SZ1 set over the whole range and keeps about 65 % top-1 near the 4:1 limit (Fig. 3.29). Random and polynomial predictor together take around 12 % of the codes for larger τ_ν, at outlier shares close to those of the SZ1 set (Fig. 3.33).

Each replacement improves on the SZ1 set, most of all at high compression ratios.

As an extreme case, replacing all three functions by the random predictor falls far behind the SZ1 set: the random predictor is useful only as a complement to deterministic functions, not as a replacement for them (section A.5).

The most interesting of those case is clearly the first one in which the polynomial predictor is replaced with the average predictor. The high share of the average predictor (Fig. 3.30) shows that the average predictor is a good match for the data. But remarkably, the share of linear predictions also increases. This is a result of

the average function creating predictions that actually allow the linear function to provide value to the predictor set and points towards the interactions between functions playing an important role in the performance of the function set. The effect is also slightly visible towards the end of the distribution sweep in Fig. 3.30 where the share of linear and average predictions actually starts to decrease again, as constant predictions become more dominant.

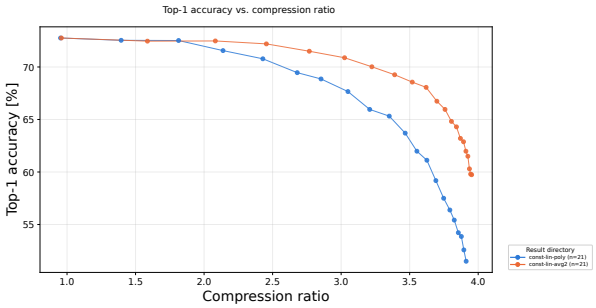


Figure 3.26: Constant, linear, avg2 (orange) vs. the SZ1 set constant, linear, polynomial (blue): top-1 accuracy against compression ratio over the same τ_v sweep. Replacing the polynomial by the two-input average keeps about 68 % top-1 at a ratio of 3.6, where the SZ1 set has dropped to about 61 %.

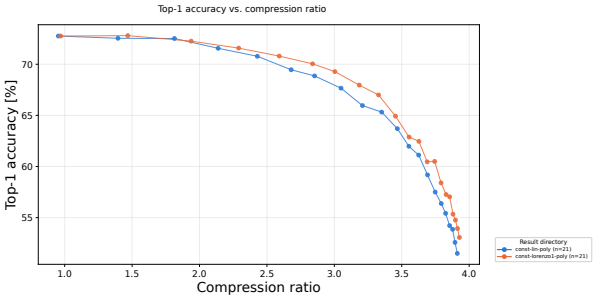


Figure 3.27: Constant, Lorenzo (one layer), polynomial (orange) vs. the SZ1 set (blue): top-1 accuracy against compression ratio over the same τ_v sweep. Replacing the linear by the Lorenzo predictor gains about one to two points of accuracy at equal ratio.

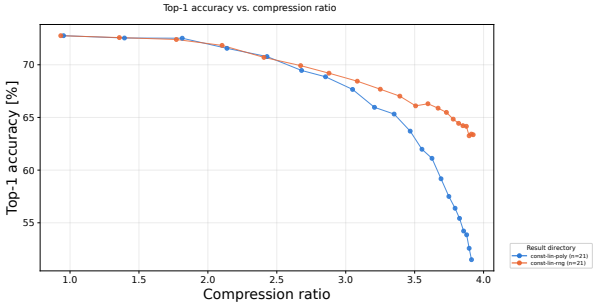


Figure 3.28: Constant, linear, random (orange) vs. the SZ1 set (blue): top-1 accuracy against compression ratio over the same τ_v sweep. Up to a ratio of about 2.5 the two coincide; beyond, the set with the random predictor loses far less accuracy and still keeps about 63 % top-1 near the 4:1 limit.

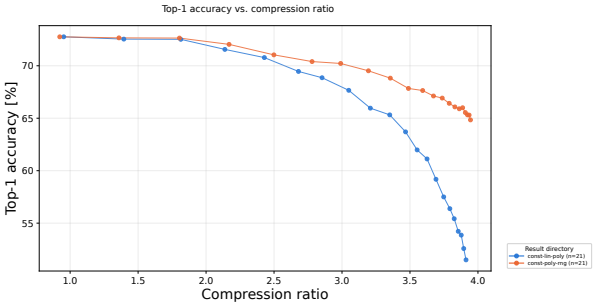


Figure 3.29: Constant, polynomial, random (orange) vs. the SZ1 set (blue): top-1 accuracy against compression ratio over the same τ_v sweep. The set with the random predictor stays above the SZ1 set over the whole range and keeps about 65 % top-1 near the 4:1 limit.

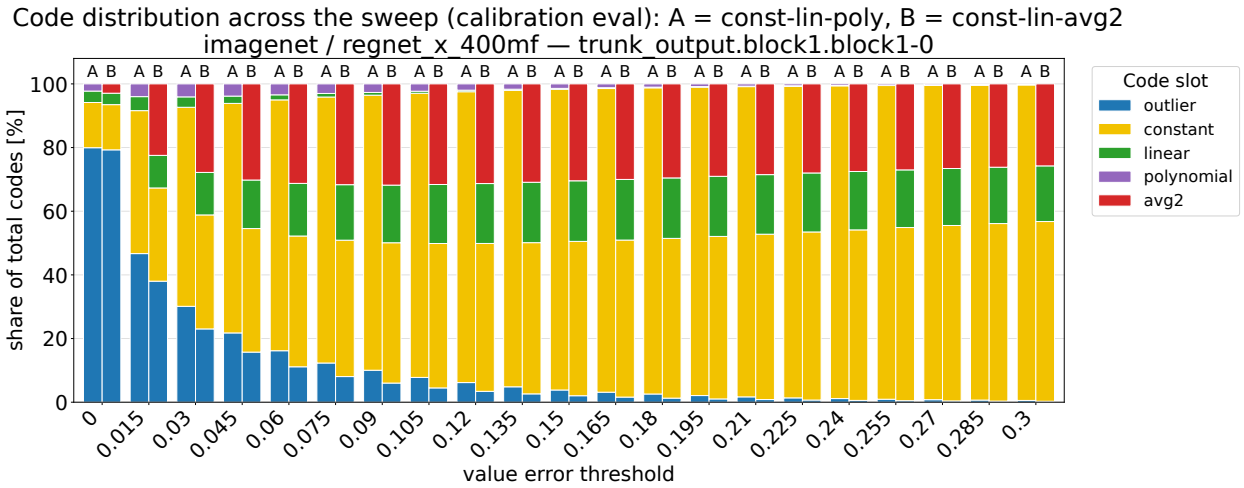


Figure 3.30: Code distribution of the SZ1 set (A) and of constant, linear, avg2 (B) for every τ_v of the sweep, pooled over all channels and batches: share of outliers and of each predictor among all codes. In B, avg2 takes about a quarter to a third of the codes and the linear predictor, which A barely uses, about a sixth, while the outlier share drops faster than in A.

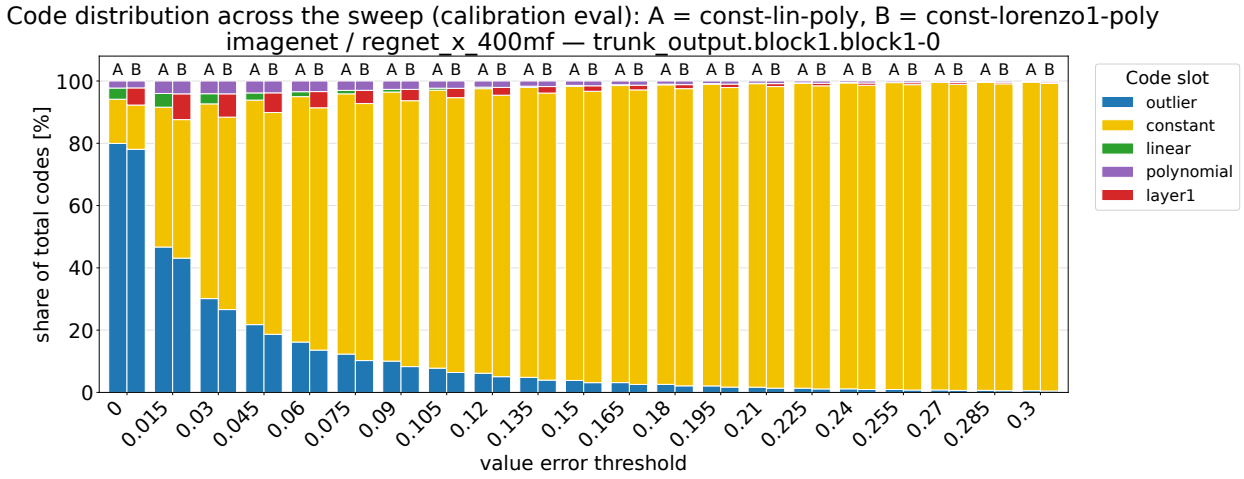


Figure 3.31: Code distribution of the SZ1 set (A) and of constant, Lorenzo, polynomial (B), as in Fig. 3.30. The Lorenzo predictor is chosen for only a few percent of the codes; for larger τ_v, both sets are dominated by the constant predictor.

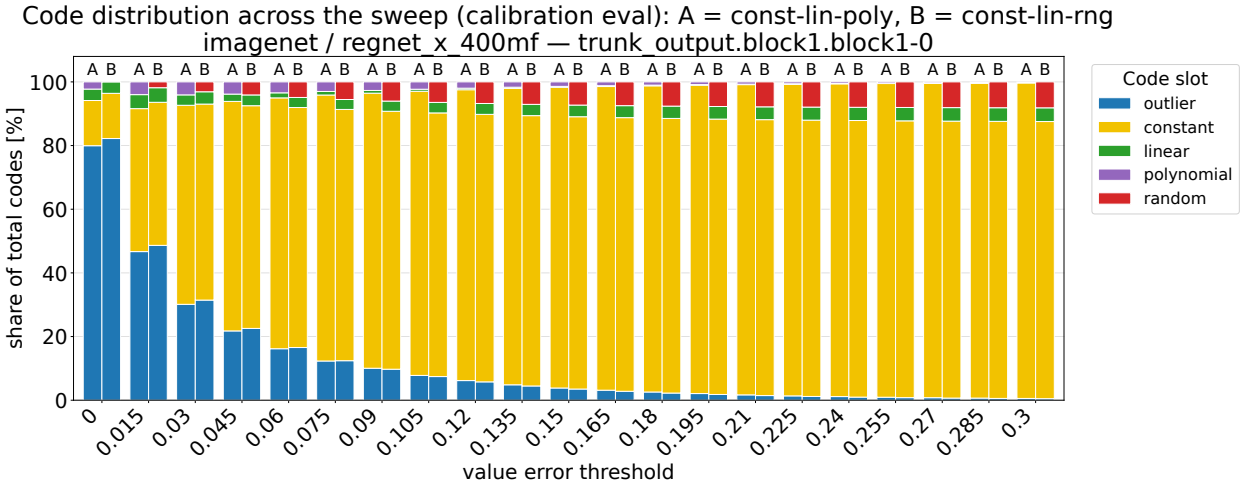


Figure 3.32: Code distribution of the SZ1 set (A) and of constant, linear, random (B), as in Fig. 3.30. The random predictor takes a share of around 8 % of the codes from $\tau_v \approx 0.1$ on, while the outlier shares of both sets stay almost identical.

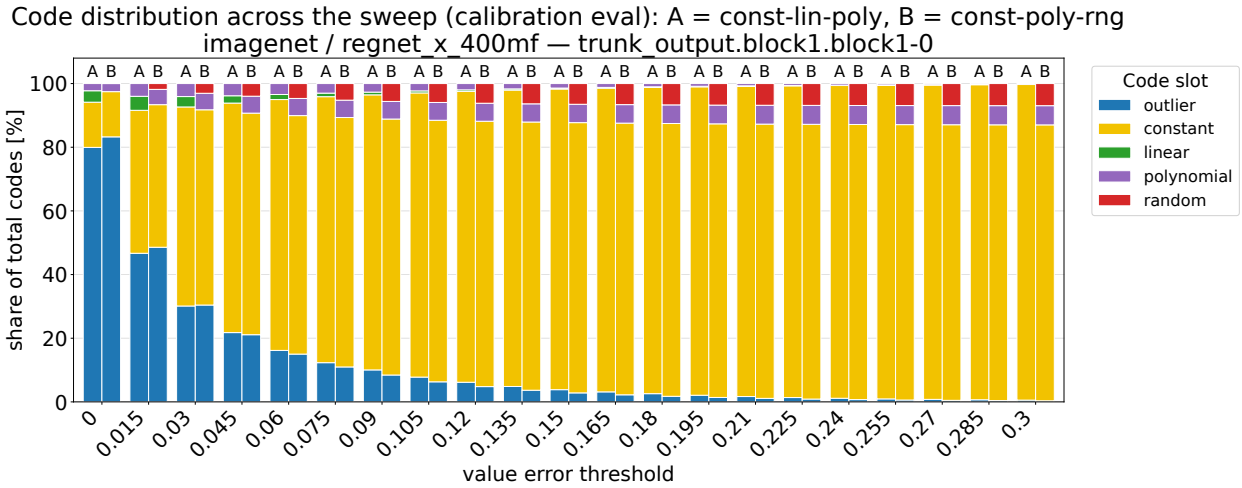


Figure 3.33: Code distribution of the SZ1 set (A) and of constant, polynomial, random (B), as in Fig. 3.30. Random and polynomial predictor together take around 12 % of the codes for larger τ_v, at outlier shares close to those of A.

3.4.5 Hardware Cost

We estimate the cost of each prediction function along two axes: the arithmetic it performs per prediction and the reconstructed values it requires the compressor to buffer. Both are given per channel; in the fully unrolled design of section 3.1.1, they are replicated C times per evicted connection.

Arithmetic We count every addition, subtraction, multiplication, division, absolute value and comparison (including the max of `clamp`) as one floating point operation (FLOP). Multiplications by constants are counted as well, although in a fixed-point implementation the factors 2 and $\frac{1}{2}$ reduce to shifts. Each function of the active set is evaluated in every cycle, so the arithmetic of the active functions adds up. On top of this, the encoding decision of section 3.3 costs one subtraction and one absolute value per active function, and $|F| - 1$ comparisons for the arg min over the $|F|$ active functions.

Buffering Ray d reads its k-th value from s_d k positions back in the stream, with the strides $s_W = 1$, $s_{NW} = W + 1$, $s_N = W$ and $s_{NE} = W - 1$ (eqs. (3.8) to (3.11)). Since the featuremap arrives as a single stream, the buffer a function requires is a delay line reaching back to the oldest value it reads,

$$B := \max_j \ \nu_j \, s_{D_{\delta_\sigma(j)}}, \tag{3.23}$$

counted in reconstructed values. Unlike the arithmetic, buffers do not add up: all active functions of a channel read from the same delay line, whose length is the maximum B over the active set. Any function that reads above the current row therefore costs on the order of a full row W per channel, while the original, west-only SZ1 functions get by with at most three values.

Function	+/−	×/÷	other	FLOPs	Buffer B
`constant`	0	0	0	0	s_d
`linear`	1	1	1	3	$2\,s_d$
`polynomial`	2	1	1	4	$3\,s_d$
`avg2`	1	1	0	2	W ($\sigma = 0$), $W + 1$ ($\sigma = 1$)
`avg3`	2	0	0	2	$W + 1$
`paeth`	5	0	6	11	$W + 1$
`linreg`J	$M - 1$	$M + J + 1$	0	$2M + J$	$\max_j m_j \, s_{d_j}$
`layer`n	$(n+1)^2 - 2$	$(n+1)^2 - 4$	0	$2(n+1)^2 - 6$	$n\,(W + 1)$
`layer_adaptive`	23	24	0	47	$4\,(W + 1)$
`random`	0	1	1	2	0

Table 3.2: Arithmetic per prediction and buffering per channel of the extended predictor set. *Other* counts absolute values, comparisons and the logarithm of `random`. For the single-ray functions, $d = D_\sigma$ is the ray selected by the variant. For `linreg`J, $M := \sum_j m_j$ is the number of values read at the current position, bounded by $m_j = K_W$ for the west ray and $m_j = K$ for the others. W is the featuremap width

Table 3.2 lists the resulting costs. The individual entries follow from the definitions as follows.

- **Baseline functions.** `constant` is a plain register read. `linear` computes $2\,p(1) - p(2)$ and the clamp, `polynomial` evaluates $3\,(p(1) - p(2)) + p(3)$ and the clamp. Rotating them onto another ray does not change their arithmetic, but scales their buffer by the stride s_d: the vertical variant of `polynomial` already needs $3W$ values instead of 3.

- **PNG filters.** `avg2` and `avg3` need two operations each. `paeth` computes the gradient base g (two operations), the three distances $|g - a|$, $|g - b|$, $|g - d|$ (three subtractions, three absolute values) and the three comparisons that select the output; it is the most expensive of the fixed-size functions but, as noted above, needs no multiplier.

- **`linreg`J.** For a fixed depth m, the least-squares line evaluated at the current position is a fixed linear combination of the m values on the ray, whose coefficients depend on m only and can be tabulated for $m \le K$. Each ray thus costs m_j multiplications and $m_j - 1$ additions, the depth-weighted combination of eq. (3.18) another J multiplications, $J - 1$ additions and one division. Since the depths m_j depend on the position only, the weights $m_j / \sum_j m_j$ could also be folded into the coefficient tables, which removes the combination and the division at the price of larger tables. With K rows buffered, `linreg4` is the most expensive function of the set.

- **`layer`n.** Equation (3.19) sums $(n+1)^2 - 1$ terms with constant coefficients; the three corner terms, $(k_1, k_2) \in \{(0,n), (n,0), (n,n)\}$, have coefficient ± 1 and need no multiplication. This gives 2, 12, 26 and 44 FLOPs for $n = 1, \ldots, 4$, so the cost grows quadratically with the number of layers. These counts, like those of `layer_adaptive`, are derived from the multilayer predictor as formulated for SZ-1.4 [76], not from a hardware implementation.

- **`layer_adaptive`.** Instantiating all four `layer`n datapaths separately would cost their sum. Instead, a single 5×5 datapath can evaluate any `layer`n with $n \leq 4$, given position-dependent coefficients (zero outside the $n \times n$ block) from a small table. Since these coefficients are no longer constant, the three corner terms of `layer4` need multipliers as well, so the datapath costs 24 multiplications and 23 additions, i. e. 47 FLOPs, three more than `layer4` alone. Its buffer is that of `layer4`, irrespective of how many layers are used at the current position.

- **`random`.** Equation (3.22) reads no reconstructed values. Drawing from $\mathrm{Exp}(\lambda_c)$ by inversion, $-\mu_c \ln U$, costs one logarithm and one multiplication, or a lookup table in place of the logarithm; the uniform U is produced by a pseudo-random generator seeded with the stream position, e. g. a linear feedback shift register (LFSR) [63], which consists of a shift register and a few XOR gates and therefore costs logic but no FLOPs. The only stored state is the profiled mean μ_c.

Cost with a single buffered row The final design uses a context of a single buffered row ($K = 1$, section 3.6.2). With it, the north-west, north and north-east rays supply at most one value each, while the west ray keeps its K_W values. Table 3.3 lists the (function, variant) pairs of table 3.1 that remain applicable under this restriction.

Function	Variants for $K = 1$	Remark
`constant`	$\sigma \in \{0, 1, 2, 3\}$	
`linear`	$\sigma = 0$	west ray only
`polynomial`	$\sigma = 0$	west ray only
`avg2`	$\sigma \in \{0, 1\}$	
`avg3`	$\sigma \in \{0, 1\}$	
`paeth`	$\sigma \in \{0, 1\}$	
`linreg`J	$\sigma \in \{0, \ldots, 4 - J\}$	$m_j \leq 1$ on all rays but west
`layer1`	identity	
`layer_adaptive`	identity	reduces to `layer1`
`random`	identity	

Table 3.3: (Function, variant) pairs of table 3.1 that remain applicable with a single buffered row ($K = 1$). `layer`n with $n \geq 2$ is not applicable. Every vertical or diagonal ray supplies only one value, so `linear` and `polynomial` keep only their west variant, `linreg1` degenerates to `constant` for $\sigma \neq 0$, and `layer_adaptive` uses at most one layer.

This restriction also bounds the arithmetic of a channel. With $K = 1$, every ray except the west one contributes at most one value, so a `linreg`J variant reads at most $M = K_W + (J - 1) = J + 2$ values if it includes the west ray, and $M = J$ otherwise. With the FLOP counts of table 3.2, `layer_adaptive`, which uses at most one layer, costs only the 2 of `layer1`, and the three most expensive pairs are

$$\begin{aligned} \texttt{linreg4},\ \sigma = 0: &\quad 2\,(3+3) + 4 = 16, \\ \texttt{linreg3},\ \sigma = 0: &\quad 2\,(3+2) + 3 = 13, \\ \texttt{paeth}: &\quad 11, \end{aligned}$$

followed by `linreg2` at $\sigma = 0$ with 10 FLOPs. An active set of these three costs at most 16 + 13 + 11 = 40 FLOPs per value, compared to 0 + 3 + 4 = 7 for the SZ1 set of constant, linear and polynomial. For either set, the encoding decision of section 3.4.5 adds a flat 8 FLOPs for its three active functions.

This is the upper limit on computation cost at $K = 1$. The average case is probably much lower as the smooth featuremaps favor constant predictors, which come free of charge.

3.5 Dynamic function selection

The previous sections have shown that there is no single best set of prediction functions. Replacing a single function of the SZ1 set changes how often the remaining functions are chosen and how many outliers remain (section 3.4): a function can have an unexpected effect on the efficacy of the others. In addition, the channels of a connection can look very different from each other (Fig. 3.23a), so a set that fits one channel well may fit another one poorly. This raises the central question of this section: how to choose the right set of prediction functions for every channel.

Modern SZ versions face a similar question. They choose the single best-fitting predictor, e. g. Lorenzo or linear regression, for every block of data from a sampled error estimate [14, 49, 50, 96]. Our setting differs in three respects. First, we select a *set* of three prediction functions per channel instead of a single one per block; within this set, the SZ1-style selection of the best-fitting function still happens for every value. Second, the selected functions only change the predictor that decorrelates the stream, rather than replacing the whole compression model of a block. Third, SZ compressors can try candidate schemes on the data before committing to one. A streaming compressor does not have that luxury: it has to make good decisions locally, with the help of prior profiling.

We therefore propose a dynamic function selection scheme, which adapts the active prediction functions of every channel from batch to batch. It can be used in two ways. Online, it adapts the selection during the operation of the system, for example when no profiling data is available. Offline, it runs adaptively on a calibration set, from which the best selection is extracted and then used statically during operation. The latter relies on a representative calibration set: on an unrepresentative one, the scheme may select functions that are a poor fit for a channel. The error thresholds still limit the accuracy loss in this case, but the compression ratio could drop considerably, which hurts the performance of the system and can even lead to pipeline stalls if the bandwidth budget is tight.

3.5.1 Selection Algorithm Description

Terminology The *pool* consists of every prediction function in every admissible variant, i. e. of all (function, variant) pairs of table 3.1, about 19 in total. Every channel has an ordered *active set* of at most three pairs, its slots. The code 0 marks an outlier and the codes 1 to 3 the slots, so the codes stay 2 bits wide. The remaining pool members form the *passive set* of the channel. We call the active sets of all channels of a connection its *configuration*.

Granularity The active set is decided per channel, once per batch, from the statistics of the previous batch, and stays constant within a batch. The first batch

has no statistics yet; it starts from a seed configuration if one is given, and otherwise from the base function set, e. g. constant, linear and polynomial, in their identity variant. Only the compressor gathers statistics, since it requires the raw values and the thresholds. The decompressor merely replays the configuration of every batch and never derives it itself. It can not do that in any case, since the swap decision requires collecting statistics that need information of the raw value stream.

Objective Since every predicted value costs the same 2-bit code, regardless of which slot it uses, the number of outliers is the only lever the configuration has on the compressed size (eq. (3.1)). Choosing the active set is therefore a maximum-coverage problem: every pool member *covers* the positions at which its prediction lies within τ_v, and the active set should leave as few positions uncovered as possible. As coverage is submodular, a greedy procedure that adds and drops one function at a time is a cheap and good fit [62]. Simply activating the three functions with the lowest mean error, in contrast, would be misleading: since only the best active prediction is used at every position, a function only matters at the positions where the other active functions fail.

Statistics During a batch, the compressor evaluates every pool member at every position and gathers two counts per channel. The *rescue count* of a passive pool member counts the positions at which the active set produced an outlier, but the pool member would have been within τ_v; it measures the benefit of adding the member. The *sole-cover count* of an active slot counts the positions at which exactly this slot was within τ_v; it measures the cost of removing the slot. In addition, the compressor tracks the mean absolute error of every pool member per channel, which only serves to break ties; a variant that was never applicable has an infinite error. All statistics are reset at the start of every batch.

Swap rule At the start of the next batch, every channel performs at most one swap:

1. The *add candidate* is the passive pool member with the highest rescue count, with ties broken by the lower mean absolute error. Its rescue count is the benefit of the swap.

2. The *drop candidate* is the active slot with the lowest sole-cover count. Its sole-cover count is the loss of the swap; if a slot is still free, nothing is dropped and the loss is 0.

3. The swap is applied if and only if the benefit exceeds the loss by more than a margin. The margin prevents oscillation between two configurations; by default it is 0, so that every strict improvement is applied. Otherwise, the channel keeps its active set.

This is a greedy, marginal-improvement heuristic: it climbs towards fewer outliers one function at a time. Allowing only a single swap per batch keeps the new configuration close to the one the statistics were measured on, and the configurations converge over successive batches. A simpler mode of the scheme keeps the functions of every channel fixed and only adapts their variants, choosing for every function the variant with the lowest mean absolute error in the previous batch. Figure 3.34 summarizes the scheme for a single channel.

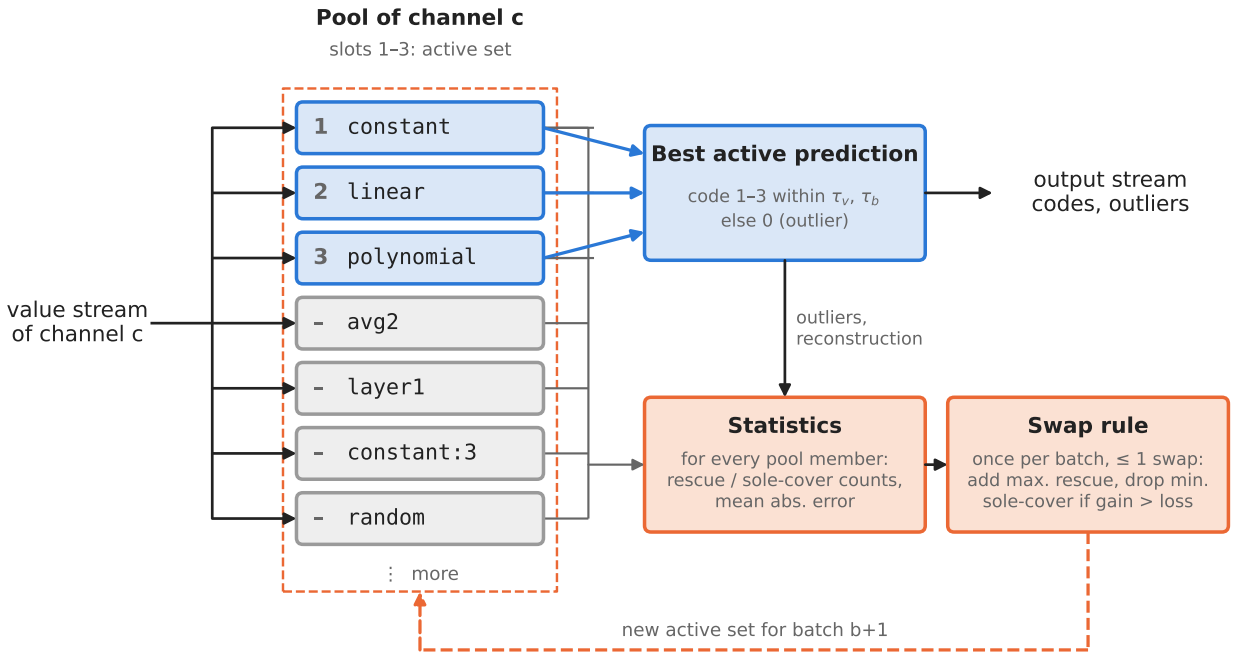


Figure 3.34: Dynamic function selection for a single channel c. Every value of the channel's stream is predicted by all members of the pool (about 19 (function, variant) pairs). Only the active set in slots 1 to 3 **(blue)** determines the output: the best active prediction within the error thresholds is encoded as its slot code, otherwise the value is stored as an outlier with code 0. The passive pool members **(grey)** are evaluated alongside. From the outliers and the reconstruction, the compressor gathers rescue and sole-cover counts and the mean absolute error of every pool member. At the end of each batch, the swap rule exchanges at most one active slot for a passive member if the rescue count of the added member exceeds the sole-cover count of the dropped slot, which yields the active set for the next batch **(orange feedback loop)**.

Harvesting the best configuration Throughout the run, the scheme records the configuration of the batch with the highest compression ratio, which, at a constant size of the transmitted configuration, is the batch with the fewest outliers. This configuration can be reused in two ways: as a *fixed* configuration, which is used for all batches with adaptation turned off, or as a *seed*, from which the adaptation continues. The static configurations in the remainder of this chapter are fixed configurations harvested from an adaptive run on the calibration set.

Cost Under online adaptation, the configuration of every batch has to reach the decompressor. It costs $C \cdot 3 \cdot \lceil \log_2 N_{\text{pool}} \rceil$ bits per batch for a pool of N_{pool} members, which we add to the compressed size. A fixed configuration, in contrast, is known statically. Online adaptation also requires evaluating the whole pool at every position, so that in hardware, all pool members would have to be instantiated in every channel compressor, whereas a fixed configuration only requires its three active functions.

Limitations The statistics are estimates of the true effect of a swap. Coverage only considers τ_v, while the block error threshold (section 3.3) can still force outliers that the heuristic does not model. The benefit and loss of a swap are measured on the reconstruction of the previous batch, i. e. with the old active set. Redundancy between active functions is accounted for this way, but not the fact that a swap changes the reconstructed values that all later predictions build on. Finally, near the borders of a featuremap, a fallback variant may be used in place of a slot's function, while its sole-cover count is still credited to the slot; this affects only a small share of the positions.

Figure 3.35 shows channel 27 compressed with a per-channel selection of the constant predictor along two rays and the random predictor. The error concentrates on the edges of the structure, where the constant predictors break down, while smooth regions are covered by long stretches of a single code.

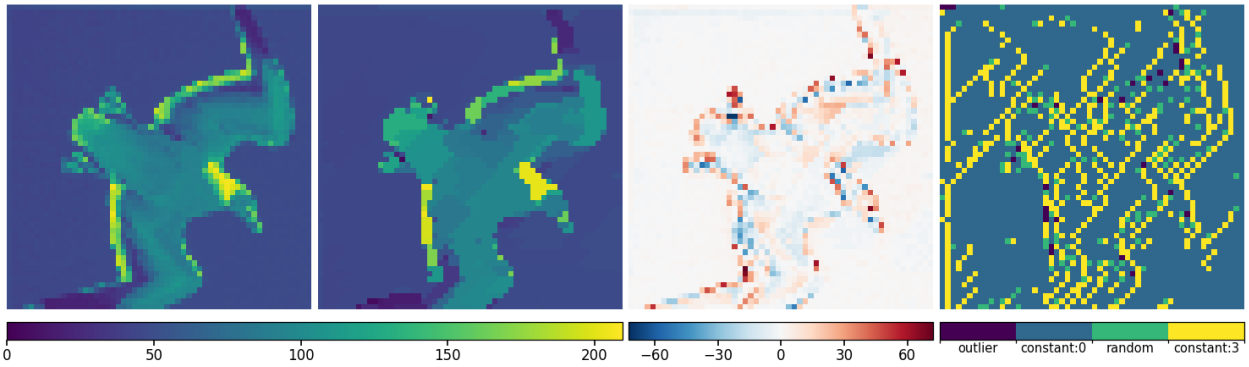


Figure 3.35: Channel 27, compressed with a per-channel selection of the constant predictor along two rays (`constant:0`, `constant:3`) and the random predictor ($\tau_v = 0.288$, $\tau_b = 0.082$). From left to right: original, reconstruction, error (reconstruction minus original) and codes. The error concentrates on the edges of the structure, where the constant predictors break down; smooth regions are covered by long stretches of a single code.

3.5.2 Selection Results

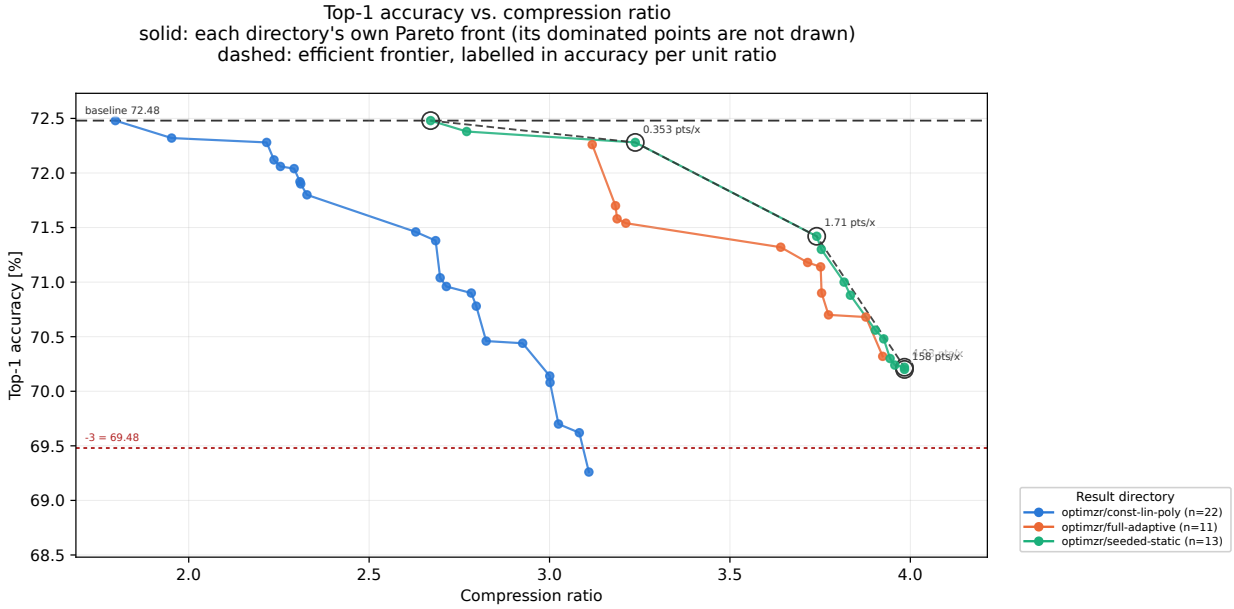


Figure 3.36: SZ predictor using the baseline prediction functions **(Blue)** vs. the fully dynamic prediction function selection scheme **(Orange)** vs. the best configuration of prediction function taken from the dynamic run and taken as a static configuration **(Green)**; Both the dynamic and the static-profiled runs out-perform the baseline configuration. The static run dominates the Pareto front, showing that the adaptive selection scheme does settle for suboptimal configurations

Figure 3.36 compares the SZ1 set with the fully adaptive scheme and with the best configuration of an adaptive run, used as a fixed configuration. Both the adaptive and the fixed configuration outperform the SZ1 set, but the fixed configuration dominates the Pareto front. The adaptive scheme thus finds a very good configuration at some point during the run, but does not retain it and replaces it by worse ones.

If the marginal improvement measured on one batch carried over to the next, the compression ratio would increase with every swap. The swap rule, however, measures benefit and loss on the reconstruction of the previous configuration. Every swap changes the reconstructed values that all later predictions build on, and thereby the behaviour of the functions that remain active. This interaction is visible in Fig. 3.30: replacing only the polynomial predictor of the SZ1 set by avg2 changes how the remaining functions are used, with the linear predictor being chosen for far more values, and the outlier share falling faster. A swap that appears beneficial on the statistics of one batch can therefore turn out worse once it is applied.

Another possible explanation is that the class composition differs between batches: one batch informs the configuration of the next, and with a batch size of 512 and 1000 classes, two successive batches can consist of almost completely disjunct sets of classes. Figure 3.37 shows the compression ratio per batch for the adaptive selection in a separate run on the 45000 images of the test set, sorted by class and split into batches of 11 images, i. e. about four batches per class. Every class change is thus a change in the batch composition. The ratio is lowest in the first batches, while the selection adapts from its initial configuration, and stays between 3.4 and 3.75 afterwards, also across class changes. Changes in the class composition between batches thus do not appear to be the reason why the adaptive scheme fails to retain good configurations.

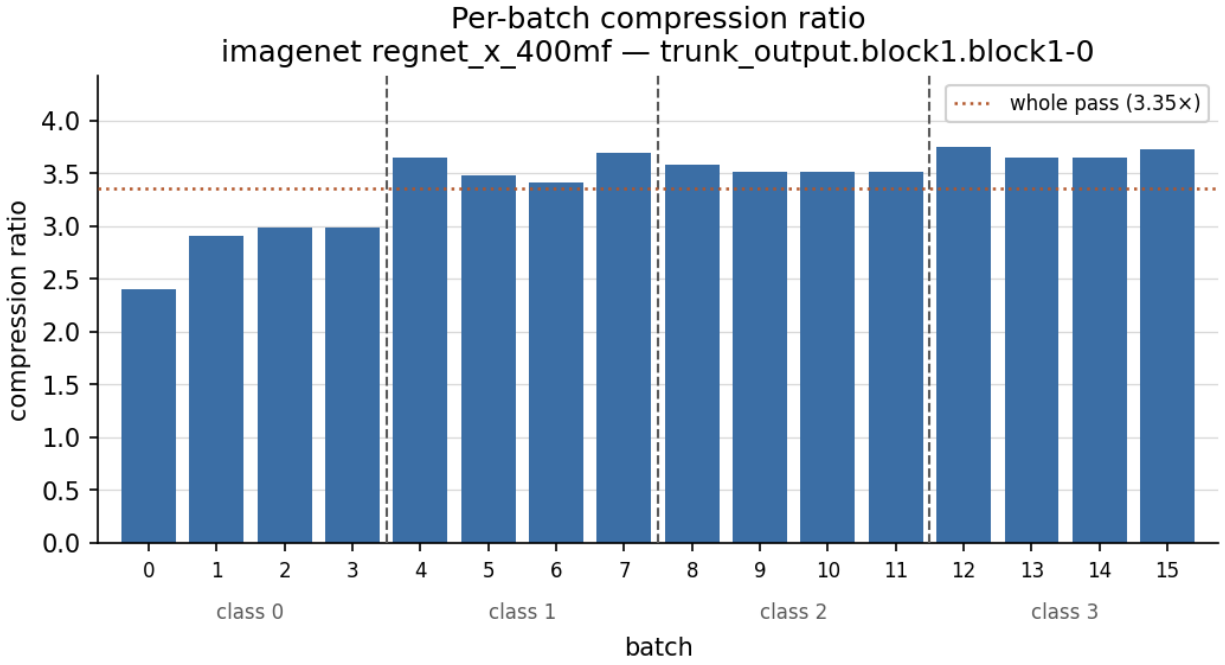


Figure 3.37: Per-batch compression ratio of the adaptive function selection on class-sorted batches, evaluated on the test set (45000 images) with batches of 11 images, i. e. about four batches per class; class boundaries dashed, the ratio of the whole pass (3.35×) dotted. With featuremaps of size $H \times W = 56 \times 56$, this means each swap decision is based on $11 \times H \times W = 34496$ predictions. The ratio is lowest in the first batches, while the selection adapts from its initial configuration, and stays between 3.4 and 3.75 afterwards, also across class changes.

Context size How much context should the prediction functions be allowed to use? We repeat the parameter search with adaptive function selection for a maximum context of $K \in \{0, 1, 2, 3\}$ buffered rows (section 3.4.1), which determines which spatial prediction functions are applicable at all. Figure 3.38 shows the resulting Pareto fronts. Without any buffered row ($K = 0$), only the

west ray and the stochastic predictor remain, and the compressor falls clearly behind. All other context sizes ($K = 1, 2, 3$) share the same front. More context is therefore not necessarily better. A larger context makes more prediction functions applicable, but the prediction functions requiring higher buffering apparently work similarly well when compared to the once that are content with less; those are also usually computationally cheaper.

If the more expensive prediction functions would truly bring more benefit, then surely the greedy compression scheme would select them more often when they are unlocked for higher context sizes, but either they are not selected, or they do not provide any benefit.

In section 3.6.2 we will evaluate different context sizes again, but in combination with Huffman coding. We will come to the conclusion that not only do the predictors chosen at higher context sizes perform — individually — better, as they cover more predictions relative to the others, but that — because they perform better — lower context sizes actually yield better results when combined with Huffman encoding.

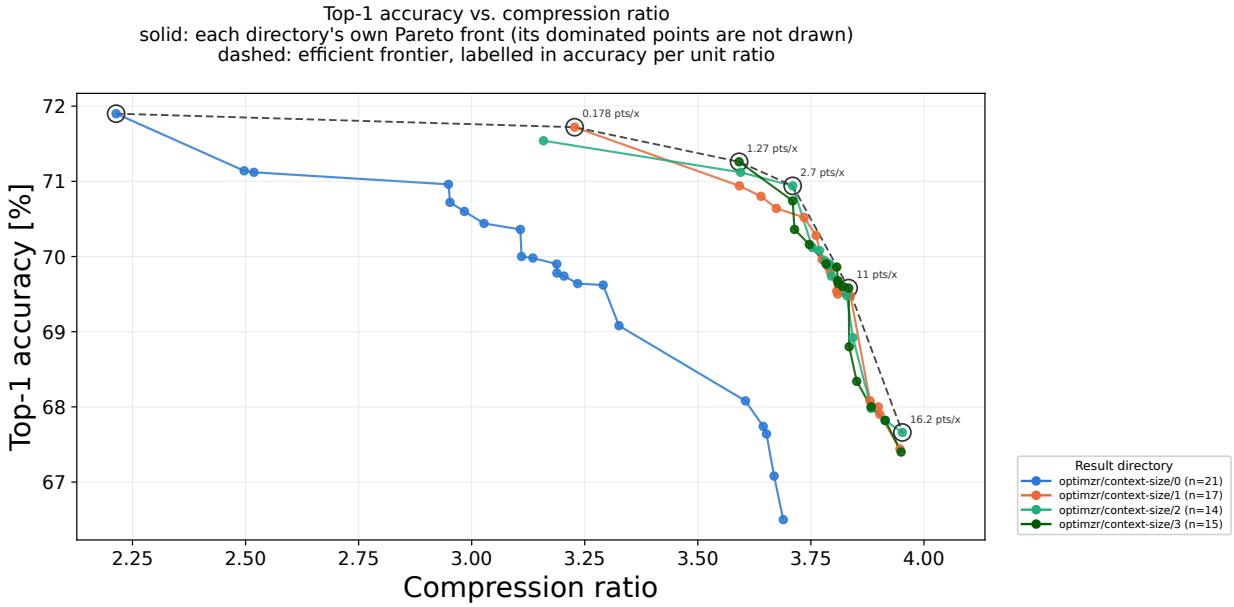


Figure 3.38: Pareto fronts of the parameter search with adaptive function selection for a maximum context of $K = 0$ (blue), 1 (orange), 2 (light green) and 3 (dark green) buffered rows, 3 pp budget, compression ratio of the SZ stage alone; dashed: common efficient frontier. Without a buffered row, the ratio stays below 3.7; from one row on, the fronts practically coincide up to the 4:1 limit of the 2-bit codes.

3.6 Lossless and Pre-processing Stages

Here we extend the SZ compression module with lossless entropy coding and a lossy preprocessing stage to further increase the compressors efficacy.

Figure 3.39 shows the complete compressor pipeline. Besides the SZ stage of the previous sections, it contains a lossy pre-processing stage and a lossless coding stage, which this section introduces. The featuremap is first downsampled by strided subsampling. Every value is then predicted by the best of up to three active prediction functions of its channel and either accepted within τ_v and τ_b or replaced by an outlier. Finally, the codes are Huffman-coded in pairs with a codec adapted per batch, while the outliers are stored as they are.

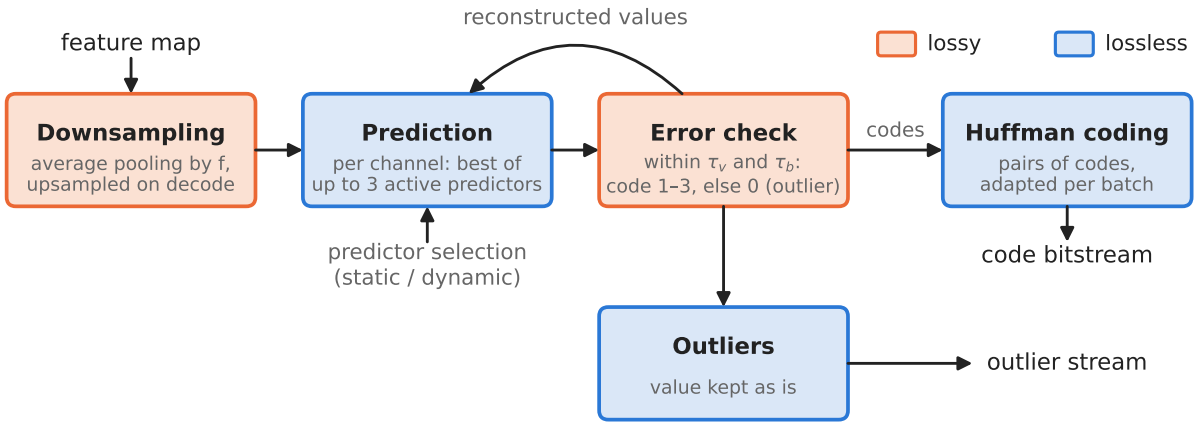


Figure 3.39: The compressor pipeline. Lossy stages in orange, lossless stages in blue. The featuremap is downsampled by strided subsampling, every value is predicted by the best of up to three active predictors of its channel, and the prediction is accepted if it stays within τ_v and τ_b (codes 1–3) or replaced by an outlier (code 0). The codes are Huffman-coded in pairs with a codec adapted per batch; outliers are stored as they are.

3.6.1 Downsampling

Every prediction-based scheme of the previous sections is fundamentally limited: even if every value is predicted, each one still costs a code, which bounds the compression ratio by $b_{\text{in}}/b_{\text{code}}$, i. e. 4:1 with 2-bit codes (eq. (3.1)). One remedy is to reduce the number of values the prediction stage has to code in the first place. We therefore add spatial decimation (section 2.5.2.2) as a pre-processing step in front of the SZ stage: the featuremap is downsampled by a factor f_{down} in both spatial dimensions, compressed by SZ, and upsampled back to full resolution after decompression. A factor of f_{down} alone yields a compression ratio of f_{down}^2, which multiplies with the ratio of the subsequent stages.

Design dimensions A downsampling stage can be configured along several dimensions. The *downsampling method* determines how the $f_{down} \times f_{down}$ values of a window collapse into one: strided subsampling keeps one of them and drops the rest, while average pooling mixes all of them. The *upsampling method* determines how the reduced featuremap is expanded again: nearest-neighbour upsampling replicates every stored value over its window, while bilinear interpolation interpolates between neighbouring stored values. The *factor* f_{down} can be restricted to the grid of the featuremap, i. e. to divisors of its height and width, or be chosen freely by interpolating values between the grid points, which allows a finer choice of the decimation at the cost of additional arithmetic. Finally, the downsampling stage and the SZ stage have to *share the error budget* of the connection.

Chosen configuration We use strided subsampling with nearest-neighbour upsampling. We evaluated all four combinations of downsampling and upsampling methods in isolation (section A.13). At the low factors that fit into our accuracy budgets, their performance is comparable: within a 3 pp budget, only a factor of 2 holds, and it does so for all four combinations. Among them, subsampling with nearest-neighbour upsampling is the one that needs no arithmetic at all. We restrict f_{down} to the grid of the featuremap. The factor is chosen by the parameter search (section 3.7.2): after the initial quantization sweep, which estimates the error tolerance of the connection and thereby the starting value of

τ_b, the search sweeps the admissible factors. For every accuracy budget, it fixes the largest factor that holds the budget on its own, and subtracts the average error $\varepsilon_{\text{down}}$ this factor induces from the starting value of τ_b, so that both stages share the budget. SZ then compresses the reduced featuremap.

Cost The factor is fixed per connection and accuracy budget before deployment, so nothing has to be transmitted. In hardware, strided subsampling needs neither arithmetic nor buffering: the compressor simply drops all values outside every f_{down}-th row and column. Nearest-neighbour upsampling on the decompressor side has to emit every reconstructed row f_{down} times, which requires buffering one row of the reduced featuremap, i. e. W/f_{down} values per channel. At the same time, downsampling lowers the cost of the subsequent SZ stage: its row buffers shrink by the factor f_{down}, since the reduced featuremap is only W/f_{down} values wide.

Downsampling alone is no alternative to the SZ stage, as it reaches only a ratio of 4 within the budget (Fig. A.53). Combined with it, however, downsampling lifts the ratios the parameter search reaches within the 3 pp budget from below 4 to 8 to 12 (Fig. 3.40).

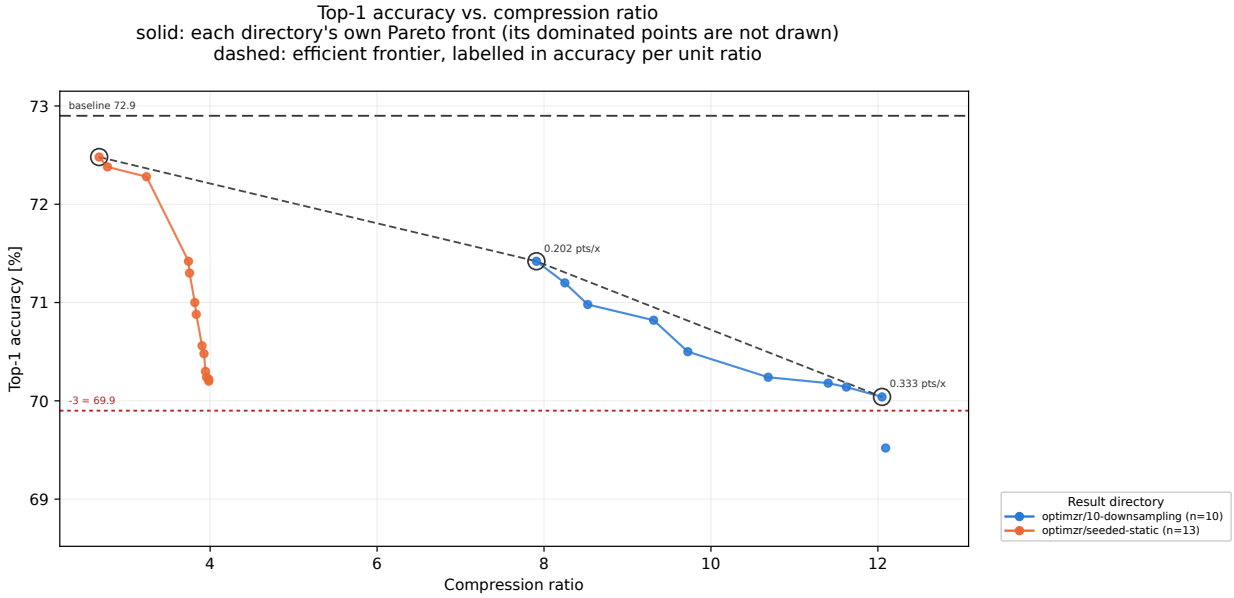


Figure 3.40: Pareto fronts of the parameter search with downsampling in front of the SZ stage (blue) and without it (orange, static selection), within a 3 pp budget. Without downsampling, the ratio saturates below 4; with downsampling, the search reaches ratios of 8 to 12 within the budget.

Limitations This gain comes with several drawbacks. The downsampling step has to fit into the accuracy budget on its own, and it takes away budget from the SZ stage. Unlike the SZ stage, it is not error-controlled: its error follows from the factor and the data, and is only measured, not bounded. And the restriction to the grid of the featuremap makes it inflexible. The 7×7 featuremaps of late RegNet layers, for example, only admit the factors 1 and 7, i. e. a reduction from 49 values to a single one. In this sense, downsampling is merely an inflexible form of spatial quantization. It can nonetheless increase the compression ratio of our prediction-based compressor considerably, where the connection tolerates it.

3.6.2 Huffman

The code distributions in Figs. 3.7 and 3.30 have shown that code streams can have uneven distributions, indicating low entropy of the code symbol stream. This is where entropy encoding, particularly Huffman encoding can be used to further push the compression ratio towards its theoretical bounds.

Design dimensions A Huffman stage for the code stream can be configured along several dimensions. The *scope* of a codec ranges from a single codec for the whole network over one codec per layer to one per channel, with groups of channels sharing a codec in between. The *block size n* determines how many successive codes are coded as one Huffman symbol (block codes, section 2.5.1.4). The codec can be *created* statically, from a distribution profiled on calibration data, or dynamically, from the distribution observed at runtime. Further dimensions are which streams are coded at all — here only the code stream including the outlier marker, while the outlier values themselves are stored as they are — and a limit on the maximum codeword length, as hardware decoders commonly impose; with alphabets of 4 or 16 symbols, such a limit is not restrictive.

Chosen configuration We use one codec per layer. One codec per channel could fit the code distributions more closely, since every channel uses its own set of prediction functions. The benefit is, however, likely small: the selection optimizes the prediction functions of every channel to fit its data as well as possible, which evens out the code distribution and leaves less for Huffman coding to exploit (see the end of this section), and with only 4 symbols and a minimum codeword length of 1 bit there is little room to adapt the code to a channel in the first place. Since some prediction functions are nevertheless always used more often than others, and these need not occupy the same slots in every channel, channels with similar code distributions could share a codec, which would save most of the on-chip storage of per-channel codecs. We evaluate block sizes of $n = 1$ and $n = 2$. With $n = 2$, pairs of successive codes form one Huffman symbol over an alphabet of $(n_{\text{active}} + 1)^2$ symbols, e. g. 16 for 4 codes; since the pair histogram is counted on the data, this also captures the correlation between neighbouring codes. The codec is updated dynamically: the compressor counts the codes of every batch and, after each batch, rebuilds the codec from the running counts of all batches so far, starting from a uniform distribution. This accommodates the adaptive function selection (section 3.5), which can change the code distribution from batch to batch.

Cost An updated codec has to reach the decompressor. It can either be sent in a header with every batch, or be kept in a table on chip that compressor and decompressor share; the latter requires an interconnect between the two, which may be expensive if they are placed far apart on the chip. With one codec per layer, the header is small, with 4 code lengths, or 16 with block codes of $n = 2$, and negligible for large batches. With one codec per channel, it grows linearly with the number of channels, which reaches 400 in later layers of RegNetX-400MF, and larger block sizes multiply it further, as the alphabet grows as $(n_{\text{active}} + 1)^n$. The hardware cost follows the same pattern: the compressor and decompressor have to buffer the code table, which is negligible for one table per layer, but grows linearly with the number of channels and exponentially with the block size for per-channel codecs.

Strictly speaking, an updated codebook would have to be accounted for in the compression ratio: unlike the statically profiled parameters of section 3.1.5, it is derived from data at runtime. We do not count it, because the codebook of a code alphabet this small is negligible at our batch size: with 2-bit codes, the

alphabet has 4 symbols, or 16 with block codes of $n = 2$, while a batch of 512 images comprises more than $5 \cdot 10^7$ codes on our connection.

Figure 3.41 compares the same operating points without and with Huffman coding of the code stream. Since Huffman coding is lossless, the accuracy of every point is unchanged, while the compression ratio rises by 0.4 to almost 1.9. The gain grows with the compression ratio, as laxer thresholds skew the code stream further towards a single predictor.

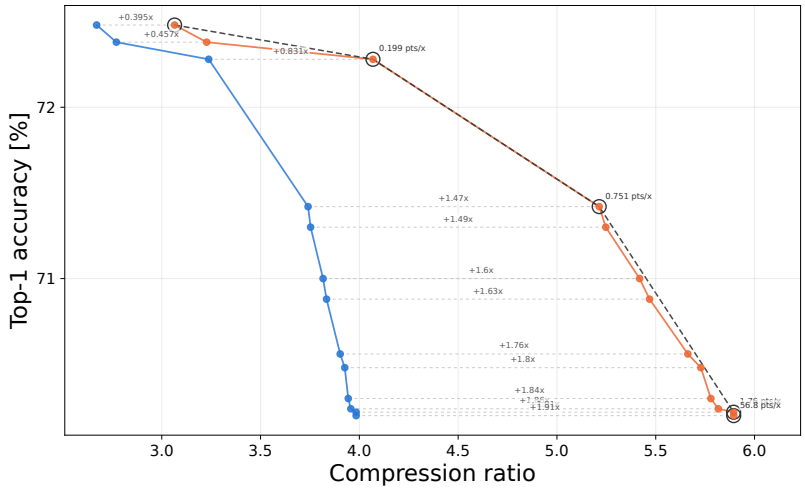


Figure 3.41: Pareto fronts of the same operating points without (blue) and with (orange) Huffman coding of the code stream. Grey lines connect corresponding points and are labelled with the gain in compression ratio. The gain grows from 0.4 to almost 1.9 with the compression ratio, as the code stream becomes more skewed towards a single predictor.

Block size Figure 3.42 compares block codes of $n = 1$ and $n = 2$ on the same operating points. The Huffman stage is lossless, so the accuracy of every point is identical, while the compression ratio rises by 18 to 26 % with $n = 2$, e. g. from 5.82× to 7.24× at $\tau_v = 0.26$. This gain is not due to correlation between neighbouring codes: per code, the entropy of the pair stream is 0.945 bit, barely below the 0.959 bit of the single codes. It comes from the rounding of codeword lengths instead. A Huffman code spends at least 1 bit per symbol, while the dominant code carries well under 1 bit of information when most codes point to the same prediction function. With $n = 1$, the code stream costs 1.354 bit per code, 0.40 bit above its entropy. With $n = 2$, the most frequent pair is coded with a single bit for two codes, which lowers this bound to 0.5 bit per code; the code stream costs 1.084 bit per code, 0.14 bit above its entropy.

These values are close to what an ideal Huffman code achieves. Part of the remaining slack stems from the dynamic codec, which starts from a uniform distribution, so that the first of the 10 batches costs about 2 bit per code for both block sizes. Without it, the code stream costs about 1.28 bit per code with $n = 1$ and 0.98 bit with $n = 2$. For independent codes with a distribution of similar entropy, e. g. $p = (0.83, 0.08, 0.05, 0.04)$ with 0.92 bit, a Huffman code reaches 1.26 bit per code with $n = 1$ and 0.98 bit with $n = 2$. Since the codes make up almost the entire compressed stream — the outliers add only about 0.02 bit per value — the compression ratio follows the codeword length almost directly: the ratio is $8/(1.354 + 0.022) = 5.8$ with $n = 1$ and $8/(1.084 + 0.022) = 7.2$ with $n = 2$, so that a 20 % shorter code stream yields a 24 % higher compression ratio.

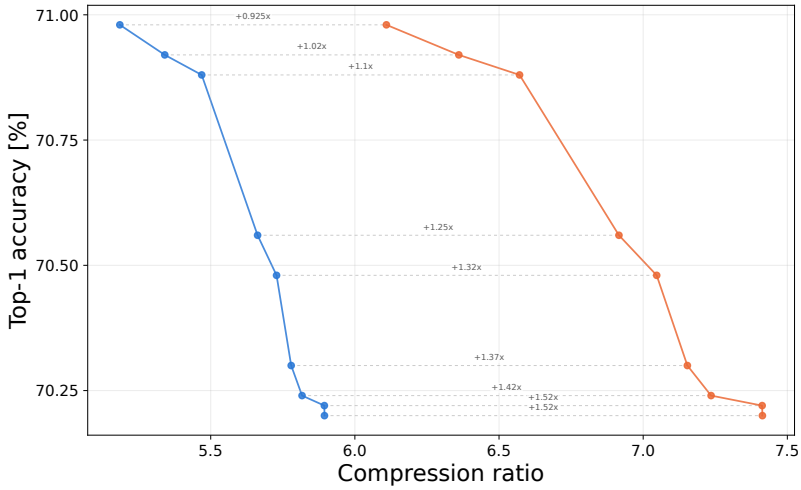


Figure 3.42: Pareto fronts of the same operating points with Huffman block codes of $n = 1$ (blue) and $n = 2$ (orange), i. e. every Huffman symbol codes one or two successive SZ codes. Grey dashed lines connect the points of equal rank on the two fronts and are labelled with the difference in compression ratio.

Sensitivity to Code Distribution Huffman coding can only gain where the entropy of the code stream is low. Throughout this chapter, however, we made the prediction functions fit the data better, extended the set of prediction functions, and let the compressor regularly replace functions by better-fitting ones. A better-fitting set produces fewer outliers, but also spreads its hits more evenly over the slots, which makes the code stream harder to compress for Huffman. This raises the question whether the gain from fewer outliers outweighs the loss in the Huffman stage, and whether a better-fitting set retains enough accuracy to compress further within the same budget.

To answer it, we compare two function selections on the same operating points (Figs. 3.43 and 3.44): selection A is fitted once per layer from a smaller pool of prediction functions and is dominated by the constant predictor, while selection B is fitted per operating point from the full pool, including the regression predictors. Both are described in detail in **??**. At equal outlier shares, A's codes concentrate on the first slot, while B's spread more evenly over all three slots (Fig. 3.43). Both selections reach nearly the same ratio in the SZ stage, but Huffman coding gains up to 1.43× on A and at most 1.17× on B (Fig. 3.44).

The outlier share is already low for both selections, and every further outlier has little effect: in a 56×56 feature map, one outlier adds 8 bits to the 6272 bits of the channel's 2-bit code stream, lowering the compression ratio by about 0.1 %. The Huffman stage, in contrast, has a much larger effect when the code stream has a low entropy. At the operating point with the largest difference between the two selections, the code stream of A has an entropy of 1.25 bit per code and that of B of 1.64 bit; with block codes of $n = 2$, Huffman coding exploits A's more skewed distribution for a gain of 1.44×, compared to 1.17× for B, and a total compression ratio of 5.01× compared to 3.99×, although the SZ stage alone compresses both to about 3.4–3.5×. A prediction-based compressor followed by Huffman coding should therefore tune its prediction functions not only towards fewer outliers, but also towards a skewed code distribution, similar to how compression-aware training [4] trains a network towards low-entropy activations. In a way, the prediction-functions selection chosen by modern SZ derivatives are maximally skewed per block, since only one is chosen at a time.

The original SZ [13] took a different route and compressed the code stream with a dictionary coder, which works well with large contexts, as the few distinct symbols promise repeating patterns, and was successful on scientific data sets. Dictionary coders are, however, costly in hardware [44, 88], and thus out of scope for this work.

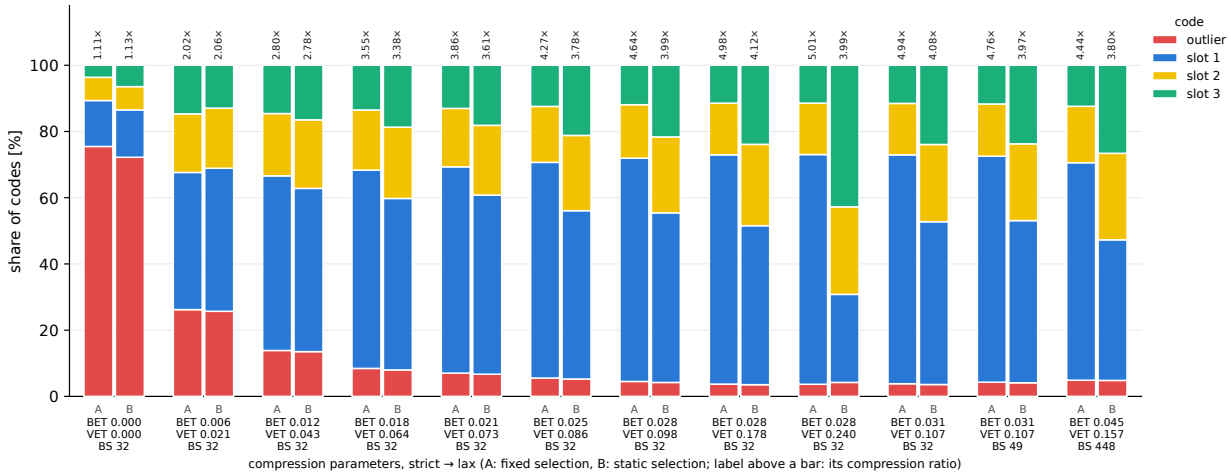


Figure 3.43: Code distribution of selection A (fixed - manually selected set of prediction functions) and B (static - prediction function set found by adaptive run, used statically) over the Pareto points of the search, ordered from strict to lax parameters (τ_b, τ_v, block size BS below each pair), pooled over channels as seen by the layer-wide Huffman codec; the total compression ratio is given above each bar. At equal outlier shares, A's codes concentrate on slot 1, while B's spread more evenly over slots 1 to 3.

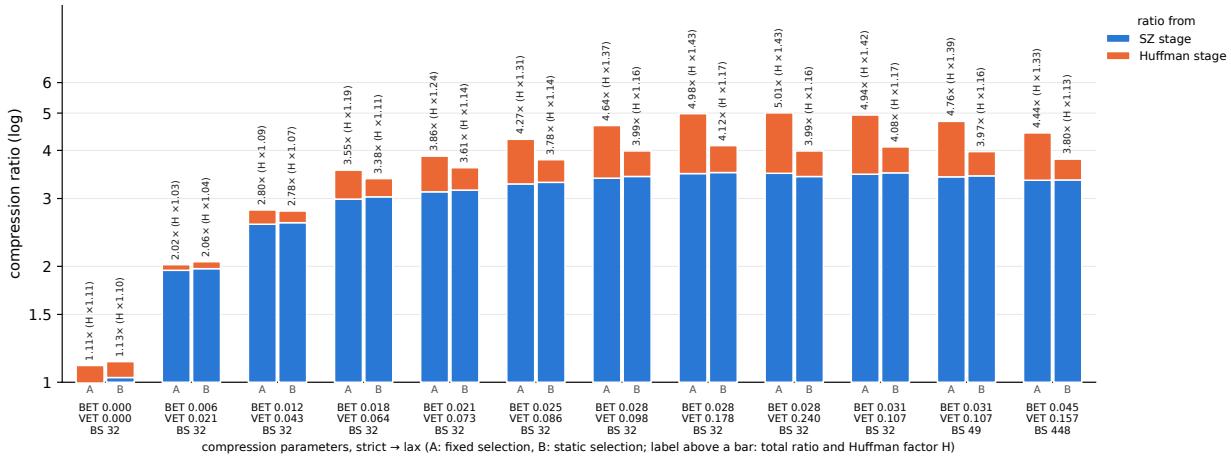


Figure 3.44: Compression ratio of the points of Fig. 3.43, split into the SZ stage (blue) and the Huffman stage (orange) on a logarithmic axis; above each bar the total ratio and the Huffman factor H. Both selections reach nearly the same SZ-stage ratio, but Huffman coding gains up to 1.43× on A and at most 1.17× on B.

Creating Skewed Distributions Figure 3.38 compared the context sizes K on the SZ stage alone. Figure 3.45 shows the same points with Huffman block codes of $n = 2$ used on the code stream of the same configurations.

As before, all context sizes with $K > 0$ dominate the $K = 0$ case in terms of compression ratio per accuracy loss. Here, however, the largest context, $K = 3$, falls behind $K = 1$ and $K = 2$. Since Huffman coding is lossless and only acts on the code stream, this difference stems from a change in the selection of prediction functions, which changes the distribution of the codes. Within the

3 pp budget, Huffman coding gains on average 1.30× for $K = 1$ and 1.31× for $K = 2$, but only 1.22× for $K = 3$. Accordingly, the code streams of $K = 1$ and $K = 2$ have an entropy of 1.43 and 1.42 bit per code, that of $K = 3$ of 1.53 bit. The difference is not a consequence of different operating points: at equal SZ-stage ratios between 3.5 and 3.8, the entropies are practically the same, 1.42 bit for $K = 1$ and $K = 2$ and 1.53 bit for $K = 3$. Without any buffered row, the codes are spread most evenly of all (1.66 bit per code, a gain of 1.12×).

The greedy selection replaces an active function whenever a function of the pool covers more values on its own, i. e. it selects for marginal improvement. With the larger pool unlocked by $K = 3$, the active functions therefore each cover a substantial share of the values, and the codes are distributed more evenly over the slots. With $K = 1$ and $K = 2$, the lower entropy indicates that a single function dominates the code stream more strongly, which increases the Huffman gain. Better-fitting prediction functions thus lead to more evenly distributed codes, a higher entropy and a lower Huffman gain — the same effect as between selections A and B above. The selection scheme is not aware of the subsequent Huffman stage; it could be modified to deliberately create a skewed code distribution.

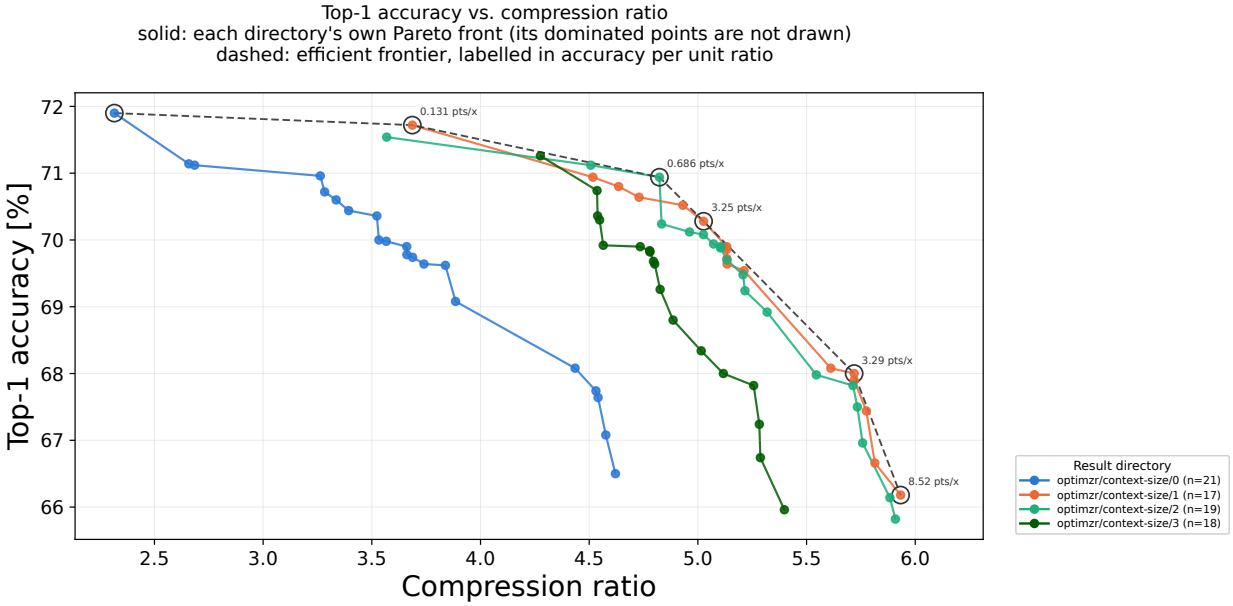


Figure 3.45: Pareto fronts of the parameter search with adaptive function selection for a maximum context of $K = 0$ (blue), 1 (orange), 2 (light green) and 3 (dark green) buffered rows, 3 pp budget, with Huffman block codes of $n = 2$; the same points as in Fig. 3.38. Dashed: common efficient frontier. Without a buffered row, the ratio stays below 4.7; one and two rows reach almost the same front up to about 5.9, while three rows fall behind both.

3.7 Parameter Optimization

The compressor developed in the previous sections spans a multi-dimensional parameter space: the block error threshold τ_b, the value error threshold τ_v and the block size, given by the featuremap subdivision N_{Subdiv}, on the one hand, and the selection of prediction functions on the other hand. Good values for these parameters are specific to the model and to the connection, since every connection carries activations of a different structure and with a different sensitivity of the model to errors. We therefore need a procedure that finds good configurations for every connection.

This section deals with the thresholds and the block size. We first derive sensible initial values for them (section 3.7.1) and then refine them with a trajectory-based search that fits the course of the accuracy drop and compression ratio by least-squares regression (section 3.7.2). The prediction functions are not part of this search; they are optimized by letting them adapt to the data with the dynamic function selection (section 3.5).

3.7.1 Initial Parameter Selection

The starting point of the parameter selection is an accuracy budget Δ, i. e. the accuracy loss the user is willing to accept for the connection. From it, we derive initial values for τ_b, τ_v and the block size, which place the compressor close to the budget and from which the search procedure starts.

3.7.1.1 Block Error Threshold

Block error thresholding has already been shown to effectively limit the average error within a block for any value error threshold (Fig. 3.20). We therefore choose τ_b first, from a quantization sweep: we requantize the activations of the connection to decreasing bitwidths and determine the largest average error, relative to the channel magnitude as in section 3.3, whose accuracy drop is still within the budget. This average error becomes the initial τ_b (see section 3.7.2 for the exact rule). Section A.9.3 shows such sweeps for every connection evaluated in chapter 4.

The resulting τ_b generally places the compressor near the budget, but not exactly at it. Quantization spreads its error evenly over the featuremap, whereas SZ concentrates the error on fewer values: within a block, single predictions may exceed τ_b as long as the block average does not. As section A.3.2 has shown, focused errors hurt the accuracy more than the same average error distributed evenly, so the same τ_b may cost SZ more accuracy than quantization.

3.7.1.2 Value Error Threshold

Given τ_b, we choose τ_v at the knee of the compression ratio: beyond it, a larger τ_v adds little compression but admits larger one-off errors (Fig. 3.14). Figure 3.46 shows the knee score, the normalized compression ratio minus the normalized τ_v, over a τ_v sweep for three block error thresholds. In all three cases, the knee lies at about 3.5 times τ_b, which is the factor the search procedure uses to tie τ_v to τ_b. While this factor does not necessarily hold across different networks, we found it to be a good starting point in practice.

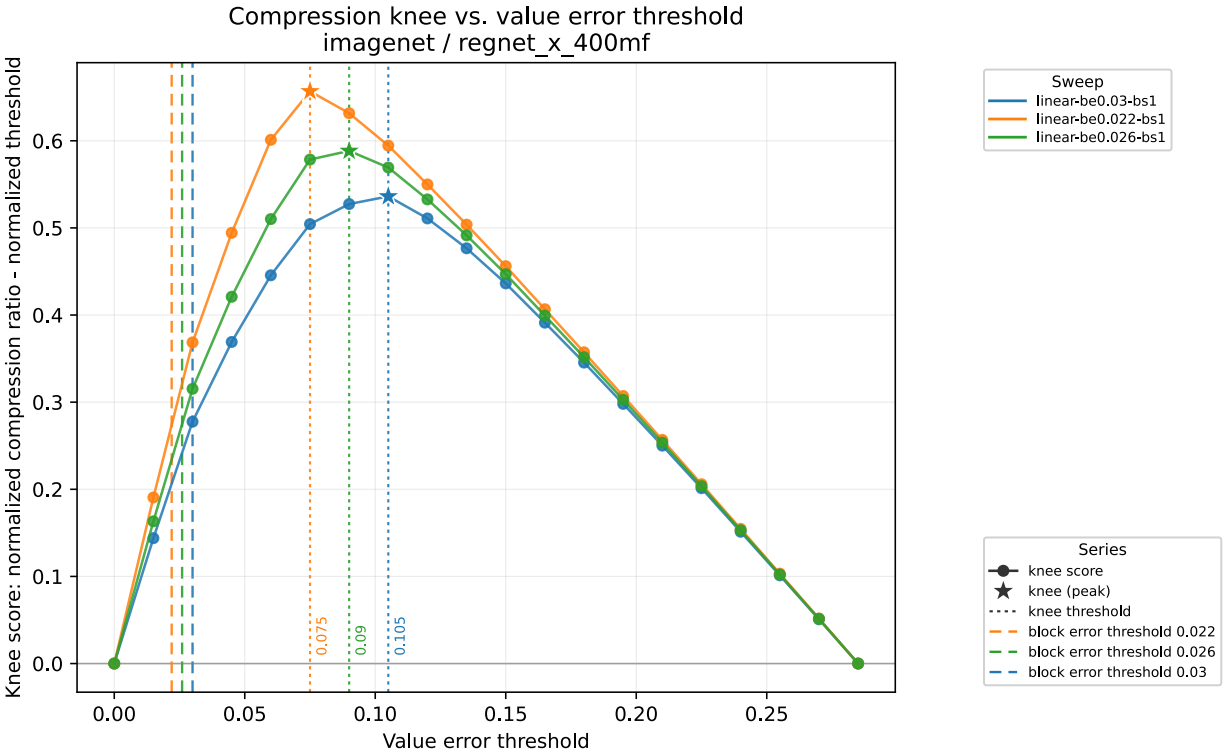


Figure 3.46: Knee score (normalized compression ratio minus normalized τ_v) against τ_v for three block error thresholds with linear blocks (bs1). Stars mark the knee of each sweep, dotted lines its τ_v, dashed lines the respective τ_b. The knee lies at about 3.5 times τ_b in all three cases.

3.7.1.3 Block Size

The block size determines how much error budget can accumulate within a block, and thus how often and by how much single predictions can exceed τ_b while staying within τ_v. Larger blocks therefore produce more of the rare but large errors which, as the error distribution experiment has shown (section A.3.2), hurt the accuracy more than evenly distributed ones. With a larger block, the accuracy can thus keep dropping although the average error is already saturated at τ_b, simply because the error is concentrated on fewer values (section A.3.3). This also shows that the average error alone is not a sufficient measure to limit the accuracy loss.

Figures 3.47 and 3.48 show two views of the same shallow exploration of the τ_b–N_{Subdiv} design space, with τ_v fixed: the first colours the points by τ_b, the second by N_{Subdiv}. Since only a small number of parameter values was explored, the front may extend further towards higher accuracy and higher compression ratios.

The two views are complementary. Points of the same τ_b spread horizontally: the highest τ_b dominates the high-compression end of the front, while lower thresholds, combined with smaller blocks, own its flat, high-accuracy part, and lowering τ_b squeezes the points towards lower compression ratios. Points of the same block size, in contrast, spread vertically: the smallest blocks occupy the top of the front, while four different block sizes share its high-compression end. Bigger blocks thus let the front reach further, but at a steeper accuracy loss per unit of compression ratio, whereas smaller blocks lose less accuracy but cannot reach as far. Going from a single block per featuremap to $N_{Subdiv} = 8$,

for example, can improve the accuracy by more than one percentage point at only a slightly lower compression ratio.

This leaves the question of how to find the combination of block size and τ_b whose accuracy drop is the flattest, but which still takes full advantage of a given budget. The search procedure in section 3.7.2 addresses this question.

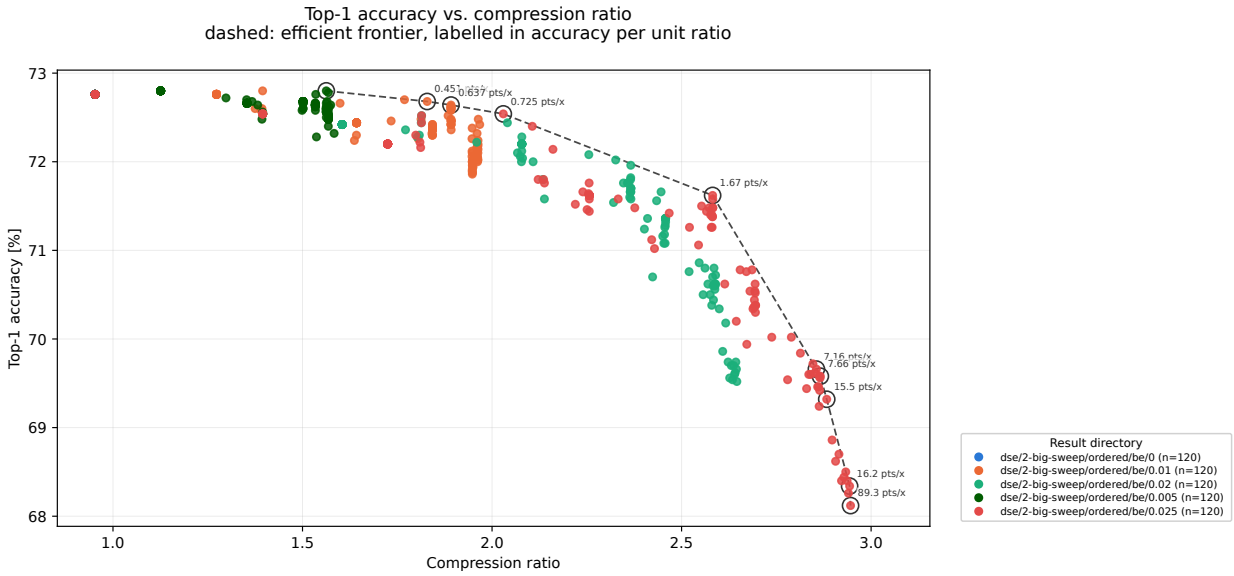


Figure 3.47: Shallow sweep over τ_b and N_{Subdiv} with τ_v fixed, coloured by τ_b. Dashed: efficient frontier, labelled with the accuracy lost per unit of compression ratio. The highest τ_b dominates the high-compression end of the front, while its flat, high-accuracy part is occupied by lower τ_b with smaller blocks; lowering τ_b squeezes the points towards lower compression ratios.

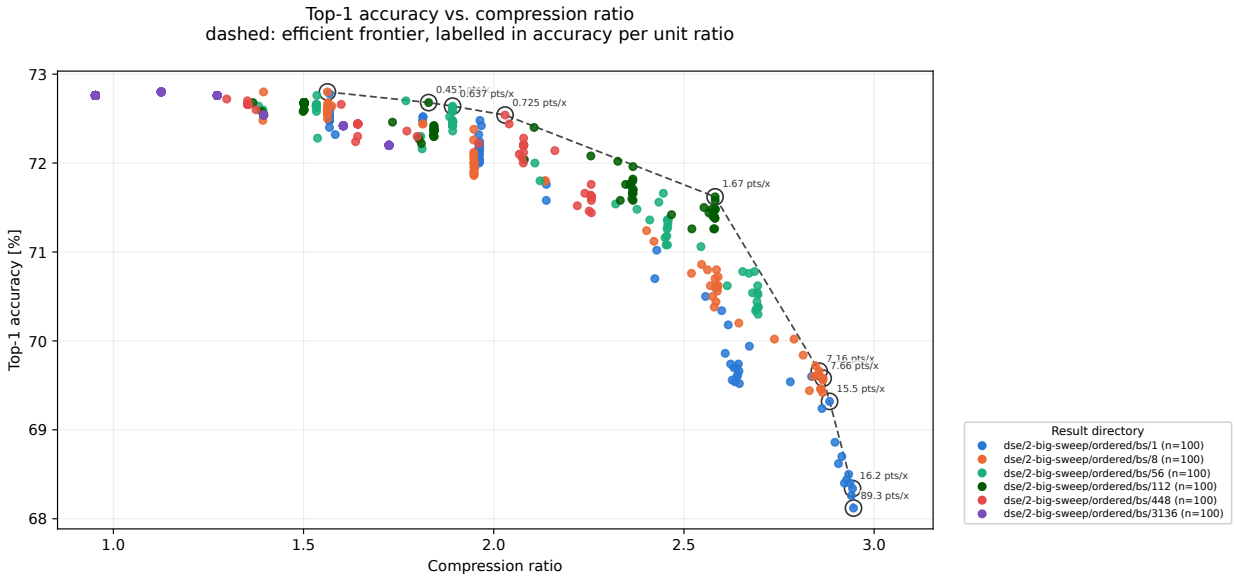


Figure 3.48: The sweep of Fig. 3.47, coloured by N_{Subdiv} (bs in the legend). The biggest blocks, a single block per featuremap ($N_{Subdiv} = 1$, blue), let predictions use more of the accumulated error budget, which pushes the compression ratio further but costs increasingly more accuracy per unit of compression ratio. Subdividing the featuremap into 8 blocks can improve the accuracy by more than one percentage point at only a slightly lower compression ratio.

3.7.2 Search Procedure

We search the parameters $(N_{Subdiv}, \tau_b, \tau_v)$ of one connection at a time, for a set of accuracy budgets Δ, given in percentage points. All evaluations run on the calibration split. Figure 3.49 gives an overview of the procedure.

Objective Let $A(\theta)$ be the top-1 accuracy with configuration $\theta = (N_{Subdiv}, \tau_b, \tau_v)$ applied to the connection, $\mathrm{CR}(\theta)$ its compression ratio and A^* the reference accuracy (see below). For each budget, the search returns

$$\theta^{\star}_{\Delta} = \arg\max_{\theta \in \Theta_{\mathrm{eval}}} \mathrm{CR}(\theta) \quad \text{s.t.} \quad A(\theta) > A^* - \Delta, \tag{3.24}$$

where Θ_{eval} is the set of all configurations evaluated so far, for this or an earlier budget. If no evaluated point lies within the budget, the connection falls back to lossless encoding.

Starting point The search starts from the initial parameters of section 3.7.1. The connection is quantized to $b = 8, 7, \ldots, 0$ bits, and for each bitwidth the accuracy A_b and the average relative error ε_b (eq. (3.3)) are measured. The best accuracy of the sweep serves as the reference accuracy, $A^* = \max_b A_b$. Let u be the first bitwidth whose accuracy leaves the budget. Its error, scaled to the budget boundary, gives the initial block error threshold

$$\tau_b^{(0)} = \varepsilon_u \cdot \frac{A^* - \Delta}{A_u} - \varepsilon_{\mathrm{down}}, \tag{3.25}$$

where $\varepsilon_{\mathrm{down}}$ is the error already spent by a downsampling stage in front of SZ, and zero without one. Until the final refinement step, τ_v is tied to τ_b by the fixed factor $\tau_v = 3.5\,\tau_b$.

Downsampling stage (optional) If a downsampling stage is used, the downsampling factors f_{down} that the feature map admits are swept in ascending order until the accuracy leaves the loosest budget. For each budget, the largest factor that holds the budget on its own is then fixed for the whole search, and its error $\varepsilon_{\mathrm{down}}$ is subtracted from $\tau_b^{(0)}$. SZ then runs on the reduced feature map, with the block sizes admissible for the reduced map. A Huffman stage, in contrast, needs no special treatment: it only changes how the SZ codes are encoded, not the reconstruction, so it leaves every accuracy unchanged and only changes the compression ratios.

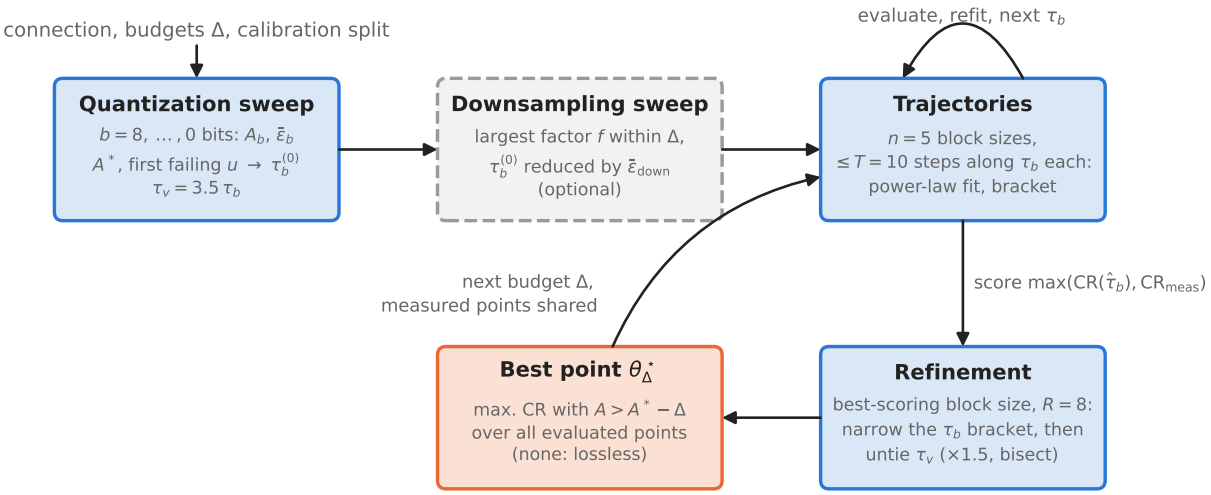


Figure 3.49: Overview of the parameter search for one connection. A quantization sweep sets the reference accuracy A^* and the starting threshold $\tau_b^{(0)}$; an optional downsampling sweep fixes the largest factor within the budget and reduces $\tau_b^{(0)}$ by its error. Trajectories for $n = 5$ block sizes then step along τ_b, refitting their power laws after every evaluation, and the best-scoring block size is refined, first in τ_b, then in τ_v. The result is the evaluated point with the highest compression ratio within the budget. The budgets are searched one after the other and share all measured points; dashed: optional stage.

Trajectories Let $\mathcal{L}$ be the ascending ladder of admissible N_{Subdiv} values. To span the whole range of block sizes, the search picks $n = 5$ entries of the ladder at equal index distance, $\mathcal{L}\left[\text{round}\left(i\,\frac{|\mathcal{L}|-1}{n-1}\right)\right]$ for $i = 0, \ldots, n-1$. Each picked block size gets a *trajectory* of up to $T = 10$ evaluations along τ_b, with τ_v tied to it. Along a trajectory, the accuracy drop $D = A^* - A$ and the compression ratio are modeled as power laws in τ_b, fitted by least squares in log-log space:

$$D(\tau_b) \approx \alpha\,\tau_b^{\beta}, \qquad \mathrm{CR}(\tau_b) \approx \gamma\,\tau_b^{\delta}. \tag{3.26}$$

Points with $D \le 0.05$ are noise around the baseline; they are left out of the drop fit as long as at least two points lie above it.

For $\beta > 0$, the fit predicts that the trajectory crosses the budget at $\hat{\tau}_b = (\Delta/\alpha)^{1/\beta}$. Once the trajectory has measured both in-budget and failing points, they form a bracket $[\tau_{lo}, \tau_{hi}]$, where τ_{hi} is the smallest failing τ_b and τ_{lo} the largest in-budget τ_b below it, and $\hat{\tau}_b$ is clamped into this bracket. Without a bracket, $\hat{\tau}_b$ may lie at most one step $s = 2$ beyond the measured range.

The first evaluation of a trajectory is at $\tau_b^{(0)}$. As long as all measured points lie within the budget, the largest measured τ_b is multiplied by s; as long as all fail, the smallest one is divided by s. Once a bracket exists, the next τ_b moves towards the prediction: it is $\exp\left(0.75 \ln \hat{\tau}_b + 0.25 \ln \sqrt{\tau_{lo}\tau_{hi}}\right)$, a weighted geometric mean of $\hat{\tau}_b$ and the bracket's midpoint, if $\hat{\tau}_b$ lies inside the bracket and is not within 2 % of a measured point, and the geometric midpoint $\sqrt{\tau_{lo}\tau_{hi}}$ otherwise. The trajectory stops once $\tau_{hi}/\tau_{lo} \le 1.02$. To save evaluations, every trajectory after the first starts at the crossing predicted by the previous one rather than at $\tau_b^{(0)}$.

A trajectory is scored by $\max\left(\mathrm{CR}(\hat{\tau}_b),\ \mathrm{CR}_{\mathrm{meas}}\right)$, where $\mathrm{CR}_{\mathrm{meas}}$ is its best measured in-budget compression ratio. If the drop fit does not grow with τ_b, the prediction is unreliable and only $\mathrm{CR}_{\mathrm{meas}}$ is used.

Figure 3.50 shows the trajectories of a search for a budget of 3 pp. Below the budget, the fitted curves of the five block sizes lie close together and only fan out beyond it. The search therefore effectively selects the block size by its compression ratio at the budget crossing, which is highest for $N_{Subdiv} = 2$ in this case.

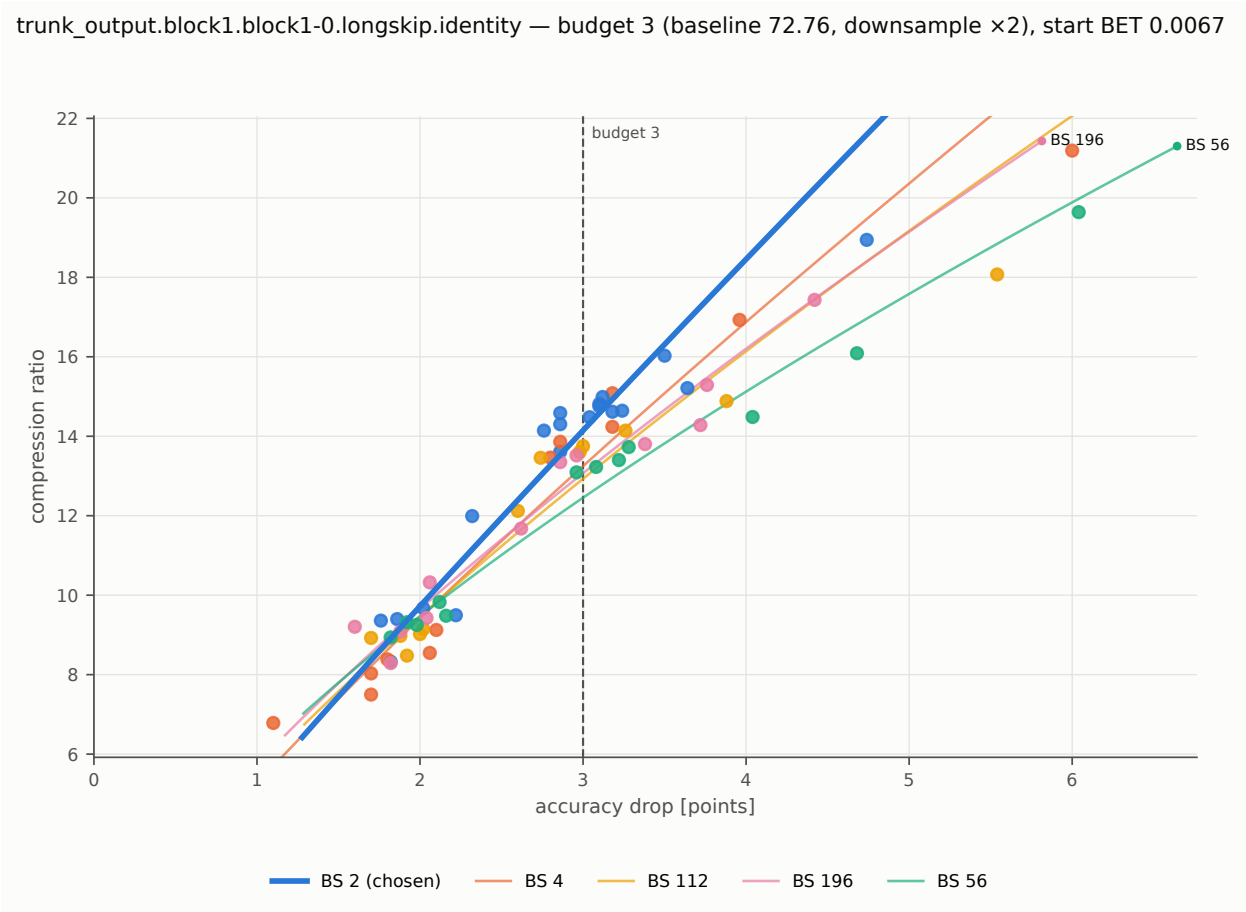


Figure 3.50: Trajectory search for a budget of 3 pp with downsampling by 2 (baseline 72.76 %, starting at $\tau_b^{(0)} = 0.0067$). Every evaluated point is drawn at its accuracy drop and compression ratio, coloured by the subdivision N_{Subdiv} of its trajectory (BS in the legend); the lines are the curves traced by the fitted power laws of each trajectory, the dashed vertical line is the budget. Below the budget, the five curves lie close together and only fan out beyond it; $N_{Subdiv} = 2$ predicts the highest ratio at the budget crossing, about 14.5, and is chosen for refinement.

Refinement The block size with the best-scoring trajectory receives $R = 8$ further evaluations. The first $\lceil R/2 \rceil$ of them narrow the τ_b bracket further, with the same rule as the trajectories. The remaining evaluations untie τ_v from τ_b, starting from the best in-budget point $(N_{Subdiv}, \tau_b, \tau_v)$: τ_v is multiplied by 1.5 until a value leaves the budget, and the resulting bracket is then bisected geometrically until $\tau_{v,hi}/\tau_{v,lo} \leq 1.02$.

Per-channel predictor selection (optional) The dynamic function selection of section 3.5 can either adapt at every evaluated point, or be fixed beforehand by a single adaptive run per connection and downsampling factor at the starting configuration. The latter assumes that which predictors win a channel depends on the connection and its resolution, but only weakly on τ_b and τ_v.

Figure 3.51 shows all points the search evaluates for accuracy budgets of 0.5 to 4 pp. Each search places its best points close to its own budget, and the common efficient frontier is formed by the best point of each budget.

The search evaluates only a few dozen configurations per budget, so it may miss better configurations that it never visits. To check this, section A.6 compares an early version of the search with a grid sweep of 600 configurations on the same connection. With 38 evaluations, the search finds points at up to about 0.2 higher compression ratio than the grid in the range of accuracy it targets, but it does not cover the high-accuracy part of the front, which it never explores.

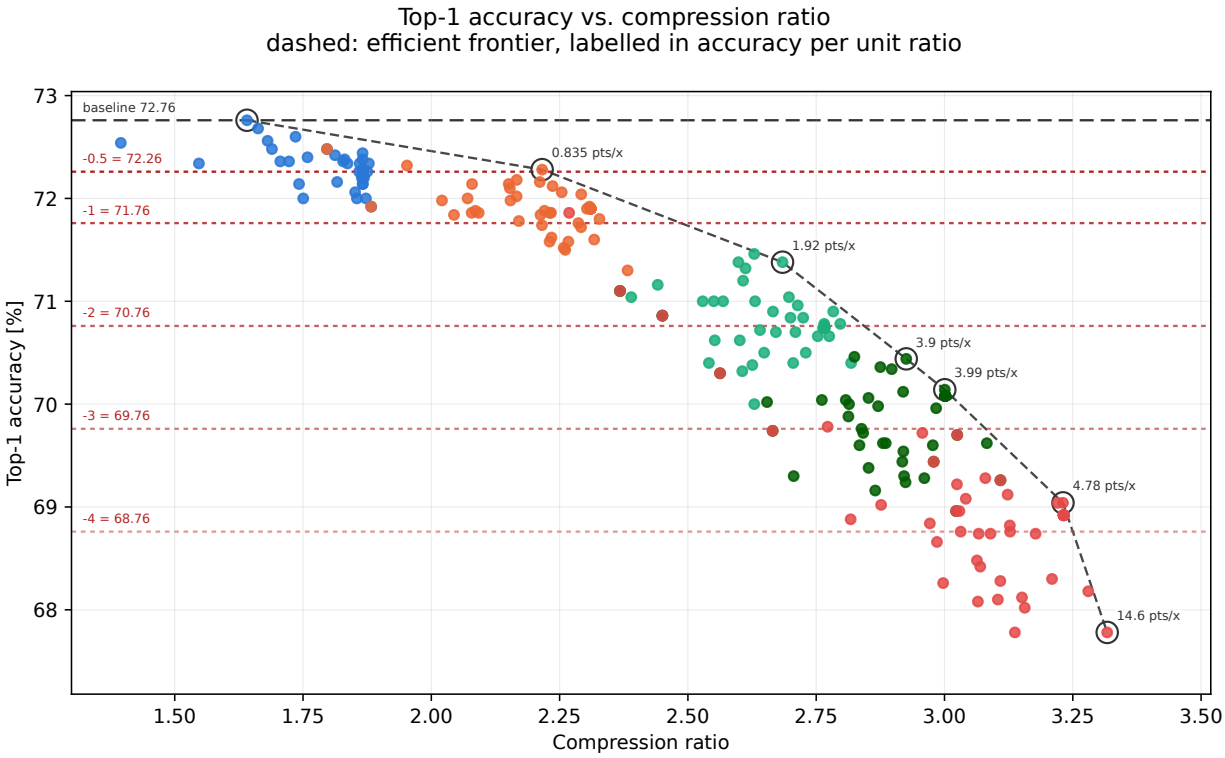


Figure 3.51: All points evaluated by the parameter search for accuracy budgets of 0.5, 1, 2, 3 and 4 pp (colours); red dashed lines: the budgets below the baseline, dashed diagonal: common efficient frontier. Each search places its best points close to its own budget, and the frontier is formed by the best point of each budget.

3.7.3 Generalization to Unseen Data

The parameter search measures compression ratio and accuracy drop on the calibration set only. Deployed, the compressor sees new inputs, so the question is whether a configuration chosen on the calibration set delivers the same average compression ratio and accuracy drop on unseen data. This depends on how representative the calibration set is of the inputs seen in operation. We test it by re-evaluating the configurations found by the search on the test split, whose images were used neither during search nor prior training.

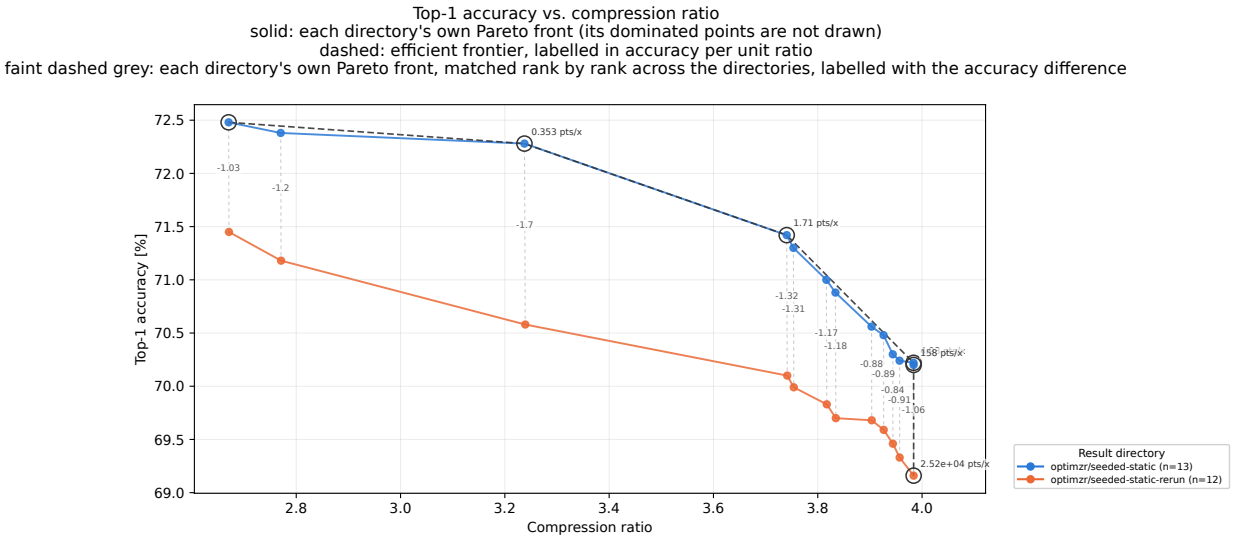


Figure 3.52: Blue: Pareto-optimal points collected from an optimization sweep using the 5000 images calibration dataset using a previously dynamically determined configuration as a static configuration for all points. Points differ in block size τ_b and τ_v **Orange**: Orange points use the same configurations as their blue siblings, connected with a dashed line, but are evaluated using the 45000 image test data set, containing previously unseen images.

Figure 3.52 reevaluates the Pareto-optimal points of a search on the calibration set on the unseen images of the test set. Figure 3.53 plots the accuracy drop on the test set against the drop on the calibration set for the Pareto points of three such reruns. Most points lie on or slightly below the diagonal, and a least-squares fit has a slope of 0.89 ± 0.05: the drop measured on the calibration set carries over to unseen data and is, if anything, slightly pessimistic. The scatter is largest for the 1 pp budget.

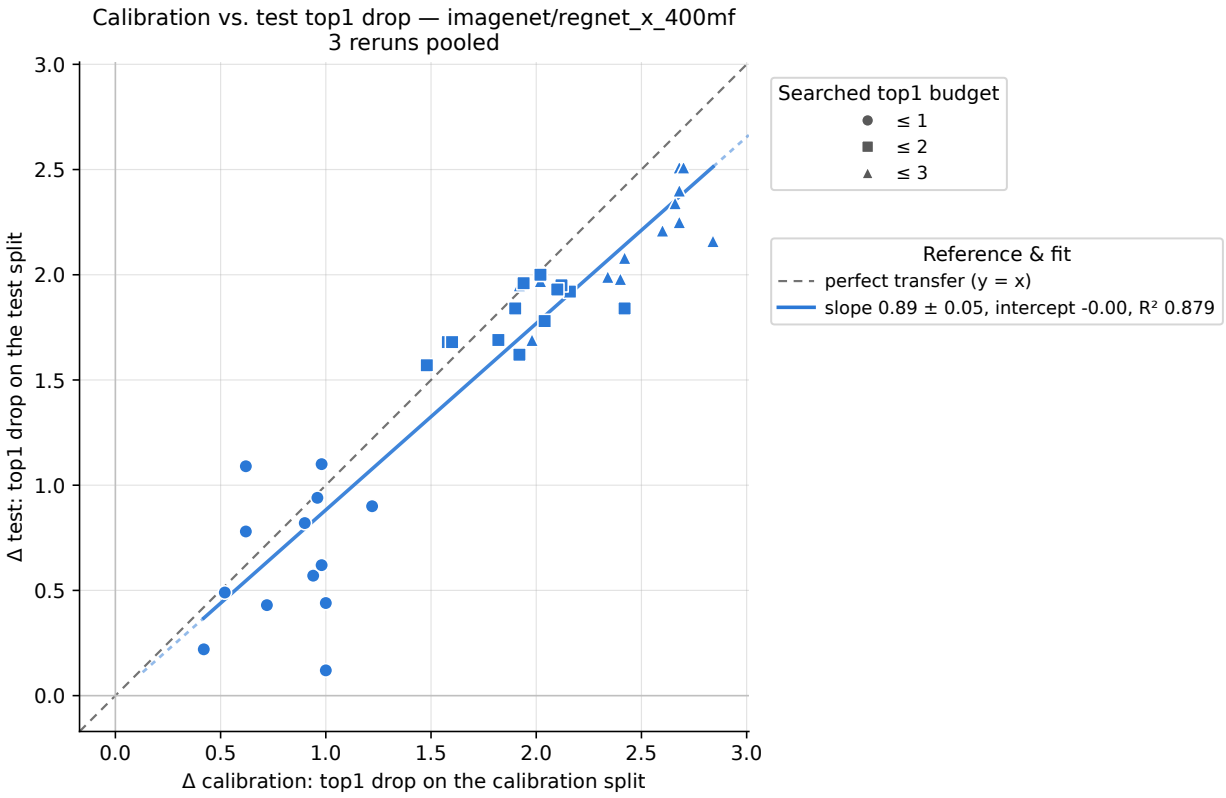


Figure 3.53: Accuracy drop on the test split against the drop on the calibration split for the Pareto points of three reruns; marker: searched budget. Dashed: perfect transfer ($y = x$); solid: least-squares fit with slope 0.89 ± 0.05. Most points lie on or slightly below the diagonal, i. e. the drop measured on the calibration split carries over to unseen data and is, if anything, slightly pessimistic; the scatter is largest for the 1 pp budget.

3.8 Conclusion

In the process of this chapter we evaluated a number of extensions to the SZ prediction-based compression module. All findings stem from a single, deliberately sensitive connection (section 3.1.7) and were mostly measured on the calibration set. The central lessons are the following.

The code width, not the predictor, limits the SZ1 model. Every value costs a code, so the compression ratio is bounded by $b_{\text{in}}/b_{\text{code}}$, i. e. 4:1 for 8-bit activations and 2-bit codes (eq. (3.1)). Plain SZ1 runs into this bound: accuracy falls roughly linearly with τ_v, while the ratio saturates towards 4 (Fig. 3.6). Going beyond it required stages outside the prediction stage. Downsampling reduces the number of values that need a code and lifts the ratio within 3 pp from below 4 to 8 to 12 (Fig. 3.40); Huffman coding shrinks the code below 2 bits and adds another 0.4 to 1.9 (Fig. 3.41). Together, the full design reaches about 14:1 within 3 pp, compared to below 4:1 for SZ1 with block error thresholding (Fig. 3.54).

Bounding the error does not bound the accuracy loss. A value error threshold alone offers only limited fidelity error and accuracy limitation. Adding the block error threshold turns it into a soft accuracy limiter: the sweep levels off at a ratio of about 2.9 and a drop of 3 to 3.7 pp (Fig. 3.20). The same average error, however, costs more accuracy when it is concentrated in few large errors than when it is spread evenly (section A.3.2). Larger blocks allow more such concentration, so the accuracy can keep falling although the average error is

already capped at τ_b, and the block size becomes a compression–accuracy trade-off of its own (Fig. 3.48). Among the block types, the simplest and cheapest, linear blocks in row-major order, dominated square blocks and sliding windows (Fig. 3.19). The combination of value error threshold, block error threshold and block size allows steering not only the error magnitude but to some extend also its distribution and gives finer control of the accuracy drop.

Prediction functions cannot be judged in isolation. Every prediction builds on values reconstructed by other functions, so the value of a function depends on the set it is part of. With laxer thresholds, constant predictions make the linear and polynomial functions of SZ1 collapse into constant predictions (Fig. 3.7). Replacing a single function changes how the others are used: with `avg2` instead of the polynomial function, the formerly barely used linear function takes about a sixth of the codes (Fig. 3.30). More context did not help either: from one buffered row on, the fronts of the SZ stage practically coincide, so a single row suffices at a third of the buffering cost of three (Fig. 3.38). Neither a best function set nor a best context size can therefore be derived from the performance of individual functions.

Cheap and unconventional predictors can outperform sophisticated ones. The random predictor reads no reconstructed values and needs neither buffering nor arithmetic on the data path. As a replacement for the polynomial or the linear function, it nonetheless retains clearly more accuracy near the 4:1 limit (Figs. 3.28 and 3.29), by covering the positions at which the deterministic functions have collapsed into each other. As a replacement for all of them, it falls far behind the SZ1 set (section A.5). Spatial functions, such as the PNG filters and the Lorenzo predictor, improve the accuracy at equal ratio, but every function that reads above the current row costs a buffer of about one row, W values per channel, compared to at most three values for the west-only functions of SZ1 (table 3.2).

Adaptation is best used offline. The greedy per-channel selection finds good configurations, but does not retain them: its best configuration, fixed for the whole run, dominates the fully adaptive scheme (Fig. 3.36). The likely cause is the function interaction above, as the swap statistics are measured on the reconstruction of the previous configuration and do not account for how a swap changes later predictions. Changes in the class composition between batches do not appear to be the reason (Fig. 3.37). Using the scheme offline is also cheaper: online adaptation requires the whole pool of functions in hardware and a configuration message per batch, a fixed configuration only its three active functions.

A better-fitting predictor set can hurt the pipeline. Sets fitted more closely to the data produce fewer outliers, but spread their codes more evenly over the slots, which raises the entropy of the code stream. Two selections with nearly the same ratio after the SZ stage, about 3.4 to 3.5, ended at 5.0× and 4.0× after Huffman coding, because one of them had the more skewed code distribution (Fig. 3.44). With a Huffman stage downstream, the prediction functions should therefore be tuned not only towards fewer outliers, but also towards a skewed code distribution. Block codes of $n = 2$ add another 18 to 26 %, a gain that stems from the rounding of codeword lengths rather than from correlation between neighbouring codes (Fig. 3.42). The same effect appears for the context size: with three buffered rows, the larger pool of functions spreads the codes more evenly (1.53 instead of 1.42 bit per code), Huffman coding gains 1.22× instead of 1.31×, and one or two rows end up ahead of three, although all three reach the same front in the SZ stage (Fig. 3.45). Since the greedy selection optimizes for coverage alone, it is not aware of this; a selection that also favours a skewed code distribution is left for future work.

The parameters can be found cheaply and transfer to unseen data. A quantization sweep yields a usable initial τ_b, and $\tau_v \approx 3.5\,\tau_b$ lies at the knee of the compression ratio (section 3.7.1). Power-law fits along τ_b trajectories, one per block size, locate the budget crossing within few evaluations (Fig. 3.50). On the unseen test set, the accuracy drop carries over with a fit slope of 0.89 ± 0.05, i. e. the estimate from the calibration set is, if anything, slightly pessimistic (Fig. 3.53).

3.8.1 Final Design Points

Figure 3.54 compares five main design points of this chapter within a 3 pp budget: SZ1 with block error thresholding, the static and dynamic per-channel selection of prediction functions, the pipeline with Huffman only, and the full design with downsampling and Huffman coding. We refer to the full design as ASZ (Activation SZ).

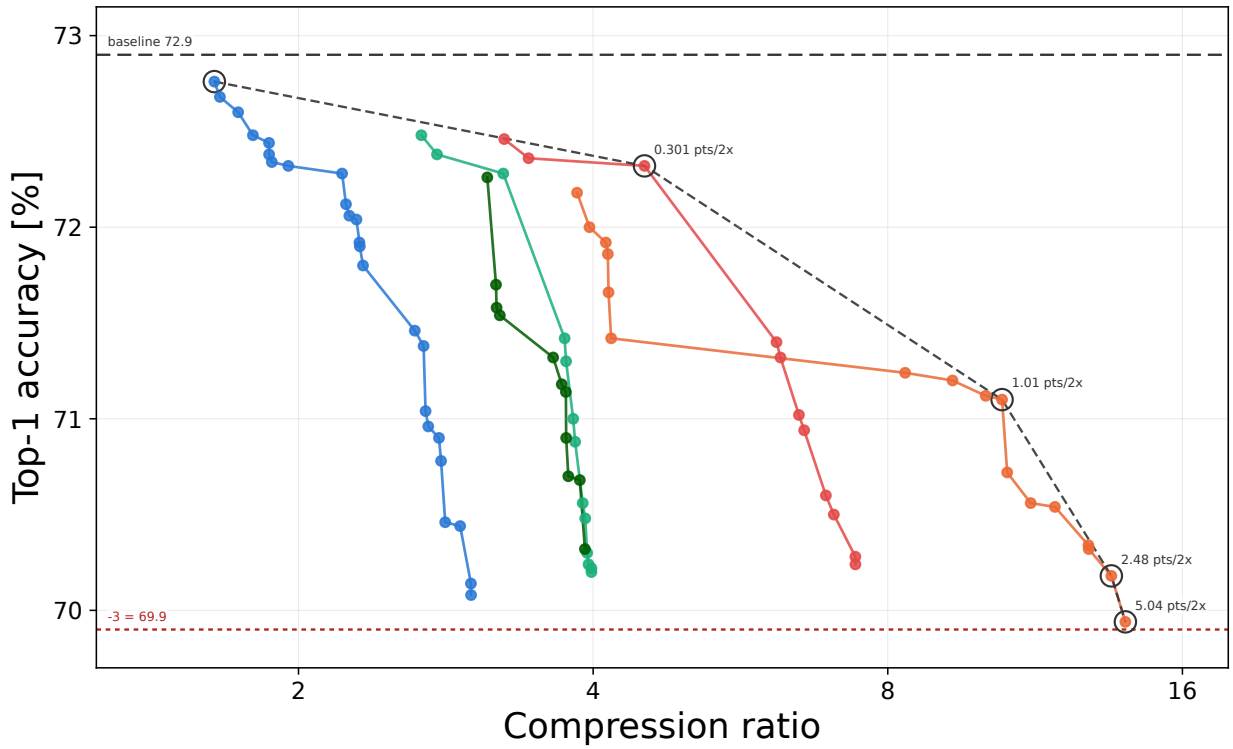


Figure 3.54: Pareto fronts of the three main design points, of the dynamic predictor selection and of the static selection with Huffman coding, within a 3 pp budget, compression ratio on a logarithmic axis: SZ1 with block error thresholding (**blue**), the static per-channel predictor selection (**light green**), the dynamic per-channel predictor selection (**dark green**), the light green points again with Huffman block codes of $n = 2$ added (**red**; same accuracy, since Huffman coding is lossless), and the full design with downsampling and Huffman coding (**orange**). Dashed: common efficient frontier, labelled in accuracy lost per doubling of the ratio. Huffman coding alone lifts the static selection from about 4 to about 7.4; the full design extends the reachable ratio to about 14 within the budget.

These design points reflect the continual improvement of the compression scheme throughout this chapter. Most remarkable is the red plot, showing the performance of a run in which we fused our insights from the combination of Huffman with the predictions-based compression module. For that run, we chose a particularly skewed distribution of codes that resulted in a lot of gain from the Huffman stage.

In Section 3.6.2 we showed that the efficacy of the Huffman stage is very dependent on the distribution of codes and also, that choosing a smaller context

size can produce such skewed distributions by favoring a single compression function.

3.8.2 Hardware Cost

Table 3.4 summarizes the estimated hardware cost of every technique of this chapter. As in section 3.1.5, arithmetic is counted in FLOPs per value and buffers in values; both are given per channel and are replicated C times in the fully unrolled design. The last column lists the side information that has to reach the decompressor with every batch, in addition to the compressed stream.

Error bounding Error bounding is the cheapest part of the compressor (section 3.3.4). The value error threshold costs a single comparison against the profiled constant Δ_c^v. The block error threshold adds a second comparison and two additions, one for the error sum S_B and one for the accumulated budget, and the registers that hold them. The block size N_{Subdiv} only determines when these registers are reset. Among the block types, linear blocks and the approximate sliding window get by with a single set of registers, while spatial blocks need one set per block across the width of the featuremap and the exact sliding window a buffer of the last N_{SW} errors.

Prediction function set The arithmetic of the prediction stage depends on the concrete functions and is listed in table 3.2. The buffer is determined by the context size K (section 3.4.1): every function that reads above the current row requires a row buffer of up to $K\,(W+1)$ values, while the west-only functions of SZ1 get by with at most three. A single buffered row already reaches the same front as two or three (Fig. 3.38); with it, the most expensive active set costs at most 40 FLOPs per value, compared to 7 for the SZ1 set (section 3.4.5, table 3.3).

Function selection The selection determines which functions have to be instantiated (section 3.5). A static configuration requires only the three active functions of every channel: their arithmetic adds up, their buffer is the maximum over the three, and the encoding decision adds $3\,|F_c| - 1$ FLOPs. Dynamic selection has to evaluate the whole pool at every position, so the arithmetic of all pool members adds up and the buffer is that of the most demanding member. On top of this come the swap statistics, i. e. the error of every pool member, its comparison against Δ_c^v and the rescue, sole-cover and mean-error accumulators, and the configuration update. Since every channel swaps at most one slot per batch, it suffices to transmit the swap instead of the full configuration: 2 bits per channel select the replaced slot, or none, and $\lceil \log_2 N_{\text{pool}} \rceil$ bits the pool member that replaces it. This costs at most $C\,(2 + \lceil \log_2 N_{\text{pool}} \rceil)$ bits per batch.

Huffman coding The static codec is a small table of $(n_{\text{active}}+1)^n$ entries, i. e. 4 for single codes and 16 for block codes of $n = 2$ (section 3.6.2). Encoding and decoding are table lookups without arithmetic. With one codec per layer, the table is negligible; with one codec per channel, it is multiplied by the channel count C. Updating the codec per batch adds one counter per symbol and the reconstruction of the code after every batch, and the code lengths have to reach the decompressor, either in a header with every batch or through a table that compressor and decompressor share. At our batch size, this header is negligible, and we do not count it in the compression ratio.

Downsampling Strided subsampling costs neither arithmetic nor buffering, as it only drops values (section 3.6.1). On the decompressor side, nearest-neighbour upsampling repeats every reconstructed row f_{down} times and therefore buffers one row of the reduced featuremap. In return, downsampling shrinks every row buffer of the subsequent SZ stage by the factor f_{down}.

The offline steps, i. e. the profiling of the per-channel constants M_c and μ_c, the parameter search of section 3.7 and the harvesting of fixed configurations, cause no cost at runtime beyond storing their results.

Technique	Section	Arithmetic per value	Buffer and state per channel	Side information per batch
Error bounding				
τ_v only	3.2, 3.3.4	1 comparison	constant Δ_c^v	—
τ_b and N_{Subdiv}	3.3, 3.3.4	+1 comparison, +2 additions	+ error sum S_B, accumulated budget, constant Δ_c^b; reset at every block boundary	—
Block types	3.3.4	linear, spatial: as τ_b; approx. sliding window: +1 subtraction, +1 division (shift)	linear: one register set; spatial: one set per block across the width; sliding window: N_{SW} errors (exact) or one register set (approx.)	—
Prediction function set				
Context size K	3.4.1, 3.4.5	per function, see table 3.2	row buffer of up to $K\,(W+1)$ values; ≤ 3 for the west-only SZ1 functions	—
Function selection				
Static	3.5	$\sum_{f\in F_c} \mathrm{FLOPs}(f) + 3\,\lvert F_c\rvert - 1$	$\max_{f\in F_c} B(f)$	—
Dynamic	3.5	$\sum_{f\in \mathrm{pool}} \mathrm{FLOPs}(f) + 3\,\lvert F_c\rvert - 1$, plus error and comparison of every pool member	$\max_{f\in \mathrm{pool}} B(f)$; rescue and mean-error accumulators per pool member, 3 sole-cover counters	swap per channel, at most $C\,(2 + \lceil \log_2 N_{\mathrm{pool}} \rceil)$ bits
Huffman coding				
Static codec	3.6.2	table lookup, no FLOPs	code table of $(n_{\mathrm{active}} + 1)^n$ entries per layer, $\times C$ per channel	—
Per-batch update	3.6.2	1 counter increment per symbol; code reconstruction per batch	$(n_{\mathrm{active}} + 1)^n$ counters per codec	$(n_{\mathrm{active}} + 1)^n$ code lengths per codec, $\times C$ per channel
Downsampling				
Strided subsampling	3.6.1	0	0	—
Nearest-neighbour upsampling	3.6.1	0	W/f_{down} values	—

Table 3.4: Estimated hardware cost of the techniques of this chapter, per channel and in addition to the stage they extend; all costs are replicated C times in the fully unrolled design. F_c is the active set of channel c, $\mathrm{FLOPs}(f)$ and $B(f)$ the arithmetic and buffer of function f (table 3.2), n the Huffman block size and f_{down} the downsampling factor. Upsampling runs on the decompressor side.

3.8.3 Limitations

Function interactions Throughout the exploration, the benefit of a prediction function has depended on the other functions it was combined with: replacing a single function changed how often the remaining ones were chosen (Fig. 3.30), the adaptive selection did not retain the configurations it found (section 3.5), and more context did not lead to better compression (Fig. 3.38). Since every prediction builds on values reconstructed by other functions, the

functions cannot be evaluated in isolation. We have not analysed these interactions further, which makes it difficult to name a clear best choice of prediction functions; a selection that accounts for them is left for future work.

Single connection The whole exploration was carried out on a single, deliberately sensitive connection of one model (section 3.1.7). Design decisions tuned on it need not transfer to other connections and models; chapter 4 evaluates the final design on a wider range of both. Thus this chapter did not discuss the selection of specific design points but rather showed a way of finding good design points for a given model and layer.

Downsampling The error of the downsampling stage follows from the factor and the data and is only measured, not bounded. Restricted to the grid of the featuremap, the factor is also coarse: a 7×7 featuremap only admits the factors 1 and 7 (section 3.6.1). The search fixes the factor on the calibration split, so it is only as reliable as that split is representative of the data at inference; chapter 4, which keeps the downsampling stage, shows a case where it is not.

4 Evaluation

This chapter serves two purposes. First, it tests whether the design points that chapter 3 determined on a single connection hold up on other connections, models and datasets. Second, it compares our compressor, ASZ, with other compression methods, both in terms of the trade-off between compression ratio and accuracy and in terms of hardware cost. Next, we introduce the methodology for the evaluation.

4.1 Evaluation Methodology

The evaluation has two sides. On the first, we compare the methods by their Pareto fronts of compression ratio and top-1 accuracy. The compression ratio serves as a proxy for the bandwidth an evicted connection requires (section 3.1.6): the more a connection is compressed, the more room the optimizer of a streaming accelerator has to improve the system as a whole. On the second side, we estimate the hardware cost of every method, which puts its compression gains into the context of what they cost. What this thesis does not provide are concrete hardware designs, which would be needed to answer which of the methods actually improves the performance of an accelerator the most. The answer to that questions is not easily decidable by looking at either compression ratio or compressor hardware cost or both; different models — when mapped to an FPGA — put different pressure points on the device and different devices have different quantities of resources. Some mappings become bandwidth-bounded, while others are fundamentally BRAM bounded [80].

For the compression side, we evaluate all methods on a number of models and connections (section 4.1.1). The fundamentals of the baseline techniques are explained in chapter 2; in this chapter, we describe how we apply them to the problem framework of section 3.1.1. For the hardware side, we estimate the cost of every method in terms of the arithmetic it requires and the buffer space it needs, as in section 3.1.5, as a proxy for the resources it would occupy in an FPGA design. In addition, we assess qualitatively how well each method suits the streaming scenario (section 4.4.1). Section 4.4 summarizes these estimates in a tabular overview.

The remainder of this section describes the datasets and models (section 4.1.1), the design points of ASZ (section 4.1.2), and the baselines along with how we adapt them to the problem framework (section 4.1.3). Section 4.2 then compares the methods on individual connections, i. e. it asks which method compresses a given connection most effectively in terms of compression ratio and accuracy when that connection is compressed on its own. Section 4.3 compresses several connections of a network at the same time, with the same method but different

configurations on each of them. This moves the evaluation closer to the scenario of SMOF [80], in which multiple skip connections may be evicted. Section 4.4 turns to the hardware side, and section 4.5 brings compression performance and hardware cost together and discusses the results.

4.1.1 Datasets and Models

We evaluate on three image classification datasets: ImageNet [10], CIFAR-10 and CIFAR-100 [45]. On ImageNet, we use the pre-trained torchvision models of RegNetX-400MF, ResNet-18, MobileNetV2 and MobileNetV3-Small; on CIFAR-10 and CIFAR-100, the pre-trained ResNet-20 and MobileNetV2 of `pytorch-cifar-models` [7]. Table 4.1 lists the models, their accuracy baselines and the compressed connections.

Dataset	Model	Top-1 (8-bit)	Compressed skip connections
ImageNet	RegNetX-400MF	71.77 %	`block1-0`, `block2-0`, `block2-1`, `block3-0`, `block3-1`, `block3-2`, `block4-0`, `block4-1`, `block4-3`, `block4-4`, `block4-10`, `block4-11`
ImageNet	ResNet-18	68.35 %	`layer1.0`, `layer1.1`, `layer2.0`, `layer2.1`, `layer3.0`, `layer3.1`, `layer4.0`, `layer4.1`
ImageNet	MobileNetV2	67.87 %	`features.{3, 5, 6, 8, 9, 10, 13, 15, 16}`
ImageNet	MobileNetV3-Small	61.52 %	`features.1.block.1`, `features.3`, `features.{4, 6, 8, 9, 10}.block.2`, `features.10`, `features.11`
CIFAR-10	ResNet-20	92.48 %	`layer{1, 2, 3}.{0, 1, 2}`
CIFAR-10	MobileNetV2	93.81 %	`features.{3, 5, 6, 8, 9, 10, 13, 15, 16}`
CIFAR-100	ResNet-20	68.33 %	`layer{1, 2, 3}.{0, 1, 2}`
CIFAR-100	MobileNetV2	74.28 %	`features.{3, 5, 6, 8, 9, 10, 13, 15, 16}`

Table 4.1: Datasets, models, accuracy baselines on the test split and compressed skip connections of the evaluation. Connection names follow the module names of the respective model.

As in chapter 3, all weights and activations are quantized to 8 bits, $x \in \{0, \dots, 255\}$, and the accuracy baseline of every network is its 8-bit model without compression. Every channel c of a connection is compressed as one stream in raster order, with the images of a batch concatenated.

Splits Every dataset is split into a *calibration split*, on which everything fitted is computed — quantization parameters, profiled quantities and the parameter search of ASZ — and a disjoint *test split*, on which all results of this chapter are reported. On ImageNet, the calibration split holds 5,000 images of the validation set, five per class, as in chapter 3; the test split is a class-balanced sample of 10,000 of the remaining 45,000 images. On CIFAR, the calibration split holds 5,000 class-balanced images of the training set, and the test split is the full test set of 10,000 images. Since the networks reach almost 100 % accuracy on their training set, the CIFAR calibration split is less representative of the test split than the ImageNet one (section 4.1.2).

Connections Per network, we compress up to 9 skip connections, selected from the sensitivity of each connection to quantization (section 4.1.3.6): the most sensitive ones, the earliest and the latest, and representatives in between. ResNet-18 has only 8. On RegNetX-400MF, we use 12 of its 22 skip connections, covering all four stages.

4.1.2 DSE Design Points

We evaluate the final design of chapter 3 (section 3.8.1), which we refer to as ASZ; the legends of the plots in this chapter label its two variants (see below)

ASZ K=1 static and *ASZ K=1 dynamic*. It compresses every channel separately with linear blocks and both error thresholds (section 3.3), uses a context of a single buffered row ($K = 1$), and Huffman-codes the code stream in blocks of $n = 2$ codes (section 3.6.2). The small context favours a few cheap prediction functions and thereby skews the code distribution, which the Huffman stage exploits (section 3.6.2).

The parameters ($N_{Subdiv}, \tau_b, \tau_v$) and, where it holds the budget, a downsampling factor are chosen per connection by the parameter search of section 3.7.2 on the calibration split. Its Pareto-optimal configurations are then rerun on the test split and completed towards lossless by scaling their thresholds back. We report two variants of the function selection (section 3.5): *dynamic*, which keeps adapting during the test run, and *static*, which freezes the selection found on the calibration split and is considerably cheaper in hardware, since only the selected functions have to be instantiated.

4.1.3 Baselines

For every baseline, we describe how it is applied to the 8-bit channel streams of our setting: what it needs from profiling on the calibration split, what it does during inference, how it handles multiple channels, and which parameter is swept. Side information that is constant at deployment time, such as code tables or bases, is not counted in the compression ratio.

We order the baselines by how they reduce the data, which also determines how far they can compress (table 4.2). The *lossless* methods code the values themselves. *Decimation*, i. e. downsampling, stores fewer values. The *transforms*, SVD and JPEG, project every featuremap or block onto a fixed basis and code the coefficients, and a coefficient that is dropped costs nothing. Section 2.5.2.3 introduces SVD as a low-rank approximation; we count it as a transform, since its bases are learned once per channel and fixed at deployment, which makes it a separable transform fitted to the data, in the spirit of the Karhunen–Loève transform that the DCT of JPEG approximates without any data [2]. The *quantizers*, QH and Lorenzo, code one quantized symbol per value, the value itself or its prediction residual, and thus a part of its magnitude. The *prediction-based* compressors, SZ1 and ASZ, code per value only which of a few predictors reconstructs it within the error threshold, and store the remaining values raw. Lorenzo also predicts, but it codes the quantized residual rather than the choice of a predictor, which places it with the quantizers.

Group	Methods	What is coded	Ratio limit	Error control
Lossless	Huffman, RLE, zero RLE	the values: one codeword per value, or runs of equal values or zeros	Huffman: 8, at least one bit per value; RLE: the run structure	none
Decimation	Downsampling	one average per $f_{down} \times f_{down}$ window, at 8 bits	HW, one value per channel	indirect, via f_{down}, in steps of f_{down}^2
Transform	SVD, JPEG	the coefficients of a fixed basis per featuremap (SVD) or 8×8 block (JPEG)	none per value; dropped coefficients cost nothing	indirect, via the rank k or the quality; no bound per value
Quantization	QH, Lorenzo	one quantized symbol per value: the value (QH) or its prediction residual (Lorenzo)	8, at least one bit per value	per value: $s_c/2$ within the profiled range (QH), ε_c (Lorenzo)
Prediction	SZ1, ASZ	one 2-bit code per value: the predictor that reconstructs it within the threshold, or an outlier stored raw	4 for the codes, 16 after Huffman coding of pairs; ASZ more only through downsampling	per value (τ_v); for ASZ also per block (τ_b)

Table 4.2: The compressor groups of the evaluation: what each group codes, the ratio its design allows at most, and how its error is controlled. The ratio limits follow from the coding: Huffman coding of single symbols spends at least one bit per symbol, of pairs at least half a bit per code.

4.1.3.1 Huffman

Huffman coding is applied losslessly to the 8-bit activation values themselves, without a lossy stage, and therefore yields a single point per connection. During profiling, we collect the value histogram of every channel over the calibration split and build a static Huffman code from it. Since the value distributions differ between channels (section A.1), every channel c gets its own code table. During inference, every value is replaced by its codeword through a table lookup, without any buffering and at one value per cycle. We code single values ($n = 1$): blocks of n values would need an alphabet of 256^n symbols, already 65 536 entries per table at $n = 2$ (section 2.5). The side information consists of C tables of up to 2^8 codewords, so its cost grows linearly with the channel count, e. g. $C = 400$ in late RegNet layers.

4.1.3.2 RLE

RLE is applied losslessly to the 8-bit values of every channel stream in raster order. Each run of L equal values is stored as (value, length) records with $d = 8$ value bits and a b_c-bit length field, which stores the length biased by one and thus covers $1, \ldots, 2^{b_c}$. Longer runs are split:

$$\text{bits}(L) = \left\lceil \frac{L}{2^{b_c}} \right\rceil (b_c + d).$$

Instead of sweeping $b \in [1, 8]$, we choose the length-field width per channel from profiling. We measure the mean chain length μ_c of every channel on the calibration split and model the run lengths as geometric with $q_c = 1 - 1/\mu_c$, the standard model for choosing the parameter of a run-length code [26]. A run then needs more than k records, i. e. is longer than $k\,2^b$ values, with probability $q_c^{k2^b}$, so that it takes $\sum_{k \geq 0} q_c^{k2^b} = 1/(1 - q_c^{2^b})$ records on average. The width minimizes the resulting expected bits per run including splits,

$$b_c = \arg\min_{b \geq 1} \frac{b + d}{1 - q_c^{2^b}}.$$

The optimum lies above $\log_2 \mu_c$, at $2^{b_c} \approx 2\text{–}4\,\mu_c$, since an overflow re-sends the whole value. A Golomb code [93], whose overflow costs a single bit, chooses

its parameter m with $q_c^m \approx 1/2$, i. e. $m \approx 0.7\,\mu_c$, much shorter [26]. During inference, every channel is its own stream with one comparator against the previous value and a run counter; a record is emitted when the value changes or the counter overflows. The achievable ratio is bounded by the run structure, $\mathrm{CR} \approx 8L/(8+b)$ for a mean chain length L. With a median chain length of 1.08 values per run over the evaluated connections, RLE expands 60 of 93 of them (appendix A.11.1, Fig. A.50).

In the zero RLE variant [95], only runs of zeros are counted: every non-zero value is sent as a (zeros before it, value) record, and longer zero runs spill into (max, 0) records. Zero is code 0 after the channel shift, i. e. the channel's minimum, which is the real zero of post-ReLU activations. Here, the width is chosen by exact count rather than a run-length model: with the record layout of section 2.5.1.1, the cost of a stream is its number of records times the record width, so that counting the records on the calibration split gives the exact cost of every width, and a search over b replaces the model. Profiling counts, per channel, the records R_c an unbounded run field needs and, for every width $b \leq 16$, the extra spill records $S_c(b)$, and we choose

$$b_c = \arg\min_{b \geq 1} \left(R_c + S_c(b)\right)(b+d).$$

Since every non-zero value costs a full record of $8+b$ bits, zero RLE only breaks even above a zero fraction of about 15 %. 41 of the 74 evaluated connections contain no zeros at all, since their operand is not a ReLU output but a projection or a linear bottleneck (all of MobileNetV2), and are expanded to $\mathrm{CR} = 8/9$ (appendix A.11.1, Fig. A.49). Appendix A.11 compares RLE and zero RLE per connection.

4.1.3.3 Downsampling

Downsampling decimates every featuremap spatially, without further coding; the stored values keep their 8 bits. We use average pooling by a factor f_{down} in both dimensions and bilinear upsampling on decompression (section 2.5), which is slightly more accurate than strided subsampling with nearest-neighbour upsampling (appendix A.13). No profiling is needed. The swept parameter is the factor f_{down}, which has to divide both H and W of the layer and fixes the ratio at $\mathrm{CR} = f_{down}^2$ (table 4.3). The same f_{down} is used for all channels of a layer, each of which is downsampled independently. During inference, the compressor accumulates the partial sums of the W/f_{down} windows of the current row, and the decompressor interpolates from two buffered rows of the reduced featuremap.

$H \times W$	f_{down}	CR
56×56	2, 4, 7, 8, 14, …	4, 16, 49, 64, 196, …
28×28	2, 4, 7, 14, 28	4, 16, 49, …
14×14	2, 7, 14	4, 49, 196
7×7	7	49
32×32 (CIFAR)	2, 4, 8, 16, 32	4, 16, 64, …
8×8 (CIFAR)	2, 4, 8	4, 16, 64

Table 4.3: Admissible downsampling factors f_{down} and the resulting compression ratios for common featuremap sizes.

The steps are coarse: the smallest one is already $\mathrm{CR} = 4$, and maps with odd or prime side lengths allow only large factors; a 7×7 map only admits $f_{down} = 7$,

i. e. one value per channel. Downsampling is therefore only usable on large, insensitive featuremaps or with a relatively large accuracy budget.

4.1.3.4 SVD

SVD compresses every channel's $H \times W$ featuremap by a rank-k projection onto fixed bases. During profiling, we learn one pair of shared bases per channel, $U_c \in \mathbb{R}^{H \times k}$ and $V_c \in \mathbb{R}^{W \times k}$, from the SVD of all calibration featuremaps of that channel, stacked side by side for U_c and on top of each other for V_c. A second pass measures the range of every coefficient position with the final bases. The bases are thus learned offline. During inference, no decomposition takes place: the compressor computes $C = U_c^\top A\, V_c$, and the decompressor $\hat{A} = U_c\, C\, V_c^\top$. The coefficients are requantized to 8 bits on a fixed grid per coefficient position, taken from the profiled range, and values outside it are clamped. Since $U_c^\top$ mixes all rows, the whole featuremap of a channel has to be buffered before the first coefficient is complete. The ratio is HW/k^2, and the swept parameter is the rank k, from full rank downwards. The side information consists of the bases, $C\,(H + W)\,k$ values, and the coefficient ranges.

4.1.3.5 JPEG

JPEG is used off the shelf (libjpeg via `torchvision`) as the whole pipeline of 8×8 DCT, quantization with the standard luminance table and Huffman coding (section 2.5.1.3); it needs no profiling. Every sample is coded on its own as an 8-bit greyscale image, in which the C channel tiles are laid out as one near-square mosaic, so that all channels share headers and entropy coder. Each tile is edge-padded to a multiple of 8, so that no DCT block mixes two channels. The padding is coded and charged, but cropped on decoding; on 7×7 maps, 64 values are coded for every 49. Since the 8×8 blocks only start after 8 rows of the mosaic have arrived, these have to be buffered. The per-image headers with the quantization and Huffman tables are constant for a fixed quality and not counted. The swept parameter is the quality $\in \{95, 85, \ldots, 5\}$.

4.1.3.6 Quantization + Huffman

This baseline requantizes every channel from 8 to b bits and Huffman-codes the resulting codes. Profiling provides the value range $[m_c, M_c]$ and the value histogram of every channel. During inference, every value is mapped onto a fixed asymmetric linear grid on the profiled range, and values outside it are clamped:

$$s_c = \frac{M_c - m_c}{2^b - 1}, \qquad q = \mathrm{clamp}\Big(\,\mathrm{round}\Big(\frac{x - m_c}{s_c}\Big), 0, 2^b - 1\Big), \qquad \hat{x} = m_c + s_c\, q.$$

This takes one multiplication, a rounding and a table lookup per value, without buffering. Every channel has its own grid (m_c, s_c), while all channels share one static Huffman table per layer, built from the calibration histograms mapped through each channel's quantizer and summed over the channels. The swept parameter is $b = 8, 7, \ldots, 0$, where $b = 0$ drops the connection.

4.1.3.7 Lorenzo + Huffman

This baseline follows the pipeline of modern SZ versions [76]: the Lorenzo predictor of section 3.4.2.4 as the *only* prediction function, quantized residuals and Huffman coding. We use a single layer ($n = 1$), since the one-layer predictor dominated the two- and three-layer variants on every one of ten RegNetX-400MF

connections (appendix A.12). Profiling provides the value range $[m_c, M_c]$ of every channel. During inference, the prediction is computed from reconstructed values, clamped into $[m_c, M_c]$ and rounded to the integer grid, which requires the previous n reconstructed rows of the channel. The residual is quantized with an error-bounded linear-scaling quantizer with a relative bound ρ,

$$\varepsilon_c = \rho\,(M_c - m_c), \qquad q = \operatorname{round}\!\left(\frac{x - \hat{x}}{2\varepsilon_c}\right), \qquad x = \hat{x} + 2\varepsilon_c\, q,$$

where ε_c is snapped down to $n + \frac{1}{2}$ code steps and $\rho = 0$ is lossless. Residuals with $|q| \le 2^{w-1} - 1$ ($w = 8$) become symbols; otherwise, an outlier symbol is emitted and the raw value is stored at 8 bits. Every channel is predicted and quantized independently with its own ε_c, while the symbols of all channels share one Huffman table per layer. The table starts uniform and is rebuilt after every batch from the cumulative symbol counts; outliers are not entropy coded. The swept parameter is $\rho \in \{0, 0.01, 0.02, 0.03, 0.05, 0.07, 0.10, 0.15\}$.

4.1.3.8 SZ1 + Huffman

This baseline is SZ1 as in section 3.2, with constant, linear and polynomial prediction, outlier code 0 and only a value error threshold, followed by Huffman coding of the code stream. As for Lorenzo, profiling provides the value range of every channel. During inference, the three functions predict from the last three reconstructed values of the channel; the first one within τ_v is coded, otherwise the value is stored raw at 8 bits. Every channel keeps its own predictor state, while the codes $\{0, 1, \ldots, n_{\text{active}}\}$ of all channels share one Huffman table per layer (section 3.6), which is adapted after every batch from the cumulative code counts, starting uniform. We code blocks of $n = 2$ successive codes ($4^2 = 16$ symbols), counted on the data so that the correlation between neighbouring codes is captured. The swept parameter is τ_v.

4.1.4 Evaluation Setup and Presentation of Results

All results are reported as top-1 accuracy on the test split (10,000 images) of the respective dataset. The accuracy baseline of every network is its 8-bit model without compression, and the accuracy budget is 1 pp below it.

The lossless methods (Huffman, RLE, zero RLE) yield a single point per connection. The lossy baselines are parameter sweeps over the parameter given in section 4.1.3, run directly on the test split. ASZ runs the trajectory search (section 3.7.2) on the calibration split (5,000 images; for CIFAR, taken from the training set) and keeps at most 8 Pareto points per connection. These are rerun on the test split in two variants: *static*, with the per-channel selection frozen to the calibration batch with the best SZ+Huffman ratio (excluding the first batch), and *dynamic*, with the selection adapting on the test split. Both are extended by a scale-back of their thresholds towards lossless, and every point keeps the downsampling factor the search fixed for its connection (section 4.1.2). On individual connections, ASZ uses a context of $K = 1$.

The results are presented in three kinds of figures:

Pareto fronts per connection One panel per connection, top-1 accuracy against the compression ratio on a logarithmic axis, one Pareto front per method. Fronts continue past the frame beyond the budget; the legend counts only the points within it. Dashed horizontal line: baseline; dotted red line: budget; vertical lines: the lossless methods.

Best ratio per connection For every method, the highest compression ratio within the budget on each connection, connections in order of depth, logarithmic ratio axis. A cross marks a connection on which a method has no point within the budget. Table A.1 in appendix A.14 lists these ratios for all connections.

Performance profiles For every connection, each method is represented by its highest ratio within the budget, and τ is the factor by which it falls short of the best ratio any method reached on that connection. A curve gives the fraction of connections with a shortfall of at most τ; a connection without a point of a method within the budget counts as a failure, so curves may stay below 100 %.

4.2 Individual Layer Compression

In this section, every connection is compressed on its own, while all other connections of the network remain uncompressed. For every connection, we ask which method reaches the highest compression ratio within the accuracy budget.

4.2.1 ImageNet

4.2.1.1 RegNetX-400MF

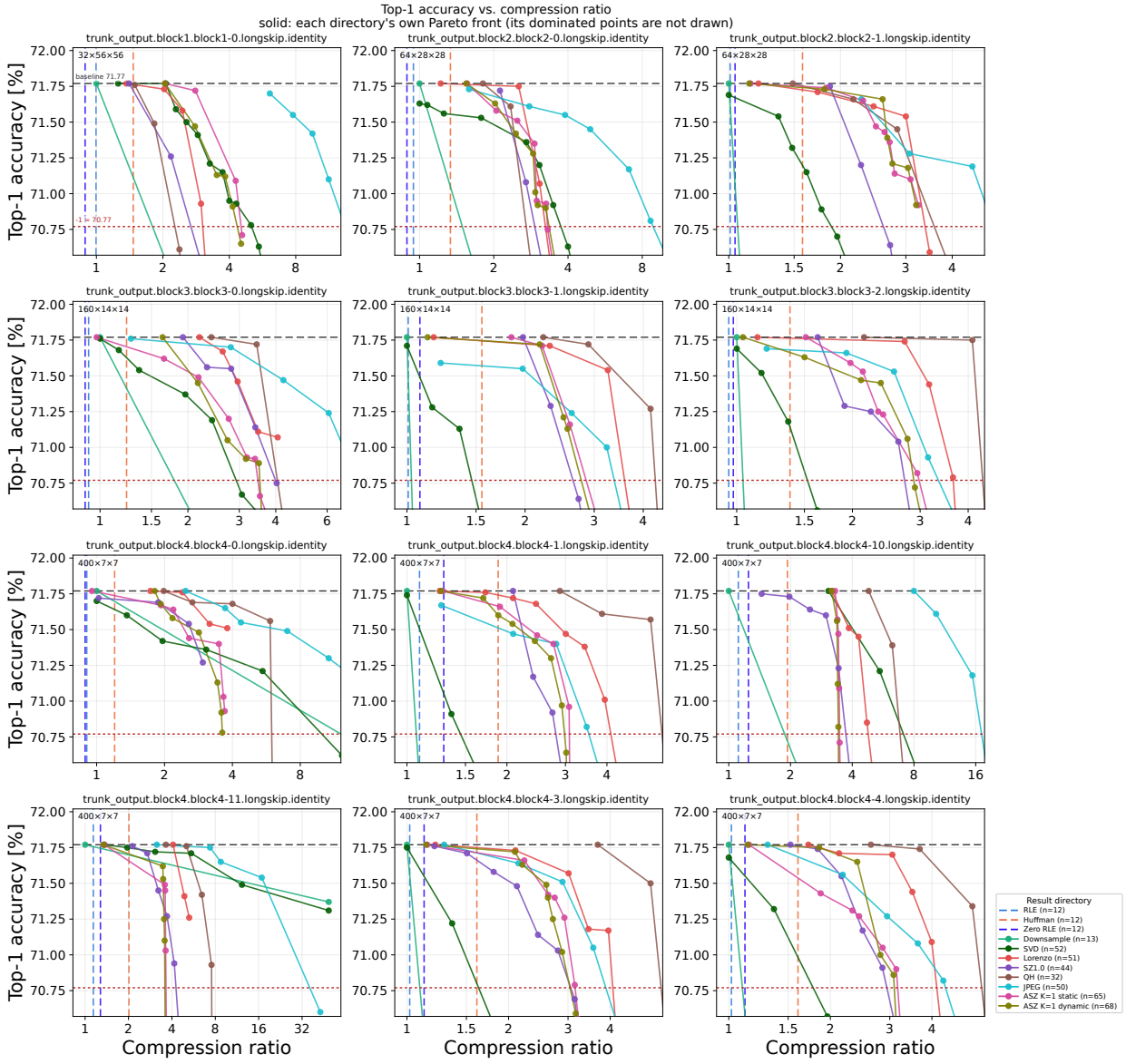


Figure 4.1: RegNetX-400MF, ImageNet, 12 of its 22 skip connections: Pareto fronts per connection (section 4.1.4), baseline 71.77 %. ASZ without downsampling, as the search chose factor 1 on every connection.

Figure 4.1 shows the Pareto fronts of all methods on the twelve connections of RegNetX-400MF, and Fig. 4.2 the best ratio of every method within the budget.

Lossless baselines Huffman coding reaches 1.2 to 2 and is clearly ahead of the other lossless methods. RLE stays around 1 and expands several connections; zero RLE is generally ahead of RLE (appendix A.11), but remains far below Huffman coding.

Decimation Downsampling is unusable on almost every connection: its smallest step is already 4:1 (table 4.3), which none of these connections tolerates within 1 pp. Only the insensitive block4-11 can be reduced to a single value per channel, i. e. by a factor of 49.

Transforms SVD and JPEG rise and fall on the same connections, and their ratios depend on the connection rather than on the size of its featuremap. SVD stays below 2 on half of the connections, where the best methods reach 4.1 to 5.4; on block3-1, block4-1, block4-3 and block4-4, it falls even below Huffman coding (on block3-2, it is level with it), which makes it, apart from downsampling, the only lossy method that is dominated by a lossless one. Like downsampling, it reduces block4-11 to a single value per channel. JPEG is the best method on the first four connections, on block4-0 and on block4-10, with ratios of 4.5 to 15.4, but falls to 3.2 to 4.3 on block3-1, block3-2, block4-1, block4-3 and block4-4, behind QH and, except on block4-4, Lorenzo. It has no near-lossless operating point: on some connections, even the highest quality loses accuracy (block3-1: about 71.6 %), so that it offers no operating point for budgets much tighter than 1 pp. Padding the 7×7 maps to 8×8, i. e. coding 64 values for every 49, does not keep it from leading on several of them.

Quantization QH is very consistent, and its ratio rises steadily with depth, from about 1.8 on block1-0 to 7.5 on block4-11. It is limited to about 8:1: at 1 bit, Huffman coding of single symbols cannot spend less than one bit per value. Block codes as in ASZ would lift this limit, but we code single symbols for quantization (section 4.1.3.6). Lorenzo reaches about 3 on the first three connections, including block1-0, the largest featuremap, 3.3 to 4.7 on the following ones and 5.3 on block4-11. It is ahead of QH on the first four connections and behind it on all others, and from the fourth connection on, its residual quantization keeps it ahead of SZ1 and ASZ.

Prediction-based compressors SZ1 is dominated by ASZ and by the quantizers almost everywhere. Static ASZ reaches 2.6 to 4.3 and lies above SZ1 on almost every connection, so that error bounding and per-channel selection pay off against SZ1, but not against Lorenzo. The lead over SZ1 comes entirely from the SZ stage: at the best points within the budget, the SZ stage of ASZ alone reaches a median of 3.0 against 2.2 for SZ1, while the Huffman stage on top adds a median factor of only 1.08 for ASZ, against 1.27 for SZ1. The per-channel selection favours predictors that complement each other and thus leave few outliers, which spreads the hits evenly over the codes (median code entropy 1.8 against 1.2 bits per code) and leaves Huffman coding little to exploit (section 3.6.2). SZ1 codes the first of its fixed functions that is within τ_v, so that the constant function, which comes first, collects a median 70 % of all codes; this skew is where Huffman coding pays off. Static ASZ is ahead of Lorenzo only on the first three connections (block1-0: 4.3 against 3.0) and never the best method overall. On block4-10 and block4-11, its fronts turn into near-vertical walls at a ratio of about 3.3 to 3.7: the ratio saturates while the accuracy keeps falling (on block4-11, it reaches 4.5 only at a loss of 4 pp), since the 2-bit code stream limits the SZ stage to 4:1 and Huffman coding with $n = 2$ adds little on top (section 3.6.2). The search chose no downsampling on any connection. Dynamic ASZ is practically identical, slightly lower on most connections and never clearly better. This confirms the finding of section 3.5 that a fixed configuration dominates the adaptive scheme, which in addition costs about four times the arithmetic (table 4.5).

Summary JPEG leads on the early, large connections, on block4-0 and on block4-10, QH on the middle ones. Lorenzo, SZ1 and ASZ cluster between about 2.2 and 5; only Lorenzo exceeds 5, on block4-11 (5.3). Block4-11 is a special case: SVD and downsampling keep little more than the average of each channel and still stay within the budget, so this connection contributes little to the result of the network. The other methods run into their limits there, the SZ-type compressors at 3.6 to 5.3 and QH at 7.5, just below its 8:1 limit, while JPEG reaches 16.8 at a loss of only 0.2 pp, a third of the ratio of SVD and downsampling.

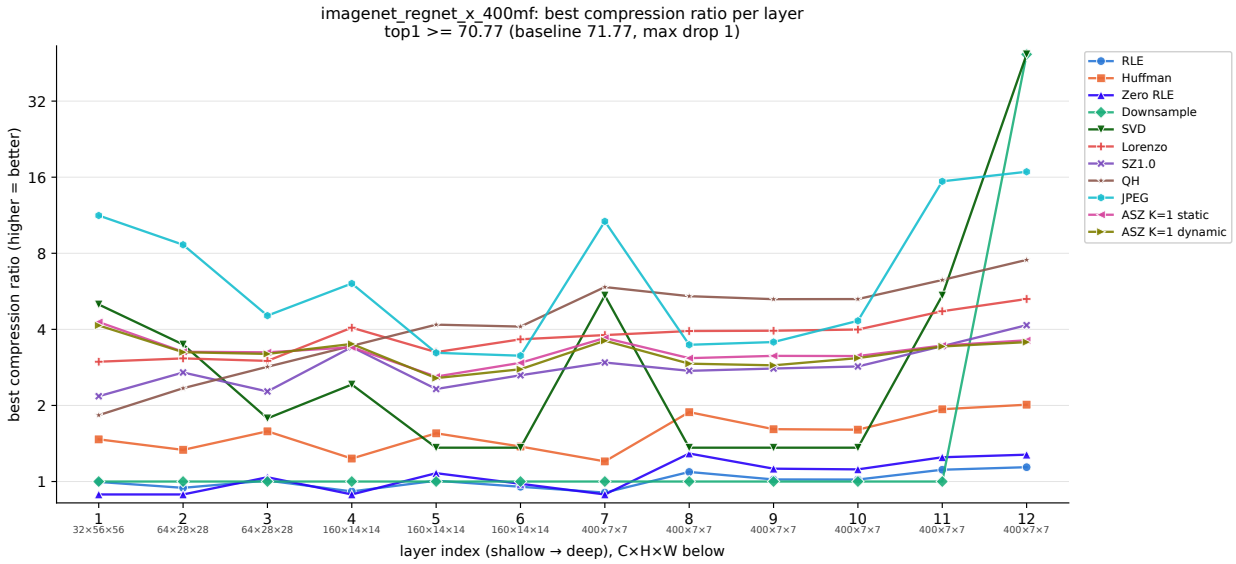


Figure 4.2: RegNetX-400MF, ImageNet: best compression ratio per connection within the 1 pp budget (top-1 at least 70.77 %), same results as Fig. 4.1.

The ratios of SVD and downsampling roughly follow the accuracy loss when a connection is cut entirely (appendix A.9.1): the connections that carry the output of a projection at the start of a stage retain part of the accuracy when cut, block4-0 hardly loses any, and the late connections are less relevant and thus tolerate more approximation, which increases their compressibility. Cutting any of the other six connections, block2-1, block3-1, block3-2, block4-1, block4-3 and block4-4, makes the network fail entirely, and these are exactly the six on which SVD stays below 2 and JPEG below 5. JPEG follows the same progression as SVD, since lower qualities zero more DCT coefficients, which its run-length coding compresses very effectively. QH, in contrast, rises steadily with depth, by a factor of 3.4 from block1-0 to block4-10, and Lorenzo and SZ1 rise by about 1.6: quantizing a connection to 3 bits costs 4.7 pp on block1-0, but at most 0.5 pp from block3-0 on (appendix A.9.2). ASZ, on the other hand, falls slightly with depth. Its lead over SZ1 shrinks from a factor of 2.0 on block1-0 to 1.0 on block4-10, so that its per-channel selection gains the most on the large early featuremaps, and its 2-bit codes cap it at about 4:1 before Huffman coding.

4.2.1.2 ResNet-18

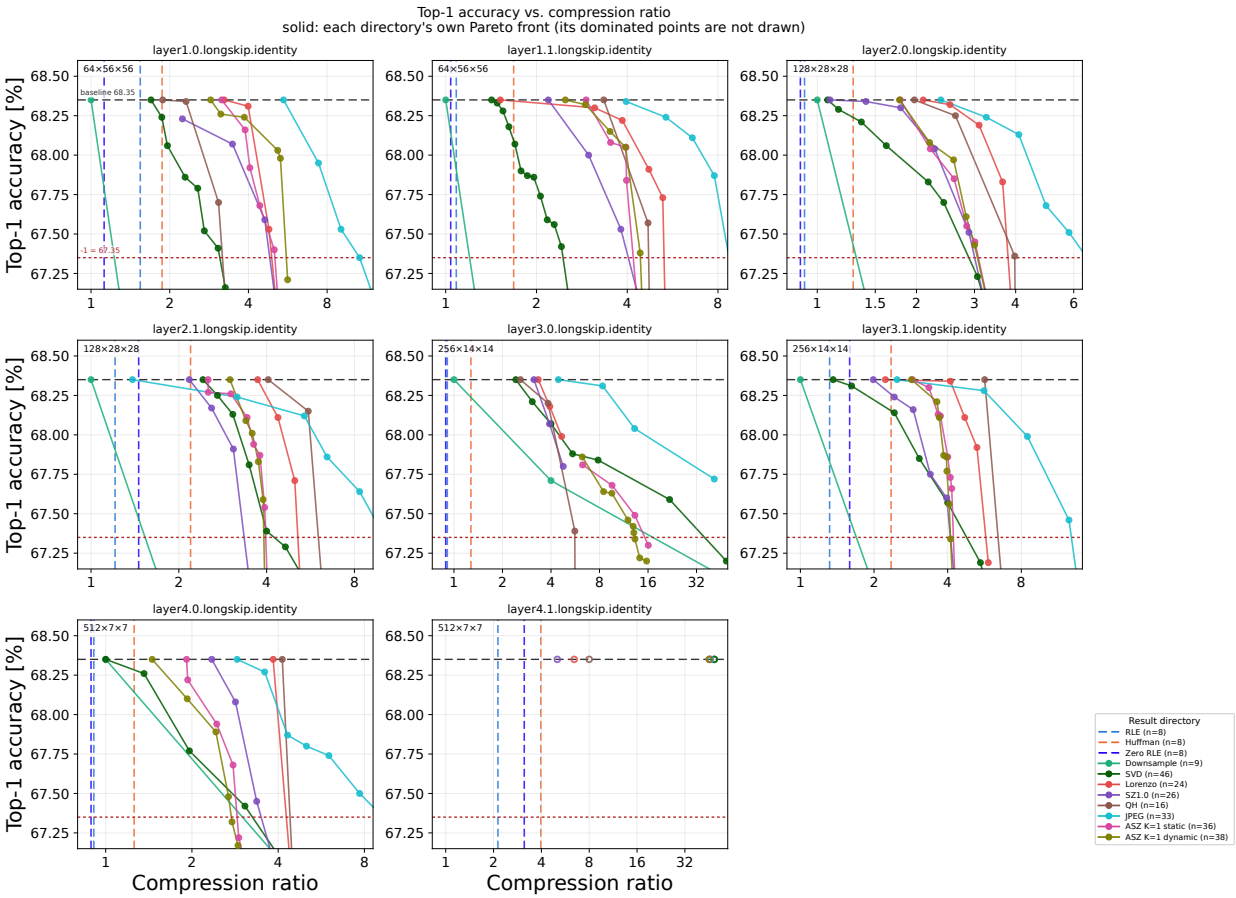


Figure 4.3: ResNet-18, ImageNet, 8 skip connections: Pareto fronts per connection (section 4.1.4), baseline 68.35 %.

Figures 4.3 and 4.4 show the results for ResNet-18. Its last connection, layer4.1, behaves differently from the others for every method and is discussed separately below; the following paragraphs cover the first seven connections.

Lossless baselines Huffman coding reaches 1.3 to 2.4 and is ahead of both RLE variants everywhere. The connections at the start of stages 2 to 4, layer2.0, layer3.0 and layer4.0, carry the output of a projection and contain no zeros (appendix A.11.1): there, Huffman coding stays at about 1.3, RLE expands them to 0.91 and 0.92, and zero RLE to 8/9. The other connections carry the ReLU output of the previous block with 20 to 55 % zeros, and zero RLE overtakes RLE on the two sparsest of them, layer2.1 (1.46 against 1.21) and layer3.1 (1.59 against 1.32).

Decimation Downsampling finds a factor within the budget only on layer3.0 (2, i. e. 4:1), the least sensitive of the seven connections: cutting it entirely costs only 1.8 pp (appendix A.9.1), against 8 to 68 pp for the others.

Transforms JPEG is the best method on all seven connections, with 5.8 to 12.5 on six of them and 41 on layer3.0. Its highest qualities reach the baseline accuracy on every connection (68.34 to 68.59 %), so that, unlike on RegNetX-400MF, it also offers operating points for budgets tighter than 1 pp. SVD reaches 2.4 to 4 on six connections and 21.8 on layer3.0. Both thus peak on the least

sensitive connection, and both are at their lowest on layer2.0, where SVD ties with layer1.1.

Quantization QH reaches 3.1 to 5.7 and rises with depth, as the sensitivity to quantization falls: 3 bits cost 4.4 pp on layer1.0, but at most 0.2 pp from layer2.1 on (appendix A.9.2). It is ahead of Lorenzo from layer2.0 on and second behind JPEG on layer2.0, layer2.1, layer3.1 and layer4.0. Lorenzo reaches 3.7 to 5.3 and is nearly flat over depth; it leads QH on the two 56×56 connections of the first stage, in line with its gain on large featuremaps (section 4.5). Both dip on the projection connections layer2.0 and layer4.0. Without zeros, the 3-bit codes of QH are spread more evenly there (code entropy 1.9 to 2.0 bits, against 1.2 to 1.5 on the other connections at 3 bits), and Huffman coding gains less (a factor of 1.5, against 1.8 to 2.1).

Prediction-based compressors Static ASZ reaches 2.8 to 5.0 on the six connections without downsampling and is ahead of SZ1 (2.9 to 4.8) on all of them except layer4.0 (2.78 against 3.37). As on RegNetX-400MF, it does best on the large connections of the first stage. On layer1.0, it is ahead of Lorenzo (5.02 against 4.79), and the dynamic variant reaches 5.30 there and 4.42 on layer1.1, its only clear gains over the static one; on the other connections, it is slightly lower, by up to 0.2. On layer3.0, the search chose downsampling by 2, and static ASZ reaches 13.3; since the factor is fixed, its most accurate point there lies 0.54 pp below the baseline.

Downsampling within ASZ On layer3.0, and on layer4.1 below, the ratio of ASZ is mostly that of its downsampling: on layer3.0, the SZ and Huffman stages contribute a factor of 3.3 on top of the 4:1 of the downsampling, about what ASZ reaches without downsampling on the other connections. This makes the comparison unfair to the other compressors. Downsampling is a preprocessing step that any of them could use, but we evaluated the baselines at full resolution only. With the same preprocessing, they would likely have gained in the same way as ASZ; Lorenzo, for instance, reaches 4.7 on layer3.0 at full resolution, more than the factor of 3.3 that the SZ and Huffman stages of ASZ add. The lead of ASZ on such connections therefore belongs to the downsampling rather than to the SZ stage. The same holds for the connections of ResNet-20 on which the search chose downsampling (sections 4.2.2.2 and 4.2.3.2).

The last connection Layer4.1, the last connection before the global average pooling, differs for every method. Cutting it entirely costs no accuracy at all (the network even gains 0.35 pp), so that the ratio a method reaches there is limited only by its design. Downsampling and SVD reduce it to a single value per channel (49), and JPEG reaches 46. ASZ reaches 45.8 only through the downsampling factor of 7 its search chose, which also leaves one value per channel; its SZ stage stores this value raw, as an outlier, so that the SZ and Huffman stages cost a factor of 0.94 on top of the downsampling. QH reaches its limit of 8 at 1 bit per value, Lorenzo 6.4 and SZ1 5.0, their highest ratios on this network but far below those of the methods that can drop values. With 80 % zeros, layer4.1 is also the sparsest connection, and the lossless methods reach their highest ratios here as well: Huffman coding 4.0, zero RLE 3.1 and RLE 2.1.

Summary JPEG leads on all of the first seven connections, followed on the first stage by Lorenzo and ASZ and from layer2.0 on by QH; on the insensitive layer3.0, SVD and, through its downsampling, ASZ come second and third. On layer4.1, every method reaches its highest ratio on this network, and those that can drop values reduce the connection to almost nothing.

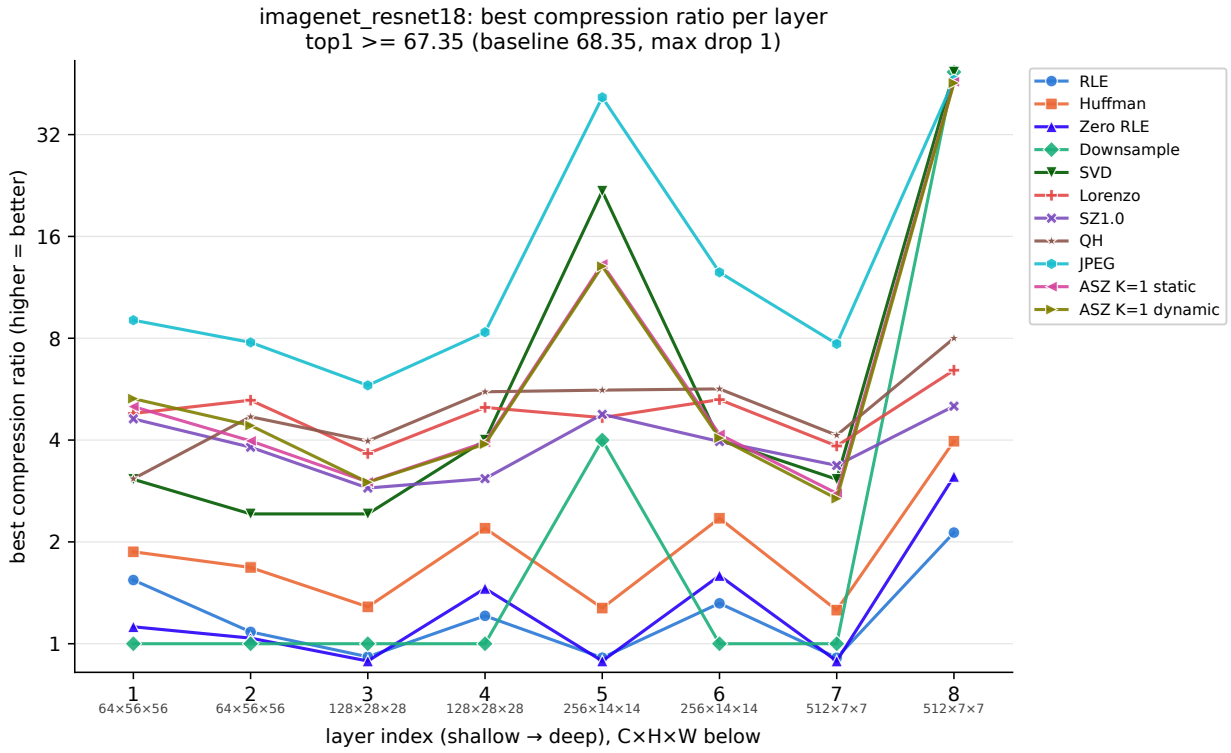


Figure 4.4: ResNet-18, ImageNet: best compression ratio per connection within the 1 pp budget (top-1 at least 67.35 %), same results as Fig. 4.3.

4.2.1.3 MobileNetV2

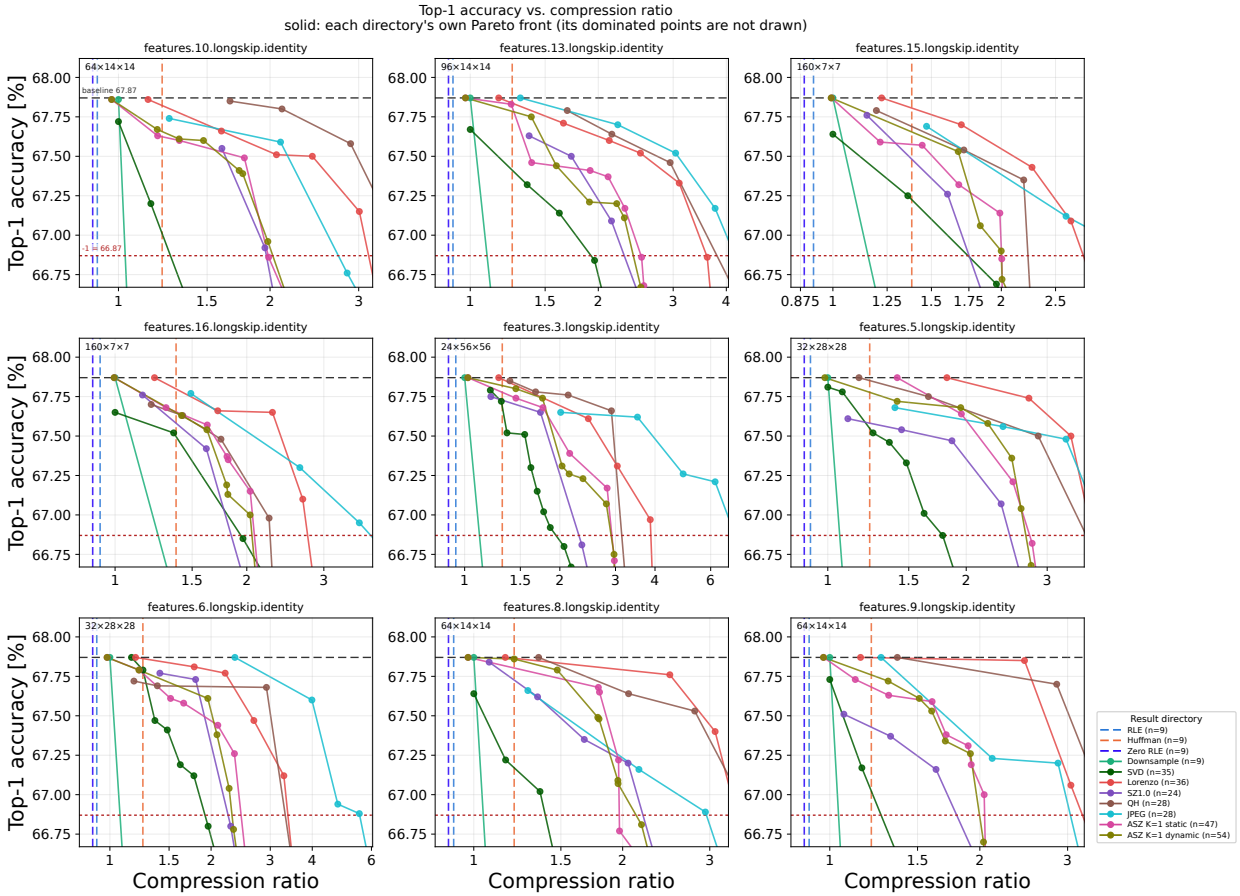


Figure 4.5: MobileNetV2, ImageNet, 9 skip connections: Pareto fronts per connection (section 4.1.4), baseline 67.87 %.

Figures 4.5 and 4.6 show the results for MobileNetV2 on ImageNet. Its connections add the outputs of linear bottlenecks, i. e. of projections without a ReLU [67], so that none of them contains zeros (appendix A.11.1). None of them is insensitive either: cutting a connection costs 47 to 68 pp (appendix A.9.1), and quantizing it to 3 bits 1.4 to 9.2 pp (appendix A.9.2). No method exceeds 6.2 within the budget on any connection.

Lossless baselines Without zeros or runs, RLE expands every connection (0.91 to 0.94), and zero RLE expands all of them to 8/9. Huffman coding is the only lossless method above 1, with 1.21 to 1.38.

Decimation Downsampling finds no factor within the budget on any connection.

Transforms JPEG is the best method on four connections (features.3: 6.2, features.6: 5.5, features.13: 3.8, features.16: 3.6) and reaches 2.1 to 3.3 on the others, behind Lorenzo and, on features.10, also behind QH. SVD stays below 2 everywhere (1.16 to 1.87), with its highest ratio on features.3 and its lowest on features.9 and features.10.

Quantization Lorenzo is the best method on the other five connections and reaches 2.66 to 3.86, the most on features.3, the largest featuremap. It is ahead of

QH (2.19 to 2.95, flat over depth) on all nine connections, unlike on RegNetX-400MF and ResNet-18, where QH leads on most connections. QH has to stay at 4 bits on every connection, since 3 bits cost 1.4 to 9.2 pp, and its maximum error doubles from one step to the next, from about 3.3 % of the channel range at 4 bits to 7.1 % at 3 bits. The error bound of Lorenzo has a setting in between, $\rho = 0.05$, which holds the budget on features.3 to features.13 and gives Lorenzo its lead there: at a similar maximum error of about 3 %, QH compresses these connections about as well as Lorenzo or better (2.80 to 2.95 against 2.43 to 3.04). Only on the two 7×7 connections, features.15 and features.16, does Lorenzo code more efficiently at the same bound (2.66 and 2.69 against 2.19 and 2.25). Without a spike at zero, the 4-bit codes of QH are spread over many symbols, so that its Huffman stage adds only a factor of 1.10 to 1.48.

Prediction-based compressors SZ1 reaches 1.60 to 2.38 and has no near-lossless point: its most accurate points lie 0.03 to 0.36 pp below the baseline. Static ASZ reaches 1.78 to 2.82; the search chose no downsampling. It is above SZ1 on seven of nine connections (below on features.8: 1.96 against 2.05, and features.10: 1.78 against 1.95), but below Lorenzo and QH on all nine. Even beyond the budget, its fronts end at 2.2 to 4.4, where those of Lorenzo reach 4.6 to 5.2; at the other end, it keeps a point at the baseline accuracy on every connection. Dynamic ASZ stays within 0.2 of the static variant on every connection (features.10: 1.98 against 1.78, features.9: 1.91 against 2.04).

Summary Lorenzo and JPEG share the lead, Lorenzo on five connections and JPEG on four, and SZ1 and ASZ trail both quantizers on every connection.

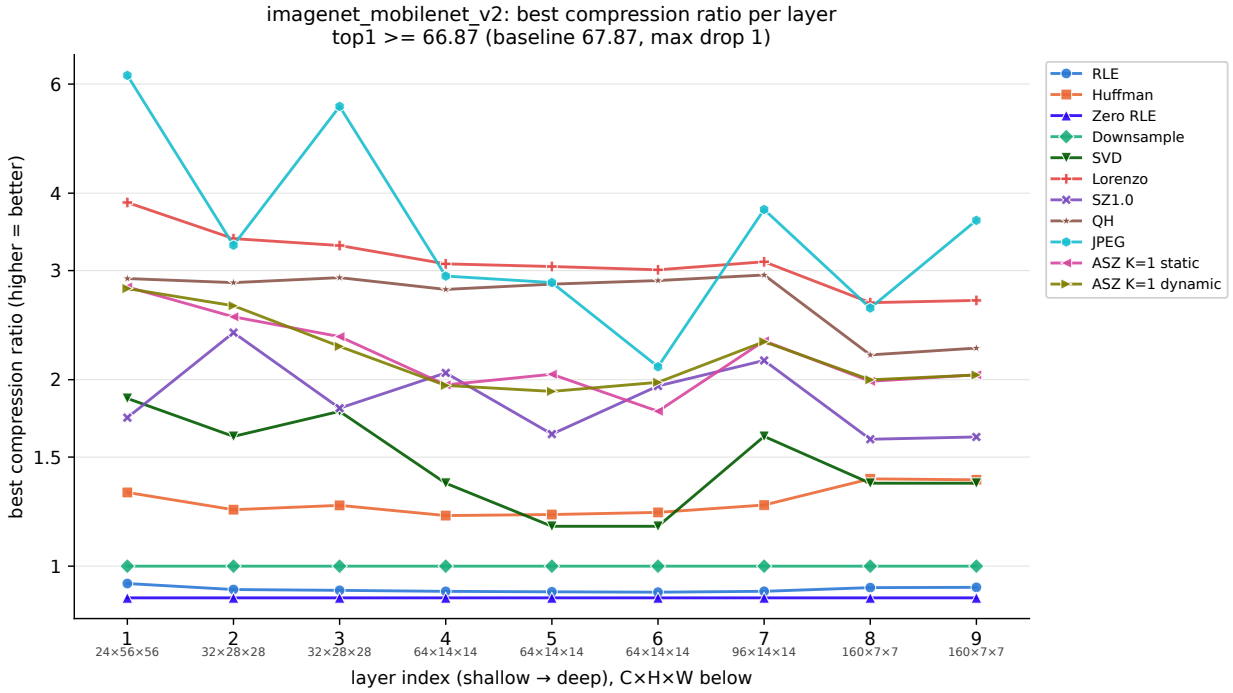


Figure 4.6: MobileNetV2, ImageNet: best compression ratio per connection within the 1 pp budget (top-1 at least 66.87 %), same results as Fig. 4.5.

4.2.1.4 MobileNetV3-Small

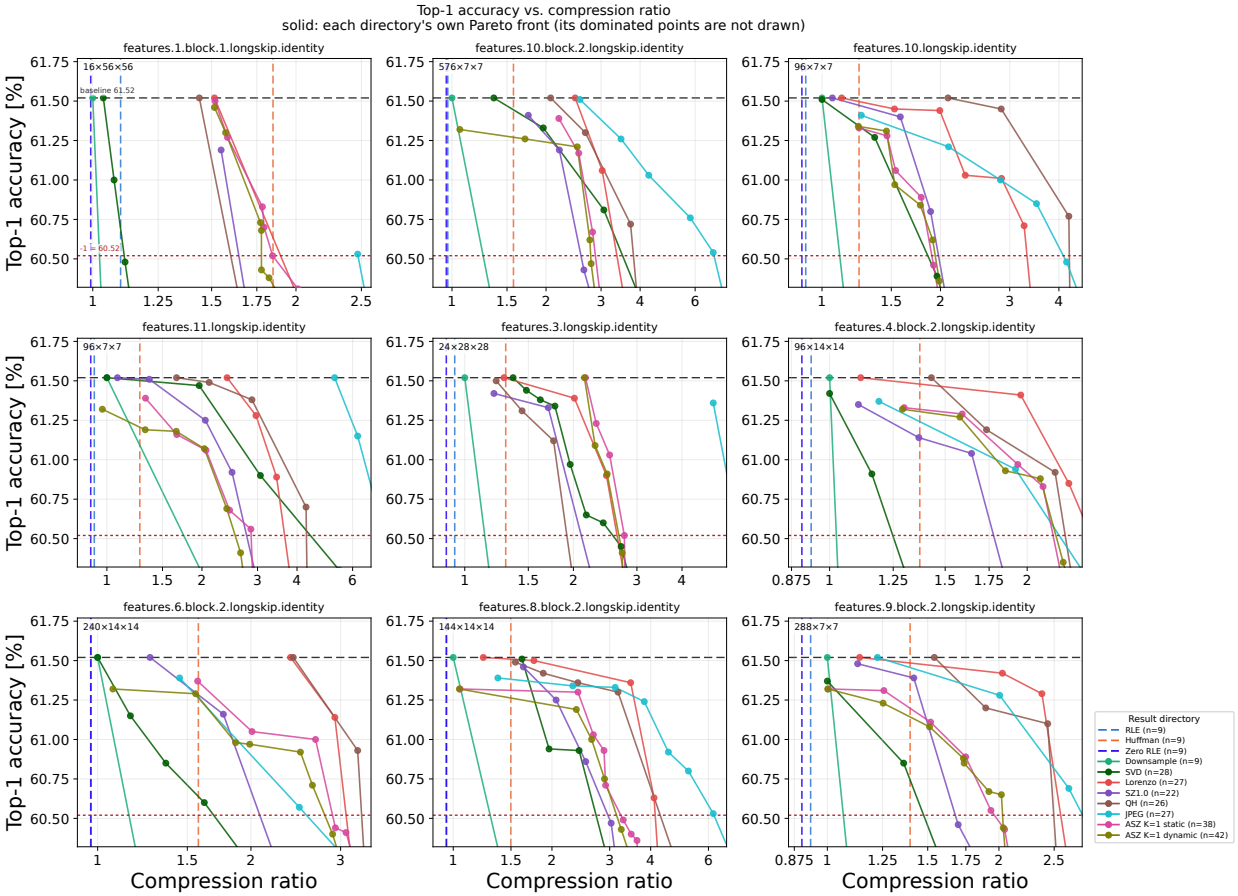


Figure 4.7: MobileNetV3-Small, ImageNet, 9 skip connections: Pareto fronts per connection (section 4.1.4), baseline 61.52 %.

Figures 4.7 and 4.8 show the results for MobileNetV3-Small. Its connections are of two kinds (Fig. B.4): the additions of the linear bottlenecks features.3, features.10 and features.11, which contain no zeros, and the inputs of the squeeze-and-excitation multiplications (`.block.1`, `.block.2`), which follow a ReLU or hard-swish activation and contain 2 to 15 % zeros. It is by far the most sensitive of the networks: quantizing a connection to 3 bits costs up to 41 pp, on features.1.block.1, and 20 pp on features.3 (appendix A.9.2). No method exceeds 6.9 within the budget.

Lossless baselines RLE expands every connection except features.1.block.1 (1.10), and zero RLE expands all of them (0.89 to 0.99), since even the sparsest one only just reaches its break-even of about 15 % zeros (section 4.1.3.2). Huffman coding reaches 1.37 to 1.85 on the squeeze-and-excitation inputs and 1.24 to 1.30 on the bottlenecks. On features.1.block.1, it is level with static ASZ and ahead of every other lossy method except JPEG.

Decimation Downsampling finds no factor within the budget on any connection.

Transforms JPEG is the best method on six connections, with 2.47 to 6.86; on features.1.block.1, its most accurate point (60.53 %) lies just at the budget. SVD reaches 1.08 to 3.06, the most on features.10.block.2 and features.11.

Quantization QH reaches 1.44 to 4.27 and is the best method on features.6.block.2 (3.24) and features.10 (4.24); it is at its lowest on the two most sensitive connections, features.1.block.1 (1.44) and features.3 (1.76). Lorenzo reaches 1.51 to 4.10 and is the best method on features.4.block.2 (2.31). As on the other networks, Lorenzo leads QH on the larger featuremaps, here the first three connections and features.8.block.2, and QH leads on features.6.block.2 and on all 7×7 connections.

Prediction-based compressors SZ1 reaches 1.42 to 2.53 and is below static ASZ on eight of nine connections (above on features.10: 1.89 against 1.79). Static ASZ reaches 1.79 to 2.91, without downsampling. On the two largest and most sensitive connections, features.1.block.1 (1.85) and features.3 (2.77), it is ahead of both quantizers; on the other seven, it is below Lorenzo. On seven connections, its most accurate points lie 0.13 to 0.20 pp below the baseline (the 8-bit reference measured in its scale-back: 61.32 %). Dynamic ASZ stays within 0.12 of the static variant, except on features.3 (2.48 against 2.77) and features.11 (2.40 against 2.86).

Summary JPEG leads on six connections, QH on two and Lorenzo on one.

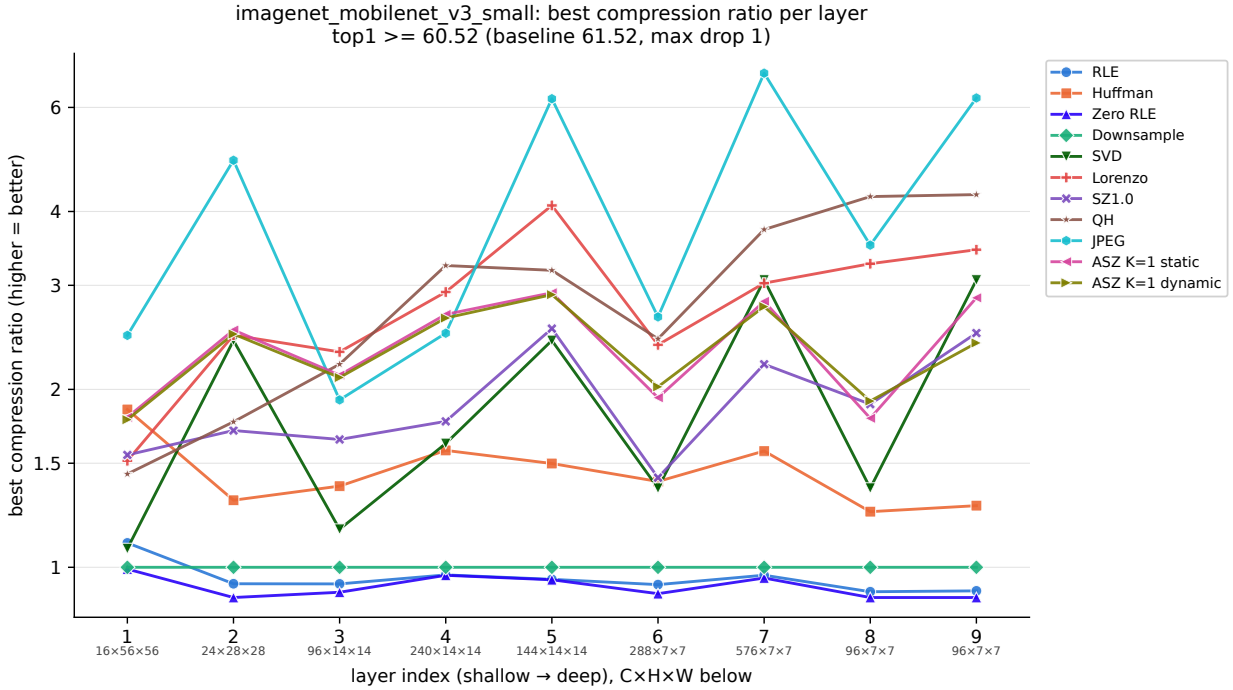


Figure 4.8: MobileNetV3-Small, ImageNet: best compression ratio per connection within the 1 pp budget (top-1 at least 60.52 %), same results as Fig. 4.7.

4.2.2 Cifar100

4.2.2.1 MobileNetV2

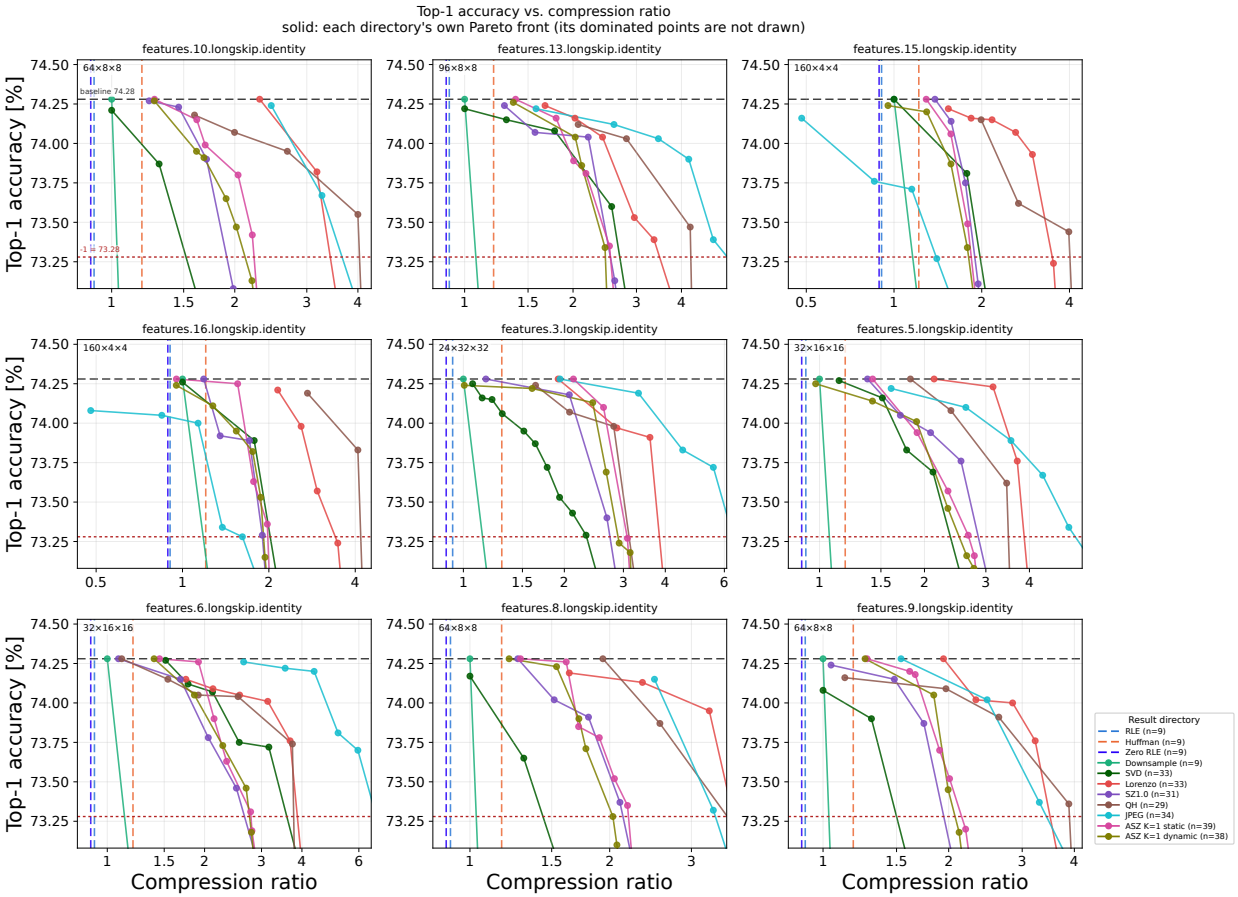


Figure 4.9: MobileNetV2, CIFAR-100, 9 skip connections: Pareto fronts per connection (section 4.1.4), baseline 74.28 %.

Figures 4.9 and 4.10 show the results for MobileNetV2 on CIFAR-100. All ratios are low compared to RegNetX-400MF; the best method reaches about 6.

Lossless baselines and decimation Since the operands of MobileNetV2 are linear bottlenecks without a ReLU, they contain neither runs nor zeros, and RLE and zero RLE expand every connection (appendix A.11.1). Huffman coding reaches 1.17 to 1.3 almost uniformly. Downsampling finds no factor within the budget on any connection.

Transforms JPEG leads on the first three connections and on the seventh with about 4.9 to 6, is on par with Lorenzo on connections 4 to 6, and collapses on the two 4×4 maps (1.15 and 1.6), since padding them to 8×8 codes 64 values for every 16 (section 4.1.3.5). SVD depends strongly on the connection: it reaches about 3.2 on the third connection ($32 \times 16 \times 16$), but only about 1.3 on the three $64 \times 8 \times 8$ connections.

Quantization Lorenzo is very consistent at 3 to 3.7 and ahead of SZ1 and ASZ on every connection. QH reaches 2.6 to 4.2 and is the best method on the deep connections 5, 6, 8 and 9, whose featuremaps are 8×8 and 4×4; as on the ImageNet networks, QH gains on the small featuremaps (section 4.5).

Prediction-based compressors Static ASZ reaches 1.8 to 2.8, above SZ1 (1.7 to 2.7) on most connections, but below Lorenzo and QH on all of them. Dynamic ASZ is lower still on most connections, e. g. 2.33 against 2.67 on the second. Several ASZ configurations that held the budget on the calibration split fall below it on the test split, which is the effect of calibrating on training images (section 4.1.1).

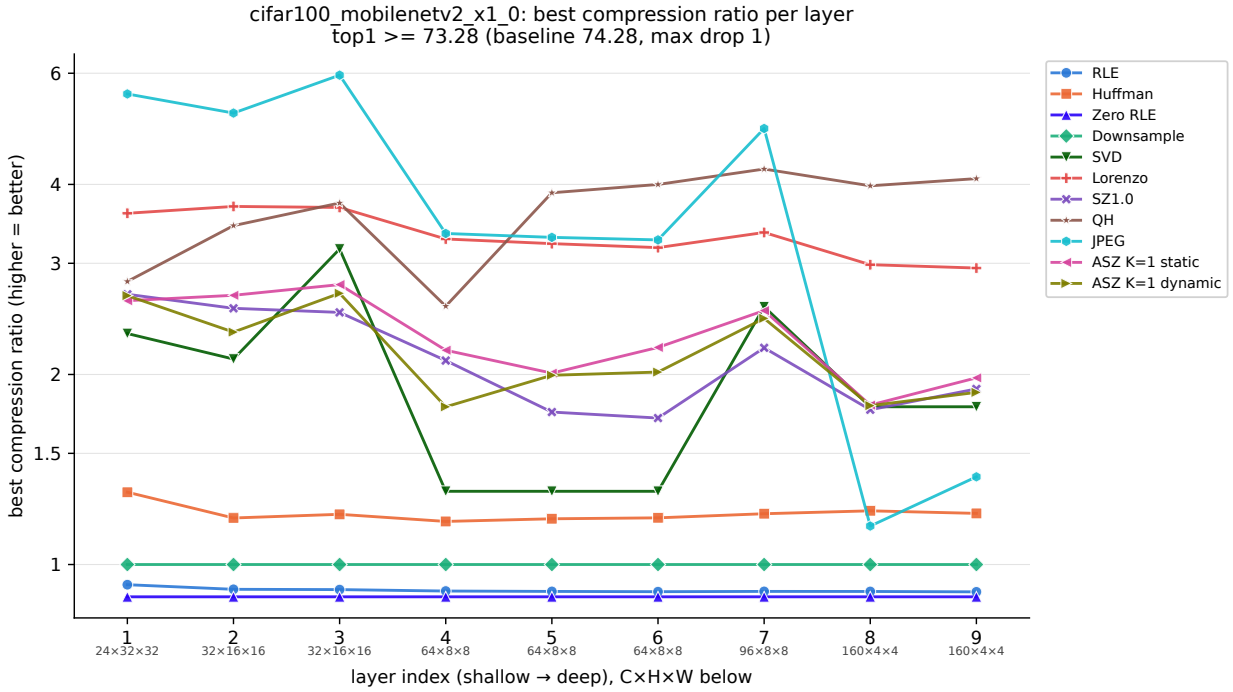


Figure 4.10: MobileNetV2, CIFAR-100: best compression ratio per connection within the 1 pp budget (top-1 at least 73.28 %), same results as Fig. 4.9.

4.2.2.2 ResNet-20

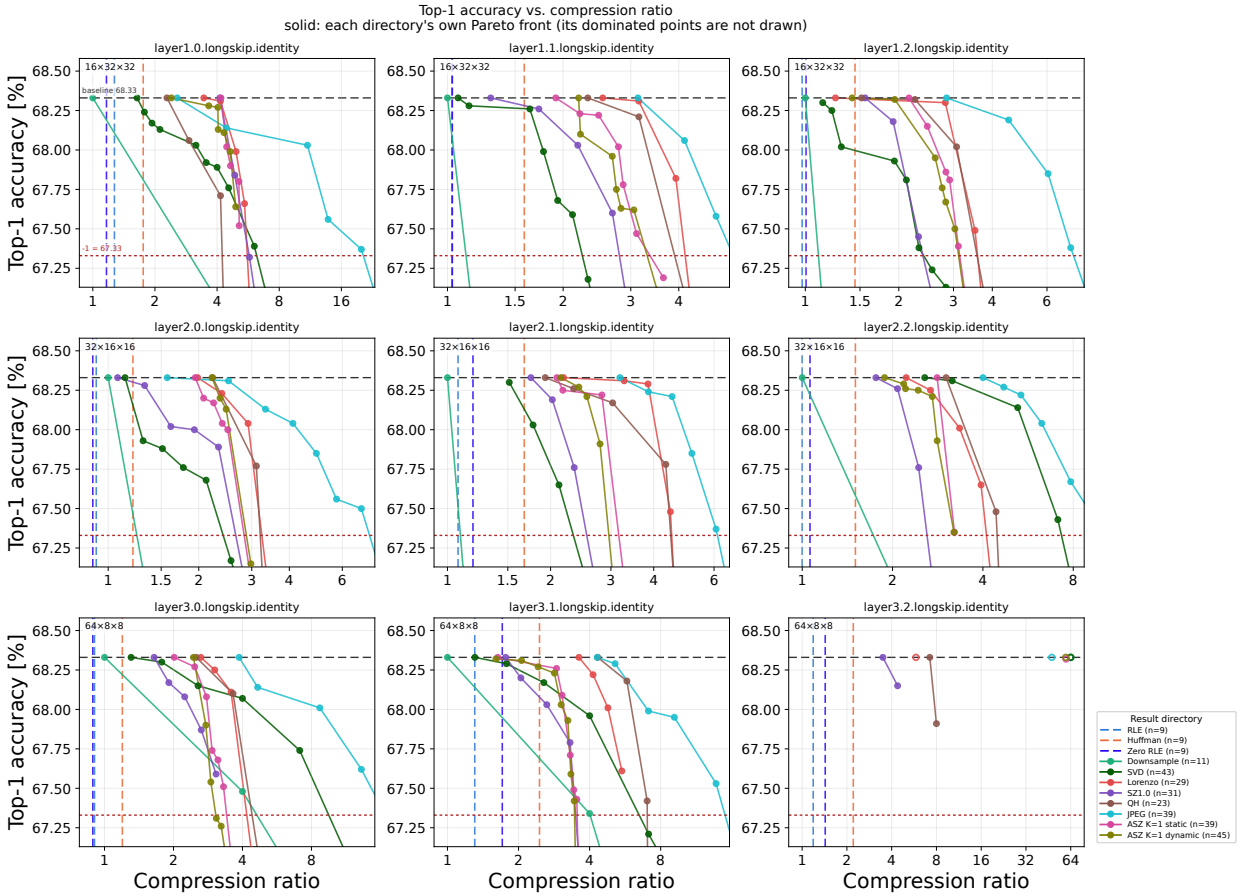


Figure 4.11: ResNet-20, CIFAR-100, 9 skip connections: Pareto fronts per connection (section 4.1.4), baseline 68.33 %.

Figures 4.11 and 4.12 show the results for ResNet-20 on CIFAR-100. Its last connection, layer3.2, is insensitive and discussed separately; the other paragraphs cover the first eight connections.

Lossless baselines and decimation Apart from the projection connections layer2.0 and layer3.0, the operands are post-ReLU and contain 17 to 59 % zeros, so that zero RLE is ahead of RLE on the deep connections (layer3.1: about 1.7 against 1.3). RLE itself reaches 0.9 to 1.3 and Huffman coding 1.2 to 2.5, the most on the 8×8 connections. Downsampling reaches 4:1 on connections 7 and 8.

Transforms JPEG is the best method on the first eight connections with 5 to 20, the most on layer1.0. SVD ranges from 2.1 to about 7 on connections 6 and 7.

Quantization Lorenzo (2.9 to 5.5) is ahead of SZ1 and ASZ on the first eight connections, and QH (3.1 to 7.0) overtakes it from layer2.2 on, most clearly on layer3.1 (7.0 against 5.5).

Prediction-based compressors Static ASZ reaches 2.5 to 5.1 on the first eight connections, below Lorenzo and above SZ1 (2.3 to 4.9); the dynamic variant is practically identical. Unlike the other CIFAR networks, this one is not affected by the calibration problem.

The last connection Layer3.2, directly before pooling and classifier, is insensitive: SVD and downsampling reach 64 within the budget, ASZ about 60, only through downsampling by 8 (section 4.2.1.2), and JPEG about 48. QH reaches its limit of 8, Lorenzo 5.8 and SZ1 4.4.

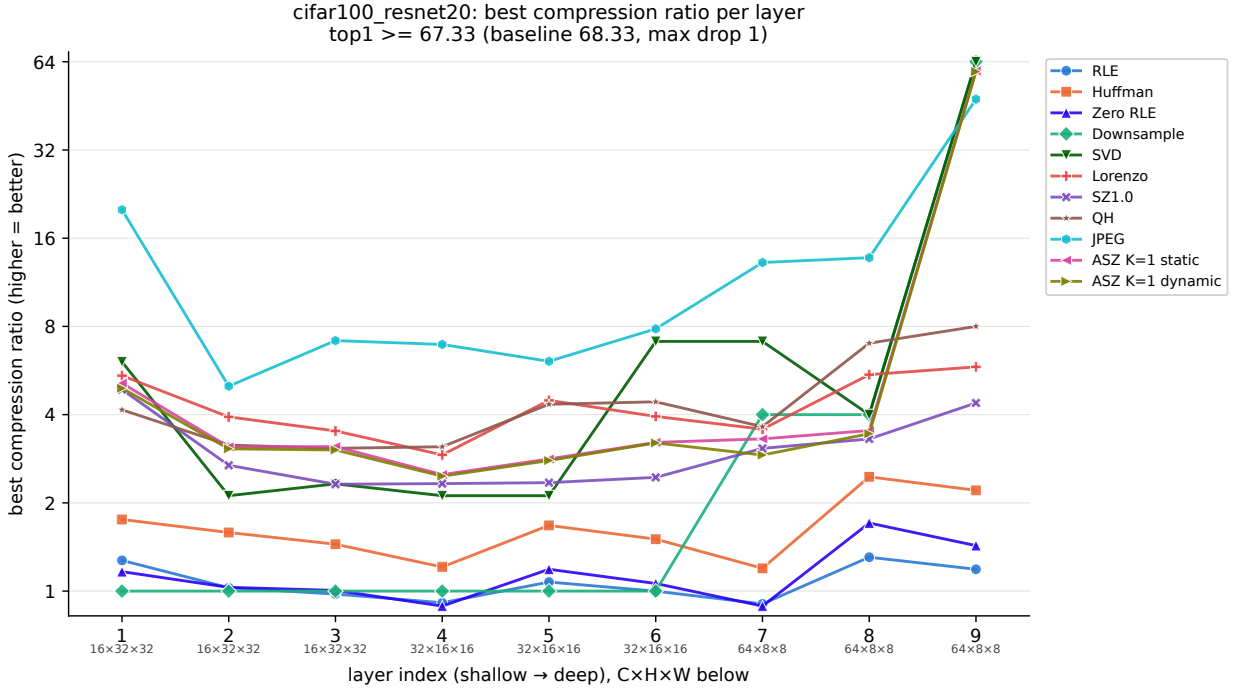


Figure 4.12: ResNet-20, CIFAR-100: best compression ratio per connection within the 1 pp budget (top-1 at least 67.33 %), same results as Fig. 4.11.

4.2.3 Cifar10

4.2.3.1 MobileNetV2

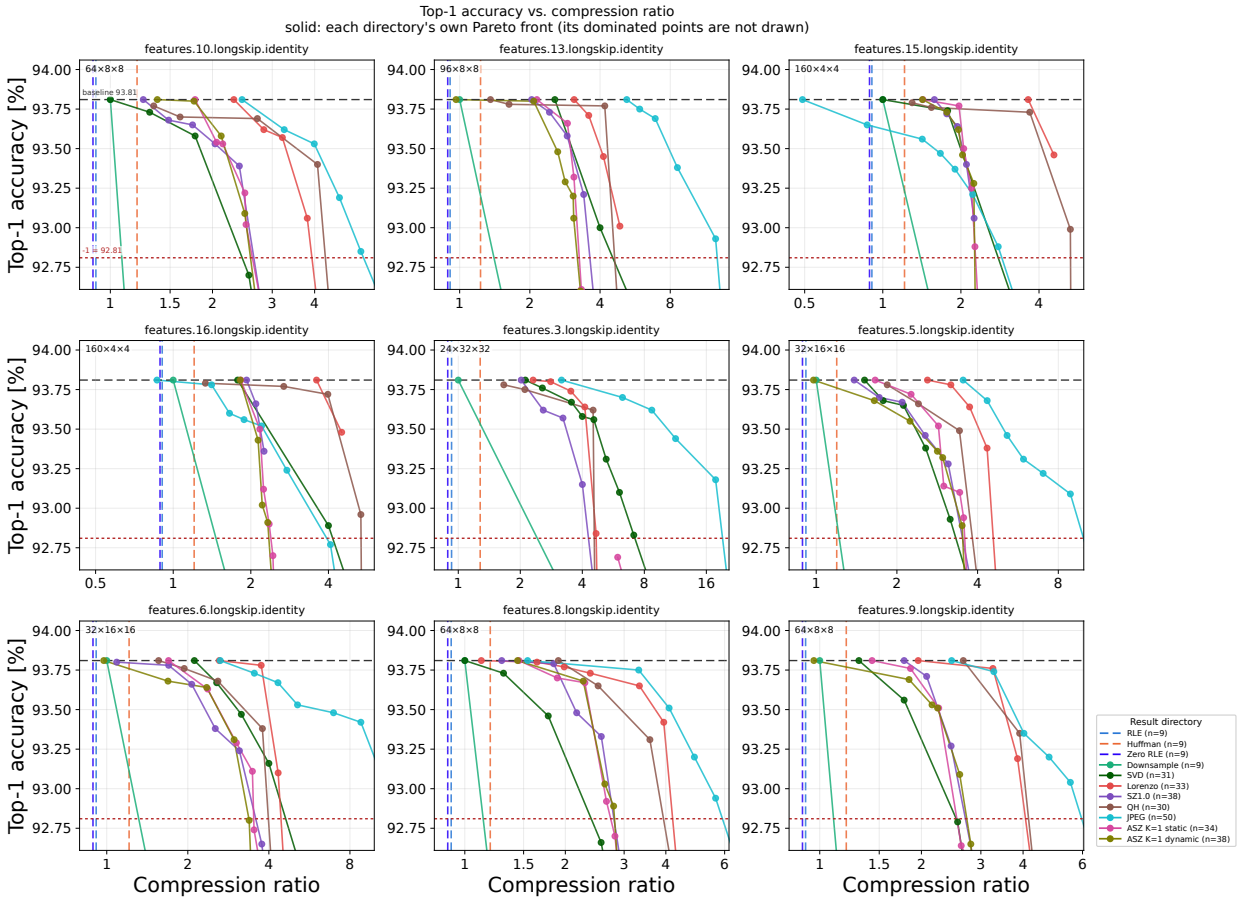


Figure 4.13: MobileNetV2, CIFAR-10, 9 skip connections: Pareto fronts per connection (section 4.1.4), baseline 93.81 %.

Figures 4.13 and 4.14 show the results for MobileNetV2 on CIFAR-10, which resemble those on CIFAR-100.

Lossless baselines and decimation RLE and zero RLE expand every connection, Huffman coding reaches about 1.2, and downsampling finds no factor within the budget.

Transforms JPEG is the best method on the first seven connections with 5.5 to 18 and again collapses on the 4×4 maps (about 2.8). SVD reaches about 7 on the first connection ($24 \times 32 \times 32$) and about 4 on connections 3, 7 and 9, but only 1.8 on the $64 \times 8 \times 8$ connections and on connection 8.

Quantization QH leads on the 4×4 maps with up to 5.3 and is also ahead of Lorenzo on connections 5 and 6. Lorenzo (3.8 to 4.9) is ahead of QH on the other five connections and ahead of SZ1 and ASZ on every connection.

Prediction-based compressors Static ASZ reaches 2.3 to 3.6, roughly on par with SZ1 (2.3 to 4), below Lorenzo everywhere and below QH everywhere except on the second connection (3.55 against 3.42). On the first connection, none of its configurations stays within the budget on the test split, although they did on the calibration split (section 4.1.1). Dynamic ASZ is slightly ahead of the

static variant on connections 4 and 5 (2.79 against 2.66 and 2.60 against 2.25); apart from the first two connections of ResNet-18 on ImageNet, this is the only case in which dynamic selection gains on several connections.

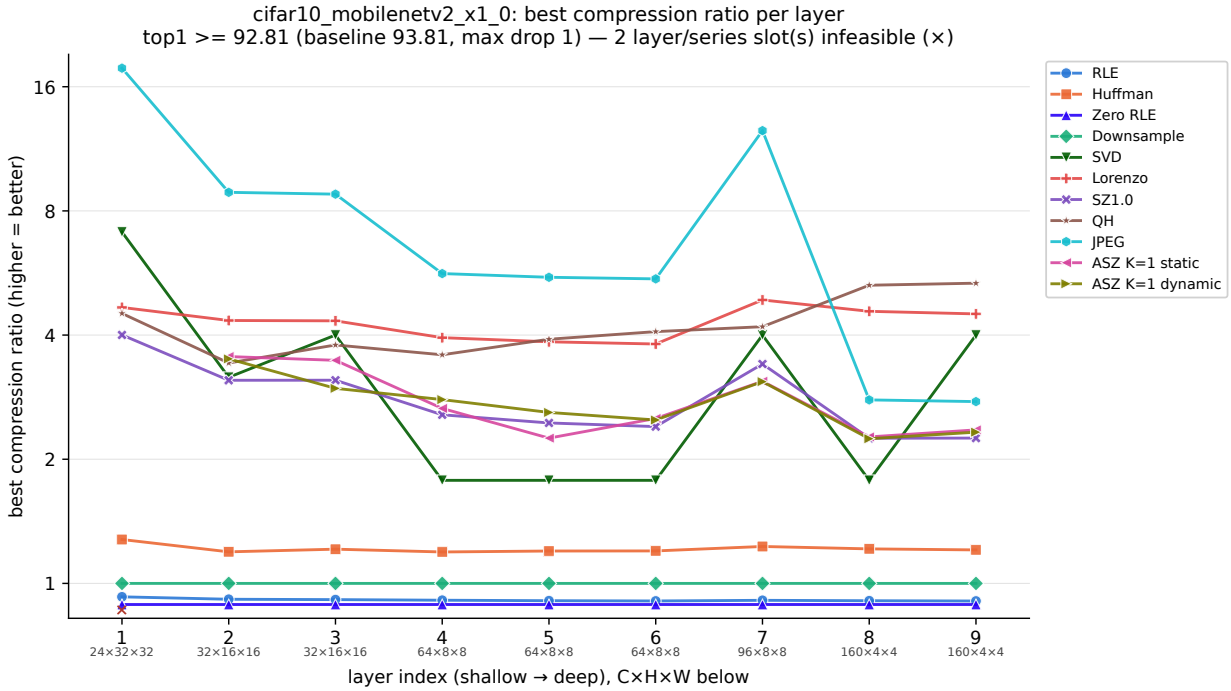


Figure 4.14: MobileNetV2, CIFAR-10: best compression ratio per connection within the 1 pp budget (top-1 at least 92.81 %), same results as Fig. 4.13.

4.2.3.2 ResNet-20

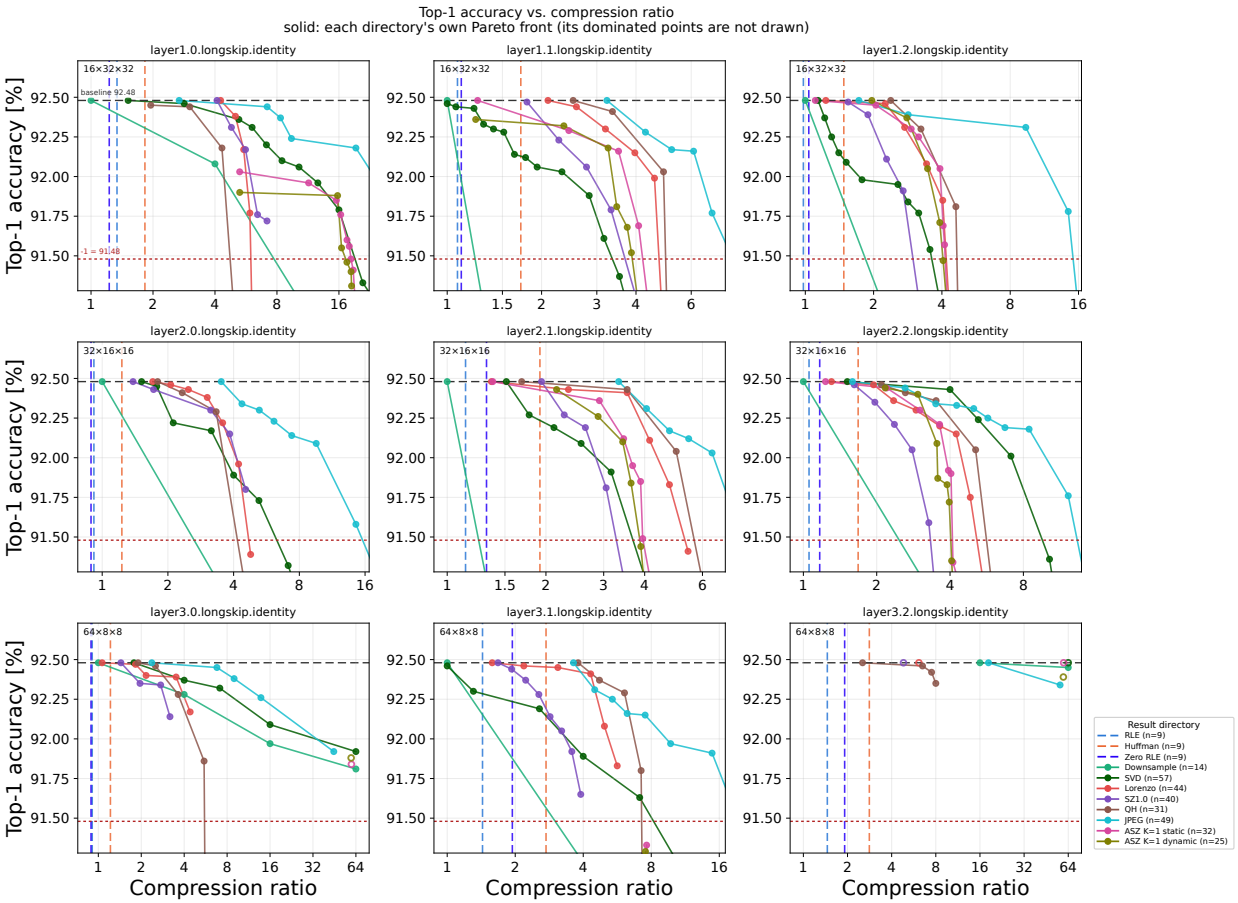


Figure 4.15: ResNet-20, CIFAR-10, 9 skip connections: Pareto fronts per connection (section 4.1.4), baseline 92.48 %.

Figures 4.15 and 4.16 show the results for ResNet-20 on CIFAR-10. Two connections, layer3.0 and layer3.2, are insensitive and discussed separately; the other paragraphs cover the remaining seven.

Lossless baselines and decimation RLE reaches 0.9 to 1.5 and expands connections 3, 4 and 7, the last two of which are projections without zeros. Zero RLE is ahead of it on the deep connections (layer3.1: about 1.9 against 1.4), and Huffman coding reaches 1.2 to 2.8. Downsampling only reaches 4:1 on layer1.0.

Transforms JPEG is the best method on the first six connections and on layer3.1, with 6 to 19, the most on layer1.0. SVD reaches 3 to 7 on the middle connections and about 16 on layer1.0.

Quantization QH (3.3 to 7.2) follows JPEG on most connections, ahead of Lorenzo (4 to 5.9) everywhere except on layer1.0 and layer2.0.

Prediction-based compressors SZ1 reaches 2.7 to 7.2, the most on layer1.0. Static ASZ reaches about 18 on layer1.0 by combining downsampling by 2 with the SZ stage, second only to JPEG and ahead of SVD. As on ResNet-18 (section 4.2.1.2), this lead belongs to the downsampling: the SZ and Huffman stages add a factor of 4.6 on top of it, less than Lorenzo (5.9) and SZ1 (7.2) reach on layer1.0 at full resolution. On the remaining sensitive connections, ASZ

reaches only about 4, below QH and mostly below Lorenzo. On layer2.0 and layer3.1, neither ASZ variant has a configuration within the budget on the test split, an effect of the calibration problem (section 4.1.1). The dynamic variant is otherwise practically identical to the static one.

The insensitive connections SVD and downsampling reduce layer3.0 and layer3.2 to a single value per channel (CR = 64) within the budget, and ASZ, whose search chose the same downsampling, reaches 59.5. JPEG reaches 44.8 and 56.3, QH 5.5 and its limit of 8, Lorenzo 4.4 and 6.1.

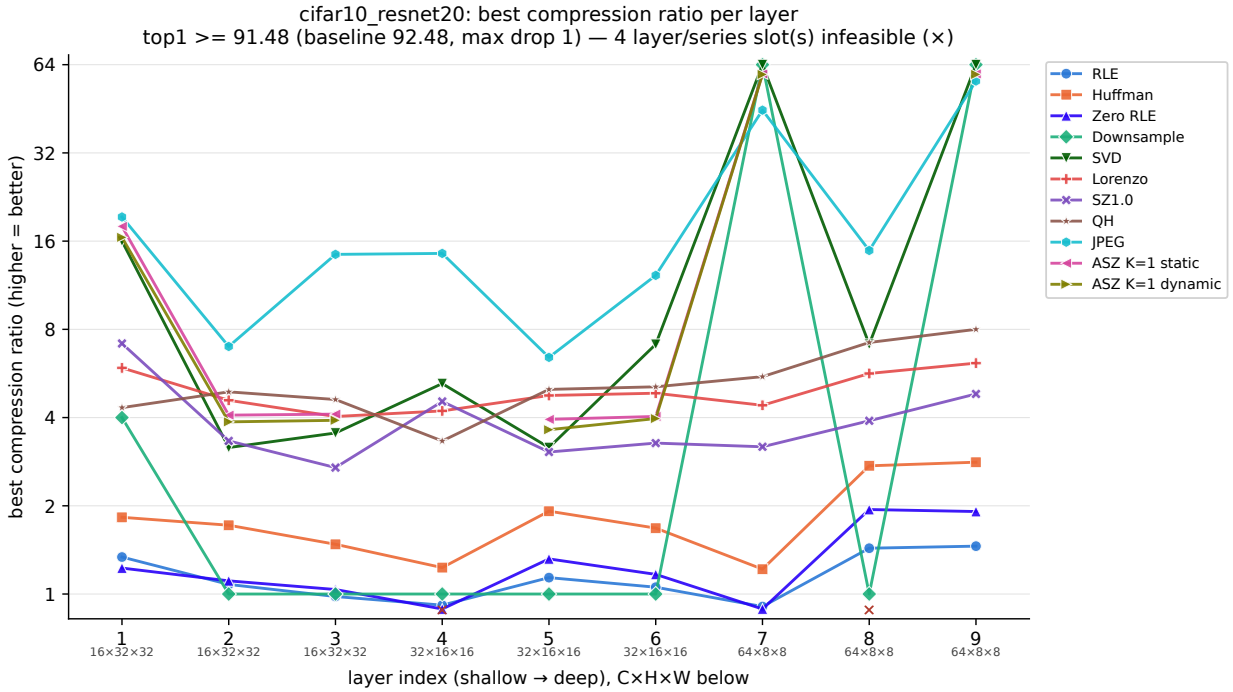


Figure 4.16: ResNet-20, CIFAR-10: best compression ratio per connection within the 1 pp budget (top-1 at least 91.48 %), same results as Fig. 4.15.

4.2.4 All Networks

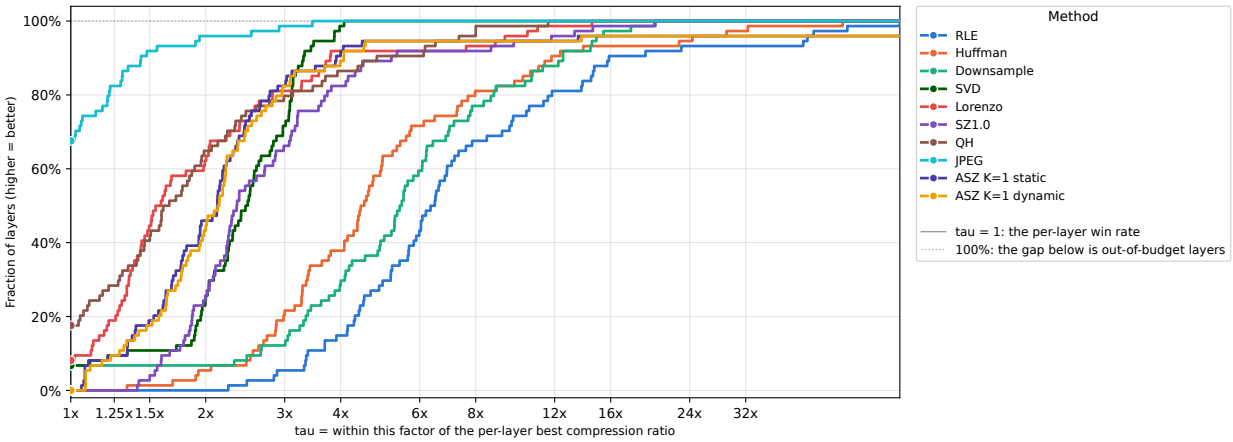


Figure 4.17: Performance profile at a 1 pp budget (section 4.1.4): all 74 connections pooled (12 of RegNetX-400MF, 8 of ResNet-18, 9 of MobileNetV2 and 9 of MobileNetV3-Small on ImageNet, 9 each of ResNet-20 and MobileNetV2 on CIFAR-10 and CIFAR-100).

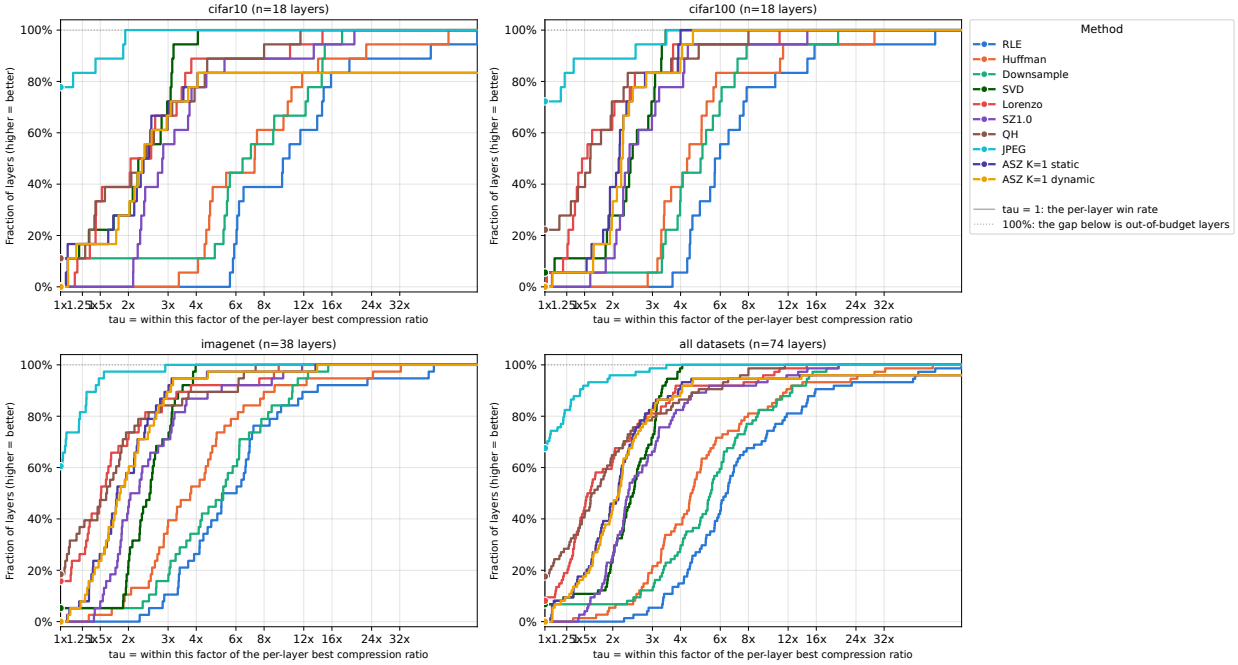


Figure 4.18: Performance profile at a 1 pp budget (section 4.1.4): one panel per dataset (CIFAR-10: 18, CIFAR-100: 18, ImageNet: 38 connections) and all of them pooled.

Figures 4.17 and 4.18 summarize the 74 connections of all networks in performance profiles. Pooled over all of them, JPEG is the best method on 50 connections (68 %), comes within a factor of 1.25 of the best on 61 and within a factor of 2 on 71. By depth, it leads on 23 of the 25 connections in the first third of their network, on 15 of 24 in the middle and on 12 of the 25 in the last third. QH is the best method on 13 connections and Lorenzo on 6; SVD and downsampling share the lead on the 5 insensitive connections on which both reduce the featuremap to a single value per channel. SZ1 and ASZ are the best method on none. Within a factor of 2 of the best, QH (48 connections) and Lorenzo (46) are followed by static and dynamic ASZ (34 and 33) and, far behind, by SZ1 (19), SVD (18) and Huffman coding (4). The geometric means of the ratio within the budget (Fig. 4.19) order the methods similarly: JPEG 6.6, QH 3.9, Lorenzo 3.8, static ASZ 3.5, dynamic ASZ 3.4, SVD 3.2, SZ1 2.7, Huffman coding 1.5, downsampling 1.4 and RLE 1.0, where the mean of ASZ covers only the 71 connections with a point within the budget. ASZ is the only method without a point within the budget on some connections, three on CIFAR-10, so that its curves end at about 96 %; the curves of its static and dynamic variant lie almost on top of each other. Per dataset, JPEG leads everywhere (CIFAR-10: 14 of 18 connections, CIFAR-100: 13 of 18, ImageNet: 23 of 38); QH wins 2 connections on CIFAR-10, 4 on CIFAR-100 and 7 on ImageNet, and Lorenzo 6 on ImageNet, five of them on MobileNetV2 and one on MobileNetV3-Small, and none on CIFAR.

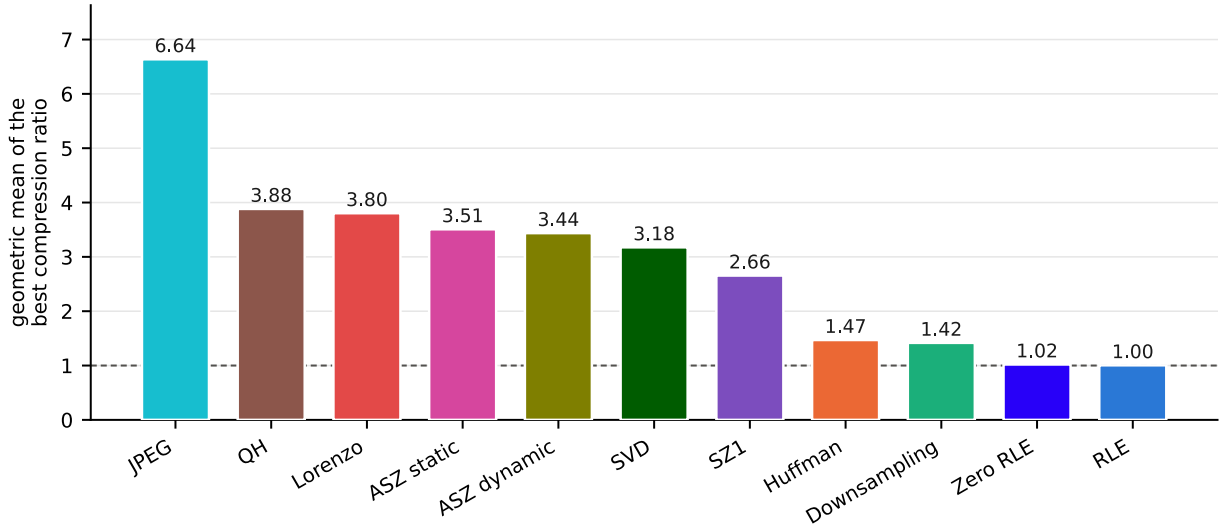


Figure 4.19: Geometric mean of the best compression ratio within the 1 pp budget over the 74 connections of all networks, sorted by magnitude; per connection, the best ratio is that of table A.1. ASZ has no point within the budget on three connections of CIFAR-10, so its means cover the remaining 71; for downsampling, a connection without a factor within the budget counts as uncompressed (ratio 1). Dashed: ratio 1. Colours as in the ratio-accuracy figures.

Lossless compression across networks How well the lossless methods compress depends on the operand of a connection rather than on the network or the dataset, namely on whether it contains zeros. 41 of the 74 connections contain none: all connections of MobileNetV2, the linear bottlenecks of MobileNetV3-Small, and the projections at the start of a stage in RegNetX-400MF and the ResNets. On them, RLE reaches at most 1.0 (median 0.91), zero RLE expands every one of them to exactly 8/9, and Huffman coding reaches only 1.17 to 1.47. MobileNetV2 is in this sense a dense network by design: its blocks end in a linear bottleneck without a ReLU [67], so that the added values contain no zeros and spread over the whole range with hardly any repetition. The 26 connections with at least 15 % zeros, all of them ReLU outputs in RegNetX-400MF and the ResNets, reach a median of 1.79 with Huffman coding, up to 3.97 on layer4.1 of ResNet-18 with 80 % zeros, and zero RLE is ahead of RLE on 22 of them. The remaining seven connections, with 2 to 15 % zeros, lie in between. Over all 74 connections, the rank correlation of the Huffman ratio with the zero fraction is 0.74.

Limits of the designs The highest ratios within the budget show how far the designs of table 4.2 can compress at all. Over all 74 connections, they are 8.0 for QH and 6.4 for Lorenzo, which spend at least one bit per value, 7.2 for SZ1 and 5.3 for ASZ without downsampling, whose 2-bit codes limit the SZ stage to 4:1, and 4.0 for Huffman coding. Only the methods that can drop values or coefficients entirely exceed these limits: downsampling and SVD reach 64, JPEG 56.3, and ASZ 59.5, but only through its downsampling stage. On the insensitive connections, the design of a compressor therefore matters more than how well it codes: a method that spends a minimum number of bits on every value cannot exploit a connection that the network barely needs.

4.3 Use Cases: Compression of Multiple Layers

In the eviction scenario of SMOF [80], several connections of a network may be evicted at the same time, and the errors their compression introduces add up. In this section, we therefore compress several connections of a network together, with the same method on every connection but a separate configuration for each of them.

The configurations are combined per method. For SZ1.0 and Lorenzo, we take the best single-connection point within the budget on every connection and then scale all thresholds back towards lossless by one shared factor. SVD, downsampling, QH and JPEG step all connections together along their rank, factor, code width or quality ladder until the budget is exceeded, and then step individual connections back. ASZ, static and dynamic, scales back from the per-connection picks of the search, here with a context of $K = 3$.

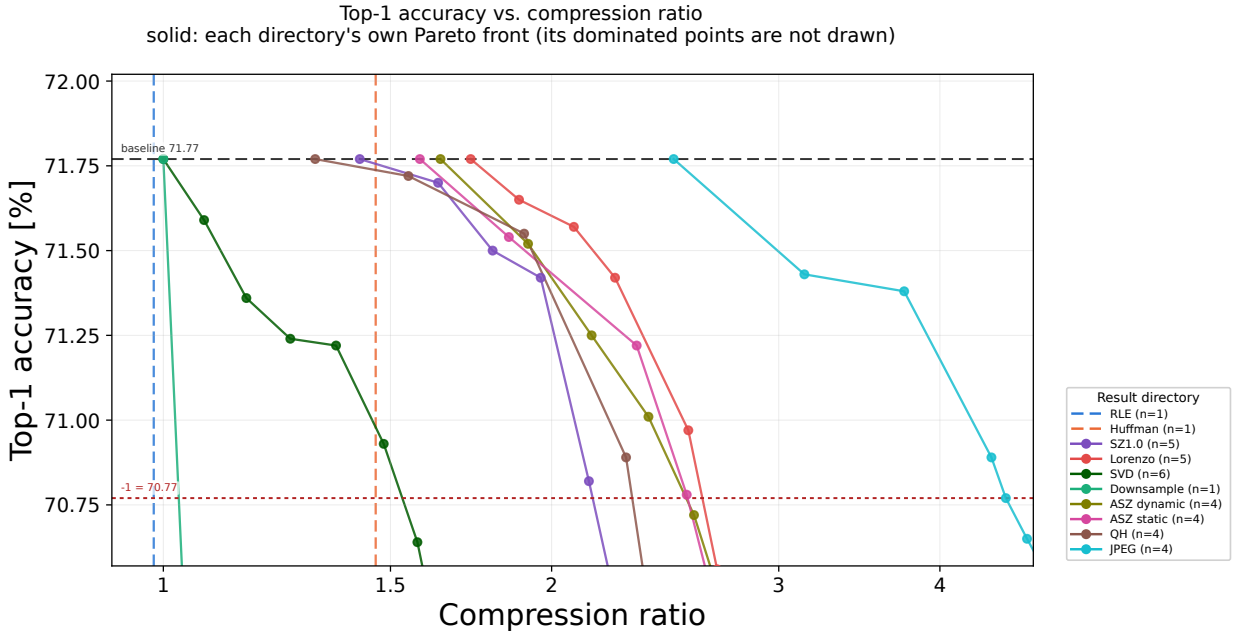


Figure 4.20: Use case: the first three skip connections of RegNetX-400MF (`block1-0`, `block2-0`, `block2-1`) compressed together, compression ratio over the three connections; plotted as the Pareto fronts per connection (section 4.1.4), baseline 71.77 %.

JPEG reaches 4.5 within the budget, the highest ratio; Lorenzo and static ASZ follow with 2.55 each, then dynamic ASZ (2.38), QH (2.28) and SZ1 (2.14). SVD reaches 1.48, about the same as Huffman coding (1.46), downsampling has no step within the budget, and RLE expands the three connections (0.98). Against the mean of the single-connection bests on the same three connections (Fig. 4.2), JPEG falls from 8.2 to 4.5, SVD from 3.4 to 1.5, Lorenzo from 3.0 to 2.55 and SZ1 from 2.4 to 2.1, while QH keeps its 2.3; JPEG and SVD lose the most. ASZ reaches 3.6 (static) on the single connections with $K = 1$; since the use case runs with $K = 3$, the two are not directly comparable.

The following use cases compress three connections of each of the other networks of table 4.1 together. Here, ASZ runs with $K = 1$, as in section 4.2, and without the downsampling stage.

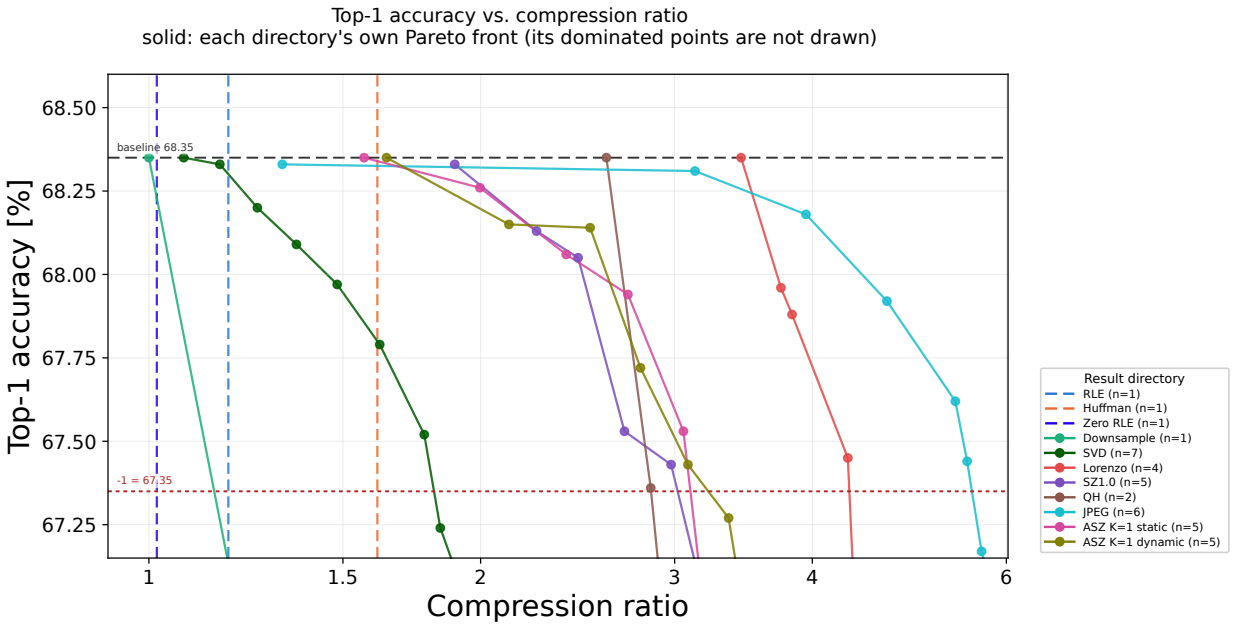


Figure 4.21: Use case: three skip connections of ResNet-18, ImageNet (`layer1.0`, `layer1.1`, `layer2.0`) compressed together, compression ratio over the three connections; plotted as the Pareto fronts per connection (section 4.1.4), baseline 68.35 %.

JPEG reaches 5.53 within the budget, the highest ratio; Lorenzo follows with 4.31, then dynamic ASZ (3.08), static ASZ (3.06), SZ1 (2.98) and QH (2.85). SVD reaches 1.78, Huffman coding 1.61, RLE 1.18 and zero RLE 1.02; downsampling has no step within the budget. Against the mean of the single-connection bests on the same three connections (Fig. 4.4), JPEG falls from 7.5 to 5.5, dynamic ASZ from 4.2 to 3.1, static ASZ from 4.0 to 3.1, QH from 3.9 to 2.85, SZ1 from 3.8 to 3.0, SVD from 2.6 to 1.8 and Lorenzo from 4.6 to 4.3.

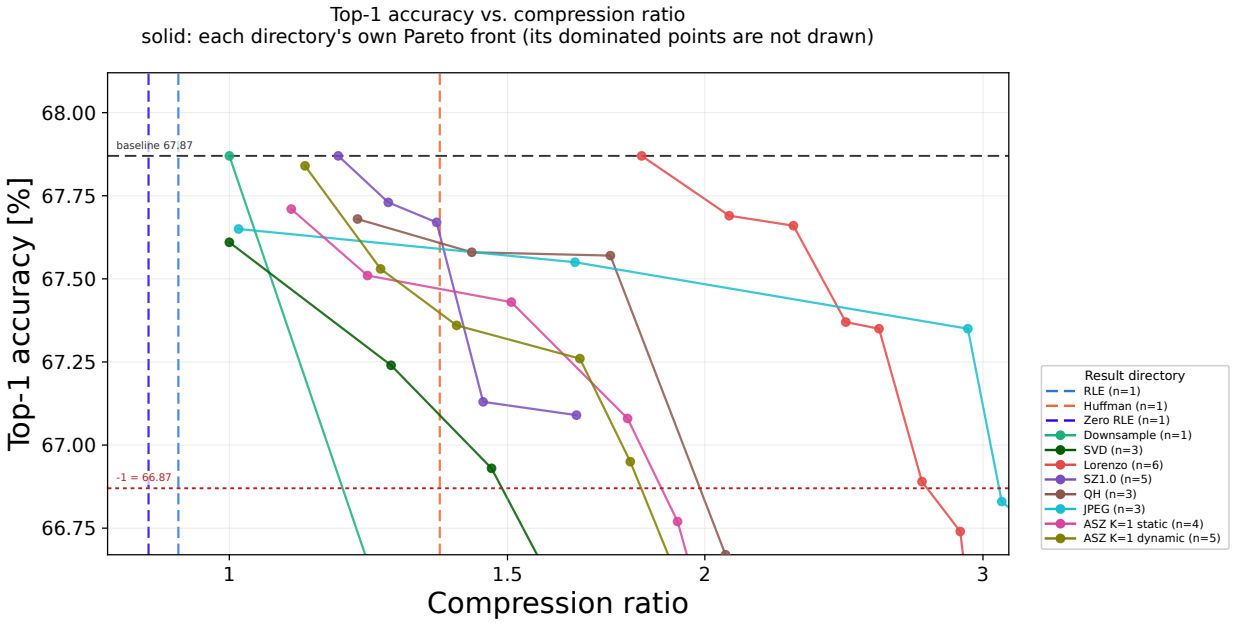


Figure 4.22: Use case: three skip connections of MobileNetV2, ImageNet (`features.3`, `features.15`, `features.16`) compressed together, compression ratio over the three connections; plotted as the Pareto fronts per connection (section 4.1.4), baseline 67.87 %.

JPEG reaches 2.93 within the budget, the highest ratio; Lorenzo follows with 2.74, then dynamic and static ASZ (1.79 each), QH (1.74) and SZ1 (1.66). SVD reaches 1.47 and Huffman coding 1.36; RLE (0.93) and zero RLE (0.89) expand the three connections, and downsampling has no step within the budget. Against the mean of the single-connection bests on the same three connections (Fig. 4.6), JPEG falls from 4.1 to 2.9, QH from 2.45 to 1.74, both ASZ variants from 2.3 to 1.8, Lorenzo from 3.1 to 2.7 and SVD from 1.53 to 1.47, while SZ1 keeps its 1.65.

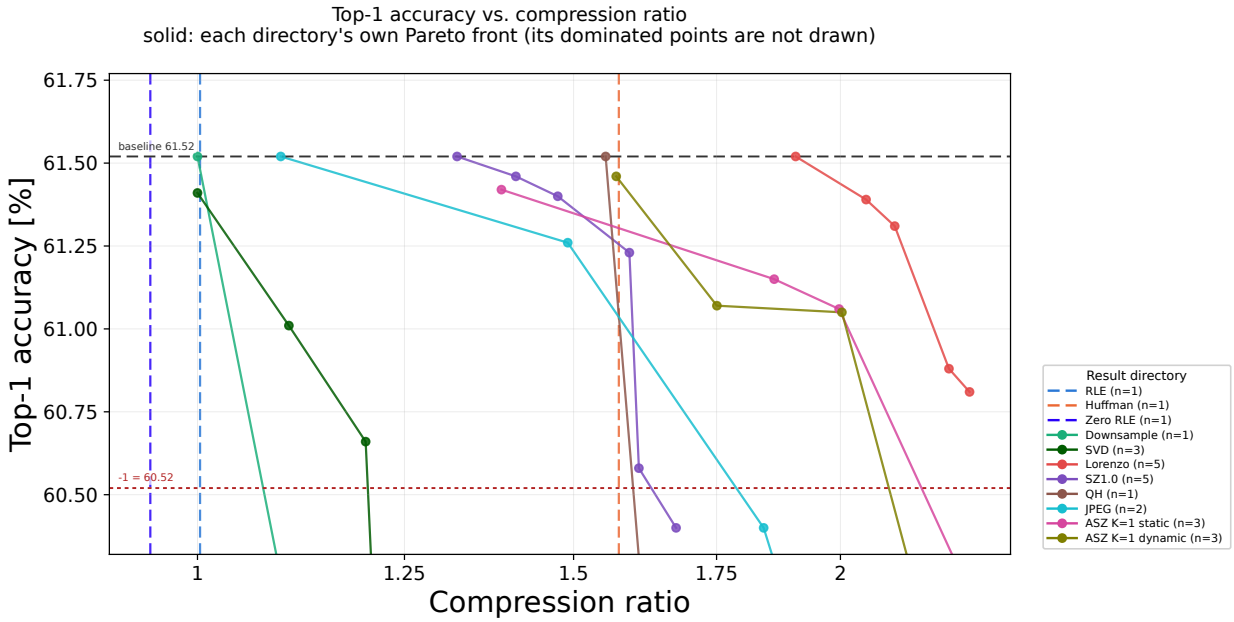


Figure 4.23: Use case: three skip connections of MobileNetV3-Small, ImageNet (`features.1.block.1`, `features.3`, `features.6.block.2`) compressed together, compression ratio over the three connections; plotted as the Pareto fronts per connection (section 4.1.4), baseline 61.52 %.

Lorenzo reaches 2.30 within the budget, the highest ratio; static and dynamic ASZ follow with 2.00 each, then SZ1 (1.61), Huffman coding (1.57), QH (1.55) and JPEG (1.49). SVD reaches 1.20. The front of QH consists of a single point (1.55, measured at 61.83 %), which dominates all its other points. Downsampling has no step within the budget, and RLE (1.00) and zero RLE (0.95) do not compress. Against the mean of the single-connection bests on the same three connections (Fig. 4.8), JPEG falls from 3.3 to 1.5, static ASZ from 2.4 to 2.0, QH from 2.2 to 1.55, SVD from 1.7 to 1.2 and SZ1 from 1.7 to 1.6, while Lorenzo keeps its 2.3. This is the only use case in which JPEG is not the best method, and the only one in which lossy methods, JPEG, QH and SVD, fall below Huffman coding.

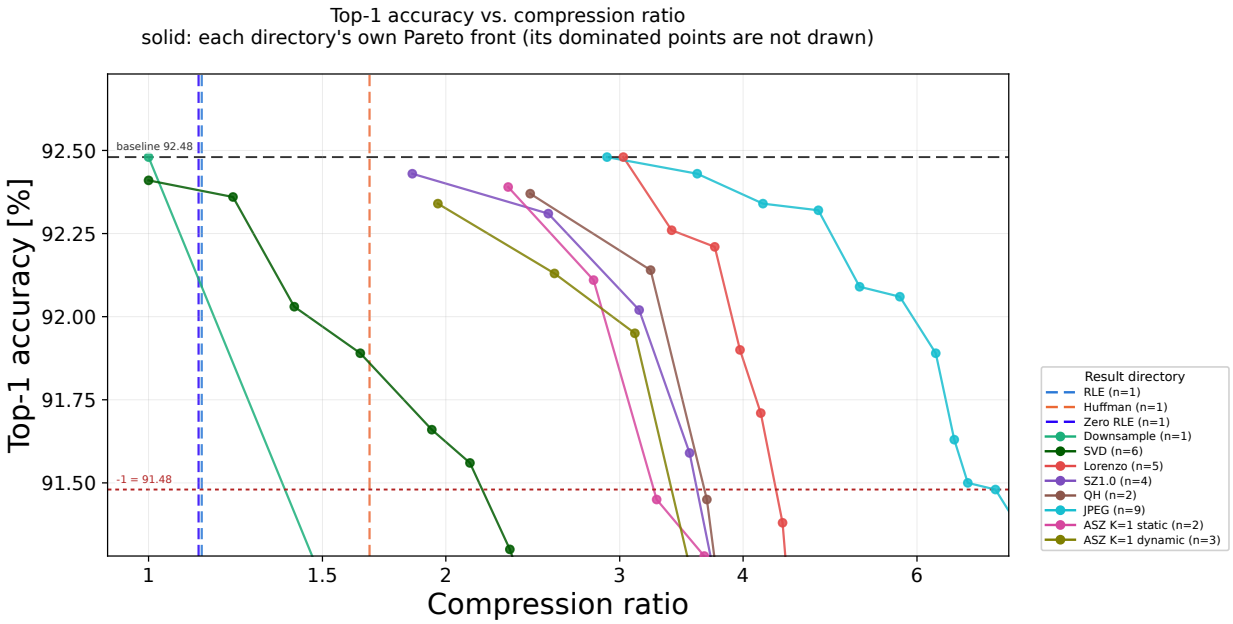


Figure 4.24: Use case: three skip connections of ResNet-20, CIFAR-10 (`layer1.0`, `layer1.1`, `layer1.2`) compressed together, compression ratio over the three connections; plotted as the Pareto fronts per connection (section 4.1.4), baseline 92.48 %.

JPEG reaches 7.20 within the budget, the highest ratio; Lorenzo follows with 4.17, then SZ1 (3.53), QH (3.22), dynamic ASZ (3.11) and static ASZ (2.82). SVD reaches 2.12, Huffman coding 1.67, RLE 1.13 and zero RLE 1.12; downsampling has no step within the budget. Against the mean of the single-connection bests on the same three connections (Fig. 4.16), JPEG falls from 13.6 to 7.2, SVD from 7.6 to 2.1, Lorenzo from 4.8 to 4.2, QH from 4.6 to 3.2 and SZ1 from 4.4 to 3.5. Static ASZ falls from 8.8 to 2.8, mostly because the single-connection search chose downsampling by 2 on layer1.0 (ratio 18.3), while the use case runs at full resolution; on layer1.1 and layer1.2, its single-connection bests are 4.1.

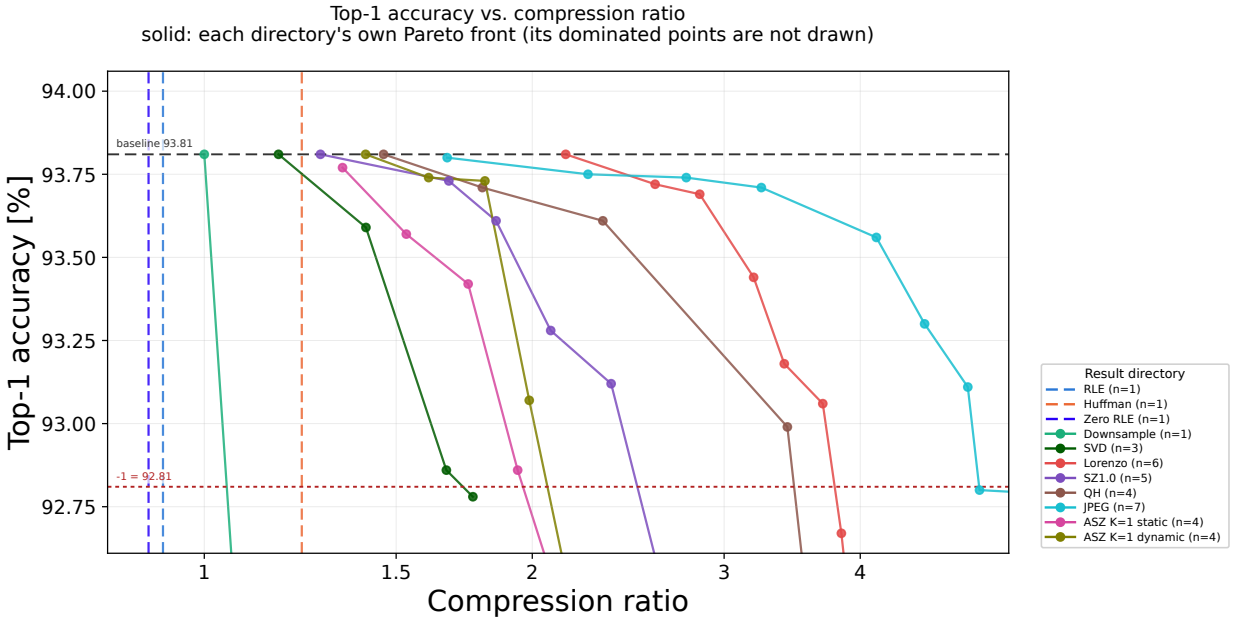


Figure 4.25: Use case: three skip connections of MobileNetV2, CIFAR-10 (`features.3`, `features.6`, `features.9`) compressed together, compression ratio over the three connections; plotted as the Pareto fronts per connection (section 4.1.4), baseline 93.81 %.

JPEG reaches 5.02 within the budget, the highest ratio; Lorenzo follows with 3.70, then QH (3.43), SZ1 (2.36), dynamic ASZ (1.99) and static ASZ (1.94). SVD reaches 1.67 and Huffman coding 1.23; RLE (0.92) and zero RLE (0.89) expand the three connections, and downsampling has no step within the budget. Against the mean of the single-connection bests on the same three connections (Fig. 4.14), JPEG falls from 10.7 to 5.0, SVD from 4.3 to 1.7, Lorenzo from 4.3 to 3.7, QH from 4.1 to 3.4 and SZ1 from 3.2 to 2.4. ASZ has no single-connection point within the budget on features.3; on features.6 and features.9, its single-connection bests (static) are 3.47 and 2.25.

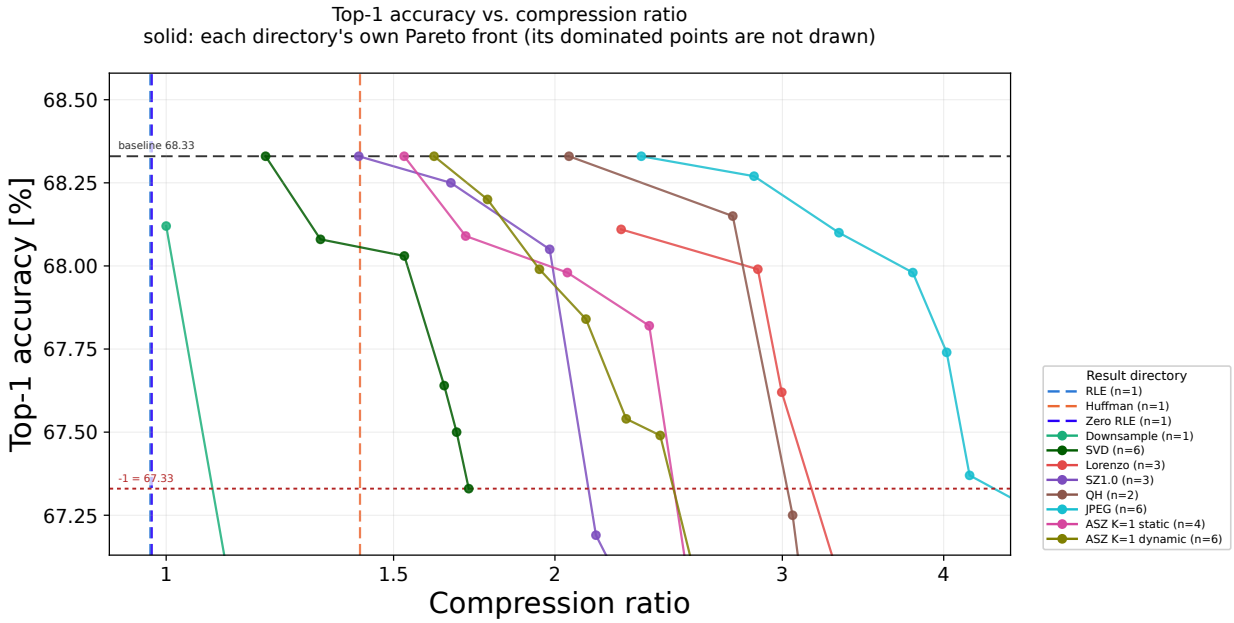


Figure 4.26: Use case: three skip connections of ResNet-20, CIFAR-100 (`layer1.1`, `layer1.2`, `layer2.0`) compressed together, compression ratio over the three connections; plotted as the Pareto fronts per connection (section 4.1.4), baseline 68.33 %.

JPEG reaches 4.19 within the budget, the highest ratio; Lorenzo follows with 3.00, then QH (2.75), dynamic ASZ (2.41), static ASZ (2.37) and SZ1 (1.98). SVD reaches 1.72 and Huffman coding 1.41; RLE and zero RLE (0.97 each) expand the three connections, and downsampling has no step within the budget. Against the mean of the single-connection bests on the same three connections (Fig. 4.12), JPEG falls from 6.4 to 4.2, Lorenzo from 3.5 to 3.0, QH from 3.1 to 2.75, static ASZ from 2.9 to 2.4, SZ1 from 2.4 to 2.0 and SVD from 2.2 to 1.7.

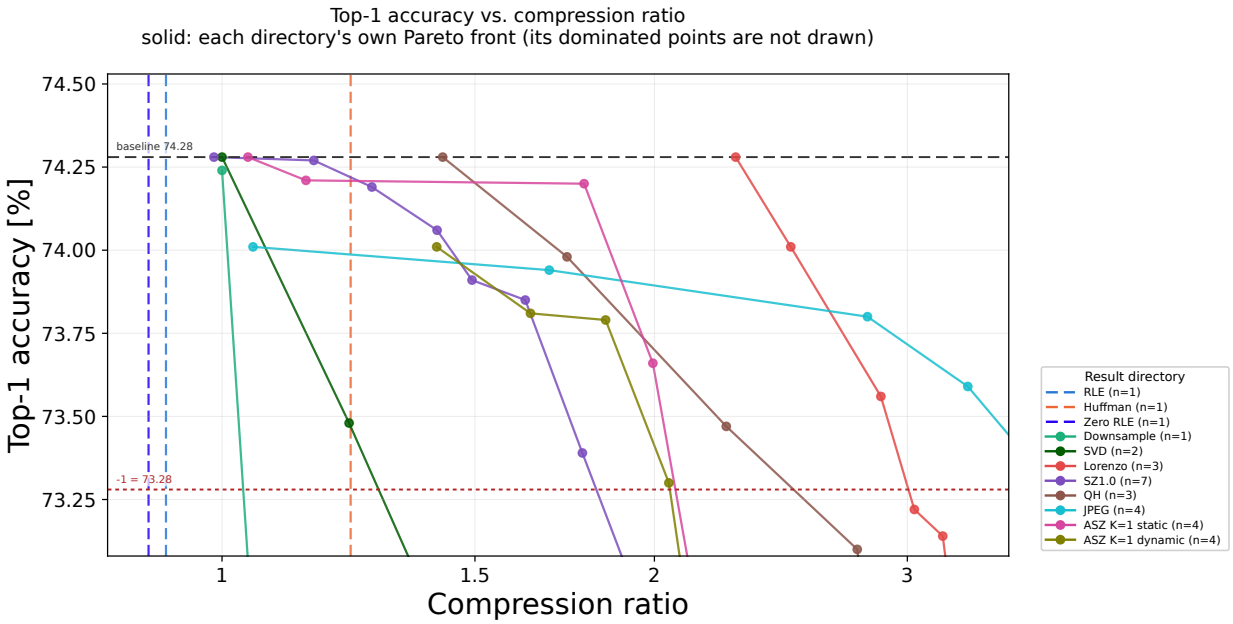


Figure 4.27: Use case: three skip connections of MobileNetV2, CIFAR-100 (`features.3`, `features.9`, `features.13`) compressed together, compression ratio over the three connections; plotted as the Pareto fronts per connection (section 4.1.4), baseline 74.28 %.

JPEG reaches 3.80 within the budget, the highest ratio; Lorenzo follows with 2.88, then QH (2.24), dynamic ASZ (2.05), static ASZ (2.00) and SZ1 (1.78). SVD and Huffman coding reach 1.23 each; RLE (0.91) and zero RLE (0.89) expand the three connections, and downsampling has no step within the budget. Against the mean of the single-connection bests on the same three connections (Fig. 4.10), JPEG falls from 4.6 to 3.8, QH from 3.6 to 2.2, Lorenzo from 3.4 to 2.9, static ASZ from 2.4 to 2.0, SZ1 from 2.2 to 1.8 and SVD from 2.1 to 1.2.

Overall JPEG has the highest ratio in the RegNetX-400MF use case, in the ResNet-18 and MobileNetV2 use cases on ImageNet and in all four CIFAR use cases, Lorenzo in the MobileNetV3-Small one, and Lorenzo is second in the other six use cases with $K = 1$. Against the mean of the single-connection bests, over the seven use cases with $K = 1$, JPEG loses 17 to 55 % of its ratio, SVD 4 to 72 %, QH 12 to 38 %, SZ1 0 to 26 % and Lorenzo 0 to 15 %: the transforms, which do not bound the error per value, lose the most, the error-bounded Lorenzo and SZ1 the least; the exception is SVD on ImageNet MobileNetV2, which loses 4 %. In the four CIFAR use cases, both ASZ variants lie below Lorenzo and QH, above SZ1 on CIFAR-100 and below it on CIFAR-10. In the ResNet-18 and MobileNetV2 use cases on ImageNet, they lie below Lorenzo but above QH and SZ1. In every use case, the best lossy method still stays 1.5 to 4.3 times above Huffman coding.

4.4 Hardware Cost Estimates

As in section 3.1.5, we count every multiplication, division, addition, subtraction, absolute value and comparison a compressor performs per value as one operation of arithmetic cost, without weighting them by their implementation cost; a multiply-accumulate (MAC) counts as two. Table lookups are not counted as arithmetic; the tables count towards buffer space instead. Buffer space is counted in values or table entries per channel, including constants; a table shared by all

channels of a layer is charged with $1/C$ of its size. Both arithmetic and buffer space are replicated C times in the fully unrolled design, which gives the cost of a layer. As in chapter 3, we count the compressor side only, since the decompressor mirrors it. Where the decompressor differs structurally, as for downsampling and SVD, it is listed as a separate stage but not included in the total. These figures are estimates, derived from our software implementation of each compressor and, for SVD and JPEG, from the cost reported in related work. They do not replace synthesis results and should be taken as a first, relative comparison rather than as exact hardware cost. Tables 4.4 and 4.5 summarize the result, split into the baselines without prediction and the prediction-based compressors.

Compressor	Stage	Arithmetic per value	Buffer space per channel (up to)	Notes
Huffman (section 4.1.3.1)				
	Huffman coding	0	2^8	table lookup; one table per channel, one entry per input value
RLE (section 4.1.3.2)				
	Run-length coding	3	2	previous value and run counter; variable rate; expands short runs
Zero RLE (section 4.1.3.2)				
	Zero run-length coding	3	1	run counter; variable rate; expands connections without zeros
Quantization + **Huffman** (section 4.1.3.6)				
	Quantization	5	2	m_c and s_c; fixed rate; alternatively a table of 2^8 entries
	Huffman coding	0	$2^b/C$	table lookup; one static table of 2^b entries per layer
	Total	5	$2 + 2^b/C$	
Downsampling (section 4.1.3.3)				
	Strided subsampling	0	0	lower bound; evaluated with average pooling
	Nearest-neighbour upsampling	0	W/f_{down}	decompressor side; evaluated with bilinear upsampling
	Total	0	0	
SVD (section 4.1.3.4)				
	Projection $U_c^\top A\, V_c$	$2k\,(1 + k/W)$	$k\,(H + W) + k + k^2$	bases U_c, V_c, row and coefficient accumulators; latency of one featuremap
	Coefficient requantization	$5k^2/(HW)$	$2k^2$	coefficient ranges
	Reconstruction $U_c\, C\, V_c^\top$	$2k\,(1 + k/W)$	k^2	decompressor side; bases shared with the compressor
	Total	$2k\,(1 + k/W) + 5k^2/(HW)$	$k\,(H + W + 1) + 3k^2$	
JPEG (section 4.1.3.5)				
	8×8 DCT	16	$8W + 64$	8 rows and one block; AP-922 factorization [61]
	Quantization	2	$64/C$	one quantization table of 64 entries per layer
	Run-length and Huffman coding	2	$174/C$	fixed DC and AC tables of 12 and 162 entries per layer
	Total	20	$8W + 64 + 238/C$	latency of one block row

Table 4.4: Estimated hardware cost of the baselines without prediction, per stage, per channel and per value, compressor side only; all costs are replicated C times in the fully unrolled design. Decompressor stages are listed but not included in the total. —: none beyond the stage above. W and H are the featuremap width and height, f_{down} the downsampling factor and k the SVD rank.

Compressor	Stage	Arithmetic per value	Buffer space per channel (up to)	Notes
Lorenzo + Huffman (section 4.1.3.7)				
	Lorenzo prediction	4	$W + 3$	`layer1` and clamp; row buffer and m_c, M_c
	Residual quantization	7	1	ε_c
	Huffman coding	1	$2\,(2^8 + 1)/C$	one table of $2^8 + 1$ entries per layer, rebuilt per batch from as many counters
	Total	12	$W + 4 + 514/C$	
SZ1 + Huffman (section 4.1.3.8)				
	Prediction (constant, linear, polynomial)	7	3	
	Encoding decision	8	—	
	Error bounding (τ_v)	1	1	Δ_c^v
	Huffman coding ($n = 2$)	$\frac{1}{2}$	$32/C$	one table of 16 entries per layer, rebuilt per batch from as many counters
	Total	16.5	$4 + 32/C$	
ASZ, static (section 4.1.2)				
	Prediction (active set, $K = 1$)	≤ 40	$W + 2$	row buffer and μ_c; `random`: uniform draw from an LFSR [63], logarithm as lookup table; 7 for the SZ1 set
	Encoding decision	8	—	
	Error bounding (τ_v, τ_b)	4	4	Δ_c^v, Δ_c^b, error sum and budget
	Huffman coding ($n = 2$)	$\frac{1}{2}$	$32/C$	as SZ1 + Huffman
	Total	≤ 52.5	$W + 6 + 32/C$	row buffer shrinks to $W/f_{down} + 1$ with downsampling
ASZ, dynamic (section 4.1.2)				
	Prediction (pool, $K = 1$)	108	$W + 2$	all 21 pairs of table 3.3; `random` as above
	Encoding decision and swap statistics	107	45	rescue and mean-error accumulators per pool member, 3 sole-cover counters
	Error bounding (τ_v, τ_b)	4	4	Δ_c^v, Δ_c^b, error sum and budget
	Huffman coding ($n = 2$)	$\frac{1}{2}$	$32/C$	as SZ1 + Huffman
	Total	≈ 220	$W + 51 + 32/C$	swap of at most $C\,(2+\lceil\log_2 21\rceil) = 7C$ bits per batch

Table 4.5: Estimated hardware cost of the prediction-based compressors, per stage, per channel and per value, compressor side only; all costs are replicated C times in the fully unrolled design. —: none beyond the stage above. W is the featuremap width and f_{down} the downsampling factor.

Per-channel compressors All compressors except JPEG run one instance per channel (section 3.1.1). This lets them adapt to the particularities of every channel, whose distributions differ considerably (appendix A.1), but it cannot exploit correlations between channels, although the featuremaps of a layer often show similar regions across channels. Block-based schemes that exploit such similarities exist [86], and compressors such as RLE might work better on interleaved channels. On the other hand, some channels are similar enough that they could share a compressor instance, which would reduce the replication by C. Which of these options pays off cannot be decided without concrete hardware designs.

Lossless baselines Huffman coding is a table lookup per value without arithmetic or buffering; its cost lies in the C code tables of up to 2^8 codewords (section 4.1.3.1). RLE compares every value with the previous one, increments the run counter and checks it for overflow, i. e. three operations, and holds the previous value and the counter. Zero RLE compares against zero instead and

needs no stored value. Both produce a variable rate, and both expand a stream whose runs are too short (section 4.1.3.2).

Quantization Requantization costs a subtraction of m_c, a multiplication by $1/s_c$, a rounding and the two comparisons of the clamp per value (section 4.1.3.6). Since the inputs are 8-bit values, it can just as well be implemented as a table of 2^8 entries per channel. The static Huffman table is shared by the layer. Without Huffman coding, quantization is the only compressor with a fixed rate, which simplifies the integration into a streaming accelerator, where a varying rate is harder to accommodate [80].

Downsampling We charge only the cost of strided subsampling and nearest-neighbour upsampling, the cheapest variant (section 3.6.1), while the results were obtained with average pooling and bilinear upsampling (section 4.1.3.3). The cost is therefore a lower bound and the comparison conservative in favour of the baseline: no arithmetic or buffering on the compressor side, and one row of the reduced featuremap on the decompressor side (table 3.4).

SVD No decomposition takes place at runtime: as in the SVD-based FPGA accelerators of Rizakis et al. [66] and Yu and Bouganis [92], the factors are computed offline and only multiplied with the data. We therefore take their cost model, in which multiplying by stored rank-k factors costs one MAC array per factor, and arithmetic and storage grow with the sum of the dimensions rather than their product. Applied to $C = U_c^\top A V_c$, this gives two MAC stages as in Yu and Bouganis [92]: k MACs per value for $A V_c$ and k^2 per row for $U_c^\top$, i. e. $k\,(1 + k/W)$ MACs per value. The requantization of the k^2 coefficients adds five operations per coefficient, which is negligible per value. Unlike in these works, the products do not replace the computation of a layer but are added to it, on both sides, and their cost is highest for the least aggressive compression, since it grows with k. Since $U_c^\top$ mixes all rows, the coefficients of a channel are complete only after its last value, and the decompressor needs all of them before its first output, which adds one featuremap of latency. For RegNetX-400MF `block1-0` ($C = 32$, $H = W = 56$) at $k = 8$, i. e. CR = 49, this amounts to about 9.1 MACs per value and side, about 256 MAC units for one value per channel and cycle, and 32,768 constants. Our SVD is simulated in 32-bit floating point; fixed-point bases and accumulators are not evaluated.

Prediction-based compressors Lorenzo, SZ1 and ASZ share the structure of table 3.4, and their entries follow from table 3.2. The single-layer Lorenzo predictor is `layer1` (2 operations) plus the clamp into $[m_c, M_c]$. It reads the north-west neighbour and thus needs a row buffer of $W + 1$ values. Its residual quantizer costs a subtraction, a multiplication, a rounding, the outlier check ($|q|$ and a comparison) and the reconstruction $\hat{x} + 2\varepsilon_c\, q$, i. e. seven operations (section 4.1.3.7). SZ1 needs only the west-only functions (7 operations, 3 buffered values), the encoding decision for three active functions (8, section 3.4.5) and the τ_v comparison. The adaptive Huffman stages are table lookups; updating the code per batch adds one counter increment per symbol, i. e. half an operation per value for blocks of $n = 2$ codes (section 3.6.2).

For the static variant of ASZ, the arithmetic depends on the active set of every channel. With a single buffered row, the most expensive active set costs 40 operations and the SZ1 set 7 (section 3.4.5); the block error threshold adds a comparison, two additions and two registers (section 3.3.4). The static ASZ thus costs at most about three times the arithmetic of SZ1 + Huffman, and a row of buffer instead of three values. The dynamic variant has to instantiate the whole pool (section 3.5). With $K = 1$, it consists of the 21 distinct pairs of table 3.3,

which cost 108 operations together. On top of the encoding decision, the swap statistics need the error and the comparison against Δ_c^v of each of the 18 passive members (54), and an update of the rescue and mean-error accumulators of all 21 members and of the three sole-cover counters (45), in total 107 operations. Dynamic selection thus costs about four times as much as the static variant, which is why we consider static selection the practical design.

JPEG We take the factorization of the transform from the FPGA JPEG decoder of Nakahara et al. [61], which decompresses the input images of a streaming CNN accelerator and, as in our setting, uses fixed tables without per-image headers. The AP-922 factorization they implement takes 32 multiplications and 32 additions per one-dimensional transform of 8 values, i. e. 16 operations per value for the two-dimensional transform. Quantization adds a multiplication and a rounding per coefficient, and the zig-zag run-length coding a comparison and a counter increment. Taken together, JPEG is the most elaborate pipeline of the evaluation, with five stages that each need their own buffers or tables: the DCT, quantization, the zig-zag reordering, which needs the coefficients of a complete block, run-length coding and Huffman coding, whose variable-length codes have to be packed into the output stream. Its 20 operations per value therefore understate its cost: nine of them are multiplications, against one for QH, and each stage adds control logic and storage that an operation count does not capture. To form 8×8 blocks from the raster-ordered stream, JPEG has to buffer 8 rows per channel, plus a block for the transform, and adds a block row of latency on each side. In our evaluation, the channels share one coder as a mosaic (section 4.1.3.5); the per-channel figures assume one instance per channel like the other compressors.

Comparison Apart from downsampling, whose cost we underestimate, the lossless baselines and quantization are the cheapest compressors, with at most five operations and no row buffer. The prediction-based compressors need a row buffer, except for SZ1, and between 12 and about 50 operations in the static case. Like SVD, JPEG needs multipliers on the data path, nine per value, and buffers eight rows per channel instead of at most one, and with its five stages, it is the most elaborate pipeline of all compressors; SVD adds a whole featuremap of latency. The dynamic variant of ASZ is by far the most expensive compressor in arithmetic. Table 4.6 ranks the compressors qualitatively by their hardware cost.

	Compressor	Arithmetic per value	Buffer per channel	Cost driver	Cost
1	Downsampling	0 (lower bound)	0	nothing on the compressor side; one reduced row on the decompressor side	very low
2	Huffman	0	table of 2^8 entries	table lookup only	very low
3	Zero RLE	3	1	run counter	very low
4	RLE	3	2	previous value and run counter	very low
5	QH	5	$2 + 2^b/C$	requantization, or a table of 2^8 entries	very low
6	Lorenzo + Huffman	12	$W + 4 + 514/C$	row buffer; loop on reconstructed values	moderate
7	SZ1 + Huffman	16.5	$4 + 32/C$	no row buffer, but more arithmetic than Lorenzo; loop on reconstructed values	moderate
8	ASZ, static	27.5 typical, ≤ 52.5	$W + 6 + 32/C$	arithmetic of the active set: median 15, at most 40	moderate to high
9	JPEG	20, with multipliers	$8W + 64 + 238/C$	8×8 DCT with multipliers on the data path, one block row of latency, five stages	high
10	SVD	$2k\,(1 + k/W) + 5k^2/(HW)$, with multipliers	$k\,(H + W + 1) + 3k^2$	MAC array for the rank-k projection; one featuremap of latency	high
11	ASZ, dynamic	≈ 220	$W + 51 + 32/C$	all 21 pool members evaluated every cycle, plus the swap statistics	very high

Table 4.6: Qualitative ranking of the compressors by hardware cost, from cheapest to most expensive, compressor side, per channel and per value (tables 4.4 and 4.5). The cost of downsampling is that of strided subsampling, a lower bound for the evaluated average pooling. The typical cost of static ASZ assumes the median active set: over all 1330 sets of three of the 21 (function, variant) pairs of table 3.3, taken as equally likely, the prediction costs 15 FLOPs in the median (10 to 21 for the middle half), to which the encoding decision (8), error bounding (4) and Huffman coding ($\frac{1}{2}$) add 12.5. The order of ranks 8 to 10 depends on the weighting: JPEG and SVD need fewer or comparable operations, but multipliers on the data path, several rows to a whole featuremap of buffer, and latency.

4.4.1 Suitability for Streaming

Beyond arithmetic and buffer space, a compressor in the eviction path of a streaming accelerator has to accept the values in the order in which they are produced, one per cycle and channel, and hand a stream to the off-chip link, where a varying rate is harder to accommodate than a fixed one [80]. Table 4.7 assesses the methods qualitatively against these requirements: how much of a featuremap they have to buffer before they can emit, the latency this adds, whether their output rate is fixed or varies with the data, how far a stream can expand in the worst case, whether coding a value depends on the reconstruction of its predecessors, and which state compressor and decompressor have to keep in step.

Method	Buffering and added latency	Output rate, worst case	Loop on reconstructed values	Shared state
Huffman	none	variable; up to the longest codeword per value	no	static tables per channel
RLE, zero RLE	previous value and run counter; a record waits for the end of its run, at most 2^b values	variable; expands to $(8 + b)/8$ without runs or zeros	no	none
Downsampling	W/f_{down} partial sums, two reduced rows for the decompressor; about f_{down} rows of latency on each side	fixed, $1/f_{down}^2$	no	none
SVD	the whole featuremap of a channel; one featuremap of latency on each side	fixed, k^2/HW	no	constant bases
JPEG	8 rows; one block row of latency on each side	variable; can exceed 8 bits per value at high quality	no	constant tables
QH	none	fixed at b bits without Huffman coding; variable with it, up to the longest codeword	no	static table per layer
Lorenzo	one row of reconstructed values; no latency	variable; an outlier costs its symbol and 8 raw bits	yes	Huffman table adapted per batch
SZ1	three reconstructed values; no latency	variable; an outlier costs its code and 8 raw bits, about 10/8 before Huffman coding	yes	Huffman table adapted per batch
ASZ	one row of reconstructed values; no latency, unless downsampling is chosen	variable; as SZ1	yes	Huffman table adapted per batch; dynamic: also the selection

Table 4.7: Qualitative assessment of the compressors for the streaming scenario, compressor and decompressor of one channel. Rates are given relative to the 8-bit input; b is the length field width of RLE, f_{down} the downsampling factor and k the rank of SVD.

Three properties separate the methods. First, the transforms need the most buffering before they can emit anything, a whole featuremap per channel for SVD and eight rows for JPEG, which adds latency on both sides of the link; all other methods work on the values in the order in which they arrive, with at most a row of history. Second, only downsampling, SVD and QH without its Huffman stage have a fixed output rate, which lets the bandwidth of an evicted connection be planned exactly. All entropy-coded methods produce a rate that varies with the data, and RLE, Lorenzo and the SZ-type compressors can even expand a stream, so that the link has to be provisioned for the worst case or the stream has to be buffered. Third, Lorenzo, SZ1 and ASZ predict from reconstructed values, so that coding a value depends on the reconstruction of its predecessor. At one value per cycle, this loop has to close within a cycle, the main challenge of the first FPGA implementation of SZ, which resolves it with a staged pipeline [88]; since every channel has its own instance, the loop only constrains the values within a channel. Finally, Lorenzo, SZ1 and ASZ adapt their Huffman tables per batch, and dynamic ASZ also its selection, so that compressor and decompressor have to update this state in step, while Huffman coding and QH use static tables.

4.5 Summary and Discussion

Tolerance to approximation We find that individual skip connections are surprisingly tolerant to approximation. When a single connection is compressed,

a lossy method exceeds the ratio of the best lossless method on every one of the 74 connections, on 71 of them at a cost of at most 0.1 pp and on 73 at most 0.3 pp. How far a connection can be compressed depends strongly on the connection, on its depth as well as on its type: cutting a single connection costs least for the late connections and, in RegNetX-400MF and the ResNets, for those that carry the output of a projection (appendix A.9.1), the sensitivity to quantization falls with depth (appendix A.9.2), and on a few insensitive connections, the methods that can drop values reduce the featuremap to a single value per channel. How far a connection can be compressed losslessly, in contrast, is decided by its operand rather than by the network: on the 41 connections without zeros, the linear bottlenecks and the projections, RLE reaches at most 1.0 and Huffman coding at most 1.47, while on the ReLU outputs with at least 15 % zeros, Huffman coding reaches a median of 1.79 (section 4.2.4).

Multiple connections The advantage of lossy compression shrinks when several connections are compressed at once. Without retraining, the errors introduced at different connections accumulate, so that a fixed accuracy budget has to be shared among all lossily compressed connections. With three connections, the methods lose up to 72 % of their single-connection ratio, the transforms, which control their error only indirectly, the most, and the error-bounded Lorenzo and SZ1 the least, and on MobileNetV3-Small, JPEG, QH and SVD even fall below Huffman coding. On a single connection, an indirect control of the error suffices, since every method is held within the budget by its measured accuracy rather than by its error bound (section 4.1.4); once the errors of several connections add up, the methods with a bound per value keep more of their ratio. The best lossy method still stays 1.5 to 4.3 times above Huffman coding in every use case. Lossy compression is therefore best reserved for a few selected connections, such as those with the largest feature maps, while the remaining ones are encoded losslessly.

Best methods on single connections For individual connections, JPEG reaches the highest ratio on 50 of the 74 connections, on 23 of the 25 in the first third of their network, but only on 12 of the 25 in the last third. QH leads on 13, all of them in the middle or at the end of a network, where the connections tolerate coarse quantization, and Lorenzo on 6, all of them on the MobileNets on ImageNet; the two quantizers follow JPEG most closely and come within a factor of 2 of the best on about two thirds of the connections. Only the transforms and downsampling go beyond the limits that their coding imposes on the other methods, 8:1 for the quantizers and about 4:1 before Huffman coding for SZ1 and ASZ, which matters on the insensitive connections (section 4.2.4). Between the two quantizers, depth decides: Lorenzo is ahead of QH on 21 of the 25 connections in the first third, on 8 of 24 in the middle and on only 4 of the 25 in the last third.

Lorenzo and QH The split follows from what each of them codes. The deep connections tolerate coarse quantization (appendix A.9.2), and after a ReLU, their values are skewed, which the Huffman stage of QH exploits directly. On the early connections, the predictor of Lorenzo pays off. A comparison at a fixed error bound isolates how well it fits the data: at a maximum error of about 3 % of the channel range ($\rho = 0.03$ against 4 bits), Lorenzo reaches only 0.83 times the ratio of QH on connections with at most 100 values per channel, but 1.25 times on those with more than 1,100, and its advantage has a rank correlation of 0.69 with the featuremap size within a network. The large featuremaps, which mostly lie early in a network and still resemble the input image, are smooth enough for the predictor to concentrate the residuals around zero.

ASZ ASZ is ahead of SZ1 on 62 of the 71 connections on which it holds the budget, through a better SZ stage rather than through its Huffman stage, but it trails the better of the two quantizers on 59 of them and is never the best method. Dynamic selection does not pay off: it is practically identical to the static variant at about four times the arithmetic. ASZ does best on the large early featuremaps, which are both the most expensive to evict and the most sensitive to approximation: on the largest connection of RegNetX-400MF, ResNet-18 and MobileNetV3-Small, it is ahead of both quantizers, by up to a factor of 1.4, but JPEG compresses these connections 1.3 to 2.6 times more.

Quantizers and prediction-based compressors The quantizers lead the prediction-based compressors because of what they code: QH and Lorenzo reach higher ratios than SZ1 and ASZ on 61 of the 74 connections, although all four bound the error per value. The quantizers spend their bits on the value itself or on its prediction residual, so that every additional bit refines its magnitude. SZ1 and ASZ act as decorrelators instead: they code only which of their predictors reconstructs a value within the threshold, and a value that none of them reconstructs costs its full 8 bits. Their 2-bit codes carry no information about the magnitude of the error, and they limit the SZ stage to 4:1.

The Huffman stage widens the gap. On the post-ReLU connections, the values fall off roughly exponentially (appendix A.5.1), so that the quantized values of QH are skewed and its Huffman stage gains a factor of up to 2.1 (section 4.2.1.2). The code stream of a prediction-based compressor is skewed only if one predictor dominates, as the constant function of SZ1 does. The better the predictors represent the values, however, the more evenly the hits spread over them: the per-channel selection of ASZ favours predictors that complement each other, which minimizes the outliers but is blind to how evenly the hits spread over the active slots. Its codes are therefore spread almost evenly, and Huffman coding, which cannot go below the entropy of the codes, adds a median factor of only 1.08 on RegNetX-400MF (section 4.2.1.1).

Cost of an evenly spread code stream This cost can be measured directly. On block1-0 of RegNetX-400MF, we ran the same seven Pareto points of static ASZ with two frozen selections: one fitted from a smaller pool of predictors, in which constant predictors fill 64 of the 96 slots and a single slot takes most hits, and one fitted per point from the full pool, in which `linreg` predictors fill 19 to 26 slots and complement the others. With the second selection, the SZ stage alone is as good or slightly better (3.16 to 3.50 against 3.12 to 3.49), i. e. it leaves as few or fewer outliers. Its code entropy, however, rises from 1.25–1.46 to 1.62–1.69 bits, the gain of the Huffman stage falls from 1.24–1.44 to 1.13–1.17, and the total ratio from up to 5.0 to at most 4.1, at accuracies within about 0.1 pp of each other. We have not verified whether the `linreg` predictors themselves cause the more even distribution or how the first selection was fitted; the effect of the evened-out code distribution on the Huffman stage, however, is measured.

Hardware cost and streaming On the hardware side, QH and the lossless methods are the cheapest compressors, while JPEG and SVD need multipliers on the data path and buffer eight rows or a whole featuremap per channel (sections 4.4 and 4.4.1). The prediction-based compressors and Lorenzo lie in between in cost, but predict from reconstructed values, which forms a loop in the pipeline. Only downsampling, SVD and QH without its Huffman stage have a fixed output rate; the rate of all entropy-coded methods varies with the data, and RLE, Lorenzo and the SZ-type compressors can even expand a stream, so that for most lossy methods, as for the lossless ones, the bandwidth of an evicted connection cannot be planned exactly. Which method improves an accelerator

the most therefore depends on whether its mapping is bound by bandwidth or by on-chip resources (section 4.1).

5 Conclusion

In this thesis, we explored the suitability of prediction-based compression for the evicted skip-connection activations of streaming CNN accelerators. We asked how far the compression ratio of these activations can be increased beyond lossless encoding by trading a bounded loss of inference accuracy, using compressors that meet the requirements of section 1.2.

For a single connection, lossy compression goes well beyond lossless encoding. On every one of the 74 connections we evaluated, a lossy method exceeds the best lossless one, on 71 of them at a cost of at most 0.1 pp (section 4.5). The compressors that meet the requirements, QH, Lorenzo and ASZ, reach geometric means of 3.9, 3.8 and 3.5 within a budget of 1 pp, against 2.7 for SZ1. This is 2.3 to 2.6 times the 1.5 of Huffman coding, while RLE does not compress on average (1.0). JPEG, with a geometric mean of 6.6, shows what relaxing the streaming requirement can buy: it buffers 8 rows per channel and needs multipliers on the data path. SVD, which buffers a whole featuremap, gains nothing over the quantizers (3.2, sections 4.4 and 4.4.1).

The more connections are compressed lossily, however, the lower the ratio within the budget, since their errors add up and the budget has to be shared among them. From one to three connections, almost every method loses ratio, up to 72 % for SVD; the transforms lose the most, the error-bounded Lorenzo and SZ1 the least. Compressing the first three connections of RegNetX-400MF together, JPEG reaches 4.5, Lorenzo and static ASZ 2.55 and QH 2.28, against 1.46 for Huffman coding. In every use case, the best lossy method still stays 1.5 to 4.3 times above Huffman coding, but on MobileNetV3-Small, JPEG, QH and SVD already fall below it with three connections (section 4.3). Since we evaluated at most three connections at a time, we can only extrapolate this trend: beyond some number of connections, we expect lossless compression alone to perform better on average within the budget. Mixed schemes could then compress a few connections lossily, e. g. the largest ones, and encode the others losslessly.

5.1 Overview of work

In chapter 2, we introduced acceleration architectures, CNN architectures and the issues and benefits of mapping CNNs to FPGAs in a streaming, dataflow manner. We introduced the basics of model compression, presented some fundamental work in that area and showed how previous work dealt with scaling issues. We then explored different compression techniques in search of an intuitively good fit for the streaming compute model and decided on a prediction-based compression model, which we explored more deeply in a design space exploration.

In chapter 3, we conducted this design space exploration to evaluate the fitness of the SZ prediction-based compression model in the context of streaming accelerators. We explored block error thresholds, extended the predictor set with novel predictors, implemented a dynamic selection mechanism to guide the selection of prediction functions, evaluated pre- and postprocessing stages for the compression module and presented a method to find good parameters for the compressor.

In chapter 4, we compared the design created in the DSE to a number of baselines from related work that seemed particularly strong in this context. The comparison covered a number of networks and included an estimation of the hardware cost to ease the transfer of the results to real hardware.

5.2 Findings

We structure our findings along the three objectives of the introduction: what we learned about the activations of skip connections, about adapting SZ1 to them, and from comparing the resulting compressor with the alternatives. We then revisit the requirements of section 1.2 in their light.

5.2.1 Characterization of Skip-Connection Activations

Tolerance to approximation Individual skip connections are surprisingly tolerant to approximation, as our preliminary experiments indicated (appendix A.9.2): a lossy method beats the best lossless one on every connection, almost always at a negligible cost (section 4.5). The tolerance, however, varies strongly between connections, which motivates configuring the compressor per connection; even the best method changes with depth (section 5.2.3). The sensitivity to quantization falls with depth (appendix A.9.2), and cutting a connection costs least for the late connections and for those that carry the output of a projection (appendix A.9.1). Some connections are almost completely redundant: cutting them costs little accuracy, and on five of them, transferring only the average of every channel instead of the whole featuremap keeps the accuracy within the budget (section 4.2.4).

Lossless compressibility How well a connection compresses losslessly is decided by its operand rather than by the network. ReLU outputs are skewed towards small values and contain many zeros (appendices A.1 and A.5.1), while projections and linear bottlenecks contain none; 41 of the 74 connections, among them all connections of MobileNetV2, are free of zeros (section 4.2.4).

Channel distributions The value distributions also differ considerably between the channels of a connection, which motivates compressing every channel separately (appendix A.1).

Error distribution Not only the magnitude of the error matters, but also its distribution: the same average error costs more accuracy when it is concentrated in a few large errors than when it is spread evenly (appendix A.3.2).

5.2.2 Adaptation of SZ1

The conclusion of chapter 3 details the lessons of the design space exploration (section 3.8); all of them stem from a single, deliberately sensitive connection.

Code width After our improvements to the prediction stage, the code width is the main limiting factor of the SZ1 model: with 2-bit codes, the ratio is bounded by 4:1, towards which plain SZ1 saturates. Better prediction functions improve the ratio at a given accuracy, since they leave fewer outliers, but they cannot go beyond this bound. Going beyond it requires stages outside the prediction stage, Huffman coding and downsampling. Together, block error thresholds, our new prediction functions, per-channel predictor selection and Huffman coding lift the ratio on the DSE connection from about 4 to 7.4, and to about 14 with downsampling, within a budget of 3 pp (section 3.8.1). At the 1 pp budget of the evaluation, ASZ reaches 4.3 on this connection, without downsampling (section 4.2.1.1).

Error bounding Bounding the error does not bound the accuracy loss. The block error threshold acts as a soft limiter of the accuracy loss, and together with the value error threshold and the block size, it also steers the distribution of the error to some extent; the simplest and cheapest block type, linear blocks, worked best (section 3.3).

Function interactions Prediction functions interact and cannot be judged in isolation. Every prediction builds on values reconstructed by other functions, so that replacing a single function changes how the others are used, and a single buffered row ($K = 1$) reaches the same front as three, at a third of the buffer.

Cheap predictors For the same reason, cheap and unconventional predictors can outperform sophisticated ones: a random predictor, without buffer or arithmetic on the data path, retains more accuracy near the 4:1 limit than the polynomial or linear function it replaces, by covering the positions at which the deterministic functions collapse into each other. By changing the reconstructed values that the other functions predict from, it can also change how well they are used.

Spatial predictors Prediction functions that read from the row above, such as the PNG filters and the Lorenzo predictor, improve the accuracy at equal ratio. They come at the cost of a row buffer of about W values per channel, compared to at most three values for the west-only functions of SZ1 (sections 3.4.2 and 3.4.5).

Offline adaptation Adaptation is best used offline. The best configuration found by the dynamic selection, fixed for the whole run, dominates the fully adaptive scheme (section 3.5). In the evaluation, static and dynamic selection perform almost the same, so that the dynamic variant, at about 220 operations per value and four times the arithmetic of the static one, does not pay off (section 4.4).

Pipeline interactions A better-fitting predictor set can hurt the pipeline: it leaves fewer outliers, but spreads the codes more evenly, which the Huffman stage cannot exploit. Two selections with nearly the same ratio after the SZ stage ended at 5.0 and 4.0 after Huffman coding (section 3.6.2). Huffman block codes of $n = 2$ add another 18 to 26 % (section 3.6.2).

Parameter search The parameter search is cheap and transfers to unseen data, with a fit slope of 0.89 between the accuracy drops on the calibration and on the test data (section 3.7.2). The exception is CIFAR, whose calibration split is taken from the training set, on which the networks reach almost 100 % accuracy, and is therefore less representative of the test data (section 4.1.1).

5.2.3 Comparison

The summary of chapter 4 details these results (section 4.5).

Lossless compression Lossless compression is not enough: Huffman coding and RLE do not work well for the particular set of models and datasets we examined, and RLE even expands most connections, since many of them contain no zeros (section 4.2.4). This need not contradict the gains that SMOF reports for RLE (section 4.5).

Choice of method Which method compresses a connection best depends on its depth, so that not only the configuration but also the method should be chosen per connection. On the early connections, JPEG leads almost everywhere and Lorenzo is ahead of QH; on the late ones, which tolerate coarse quantization, QH takes over from Lorenzo and the lead of JPEG shrinks (sections 4.2.4 and 4.5). Only the methods that can drop values or coefficients go beyond the limits that coding imposes on the others; on insensitive connections, the design of a compressor therefore matters more than how well it codes (section 4.2.4).

JPEG and QH QH works exceptionally well and is very cheap. Post-ReLU and deeper featuremaps have a low entropy and survive heavy quantization, and their skewed distribution survives it as well, so that Huffman coding compresses the quantized values very effectively. QH is second only to JPEG in ratio, and it is the cheapest lossy compressor (sections 4.2.4 and 4.4). JPEG compresses best overall, but its pipeline is the most expensive and fits streaming the least (sections 4.4 and 4.4.1).

ASZ ASZ improves on SZ1: it is ahead on 62 of the 71 connections on which it holds the budget, through a better SZ stage rather than through its Huffman stage. It is, however, never the best method and trails the quantizers on most connections. It does best on the large early featuremaps, which are the most expensive to evict, but JPEG still compresses them more (section 4.5).

Prediction model The prediction model of SZ1 fits the streaming scenario, as the introduction assumed (chapter 1 and section 4.4.1), but its representation limits the ratio: QH and Lorenzo compress better than SZ1 and ASZ on most connections (section 4.5). The quantizers spend their bits on an additional error term, the value itself or its residual, while SZ1 and ASZ spend them on the choice of a prediction function. The former delivers additional information about the signal, the latter a distribution that is unrelated to it. The better the predictor of Lorenzo fits, the more its residuals concentrate around zero, which its Huffman stage exploits; it fits best on the large, smooth featuremaps early in a network. Well-fitting predictors of a decorrelator such as SZ1, in contrast, spread its codes evenly, which is very bad for entropy coders. A dictionary coder, with which SZ1 itself compresses its code stream [13], could only do better if the choices of prediction function repeat over longer ranges: neighbouring choices are almost independent, as the joint entropy of code pairs lies barely below that of single codes (section 3.6.2). Dictionary coders are, however, too expensive for an already constrained hardware setting [44]. Lorenzo, the predictor of SZ-1.4, keeps the streaming fit of SZ1 with one buffered row, and later SZ versions build on its model (section 2.6.2.1).

5.2.4 The Requirements Revisited

No retraining All compressors are applied post hoc. Without retraining, however, nothing compensates the errors they introduce, so that the errors accumulate when several connections are compressed (section 4.3).

Controllable error The introduction argued that bounding the magnitude of the error is not enough, since its distribution matters as well (section 1.2). The characterization confirmed this: the same average error costs more accuracy when it is concentrated (appendix A.3.2). ASZ therefore adds the block error threshold, which acts as a soft limiter of the accuracy loss and, together with the block size, limits how far the error can concentrate (sections 3.3 and 3.8). In the evaluation, however, the baselines are held within the budget by their measured accuracy on the test split rather than by their error bounds (section 4.1.4), so that on a single connection, even the indirect control of JPEG and SVD through quality and rank suffices. ASZ is different: its error bounds are chosen on calibration data, and they hold up on the test data, except for a few connections on CIFAR (section 4.1.1). The bound per value pays off once the errors of several connections add up: from one to three connections, JPEG loses 17 to 55 % of its ratio and SVD up to 72 %, against 0 to 15 % for Lorenzo and 0 to 26 % for SZ1; QH, which also bounds the error per value, lies in between with 12 to 38 % (section 4.3).

Streaming operation The transforms need the most buffering, a whole featuremap per channel for SVD and eight rows for JPEG, and add latency on both sides of the link; all other methods work on the values in arrival order, with at most a row of history (section 4.4.1). Lorenzo, SZ1 and ASZ predict from reconstructed values, a loop that has to close within a cycle, as in the staged pipeline of GhostSZ [88], and they adapt their Huffman tables per batch, which compressor and decompressor have to do in step. The introduction noted that the rate of lossless codes varies with the data, so that the bandwidth of an evicted connection cannot be determined exactly at compile time (section 1.2). Most lossy methods share this property: only downsampling, SVD and QH without its Huffman stage have a fixed rate, and RLE, Lorenzo and the SZ-type compressors can even expand a stream (section 4.4.1).

Low hardware cost QH and the lossless methods are the cheapest compressors, with at most five operations per value and no row buffer. Lorenzo and the static prediction-based compressors need between 12 and about 50 operations and, except for SZ1, a row buffer. JPEG and SVD need multipliers on the data path, and dynamic ASZ is by far the most expensive compressor in arithmetic (section 4.4). Which method improves an accelerator the most cannot be decided from ratio and cost alone; it depends on whether its mapping is bound by bandwidth or by on-chip resources (section 4.1).

5.3 Reflections

In this section, we reflect on some of the choices we made in this thesis.

DSE Layer We chose the connection for the design space exploration, RegNetX-400MF block1-0, for its sensitivity, and because we saw early on that the compression model can achieve a good performance on it. It is, however, not representative of skip connections in general: on most other connections, QH compresses better. We have thus essentially optimized for a case that most methods do not handle well. On block1-0, ASZ reaches 4.3 within 1 pp, ahead of Lorenzo (3.0), SZ1 (2.2) and QH (1.8, table A.1). Only JPEG does better there (11.3), but its lead has to be weighed against its weak fit to streaming: it buffers eight rows per channel instead of one, which eats into the on-chip memory that eviction is meant to free, adds a block row of latency on both sides and produces a variable rate (section 4.4.1), and its five-stage pipeline is expensive in itself, with nine multiplications per value (section 4.4).

Prediction Model We set out to find better prediction functions and kept faith in the prediction model of the first SZ version, although most of the research has moved on to pipelines of a more sophisticated predictor, quantization, Huffman coding and a dictionary coder (section 2.6.2.1). We chose to explore the SZ1 model regardless, as its utility for streaming compressors was largely unexplored. We have now seen that the complete decorrelation that results from mapping the value stream to coded choices of prediction function does not work as well as methods that spend their bits on the values themselves or on their residuals, such as QH and Lorenzo.

Predictor Optimization and Pipeline If the goal is the maximum compression ratio, the complete pipeline has to be considered, since its stages can influence each other considerably through the distribution of the data they output. Our improvements to the prediction functions in fact reduced the efficacy of the Huffman stage, as they made the distribution of the code stream more uniform. Since we initially only looked at the compression ratio of the SZ stage, we largely missed this in the beginning. The objective of the selection, the fewest outliers, and that of the pipeline, the fewest bits after Huffman coding, diverge: adding predictors, e. g. the regression ones, can push the selection towards the former at the cost of the latter. When designing a compressor, the potential of its stages to affect each other's efficacy therefore has to be kept in mind; the predictor selection, for instance, should be scored on the entropy of the codes or the size after Huffman coding, not on the coverage of outliers alone.

5.4 Limitations and Future Work

Hardware Designs Due to time constraints, we decided to leave concrete hardware implementations out of the scope of this thesis and instead focused on providing a broader view of the design space. This makes it harder to compare our work to the current state of the art: SMOF [80], which served as the baseline and motivation for this thesis, reports the direct impact of compression on the overall system performance. Given these restrictions, we chose the trade-off between compression ratio and inference accuracy as our main evaluation metric. The compression ratio serves as a proxy for possible performance improvements of the hardware design, complemented by estimates of the hardware cost of the techniques we examined.

Network Architectures and Skip Connections We restricted our evaluation to image classifiers whose skip connections merge through element-wise addition, i. e. residual and inverted residual connections (section 3.1.1). Other merge operations, such as the concatenation of U-Net architectures [9], may produce activations with different characteristics, so that our results do not necessarily carry over to them.

Compression Aware Training Compression-aware training [4] could make the parameters of the compressors part of the training process and allow the model to learn how to deal with the errors that compression introduces.

Design Space Exploration The design space exploration was conducted on a single connection, RegNetX-400MF block1-0, which is not representative of most skip connections (section 5.3). The optimizations were tailored towards this connection, and the evaluation made clear that they do not transfer to the others. Similar explorations on connections of different depth, size and operand are needed. In addition, the predictor selection should be scored on the entropy

of the codes or the size after Huffman coding instead of the coverage of outliers alone (section 5.3).

Multi Layer Compression When several connections are compressed, it may pay off to use a different compressor on each of them, e. g. the best one for each connection; how the errors compound in this case remains to be evaluated. MemSZ [18] uses a third error bound to protect blocks that have already been compressed lossily from being compressed too much when they are compressed again. Something similar could be used for the compression of multiple connections with error-bounded compressors: the compressors could share how much error the earlier ones introduced, so that later compressors can adapt their error bounds for the featuremap dynamically, for a single sample or across multiple samples. This requires an analysis of the relative error tolerance of the connections. A relative error bound is not enough, since the same relative error may have a bigger impact on another connection. A factor per connection that represents its relative sensitivity would allow defining an error bound that spans multiple connections and can be adjusted to the accuracy budget.

Channel Similarities The channels are compressed completely in parallel: we focused on the streaming scenario and assumed full unrolling without folding or interleaving as a simplification. Our own initial experiments with correlations between channels did not seem promising, but Weng et al. [86] exploit them successfully and show clear correlations, much clearer than in our examples. Exploiting them deserves another try, although the benefit probably depends on the model. Since interleaving is often used to save on-chip space, an evaluation of channel similarities could also inform which channels can be interleaved and share the prediction functions of ASZ.

A Further Experiments and Evaluations

This appendix collects material that supports, but is not needed for, the main argument of the thesis. It opens with profiling results for the activations being compressed, followed by further experiments and evaluations that are supplementary to the design space exploration and the evaluation chapters.

A.1 Profiling

Before compressing anything, we profiled the activations to see what the data looks like. ResNet-20 was run over 5000 calibration images of CIFAR-10 with per-channel asymmetric linear quantization at 8 bits, and the quantized codes leaving each long-skip connection were histogrammed per channel over the full range of 256 codes.

Figure A.1 shows the histograms of `layer1.0`. The distributions are strongly skewed towards small codes and decay roughly exponentially, close to a straight line on the logarithmic axis. On average, only 119 of the 256 codes occur in a channel, and the per-channel entropy is 4.19 of the 8 stored bits. Channels 1 and 10 are constant: all of their 5.12 million samples fall into the zero bin. Deeper layers occupy more of the code range: in `layer2.0`, the histograms are unimodal humps over almost the entire range (230 codes, 6.50 bits), and in `layer3.0` they are broad and close to symmetric (225 codes, 6.62 bits; both not shown). Early layers therefore carry less information per stored byte than late ones and are correspondingly more compressible.

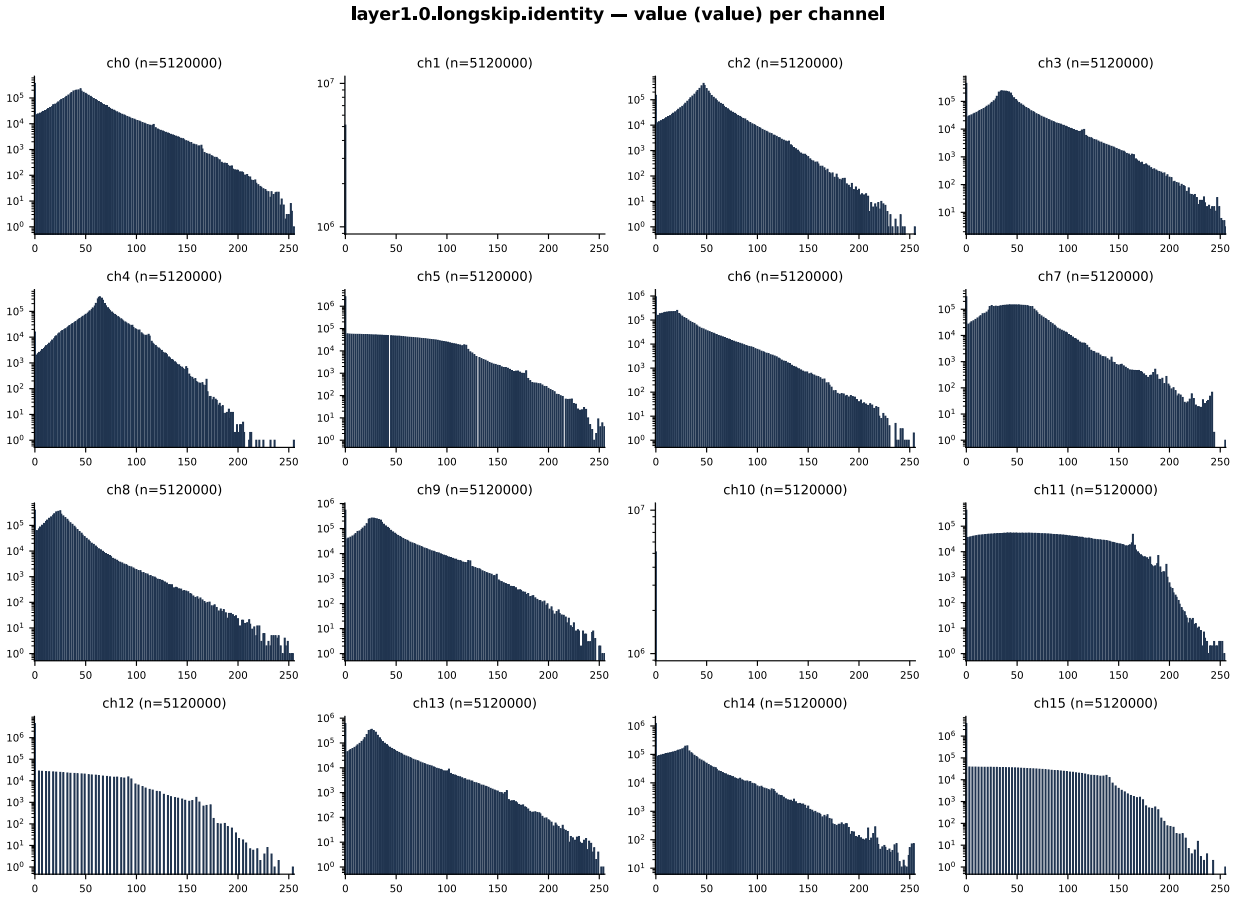


Figure A.1: Per-channel distribution of the 8-bit quantized activation codes leaving the long-skip connection of `layer1.0`, one panel per channel, counts on a logarithmic axis. The distributions are heavily skewed towards small codes and the upper half of the range is reached only rarely, leaving 4.19 bits of entropy per channel on average. Channels 1 and 10 appear empty because they are constant: all of their samples fall into the zero bin, which the logarithmic axis cannot show.

A.2 Linear Blocks vs. Sliding Window

Figure 3.19 compares the block types by their Pareto fronts. Figure A.2 looks at the same comparison along a single τ_v sweep, for a sliding window of 196 values and linear blocks of 224 values, the linear block size closest to the window, both at $\tau_b = 0.026$.

Up to $\tau_v \approx 0.06$, the two mechanisms behave practically the same. Beyond it, the sliding window is worse on both axes at once: its compression ratio saturates at about 2.78 instead of 2.88, and its accuracy keeps falling to 68.4 %, while that of the linear blocks levels off at about 69.5 %. Linear blocks are better in both accuracy and compression ratio at 15 of the 19 non-zero τ_v.

A likely cause is that the window never resets. Linear blocks discard their unused budget at every block boundary, which also caps how much budget a single prediction can draw on. The window instead carries the budget across row and featuremap boundaries, so a prediction can spend budget that accumulated in an unrelated part of the stream, and errors can concentrate where the stream crosses such a boundary. Since the sliding window is also more expensive in hardware (section 3.3.4), we did not pursue it further.

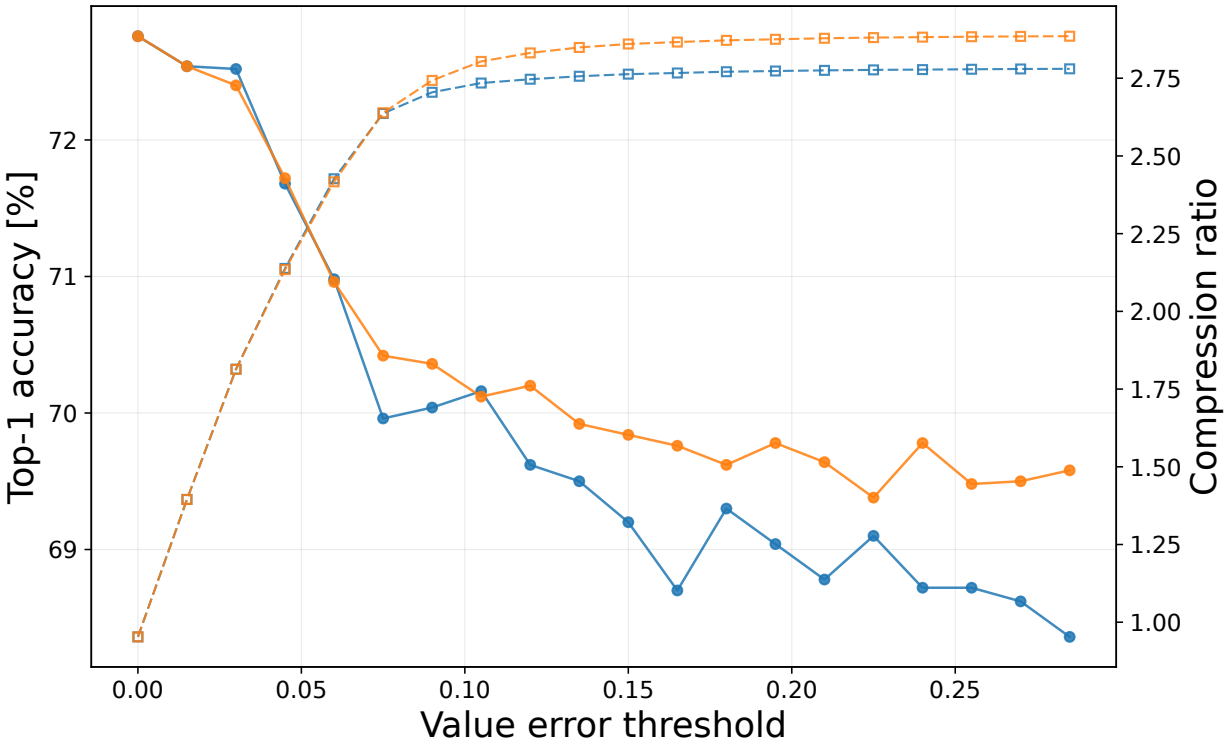


Figure A.2: Sliding window of 196 values (**blue**) vs. linear blocks of 224 values (**orange**), both at $\tau_b = 0.026$, over the same τ_v sweep, RegNetX-400MF, `block1-0`, calibration split. Circles and solid lines: top-1 accuracy (left axis); squares and dashed lines: compression ratio (right axis). Beyond $\tau_v \approx 0.06$, the sliding window reaches a lower compression ratio and loses more accuracy at the same time; it is dominated by linear blocks on both axes.

A.3 Frontloading vs. backloading

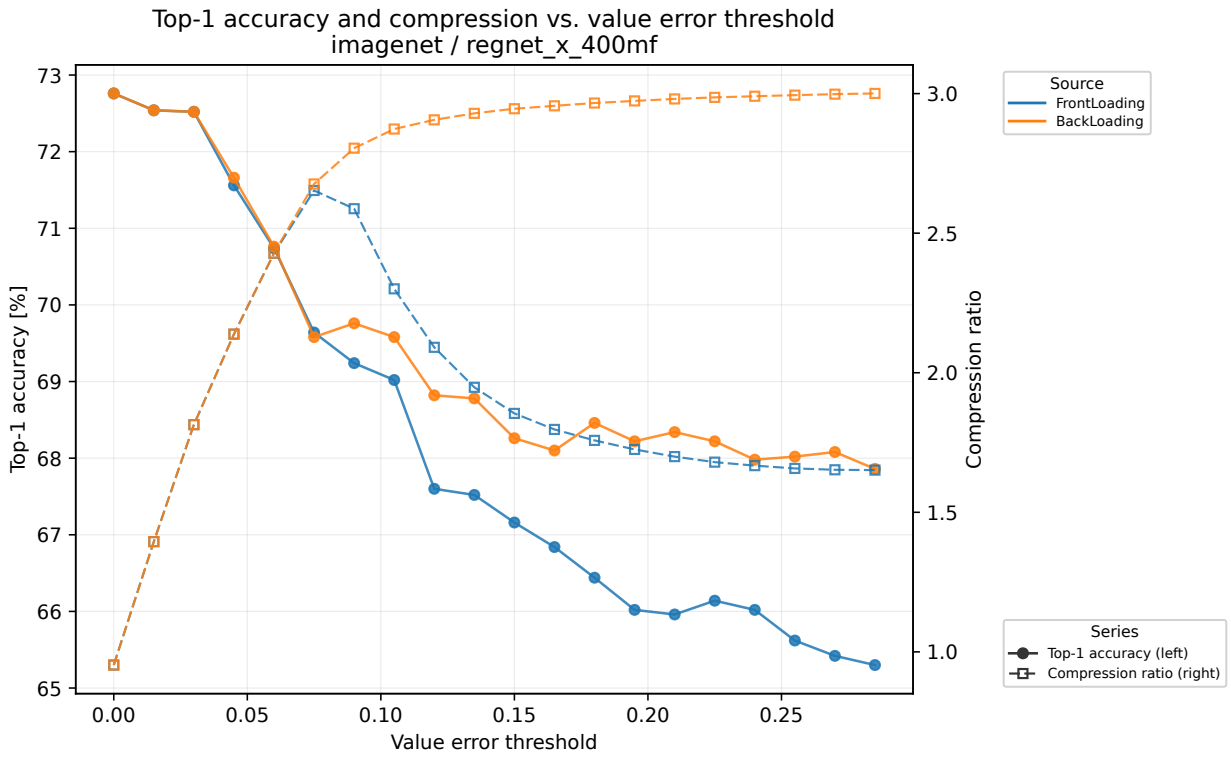


Figure A.3

A.3.1 Block Size and Block Error Interaction

Block Size (BS) and Block Error Threshold (BET) are both influencing the compression ratio - accuracy trade-off. While BET limits the average error within a block, BS can control how much budget can accumulate and thus how often and to what degree a single prediction can surpass the BET locally and use available error budget accumulated by below-BET prior predictions. These interactions are not easy to analyze, as much depends on the interactions between specific prediction functions that are being used: Different prediction functions (and sets of prediction functions) may create patterns in the data that are better or worse preconditions for successive predictions to fall within the error thresholds. Since an in-depth analysis into those interactions between prediction functions is not the focus of this work, we instead try to derive some useful heuristics from the experiments we conducted to inform our search for pareto-optimal points on the compression ratio - accuracy front.

A.3.2 Prediction sensitivity to error distribution across feature maps

When using both VET and BET, there is two extremes to how errors can be distributed across the feature map: **(1)** the error for each pixel in the featuremap is - individually - within the BET and distributed evenly within a block, or **(2)** the error focuses on a few pixels per feature map, using the budget gained by little to no error in other pixels to push to the VET.

In both cases, the average error stays within the BET, but is either distributed evenly, or focused on a few pixels.

A.3.2.1 Experiment Setup

To evaluate the two scenarios, we used the same layer as used throughout chapter 3, as it proved to be especially sensitive to approximations.

For scenario **(1)**, we determine the channel magnitude via a calibration pass and then use it to calculate the absolute maximum average error allowed within a given BET. We then either add to or subtract that absolute error value from each value in the channels. The choice whether to add or subtract is equally likely.

For **(2)**, we chose values from the feature map with a probability of p. Each of those values gets the full absolute error allowed within VET either deducted or added. The probability of a pixel to be selected for receiving an error is determined by $p = \frac{BET}{VET}$. With that probability, the average error will be the same as in (1)

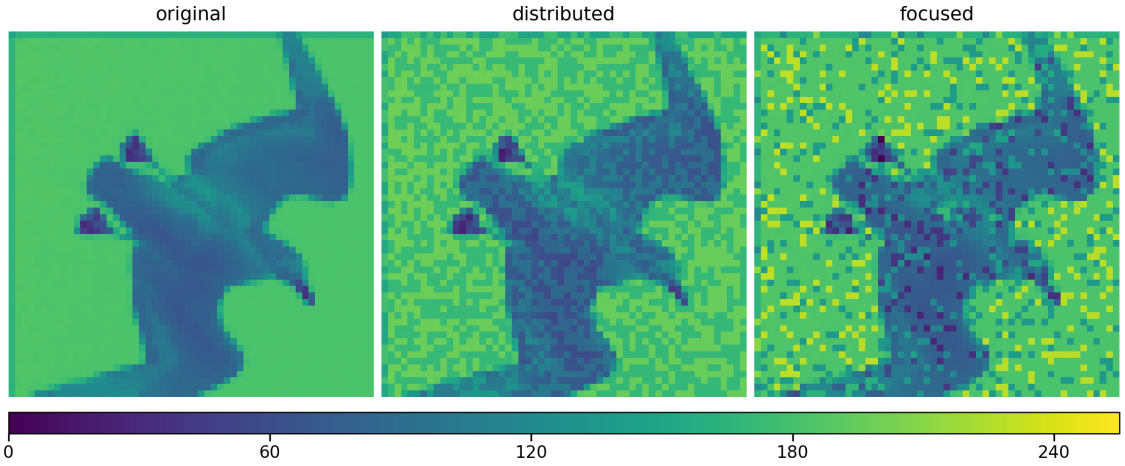


Figure A.4

Remarks We conducted the experiment on a single connection, `block1-0` of RegNetX-400MF on ImageNet. We realize, however, that this may not be representative for all skip connections and certainly not for all connections in a network in general. Particularly later layers and such that are very robust to approximation could show different behaviour. For more robust results, an examination of the effect across a wider spectrum of layers, models and datasets is necessary, which we did not conduct, since the findings here are not central to our problem.

Block size essentially determines, given the back-loading error mechanism, how often values can actually surpass the BET and consume budget that would on average be assigned to other values. This budget has to accumulate over preceding values first. There is two extremes to how the combined usage of VET and BET can distribute errors across the featuremap. In Scenario **(A)**, each value suffers the maximum BET. The error will be, by equal chance, positive or negative. In scenario **(B)** only one in n values suffers any error at all, but then the full error permissible within VET. n is determined by $round(VET/BET)$.

Thus, (A) represents a scenario where the error is evenly distributed across the activation, while (B) has all the error focus on a smaller number of elements. Both scenarios have the same average error.

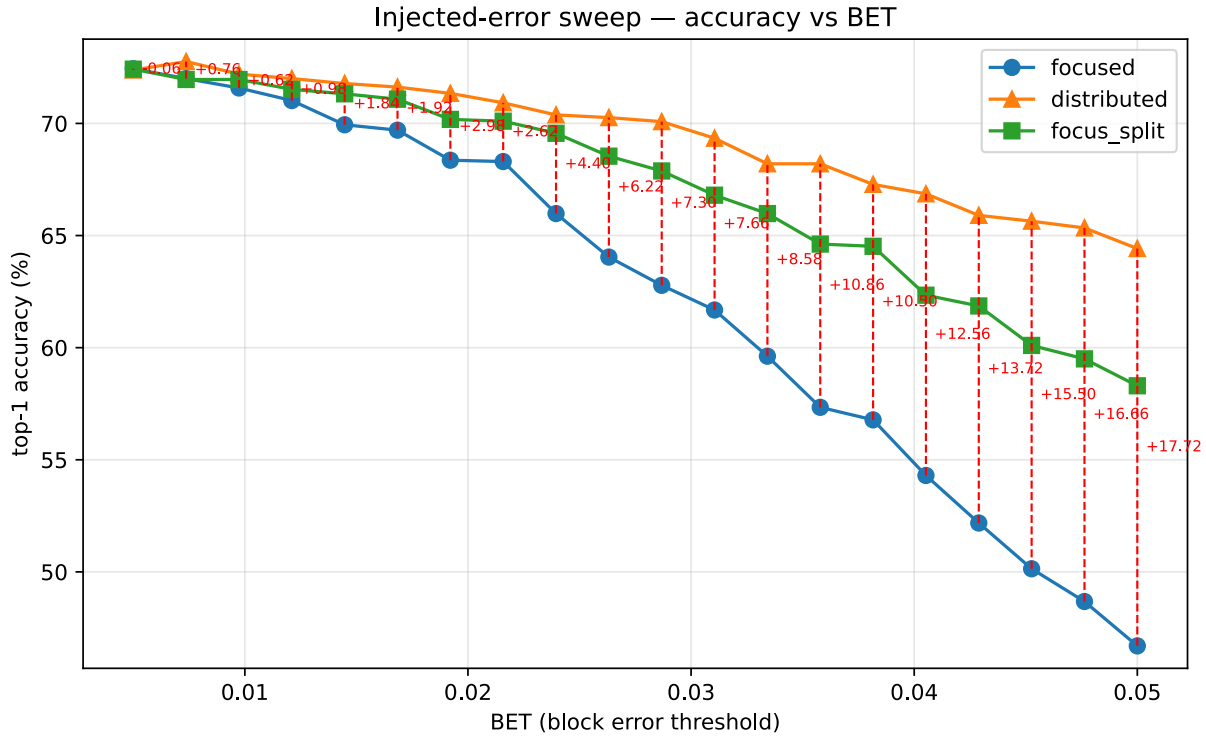


Figure A.5: The orange points represent scenario **(1)**, where errors are distributed evenly across the feature map; blue points, as in scenarioin **(2)** have the whole error focused in fewer pixels, with the rest completely error-free; green dots are an intermediate scenario, where the error is focused on twice as many points as in scenario (2).

A.3.3 Error Concentration and Block Size

The experiment above imposed evenly distributed and focused errors artificially. The block size of the block error threshold produces the same contrast inside the compressor itself: the larger a block, the more budget a value can draw from its predecessors, and the more often single predictions can exceed τ_b while the block average stays within it.

Setup We swept τ_v from 0 to 0.285 in 20 steps at a fixed $\tau_b = 0.026$ with linear blocks, for twelve block sizes between a single value and the whole featuremap ($H \cdot W = 3136$ values), on the connection of chapter 3 and its calibration split. For every point, Fig. A.6 plots the accuracy drop against the NMSE the compression actually introduced, normalized per sweep from 0 (lowest NMSE of the sweep) to 1 (highest).

Observations

- **The larger the block, the more accuracy the same growth in error costs.** The slope of the accuracy drop over the normalized NMSE increases steadily with the block size: at the end of the sweep, a single block per featuremap costs about 4.9 pp, blocks of 14 values about 1.2 pp and blocks of a single value about 0.6 pp.

- **Once the average error saturates, large blocks keep losing accuracy.** Every sweep levels off in ε as τ_v grows, which shows as a stack of points at a normalized NMSE of 1. With a single block per featuremap, ε stays at about 0.024 for the last 14 points of the sweep, while the drop grows from 3.0 to 4.9 pp. With blocks of 14 values, it stays between 0.7 and 1.2 pp. The growing τ_v admits ever larger single errors, which only a large block has the budget for.

- **Smaller blocks also reach a lower average error.** With back-loading, the budget of a block has to accumulate before it can be spent, and unspent budget is lost at every block boundary. Small blocks therefore stay well below τ_b: blocks of a single value, where every prediction is bounded by τ_b itself, saturate at $\varepsilon \approx 0.0066$, and their compression ratio at 1.72 compared to 3.0 for a single block per featuremap.

Together with the experiment above, this supports the reasoning of section 3.7.1: the block size controls how strongly the error concentrates on single values, and concentrated errors cost more accuracy than the same average error spread evenly. The average error alone is therefore not a sufficient proxy for the accuracy loss.

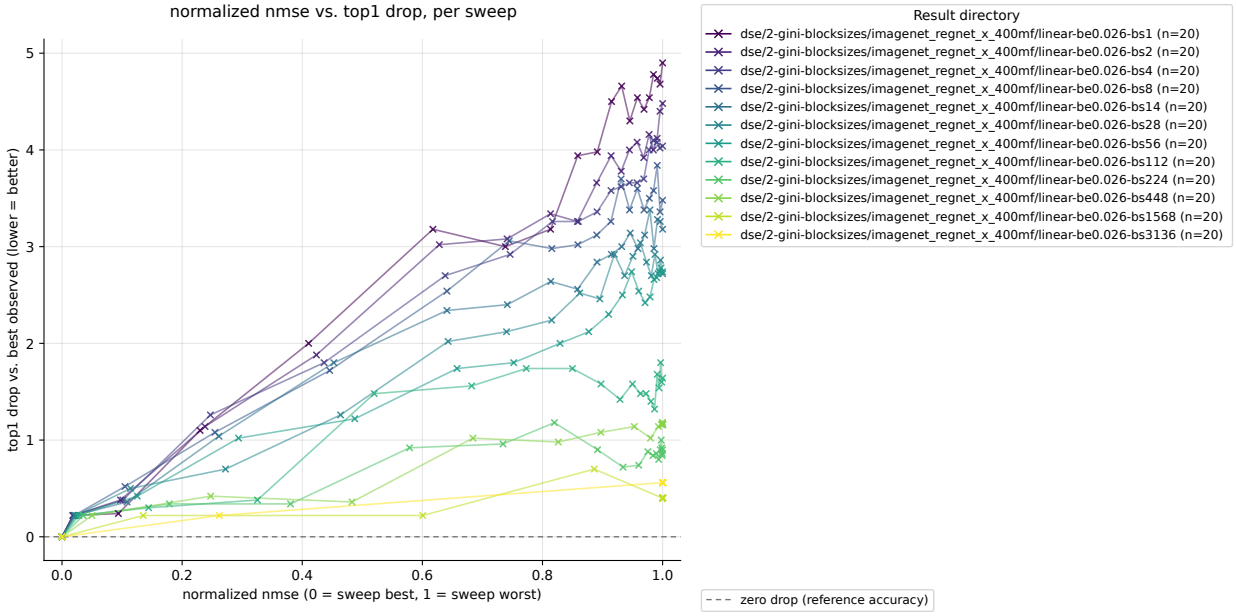


Figure A.6: Top-1 accuracy drop against the NMSE, normalized per sweep (0: lowest, 1: highest NMSE of the sweep), for a τ_v sweep (0 to 0.285, $\tau_b = 0.026$, linear blocks) per block size. The legend gives the featuremap subdivision N_{Subdiv} (bs$N$), i. e. a block size of $3136/N_{Subdiv}$ values, from the whole featuremap (bs1, dark) to a single value (bs3136, light). The slope grows with the block size, and once the NMSE saturates, large blocks keep losing accuracy (stacks of points at 1) because the growing τ_v concentrates the error on fewer values.

A.4 Outlier Preconditions

Section 3.2 argued that prediction functions interact: since every function predicts from reconstructed values, a run of constant predictions leaves a flat history, on which the linear and polynomial functions reduce to the constant prediction. To see how this shows up where compression actually fails, we recorded, for every outlier, the three reconstructed values preceding it and the candidate predictions of all active functions.

Setup An early version of our compressor with the set constant, polynomial and random on the output of the first residual block of ResNet-20 on CIFAR-10 (`layer1.0`), bounded by a value error threshold only. Figure A.7 shows ten outliers drawn at random from the 1.68 million of the run, each centred on its outlier value.

Observations In most of the rows, the three predecessors are flat: the history consists of identical values, the result of preceding constant predictions. On such a history, the polynomial prediction coincides with the constant one (orange and blue points at x lie on top of each other), so the set effectively offers one deterministic candidate instead of two, and both miss the outlier by the same amount. The polynomial differs from the constant prediction only in the rows with a curved history; in the most strongly curved one, it overshoots far beyond the acceptance band. With ten rows, the figure is an illustration rather than a statistic. The random predictor is the only function whose candidate does not depend on the history; it misses as well in these rows, but it is the only candidate that differs from the collapsed ones.

These outliers are thus not caused by a single bad prediction function, but by the sequence in which functions follow each other: constant predictions create exactly the precondition under which the polynomial function becomes redundant. This motivated replacing the polynomial or linear function by the random predictor (section 3.4), and it is the reason why the benefit of a prediction function can only be judged together with the functions it is combined with.

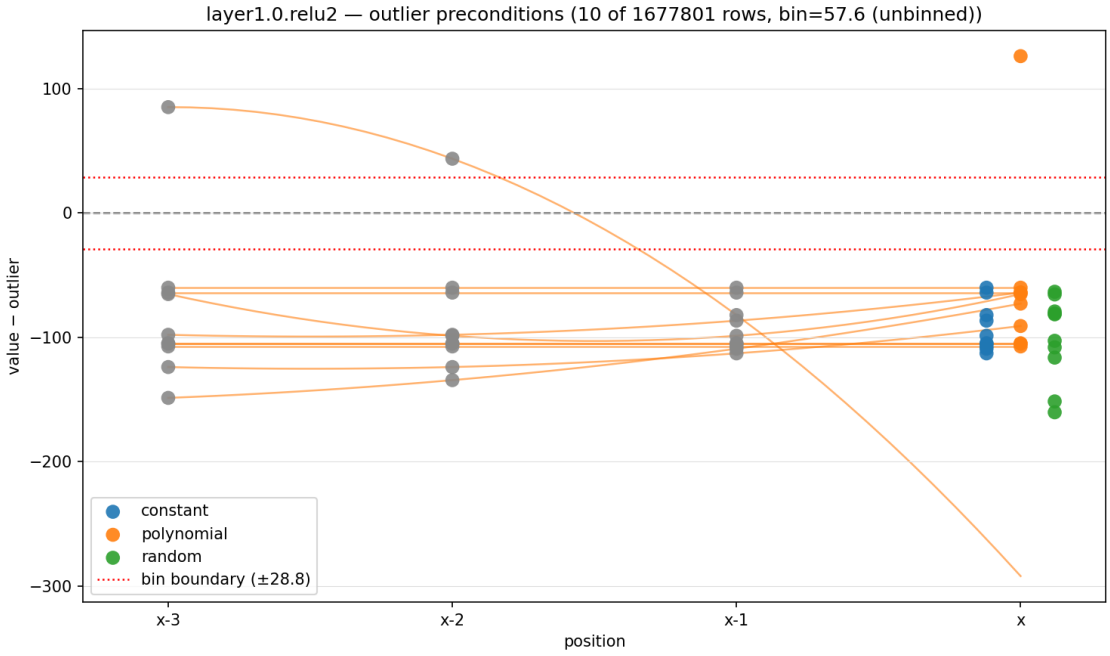


Figure A.7: Ten randomly drawn outliers of the set constant, polynomial, random on ResNet-20, CIFAR-10, `layer1.0`, value error threshold only. Each row is centred on its outlier (0 on the vertical axis): grey points are the three reconstructed predecessors at $x-3$ to $x-1$, the orange curves the polynomial extrapolation through them, and the coloured points at x the candidate predictions of the constant (**blue**), polynomial (**orange**) and random (**green**) function. Red dotted: the acceptance band of the value error threshold. Most histories are flat, so the polynomial prediction collapses onto the constant one and both miss together; where the history is curved, the polynomial overshoots.

A.5 Stochastic Encoding

As an extreme case of the predictor combinations of section 3.4, we replace all three functions of the SZ1 set by the random predictor (eq. (3.22)) and repeat the same τ_v sweep. This set falls far behind the SZ1 set (Fig. A.8): its accuracy drops steadily from the start of the sweep and reaches about 41 % top-1 at a ratio of only 2.6, while the SZ1 set still keeps about 70 % at that ratio. At $\tau_v = 0$, every value is an outlier and additionally costs its 2-bit code, so the ratio even drops below 1. The code distribution (Fig. A.9) shows why: the outlier share stays far above that of the SZ1 set at every τ_v, e. g. about 55 % against 8 % at $\tau_v = 0.105$, and the three random slots share the remaining codes about equally. A random draw does not model the local structure of the data at all; it only hits a value by chance, and this chance grows with the tolerance τ_v, at the price of large errors. The random predictor is therefore useful only as a complement to deterministic functions, at the positions where these have collapsed into each other, not as a replacement for them.

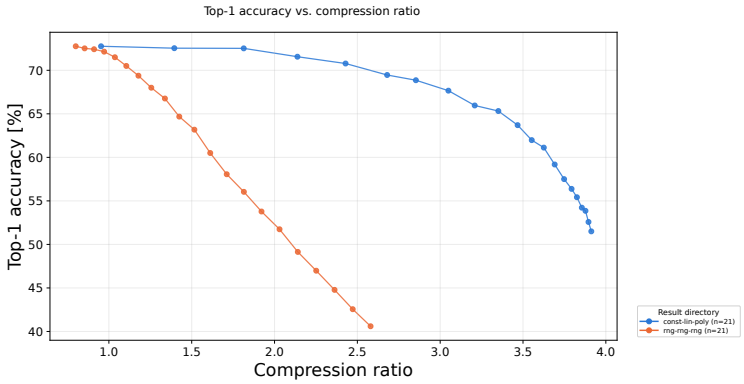


Figure A.8: Three random predictors (orange) vs. the SZ1 set (blue): top-1 accuracy against compression ratio over the same τ_v sweep. The random set loses accuracy from the start of the sweep and reaches only a ratio of 2.6 at about 41 % top-1; at $\tau_v = 0$, its ratio is below 1.

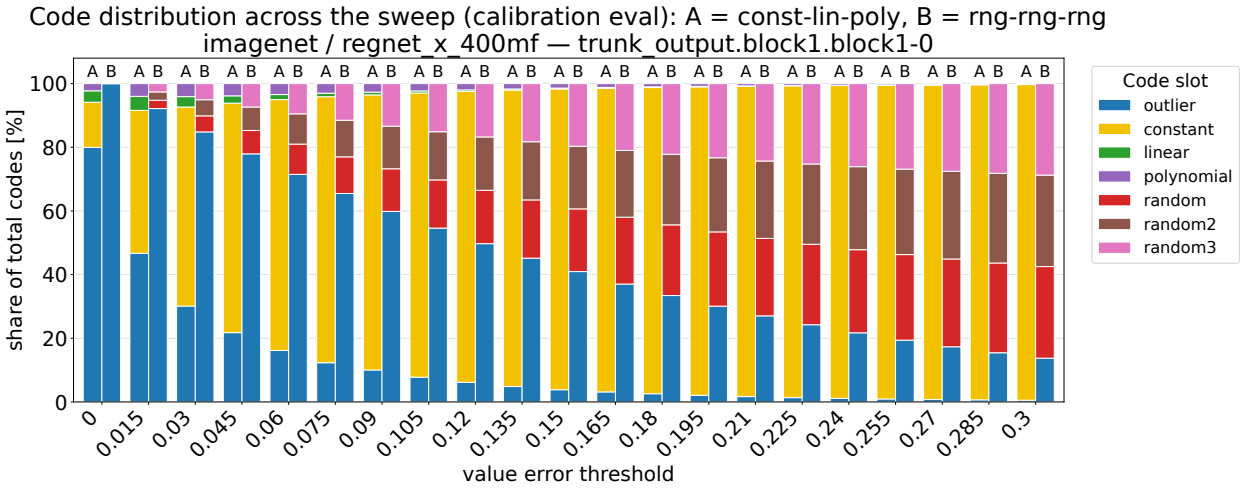


Figure A.9: Code distribution of the SZ1 set (A) and of three random predictors (B), as in Fig. 3.30. The outlier share of B stays far above that of A at every τ_v; the three random slots share the remaining codes about equally.

A.5.1 Fit of the Exponential Distribution

The random predictor was motivated by an early experiment on the output of the first residual block of ResNet-20 on CIFAR-10 (`layer1.0`, after the block's final ReLU), with an early version of our compressor (value error threshold only, row-major traversal).

Distribution of the activations Figure A.11 shows the distribution of the 8-bit codes of this connection, pooled over its 16 channels. It is dominated by small values: 27.6 % of all codes are zero, the output of the ReLU for negative inputs, the mean code is 23, and beyond a short plateau, the counts fall off almost linearly on the logarithmic axis, i.e. roughly exponentially, up to the largest codes. Values drawn from an exponential distribution show the same kind of decay (Fig. A.12), although they do not model the excess of exact zeros, and in this early run, their mean (14) lies below that of the data. Drawing predictions from it therefore places most draws where most of the values lie.

Effect on compression We replaced the polynomial function of the SZ1 set (constant, linear, polynomial) by a predictor drawing from this exponential distribution and repeated the τ_v sweep (Fig. A.10). Up to a ratio of about 1.5, both sets behave alike. Beyond it, the SZ1 set loses accuracy quickly and ends at 85.8 % at a ratio of 2.88, close to its limit, 5.5 pp below the baseline of 91.2 %. The set with the random predictor stays above 91 % up to a ratio of about 3.0 and still reaches 89.5 % at a ratio of 3.39. The random predictor takes a large share of the codes at the positions where the SZ1 set produces outliers, which is where both the ratio and the accuracy gain come from.

Code distribution Figures A.13 and A.14 show where the gain comes from. Whether the random predictor replaces the linear or the polynomial function, it takes 25 to 28 % of all codes from $\tau_v \approx 0.1$ on, and the outlier share falls to about 4 %, against about 10 % for the SZ1 set at the same τ_v. Remarkably, the remaining deterministic function is used far more often as well: the polynomial takes about 13 % of the codes and the linear function about 11 %, where the SZ1 set uses them for only about 4 and 1 %. The two random sets are practically indistinguishable, i.e. it does not matter which of the two extrapolating functions is replaced.

A connection with a different distribution The connection of chapter 3 does not share this distribution. Figure A.15 shows four of its channels together with the distribution the random predictor draws from. The channels are distributed in humps, some of them sharply peaked, with means between 80 and 210 of the 256 codes, and the smallest codes hardly occur at all. The exponential distribution, in contrast, is largest at code 0 and decays slowly over the whole range, so it frequently suggests values that rarely occur in the data; for channel 0, about 30 % of its draws even lie beyond the largest code. The cause is the structure of the block: on this connection, the operand of the residual addition is the output of the projection, a strided 1×1 convolution followed by batch normalization without a ReLU. Without the ReLU, there is no mass of zeros and small values, and batch normalization centres the channels instead.

Fitting the distribution to the connection A random predictor that fits every connection would therefore need a profiling step that determines the kind of distribution of each channel, e.g. from the per-channel histograms we already collect on the calibration split, and chooses the generator accordingly: an exponential distribution for post-ReLU channels, a distribution centred on the channel's mode for normalized ones, or simply a draw from the profiled histogram itself. We nevertheless consider the exponential distribution a particularly good fit where it applies. Its values cluster in one region, at the small

codes, so a large share of the draws lands close to a large share of the data. As τ_v grows, the window around each value widens, and the probability that a draw falls into it grows quickly, because the draws are concentrated where the values are. A broad, hump-shaped distribution spreads its draws, and its mass, over a wider range, so the same tolerance yields fewer hits.

Why the random predictor helps The improvement on ResNet-20 goes beyond the hits of the random predictor itself, which is visible in the higher usage of the remaining deterministic function. It follows from the collapse of the extrapolating functions described in section 3.2 and observed at the outliers (section A.4). Every constant prediction repeats the previous value, and after one or two of them, the linear and polynomial functions read a flat history and predict the same value as the constant function. A run of constant predictions thus reduces the set to effectively a single candidate, and when that candidate misses, the value becomes an outlier. The random predictor does not read the history and is therefore never collapsed. Whenever one of its draws lies within the tolerance and closer to the value than the constant prediction, it is chosen, and the reconstructed history is no longer flat. The linear and polynomial functions then extrapolate from distinct values again and can contribute their own predictions. On post-ReLU activations, where most values are small and the draws of the exponential distribution land there as well, this happens often enough to break the runs of constant predictions regularly, which lowers the outlier share and makes the remaining deterministic function useful again.

One could conceive a random predictor that is conditioned on the local history in addition to the profiled distribution of its channel, e.g. by drawing from a distribution centred on the previous reconstructed value. Its draws would lie closer to the data on average, but they would also follow the history: after a flat run, they would cluster around the very value the collapsed functions already predict. This would undo the property that makes the random predictor useful in the first place, namely that its candidates are independent of the local history and can therefore break out of a flat one.

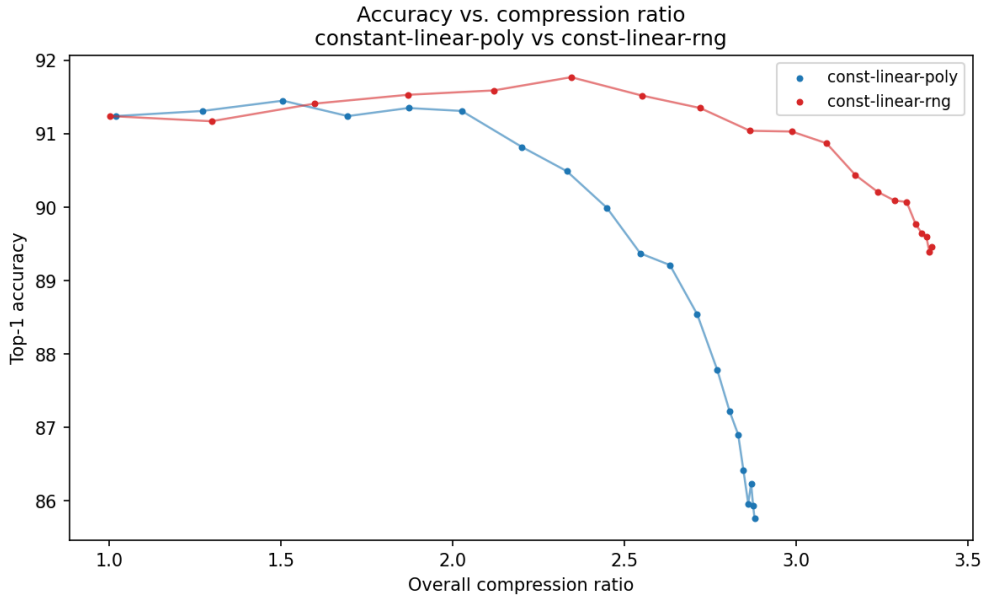


Figure A.10: SZ1 set constant, linear, polynomial (**blue**) vs. constant, linear, random (**red**), ResNet-20, CIFAR-10, `layer1.0`, early compressor version with value error threshold only; top-1 accuracy against compression ratio over a τ_v sweep from 0 to 0.2375. Beyond a ratio of 1.5, the SZ1 set loses accuracy quickly and ends at 85.8 % at 2.88; with the random predictor, the accuracy stays above 91 % up to a ratio of about 3.0 and reaches 89.5 % at 3.39.

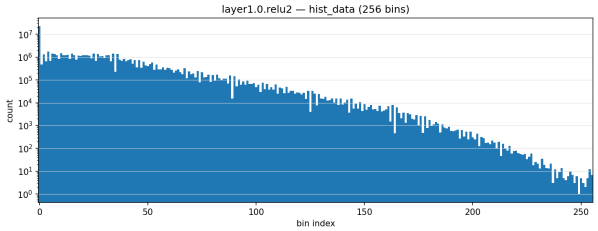


Figure A.11: Observed distribution of the 8-bit activation codes of ResNet-20, CIFAR-10, `layer1.0` after its final ReLU, pooled over all channels (logarithmic counts). 27.6 % of the codes are zero; beyond a plateau, the counts decay roughly exponentially.

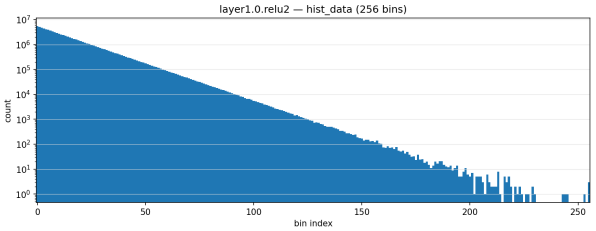


Figure A.12: Distribution of the same number of values drawn from an exponential distribution by an early version of the random predictor (logarithmic counts). It reproduces the decay of Fig. A.11, but not its excess of exact zeros.

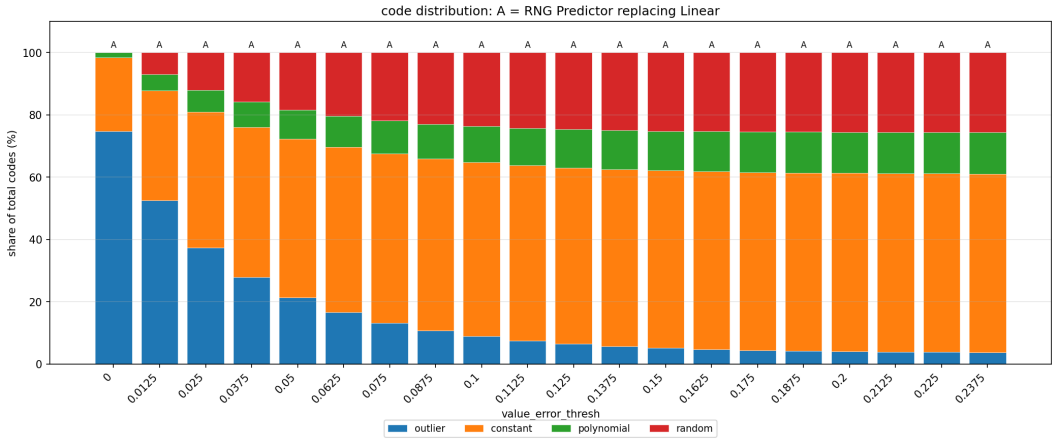


Figure A.13: Code distribution of the set constant, polynomial, random, i.e. the random predictor replacing the linear function, over the τ_v sweep, ResNet-20, CIFAR-10, `layer1.0`, pooled over all channels. From $\tau_v \approx 0.1$ on, the random predictor takes about a quarter of the codes, the polynomial about 13 %, and the outlier share falls to about 4 %.

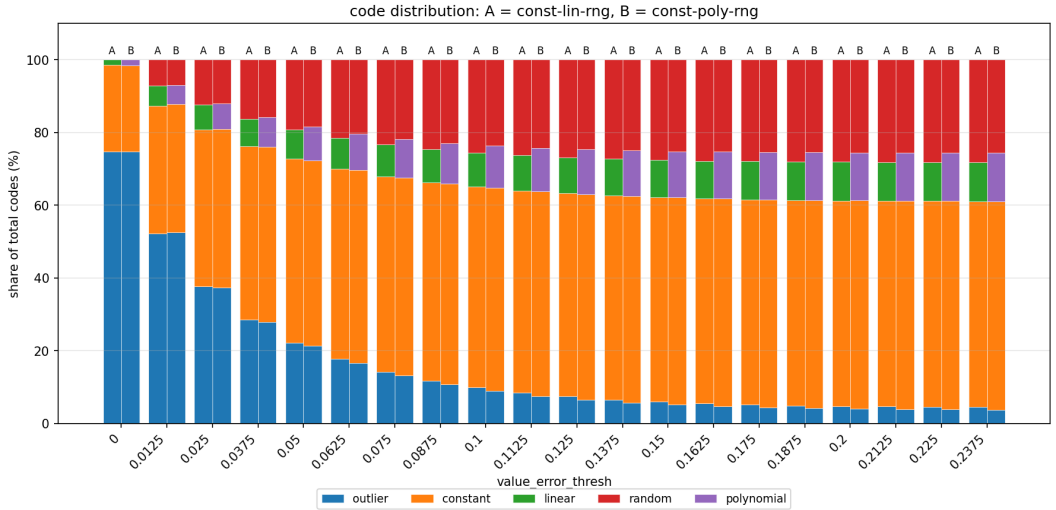


Figure A.14: Code distribution of the random predictor replacing the polynomial (A: constant, linear, random) and the linear function (B: constant, polynomial, random), over the same sweep as Fig. A.13. Both sets are practically indistinguishable: the random predictor takes 25 to 28 % of the codes, the remaining deterministic function 11 to 13 %, and the outliers about 4 %.

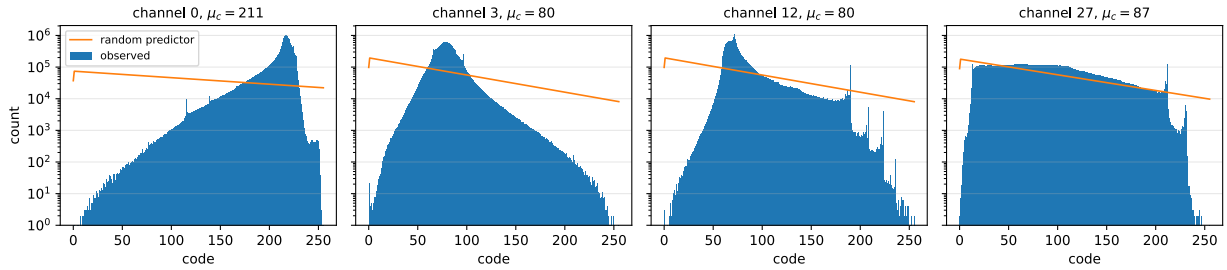


Figure A.15: Observed distribution of the 8-bit codes (**blue**) and the distribution the random predictor draws from, $\mu_c \cdot \mathrm{Exp}(1)$ (**orange**), for four channels of RegNetX-400MF, `block1-0`, calibration split (logarithmic counts). The channels are hump-shaped, centred far from code 0, while the exponential distribution is largest at code 0 and decays slowly: it frequently suggests values that rarely occur, and for channel 0, about 30 % of its draws lie beyond the largest code. Only channel 27, which is spread almost evenly over a wide range, is matched roughly.

A.6 Trajectory Search vs. Grid Sweep

The trajectory search of section 3.7.2 evaluates only a few dozen configurations, chosen by fitting the course of accuracy drop and compression ratio. To check whether it misses better configurations, we compared an early version of the search with a grid sweep over the same parameters on the connection of chapter 3.

Setup The grid sweep covers 5 block error thresholds ($\tau_b \in \{0, 0.005, 0.01, 0.02, 0.025\}$), 20 value error thresholds (τ_v from 0 to 0.285) and 6 featuremap subdivisions ($N_{Subdiv} \in \{1, 8, 56, 112, 448, 3136\}$), 600 configurations in total. The search ran 38 evaluations with $\tau_v = 3.5\,\tau_b$, τ_b between 0.016 and 0.079, and 21 different subdivisions. It is a predecessor of the procedure in section 3.7.2, without the final refinement of τ_v. Both ran on the calibration split; the grid sweep used the SZ1 predictor set.

Results Figure A.16 shows both point sets and their common efficient frontier. The search concentrates its evaluations around the budget it was steering towards: 80 % of them lie between 69.3 and 70.4 % top-1. There, its points lie on or beyond the front of the grid sweep: within a drop to 70.3 %, the search reaches a compression ratio of 2.93, the grid 2.70; within 69.5 %, 3.08 against 2.87. The four rightmost vertices of the common frontier are all points of the search. Above about 70.5 %, the grid reaches the higher ratios, since the search never evaluated configurations in that range.

The search therefore does not need the density of a grid to find good configurations: with about 6 % of the evaluations, it finds better ones near its target. The configurations it finds are, however, only good for the accuracy range it explores, which is why the full procedure runs one search per budget (Fig. 3.51). The gain over the grid also stems partly from the τ_b values above 0.025 and the finer subdivisions that the grid does not contain; a denser grid would close part of the gap at a correspondingly higher number of evaluations.

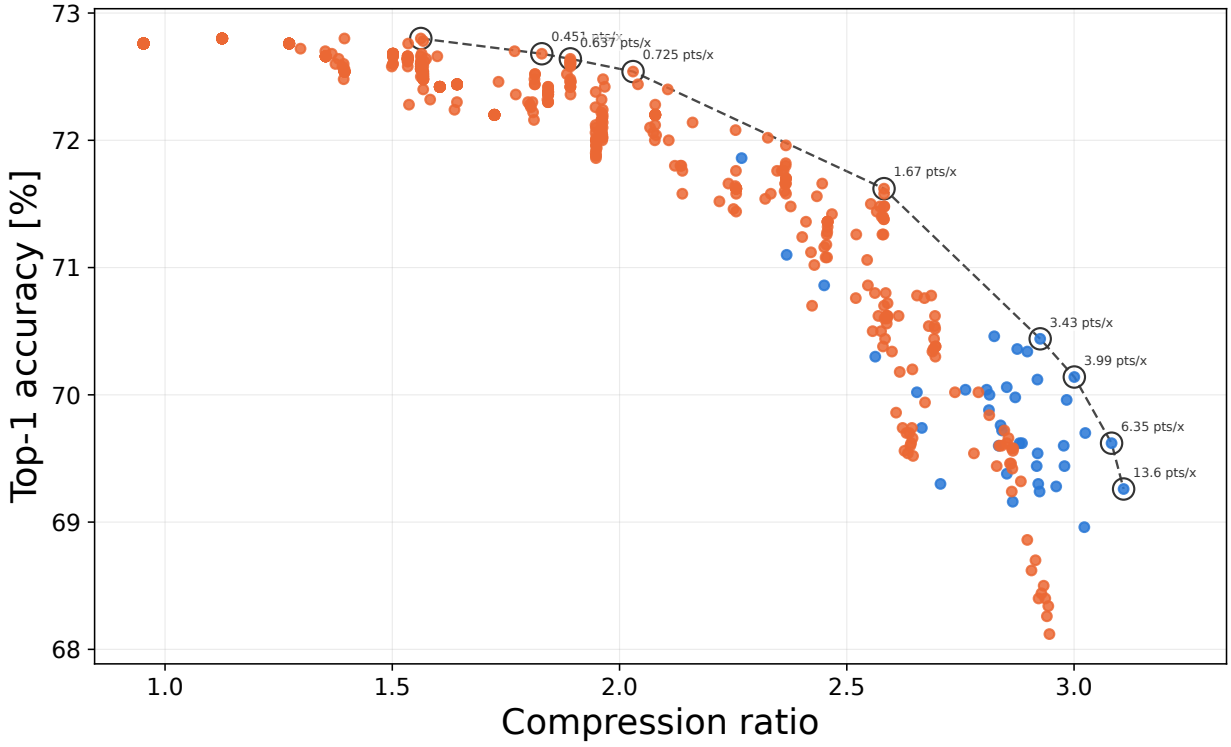


Figure A.16: Configurations evaluated by an early version of the trajectory search (38 points, **blue**) and by a grid sweep over τ_b, τ_v and N_{Subdiv} (600 points, **orange**), RegNetX-400MF, `block1-0`, calibration split; top-1 accuracy against compression ratio. Dashed: common efficient frontier, labelled with the accuracy lost per unit of compression ratio. In the accuracy range it targets, the search reaches up to about 0.2 higher compression ratio than the grid with about 6 % of its evaluations; it does not explore the high-accuracy part of the front.

A.7 Huffman gain after different lossy stages

Here we evaluate the effectiveness of Huffman encoding when used on the output streams of different lossy stages, namely our SZ implementation and the simple quantization stage.

Figures A.17 and A.18 show the compression ratio of the SZ module only vs. the SZ module with a Huffman encoder used on the code stream. We can see clear improvements in the early layers of the model. For most of the layers improvements are real but only marginal.

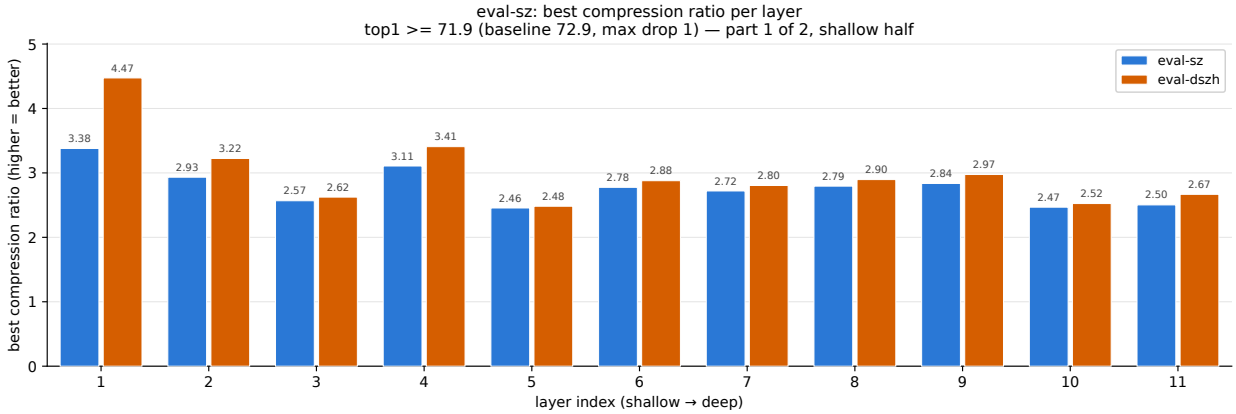


Figure A.17: Best compression ratio per layer within a top-1 drop of 1 pp, layers 1 to 11 (shallow half): SZ (**blue**) vs. SZ + Huffman (**orange**).

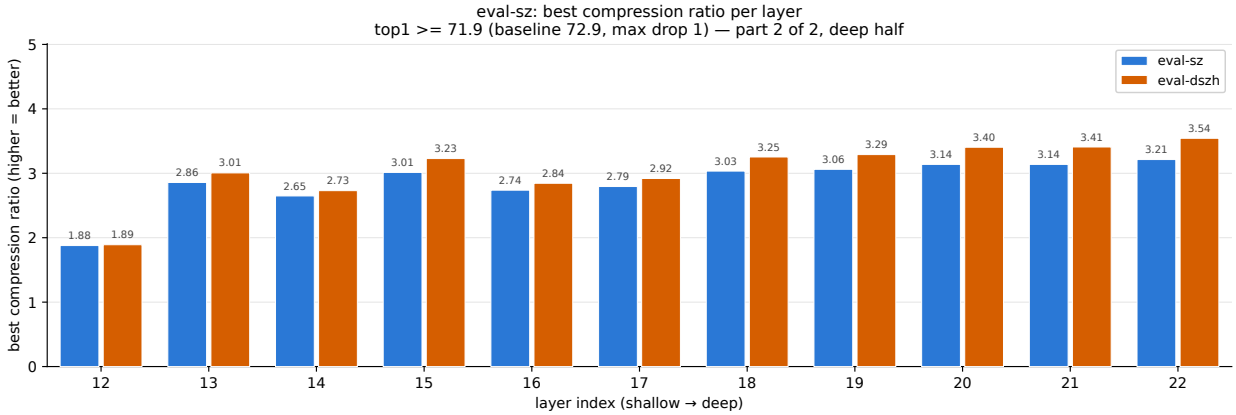


Figure A.18: Best compression ratio per layer within a top-1 drop of 1 pp, layers 12 to 22 (deep half): SZ (**blue**) vs. SZ + Huffman (**orange**).

Figures A.19 and A.20 show the same arrangement just with the quantization lossy stage. The image is inverted however — the early layer Huffman encoder yields only little improvement while there is massive improvement in the late layers.

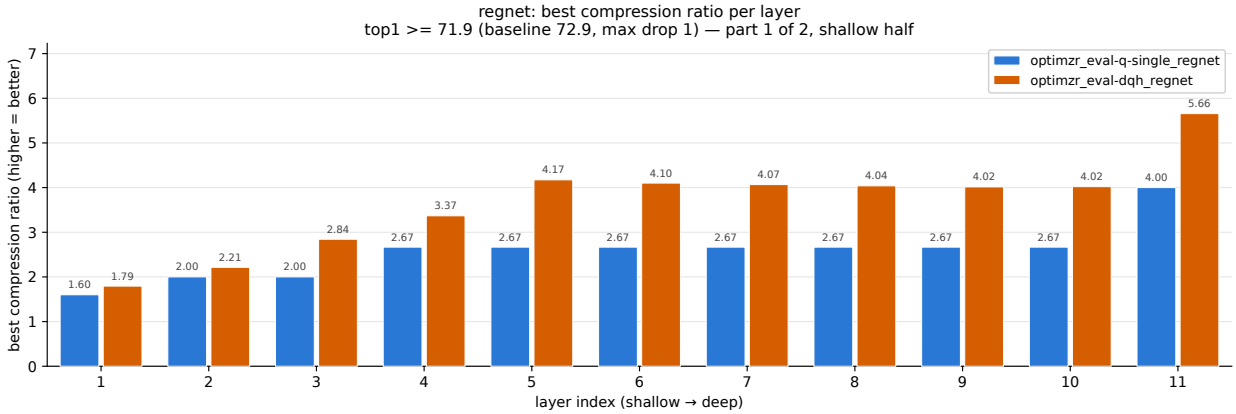


Figure A.19: Best compression ratio per layer within a top-1 drop of 1 pp, layers 1 to 11 (shallow half): quantization (**blue**) vs. quantization + Huffman (**orange**).

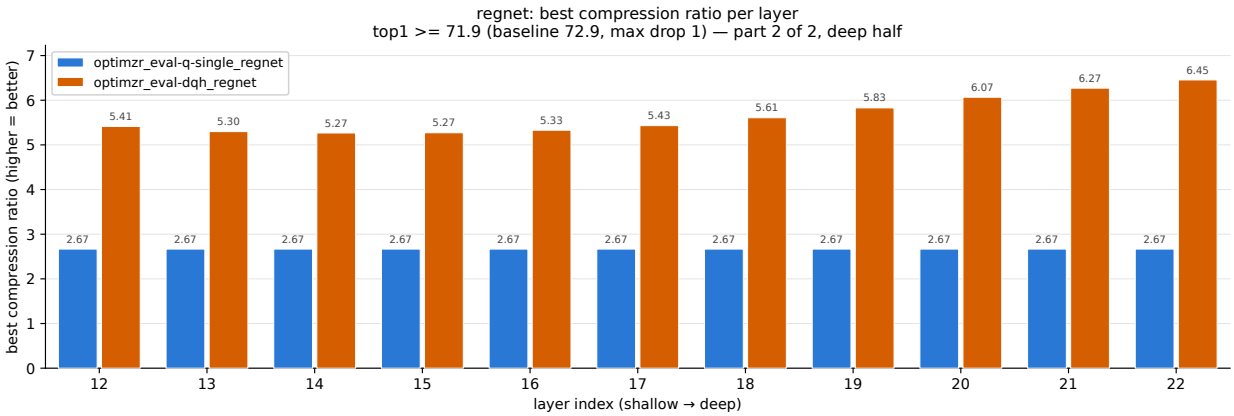


Figure A.20: Best compression ratio per layer within a top-1 drop of 1 pp, layers 12 to 22 (deep half): quantization (**blue**) vs. quantization + Huffman (**orange**).

In Figs. A.21 and A.22, we directly compare the quantization and SZ lossy stages, without any postprocessing. SZ wins 4 out of 22 layers, most by a small margin but the first 4 by a clear margin.

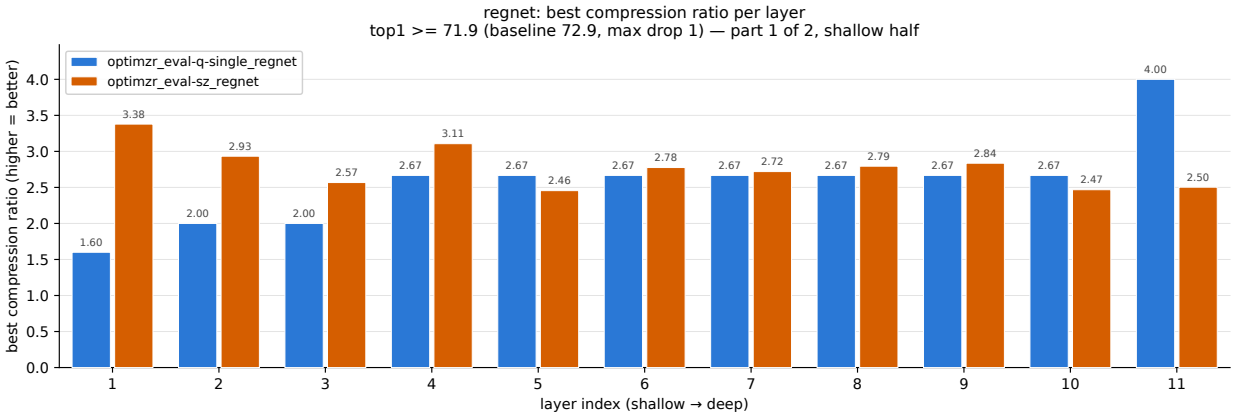


Figure A.21: Best compression ratio per layer within a top-1 drop of 1 pp, layers 1 to 11 (shallow half): quantization (**blue**) vs. SZ (**orange**).

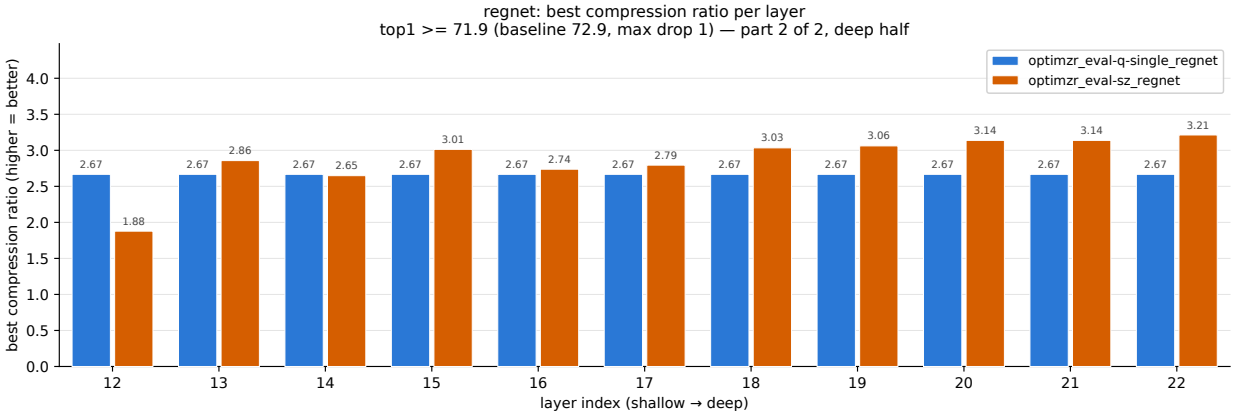


Figure A.22: Best compression ratio per layer within a top-1 drop of 1 pp, layers 12 to 22 (deep half): quantization (**blue**) vs. SZ (**orange**).

Figures A.23 and A.24 show the direct comparison of SZ and quantization with a huffman postprocessing stage. SZ wins the early layers but looses by a lot in any other layer.

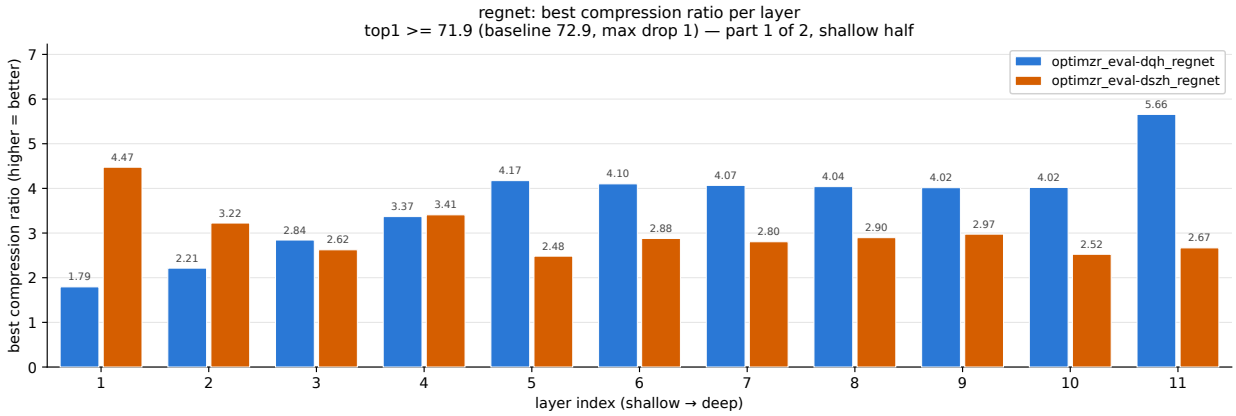


Figure A.23: Best compression ratio per layer within a top-1 drop of 1 pp, layers 1 to 11 (shallow half): quantization + Huffman (**blue**) vs. SZ + Huffman (**orange**).

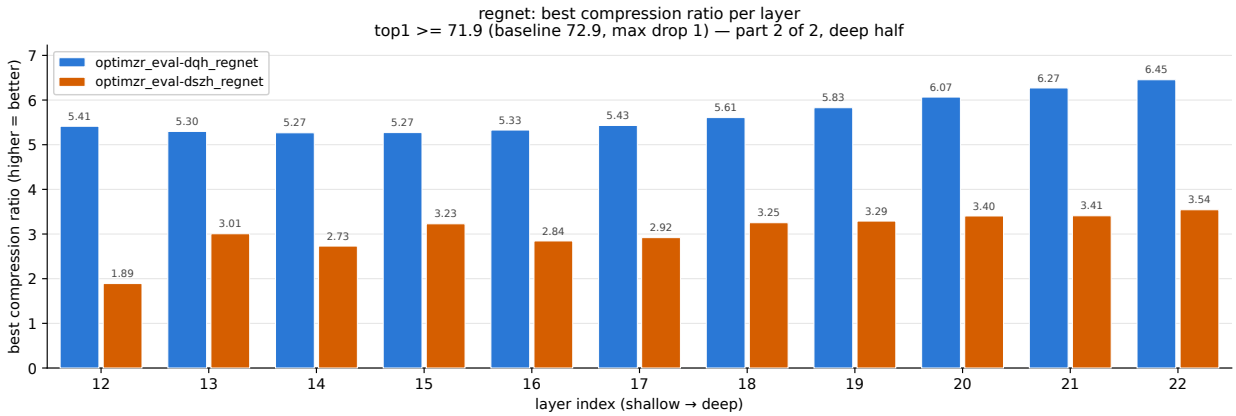


Figure A.24: Best compression ratio per layer within a top-1 drop of 1 pp, layers 12 to 22 (deep half): quantization + Huffman (**blue**) vs. SZ + Huffman (**orange**).

These bar plots show that the Huffman gain is clearly higher on the skewed output of the quantization compressor, which retains the distribution of the original data stream. The SZ prediction model completely decorrelates the input stream from the output stream. Now the Huffman gain depends on the skew in the code distribution but with codes being equally strong, the only clearly longer code word will usually be the outlier code. SZ1 [13] showed that the code stream can be compressed quite well using a dictionary based compression scheme, but these require big buffers and parallelization logic to efficiently find matches in the streams history. Ko et al. [44] found them unfeasible for implementation in hardware where other designs have to live as well (i.d. a whole neural network) and who are we to disagree.

A.8 Prediction Function Selection by Context Size

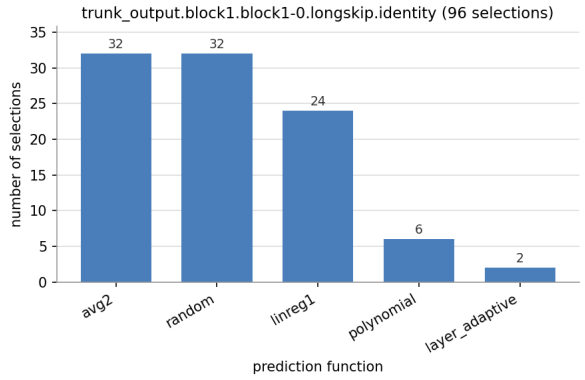


Figure A.25: Prediction functions in the static per-channel selection of RegNetX-400MF, `block1-0`, with a maximum context of $K = 0$ buffered rows: number of channels whose selection contains each function (32 channels with three functions each, 96 selections).

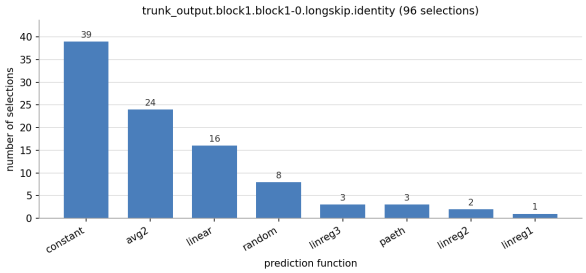


Figure A.26: As Fig. A.25, with a maximum context of $K = 2$ buffered rows.

A.9 Layer Sensitivity

A.9.1 Impact of cutting connections

To gauge the networks sensitivity to approximation in the extreme case, we conducted the following experiment: We take the set of skip-connection per network and run a inference pass over the ImageNet test dataset for each skip-connection, setting the output of that skip connection to zeroes-only. Since all these skip connections are residual additions, this effectively simulates a "cut" connection: No information is transported across those connections anymore.

The results of this experiment for RegNet can be seen in Fig. A.27. Apart from the skip-connections at depth 4 and 11 which can be cut with decreased penalty to accuracy, the general trend that shows up for all the layers we examined is that the deepest skip connection start to gradually decrease in impact to the accuracy when cut.

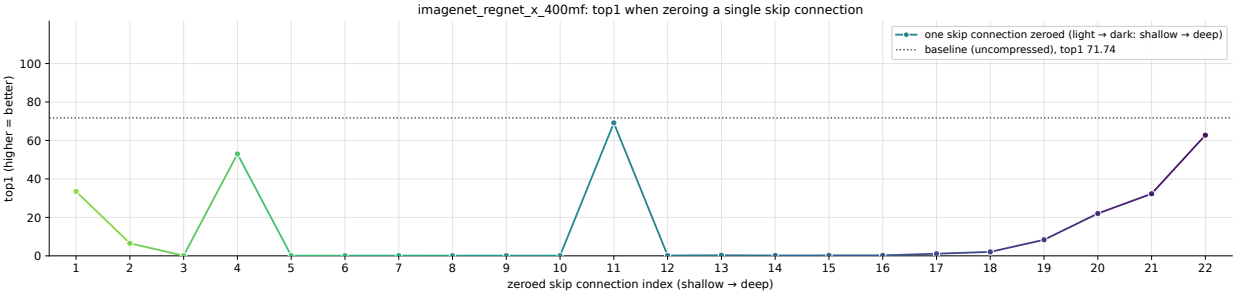


Figure A.27: Horizontal axis: depth-wise index of the skip connection that was zeroed out; vertical axis: accuracy achieved by the model with the skip connection zeroed out; the horizontal dashed line is the accuracy of the model achieved without any approximation; points at the very bottom mean that the models inference accuracy is reduced to mere chance; surprisingly, some connections can be cut without completely destroying the models inferential abilities

All evaluated networks Figures A.28 to A.34 repeat the experiment for the other networks of chapter 4, over all skip connections of each network, not only the evaluated ones. Three observations carry over from RegNetX-400MF:

- **Most connections are indispensable.** Cutting a single connection drops most networks to chance level. The CIFAR-10 networks are the most robust to it, with ResNet-20 keeping 81 to 92 % on four of its nine connections, but even there, most connections cost more than 35 pp when cut.
- **Late connections are the least sensitive.** The last connections of every network are among those that can be cut at the smallest cost, e.g. ResNet-18 keeps 69 % of its ~70 % baseline. MobileNetV2 is the exception on ImageNet: no connection can be cut without losing at least 50 pp.
- **The first connection of a stage is an outlier.** Besides the late connections, the first connection of a later stage tolerates the cut best: `block3-0` and `block4-0` of RegNetX-400MF keep 53 and 69 %, `layer3.0` and `layer4.0` of ResNet-18 67 and 60 %, and `layer2.0` and `layer3.0` of ResNet-20 on CIFAR-10 81 and 91 %. In these blocks, the skip connection carries the output of a projection, which changes resolution and channel count, while the residual branch still carries the information of the block. Cutting the connection of an identity block, in contrast, removes the only path of the previous activations past the block.

The sensitivity of a connection thus depends on both its depth and its type, which is why the connections evaluated in chapter 4 were chosen by a sensitivity sweep (section A.9.3) rather than by depth alone. Note that cutting is an extreme case: the next experiment shows that the ranking of connections under strong but incomplete approximation can differ from the ranking under a cut.

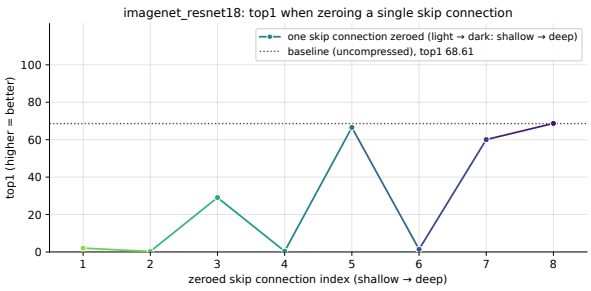


Figure A.28: Top-1 accuracy with a single skip connection cut, ResNet-18, ImageNet, connections in order of depth; dotted: baseline.

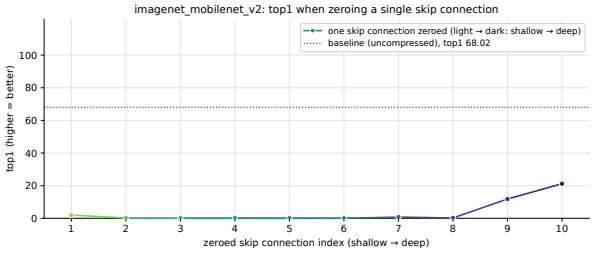


Figure A.29: As Fig. A.28, MobileNetV2, ImageNet.

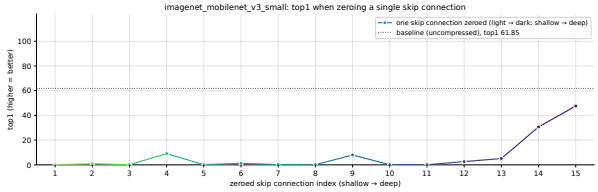


Figure A.30: As Fig. A.28, MobileNetV3-Small, ImageNet.

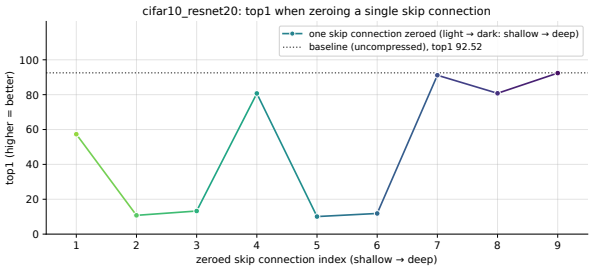


Figure A.31: As Fig. A.28, ResNet-20, CIFAR-10.

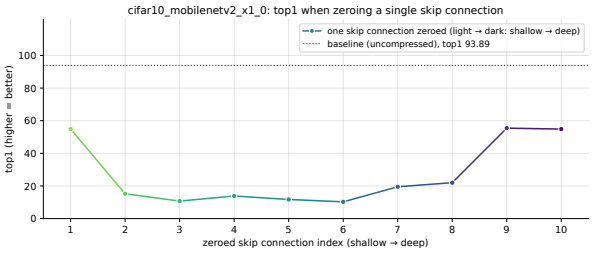


Figure A.32: As Fig. A.28, MobileNetV2, CIFAR-10.

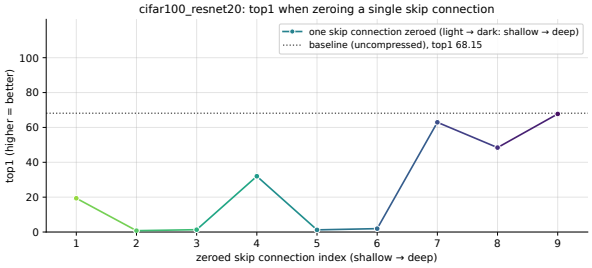


Figure A.33: As Fig. A.28, ResNet-20, CIFAR-100.

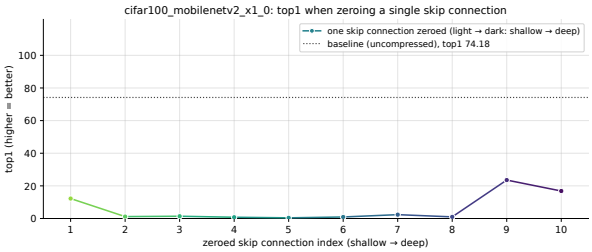


Figure A.34: As Fig. A.28, MobileNetV2, CIFAR-100.

A.9.2 Impact of quantizing connections

In this experiment we try gauge the error tolerance with increasing approximation to its data. We used quantization with decreasing number of quantization bins to achieve a steady increase in reconstruction error: For each skip connection individually, we subjected the activation tensors to linear quantization with 2^N bins, where N ranged from 8 to 0. At $N = 8$ bit, there is no approximation happening, as we are only treating 8-bit quantized models. At $N = 0$ bit, the effects are the same as in experiment A.9.1.

Fig. A.35 shows the impact of quantization on RegNet. Here it generally holds that the model can withstand approximations in deeper layers much better than approximations in earlier layers, which was already hinted at by experiment A.9.1. Surprising is, however, that some connections will actually impact the model less when completely cut rather than approximated strongly. E.g. connection 4 of the previous experiment will push the model accuracy below 10 percent when assigned only 1 bit for quantization, but only pushes the model down to just above 50 percent when completely cut.

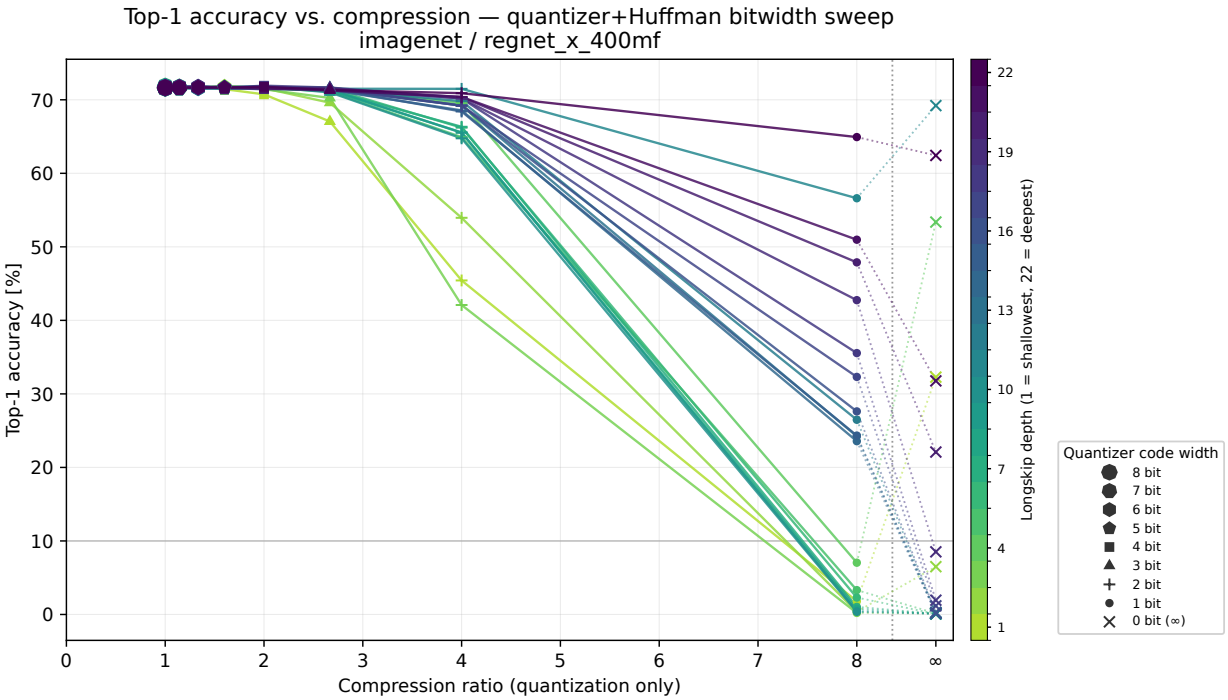

Top-1 accuracy vs. compression — quantizer+Huffman bitwidth sweep
imagenet / regnet_x_400mf
Top-1 accuracy [%]
Compression ratio (quantization only)
Longskip depth (1 = shallowest, 22 = deepest)
Quantizer code width
8 bit
7 bit
6 bit
5 bit
4 bit
3 bit
2 bit
1 bit
0 bit (∞)

Figure A.35

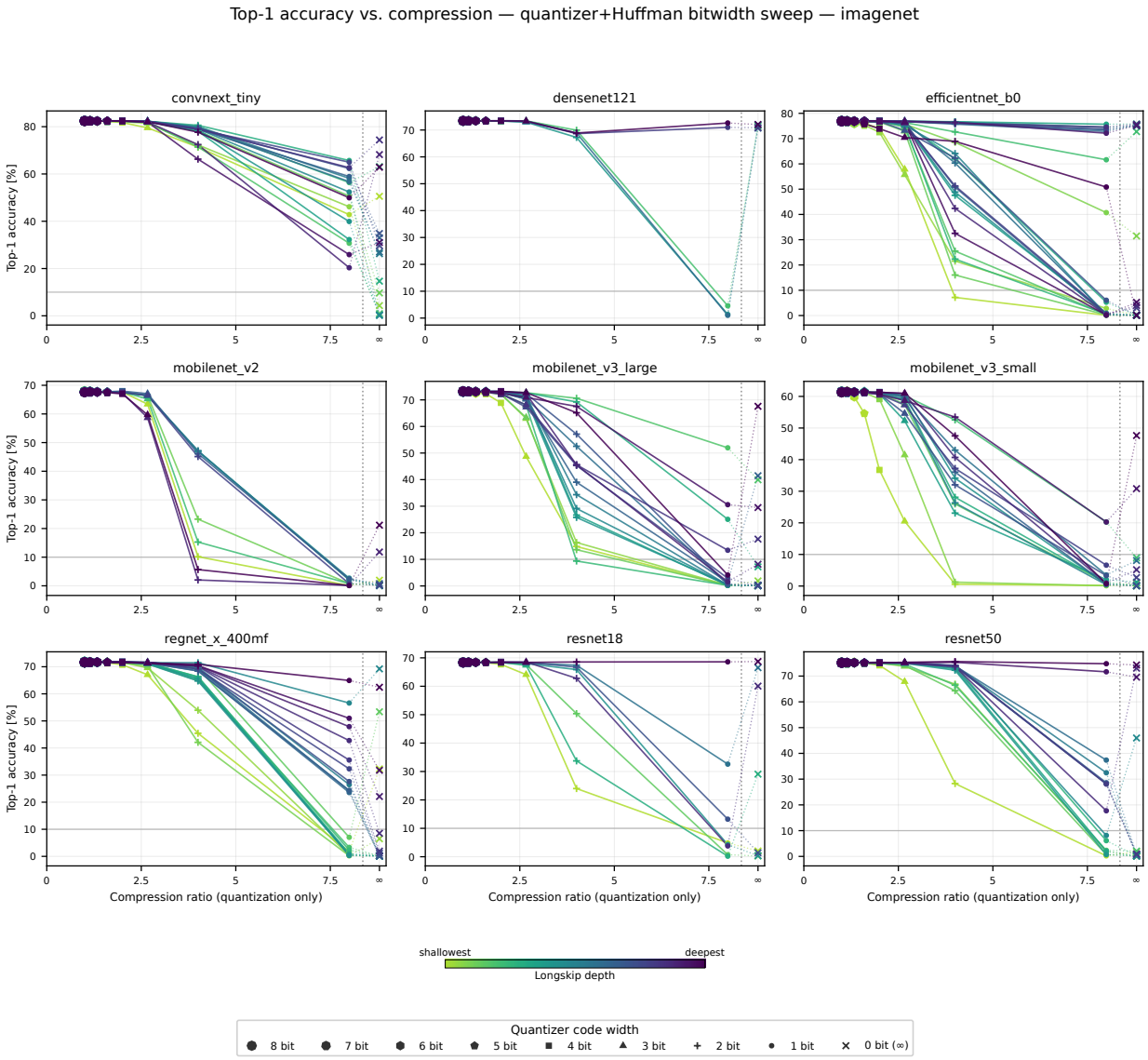

Top-1 accuracy vs. compression — quantizer+Huffman bitwidth sweep — imagenet
convnext_tiny
densenet121
efficientnet_b0
mobilenet_v2
mobilenet_v3_large
mobilenet_v3_small
regnet_x_400mf
resnet18
resnet50
Top-1 accuracy [%]
Compression ratio (quantization only)
shallowest
deepest
Longskip depth
Quantizer code width
8 bit
7 bit
6 bit
5 bit
4 bit
3 bit
2 bit
1 bit
0 bit (∞)

Figure A.36

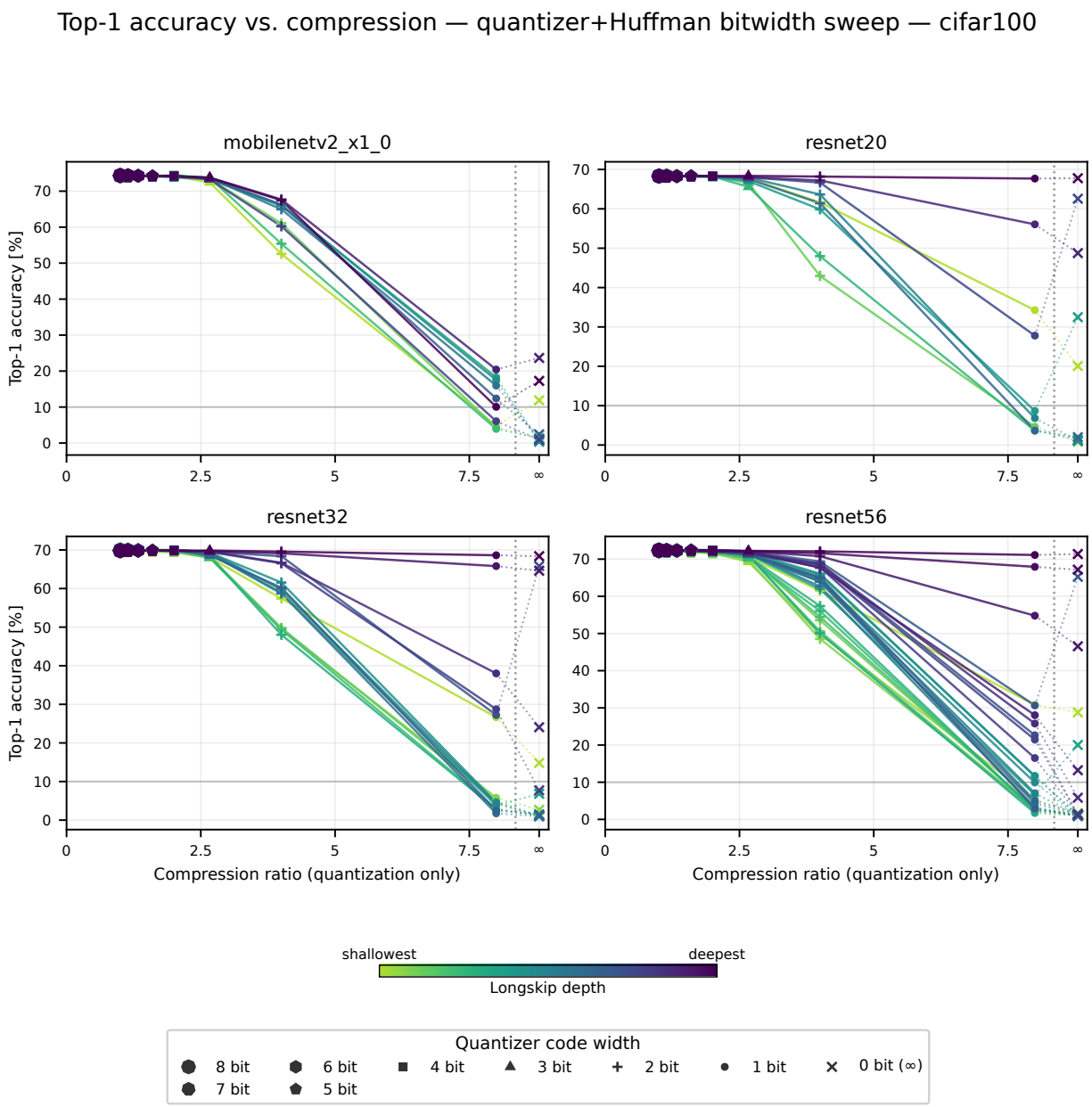

Top-1 accuracy vs. compression — quantizer+Huffman bitwidth sweep — cifar100
mobilenetv2_x1_0
resnet20
resnet32
resnet56
Top-1 accuracy [%]
Compression ratio (quantization only)
shallowest
deepest
Longskip depth
Quantizer code width
8 bit
7 bit
6 bit
5 bit
4 bit
3 bit
2 bit
1 bit
0 bit (∞)

Figure A.37

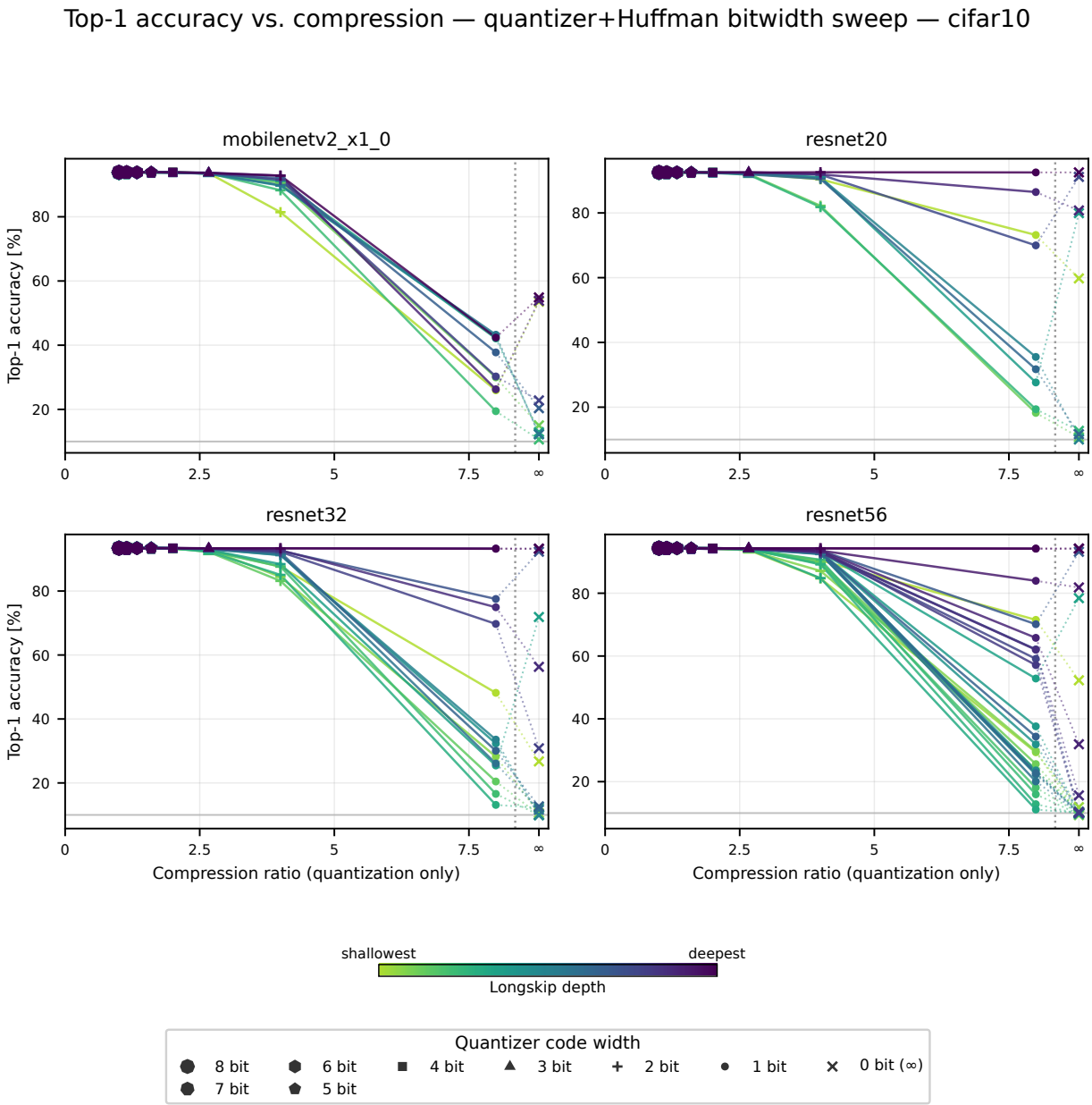


Figure A.38

A.9.3 Sensitivity of the evaluated connections

Figures A.39 to A.46 repeat the quantization sweep of section A.9.2 for every network of chapter 4, restricted to the skip connections that are evaluated there. Each connection is requantized on its own to $N = 8, 7, \ldots, 0$ bits and the resulting codes are Huffman coded; every line is one connection, every marker one bitwidth, and the horizontal axis gives the compression ratio of quantization and Huffman coding together. The dashed line is the accuracy of the sweep at full word length, averaged over the connections, the dotted red line the accuracy budget of the dataset used for the sweeps (3 pp for CIFAR-10, 4 pp for CIFAR-100, 5 pp for ImageNet).

These sweeps serve two purposes. First, they are the sensitivity analysis from which the evaluated connections were chosen. For every network, we ranked the

connections by the accuracy drop at 3 bits (at 2 bits where 3 bits cost practically nothing) and selected up to nine of them: the most sensitive ones, the earliest and the latest connection, and representatives in between. ResNet-18 has only eight skip connections, all of which are used. RegNetX-400MF, the model of chapter 3, is shown with the twelve of its 22 connections that chapter 4 compares.

Second, the same kind of sweep is the starting point of the parameter search (sections 3.7.1 and 3.7.2): for a given connection and budget Δ, the first bitwidth whose accuracy leaves the budget determines, through its average error, the initial block error threshold $\tau_b^{(0)}$. The figures therefore also show where this happens for each connection, i. e. how much quantization error a connection tolerates before it leaves a given budget, and thus how far apart the initial τ_b of different connections lie.

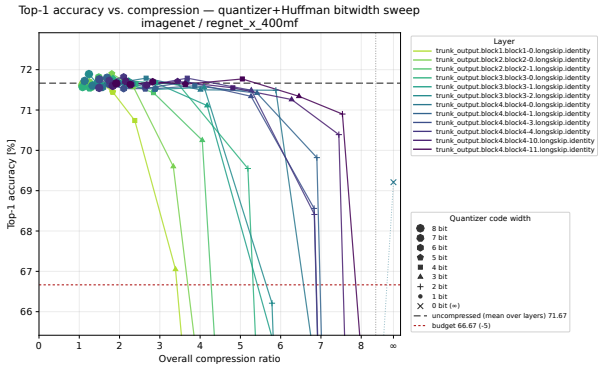


Figure A.39: Quantizer and Huffman bitwidth sweep, RegNetX-400MF, ImageNet, 12 connections.

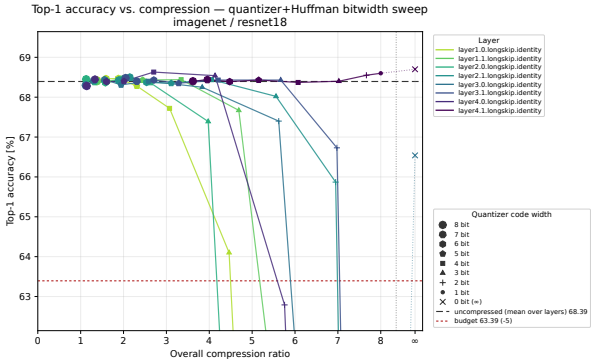


Figure A.40: Quantizer and Huffman bitwidth sweep, ResNet-18, ImageNet, 8 connections.

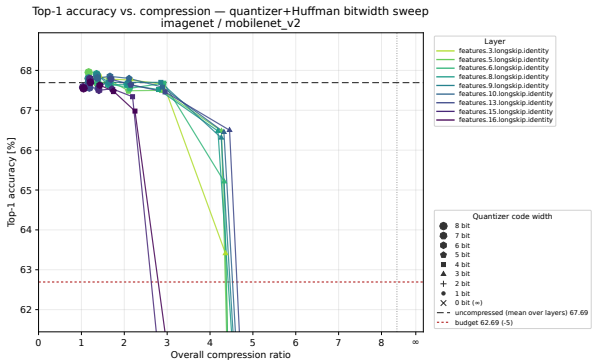


Figure A.41: Quantizer and Huffman bitwidth sweep, MobileNetV2, ImageNet, 9 connections.

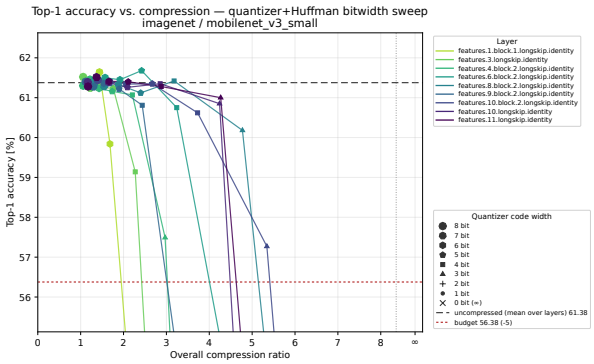


Figure A.42: Quantizer and Huffman bitwidth sweep, MobileNetV3-Small, ImageNet, 9 connections.

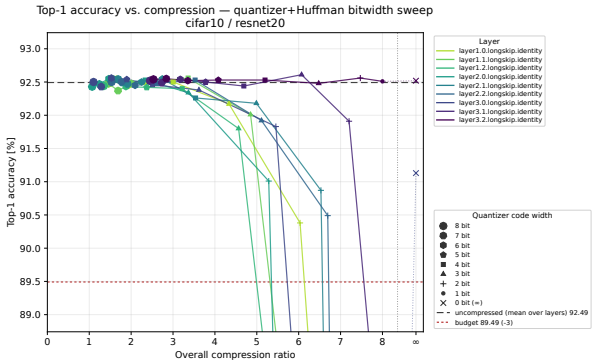


Figure A.43: Quantizer and Huffman bitwidth sweep, ResNet-20, CIFAR-10, 9 connections.

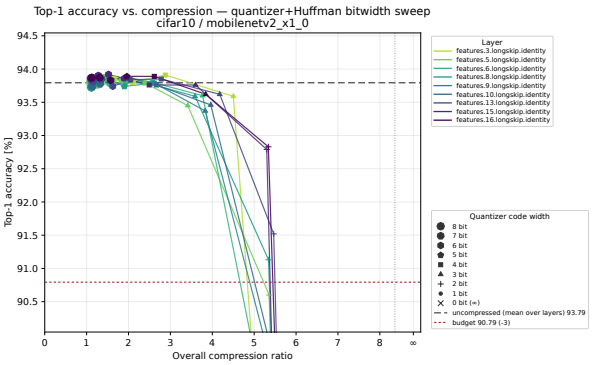


Figure A.44: Quantizer and Huffman bitwidth sweep, MobileNetV2, CIFAR-10, 9 connections.

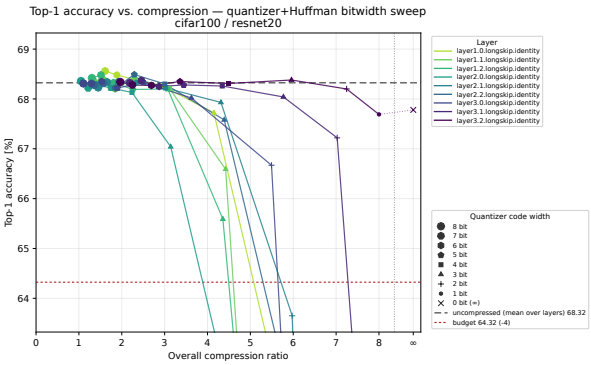


Figure A.45: Quantizer and Huffman bitwidth sweep, ResNet-20, CIFAR-100, 9 connections.

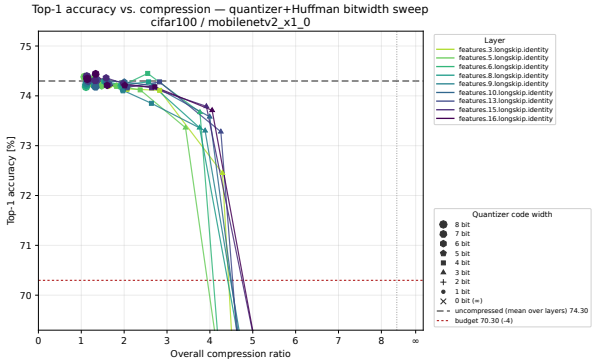


Figure A.46: Quantizer and Huffman bitwidth sweep, MobileNetV2, CIFAR-100, 9 connections.

A.10 Per-Channel Thresholding

Throughout this work, one value error threshold τ_v is set per layer and translated into a per-channel bound $\Delta_c^v = M_c\,\tau_v$ through the profiled channel magnitude M_c (section 3.2). The channels of a layer, however, do not contribute equally to the accuracy of the network. This raises the question whether channels that the network is less sensitive to could be given a laxer bound, and sensitive channels

a stricter one, to reach a higher compression ratio within the same accuracy budget.

Setup To measure how much each channel contributes, we compressed one channel at a time. We used an early version of our SZ compressor (constant, linear and quadratic prediction, row-major traversal, value error bounded relative to the channel magnitude) on the output of the first residual block (`layer1.0`) of ResNet-20 on CIFAR-10, which has 16 channels. For every channel c, we ran a separate sweep in which only channel c was compressed, while the other 15 channels passed the compressor unchanged. Each sweep increased τ_v from 0 to 0.5 in steps of 0.025 and measured the top-1 accuracy on 5000 test images. Since only one of 16 channels is compressed, the compression ratio of the layer stays at about 1.25 in every sweep; the accuracy drop at a given τ_v is therefore the impact of that channel alone.

Results Figure A.47 shows the accuracy of every sweep. Without compression, all sweeps reach about 92.3 %. At $\tau_v = 0.5$, most channels cost the network only 0.3 to 1.5 percentage points, while a few channels are considerably more sensitive: channel 11 drops the accuracy to 80.9 %, followed by channels 14, 7, 9 and 5 at 88.3 to 89.4 %.

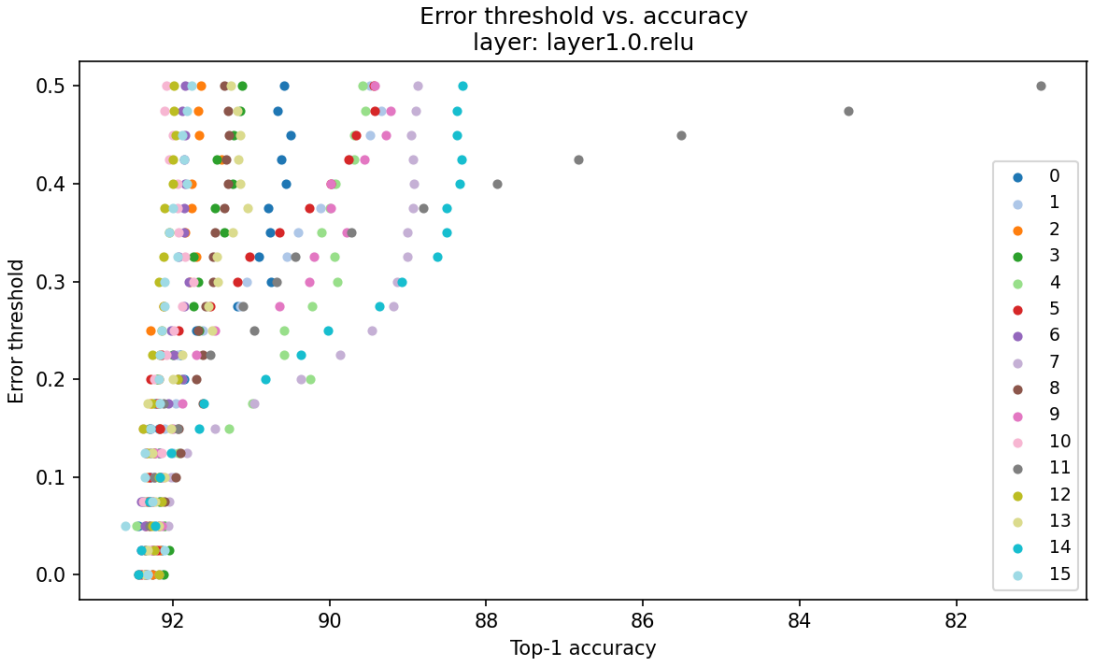


Figure A.47

Per-channel thresholds Based on these measurements, we derived per-channel thresholds from the relative impact of the channels: every channel received a coefficient κ_c that scales its bound to $\Delta_c^v = M_c\,\kappa_c\,\tau_v$, so that a single τ_v can still be swept for the whole layer. A coefficient of 0 keeps a channel lossless. We tried three sets of coefficients on the same layer:

- only five low-impact channels compressed (channels 1, 2, 6, 10 and 12, $\kappa_c = 1$), all others lossless ($\kappa_c = 0$);

- coefficients graded by impact: $\varkappa_c = 3$ for the same five channels, $\varkappa_c = 2$ for channels 0, 3, 8 and 13, $\varkappa_c = 1$ for the remaining ones, and $\varkappa_{11} = 0$ for the most sensitive channel;

- all channels at $\varkappa_c = 1$ except the most sensitive channel, $\varkappa_{11} = 0$.

None of these performed better than simply scaling the thresholds of all channels together. Within an accuracy drop of 1 percentage point from the 91.1 % of the uniform sweep, the uniform threshold reached a compression ratio of 2.20, compared to 1.28, 2.17 and 2.08 for the three sets; within 2 points, 2.48 compared to 1.29, 2.17 and 2.30. Every channel kept lossless caps the compression ratio, which limits the first set to at most 1.32. The other two sets fell behind the uniform threshold at every accuracy level below 91 %, by up to 0.6 in compression ratio; only at the near-lossless end did exempting channel 11 reach a slightly higher ratio (2.08 compared to 1.88 at 91.0 %). Note that the uniform sweep was evaluated on 5000 and the per-channel sweeps on 10000 test images. We therefore kept one threshold per layer.

A.11 Run-Length Encoding vs. Zero Run-Length Encoding

Both run-length variants of section 4.1.3.2 are lossless, so every configuration reaches the baseline accuracy and the two only differ in compression ratio. Figure A.48 therefore plots, for every skip connection of RegNetX-400MF, the compression ratio of RLE against that of zero RLE. Points above the diagonal are connections that zero RLE compresses better than RLE.

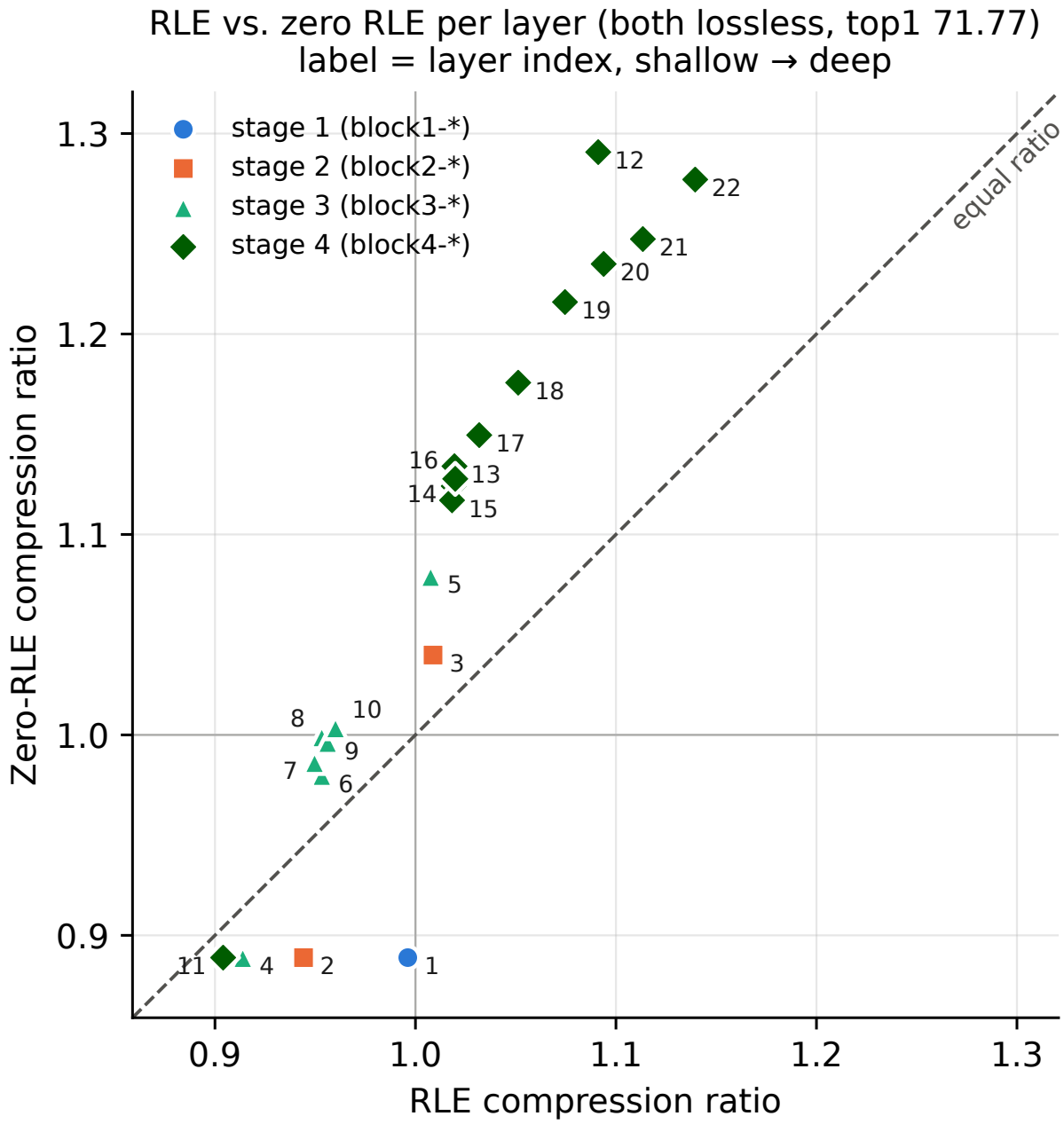


Figure A.48: Compression ratio of RLE (horizontal) against zero RLE (vertical) per skip connection, RegNetX-400MF; colour and shape mark the network stage, numbers the connection index from shallow to deep; dashed: equal ratio.

A.11.1 Run Structure of the Evaluated Connections

Both run-length variants expand most connections of chapter 4: RLE compresses only 30 of the 84 skip connections we measured, zero RLE only 33. These are the 74 connections of chapter 4 and the ten further connections of RegNetX-400MF that it does not evaluate. How far a connection can be compressed depends on the structure of its runs, which Figs. A.49 and A.50 show for all of them, measured on the test split.

Cost model Both variants spend a record of $8 + b$ bits, the value and a b-bit run field, on every run. RLE stores one record per run of equal values, so with a

mean chain length of L values per run, it reaches

$$\mathrm{CR}_{\mathrm{RLE}} \approx \frac{8\,L}{8+b},$$

and breaks even only at $L > 1 + b/8$. Zero RLE stores one record per non-zero value, with the zeros preceding it folded into its run field, so with a zero fraction z,

$$\mathrm{CR}_{\mathrm{ZRLE}} \approx \frac{8}{(1-z)\,(8+b)},$$

which breaks even at $z > b/(8+b)$, about 11 % for $b = 1$ and 20 % for $b = 2$. Both approximations ignore the spill records of runs longer than the run field.

The zero fraction alone does not decide For sparse formats such as CSR or CSC, the share of zeros is what counts. For run-length coding, the arrangement of the values matters as well: RLE gains only from runs of equal values, zero RLE only from zeros, and both pay a full record for every value that breaks a run. Zero RLE profits from every zero, wherever it lies, since a single zero between two non-zero values still saves a record; the length of the zero runs only determines how wide the run field has to be. RLE, in contrast, gains nothing from a zero, or any other value, that is not repeated by its neighbour. The two variants therefore respond to different properties of the same data, which Fig. A.48 shows per connection.

Zero fraction Figure A.49 shows that 41 of the 84 connections contain no zeros at all. These are the connections whose operand is not the output of a ReLU: all connections of MobileNetV2 and the additions of MobileNetV3-Small, whose inverted residual blocks end in a linear bottleneck, and the connections that carry the output of a projection, i.e. the first connection of every stage in RegNetX-400MF and of every stage but the first in ResNet-18 and ResNet-20. Here, zero means the smallest code of a channel after the per-channel shift (section 4.1.3.2); without a ReLU, this minimum is reached only rarely, and zero RLE costs a record of 9 bits for every 8-bit value (CR = 0.89). The ReLU outputs of the identity blocks reach zero fractions of 13 to 80 %, rising towards the deeper layers.

Ratio against run structure Figure A.50 plots the ratio of both variants against the property that drives it. The zero-RLE ratio follows the zero fraction closely and crosses 1 at about 15 % zeros; the 51 connections below this share are all expanded. Only the sparsest connections reach a substantial ratio, at most 3.1 for the last connection of ResNet-18 with 80 % zeros. The RLE ratio follows the average chain length in the same way. The median connection has a mean run of only 1.08 values, i.e. almost every value differs from its predecessor, and 54 of the 84 connections are expanded. Both follow the cost model with run fields of $b \approx 1$ to 2 bits for short runs and up to 4 to 5 bits for the longest ones.

Neither variant can therefore reach high ratios on 8-bit activations: even the connection with the longest runs, at 3.4 values per run, reaches only 2.1 with RLE. This is consistent with the per-connection results of chapter 4, where both variants rarely exceed a ratio of 1.5.

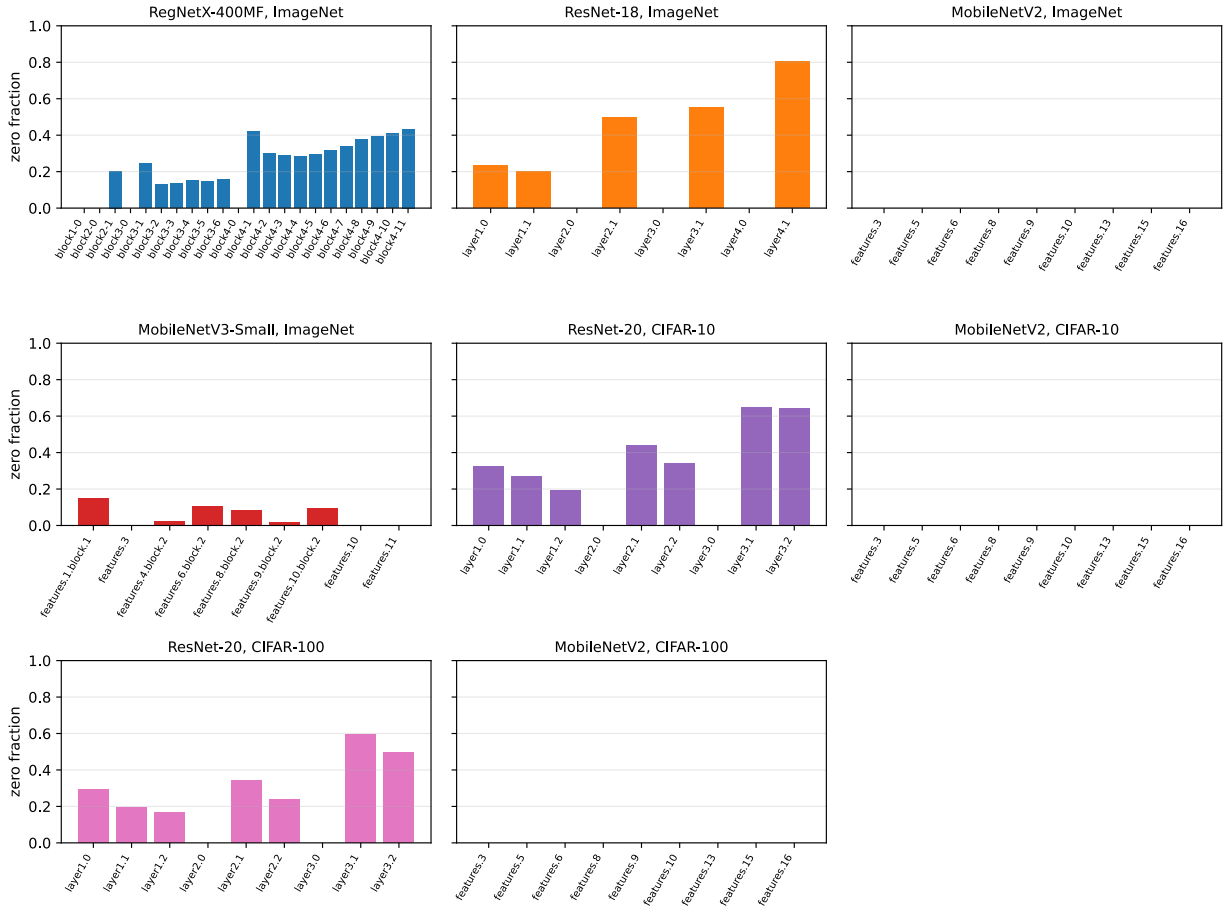


Figure A.49: Fraction of zero codes per skip connection, i.e. of values at the channel minimum after the per-channel shift, for the 74 skip connections of chapter 4 and the ten further connections of RegNetX-400MF, in order of depth, test split. Connections whose operand is a projection or a linear bottleneck, i.e. not a ReLU output, contain no zeros at all; this applies to every connection of MobileNetV2, the additions of MobileNetV3-Small, and the projection connections of RegNetX-400MF, ResNet-18 and ResNet-20.

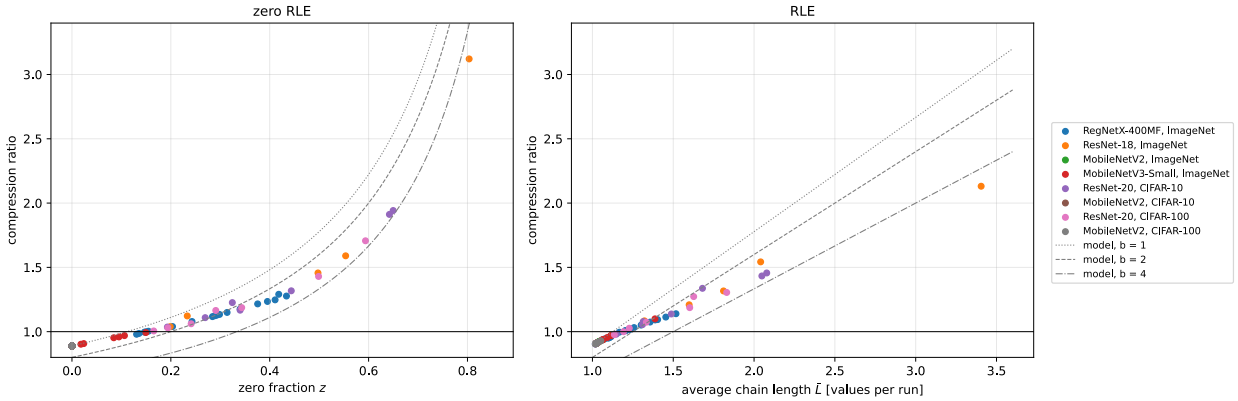


Figure A.50: Compression ratio of zero RLE against the zero fraction (left) and of RLE against the average chain length, the mean number of equal values per run (right), one point for each of the 84 connections of Fig. A.49, test split; colour: network. Grey: the cost model for a fixed run-field width b, $8/((1-z)(8+b))$ and $8L/(8+b)$; solid black: ratio 1. Zero RLE breaks even at about 15 % zeros, RLE at a chain length of about 1.2; most connections lie below either point and are expanded.

A.12 Number of Lorenzo Layers

The Lorenzo + Huffman baseline of section 4.1.3.7 uses the one-layer Lorenzo predictor, although SZ-1.4 [76] introduced it in its multilayer generalisation (eq. (3.19)). To choose the number of layers, we ran the same sweep of the relative error bound ρ with $n = 1$, 2 and 3 layers on ten skip connections of RegNetX-400MF, evaluated on the test split.

Results Figure A.51 shows the Pareto front of each variant per connection. The one-layer predictor dominates on all ten connections: within a 1 pp budget, it reaches a compression ratio of 3.10 on average, compared to 2.45 with two and 2.19 with three layers, and it reaches the highest ratio of the three on every connection. Its front also forms the common efficient frontier almost everywhere.

Interpretation The n-layer predictor extrapolates a polynomial surface of degree n from an $(n+1) \times (n+1)$ neighbourhood, with coefficients $\binom{n}{k_1}\binom{n}{k_2}$ that grow quickly with n (eq. (3.19)). The neighbours it reads are reconstructed values, each of which carries an error of up to the error bound. These errors enter the prediction weighted by the coefficients, so for independent errors, their effect on the prediction grows with the root of the summed squared coefficients: 1.7, 5.9 and 20 times the error of a single neighbour for $n = 1$, 2 and 3. Higher layers also assume a smoother signal than activations offer: post-ReLU featuremaps contain hard edges and runs of zeros, and their 8-bit values change in integer steps, so a higher-order extrapolation overshoots exactly where the data is not smooth. Both effects enlarge the residuals, which then fall into higher quantization bins or become outliers, and lower the compression ratio. Finally, an n-layer predictor needs n buffered rows and only applies at a distance of n from the top and left border, which matters on small featuremaps.

This matches the observation of Tao et al. [76], who introduced the multilayer predictor but found the one-layer version to be the best choice in most cases, for the same reason: higher layers amplify the error of the reconstructed values they predict from. Later SZ versions accordingly use the one-layer Lorenzo predictor [49].

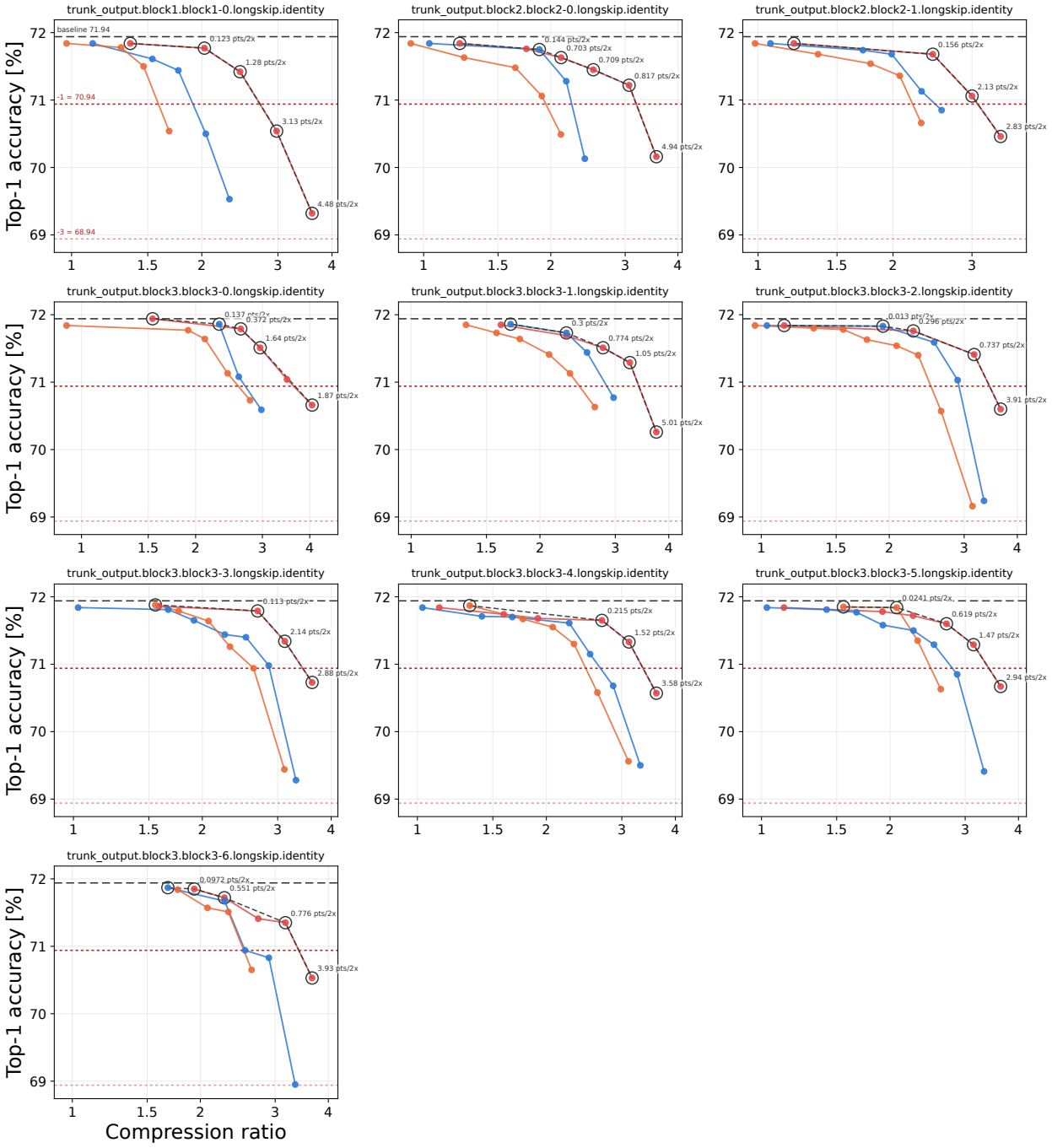


Figure A.51: Lorenzo + Huffman with one (**red**), two (**blue**) and three (**orange**) Lorenzo layers on ten skip connections of RegNetX-400MF, one panel per connection; top-1 accuracy on the test split against the compression ratio (logarithmic), each variant drawn with its own Pareto front over a sweep of the relative error bound ρ. Dashed black: baseline (71.94 %, an earlier version of the evaluation setup); dotted red: 1 and 3 pp budgets; dashed grey: common efficient frontier, labelled with the accuracy lost per doubling of the ratio. The one-layer predictor dominates on every connection; the higher layers amplify the error of the reconstructed neighbours they predict from.

A.13 Downsampling in Isolation

To choose a downsampling method for the pipeline of section 3.6.1, we evaluated downsampling without any subsequent compression stage on the connection used throughout chapter 3. We combined the two downsampling methods of

section 2.5.2.2, strided subsampling and average pooling, with nearest-neighbour and bilinear upsampling, and swept the downsampling factor. Since nothing else is compressed, the compression ratio is exactly f_{down}^2 for a factor f_{down}.

Figure A.52 shows the results. Average pooling loses less accuracy than strided subsampling at every factor, and nearest-neighbour upsampling is slightly ahead of bilinear interpolation. The differences only become large at higher factors, however: within a 3 pp budget, only a factor of 2, i. e. a compression ratio of 4, holds, and it does so for all four combinations (Fig. A.53). At the factors relevant for our pipeline, the four combinations thus perform comparably.

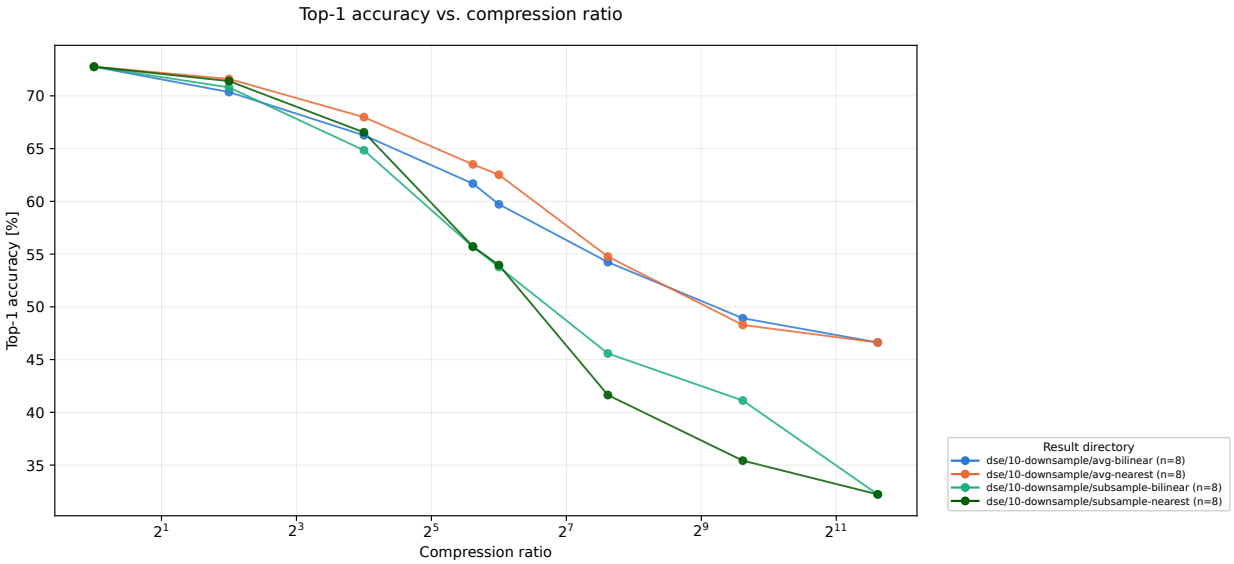


Figure A.52: Downsampling alone, RegNetX-400MF, `block1-0`: top-1 accuracy against compression ratio (logarithmic) for average pooling and strided subsampling, each upsampled again by nearest-neighbour or bilinear interpolation, over a sweep of the downsampling factor. Average pooling loses less accuracy than subsampling at every factor, with nearest-neighbour upsampling slightly ahead of bilinear.

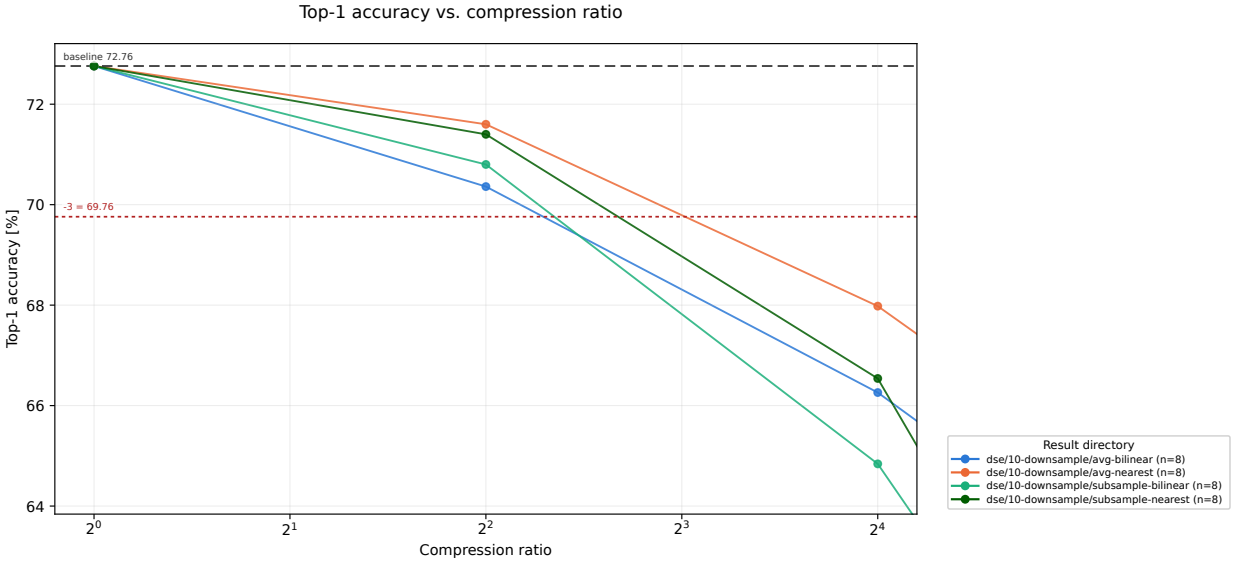


Figure A.53: Detail of Fig. A.52 within a 3 pp budget (red dashed line below the baseline). Only factor 2 (4×) stays within the budget, for all four variants.

A.14 Best Compression Ratio per Connection

Table A.1 lists, for every connection of section 4.2, the best compression ratio of every method within the 1 pp budget, together with the shape of the connection.

		Lossless			Decim.	Transform		Quantization		Prediction		
Connection	$C \times H \times W$	RLE	ZRLE	Huff.	DS	SVD	JPEG	QH	Lor.	SZ1	ASZ_{st}	ASZ_{dyn}
ImageNet, RegNetX-400MF (baseline 71.77 %)												
`block1-0`	32 × 56 × 56	1.00	0.89	1.47	–	5.02	**11.29**	1.83	2.98	2.18	4.28	4.14
`block2-0`	64 × 28 × 28	0.94	0.89	1.33	–	3.48	**8.65**	2.34	3.07	2.70	3.26	3.25
`block2-1`	64 × 28 × 28	1.01	1.04	1.58	–	1.78	**4.53**	2.84	3.00	2.27	3.25	3.20
`block3-0`	160 × 14 × 14	0.91	0.89	1.23	–	2.42	**6.07**	3.44	4.06	3.40	3.40	3.50
`block3-1`	160 × 14 × 14	1.01	1.08	1.55	–	1.36	3.23	**4.18**	3.25	2.32	2.60	2.57
`block3-2`	160 × 14 × 14	0.95	0.98	1.38	–	1.36	3.15	**4.10**	3.65	2.63	2.95	2.78
`block4-0`	400 × 7 × 7	0.90	0.89	1.20	–	5.44	**10.70**	5.88	3.80	2.96	3.70	3.61
`block4-1`	400 × 7 × 7	1.09	1.29	1.88	–	1.36	3.48	**5.41**	3.94	2.74	3.08	2.93
`block4-3`	400 × 7 × 7	1.02	1.12	1.61	–	1.36	3.56	**5.27**	3.95	2.80	3.14	2.88
`block4-4`	400 × 7 × 7	1.02	1.12	1.60	–	1.36	4.32	**5.27**	3.99	2.85	3.14	3.08
`block4-10`	400 × 7 × 7	1.11	1.25	1.93	–	5.44	**15.41**	6.27	4.72	3.43	3.45	3.42
`block4-11`	400 × 7 × 7	1.14	1.28	2.01	**49.00**	**49.00**	16.81	7.53	5.27	4.16	3.62	3.56
ImageNet, ResNet-18 (baseline 68.35 %)												
`layer1.0`	64 × 56 × 56	1.54	1.12	1.87	–	3.06	**10.65**	3.07	4.79	4.62	5.02	5.30
`layer1.1`	64 × 56 × 56	1.08	1.04	1.68	–	2.42	**7.78**	4.69	5.25	3.81	3.98	4.42
`layer2.0`	128 × 28 × 28	0.92	0.89	1.29	–	2.42	**5.80**	3.97	3.65	2.89	3.01	3.00
`layer2.1`	128 × 28 × 28	1.21	1.46	2.19	–	4.00	**8.34**	5.56	5.00	3.08	3.94	3.89
`layer3.0`	256 × 14 × 14	0.91	0.89	1.27	4.00	21.78	**41.24**	5.62	4.66	4.77	13.26†	13.06†
`layer3.1`	256 × 14 × 14	1.32	1.59	2.35	–	4.00	**12.53**	5.67	5.27	3.96	4.16	4.05
`layer4.0`	512 × 7 × 7	0.91	0.89	1.26	–	3.06	**7.70**	4.13	3.84	3.37	2.78	2.68
`layer4.1`	512 × 7 × 7	2.13	3.12	3.97	**49.00**	**49.00**	46.45	8.00	6.44	5.04	45.84†	45.46†
ImageNet, MobileNetV2 (baseline 67.87 %)												
`features.3`	24 × 56 × 56	0.94	0.89	1.32	–	1.87	**6.20**	2.91	3.86	1.74	2.82	2.81
`features.5`	32 × 28 × 28	0.92	0.89	1.23	–	1.78	3.30	2.87	**3.38**	2.38	2.53	2.63
`features.6`	32 × 28 × 28	0.91	0.89	1.25	–	1.78	**5.52**	2.92	3.29	1.80	2.35	2.26
`features.8`	64 × 14 × 14	0.91	0.89	1.21	–	1.36	2.94	2.80	**3.08**	2.05	1.96	1.96
`features.9`	64 × 14 × 14	0.91	0.89	1.21	–	1.16	2.87	2.85	**3.05**	1.63	2.04	1.91
`features.10`	64 × 14 × 14	0.91	0.89	1.22	–	1.16	2.10	2.89	**3.01**	1.95	1.78	1.98
`features.13`	96 × 14 × 14	0.91	0.89	1.26	–	1.62	**3.76**	2.95	3.10	2.15	2.31	2.30
`features.15`	160 × 7 × 7	0.92	0.89	1.38	–	1.36	2.61	2.19	**2.66**	1.60	1.99	2.00
`features.16`	160 × 7 × 7	0.92	0.89	1.38	–	1.36	**3.62**	2.25	2.69	1.62	2.04	2.04
ImageNet, MobileNetV3-Small (baseline 61.52 %)												
`features.1.block.1`	16 × 56 × 56	1.10	0.99	1.85	–	1.08	**2.47**	1.44	1.51	1.55	1.85	1.78
`features.3`	24 × 28 × 28	0.94	0.89	1.30	–	2.42	**4.88**	1.76	2.47	1.70	2.77	2.48
`features.4.block.2`	96 × 14 × 14	0.94	0.91	1.37	–	1.16	1.92	2.21	**2.31**	1.65	2.11	2.09
`features.6.block.2`	240 × 14 × 14	0.97	0.97	1.58	–	1.62	2.49	**3.24**	2.92	1.77	2.68	2.64
`features.8.block.2`	144 × 14 × 14	0.95	0.95	1.50	–	2.42	**6.21**	3.18	4.10	2.53	2.91	2.89
`features.9.block.2`	288 × 7 × 7	0.93	0.90	1.40	–	1.36	**2.65**	2.44	2.38	1.42	1.94	2.02
`features.10.block.2`	576 × 7 × 7	0.97	0.96	1.57	–	3.06	**6.86**	3.73	3.03	2.21	2.82	2.76
`features.10`	96 × 7 × 7	0.91	0.89	1.24	–	1.36	3.51	**4.24**	3.26	1.89	1.79	1.91
`features.11`	96 × 7 × 7	0.91	0.89	1.27	–	3.06	**6.23**	4.27	3.45	2.49	2.86	2.40
CIFAR-10, ResNet-20 (baseline 92.48 %)												
`layer1.0`	16 × 32 × 32	1.34	1.23	1.83	4.00	16.00	**19.34**	4.32	5.91	7.16	18.31†	16.48†
`layer1.1`	16 × 32 × 32	1.08	1.11	1.72	–	3.16	**6.99**	4.90	4.58	3.33	4.08	3.87
`layer1.2`	16 × 32 × 32	0.98	1.04	1.48	–	3.54	**14.42**	4.61	4.03	2.70	4.11	3.91
`layer2.0`	32 × 16 × 16	0.92	0.89	1.23	–	5.22	**14.53**	3.33	4.21	4.54	–	–
`layer2.1`	32 × 16 × 16	1.14	1.32	1.92	–	3.16	**6.42**	4.99	4.76	3.05	3.94	3.64
`layer2.2`	32 × 16 × 16	1.06	1.17	1.68	–	7.11	**12.23**	5.09	4.84	3.27	4.04	3.97
`layer3.0`	64 × 8 × 8	0.91	0.89	1.22	**64.00**	**64.00**	44.77	5.52	4.41	3.18	59.52†	59.27†
`layer3.1`	64 × 8 × 8	1.43	1.94	2.74	–	7.11	**14.87**	7.22	5.66	3.90	–	–
`layer3.2`	64 × 8 × 8	1.46	1.91	2.82	**64.00**	**64.00**	56.30	8.00	6.14	4.82	59.52†	59.27†
CIFAR-10, MobileNetV2 (baseline 93.81 %)												
`features.3`	24 × 32 × 32	0.93	0.89	1.28	–	7.11	**17.75**	4.51	4.67	4.00	–	–
`features.5`	32 × 16 × 16	0.92	0.89	1.19	–	3.16	**8.87**	3.42	4.34	3.11	3.55	3.50
`features.6`	32 × 16 × 16	0.91	0.89	1.21	–	4.00	**8.78**	3.78	4.33	3.11	3.47	2.97
`features.8`	64 × 8 × 8	0.91	0.89	1.19	–	1.78	**5.64**	3.58	3.94	2.56	2.66	2.79
`features.9`	64 × 8 × 8	0.91	0.89	1.20	–	1.78	**5.52**	3.91	3.85	2.45	2.25	2.60

Table A.1, continued

		Lossless			Decim.	Transform		Quantization		Prediction		
Connection	$C \times H \times W$	RLE	ZRLE	Huff.	DS	SVD	JPEG	QH	Lor.	SZ1	ASZ_{st}	ASZ_{dyn}
`features.10`	64 × 8 × 8	0.91	0.89	1.20	–	1.78	**5.47**	4.08	3.81	2.40	2.51	2.49
`features.13`	96 × 8 × 8	0.91	0.89	1.23	–	4.00	**12.52**	4.19	4.87	3.40	3.09	3.08
`features.15`	160 × 4 × 4	0.91	0.89	1.21	–	1.78	2.79	**5.28**	4.57	2.25	2.27	2.24
`features.16`	160 × 4 × 4	0.91	0.89	1.21	–	4.00	2.76	**5.34**	4.50	2.25	2.36	2.33
CIFAR-100, ResNet-20 (baseline 68.33 %)												
`layer1.0`	16 × 32 × 32	1.27	1.16	1.76	–	6.06	**20.00**	4.16	5.43	4.87	5.12	4.92
`layer1.1`	16 × 32 × 32	1.03	1.03	1.58	–	2.12	**5.00**	3.15	3.93	2.69	3.11	3.06
`layer1.2`	16 × 32 × 32	0.98	1.01	1.45	–	2.32	**7.16**	3.07	3.52	2.32	3.11	3.03
`layer2.0`	32 × 16 × 16	0.91	0.89	1.21	–	2.12	**6.94**	3.11	2.92	2.33	2.50	2.47
`layer2.1`	32 × 16 × 16	1.07	1.19	1.68	–	2.12	**6.08**	4.34	4.47	2.34	2.82	2.79
`layer2.2`	32 × 16 × 16	1.00	1.06	1.50	–	7.11	**7.85**	4.42	3.94	2.44	3.22	3.20
`layer3.0`	64 × 8 × 8	0.90	0.89	1.20	4.00	7.11	**13.22**	3.64	3.57	3.07	3.31	2.91
`layer3.1`	64 × 8 × 8	1.30	1.71	2.45	4.00	4.00	**13.73**	7.01	5.48	3.30	3.53	3.44
`layer3.2`	64 × 8 × 8	1.19	1.43	2.21	**64.00**	**64.00**	47.70	8.00	5.82	4.38	59.52†	59.27†
CIFAR-100, MobileNetV2 (baseline 74.28 %)												
`features.3`	24 × 32 × 32	0.93	0.89	1.30	–	2.32	**5.56**	2.81	3.60	2.68	2.62	2.67
`features.5`	32 × 16 × 16	0.91	0.89	1.18	–	2.12	**5.19**	3.44	3.69	2.55	2.67	2.33
`features.6`	32 × 16 × 16	0.91	0.89	1.20	–	3.16	**5.96**	3.74	3.68	2.51	2.78	2.69
`features.8`	64 × 8 × 8	0.91	0.89	1.17	–	1.31	**3.34**	2.57	3.28	2.10	2.18	2.03
`features.9`	64 × 8 × 8	0.91	0.89	1.18	–	1.31	3.30	**3.88**	3.22	1.74	2.01	1.99
`features.10`	64 × 8 × 8	0.91	0.89	1.19	–	1.31	3.27	**4.00**	3.18	1.71	2.21	2.02
`features.13`	96 × 8 × 8	0.91	0.89	1.20	–	2.56	**4.91**	4.23	3.36	2.20	2.52	2.45
`features.15`	160 × 4 × 4	0.91	0.89	1.22	–	1.78	1.15	**3.98**	2.98	1.76	1.79	1.79
`features.16`	160 × 4 × 4	0.91	0.89	1.21	–	1.78	1.62	**4.09**	2.95	1.90	1.98	1.87

Table A.1: Best compression ratio of every method on every evaluated connection within the 1 pp budget, i. e. the highest ratio among the results on the test split with a top-1 accuracy of at least the baseline minus 1 pp, the quantity the layer-line figures of section 4.2 draw. Rows are ordered by dataset, model and depth, columns by compressor group. Bold: highest ratio of the connection. –: no operating point within the budget (for downsampling, only the factor 1, i. e. no reduction). †: the parameter search of ASZ chose a downsampling factor. Lossless methods have a single operating point; ratios below 1 are expansions.

B | Model Graphs

This appendix shows the compute graphs of the models of table 4.1, as traced with `torch.fx`. Each graph is read column by column from left to right and top to bottom. A dashed arrow marks where a column continues in the next one. Every linear chain of operations is merged into one node, which lists the operations of the chain in order: Conv (convolution), BN (batch normalization), ReLU, ReLU6, HSwish and HSigmoid (hard swish and hard sigmoid), AvgPool and MaxPool (average and max pooling), Flatten, Dropout, FC (fully connected layer), + (residual addition) and × (element-wise multiplication). Green nodes are the skip connections at which activations can be compressed, named after their module as in table 4.1.

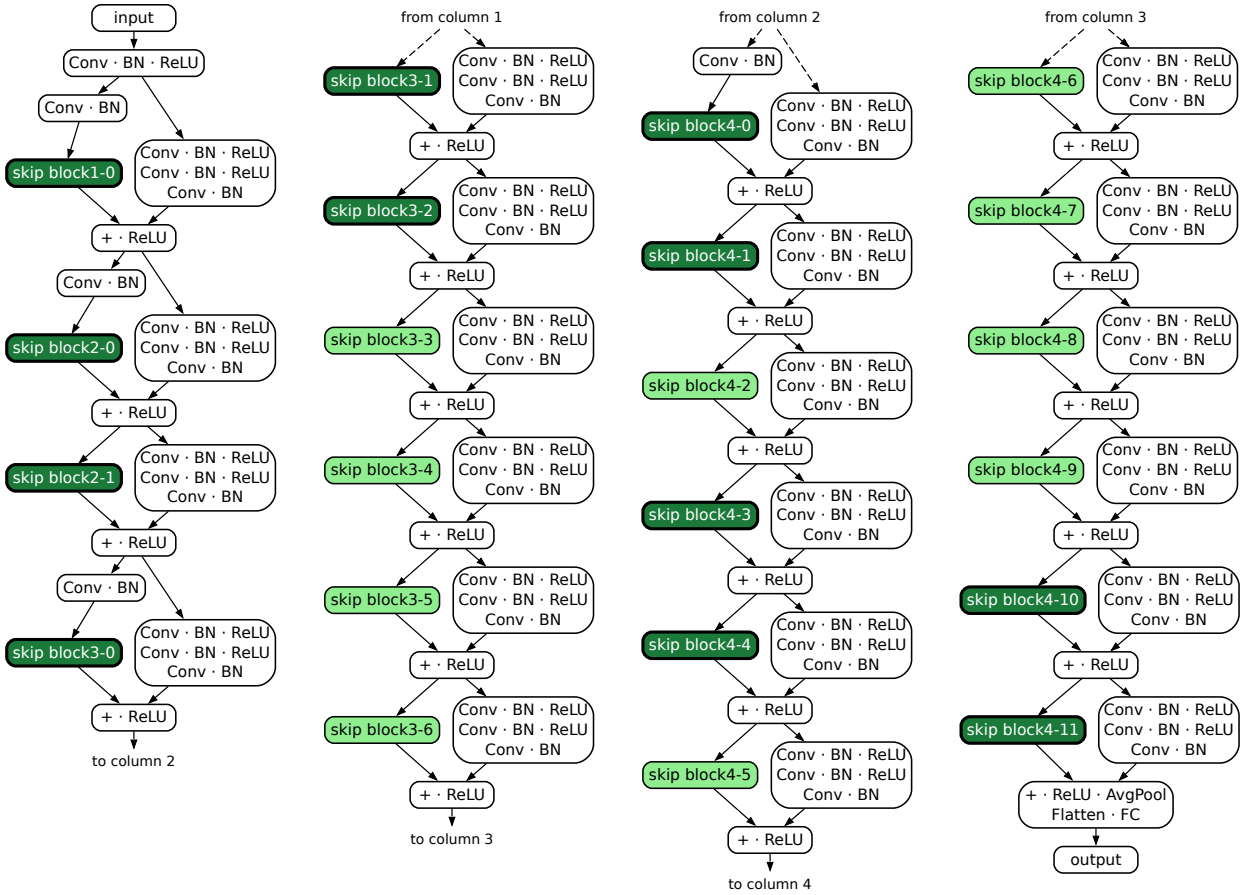


Figure B.1: Compute graph of RegNetX-400MF (ImageNet), with its 22 skip connections. Dark green nodes with a bold outline: the 12 connections compressed in the evaluation (table 4.1); light green nodes: the remaining skip connections.

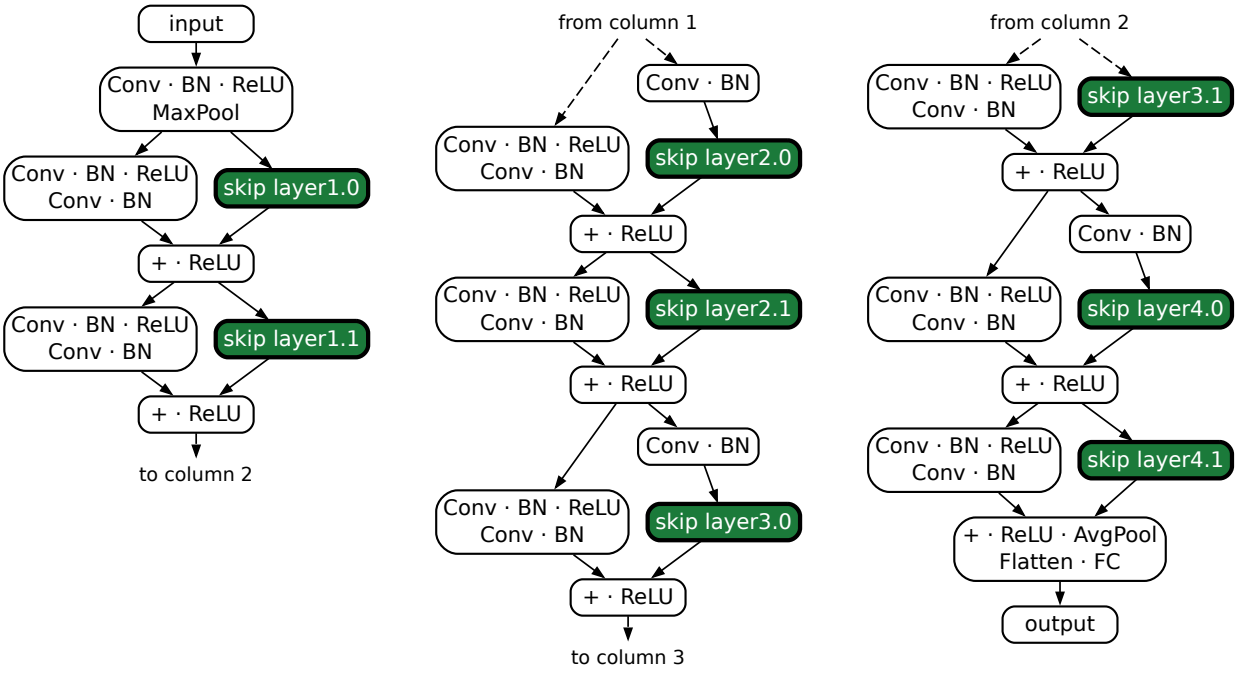


Figure B.2: Compute graph of ResNet-18 (ImageNet), with its 8 skip connections. Dark green nodes with a bold outline: the 8 connections compressed in the evaluation (table 4.1); light green nodes: the remaining skip connections.

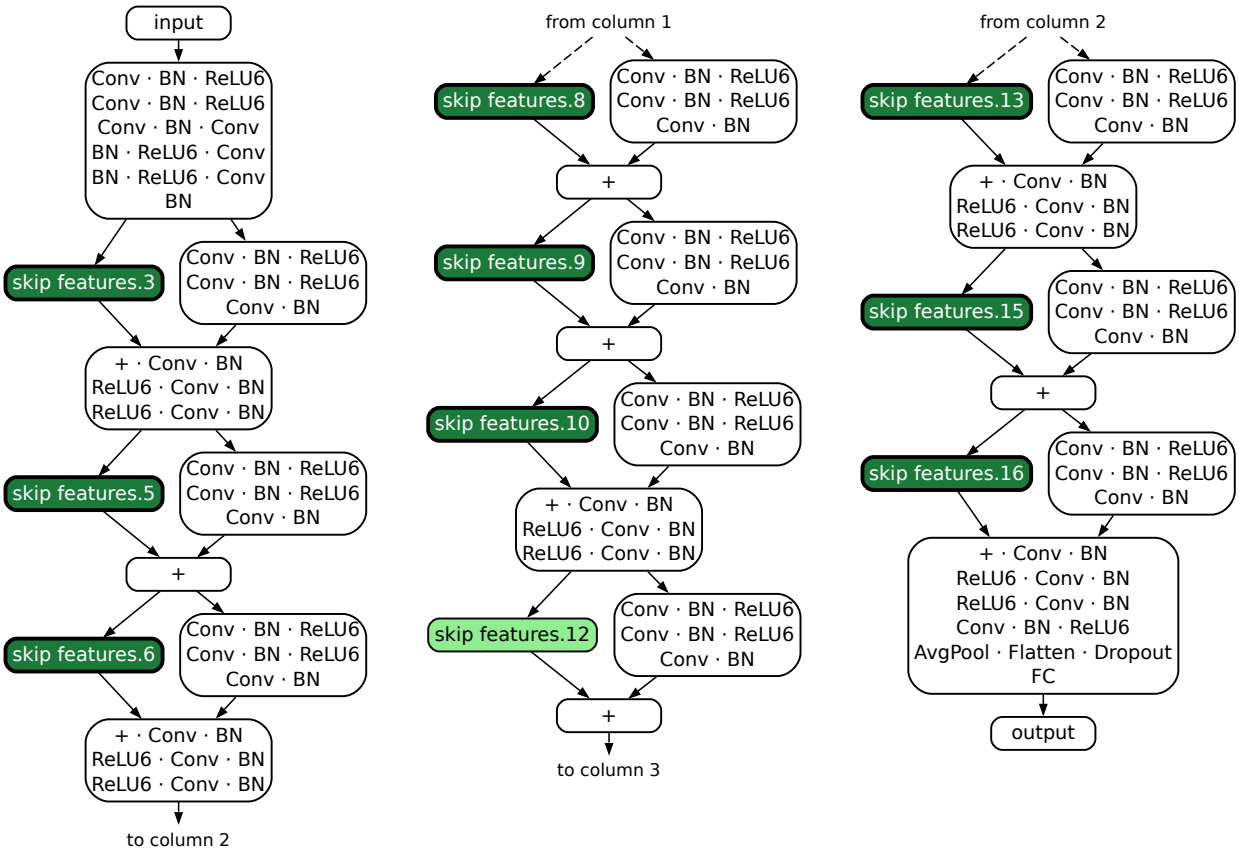


Figure B.3: Compute graph of MobileNetV2, with its 10 skip connections. The graphs of the ImageNet, CIFAR-10 and CIFAR-100 versions consist of the same operations. Dark green nodes with a bold outline: the 9 connections compressed in the evaluation (table 4.1); light green nodes: the remaining skip connections.

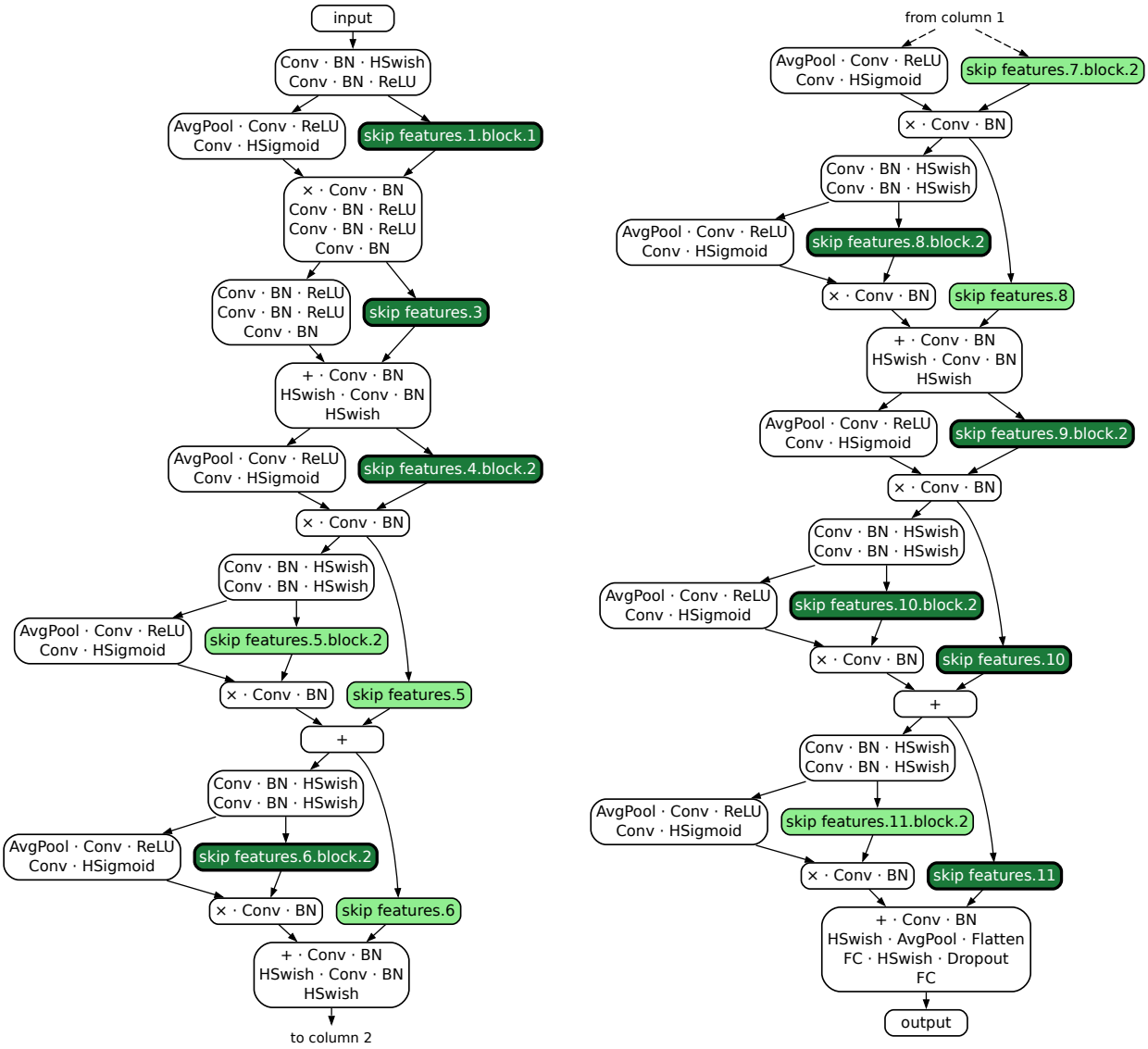


Figure B.4: Compute graph of MobileNetV3-Small (ImageNet), with its 15 skip connections, including the inputs of the squeeze-and-excitation multiplications (`.block.1`, `.block.2`). Dark green nodes with a bold outline: the 9 connections compressed in the evaluation (table 4.1); light green nodes: the remaining skip connections.

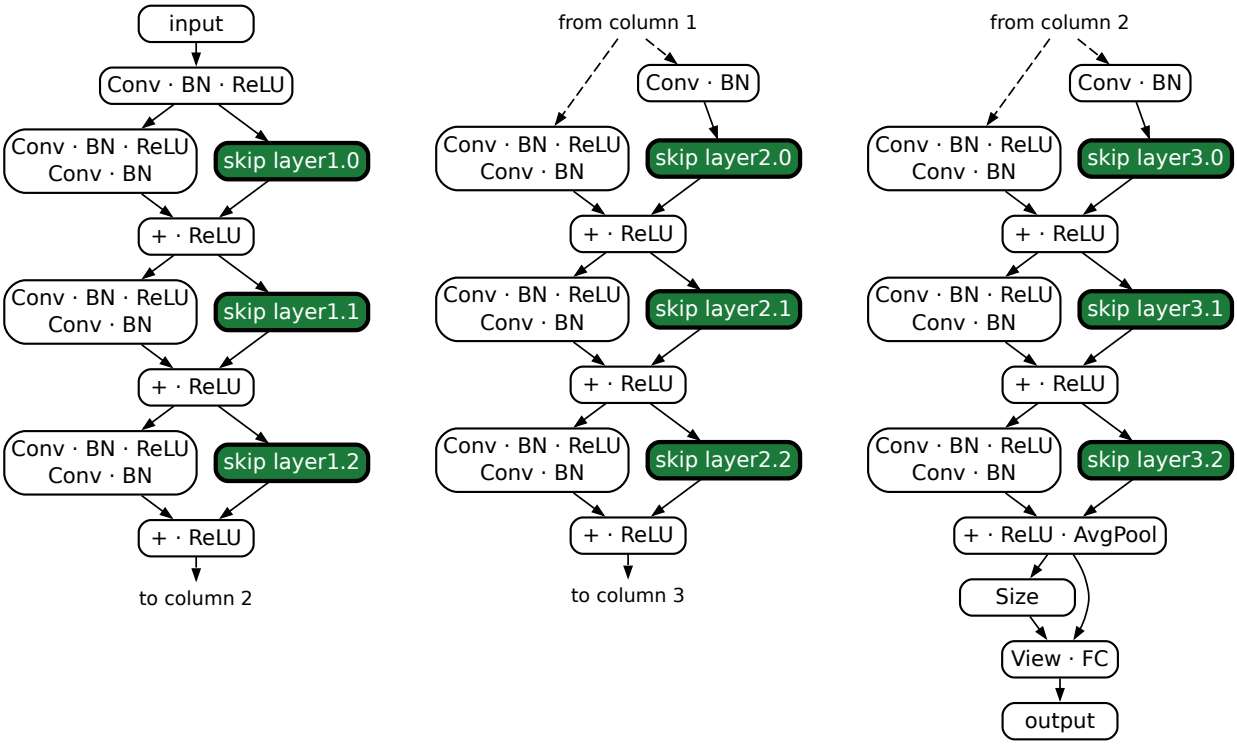


Figure B.5: Compute graph of ResNet-20, with its 9 skip connections. The graphs of the CIFAR-10 and CIFAR-100 versions consist of the same operations. Dark green nodes with a bold outline: the 9 connections compressed in the evaluation (table 4.1); light green nodes: the remaining skip connections.

References

[1] A Agrawal et al. Single-Stage Huffman Encoder for ML Compression. In: *arXiv preprint arXiv:2601.10673*:(2026) (page 28).

[2] N Ahmed, T Natarajan, and KR Rao. Discrete Cosine Transform. In: *IEEE Transactions on Computers* C-23(1):(1974), 90–93 (pages 19, 95).

[3] T Alonso et al. Elastic-df: Scaling performance of dnn inference in fpga clouds through automatic partitioning. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 15(2):(2021), 1–34 (page 12).

[4] C Baskin et al. CAT: Compression-Aware Training for Bandwidth Reduction. In: *Journal of Machine Learning Research* 22:(2021), 1–20 (pages 29, 32, 76, 142).

[5] J Boudjadar, SU Islam, and R Buyya. Dynamic FPGA reconfiguration for scalable embedded artificial intelligence (AI): A co-design methodology for convolutional neural networks (CNN) acceleration. In: *Future Generation Computer Systems* 169:(2025), 107777 (page 12).

[6] S Chandak et al. LFZip: Lossy compression of multivariate floating-point time series data via improved prediction. In: *2020 Data Compression Conference (DCC)*. IEEE. 2020, pp. 342–351 (pages 30, 52).

[7] Y Chen. PyTorch CIFAR Models. https : / / github . com/chenyaofo/pytorch-cifar-models. Accessed: 2026-10-22 (page 94).

[8] H Cheng, M Zhang, and JQ Shi. A survey on deep neural network pruning: Taxonomy, comparison, analysis, and recommendations. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 46(12):(2024), 10558–10578 (pages 2, 13).

[9] Ö Çiçek et al. 3D U-Net: learning dense volumetric segmentation from sparse annotation. In: *International conference on medical image computing and computer-assisted intervention*. Springer. 2016, pp. 424–432 (page 142).

[10] J Deng et al. ImageNet: A Large-Scale Hierarchical Image Database. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2009, pp. 248–255 (page 94).

[11] EL Denton et al. Exploiting linear structure within convolutional networks for efficient evaluation. In: *Advances in Neural Information Processing Systems*. Vol. 27. 2014 (page 14).

[12] P Dhilleswararao et al. Efficient hardware architectures for accelerating deep neural networks: Survey. In: *IEEE access* 10:(2022), 131788–131828 (pages 1, 7–9).

[13] S Di and F Cappello. Fast error-bounded lossy HPC data compression with SZ. In: *2016 ieee international parallel and distributed processing symposium (ipdps)*. IEEE. 2016, pp. 730–739 (pages 2, 4, 17, 23–27, 29, 33, 36, 40, 41, 48, 77, 140, 161).

[14] S Di et al. A survey on error-bounded lossy compression for scientific datasets. In: *ACM computing surveys* 57(11):(2025), 1–38 (pages 4, 17, 20, 21, 23, 24, 29, 43, 66).

[15] C Ding and J Ye. Two-dimensional singular value decomposition (2DSVD) for 2D maps and images. In: *Proceedings of the 2005 SIAM International Conference on Data Mining*. 2005, pp. 32–43 (page 23).

[16] Z Dong et al. Hawq: Hessian aware quantization of neural networks with mixed-precision. In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2019, pp. 293–302 (pages 14, 15, 21).

[17] C Eckart and G Young. The approximation of one matrix by another of lower rank. In: *Psychometrika* 1(3):(1936), 211–218 (page 22).

[18] A Eldstål-Ahrens and I Sourdis. Memsz: Squeezing memory traffic with lossy compression. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 17(4):(2020), 1–25 (pages 2, 4, 30, 32, 33, 44, 48, 143).

[19] A Eldstål-Damlin, P Trancoso, and I Sourdis. AVR: Reducing Memory Traffic with Approximate Value Reconstruction. In: *Proceedings of the International Conference on Parallel Processing (ICPP)*. 2019 (pages 4, 30, 32).

[20] RD Evans and T Aamodt. Ac-gc: Lossy activation compression with guaranteed convergence. In: *Advances in Neural Information Processing Systems* 34:(2021), 27434–27448 (page 28).

[21] RD Evans, L Liu, and TM Aamodt. JPEG-ACT: Accelerating deep learning via transform-based lossy compression. In: *2020 acm/ieee 47th annual international symposium on computer architecture (isca)*. IEEE. 2020, pp. 860–873 (pages 3, 4, 28, 54).

[22] J Fowers et al. A configurable cloud-scale DNN processor for real-time AI. In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2018, pp. 1–14 (page 10).

[23] K Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. In: *Biological cybernetics* 36(4):(1980), 193–202 (page 9).

[24] G Gambardella et al. Efficient error-tolerant quantized neural network accelerators. In: *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE. 2019, pp. 1–6 (pages 4, 31).

[25] V Gokhale et al. Snowflake: A model agnostic accelerator for deep convolutional neural networks. In: *arXiv preprint arXiv:1708.02579*:(2017) (pages 1, 10).

[26] SW Golomb. Run-Length Encodings. In: *IEEE Transactions on Information Theory* 12(3):(1966), 399–401 (pages 96, 97).

[27] GH Golub and CF Van Loan. *Matrix Computations*. 4th ed. Johns Hopkins University Press, 2013 (page 22).

[28] RM Gray and DL Neuhoff. Quantization. In: *IEEE Transactions on Information Theory* 44(6):(1998), 2325–2383 (page 21).

[29] K Guo et al. Angel-eye: A complete design flow for mapping CNN onto embedded FPGA. In: *IEEE transactions on computer-aided design of integrated circuits and systems* 37(1):(2017), 35–47 (pages 1, 10).

[30] M Hall and V Betz. HPIPE: Heterogeneous layer-pipelined and sparse-aware CNN inference for FPGAs. In: *arXiv preprint arXiv:2007.10451*:(2020) (pages 10, 14).

[31] S Han, H Mao, and WJ Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In: *arXiv preprint arXiv:1510.00149*:(2015) (pages 2, 4, 13, 21, 24, 28).

[32] K He et al. Deep residual learning for image recognition. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778 (pages 1, 3, 9, 35).

[33] A Heilper and D Singer. Lossless compression of neural network components: Weights, checkpoints, and k/v caches in low-precision formats. In: *arXiv preprint arXiv:2508.19263*:(2025) (page 28).

[34] M Hershcovitch et al. Zipnn: Lossless compression for ai models. In: *2025 IEEE 18th International Conference on Cloud Computing (CLOUD)*. IEEE. 2025, pp. 186–198 (page 28).

[35] G Hinton, O Vinyals, and J Dean. Distilling the knowledge in a neural network. In: *arXiv preprint arXiv:1503.02531*:(2015) (page 14).

[36] Y Hu, Y Liu, and Z Liu. A survey on convolutional neural network accelerators: GPU, FPGA and ASIC. In: *2022 14th International Conference on Computer Research and Development (ICCRD)*. IEEE. 2022, pp. 100–107 (pages 7–9).

[37] DA Huffman. A method for the construction of minimum-redundancy codes. In: *Proceedings of the IRE* 40(9):(1952), 1098–1101 (page 19).

[38] M Ibrahim et al. Extending data flow architectures for convolutional neural networks to multiple fpgas. In: *2023 International Conference on Field Programmable Technology (ICFPT)*. IEEE. 2023, pp. 132–141 (page 12).

[39] *IEEE Standard for Floating-Point Arithmetic*. IEEE, 2019 (page 13).

[40] M Jaderberg, A Vedaldi, and A Zisserman. Speeding up convolutional neural networks with low rank expansions. In: *Proceedings of the British Machine Vision Conference (BMVC)*. 2014 (page 14).

[41] S Jin et al. DeepSZ: A novel framework to compress deep neural networks by using error-bounded lossy compression. In: *Proceedings of the 28th international symposium on high-performance parallel and distributed computing*. 2019, pp. 159–170 (pages 2, 4, 9, 31, 32).

[42] S Jin et al. Comet: a novel memory-efficient deep learning training framework by using error-bounded lossy compression. In: *arXiv preprint arXiv:2111.09562*:(2021) (pages 4, 28, 31, 32).

[43] YD Kim et al. Compression of deep convolutional neural networks for fast and low power mobile applications. In: *International Conference on Learning Representations (ICLR)*. 2016 (page 14).

[44] Y Ko et al. Lane compression: A lightweight lossless compression method for machine learning on embedded systems. In: *ACM Transactions on Embedded Computing Systems (TECS)* 20(2):(2021), 1–26 (pages 3, 17, 27, 29, 32, 77, 140, 161).

[45] A Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. Tech. rep. University of Toronto, 2009 (page 94).

[46] A Krizhevsky, I Sutskever, and GE Hinton. Imagenet classification with deep convolutional neural networks. In: *Advances in neural information processing systems* 25:(2012) (pages 1, 9).

[47] V Lebedev et al. Speeding-up convolutional neural networks using fine-tuned CP-decomposition. In: *International Conference on Learning Representations (ICLR)*. 2015 (page 14).

[48] Y LeCun, J Denker, and S Solla. Optimal brain damage. In: *Advances in neural information processing systems* 2:(1989) (page 13).

[49] X Liang et al. Error-Controlled Lossy Compression Optimized for High Compression Ratios of Scientific Datasets. In: *IEEE International Conference on Big Data (Big Data)*. 2018, pp. 438–447 (pages 27, 28, 30, 66, 176).

[50] X Liang et al. Sz3: A modular framework for composing prediction-based error-bounded lossy compressors. In: *IEEE Transactions on Big Data* 9(2):(2022), 485–498 (pages 4, 27, 30, 66).

[51] SM Lim and SW Jun. Mobilenets can be lossily compressed: Neural network compression for embedded accelerators. In: *Electronics* 11(6):(2022), 858 (pages 29, 32).

[52] J Liu et al. Dynamic Quality Metric Oriented Error Bounded Lossy Compression for Scientific Datasets. In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2022 (page 30).

[53] C Metz, O Bichler, and A Dupret. Efficient neural compression with inference-time decoding. In: *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2024, pp. 1–5 (page 28).

[54] S Mittal. A survey of FPGA-based accelerators for convolutional neural networks. In: *Neural computing and applications* 32(4):(2020), 1109–1139 (pages 1, 12).

[55] A Montgomerie-Corcoran et al. SATAY: A Streaming Architecture Toolflow for Accelerating YOLO Models on FPGA Devices. arXiv 2023. In: *arXiv preprint arXiv:2309.01587*:(2023) (pages 1, 11, 12, 16).

[56] A Montgomerie-Corcoran and CS Bouganis. DEF: Differential Encoding of Featuremaps for Low Power Convolutional Neural Network Accelerators. In: *Proceedings of the 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2021, pp. 703–708 (pages 29, 32).

[57] A Montgomerie-Corcoran et al. Pass: Exploiting post-activation sparsity in streaming architectures for cnn acceleration. In: *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE. 2023, pp. 288–293 (pages 9–11, 15).

[58] T Moreau et al. A hardware–software blueprint for flexible deep learning specialization. In: *IEEE Micro* 39(5):(2019), 8–16 (page 10).

[59] MC Mozer and P Smolensky. Skeletonization: A technique for trimming the fat from a network via relevance assessment. In: *Advances in neural information processing systems* 1:(1988) (page 13).

[60] M Nagel et al. A white paper on neural network quantization (2021). In: *URL https://arxiv. org/abs/2106.08295* 3:(2021) (pages 2, 3, 13, 14).

[61] H Nakahara, Z Que, and W Luk. High-Throughput Convolutional Neural Network on an FPGA by Customized JPEG Compression. In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2020, pp. 1–9 (pages 29, 127, 130).

[62] GL Nemhauser, LA Wolsey, and ML Fisher. An Analysis of Approximations for Maximizing Submodular Set Functions—I. In: *Mathematical Programming* 14(1):(1978), 265–294 (page 67).

[63] H Nobach. Pseudo-random generators using linear feedback shift registers with output extraction. In: *arXiv preprint arXiv:2404.12011*:(2024) (pages 65, 128).

[64] I Radosavovic et al. Designing Network Design Spaces. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020, pp. 10428–10436 (pages 9, 39).

[65] B Reagan et al. Weightless: Lossy weight encoding for deep neural network compression. In: *International Conference on Machine Learning*. PMLR. 2018, pp. 4324–4333 (page 28).

[66] M Rizakis et al. Approximate FPGA-based LSTMs under Computation Time Constraints. 2018. arXiv: 1801.02190 (page 129).

[67] M Sandler et al. Mobilenetv2: Inverted residuals and linear bottlenecks. In: *2018 IEEE/CVF conference on computer vision and pattern recognition*. Ieee. 2018, pp. 4510–4520 (pages 3, 35, 107, 120).

[68] M Shafiq and Z Gu. Deep residual learning for image recognition: A survey. In: *Applied sciences* 12(18):(2022), 8972 (page 1).

[69] CE Shannon. A mathematical theory of communication. In: *The Bell system technical journal* 27(3):(1948), 379–423 (pages 16, 19).

[70] Z Shao et al. Memory-efficient CNN accelerator based on interlayer feature map compression. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 69(2):(2021), 668–681 (pages 29, 32).

[71] F Silfa, JM Arnau, and A González. Exploiting Kernel Compression on BNNs. In: *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2023, pp. 1–6 (page 28).

[72] I Sourdis. DAT480 Reconfigurable Computing. Lecture series, Chalmers University of Technology. Course lectures, 2025–2026. 2025 (pages 8, 9).

[73] C Sun et al. Gradient-based automatic mixed precision quantization for neural networks on-chip. In: *arXiv e-prints*:(2024), arXiv–2405 (page 15).

[74] G Sun and SW Jun. ZFP-V: Hardware-Optimized Lossy Floating Point Compression. In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE. 2019, pp. 117–125 (pages 2, 30).

[75] Taalas. The Path to Ubiquitous AI. https://taalas.com/the-path-to-ubiquitous-ai/. https://taalas.com/the-path-to-ubiquitous-ai/ (visited on 10/17/2026) (page 8).

[76] D Tao et al. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2017, pp. 1129–1139 (pages 4, 27, 28, 30, 57, 65, 98, 176).

[77] J Tian et al. cuSZ: An Efficient GPU-Based Error-Bounded Lossy Compression Framework for Scientific Data. In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2020, pp. 3–15 (page 30).

[78] J Tian et al. waveSZ: A Hardware-Algorithm Co-Design of Efficient Lossy Compression for Scientific Data. In: *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2020, pp. 74–88 (page 30).

[79] A Todi et al. Convnext: A contemporary architecture for convolutional neural networks for image classification. In: *2023 3rd International Conference on Innovative Sustainable Computational Technologies (CISCT)*. IEEE. 2023, pp. 1–6 (page 9).

[80] P Toupas et al. SMOF: Streaming Modern CNNs on FPGAs with Smart Off-Chip Eviction. In: *2024 IEEE 32nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2024, pp. 185–196 (pages 1–4, 10–12, 15, 18, 29, 32, 34, 38, 39, 93, 94, 121, 129, 131, 142).

[81] Y Umuroglu et al. Finn: A framework for fast, scalable binarized neural network inference. In: *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays*. 2017, pp. 65–74 (pages 1, 10).

[82] SI Venieris and CS Bouganis. fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs. In: *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2016, pp. 40–47 (pages 1, 2, 8, 10, 40).

[83] SI Venieris, A Kouris, and CS Bouganis. Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions. In: *ACM Computing Surveys (CSUR)* 51(3):(2018), 1–39 (pages 1, 10, 11).

[84] W3C. PNG (Portable Network Graphics) Specification: Filter Algorithms. https://www.w3.org/TR/PNG-Filters.html. Accessed: 2026. 2003 (pages 18, 55).

[85] O Weng. Neural network quantization for efficient inference: A survey. In: *arXiv preprint arXiv:2112.06126*:(2021) (pages 13, 14).

[86] YK Weng, SH Huang, and HY Kao. Block-based compression and corresponding hardware circuits for sparse activations. In: *Sensors* 21(22):(2021), 7468 (pages 29, 35, 128, 143).

[87] S Woo et al. Convnext v2: Co-designing and scaling convnets with masked autoencoders. In: *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Ieee. 2023, pp. 16133–16142 (page 1).

[88] Q Xiong et al. GhostSZ: A Transparent FPGA-Accelerated Lossy Compression Framework. In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2019, pp. 258–266 (pages 2, 30, 32, 77, 132, 141).

[89] J Ye. Generalized low rank approximations of matrices. In: *Machine Learning* 61:(2005), 167–191 (page 23).

[90] X Yu et al. SZx: An Ultra-fast Error-bounded Lossy Compressor for Scientific Datasets. In: *arXiv preprint arXiv:2201.13020*:(2022) (page 30).

[91] Y Yu et al. OPU: An FPGA-based overlay processor for convolutional neural networks. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28(1):(2020), 35–47 (page 10).

[92] Z Yu and CS Bouganis. Mixed-TD: Efficient Neural Network Accelerator with Layer-Specific Tensor Decomposition. In: *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE. 2023, pp. 204–211 (pages 1, 4, 10, 12, 15, 23, 129).

[93] Y Yuan and JE Fowler. QccPack Documentation: QccENTGolombEncodeChannel, Adaptive Golomb Coding. https://qccpack.sourceforge.net/Documentation/QccENTGolombDecodeChannel.3.html. Accessed: 2026. 2021 (page 96).

[94] C Zhang et al. CEAZ: Accelerating Parallel I/O via Hardware-Algorithm Co-Designed Adaptive Lossy Compression. In: *Proceedings of the ACM International Conference on Supercomputing (ICS)*. 2022 (page 30).

[95] J Zhang et al. CompAct: On-chip compression of activations for low power systolic array based CNN acceleration. In: *ACM Transactions on Embedded Computing Systems (TECS)* 18(5s):(2019), 1–24 (pages 18, 29, 32, 97).

[96] K Zhao et al. Significantly improving lossy compression for HPC datasets with second-order prediction and parameter optimization. In: *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. 2020, pp. 89–100 (pages 4, 27, 30, 56, 66).

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY