UNIVERSITY OF CALIFORNIA SAN DIEGO

End-to-End Inference Optimization for Deep Learning-based Image Upsampling Networks

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Electrical Engineering (Machine Learning and Data Science)

by

Ian Colbert

Committee in charge:

Professor Kenneth Kreutz-Delgado, Chair
Professor Alexander Cloninger
Professor Srinjoy Das
Professor Truong Nguyen
Professor Nuno Vasconcelos

2023

The Dissertation of Ian Colbert is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2023

DEDICATION

To my family.

EPIGRAPH

If you don't have time to do it right, when will you have time to do it over?

John Wooden

Achievement is just a moment in pencil unless you can share it with the people you care about.

Scott Galloway

TABLE OF CONTENTS

Dissertation Approval Page iii

Dedication iv

Epigraph v

Table of Contents vi

List of Figures ix

List of Tables xi

List of Algorithms xii

Acknowledgements xiii

Vita xv

Abstract of the Dissertation xvii

Chapter 1 Introduction 1

Chapter 2 Integer-Quantized Neural Networks 6
2.1 Introduction 7
2.2 Related Works 9
2.3 Background 10
2.3.1 Quantization-Aware Training (QAT) 11
2.3.2 Weight Normalization 12
2.4 Motivation 13
2.4.1 Generating Streaming Architectures with FINN 13
2.4.2 Accumulator Impact on Resource Utilization 14
2.5 Accumulator Bit Width Bounds 16
2.5.1 Deriving Lower Bounds Using Data Types 17
2.5.2 Deriving Lower Bounds Using Learned Weights 18
2.6 Training Quantized Neural Networks to Avoid Overflow 19
2.6.1 Constructing Our Quantization Operator 19
2.6.2 Regularization with Lagrangian Penalties 21
2.7 Experiments 22
2.7.1 Image Classification Benchmarks 23
2.7.2 Single-Image Super Resolution Benchmarks 24
2.7.3 Experiment Setup and Research Questions 24
2.7.4 Accumulator Impact on Model Performance 27
2.7.5 Trade-Offs Between Resources and Accuracy 28
2.7.6 Evaluating Resource Savings 29

2.7.7 A Deeper Look at the Impact of Our Constraints 30
2.8 Conclusions and Future Work 31

Chapter 3 Low-Precision Structured Subnetworks 34
3.1 Introduction 35
3.2 Background 38
3.2.1 Pruning 38
3.2.2 Quantization 39
3.3 Motivation 42
3.4 Algorithms 47
3.4.1 Layerwise Channel Pruning using Non-parametric Statistics 47
3.4.2 Uniform Quantization-Aware Training 49
3.5 Experiments 51
3.5.1 Evaluating Pruning Schedules and Importance Measures 52
3.5.2 Evaluation of Joint Pruning and Quantization 54
3.6 Conclusions and Future Work 57

Chapter 4 Unified Quantization and Pruning 60
4.1 Introduction 61
4.2 Discriminative Deep Belief Networks (DDBNs) 64
4.2.1 Discriminative Restricted Boltzmann Machines 64
4.2.2 Discriminative Deep Belief Networks 67
4.2.3 Network Architecture 68
4.3 Limited Precision and Function Approximation 69
4.3.1 Limited Precision Approximation 69
4.3.2 Sigmoid Function Approximation 69
4.4 Neuron Ordering using Criticality Analysis 71
4.5 AX-DBN Design Methodology 73
4.5.1 Cloud-based Model Training and Approximation 73
4.5.2 Inference on Embedded Hardware 74
4.6 Power Model for Compute and Memory Requirements 75
4.7 Experimental Results 76
4.8 Conclusions 77

Chapter 5 Improving Inference Algorithms for Image Upsampling 84
5.1 Introduction 84
5.2 Sub-Pixel Convolution 86
5.3 Resize Convolution 86
5.4 Deconvolution 88
5.4.1 Reverse Looping Deconvolution 89
5.4.2 Fractionally Strided Deconvolution 90
5.4.3 Transforming Deconvolution to Convolution 91
5.4.4 Comparison of Compute and Memory Requirements 92
5.5 Improving Reverse Looping Deconvolution 93

5.6 Conclusions 95

Chapter 6 Kernel Transforms, Layer Fusions, and Time-Energy Analysis 97
6.1 Introduction 98
6.2 Related Works 100
6.3 Deep Learning-based Image Upsampling at the Edge 102
6.4 Kernel Transformations 104
6.4.1 Sub-Pixel Convolution to Deconvolution 105
6.4.2 Resize Convolution to Deconvolution 107
6.5 Time and Energy Analysis at Inference 109
6.5.1 Quantitative Models for Time and Energy 110
6.5.2 Estimating Energy Efficiency by Data Reuse 114
6.5.3 Roofline Models of Time and Energy 115
6.6 Results of Quantitative Analysis 116
6.7 Conclusions and Future Work 120

Chapter 7 Hardware Acceleration for Deconvolution Inference 126
7.1 Introduction 127
7.2 Related Works 128
7.3 Deconvolution Algorithm 129
7.4 FPGA Hardware Architecture 132
7.5 Experimental Results 133
7.5.1 Design Space Exploration 134
7.5.2 Performance-per-Watt Comparison with Edge GPU 135
7.5.3 Sparsity Experiments 136
7.6 Conclusions and Future Work 137

Chapter 8 Conclusion 139

Bibliography 141

LIST OF FIGURES

Figure 1.1. End-to-end inference optimization pipeline 4

Figure 2.1. Illustration of fixed-point arithmetic in QNN inference 9

Figure 2.2. Overview of the FINN framework 12

Figure 2.3. Accumulator bit width impact on LUT utilization 13

Figure 2.4. Visualizing our derived accumulator bit width bounds 19

Figure 2.5. Trade-offs between accumulator bit width and model quality 22

Figure 2.6. Trade-offs between accumulator bit width and LUT utilization 25

Figure 2.7. LUT utilization break down across Pareto fronts 27

Figure 2.8. Visualizing impact of ℓ_1-norm constraints on sparsity 31

Figure 3.1. Joint layerwise channel pruning and uniform quantization 37

Figure 3.2. Structured and unstructured pruning 40

Figure 3.3. Visualizing images generated from VAE 44

Figure 3.4. Visualizing impact of input activation sparsity on importance metrics 46

Figure 3.5. Stepwise and layerwise pruning schedules 50

Figure 3.6. Visualizing images generated from quantized and pruned CycleGAN 52

Figure 4.1. Illustration of edge computing paradigm for deep learning 62

Figure 4.2. Discriminative restricted Boltzmann machine 64

Figure 4.3. Discriminative deep belief network 68

Figure 4.4. Approximating deep belief networks 70

Figure 4.5. Illustration of our proposed approximate computing framework 73

Figure 4.6. Proposed hardware inference path for deep belief networks 76

Figure 4.7. Visualizing results for DRBM-300 80

Figure 4.8. Visualizing results for DDBN-100-200 81

Figure 4.9. Visualizing results for DRBM-600 82
Figure 4.10. Visualizing results for DDBN-100-200-300 83
Figure 5.1. The image upsampling taxonomy 86
Figure 5.2. Visualizing the sub-pixel convolution 87
Figure 5.3. Visualizing the nearest neighbor resize convolution 88
Figure 5.4. Visualizing the standard deconvolution 89
Figure 5.5. Visualizing the fractionally strided deconvolution 91
Figure 5.6. Visualizing overheads for variant deconvolution algorithms 93
Figure 6.1. Deploying image upsampling networks for inference at the edge 99
Figure 6.2. Edge computing paradigm for image upsampling 102
Figure 6.3. An example use case for kernel transformations 104
Figure 6.4. Visualizing our weight shuffle algorithm 107
Figure 6.5. Visualizing our weight convolution algorithm 110
Figure 6.6. Time and energy costs of deconvolution variants 122
Figure 6.7. Time and energy costs of transformed upsampling algorithms 123
Figure 6.8. Activation reuse of transformed upsampling algorithms 123
Figure 6.9. Roofline models of time and energy 124
Figure 6.10. Visualizing algorithm scalability 125
Figure 7.1. Generative adversarial network architecture 128
Figure 7.2. Reverse looping deconvolution algorithm 129
Figure 7.3. Exploiting deconvolution dataflow with fine-grained parallelism 132
Figure 7.4. Example network architectures for inference acceleration 133
Figure 7.5. Design space exploration 134
Figure 7.6. Trade-offs between generative quality and hardware performance 136

LIST OF TABLES

Table 3.1. Comparing pruning schedules 55

Table 3.2. Comparing neuron importance metrics 55

Table 3.3. Results from joint pruning and quantization 56

Table 3.4. Evaluating performance-per-memory footprint 58

Table 4.1. Results for activation function approximation 70

Table 4.2. Uniform bit width reduction results 79

Table 4.3. Mixed-precision results with 1% accuracy loss constraint 79

Table 4.4. Mixed-precision results with 5% accuracy loss constraint 80

Table 5.1. Compute and memory requirements of variant deconvolution algorithms . . 92

Table 6.1. Capabilities of assumed hardware target 112

Table 6.2. Compute and memory requirements of transformed upsampling algorithms 114

Table 7.1. Resource utilization 135

Table 7.2. Evaluating throughput-to-power ratio 135

LIST OF ALGORITHMS

Algorithm 1. Layerwise channel pruning 48

Algorithm 2. Adaptive asymmetric activation quantization 50

Algorithm 3. Adaptive symmetric weight quantization 51

Algorithm 4. AX-DBN approximation algorithm 75

Algorithm 5. Standard convolution 87

Algorithm 6. Pixel shuffle 88

Algorithm 7. Standard deconvolution 89

Algorithm 8. Reverse looping deconvolution (REVD) 90

Algorithm 9. Transforming deconvolution to convolution (weight transform) 92

Algorithm 10. Improved reverse looping deconvolution (REVD2) 95

Algorithm 11. Weight shuffle 106

Algorithm 12. Weight convolution 109

Algorithm 13. Reverse looping deconvolution dataflow 130

ACKNOWLEDGEMENTS

I would like to first acknowledge Professor Srinjoy Das, who has been a mentor, colleague, and friend throughout the entirety of my academic journey. I would like to acknowledge Professor Kenneth Kreutz-Delgado whose support, guidance, and wisdom have helped shape many of the key ideas in this work as well as my overall approach to research. I am honored to also have Professor Alex Cloninger, Professor Truong Nguyen, and Professor Nuno Vasconcelos serve on my doctoral committee. Their feedback and advice have been valuable assets.

I am grateful to the many colleagues I have worked beside at the University of California San Diego, and Advanced Micro Devices, Inc. This endeavor would not have been possible without their constructive discourse, late-night feedback sessions, and moral support. Lastly, I would be remiss in not mentioning my family and loved ones. Their constant support and unyielding belief have helped me in an immeasurable way.

Although this dissertation lists only one author, in reality, the ideas are a collection of contributions synthesized and refined through collaboration with many insightful colleagues.

Chapter 2 is based on unpublished material (Ian Colbert, Alessandro Pappalardo, and Jakoba Petri-Koenig, "Quantized Neural Networks for Low-Precision Accumulation with Guaranteed Overflow Avoidance"). The dissertation author was the primary investigator and author of this paper.

Chapter 3 is based on material as it appears in the 2022 MDPI Applied Sciences Special Issue on Hardware-Aware Deep Learning (Xinyu Zhang, Ian Colbert, and Srinjoy Das, "Learning Low-Precision Structured Subnetworks Using Joint Layerwise Channel Pruning and Uniform Quantization"). The dissertation author was a co-primary investigator and author of this paper.

Chapter 4 is based on material as it appears in the 2019 International Joint Conference on Neural Networks (Ian Colbert, Kenneth Kreutz-Delgado, and Srinjoy Das, "AX-DBN: An Approximate Computing Framework for the Design of Low-Power Discriminative Deep Belief Networks"). The dissertation author was the primary investigator and author of this paper.

Chapters 5 and 6 are based on material as it appears in the 2021 IEEE Access Journal

(Ian Colbert, Kenneth Kreutz-Delgado, and Srinjoy Das, “An Energy-Efficient Edge Computing Paradigm for Convolution-Based Image Upsampling”). The dissertation author was the primary investigator and author of this paper.

Chapter 7 is based on unpublished material (Ian Colbert, Jake Daly, Kenneth Kreutz-Delgado, and Srinjoy Das, “A Competitive Edge: Can FPGAs Beat GPUs at DCNN Inference Acceleration in Resource-Limited Edge Computing Applications?”). The dissertation author is the primary investigator and author of this paper.

VITA

2017	Bachelor of Science, University of California San Diego
2018–Present	Software Engineer, Advanced Micro Devices, Inc.
2019	Master of Science, University of California San Diego
2023	Doctor of Philosophy, University of California San Diego

PUBLICATIONS

Ian Colbert, Kenneth Kreutz-Delgado, and Srinjoy Das. "AX-DBN: An Approximate Computing Framework for the Design of Low-Power Discriminative Deep Belief Networks," 2019 International Joint Conference on Neural Networks, pp. 1-9.

Alexander Potapov, **Ian Colbert**, Kenneth Kreutz-Delgado, Alexander Cloninger, and Srinjoy Das. "PT-MMD: A Novel Statistical Framework for the Evaluation of Generative Systems," 2019 53rd Asilomar Conference on Signals, Systems, and Computers, pp. 2219-2223.

Ian Colbert, Jake Daly, Kenneth Kreutz-Delgado, and Srinjoy Das. "A Competitive Edge: Can FPGAs Beat GPUs at DCNN Inference Acceleration in Resource-Constrained Edge Computing Applications?" arXiv preprint:2102.00294, 2021.

Siqiao Ruan, **Ian Colbert**, Ken Kreutz-Delgado, Srinjoy Das. "Generative and Discriminative Deep Belief Network Classifiers: Comparisons Under an Approximate Computing Framework," arXiv preprint:2102.00534, 2021.

Ian Colbert, Kenneth Kreutz-Delgado, and Srinjoy Das. "An Energy-Efficient Edge Computing Paradigm for Convolution-Based Image Upsampling," in IEEE Access, vol. 9, pp. 147967-147984, 2021.

Ian Colbert, Jake Daly, Norm Rubin. "Generating GPU Compiler Heuristics Using Reinforcement Learning," arXiv preprint:2111.12055, 2021.

Xinyu Zhang, **Ian Colbert**, Kenneth Kreutz-Delgado. "Learning Low-Precision Structured Subnetworks Using Joint Layerwise Channel Pruning and Uniform Quantization," Applied Sciences. 2022; 12(15):7829.

Ian Colbert and Mehdi Saeedi. "Evaluating Navigation Behavior of Agents in Games using Non-Parametric Statistics," 2022 IEEE Conference on Games, pp. 544-547.

Alexander Cann, **Ian Colbert,** Ihab Amer. "Robust Transferable Feature Extractors: Learning to Defend Pre-Trained Networks Against White Box Adversaries," arXiv preprint:2209.06931, 2022.

Ian Colbert, Alessandro Pappalardo, Jakoba Petri-Koenig. "Quantized Neural Networks for Low-Precision Accumulation with Guaranteed Overflow Avoidance," arXiv preprint:2301.13376, 2023.

ABSTRACT OF THE DISSERTATION

End-to-End Inference Optimization for Deep Learning-based Image Upsampling Networks

by

Ian Colbert

Doctor of Philosophy in Electrical Engineering (Machine Learning and Data Science)

University of California San Diego, 2023

Professor Kenneth Kreutz-Delgado, Chair

Many computer vision problems require image upsampling, where the number of pixels per unit area is increased by inferring values in high-dimensional image space from low-dimensional representations. Recent research has shown that deep learning-based solutions achieve state-of-the-art performance on such tasks by training deep neural networks (DNNs) on large annotated datasets. Yet, their adoption in real-time applications is predicated on the deployment costs of the resulting models since end-user devices impose significant compute and memory constraints on inference pipelines. To address this, many researchers and practitioners

have proposed methods to reduce inference costs without sacrificing model quality. However, many of these works focus on DNNs designed for image downsampling. In this thesis, we study inference optimization techniques designed for deep learning-based image upsampling networks. While some inference optimizations are applicable to both upsampling and downsampling networks, we show that specifically tailoring optimizations for image upsampling workloads can lead to more efficient and effective deployment.

We maintain a holistic view of inference optimization, from training through deployment to execution, by integrating hardware-aware deep learning techniques, compute graph transformations, and computer architecture optimizations into an end-to-end pipeline. We begin by characterizing this pipeline and the different requirements for image upsampling and downsampling workloads. We then introduce novel statistical approaches to hardware-aware deep learning techniques based on quantization and pruning. Once trained, we then introduce novel compute kernels and graph transformations that reduce the compute costs of common upsampling workloads by up to a factor of 3.3. Finally, we adapt our novel inference algorithms to a specialized hardware architecture that reduces resource utilization and improves dataflow on FPGA-based accelerators.

We evaluate a wide range of computer vision benchmarks covering both stochastic and deterministic models to show that our approaches improve power efficiency, throughput, and resource utilization without damaging model quality. Our research highlights the importance of end-to-end inference optimization for deep learning-based image upsampling networks and provides an effective solution for reducing the deployment costs of DNNs designed for real-time computer vision applications on resource-constrained platforms.

Chapter 1

Introduction

Since the success of deep convolutional neural networks in the 2012 ILSVRC Computer Vision Challenge [94, 99, 138], deep learning has become the primary approach to developing state-of-the-art solutions for many prominent computer vision problems (*e.g.*, image classification [66, 77], scene segmentation [52, 106], and object detection [65, 133]). Over the past decade, numerous architectural variations have been proposed, each with unique benefits and capabilities [66, 77, 89, 134, 136, 144]. As the availability of both data and computation has increased, both the size and quality of these models have scaled [69], resulting in widespread adoption across real-world applications including smart cities [12], medical imaging [49], and industrial robotics [172].

Many emerging computer vision applications, however, require real-time inference (*e.g.*, autonomous driver assistance systems [50, 146] and high-fidelity rendering [88, 149]). Thus, deep learning models are increasingly being moved closer to their data sources to run inference directly on end-user devices at the edge [44]. This migration reduces strain on network bandwidth and decreases system latency by removing any reliance on internet connectivity. However, it also imposes significant constraints on the inference pipeline since edge devices have form-factor and cost considerations that limit power budgets, compute capabilities, and memory storage [33, 44]. These constraints make it challenging to deploy state-of-the-art models for real-time inference on edge devices as the compute and memory requirements of large models often exceed the

capabilities of such platforms [53, 63].

To address the rising inference costs, researchers and practitioners have introduced hardware, software, and algorithmic optimizations in the pursuit of minimizing the trade-offs between hardware performance and model quality [2, 35, 54, 63, 75, 81, 176, 179, 183, 191]. The resulting body of work surrounding inference optimization research has rapidly expanded to cover many modalities ranging from images [33, 81, 149] to natural language [55, 165, 179], all with a shared goal of minimizing hardware costs while maintaining model quality. We have focused our attention on computer vision applications, where we have observed inference optimization research to be heavily skewed towards image downsampling workloads [2, 35, 61, 62, 75, 86, 97, 111, 117], leaving the optimization of image upsampling workloads an open challenge approached by few [23, 33, 161, 177].

Many recent state-of-the-art neural architectures have increasingly employed image upsampling [74, 89, 134, 136, 144], where the number of pixels per unit area (*i.e.*, resolution) is increased by inferring values in high-dimensional image space from low-dimensional representations. This class of architectures, which we refer to as *deep learning-based image upsampling networks*, is now commonly used to train models for tasks such as super resolution [144, 149], image generation [57, 74, 89], and style transfer [134, 190]. In contrast to downsampling, which is a many-to-one mapping used to extract or encode high-level features from an image by reducing dimensionality, upsampling is a one-to-many mapping used to infer or recover information in an image by increasing dimensionality. The differences in these workloads present unique and interesting inference optimization problems. While some inference optimization techniques are applicable to both upsampling and downsampling workloads, we show that specifically tailoring optimizations for image upsampling workloads can lead to more efficient and effective deployment of deep learning-based image upsampling networks.

To demonstrate one aspect of these differences, let us compare the weight and activation requirements of common downsampling and upsampling networks assuming 32-bit floating-point implementations. When trained on CIFAR10 [93], LeNet5 [100] requires roughly 248 KB for

weights and 38 KB for activations. Conversely, when trained for Set5 [14], ESPCN [144] requires roughly 239 KB for weights and 7 MB for activations, nearly 180x more activations than LeNet5 with 4% fewer weights. Let us additionally consider large-scale models. When trained for ImageNet [42], ResNet50 [66] requires roughly 102 MB for weights and 31 MB for activations. Conversely, when trained on DIV2K [5], U-Net [136] requires roughly 45 MB for weights and 1.4 GB for activations, nearly 46x more activations than ResNet50 with 56% fewer weights. This is in large part because image upsampling applications often have higher dimensional inputs when compared to their downsampling counterparts, which is only exacerbated by the expanding output fields [33]. Such differences often render image upsampling workloads memory-bound, where optimizations that improve dataflow and reduce memory requirements can yield substantial gains [33].

We study inference optimization techniques designed for deep learning-based image upsampling networks. Since their benefits often depend on the characteristics of the deep learning workload and the capabilities of the target hardware, maximally exploiting inference optimizations often requires aggressive hardware-software (HW-SW) co-design. For example, unstructured weight pruning can offer high compression rates with minimal accuracy degradation due to its inherent topological flexibility [51, 63]; however, such flexibility results in irregular sparsity patterns that lead to poor data locality and can result in low hardware efficiency if not exploited by sparse inference kernels [54] or specialized hardware accelerators [23, 31, 122]. In an effort to fully exploit our contributions, we maintain a holistic view of inference optimization, integrating hardware-aware deep learning techniques, compute graph transformations, and specialized hardware accelerators into an end-to-end pipeline.

We visualize our inference optimization pipeline in Fig. 1.1, where we split the process into three stages: training, deployment, and inference. We first use hardware-aware deep learning techniques based on quantization and pruning to train our models to take advantage of the hardware characteristics of a specific platform (described in Chapters 2, 3, and 4). While we consider both general-purpose platforms and programmable hardware, we primarily study

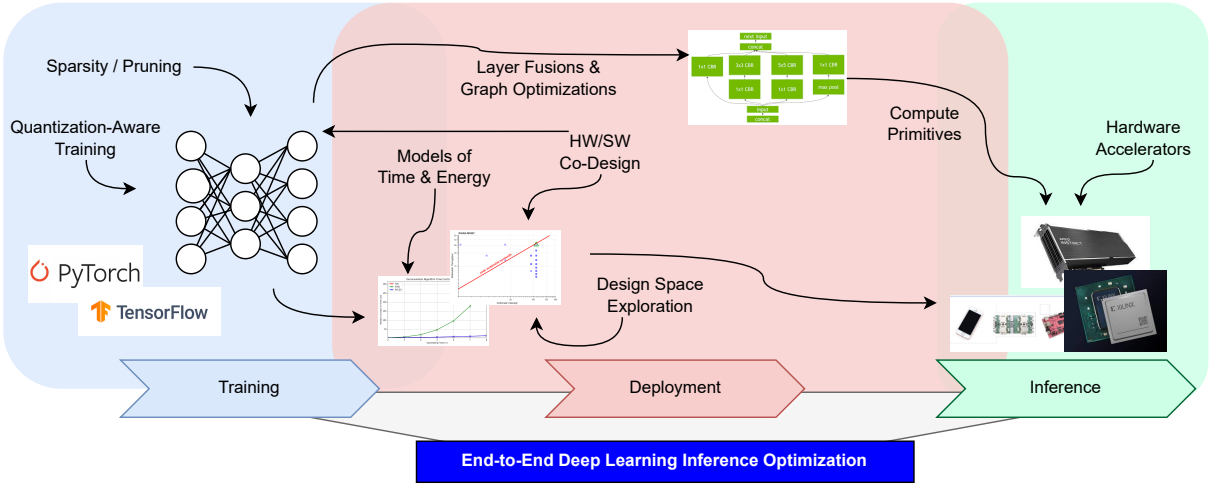


Figure 1.1. A high-level flow diagram of our end-to-end inference optimization pipeline.

deployment on FPGA-based accelerators as they can take advantage of fine-grained parallelism and custom data types to a greater extent [34]. Given the trained model, we introduce novel graph transformations and compute kernels (described in Chapters 5 and 6, respectively) that reduce the compute costs of common image upsampling workloads by up to a factor of 3.3. We then adapt our novel image upsampling algorithm to an FPGA-based accelerator (described in Chapter 7). Finally, we conclude with additional deployment considerations as well as implications for broader impact. The contributions of our work are summarized as follows:

1. In Chapter 2, we introduce a novel quantization-aware training algorithm that constrains an integer-quantized neural network to use a low-precision accumulator without numerical overflow while inherently increasing the sparsity of the resulting weights.

2. In Chapter 3, we introduce a data-driven structured weight pruning algorithm that uses non-parametric measures of importance to greedily remove redundant channels in quantized neural networks trained for computer vision applications.

3. In Chapter 4, we introduce a systematic approximation framework that optimizes the power consumption of stochastic neural networks using a unified pruning and quantization formulation guided by surrogate measures of neuron criticality.

4. In Chapter 5, we introduce a novel deconvolution inference algorithm that exposes more opportunities for concurrent execution to improve adaptability to resource-constrained settings.

5. In Chapter 6, we introduce novel graph transformations that can significantly reduce the data movement costs of common image upsampling workloads.

6. In Chapter 7, we adapt our novel compute kernels to a specialized hardware architecture that reduces resource utilization and improves dataflow on an FPGA-based accelerator.

Chapter 2

Integer-Quantized Neural Networks

Quantizing the weights and activations of neural networks significantly reduces their inference costs, often in exchange for minor reductions in model accuracy. This is in large part due to compute and memory cost savings in operations like convolutions and matrix multiplications, whose resulting products are typically accumulated into high-precision registers, referred to as accumulators. While many researchers and practitioners have taken to leveraging low-precision representations for the weights and activations of a model, few have focused attention on reducing the size of accumulators. Part of the issue is that accumulating into low-precision registers introduces a high risk of numerical overflow which, due to wraparound arithmetic, can significantly degrade model accuracy. In this chapter, we introduce a quantization-aware training algorithm that guarantees avoiding numerical overflow when reducing the precision of accumulators during inference. We leverage weight normalization as a means of constraining parameters during training using accumulator bit width bounds that we derive. We evaluate our algorithm across multiple quantized models that we train for different tasks, showing that our approach can reduce the precision of accumulators while maintaining model accuracy with respect to a floating-point baseline. We then show that this reduction translates to increased design efficiency for custom FPGA-based accelerators. Finally, we show that our algorithm not only constrains weights to fit into an accumulator of user-defined bit width but also increases the sparsity and compressibility of the resulting weights. Across all of our benchmark models trained

with 8-bit weights and activations, we observe that constraining the hidden layers of quantized neural networks to fit into 16-bit accumulators yields an average 98.2% sparsity with an estimated compression rate of 46.5x all while maintaining 99.2% of the floating-point performance.

2.1 Introduction

Quantization is the process of reducing the range and precision of the numerical representation of data. Among the many techniques used to reduce the inference costs of neural networks (NNs), integer quantization is one of the most widely applied in practice [56]. The reduction in compute and memory requirements resulting from low-precision quantization provides increased throughput, power savings, and resource efficiency, usually in exchange for minor reductions in model accuracy [80]. During inference, information is propagated through the layers of an NN, where most of the compute workload is concentrated in the multiply-and-accumulates (MACs) of operators such as convolutions and matrix multiplications. It has been shown that reducing the bit width of the accumulator can increase throughput and bandwidth efficiency for general-purpose processors by creating more opportunities to increase parallelism [40, 121, 170]. However, exploiting such an optimization is non-trivial, as doing so incurs a high risk of overflow which can introduce numerical errors that significantly degrade model accuracy due to wraparound twos-complement arithmetic [121].

Previous work has sought to either reduce the risk of numerical overflow [139, 170] or mitigate its impact on model accuracy [121]. In this work, we train quantized NNs (QNNs) to avoid numerical overflow altogether when using low-precision accumulators during inference. To fully exploit the wider design space exposed by considering low-precision weights, activations, and accumulators, we target model deployment on FPGA accelerators with custom spatial streaming dataflow rather than general-purposes platforms like CPUs or GPUs. The flexibility of FPGAs makes them ideal devices for low-precision inference engines as they allow for bit-level control over every part of a network; the precision of weights, activations, and accumulators can

be individually tuned to custom data types for each layer without being restricted to power-of-2 bit widths like a CPU or a GPU would be. The contributions of our work are summarized as follows:

- We show that reducing the bit width of the accumulator can reduce the resource utilization of custom low-precision QNN inference accelerators.
- We derive comprehensive bounds on accumulator bit widths with finer granularity than existing literature.
- We introduce a novel quantization-aware training (QAT) algorithm that constrains learned parameters to avoid numerical overflow when reducing the precision of accumulators during inference.
- We show that our algorithm not only constrains weights to fit into an accumulator of user-defined bit width, but also significantly increases the sparsity and compressibility of the resulting weights.
- We integrate our algorithm into the Brevitas quantization library [126] and the FINN compiler [7] to demonstrate an end-to-end flow for training and deploying QNNs using low-precision accumulators with custom streaming architectures on AMD-Xilinx FPGAs.

To the best of our knowledge, we are the first to explore the use of low-precision accumulators to improve the design efficiency of programmable QNN inference accelerators. However, our results have implications outside of the accelerators generated by FINN. Constraining the accumulator bit width to a user-defined upper bound has been shown to increase throughput and bandwidth efficiency on general-purpose processors [40, 121, 170] and reduce the compute overhead of homomorphic encryption arithmetic [107]. Furthermore, our experiments show that our algorithm can offer a better trade-off between resource utilization and model accuracy than existing approaches, confirming the benefit of including the accumulator bit width in the overall hardware-software (HW) co-design space.

2.2 Related Works

As activations propagate through the layers of a QNN, the intermediate partial sums resulting from convolutions and matrix multiplications are typically accumulated in a high-precision register before being requantized and passed to the next layer, which we depict in Figure 2.1. While many researchers and practitioners have taken to leveraging reduced precision representations for weights and activations [56, 81, 119, 183], few works have focused attention on reduced precision accumulators [40, 121, 139, 170].

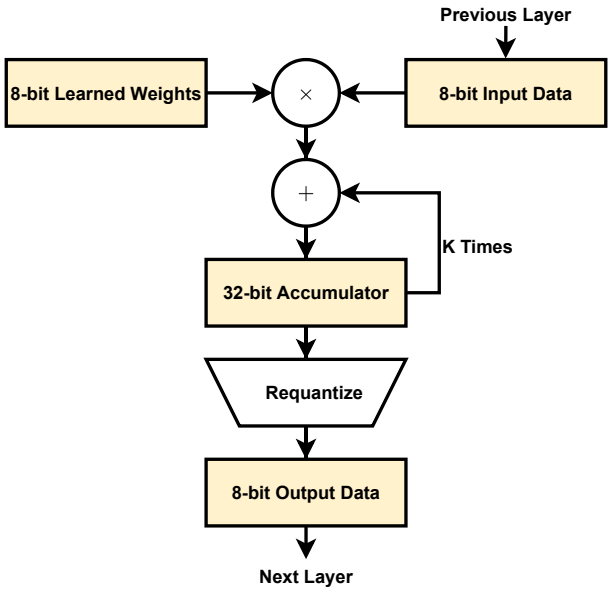


Figure 2.1. A simplified illustration of fixed-point arithmetic in neural network inference. Quantized weights are frozen during inference. Input/output data is dynamic and thus, scaled then clipped as the hidden representations (*i.e.*, activations) are passed through the network. The accumulator needs to be big enough to fit the dot product of the learned weights with input data vectors, which are assumed to both be K-dimensional.

One approach to training QNNs to use low-precision accumulators is to mitigate the impact of overflow on model accuracy. Xie *et al.* [170] sought to reduce the risk of overflow using an adaptive scaling factor tuned during training; however, their approach relies on distributional assumptions that cannot guarantee overflow avoidance during inference. Alternatively, Ni *et al.* [121] proposed training QNNs to be robust to overflow using a cyclic activation function based

on expensive modulo arithmetic. They also use a regularization penalty to control the number of overflows. In both approaches, overflow is accounted for at the outer-most level, which fails to consider possible overflow when accumulating intermediate partial sums. Moreover, modeling overflow at the inner-most accumulation level during QAT is not easily supported by off-the-shelf deep learning frameworks as it is not directly compatible with fake-quantization over pre-existing floating-point backends. As such, the current practice is to either use high-precision registers or simply saturate values as they are accumulated; however, such clipping can still: (1) introduce errors that cascade when propagated through a QNN; and (2) require saturation logic, which can break associativity and add to latency and area requirements [8]. Thus, in our work, we train QNNs to completely avoid overflow rather than simply reducing its impact on model accuracy.

Most similar to our work is that of [40], which proposed an iterative layer-wise optimization strategy to select mixed-precision bit widths to avoid overflow using computationally expensive heuristics that assume signed bit widths for all input data types. Our proposed method constrains weights to avoid numerical overflow through the construction of our weight normalization-based quantization formulation, which accounts for both signed and unsigned input data types while adding negligible training overhead.

Tangential to our work, Wang *et al.* [160] and Sakr *et al.* [139] study the impact of reduced precision floating-point accumulators for the purpose of accelerating training. Such methods do not directly translate to fixed-point arithmetic, which is the focus of this work.

2.3 Background

Our work explores the use of weight normalization as a means of constraining weights during quantization-aware training (QAT) for the purpose of avoiding overflow when using low-precision accumulators. Here, we provide background related to this objective.

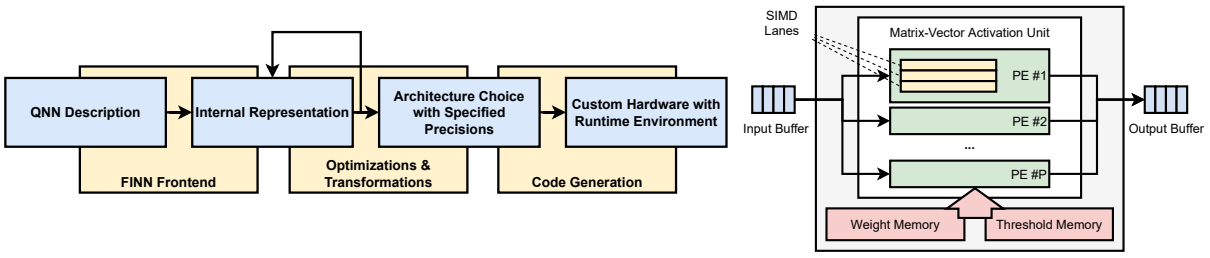
2.3.1 Quantization-Aware Training (QAT)

The standard operators used to emulate quantization during training rely on uniform affine mappings from a high-precision real number to a low-precision quantized number, allowing for the core computations to use integer-only arithmetic [81]. The quantizer (Eq. 2.1) and dequantizer (Eq. 2.2) are parameterized by zero-point z and scaling factor s. Here, z is an integer value that maps to the real zero such that the real zero is exactly represented in the quantized domain, and s is a strictly positive real scalar that corresponds to the resolution of the quantization function. Scaled values are rounded to the nearest integers using half-way rounding, denoted by $\lfloor \cdot \rceil$, and elements that exceed the largest supported values in the quantized domain are clipped: $\text{clip}(x;n,p) = \min(\max(x;n);p)$, where n and p are dependent on the data type of x. For signed integers of bit width b, we assume $n = -2^{b-1}$ and $p = 2^{b-1} - 1$ and assume $n = 0$ and $p = 2^b - 1$ when unsigned.

$$\text{quantize}(x;s,z) := \text{clip}(\left\lfloor \frac{x}{s} \right\rceil + z; n, p) \tag{2.1}$$

$$\text{dequantize}(x;s,z) := s \cdot (x - z) \tag{2.2}$$

It has become increasingly common to use unique scaling factors for each of the output channels of the learned weights to adjust for varied dynamic ranges [118]. However, extending this strategy to the activations incurs additional overhead as it requires either storing partial sums or introducing additional control logic. As such, it is standard practice to use per-tensor scaling factors for activations and per-channel scaling factors on only the weights. It is also common to constrain the weight quantization scheme such that $z = 0$, which is referred to as symmetric quantization. Eliminating these zero points reduces the computational overhead of cross-terms when executing inference using integer-only arithmetic [83]. During training, the straight-through estimator (STE) [13] is used to allow local gradients to permeate the rounding



(a) FINN Framework Overview **(b)** MVAU

Figure 2.2. We adapt images from [18, 153] to provide: (a) an overview of the FINN framework; and (b) an abstraction of the matrix-vector-activation unit (MVAU), which is one of the primary building blocks used by the FINN compiler to generate custom streaming architectures.

function such that $\nabla_x \lfloor x \rceil = 1$ everywhere, where ∇_x denotes the local gradient with respect to x.

2.3.2 Weight Normalization

Weight normalization reparameterizes each weight vector $\boldsymbol{w}$ in terms of a parameter vector $\boldsymbol{v}$ and a scalar parameter g as given in Eq. 2.3, where $\|\boldsymbol{v}\|_2$ is the Euclidean norm of the K-dimensional vector $\boldsymbol{v}$ [141]. This simple reparameterization fixes the Euclidean norm of weight vector $\boldsymbol{w}$ such that $\|\boldsymbol{w}\|_2 = g$, which enables the magnitude and direction to be independently learned.

$$\boldsymbol{w} = g \cdot \frac{\boldsymbol{v}}{\|\boldsymbol{v}\|_2} \tag{2.3}$$

Tangential to our work, prior research has sought to leverage weight normalization as a means of regularizing long-tail weight distributions during QAT [20]. They replace the standard ℓ_2-norm with an ℓ_∞-norm and derive a projection operator to map real values into the quantized domain. As further detailed in Section 2.6, we replace the ℓ_2-norm with an ℓ_1-norm to use the weight normalization parameterization as a means of constraining learned weights during training to use a pre-defined accumulator bit width during inference.

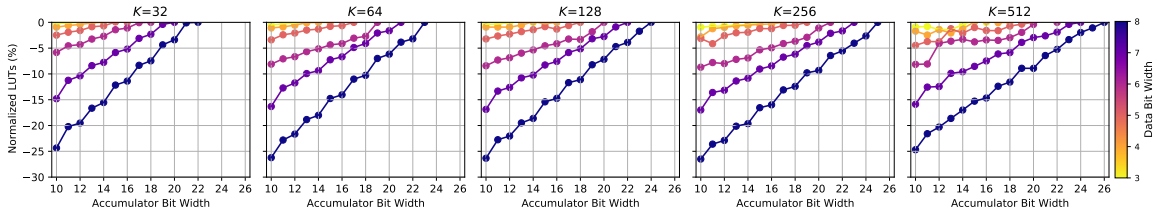


Figure 2.3. Reducing the size of the accumulator in turn reduces LUT utilization as we vary the size of the dot product (K) and the input and weights bit widths, N and M respectively. For simplicity, we use the same bit width for the weights and activations such that $N = M$ for all data points and jointly refer to them as "data bit width." For a given dot product size and data bit width, we normalize the LUT utilization to the largest lower bound on the accumulator bit width as determined by the data types of the inputs and weights.

2.4 Motivation

To motivate our research objective, we evaluate the impact of accumulator bit width on the resource utilization of custom FPGA accelerators with spatial dataflow architectures. To do so, we adopt FINN [18, 153], an open-source framework designed to generate specialized streaming architectures for QNN inference acceleration on AMD-Xilinx FPGAs.

2.4.1 Generating Streaming Architectures with FINN

The FINN framework, depicted in Figure 2.2a, generates specialized QNN accelerators for AMD-Xilinx FPGAs using spatial streaming dataflow architectures that are individually customized for the network topology and the data types used. At the core of FINN is its compiler, which empowers flexible hardware-software (HW-SW) co-design by allowing a user to have per-layer control over the generated accelerator. Weight and activation precisions can be individually specified for each layer in a QNN, and each layer is instantiated as its own dedicated compute unit (CU) that can be independently optimized with fine-grained parallelism.

As an example of how a layer is instantiated as its own CU, we provide a simplified abstraction of the matrix-vector-activation unit (MVAU) in Figure 2.2b. The MVAU is one of the primary building blocks used by the FINN compiler for linear and convolutional layers [18]. Each CU consists of processing elements (PEs), which parallelize work along the data-independent

output dimension, and single-instruction multiple-data (SIMD) lanes, which parallelize work along the data-dependent input dimension. Execution over SIMDs and PEs within a layer is concurrent (*i.e.*, spatial parallelism), while execution over layers within a network is pipelined (*i.e.*, temporal parallelism). All quantized monotonic activation functions in the network are implemented as threshold comparisons that map high-precision accumulated results from the preceding layer into low-precision output values. During compilation, batch normalization, biases, and even scaling factors are absorbed into this threshold logic via mathematical manipulation [154]. The input and output data for the generated accelerators are streamed into and out of the chip using AXI-Stream protocols while on-chip data streams are used to interconnect these CUs to propagate intermediate activations through the layers of the network. During inference, all network parameters are stored on-chip to avoid external memory bottlenecks. For more information on the FINN framework, we refer the interested reader to [7, 18, 153].

2.4.2 Accumulator Impact on Resource Utilization

FINN typically relies on look-up tables (LUTs) to perform MACs at low precision; in such scenarios, LUTs are often the resource bottleneck for the low-precision streaming accelerators it generates. Furthermore, because activation functions are implemented as threshold comparisons, their resource utilization exponentially grows with the precision of the accumulator and output activations [18]. Thus, reducing the size of the accumulator has a direct influence on both the compute and memory requirements.

To evaluate the impact of accumulator bit width on LUT utilization, we consider a fully connected QNN with one hidden layer that is parameterized by a matrix of signed integers. The QNN takes as input a K-dimensional vector of unsigned integers and gives as output a 10-dimensional vector of signed integers. We use the FINN compiler to generate a streaming architecture with a single MVAU targeting an AMD-Xilinx PYNQ-Z2 board with a frequency of 100 MHz. We report the resource utilization of the resulting RTL post-synthesis. To simplify our analysis, we assume that LUTs are the only type of resources available and configure the FINN

compiler to target LUTs for both compute and memory so that we can evaluate the impact of accumulator bit width on resource utilization using just one resource.

As further discussed in Section 2.5, the minimum accumulator bit width that can be used to avoid overflow is a function of the size of the dot product (K) as well as the bit widths of the input and weight vectors $\boldsymbol{x}$ and $\boldsymbol{w}$, respectively denoted as N and M. In Figure 2.3, we visualize how further reducing the accumulator bit width in turn decreases resource utilization as we vary K, N, and M. For a given dot product size and data bit width, we normalize the LUT utilization to the largest lower bound on the accumulator bit width as determined by the data types of the inputs and weights. To control for the resource utilization of dataflow logic, we use a single PE without applying optimizations such as loop unrolling, which increase the amount of SIMD lanes.

We observe that the impact of accumulator bit width on resource utilization grows exponentially with the precision of the data (*i.e.*, M and N). As we reduce the size of the accumulator, we observe up to a 25% reduction in the LUT utilization of a layer when $N = M = 8$, but only up to a 1% reduction in LUT utilization when $N = M = 3$. This is expected as compute and memory requirements exponentially grow with precision and thus have larger proportional savings opportunities. We also observe that K has a dampening effect on the impact of accumulator bit width reductions that is also proportional to the precision of the data. When $N = M = 8$ and $K = 32$, we observe on average a 2.1% LUT reduction for every bit that we reduce the accumulator, but only a 1.5% LUT reduction when $K = 512$. Conversely, we observe on average a 0.2% LUT reduction for every bit that we reduce the accumulator when $N = M = 3$ regardless of K. We hypothesize that this dampening effect is in part due to the increased storage costs of larger weight matrices because, unlike threshold storage, the memory requirements of weights are not directly impacted by the accumulator bit width. Therefore, the relative savings from accumulator bit width reductions are diluted by the constant memory requirements of the weights. We explore this further in Section 3.5, where we break down the resource utilization of FPGA accelerators generated for our benchmark models.

2.5 Accumulator Bit Width Bounds

Figure 2.1 illustrates a simplified abstraction of accumulation in QNN inference. As activations are propagated through the layers, the intermediate partial sums resulting from operations such as convolutions or matrix multiplications are accumulated into a register before being requantized and passed to the next layer. To avoid numerical overflow, the register storing these accumulated values needs to be wide enough to not only store the result of the dot product, but also all intermediate partial sums.

Consider the dot product of input data $\boldsymbol{x}$ and learned weights $\boldsymbol{w}$, which are each K-dimensional vectors of integers. Let y be the scalar result of their dot product given by Eq. 2.4, where x_i and w_i denote element i of vectors $\boldsymbol{x}$ and $\boldsymbol{w}$, respectively. Since the representation range of y is bounded by that of $\boldsymbol{x}$ and $\boldsymbol{w}$, we use their ranges to derive lower bounds on the bit width P of the accumulation register, or accumulator.

$$y = \sum_{i=1}^{K} x_i w_i \tag{2.4}$$

It is common for input data to be represented with unsigned integers either when following activation functions with non-negative dynamic ranges (*e.g.*, rectified linear units, or ReLUs), or when an appropriate zero point is adopted (*i.e.*, asymmetric quantization). Otherwise, signed integers are used. Since weights are most often represented with signed integers, we assume the accumulator is always signed in our work. Therefore, given that the scalar result of the dot product between $\boldsymbol{x}$ and $\boldsymbol{w}$ is a P-bit integer defined by Eq. 2.4, it follows that $\sum_{i=1}^{K} x_i w_i$ is bounded such that:

$$-2^{P-1} \leq \sum_{i=1}^{K} x_i w_i \leq 2^{P-1} - 1 \tag{2.5}$$

To satisfy the right-hand side of this double inequality, it follows that $|\sum_{i=1}^{K} x_i w_i| \leq 2^{P-1} - 1$. However, the accumulator needs to be wide enough to not only store the result of the dot product, but also all intermediate partial sums.

Since input data is not known *a priori*, our bounds must consider the worst-case values for every MAC. Thus, because the magnitude of the sum of products is upper-bounded by the sum of the product of magnitudes, it follows that if $\sum_{i=1}^{K} |x_i||w_i| \leq 2^{P-1} - 1$, then the dot product between $\boldsymbol{x}$ and $\boldsymbol{w}$ fits into a P-bit accumulator without numerical overflow, as shown below.

$$|\sum_i x_i w_i| \leq \sum_i |x_i w_i| \leq \sum_i |x_i||w_i| \leq 2^{P-1} - 1 \tag{2.6}$$

2.5.1 Deriving Lower Bounds Using Data Types

The worst-case values for each MAC can naïvely be inferred from the representation range of the data types used. When x_i and w_i are signed integers, their magnitudes are bounded such that $|x_i| \leq 2^{N-1}$ and $|w_i| \leq 2^{M-1}$, respectively. In scenarios where x_i is an unsigned integer, the magnitude of each input value is upper-bounded such that $|x_i| \leq 2^N - 1$; however, we simplify this upper bound to be $|x_i| \leq 2^N$ for convenience of notation[1]. Combining the signed and unsigned upper bounds, it follows that $|x_i| \leq 2^{N - \mathbb{1}_{\text{signed}}(\boldsymbol{x})}$, where $\mathbb{1}_{\text{signed}}(\boldsymbol{x})$ is an indicator function that returns 1 if and only if $\boldsymbol{x}$ is a vector of signed integers.

Building from Eq. 2.6, it follows that the sum of the product of the magnitudes is bounded such that:

$$\sum_{i=1}^{K} |x_i||w_i| \leq K \cdot 2^{N+M-1-\mathbb{1}_{\text{signed}}(\boldsymbol{x})} \leq 2^{P-1} - 1 \tag{2.7}$$

Taking the log of both sides, we can derive a lower bound on the accumulator bit width P:

$$\log_2 \left(2^{\log_2(K)+N+M-1-\mathbb{1}_{\text{signed}}(\boldsymbol{x})} + 1 \right) + 1 \leq P \tag{2.8}$$

[1]Note that our simplification of the upper bound for unsigned input data does not compromise overflow avoidance.

This simplifies to the following lower bound on P:

$$P \geq \alpha + \phi(\alpha) + 1 \tag{2.9}$$

$$\alpha = \log_2(K) + N + M - 1 - \mathbb{1}_{\text{signed}}(\boldsymbol{x}) \tag{2.10}$$

$$\phi(\alpha) = \log_2(1 + 2^{-\alpha}) \tag{2.11}$$

In Figure 2.4, we visualize this bound assuming that $\boldsymbol{x}$ is a vector of unsigned integers such that $\mathbb{1}_{\text{signed}}(\boldsymbol{x}) = 0$. There, we show how the lower bound on the accumulator bit width increases as we vary both the size of the dot product (K) and the bit width of the weights and activations.

2.5.2 Deriving Lower Bounds Using Learned Weights

Since learned weights are frozen during inference time, we can use knowledge of their magnitudes to derive a tighter lower bound on the accumulator bit width.

Building again from Eq. 2.6, the sum of the product of magnitudes is bounded by Eq. 2.12, where $\|\boldsymbol{w}\|_1$ denotes the standard ℓ_1-norm over vector $\boldsymbol{w}$.

$$\textstyle\sum_{i=1}^{K} |x_i||w_i| \leq 2^{N - \mathbb{1}_{\text{signed}}(\boldsymbol{x})} \cdot \|\boldsymbol{w}\|_1 \leq 2^{P-1} - 1 \tag{2.12}$$

Here, we define a tighter lower bound on P:

$$P \geq \beta + \phi(\beta) + 1 \tag{2.13}$$

$$\beta = \log_2(\|\boldsymbol{w}\|_1) + N - \mathbb{1}_{\text{signed}}(\boldsymbol{x}) \tag{2.14}$$

$$\phi(\beta) = \log_2(1 + 2^{-\beta}) \tag{2.15}$$

In Figure 2.4, we visualize this bound again assuming that $\boldsymbol{x}$ is a vector of unsigned integers. Because Eq. 2.14 is dependent on the values of the learned weights, we randomly

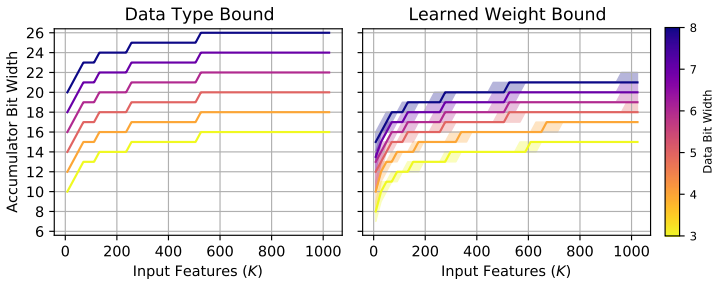


Figure 2.4. We visualize the differences between our accumulator bit width bounds as we vary the size of the dot product (K) as well as the bit width of the weights (M) and input activations (N), which we jointly refer to as "data bit width" such that $M = N$.

sample each K-dimensional vector from a discrete Gaussian distribution and show the median accumulator bit width along with the minimum and maximum observed over 300 random samples. We show that using learned weights (right) provides a tighter lower bound on the bit width of the accumulator than using data types (left) as we vary both the size of the dot product (K) and the bit width of the weights and input activations.

2.6 Training Quantized Neural Networks to Avoid Overflow

To train QNNs to use low-precision accumulators without overflow, we use weight normalization as a means of constraining learned weights $\boldsymbol{w}$ to satisfy the bound derived in Section 2.5.2. Building from Eq. 2.12, we transform our lower bound on accumulator bit width P to be the upper bound on the ℓ_1-norm of $\boldsymbol{w}$ given by Eq. 2.16. Note that because each output neuron requires its own accumulator, this upper bound needs to be enforced channelwise.

$$\|\boldsymbol{w}\|_1 \leq \left(2^{P-1} - 1\right) \cdot 2^{\mathbb{1}_{\text{signed}}(\boldsymbol{x}) - N} \tag{2.16}$$

2.6.1 Constructing Our Quantization Operator

To enforce this constraint during QAT, we reparameterize our quantizer such that each weight vector $\boldsymbol{w}$ is represented in terms of parameter vectors $\boldsymbol{g}$ and $\boldsymbol{v}$. Similar to the standard weight normalization formulation discussed in Section 2.3.2, this reparameterization decouples

the norm from the weight vector; however, unlike the standard formulation, the norm is learned for each output channel. For a given layer with C output channels, we replace the per-tensor ℓ_2-norm of the standard formulation (Eq. 2.3) with a per-channel ℓ_1-norm. This reparameterization, given by Eq. 2.17, allows for the ℓ_1-norm of weight vector $\boldsymbol{w}$ to be independently learned per-channel such that $g_i = \|\boldsymbol{w}_i\|_1$ for all $i \in \{1, \cdots, C\}$, where $\boldsymbol{w}_i$ denotes the weights of channel i and g_i denotes element i in parameter vector $\boldsymbol{g}$.

$$\boldsymbol{w}_i = g_i \cdot \frac{\boldsymbol{v_i}}{\|\boldsymbol{v_i}\|_1} \quad \forall\, i \in \{1, \cdots, C\} \tag{2.17}$$

Similar to the standard quantization operator, our weight normalization-based quantization relies on a uniform affine mapping from the high-precision real domain to the low-precision quantized domain using learned per-channel scaling factors $\boldsymbol{s} = \{s_i\}_{i=1}^{C}$. Thus, by constraining g_i to satisfy Eq. 2.18, we can learn quantized weights that satisfy our accumulator bit width bound and avoid overflow.

$$g_i \leq s_i \cdot \left(2^{P-1} - 1\right) \cdot 2^{\mathbb{1}_{\text{signed}}(\boldsymbol{x}) - N} \tag{2.18}$$

Below, we articulate our weight normalization-based quantization operator. For clarity and convenience of notation, we consider a layer with one output channel (*i.e.*, $C = 1$) such that parameter vectors $\boldsymbol{g} = \{g_i\}_{i=1}^{C}$ and $\boldsymbol{s} = \{s_i\}_{i=1}^{C}$ can be represented as scalars g and s, respectively.

$$\text{quantize}(\boldsymbol{w}; s, z) := \text{clip}\left(\left\lfloor \frac{g}{s} \frac{\boldsymbol{v}}{\|\boldsymbol{v}\|_1} \right\rfloor + z; n, p\right) \tag{2.19}$$

During training, our weight quantization operator applies fours elementwise operations the following in order: scale, round, clip, then dequantize. As with the standard operator, we eliminate the zero points in our mapping such that $z = 0$. We use an exponential parameterization of both the scaling factor $s = 2^d$ and the norm parameter $g = 2^t$, where d and t are both log-scale parameters to be learned through stochastic gradient descent. This is similar to the work

of [83] with the caveat that we remove integer power-of-2 constraints, which provide no added benefit to the streaming architectures generated by FINN as floating-point scaling factors can be absorbed into the threshold logic via mathematical manipulation [18]. The scaled tensors are then rounded to zero, which we denote by $\lfloor \cdot \rfloor$. This prevents any upward rounding that may cause the norm to increase past our constraint. Note that this is another difference from the conventional quantization operators, which use half-way rounding. Finally, once scaled and rounded, the elements in the tensor are then clipped and dequantized using Eq. 2.2. Our resulting quantization operator used during training is given by Eq. 2.20, where n and p depend on the representation range of weight bit width M.

$$q(\boldsymbol{w}; s) := \text{clip}\left(\left\lfloor \frac{g}{s} \frac{\boldsymbol{v}}{\|\boldsymbol{v}\|_1} \right\rfloor; n, p\right) \cdot s \tag{2.20}$$

$$\text{where } s = 2^d \tag{2.21}$$

$$\text{and } g = 2^{\min(T,t)} \tag{2.22}$$

$$\text{and } T = \mathbb{1}_{\text{signed}}(\boldsymbol{x}) + \log_2(2^{P-1} - 1) + d - N \tag{2.23}$$

When quantizing our activations, we use the standard quantization operators discussed in Section 2.3.1. All activations that follow non-negative functions (*i.e.*, ReLU) are represented using unsigned integers, otherwise they are signed.

To update our learnable parameters during training, we use the straight-through estimator (STE) [13] to allow local gradients to permeate our rounding function such that $\nabla_x \lfloor x \rfloor = 1$ everywhere, as is common practice.

2.6.2 Regularization with Lagrangian Penalties

To avoid t getting stuck when $t > T$ in Eq. 2.22, we introduce the Lagrangian penalty $\mathscr{L}_{\text{penalty}}$ given by Eq. 2.24. For a neural network with L layers, each with C_l output channels, $t_{i,l}$ denotes the log-scale parameter of the norm for channel i in layer l and $T_{i,l}$ denotes its upper bound as given by Eq. 2.23. Note that, even when $\mathscr{L}_{\text{penalty}} > 0$, we still satisfy our accumulator

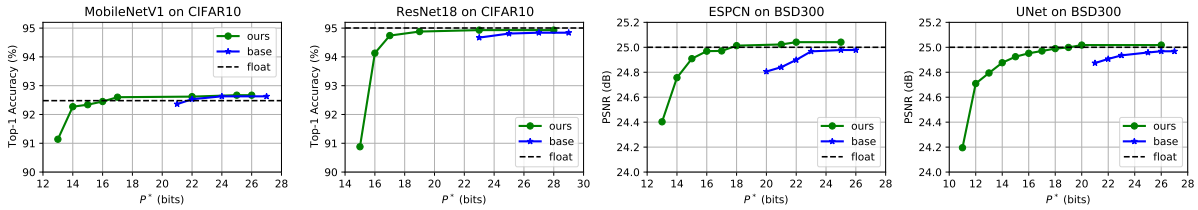


Figure 2.5. Using image classification and single-image super resolution benchmarks, we show that we are able to maintain model performance with respect to the floating-point baseline even with significant reductions to the accumulator bit width. We visualize this trade-off using Pareto frontiers estimated using a grid search over various weight, activation, and accumulator bit widths. Here, we use P^* to denote the largest accumulator bit width allowed across all layers in the network. We compare our algorithm (**green dots**) against the standard quantization baseline algorithm (**blue stars**) and repeat each experiment 3 times, totaling over 500 runs per model to form each set of Pareto frontiers. We observe that our algorithm dominates the baseline in all benchmarks, showing that we can reduce the accumulator bit width without sacrificing significant model performance even with respect to a floating-point baseline.

constraints by clipping the norm in Eq. 2.22. However, our Lagrangian penalty encourages norm g_i and scale s_i of each channel i in each layer to jointly satisfy the bound without clipping, which allows their log-scale parameters to be updated with respect to only the task error.

$$\mathscr{L}_{\text{penalty}} = \sum_{l=1}^{L} \sum_{i=1}^{C_l} \left(t_{i,l} - T_{i,l}\right)_+ \tag{2.24}$$

It is important to note that our formulation is task agnostic. Assuming base task loss $\mathscr{L}_{\text{task}}$, our total loss $\mathscr{L}_{\text{total}}$ is given by Eq. 2.25, where λ is a Lagrange multiplier. In our experiments, we fix λ to be a constant scalar, although adaptive approaches such as [137] could be explored.

$$\mathscr{L}_{\text{total}} = \mathscr{L}_{\text{task}} + \lambda \mathscr{L}_{\text{penalty}} \tag{2.25}$$

2.7 Experiments

We evaluate our algorithm using image classification and single-image super resolution benchmarks. Because our algorithm is the first of its kind, we compare our approach to the standard QAT formulation discussed in Section 2.3.1. We implement all algorithms in PyTorch

using Brevitas v0.7.2 [126], where single-GPU quantization-aware training times range from 5 minutes (*e.g.*, ESPCN) to 2 hours (*e.g.*, MobileNetV1) on an AMD MI100 accelerator. To generate custom FPGA architectures for the resulting QNNs, we work from FINN v0.8.1 [7] and add extensions to support our work.

2.7.1 Image Classification Benchmarks

We train MobileNetV1 [77] and ResNet18 [66] to classify images using the CIFAR10 dataset [93]. We closely follow the network architectures originally proposed by the respective authors, but introduce minor variations that yield more amenable intermediate representations given the image size as we discuss below. As is common practice, we fix the input and output layers to 8-bit weights and activations for all configurations, and initialize all models from floating-point counterparts pre-trained to convergence on CIFAR10. We evaluate all models by the observed top-1 test accuracy.

For MobileNetV1, we use a stride of 2 for both the first convolution layer and the final average pooling layer. This reduces the degree of downscaling to be more amenable to training over smaller images. All other layer configurations remain the same as proposed in [77]. We use the stochastic gradient descent (SGD) optimizer to fine-tune all models for 100 epochs in batches of 64 images using a weight decay of 1e-5. We use an initial learning rate of 1e-3 that is reduced by a factor of 0.9 every epoch.

For ResNet18, we alter the first convolution layer to use a stride and padding of 1 with a kernel size of 3. Similar to MobileNetV1, we remove the preceding max pool layer to reduce the amount of downscaling throughout the network. We also use a convolution shortcut [67] rather than the standard identity as it empirically proved to yield superior results in our experiments. All other layer configurations remain the same as proposed in [66]. We use the SGD optimizer to fine-tune all models for 100 epochs in batches of 256 using a weight decay of 1e-5. We use an initial learning rate of 1e-3 that is reduced by a factor of 0.1 every 30 epochs.

2.7.2 Single-Image Super Resolution Benchmarks

We train ESPCN [144] and UNet [136] to upscale single images by a factor of 3x using the BSD300 dataset [113]. Again, we closely follow the network architectures originally proposed by the respective authors, but introduce minor variations that yield more hardware-friendly network architectures. As is common practice, we fix the input and output layers to 8-bit weights and activations for all configurations; however, we train all super resolution models from scratch. We empirically evaluate all models by the peak signal-to-noise ratio (PSNR) observed over the test dataset.

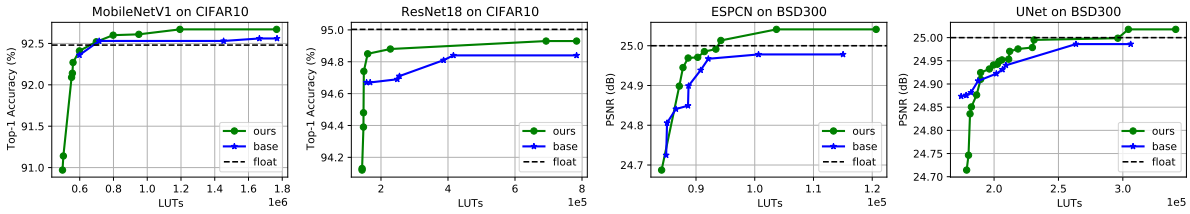
For ESPCN, we replace the sub-pixel convolution with a nearest neighbor resize convolution (NNRC), which has been shown to reduce checkerboard artifacts during training [124] and can be efficiently executed during inference [33]. All other layer configurations remain the same as proposed in [144]. We use the Adam optimizer [90] to fine-tune all models for 100 epochs in batches of 16 images using a weight decay of 1e-4. We use an initial learning rate of 1e-4 that is reduced by a factor of 0.98 every epoch.

For UNet, we use only 3 encoders and decoders to create a smaller architecture than originally proposed by [136]. We replace transposed convolutions with NNRCs, which have been shown to be functionally equivalent during inference [33], but have more favorable behavior during training [124]. We replace all concatenations with additions and reduce the input channels accordingly. We use the Adam optimizer to fine-tune all models for 200 epochs in batches of 16 images using a weight decay of 1e-4. We use an initial learning rate of 1e-3 that is reduced by a factor of 0.3 every 50 epochs.

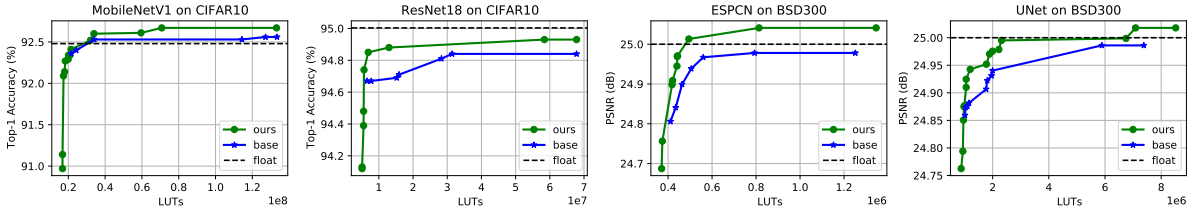
2.7.3 Experiment Setup and Research Questions

We design our experiments around the following questions:

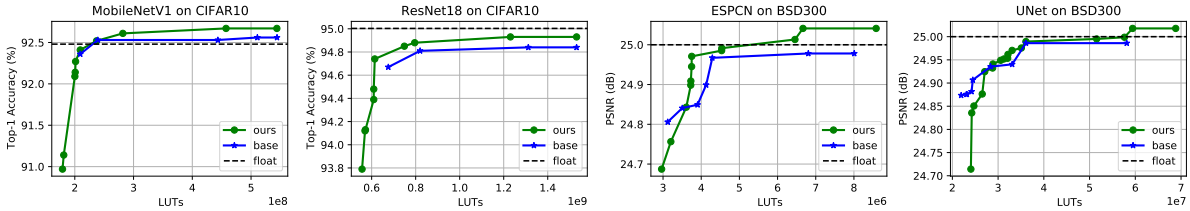
- How does reducing the accumulator bit width impact model performance?
- What are the trade-offs between resource utilization and model performance?



(a) No Parallelism



(b) Max PEs



(c) Max PEs and SIMDs

Figure 2.6. To evaluate the trade-off between resource utilization and accuracy, we visualize the Pareto frontier observed over our image classification and single-image super resolution benchmarks. We evaluate these Pareto frontiers in the following scenarios: (a) no spatial parallelism; (b) maximizing the PEs for each layer; and (c) maximizing the PEs and SIMDs for each layer. In each scenario, we observe that our algorithm (**green dots**) provides a dominant Pareto frontier across each model when compared to the standard quantization algorithm (**blue stars**), showing that our algorithm can reduce LUT utilization without sacrificing significant model performance.

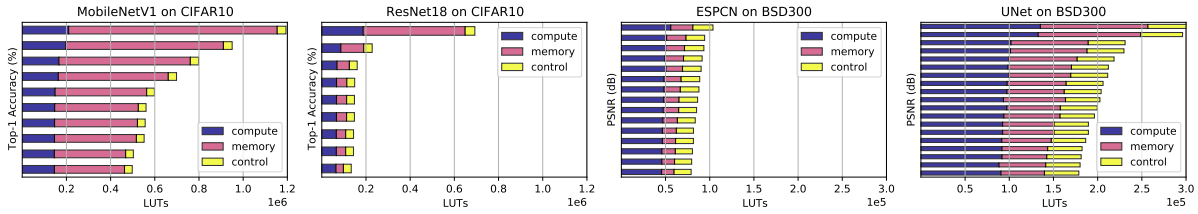
- Where do our resource savings come from as we reduce accumulator bit width?

Similar to the experiments described in Section 3.3, we simplify our analysis and assume that LUTs are the only type of resources available. We configure the FINN compiler to target LUTs for both compute and memory wherever possible so that we can evaluate the impact of accumulator bit width on resource utilization using just one type of resource.

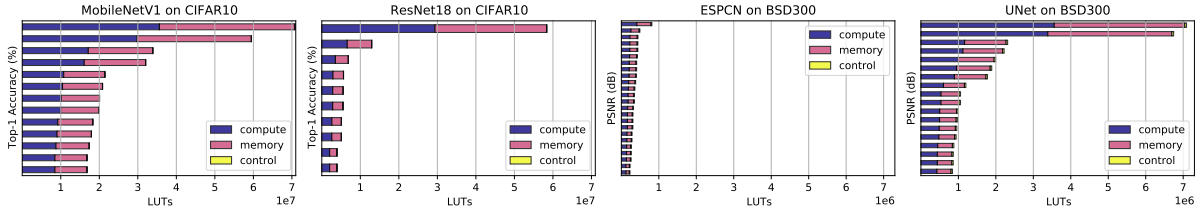
Throughout our experiments, we focus our attention on data bit widths between 5 and 8 bits for two reasons: (1) reducing precision below 5 bits often requires uniquely tailored

hyperparameters, which would not make for an even comparison across bit widths; and (2) reducing the size of the accumulator has a negligible impact on LUT utilization at lower data bit widths, as shown in Section 3.3. Still, even with a reduced set of possible data bit widths, it is computationally intractable to test every combination of weight, activation, and accumulator bit widths within the design space exposed by the QNNs that we use as benchmarks. Thus, for weight and activation bit widths, we uniformly enforce precision for each hidden layer in the network such that M and N are constant scalars, aside from the first and last layers that remain at 8 bits such that $M = N = 8$. The accumulator bit width, however, is dependent on not only the weight and activation bit widths, but also the size of the dot product (K), as discussed in Section 2.5. To simplify our design space, we constrain all layers in a given network to use the same accumulator bit width that we denote as P^* such that the maximum accumulator bit width for any layer is P^* bits. Recall that the value for P^* is used by Eq. 2.16 to upper bound the ℓ_1-norm of each weight per-channel and used by Eq. 2.23 to enforce this constraint during training.

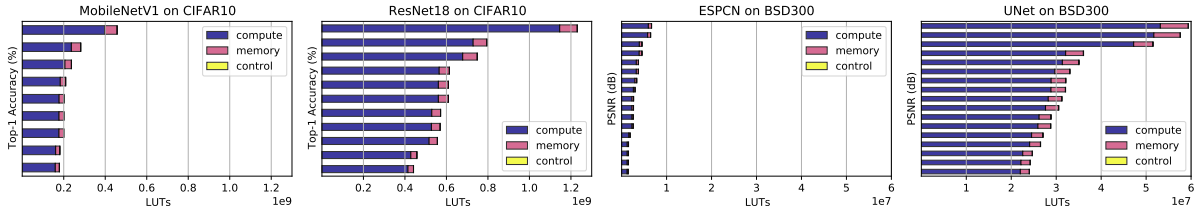
To collect enough data to investigate our research questions, we perform a grid search over weight and activation bit widths from 5 to 8 bits. For each of these 16 combinations, we calculate the largest lower bound on the accumulator bit width as determined by the data type bound (Eq. 2.9) of the largest layer in the network. Using this to initialize P^*, we evaluate up to a 10-bit reduction in the accumulator bit width to create a total of 160 configurations. Finally, we benchmark our results against the standard quantization algorithm discussed in Section 2.3.1; however, because it does not expose control over accumulator bit width, this grid search is restricted to the 16 combinations of weight and activation bit widths. We run each configuration 3 times to form a total of 528 runs per model. In the following sections, we summarize our findings.



(a) No Paralleism



(b) Max PEs



(c) Max PEs and SIMDs

Figure 2.7. We break down LUT utilization into compute, memory, and control flow for each of our Pareto optimal models from Figure 2.6.

2.7.4 Accumulator Impact on Model Performance

Our algorithm introduces a novel means of constraining the weights of a QNN to use a pre-defined accumulator bit width without overflow. As an alternative to our algorithm, a designer can choose to heuristically manipulate data bit widths based on our data type bound given by Eq. 2.9. Such an approach would still guarantee overflow avoidance when using a pre-defined accumulator bit width P, but is an indirect means of enforcing such a constraint. Given a pre-defined accumulator bit width upper bound P^*, we compare the performance of models trained with our algorithm against this heuristic approach. We visualize this comparison as a Pareto frontier in Figure 2.5. It is important to note that, while this is not a direct comparison against the algorithm proposed by [40], the experiment is similar in principle. Unlike [40],

we use the more advanced quantization techniques detailed in Section 2.3.1, and replace the computationally expensive loss-guided search technique with an even more expensive, but more comprehensive grid search.

The Pareto frontier shows the maximum observed model performance for a given P^*. We observe that our algorithm can push the accumulator bit width lower than what is attainable using current methods while also maintaining model performance. Furthermore, most models show less than a 1% performance drop from even the floating-point baseline with a 16-bit accumulator, which is most often the target bit width for low-precision accumulation in general-purpose processors [40, 170].

2.7.5 Trade-Offs Between Resources and Accuracy

To understand the impact that accumulator bit width can have on the design space of the accelerators generated by FINN, we evaluate the trade-offs between resource utilization and model performance. For each of the models trained in the grid search detailed in Section 2.7.3, we use the FINN compiler to generate resource utilization estimates and use Pareto frontiers to visualize the data. In Figure 2.6, we provide the maximum observed model performance for the total LUTs used by the accelerator. We evaluate these Pareto frontiers with three optimization configurations. First, we instantiate each layer in each model as a CU without any spatial parallelism optimization and visualize the Pareto frontiers in Figure 2.6a. Second, we maximize the number of PEs used in each layer in each model and visualize the Pareto frontiers in Figure 2.6b. Finally, we maximize both the number of PEs and the SIMD lanes used and visualize the Pareto frontiers in Figure 2.6c.

To ensure that the trends we analyze from these estimates are meaningful, we return to the experiments carried out in Section 3.3. We compare the absolute LUT utilization reported from post-synthesis RTL against the corresponding FINN estimates and observe a 94% correlation.

Reducing the precision of weights and activations provides resource utilization savings in exchange for model performance; however, we observe that adding the accumulator bit width

to the design space provides a better overall trade-off. Our results show that, for a given target accuracy or resource budget, our algorithm can offer a better trade-off between LUT utilization and model performance than existing baselines for various optimization strategies, confirming the benefit of including the accumulator bit width in the overall HW-SW co-design space.

2.7.6 Evaluating Resource Savings

Because we force the FINN compiler to use LUTs for compute and memory resources wherever possible, we evaluate where our resource savings come from. To do so, we separate LUT utilization into compute, memory, and control flow. For compute, we aggregate the LUTs used for adder trees, MACs, and comparison logic; for memory, the LUTs used to store weights, thresholds, and intermediate representation; and for control flow, the LUTs used for on-chip interconnects and AXI-Stream protocols. In Figure 2.7, we visualize this break down for each of the Pareto optimal models that correspond to our Pareto frontier in Figure 2.6.

We observe that the majority of LUT savings come from reductions to memory resources when accelerators are generated without spatial parallelism, but primarily come from compute resources as parallelism is increased. Without parallelism, the reductions in LUT utilization are largely from the reduced storage costs of thresholds and intermediate activations, which are directly impacted by the precision of the accumulator and output activations. As parallelism is increased, the reductions in compute LUTs primarily come from the reduced cost of MACs, which are directly impacted by the precision of the weights, inputs, and accumulators.

Finally, we observe that the control flow LUTs largely remain constant for each network, which is expected as the network architecture is not impacted by changes to the data types used. Noticeably, the relative share of LUTs contributed by control flow logic is higher for networks with skip connections (*e.g.*, ResNet18 and UNet) than without (*e.g.*, MobileNetV1 and ESPCN), and is relatively less impactful as parallelism is increased.

2.7.7 A Deeper Look at the Impact of Our Constraints

As a byproduct of our weight normalization formulation, our quantization algorithm provides a means of not only constraining weights to fit into an accumulator of user-defined bit width, but also of increasing the sparsity and compressibility of the resulting weights, as shown in Figure 2.8.

Sparsity is the proportion of zero-valued elements in a tensor. The most common use of sparsity in machine learning workloads is to accelerate inference by reducing compute and memory requirements [54]. We direct the interested reader to [75] for a recent survey of prior work on sparsity in deep learning. Among this work are various studies regarding the use of ℓ_1-norm weight regularization as a means of introducing sparsity [25, 174]. Our quantization algorithm has a similar effect. By replacing the ℓ_2-norm of the standard weight normalization formulation with our log-scale ℓ_1-norm parameter, we introduce a novel means of encouraging unstructured weight sparsity. Recall that the value of P^* is used by Eq. 2.16 to upper bound the ℓ_1-norm of the weights. Consequently, reducing P^* further constrains this upper bound to encourage weight sparsity as a form of ℓ_1-norm weight regularization. In Figure 2.8 on the left, we visualize the average sparsity across all of our benchmark models as we reduce the accumulator bit width P^*.

Compressibility is often estimated using the theoretical lower bound on the amount of bits per element as measured by the entropy [143]. Reducing the entropy reduces the amount of information required for lossless compression, increasing the compression rate. Prior work has studied the use of entropy regularization as a means of improving weight compression [4, 11]. We observe that our ℓ_1-norm constraints have a similar effect as these techniques. As shown in Figure 2.8 in the middle, we observe that the entropy decreases as we reduce the accumulator bit width P^*.

In Figure 2.8, we visualize how the sparsity, compressibility, and relative model performance are effected by reductions to P^* using the models from our grid search described in

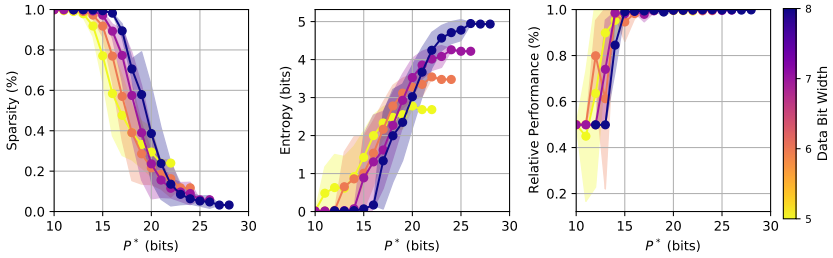


Figure 2.8. As a result of our ℓ_1-norm constraints, reducing the accumulator bit width exposes opportunities to exploit unstructured sparsity (left) and weight compression (middle) without sacrificing model performance relative to the floating-point baseline (right).

Section 2.7.3. To simplify our analysis, we focus on configurations where the weight and input activation bit widths were the same (*e.g.*, $M = N$), and plot the averages observed across all 4 of our benchmark models. For models with 8-bit weights and activations, we observe that reducing P^* to 16 bits yields an average sparsity of 98.2% with an estimated compression rate of 46.5x while maintaining 99.2% of the floating-point performance.

2.8 Conclusions and Future Work

We propose a novel quantization algorithm to train QNNs for low-precision accumulation. Our algorithm leverages weight normalization as a means of constraining learned parameters to fit into an accumulator of a pre-defined bit width. Unlike previous work, which has sought to merely reduce the risk of overflow or mitigate its impact on model accuracy, our approach guarantees overflow avoidance.

Our study is the first to our knowledge that explores the use of low-precision accumulators as a means of improving the design efficiency of programmable hardware used as QNN inference accelerators. As such, we theoretically evaluate overflow and derive comprehensive bounds on accumulator bit width with finer granularity than existing literature. Our experiments show that using our algorithm to train QNNs for AMD-Xilinx FPGAs improves the trade-offs between resource utilization and model accuracy when compared to the standard baseline. Our results inform the following takeaways:

- While reducing the size of the accumulator invariably degrades model accuracy, our algorithm significantly alleviates this trade-off.
- Using our algorithm to train QNNs for lower precision accumulators yields higher-performing models for the same resource budget when compared to the baseline.
- Without spatial parallelism, the majority of our resource savings come from reductions to memory requirements because reducing the accumulator bit width also reduces the cost of storing thresholds and intermediate activations.
- As spatial parallelism is increased, reductions in compute costs dominate our resource savings because reducing the accumulator bit width reduces the cost of creating more MACs.
- Our algorithm inherently encourages extreme unstructured sparsity and increased compressibility of the resulting weights of the QNN while maintaining performance relative to the floating-point baseline.

The flexibility of FPGAs is a double-edged sword. The bit-level control allows for the precisions of weights, activations, and now accumulators to be individually tuned for each layer in a QNN; however, the design space exposed by so many degrees of freedom introduces a complex optimization problem. Our algorithm increases the flexibility of HW-SW co-design by exposing the accumulator bit width as yet another parameter that can be tuned when simultaneously optimizing QNNs and their corresponding inference accelerators. In future work, we hope to explore the use of state-of-the-art neural architecture search algorithms as a means of navigating this large design space more efficiently.

Acknowledgements: This chapter is based on unpublished material (Ian Colbert, Alessandro Pappalardo, and Jakoba Petri-Koenig, "Quantized Neural Networks for Low-Precision Accumulation with Guaranteed Overflow Avoidance"). The dissertation author was the primary

investigator and author of this paper. We would also like to thank Gabor Sines, Michaela Blott, Nicholas Fraser, Yaman Umuroglu, Thomas Preusser, Mehdi Saeedi, Ihab Amer, Alex Cann, Arun coimbatore Ramachandran, Chandra Kumar Ramasamy, Prakash Raghavendra, and the rest of the AMD RTG Software Technology, Edge Inference, and AECG Research teams for insightful discussions and infrastructure support.

Chapter 3

Low-Precision Structured Subnetworks

Pruning and quantization are core techniques used to reduce the inference costs of deep neural networks. Among the state-of-the-art pruning techniques, magnitude-based pruning algorithms have demonstrated consistent success in the reduction of both weight and feature map complexity. However, we find that existing measures of neuron (or channel) importance estimation have at least one of two limitations: (1) failure to consider the interdependence between successive layers; and/or (2) parametric estimation that relies on distributional assumptions of the feature maps. In this chapter, we demonstrate that the importance rankings of the output neurons of a given layer strongly depend on the sparsity level of the preceding layer, and therefore naïvely estimating neuron importance to drive magnitude-based pruning will lead to sub-optimal performance. Informed by this observation, we introduce a purely data-driven non-parametric, magnitude-based channel pruning strategy that we greedily apply using the activations of the previous sparsified layer. We demonstrate that our proposed method works effectively in combination with statistics-based quantization techniques to generate low-precision structured subnetworks that can be efficiently accelerated by hardware platforms such as GPUs and FPGAs. Using our proposed algorithms, we demonstrate an increased performance-per-memory footprint over existing solutions across a range of discriminative and generative networks.

3.1 Introduction

The performance of deep neural networks (DNNs) has been shown to scale with the size of both the training dataset and model architecture [69]; however, the resources required to deploy larger networks for inference can be prohibitive as they often exceed the compute and storage budgets of resource-constrained platforms such as mobile or edge devices [53, 63]. Therefore, as the usage of deep learning has proliferated in real-time applications with tight energy consumption budgets and low latency requirements, the field of research focused on reducing inference costs while maintaining model performance has rapidly expanded in recent years. Of the many techniques studied to accomplish this task, *pruning* and *quantization* are the most widely used, and are often complementary to other approaches such as network distillation [131] and neural architecture search [47, 162].

Pruning (*i.e.*, the process of removing identified redundant elements from a neural network) and quantization (*i.e.*, the processing of reducing their precision) are often considered to be independent problems [103, 128]; however, recent work has begun to study the application of both in either a joint [63, 162, 178, 188] or unified [32, 156] setting. *Unified* algorithms typically use mixed precision quantization and integrate pruning by reducing the precision of an element (or a set of elements) to 0. On the other hand, *joint* algorithms combine separate optimization objectives for pruning and quantization under one learning framework often using the standard "prune-then-quantize" paradigm [63, 184]. In this work, we study the joint application of pruning and quantization under this paradigm across both discriminative and generative tasks for the purpose of learning low-precision structured subnetworks that can be efficiently accelerated by highly-parallelized hardware platforms.

Motivated by our observation that data-driven measures of neuron importance strongly depend on the activation distribution of the preceding layer, we design a greedy layerwise channel pruning algorithm that is heuristically guided by non-parametric estimates. We evaluate the performance of our algorithm using various pruning schedules and alternative measures

of neuron importance based on pre-existing literature, and observe that our greedy layerwise algorithm yields consistent benefits. Intuitively, our results suggest that, by allowing a given layer to adjust to abrupt shifts to its input activation distribution *before* pruning, the heuristics used to rank order neurons by importance become more effective, which leads to improved network performance. Furthermore, when combined with our moving average statistics-based uniform quantization procedure, we are able to learn low-precision structured subnetworks with minimal performance degradation for both discriminative and generative tasks. Our joint pruning and quantization algorithm is visualized in Figure 3.1, where we depict the sequence of pruning and quantization steps used during training. As further described in Section 3.4, our framework uses data-driven importance measures to guide our greedy layerwise channel pruning algorithm before our quantization operator is activated to reduce the precision of both the weights $\boldsymbol{w}_i$ and activations $\boldsymbol{h}_i$ for each layer $i \in \{1, \cdots, L\}$. The pruning mask for each layer i is evaluated for t_p steps before moving to layer $i+1$. After all layers are pruned to a target sparsity, we activate the quantization operators and fine-tune for t_q steps. It is important to note that, while only the pruning mask of one layer is tuned every t_p steps, the weights in all layers are updated through gradient descent in every step, and latent operators act as identity functions until activated. The primary contributions of our work are summarized as follows:

1. We design a greedy layerwise channel pruning strategy using a non-parametric data-driven importance measure built without invoking any distributional assumptions.

2. We build a fully data-driven non-parametric framework to learn performant low-precision structured subnetworks by combining our layerwise channel pruning algorithm with quantization-aware training.

3. We evaluate our algorithm using alternative pruning schedules and neuron importance measures, and demonstrate clear advantages over pre-existing approaches.

4. We demonstrate increased performance per memory footprint over existing solutions across

a wide range of discriminative and generative computer vision tasks.

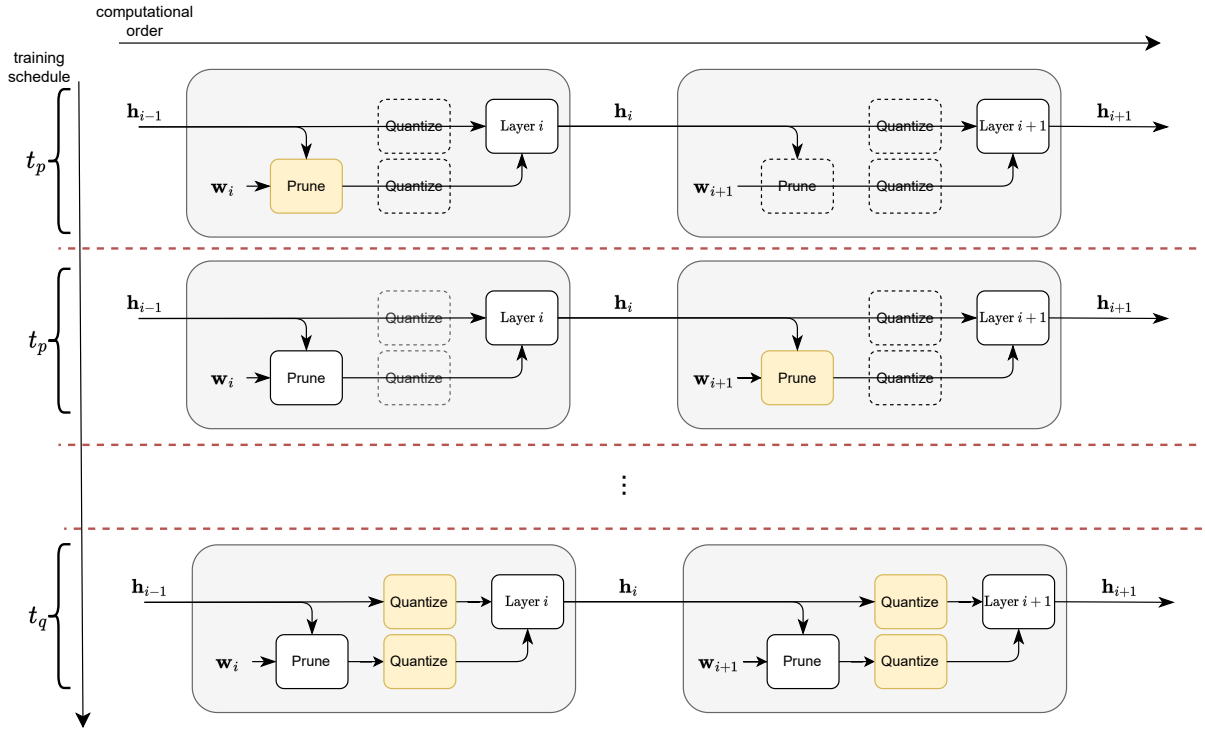


Figure 3.1. We introduce a joint layerwise channel pruning and uniform quantization framework built from algorithms formulated using moving average statistics. Here, yellow blocks denote operators actively being evaluated, clear blocks with dotted lines denote latent operators that have yet to be activated, and white blocks denote activated operators that have already been evaluated.

The outline of the rest of the chapter is as follows. In Section 3.2, we review prior work in neural network pruning and quantization. In Section 3.3, we motivate our intuition for data-driven layerwise pruning using non-parametric measures of correlation and distance between rank orderings of neuron importance. In Section 3.4, we describe our layer-by-layer channel pruning and moving average statistics-based quantization-aware training algorithms. In Section 3.5, we evaluate the performance results of our joint channel pruning and uniform quantization framework using discriminative and generative networks. In Section 6.7, we conclude the paper and discuss directions for future work.

3.2 Background

Our work explores the joint application of channel pruning and uniform quantization on both the weights and activations of deep neural networks. Here, we motivate our selected configurations for both.

3.2.1 Pruning

Neural network pruning techniques aim to reduce the inference costs of overparameterized models by identifying subnetworks that minimize memory requirements while maintaining task performance. In practice, pruning is often performed by setting the values of identified elements to zero, and the proportion of zero-valued elements is referred to as *sparsity*, where higher values correspond to fewer non-zero elements. Given a target sparsity, there are a variety of criteria used to identify which elements to prune; the most important of which are the topology constraints on the resulting subnetwork, the measure used to rank elements by importance, and the schedule in which elements are pruned.

Topology constraints for pruning techniques can be divided into *structured* or *unstructured* approaches. Structured pruning techniques introduce sparsity in varied levels of granularity (*e.g.*, entire kernels or channels), whereas unstructured pruning techniques impose no constraint on the topology of the sparsity scheme, as shown in Figure 3.2. Due to their inherent flexibility, unstructured pruning techniques can offer high compression rates with minimal accuracy degradation [51, 63]; however, they bring little hardware efficiency due to poor data locality caused by irregular sparsity patterns which create minimal opportunities for parallelism [68, 111]. Alternatively, structured pruning techniques offer more hardware-friendly implementations often by removing entire channels. This not only reduces the compute workload in a manner that is easily accelerated by most computing platforms [111], but also reduces energy consumption as activations dominate data transfer costs for both discriminative [85] and generative [33] models. Therefore, we focus on channel pruning in this work.

Various measures have been proposed to heuristically determine the relative importance of channels in a neural network. Weight magnitude has become the standard importance measure used to guide unstructured pruning algorithms [17, 51, 63], intuitively suggesting that larger magnitudes have greater influence in the network. Thus, weight magnitude can be extended to rank order channels using the ℓ_1-norm of channel weights [26]. We refer to such importance measures as *data-free*, as they do not require a data distribution to estimate neuron importance. Alternatively, purely *data-driven* approaches such as [78] prioritize removing redundancy from the information (*i.e.*, hidden activations) propagated through the network from the input data. This class of techniques has shown tremendous promise in finding performant structured subnetworks by heuristically guiding pruning algorithms to minimize feature reconstruction error [68], remove underutilized channels [78], or minimize the mutual information between successive layers [37]. However, we find that these approaches are limited by one or two key oversights:

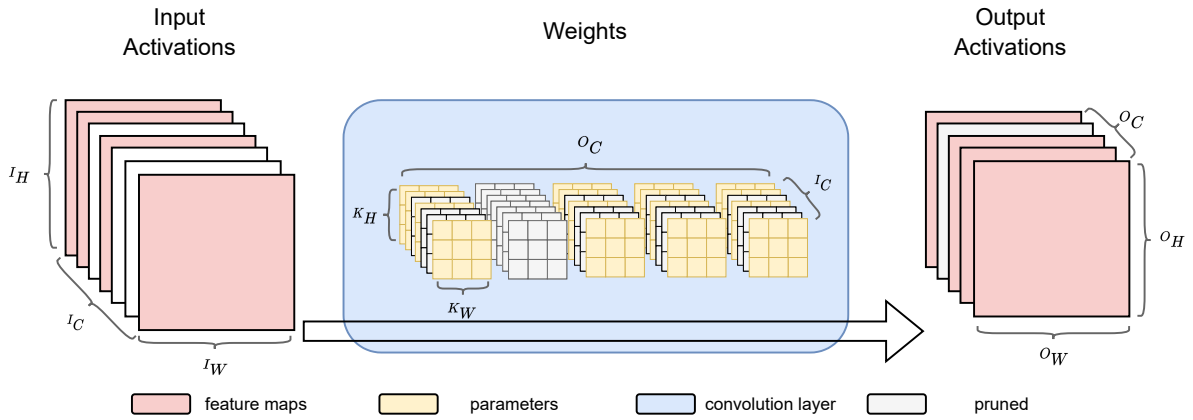
1. They do not consider the interdependence between successive layers in a neural network when measuring neuron importance, as in the case of [26, 78, 116].

2. They either use a parametric setting or invoke distributional assumptions on the activation data that may not hold true for all network architectures, as in the case of [37, 68, 108, 191].

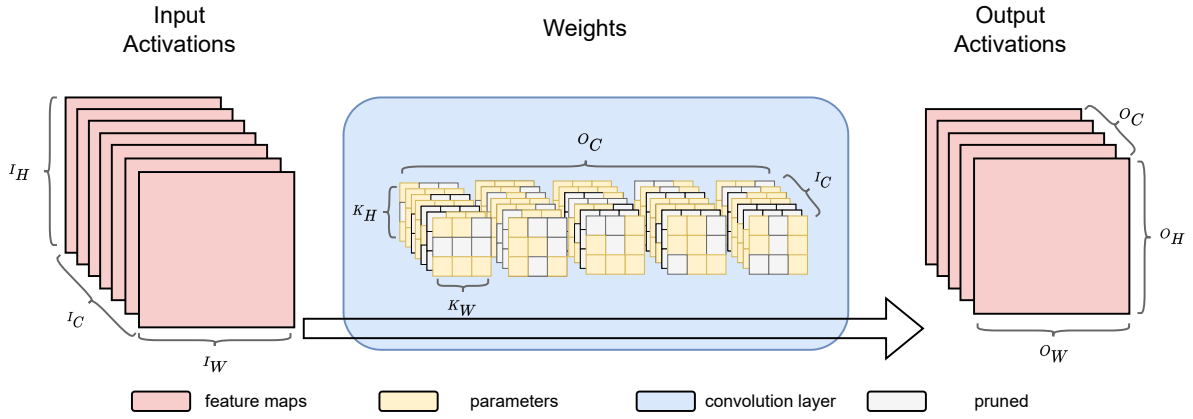
In this paper, using a completely non-parametric framework, we design a layerwise channel pruning algorithm using a data-driven measure of channel importance. Using targeted statistical experiments, we focus on some important differences between data-driven and data-free approaches when constructing our pruning algorithms. These experiments, further discussed in Section 3.3, also motivate the rationale for our layerwise approach.

3.2.2 Quantization

We are interested in quantization for the purpose of accelerating our structured subnetworks on mobile or edge devices; thus, we construct our "quantization" and "dequantization" operators from the standard *uniform affine* mapping from a high-precision real number r to a



(a) Structured Pruning



(b) Unstructured Pruning

Figure 3.2. We depict the differences between structured (top) and unstructured (bottom) pruning.

low-precision quantized number q using a scaling factor s and zero-point z, as given by Eq. 3.1. Although more complex non-uniform and non-linear mappings have been considered [131, 163], their utility and practicality are often circumstantial as they require dequantization before performing computation in the high-precision real domain and are thus far less efficient on resource-constrained hardware [168].

$$r = s \cdot (q - z) \tag{3.1}$$

As is standard practice, our quantizer (Eq. 3.2) and dequantizer (Eq. 3.3) are parameterized by scaling factor s and zero-point z. Here, s is a strictly positive real scaling factor and z is an integer value that maps to the real zero such that the real zero is exactly representable in the quantized domain. This ensures the numerical fidelity of common operations like zero-padding [82, 92]. Here, $\lfloor \cdot \rceil$ denotes the half-way rounding function and $\text{clip}(x; n, p) = \min(\max(x, n), p)$, where n and p are the clipping limits defined by the bit width b. For signed integers, $n = -2^{b-1}$ and $p = 2^{b-1} - 1$. For unsigned integers, $n = 0$ and $p = 2^b - 1$. As is standard practice, we use per-tensor scaling factors on the activations and per-channel scaling factors on only the weights [56].

$$\text{quantize}(x; s, z) := \text{clip}(\left\lfloor \frac{x}{s} \right\rceil + z; n, p) \tag{3.2}$$

$$\text{dequantize}(x; s, z) := s \cdot (x - z) \tag{3.3}$$

As reported in previous studies, eliminating zero points in the quantization mapping such that $z = 0$ reduces the computational overhead of cross-terms when executing inference using integer-only arithmetic [56, 83]. This strategy is commonly referred to as *symmetric quantization*, with *asymmetric quantization* as its alternative. To demonstrate the computational overhead of cross-terms introduced by asymmetric quantization, consider the real values for input activation x, weight w, and output activation y mapping to quantized values q_x, q_w, and q_y, respectively. The arithmetic for their product, $y = x \cdot w$, becomes the following:

$$s_y(q_y - z_y) = s_x(q_x - z_x) \cdot s_w(q_w - z_w) \tag{3.4}$$

where s_x, s_w, and s_y represent their respective scaling factors and z_x, z_w, and z_y represent their respective zero points. This simplifies to the following:

$$q_y = \frac{s_x s_w}{s_y}(q_x q_w - q_x z_w - q_w z_x + z_x z_w) + z_y \tag{3.5}$$

These cross-terms often require non-trivial optimizations to remain efficient; however, by constraining the quantization scheme of all weights in a DNN to be symmetric (*i.e.*, $z_w = 0$), we can mask the overhead of asymmetric quantization on the activations without any additional hardware or software optimizations as the arithmetic simplifies to Eq. 3.6, where l denotes layer $l \in \{0, \cdots, L\}$ in a neural network with L sequential layers.

$$\left(q_x^{(l+1)} - z_x^{(l+1)}\right) = \frac{s_x^{(l)} s_w^{(l)}}{s_x^{(l+1)}} q_w^{(l)} \left(q_x^{(l)} - z_x^{(l)}\right) \tag{3.6}$$

Note that, when using a symmetric quantizer on both the inputs and outputs to the neural network (*i.e.*, $x^{(0)}$ and $x^{(L)}$, respectively), we can freely use asymmetric quantization on the hidden layers without any overhead caused by the cross terms in Eq. 3.5. While previous works have alluded to this optimization [168], we are not aware of any research that has explicitly exploited it as we do. In our work, we focus on uniform affine quantization where we apply symmetric, per-channel quantization to the weights and asymmetric, per-tensor quantization to the activations.

3.3 Motivation

To demonstrate the interdependence between successive layers in a neural network, we inject structured sparsity into the input activations of a given layer within the network and evaluate how this leads to a shift in output channel importance rankings with respect to the dense input activations (*i.e.*, when there is no input sparsity). To rank order output channels by importance, we use two measures: (1) the mean ℓ_1-norm of the output activations generated by each channel; and (2) the ℓ_1-norm of the learned weights for each output channel. For a given layer i with hidden activations $\boldsymbol{h}_i$ and C_i output channels, we denote the importance estimates for each channel as $\mu_{i,c}$ where $c \in \{1, \cdots, C_i\}$. In order to compare two sets of rankings, we use the following non-parametric measures:

- **Kendall's Coefficient of Rank Correlation [91]:**

For a given layer i with C_i outputs channels, let the rank orderings of 2 sets of importance estimates $\boldsymbol{\mu}_i = \{\mu_{i,1}, \cdots, \mu_{i,C_i}\}$ and $\boldsymbol{\nu}_i = \{\nu_{i,1}, \cdots, \nu_{i,C_i}\}$ be given by $r_{\boldsymbol{\mu}_i}$ and $r_{\boldsymbol{\nu}_i}$, respectively. The Kendall's coefficient of rank correlation measures the similarity between $r_{\boldsymbol{\mu}_i}$ and $r_{\boldsymbol{\nu}_i}$. The statistic τ (referred to as Kendall's Tau) is given below in Eq. 3.7, where sign(x) is given by Eq. 3.8. Here, $\tau = 1$ is a perfect relationship, $\tau = 0$ is no relationship at all, and $\tau = -1$ is a perfect negative relationship.

$$\tau = \frac{2}{n(n-1)} \sum_{k<j} \text{sign}(\mu_{i,k} - \mu_{i,j}) \cdot \text{sign}(\nu_{i,k} - \nu_{i,j}) \tag{3.7}$$

$$\text{sign}(x) = \begin{cases} +1 & \text{if x} > 0 \\ 0 & \text{if x = 0} \\ -1 & \text{if x} < 0 \end{cases} \tag{3.8}$$

- **Levenshtein Distance [101]:**

  For a given layer i with C_i outputs channels, let the rank orderings of 2 sets of importance estimates $\boldsymbol{\mu}_i = \{\mu_{i,1}, \cdots, \mu_{i,C_i}\}$ and $\boldsymbol{\nu}_i = \{\nu_{i,1}, \cdots, \nu_{i,C_i}\}$ be given by $r_{\boldsymbol{\mu}_i}$ and $r_{\boldsymbol{\nu}_i}$, respectively. The Levenshtein distance between these two sequences of ranks, which we denote as lev$(r_{\boldsymbol{\mu}_i}, r_{\boldsymbol{\nu}_i})$, is defined as the minimum number of single element edits required to change $r_{\boldsymbol{\mu}_i}$ to $r_{\boldsymbol{\nu}_i}$. The distance is formally defined using recursion as given by Eq. 3.9 and Eq. 3.10. The function tail(r) of an ordered set of n elements returns all but the first element of the string such that $r = \{r^{(1)}, \cdots, r^{(n)}\}$ and tail$(r) = \{r^{(2)}, \cdots, r^{(n)}\}$, where we denote element j of ranked set $r_{\boldsymbol{\mu}_i}$ as $r^{(j)}_{\boldsymbol{\mu}_i}$. Here, lev$(r_{\boldsymbol{\mu}_i}, r_{\boldsymbol{\nu}_i})$ will have a low value close

to 0 if r_{μ_i} and r_{ν_i} are very similar, else it will have a high value.

$$\text{lev}(r_{\mu_i}, r_{\nu_i}) = \begin{cases} \text{length}(r_{\mu_i}) & \text{if length}(r_{\nu_i}) = 0 \\ \text{length}(r_{\nu_i}) & \text{if length}(r_{\mu_i}) = 0 \\ \text{lev}\left(\text{tail}(r_{\mu_i}), \text{tail}(r_{\nu_i})\right) & \text{if } r_{\mu_i}^{(1)} = r_{\nu_i}^{(1)} \\ 1 + f(r_{\mu_i}, r_{\nu_i}) & \text{otherwise} \end{cases} \tag{3.9}$$

$$f(r_{\mu_i}, r_{\nu_i}) = \min\left(\text{lev}(\text{tail}(r_{\mu_i}), r_{\nu_i}), \text{lev}(r_{\mu_i}, \text{tail}(r_{\nu_i})), \text{lev}(\text{tail}(r_{\mu_i}), \text{tail}(r_{\nu_i}))\right) \tag{3.10}$$

To perform our evaluation for a discriminative task, we train LeNet5 [100] models to classify MNIST [98] images. For a generative task, we train convolutional variational autoencoders (VAEs) to generate MNIST images[1]. For each case, we independently train 30 models using different random seeds. Our 30 LeNet5 models have an average test accuracy of 98.2% with a standard deviation of 0.003%, and our 30 VAEs have a mean Fréchet inception distance (FID) [70] of 9.81 with a standard deviation of 0.725. In Figure 3.3, we provide sampled images generated from a randomly selected VAE. These metrics and images are provided as supporting evidence that we use fully trained models in our statistical analysis, which form the basis of our conclusions in this section.

(a) Original Images **(b)** Generated Images

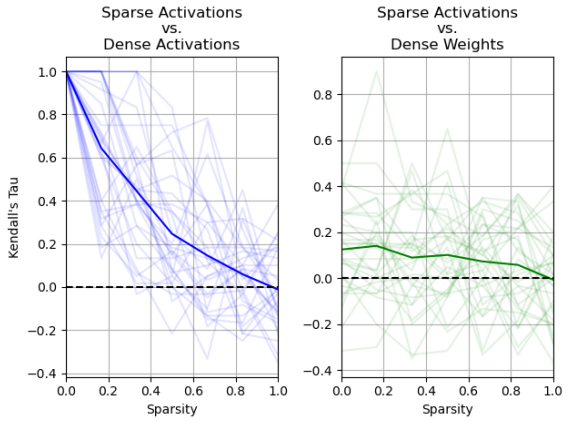
Figure 3.3. We provide random images generated from one of our VAEs trained on MNIST [98].

For LeNet5, we evaluate the importance of the 16 output channels of the second layer when iteratively pruning its 6 input channels. For the VAE, we evaluate the importance of the 16

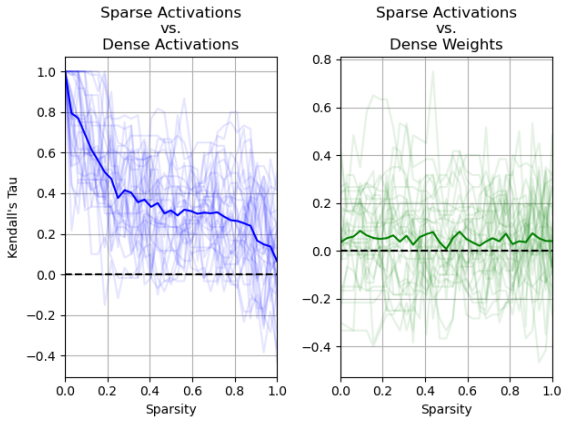
[1]We use a network architecture comparable to LeNet for the purpose of a reasonably symmetric evaluation across discriminative and generative tasks. Our encoder uses 2 convolution layers, each followed by a ReLU and max pooling layer. Our decoder uses 3 deconvolution layers, each followed by a ReLU except for the final deconvolution layer, which is followed by a sigmoid.

output channels of the second layer of our decoder, which has 32 input channels. We increase the sparsity of the input channels using the rank ordering of the original unpruned network as determined by the mean ℓ_1-norm of its activations when estimated over the test dataset. We denote the importance estimate of the preceding layer as $\boldsymbol{\mu}_{i-1}$ and the respective rank ordering as $\boldsymbol{r}_{\boldsymbol{\mu}_{i-1}}$. Note that $\boldsymbol{\mu}_{i-1} = \{\mu_{i-1,1}, \cdots, \mu_{i-1,6}\}$ for LeNet5 and $\boldsymbol{\mu}_{i-1} = \{\mu_{i-1,1}, \cdots, \mu_{i-1,32}\}$ for our VAE. For each level of input sparsity, which we denote as s, we then rank order the output channels using the mean ℓ_1-norm over their respective activations also estimated over the test dataset. We denote the rank ordering of output channel activations by importance measure $\boldsymbol{\mu}_i$ for a given sparsity level (s) as $\boldsymbol{r}_{\boldsymbol{\mu}_i|s}$. Note that this sparsity level is over the input of our activation distribution, not the layer being evaluated, and the importance estimates of the evaluated layer i is $\boldsymbol{\mu}_i = \{\mu_{i,1}, \cdots, \mu_{i,16}\}$ for both LeNet5 and our VAE.

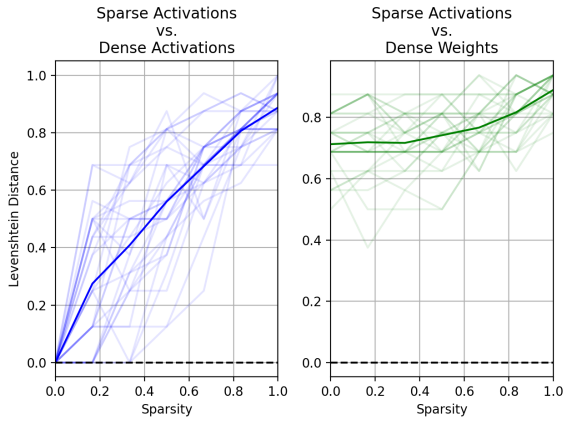
To evaluate the interdependence between successive layers, we iteratively increase the sparsity (s) of the input activations according to $\boldsymbol{\mu}_{i-1}$ for both our discriminative and generative models, and measure the coefficient Kendall's Tau τ and Levenshtein distance between $\boldsymbol{r}_{\boldsymbol{\mu}_i|s}$ (*i.e.*, the rank order of the output channels of layer i by importance measure $\boldsymbol{\mu}_i$ when input activations $\boldsymbol{h}_{i-1}$ have a sparsity of $s > 0$) and $\boldsymbol{r}_{\boldsymbol{\mu}_i}$ (*i.e.*, the rank order of output channels of layer i by importance measure $\boldsymbol{\mu}_i$ when input activations have *no* sparsity). We visualize the results in Figure 3.4. For brevity in the titles of our plots, we use *sparse activations* to refer to the rank order of the output channels of layer i when using the mean ℓ_1-norm of the output channel activations as the importance measure $\boldsymbol{\mu}_i$ when input activations $\boldsymbol{h}_{i-1}$ have a sparsity of s (*i.e.*, $\boldsymbol{r}_{\boldsymbol{\mu}_i|s}$); alternatively, we use *dense activations* when input activations $\boldsymbol{h}_{i-1}$ have no sparsity (*i.e.*, $\boldsymbol{r}_{\boldsymbol{\mu}_i}$). We use *dense weights* to refer to the rank order of the output channels of layer i when using the ℓ_1-norm of the channel weights as the importance measure $\boldsymbol{\nu}_i$ (*i.e.*, $\boldsymbol{r}_{\boldsymbol{\nu}_i}$). We observe that as the input activation sparsity s increases from 0 to 100%, the correlation between $\boldsymbol{r}_{\boldsymbol{\mu}_i|s}$ and $\boldsymbol{r}_{\boldsymbol{\mu}_i}$ gradually decreases and the Levenshtein distance gradually increases. We repeat this process using the ℓ_1-norm of our channel weights as our measure of importance, which we will denote as $\boldsymbol{\nu}_i$; however, the ℓ_1-norm of our channel weights is a data-free measure that is by definition



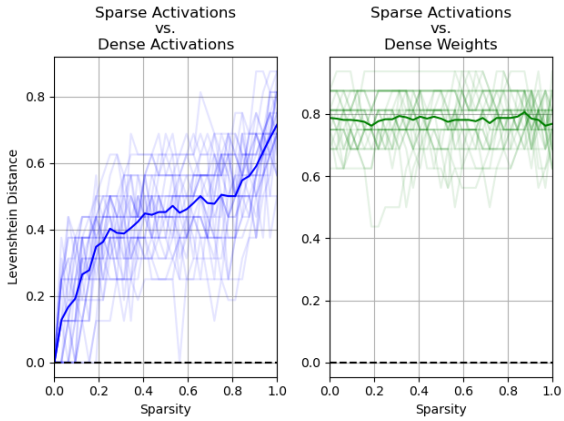
(a) LeNet5 Correlation Evaluation



(b) VAE Correlation Evaluation



(c) LeNet5 Distance Evaluation



(d) VAE Distance Evaluation

Figure 3.4. As discussed in Section 3.3, we evaluate Kendall's coefficient of rank correlation (top row) and the Levenshtein distance (bottom row) over 30 independently trained discriminative models (left column) and generative models (right column). We plot the correlation and distance between $\boldsymbol{r}_{\boldsymbol{\mu}_i}$ and $\boldsymbol{r}_{\boldsymbol{\mu}_i|s}$ in **blue**, and the correlation and distance between $\boldsymbol{r}_{\boldsymbol{\mu}_i|s}$ and $\boldsymbol{r}_{\boldsymbol{\nu}_i}$ in **green**.

invariant to this sparsity injection. Therefore, when iteratively increasing the sparsity of the input activations according to $\boldsymbol{\nu}_{i-1}$, we measure the relationship between our data-driven rankings $\boldsymbol{r}_{\boldsymbol{\mu}_i|s}$ and data-free ranking $\boldsymbol{r}_{\boldsymbol{\nu}_i}$. We observe that the correlation between $\boldsymbol{r}_{\boldsymbol{\mu}_i|s}$ and $\boldsymbol{r}_{\boldsymbol{\nu}_i}$ is much weaker, and their relationship is not as severely affected by the sparsity of the input activations.

From these experiments, we draw the following conclusions. First, data-driven channel importance rankings (*e.g.*, the rankings of output channels by the ℓ_1-norm of activation distri-

butions) are heavily impacted by the sparsity of the preceding input activation distribution[2]. Given this, we hypothesize that we can increase the effectiveness of data-driven pruning criterion by allowing a given layer to adjust to shifts in its input activation *before* applying pruning. Second, data-driven and data-free channel importance measurements are weakly correlated. Therefore, when used to heuristically guide channel pruning algorithms, we expect that these two importance measurements will behave differently under extreme levels of sparsity. It is important to note that we cannot conclude which ranking is necessarily "correct", as these experiments only demonstrate that they are "different". In Section 3.5, we evaluate the performance of these neuron importance measurements using various pruning schedules to provide further insights.

3.4 Algorithms

For the purpose of describing our pruning and quantization algorithms, we first introduce our notation. We denote the input data to a neural network with L layers as $\boldsymbol{x}$ and, for discriminative tasks, its associated output label as $\boldsymbol{y}$. We denote the activations of the network as $\{\boldsymbol{h}_i\}_{i=1}^{L}$, where $\boldsymbol{h}_i$ denotes the activations of hidden layer i and $\boldsymbol{h}_0 = \boldsymbol{x}$ for convenience. The set of weights of each layer are denoted as $\{\boldsymbol{w}_i\}_{i=1}^{L}$, where $\boldsymbol{w}_i$ is the set of weights for layer i with C_i output channels, which we refer to as neurons. Finally, we denote the binary mask used to prune each layer as $\{\boldsymbol{m}_i\}_{i=1}^{L}$ and the importance estimates of each neuron as $\{\boldsymbol{\mu}_i\}_{i=1}^{L}$, where $m_{i,c} \in \{0,1\}$ and $\mu_{i,j} \in \mathscr{R}^{+}$ are respectively a scalar value for the binary mask and non-negative importance estimate for each neuron c in layer i, where $c \in \{1, \cdots, C_i\}$.

3.4.1 Layerwise Channel Pruning using Non-parametric Statistics

When viewing the input data $\boldsymbol{x}$ as a random variable, neural networks are sometimes interpreted as Markov chains, where every hidden layer i defines the conditional probability $p(\boldsymbol{h}_i|\boldsymbol{h}_{i-1})$ [37, 150]. Such a formulation motivates our observations that the effectiveness of neuron importance measurements relying on $\boldsymbol{h}_i$ is dependent on the stability of input activation

[2]We repeated these experiments using the ℓ_0-norm of the activation distribution as see very similar results.

Algorithm 1. Our proposed layerwise channel pruning algorithm using per-channel ℓ_1-norm of activations to measure importance. All channel masks $\{\boldsymbol{m}_i\}_{i=1}^{L}$ are initialized to 1 and all importance measurements $\{\boldsymbol{\mu}_i\}_{i=1}^{L}$ are initialized to 0. We update learned weights of all layers in the network $\{\boldsymbol{w}_i\}_{i=1}^{L}$ every step using backpropagation, but only update the mask $\boldsymbol{m}_i$ for layer i at step i.

Input: $s :=$ sparsity target, $t_p :=$ number of evaluation steps, $\Delta_p :=$ update frequency
Output: $\{\boldsymbol{m}_i\}_{i=1}^{L} :=$ sparsity mask m_i for layers $i \in \{1, \cdots, L\}$

$\{\boldsymbol{m}_i\}_{i=1}^{L} \leftarrow$ initializeMasksToOne()
$\{\boldsymbol{\mu}_i\}_{i=1}^{L} \leftarrow$ initializeImportanceEstimatesToZero()
for $i \in \{1, \cdots, L\}$ **do**
  for $t \in \{1, \cdots t_p\}$ **do**
    networksForwardPass()
    for $c \in \{1, \cdots, C_i\}$ **do**
      $\mu_{i,c} \leftarrow (\mu_{i,c} \cdot (t-1) + \|\boldsymbol{h}_{i,c}\|_1)\,/t$
    networkBackwardPass()
    if $\text{mod}(i, \Delta_p) = 0$ **then**
      for $c \in \{1, \cdots, C_i\}$ **do**
        if $m_{i,c} < \text{quantile}(\boldsymbol{\mu}_i, s)$ **then**
          $m_{i,c} \leftarrow 0$

distribution $\boldsymbol{h}_{i-1}$. In Section 3.3, we show that iteratively increasing the sparsity of the input activations of a given layer results in a monotonic decorrelation between the rank orderings of its output activations when using sparse versus dense input activations. Thus, we hypothesize that by allowing a given layer to adjust to these abrupt shifts in its input activation distribution $\boldsymbol{h}_{i-1}$ *before* pruning $\boldsymbol{h}_i$, we can increase the effectiveness of the heuristics used to rank order neurons by importance. As such, we design an iterative channel pruning algorithm that greedily traverses the topology of a feedforward neural network starting from the first hidden layer $\boldsymbol{h}_1$ and ending at the final hidden layer $\boldsymbol{h}_L$.

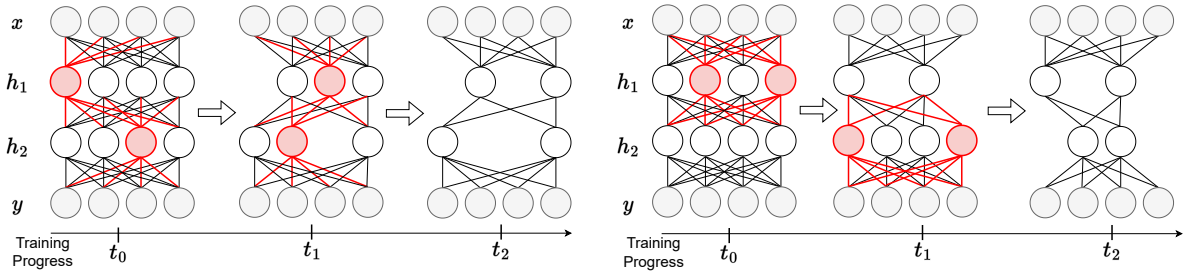
Our layerwise iterative channel pruning algorithm is summarized in Algorithm 1. We first initialize the values of all masks $\{\boldsymbol{m}_i\}_{i=1}^{L}$ to 1 and the values of all importance measurements $\{\boldsymbol{m}_i\}_{i=1}^{L}$ to 0. Note that the mask for a given neuron $m_{i,c}$ is a binary value that prunes the neuron when $m_{i,c} = 0$, and $\mu_{i,c}$ is a non-negative value where $\mu_{i,1} > \mu_{i,2}$ is interpreted as neuron 1 being more important than neuron 2 in layer i. When pruning layer i, we estimate the importance of each neuron c using the moving average of the ℓ_1-norm of activations over channel c from $\boldsymbol{h}_i$

over t_p steps. We evaluate our pruning mask every Δ_p steps, where $\Delta_p \leq t_p$, at which point we prune neurons according to the $\boldsymbol{\mu}_i$ using a pre-defined sparsity target s_T. The entire network is pruned to the given target sparsity once all L layers are processed. As is standard practice, we do not prune visible activations (*i.e.*, the input and output activations $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively). In practice, we apply this procedure on a pre-trained network, and continue to fine-tune our structured subnetwork for T more epochs after pruning. As we greedily increase the sparsity of the network layer-by-layer, we refer to this iterative pruning schedule as "layerwise" channel pruning. Throughout our algorithm, we continue to update the weights of each layer $\{\boldsymbol{w}_i\}_{i=1}^{L}$ using gradient descent such that all weights are updated for $L \cdot t_p + T$ gradient steps.

The concept of using an iterative pruning schedule to alleviate the impact of abruptly removing neurons has been explored in prior work [32, 66, 105, 191]; however, they iteratively introduce sparsity globally at each step. Because these pruning schedules gradually introduce sparsity according to a per-step heuristic, we refer to this class of algorithms as "stepwise" pruning. We visualize the differences in these algorithms in Figure 3.5. Unlike our layerwise pruning schedule, which determines $\boldsymbol{\mu}_i$ only after $\boldsymbol{h}_{i-1}$ has stabilized, stepwise pruning schedules determine $\boldsymbol{\mu}_i$ and $\boldsymbol{\mu}_{i-1}$ jointly. As discussed in Section 3.3, this could lead to less effective measures of neuron importance. In Section 3.5, we compare our layerwise pruning schedule against the standard stepwise approach.

3.4.2 Uniform Quantization-Aware Training

To train our structured subnetworks for low-precision quantization, we use the straight-through estimator (STE) [13] to allow gradients to permeate our rounding function such that $\nabla_x \lfloor x \rceil = 1$, but $\lfloor x \rceil \neq x$. To apply asymmetric quantization to our hidden activations $\{\boldsymbol{h}_i\}_{i=1}^{L}$, we adaptively fit our per-tensor scaling factor s_i and zero-point z_i. To apply symmetric quantization to our weights $\{\boldsymbol{w}_i\}_{i=1}^{L}$, we adaptively fit our per-channel scaling factor $\boldsymbol{s_i}$ where $s_{i,c}$ denotes the scaling factor for channel $c \in \{1, \cdots, C_i\}$. For each layer i, we estimate the upper and lower bounds of activation $\boldsymbol{h}_i$ and the maximum magnitude of weight $\boldsymbol{w}_i$ using moving average



(a) Stepwise Pruning Schedule

(b) Layerwise Pruning Schedule

Figure 3.5. We depict the differences between the standard stepwise pruning schedule (left) and our layerwise pruning schedule (right). While stepwise pruning algorithms iteratively increase sparsity globally in the network with each step, our layerwise pruning algorithm iteratively increases sparsity layer-by-layer with each step. Throughout the training progress (horizontal flow), gray nodes denote visible neurons that are not pruned, red nodes denote hidden neurons that have been identified to be pruned in a given step, and white nodes denote hidden neurons that remain active.

statistics, similar to our technique in Algorithm 1. Following the work of [92], we summarize the adaptive asymmetric and symmetric quantization algorithms used in this paper in Algorithms 2 and 3, respectively.

Algorithm 2. Our adaptive asymmetric quantization algorithm for our activations $\boldsymbol{h}_i$ using per-tensor scaling factors. We use moving average statistics over hidden activation $\boldsymbol{h}_i$ to estimate the bounds on its dynamic range for the purpose of deriving scaling factorss_i and zero-point z_i for layer i.

Input: $\left(l_i^{(t)}, u_i^{(t)}\right)$:= estimated bounds on hidden activation $\boldsymbol{h}_i$ at time step t

Output: Quantized activation $\hat{\boldsymbol{h}}_i$; Updated bounds $\left(l_i^{(t+1)}, u_i^{(t+1)}\right)$

$l_i^{(t+1)} \leftarrow \left(l_t \cdot t + \min\left(\boldsymbol{h}_i\right)\right)/(t+1)$

$u_i^{(t+1)} \leftarrow \left(u_t \cdot t + \max\left(\boldsymbol{h}_i\right)\right)/(t+1)$

$s_i \leftarrow (u_i^{(t+1)} - l_i^{(t+1)})/2^b$

$z_i \leftarrow \lfloor -l_i^{(t+1)}/s \rceil$

$\hat{\boldsymbol{h}}_i \leftarrow \text{clip}\left(\lfloor \boldsymbol{h}_i/s_i \rceil + z_i; 0, 2^b - 1\right)$

Algorithm 3. Our adaptive symmetric quantization algorithm for the set of weights $\boldsymbol{w}_i$ for layer i using per-channel scaling factors. We use moving average statistics to estimate the maximum weight magnitude for each channel c to derive our per-channel scaling factors s_c.

Input: $s_{i,c}^{(t)} :=$ estimated scaling factor s for channel c of weight $\boldsymbol{w}_i$ at time t

Output: Quantized weight $\hat{\boldsymbol{w}}_i$; Updated scaling factor $s_{i,c}^{(t+1)}$

for $c \in \{1, \cdots, C_i\}$ **do**

$$s_{i,c}^{(t+1)} \leftarrow \left(s_{i,c}^{(t)} \cdot t + \frac{\max\left(\boldsymbol{w}_i\right)}{2^{b-1}} \right) / \left(t+1\right)$$

$$\hat{\boldsymbol{w}}_i \leftarrow \text{clip}(\lfloor \boldsymbol{w}_i / s_{t+1} \rceil; -2^{b-1}, 2^{b-1}-1)$$

3.5 Experiments

In Section 3.3, we motivate our hypothesis that, by allowing a given layer to adjust to abrupt shifts to its input distribution *before* pruning its channels, the heuristics used to rank order neurons by importance become more effective. Here, we compare our greedy layerwise channel pruning algorithm against alternative pruning schedules and importance measures to demonstrate the increased performance of our approach versus existing baselines. For the purpose of enabling inference acceleration on resource-constrained hardware such as mobile and edge devices, we combine the use of our channel pruning algorithm with moving average statistics-based uniform quantization, as described in Section 3.4. We evaluate the performance of our algorithms using the following discriminative and generative computer vision tasks:

1. Image classification with DenseNet121 [79], and MobileNetV2 [142] on CIFAR100 [93]
2. Semantic segmentation with UNet [136], and FRRNet [130] on Cityscape [36]
3. Image style transfer with CycleGAN [190] on Cityscape

We implement our pruning algorithms using the PyTorch deep learning framework [127], and create a custom neural network pruning module to compute neuron importance estimates and derive binary masks using local buffers[3]. For each task, our baselines are built using the existing

[3]The code for the algorithms discussed in this paper can be found at https://github.com/mlzxy/mdpi2022.

open-source implementations provided by the original respective authors. To the extent possible, we use the same weight initialization techniques, network architecture, and hyperparameter configurations as reported in the original papers. All models are trained through Nautilus, a distributed research compute infrastructure from the Pacific Research Platform (PRP) [155]. Single-GPU training time for each model ranges from 3 hours (*e.g.*, MobileNetV2 on CIFAR100) to 12 hours (*e.g.*, CycleGAN on Cityscape) depending on the complexity of the task. Furthermore, for the style transfer task, CycleGAN uses a *cycle consistency loss* that includes two mappings: (1) generator G_θ, parameterized by θ, that maps input image $\boldsymbol{x}$ to the target domain $\hat{\boldsymbol{x}}$; and (2) generator F_ϕ, parameterized by ϕ, that maps $\hat{\boldsymbol{x}}$ back to the original image $\boldsymbol{x}$. During training, we only prune the forward mapping into the target domain $G_\theta(\boldsymbol{x})$, as we are only interested in accelerating inference, and the reverse transform is discarded at inference time. Given an input label map, the forward mapping G_θ of our pruned and quantized CycleGAN model generates realistic 128 $\times$ 128 images in the first-person driving view, as shown in Figure 3.6.

Input | Output (Baseline) | Output (P50%) | Output (P75%) | Output (P50%, W8A8) | Output (P75%, W8A8)

Figure 3.6. We provide examples of images generated from CycleGAN given the same input (left). Here, P50% and P75% denote 50% and 75% channel pruning, respectively. We use "W8A8" to denote that the weights and activations have both been quantized to 8 bits.

3.5.1 Evaluating Pruning Schedules and Importance Measures

Here, we evaluate the importance of our layerwise iterative pruning schedule by comparing it to three alternative pruning schedules:

1. **Training from scratch.** Prior work has demonstrated that in some cases, there is no need to implement a pruning schedule because pre-defined structured subnetwork architectures

can be trained from scratch to match or surpass the performance of the original larger network [105]. As such, we evaluate our pruning algorithm against this baseline.

2. **One-shot.** We compare against the common "prune-then-fine-tune" strategy [78], where we train a fully connected baseline, and then prune the converged model to our target sparsity in one step before fine-tuning to heal the network.

3. **Stepwise.** Unlike one-shot pruning schedules, which jump to the target sparsity in one step, stepwise pruning schedules iteratively increase the sparsity in the network over many steps throughout training. We benchmark against the state-of-the-art iterative pruning schedule proposed by Zhu and Gupta [191].

In Table 3.1, we report the performance of each of these algorithms with target sparsity levels of s_T =50% and s_T =75% across our discriminative and generative tasks. Each entry is the averaged result over 3 independent runs. We use the same learning rate schedule and optimizer for each experiment to ensure an even comparison. We use top-1 accuracy to evaluate image classification models, mean intersection-over-union (mIOU) to evaluate semantic segmentation models, and Fréchet inception distance (FID) [70] to measure the difference between real images and images generated from our CycleGAN. Based on these metrics, we observe that our layerwise channel pruning algorithm performs stronger at more extreme levels of sparsity in a majority of cases.

Next, we compare the performance of data-driven and data-free neuron importance measures when used to guide our greedy layerwise channel pruning algorithm. In Table 3.2, we compare the performance of our algorithm when using the ℓ_1-norm of the channel activations against using the ℓ_1-norm of the channel weights. Although prior work had experimented with using the ℓ_0-norm as a data-driven importance measure [78], we do not compare against this as not all layers in our baselines are followed by a ReLU. We observe that using the ℓ_1-norm of the output activation channels yields superior results than using the ℓ_1-norm of channel weights in a majority of cases. Furthermore, we also observe that using the ℓ_1-norm of channel weights

to guide our greedy layerwise pruning algorithm often yields better results than other pruning schedules reported in Table 3.1.

The results reported in Tables 3.1 and 3.2 suggest that we can increase the effectiveness of non-parametric data-driven pruning criteria by only pruning a given layer *after* its preceding layer has been fully pruned, which supports our motivating hypothesis. Our experiments in Section 3.3 show that the rank ordering of channels for a given layer i is strongly impacted by the sparsity of the input activations $\boldsymbol{h}_{i-1}$ to that layer. Equation-based pruning schedules such as [191] gradually introduce sparsity throughout the network with each step, as shown in Figure 3.5. Thus, at time t, they approximate the neuron importance with respect to the target sparsity s_T using the neuron importance with respect to the current sparsity s_t, where $s_t \leq s_T$ and $t \leq T$. As such, this class of algorithms use rankings $\boldsymbol{r}_{\boldsymbol{\mu}_i|s_t}$ to heuristically guide pruning. Furthermore, one-shot pruning schedules approximate the neuron importance with respect to the target sparsity s_T using estimates at step 0, when the input activations have no sparsity. As such, this class of algorithms use rankings $\boldsymbol{r}_{\boldsymbol{\mu}_i}$ to heuristically guide pruning. In contrast, our layerwise pruning schedule directly uses the neuron importance measure with respect to the target sparsity s_T, where we use $\boldsymbol{r}_{\boldsymbol{\mu}_i|s_T}$ to heuristically guide pruning. As the sparsity target s_T increases to more extreme levels, there is a gradual decorrelation between $\boldsymbol{r}_{\boldsymbol{\mu}_i}$ and $\boldsymbol{r}_{\boldsymbol{\mu}_i|s_T}$, as shown in Section 3.3. Our results given in Table 3.1 show that directly using $\boldsymbol{r}_{\boldsymbol{\mu}_i|s_T}$ within our greedy layerwise framework increases the effectiveness of data-driven rankings. We conjecture this is because our greedy layerwise channel pruning algorithm allows an unpruned layer to freely adjust to abrupt shifts in its input activation distribution caused by pruning the channels of its preceding layer.

3.5.2 Evaluation of Joint Pruning and Quantization

We evaluate our greedy layerwise channel pruning algorithm when jointly used with the quantization algorithms detailed in Section 3.4.2 to learn low-precision structured subnetworks through joint pruning and quantization. As discussed in Section 3.2, we apply symmetric,

Table 3.1. Comparing pruning schedules across discriminative and generative tasks. With higher levels of sparsity, our greedy layerwise channel pruning algorithm performs better than existing baselines across both discriminative and generative tasks. Note that for top-1 accuracy and mIOU higher is better, but for FID lower is better.

		Classification (Accuracy)		Segmentation (mIOU)		Style Transfer (FID)
		DenseNet121	MobileNetV2	UNet	FRRNet	CycleGAN
Baseline	Full Model	78.70	68.31	61.69	66.32	47.17
50% Sparsity	From Scratch	76.75	65.36	58.90	61.84	50.39
	One-Shot	78.40	67.39	**60.59**	64.53	49.43
	Stepwise	77.14	**68.05**	58.13	64.90	52.83
	Layerwise (Ours)	**78.77**	67.58	60.57	**65.52**	**48.06**
75% Sparsity	From Scratch	73.07	56.66	**56.71**	59.29	59.55
	One-Shot	76.40	56.16	52.77	58.71	57.07
	Stepwise	72.55	60.40	51.05	56.59	65.67
	Layerwise (Ours)	**76.44**	**60.71**	56.48	**61.14**	**55.48**

Table 3.2. Comparing data-free vs. data-driven neuron importance measures across discriminative and generative tasks using our greedy layerwise channel pruning algorithm. We observe that the mean ℓ_1-norm of the output activations (*i.e.*, "Layerwise (A)") performs better than the ℓ_1-norm of channel weights (*i.e.*, "Layerwise (W)").

		Classification (Accuracy)		Segmentation (mIOU)		Style Transfer (FID)
		DenseNet121	MobileNetV2	UNet	FRRNet	CycleGAN
Baseline	Full Model	78.70	68.31	61.69	66.32	47.17
50% Sparsity	Layerwise (A)	**78.77**	**67.58**	**60.57**	65.52	**48.06**
	Layerwise (W)	77.88	67.56	59.41	**66.16**	52.52
75% Sparsity	Layerwise (A)	**76.44**	**60.71**	**56.48**	**61.14**	**55.48**
	Layerwise (W)	73.53	60.62	54.38	60.18	67.97

per-channel quantization to the weights of our deep neural networks and asymmetric, per-tensor quantization to the activations. To implement Algorithms 2 and 3, we create another custom quantization PyTorch module that accumulates the respective per-tensor or per-channel moving average statistics. As is standard practice, our modules produce a "fake-quantized" output using the derived scale s and zero-point z [82, 92]. We apply these modules to every hidden activation $\{\boldsymbol{h}_i\}_{i=1}^{L}$ and set of weights $\{\boldsymbol{w}_i\}_{i=1}^{L}$ in the network to prepare our models for integer-only inference [82]. We follow the standard "prune-then-quantize" training paradigm and activate the quantization operators only after the completion of layerwise pruning. Figure 3.1 shows both the training schedule and computational order of our joint pruning and quantization pipeline.

In Table 3.3, we report our joint pruning and quantization results for 50% and 75%

sparsity when training 4-bit and 8-bit models. We use "W4A8" to denote quantizing weights to 4 bits and activations to 8 bits. For all networks except for CycleGAN, we quantize the weights of all layers $\{\boldsymbol{w}_i\}_{i=1}^{L}$ to the specified bit width including the first and last layer; and quantize the activations of all layers $\{\boldsymbol{h}_i\}_{i=1}^{L}$ to the specified bit width except for network input $\boldsymbol{x}$ and output $\boldsymbol{x}$. For CycleGAN, we always quantize the weights and input activation of the last layer to 8-bit regardless of global bit width configuration, as is standard practice [149]. For each experiment reported in Table 3.3, we use our greedy layerwise pruning algorithm with the ℓ_1-norm of the output channel activations as our importance measure.

Table 3.3. Joint channel pruning and uniform quantization to learn low-precision structured subnetworks using our algorithms depicted in Figure 3.1 and discussed in Section 3.4.

		Classification (Accuracy)		**Segmentation (mIOU)**		**Style Transfer (FID)**
		DenseNet121	**MobileNetV2**	**UNet**	**FRRNet**	**CycleGAN**
Baseline	Full Model	78.70	68.31	61.69	66.32	47.17
	W8A8	79.04	68.26	62.03	66.43	49.89
50% Sparsity	W8A8	79.01	67.58	60.44	64.81	58.12
	W4A8	78.41	64.67	60.15	63.42	64.25
	W4A4	75.99	59.25	56.12	59.04	92.71
75% Sparsity	W8A8	76.17	61.02	56.15	61.28	85.89
	W4A8	76.12	55.41	55.60	60.90	92.12
	W4A4	73.62	49.49	51.90	54.16	119.19

Finally, we apply our joint layerwise channel pruning and uniform quantization framework to train ResNet-32 [66] on CIFAR10 dataset with sparsity ratios of 25%, 40% and 50%, and uniform bit widths of 8 and 4. We compare against existing solutions and summarize the results in Table 3.4. Here, N_W and N_A denote the average number of bits used to quantize weights or activations, respectively, and s_W and s_A denote the global weight and activation sparsity, respectively. Note that, as discussed in Section 3.2.1, channel pruning reduces both the storage and operating memory[4] requirements of both the weights and activations, while unstructured weight pruning only reduces the memory storage requirements of the weights. Therefore, structured pruning can result in lower operating memory requirements, despite lower compression ratios with respect to

[4]We define operating memory as the aggregate hardware storage area used for weights and activations of the network during inference, all of which are required to be kept so they can be readily accessed.

unstructured pruning. To compare different approaches across these configurations, we compute the performance per operating memory size (*i.e.*, performance density) for each model, as listed in the last column. For existing solutions, we summarize the results reported in prior work. For solutions that do not apply quantization to the weights or activations, we estimate their precision at 16 bits per element rather than 32, since neural networks can be quantized to 16-bit fixed-point through post-training quantization without significant accuracy degradation [135].

Table 3.4 shows that our method is able to achieve a higher performance density (8.92) than the highest of previous solutions (8.59), while having very competitive performance (92.53% versus 91.66% top-1 accuracy). When applying 50% pruning and 4-bit quantization, our method achieves the smallest memory footprint and highest performance density of 25.25, three times larger than the existing highest, while still maintaining sufficient accuracy (87.3%). Moreover, unlike methods such as [156, 175], our framework provides direct control over the target bit width and sparsity, which is more useful in practical scenarios under tight design constraints, as is the case when deployed on edge GPUs or FPGAs [15, 31, 185].

3.6 Conclusions and Future Work

In this work, we demonstrate that, when using data-driven measures of neuron importance, the rank orderings of the output channels of a given layer strongly depend on the sparsity level of the preceding layer. In Section 3.3, we show that as we increase the sparsity of the input activations for a given layer, the correlation trends toward 0 when comparing: (1) the rank order of its output channels using input activations with no sparsity; versus (2) the rank order of its output channels when the input activations have sparsity $s > 0$. Informed by this observation, we propose a data-driven non-parametric, magnitude-based channel pruning algorithm that works in a greedy manner based on the activations of the previous sparsified layer, as detailed in Section 3.4. For the purpose of learning low-precision structured subnetworks, we demonstrate that our proposed algorithm works effectively in combination with statistics-based

Table 3.4. We demonstrate the superior performance-per-memory footprint (*i.e.*, performance density) of our framework when compared to existing image classification solutions trained on CIFAR10. By jointly applying uniform quantization and unstructured pruning to both the weights and activations of the DNN, we achieve a higher performance density (PD) with higher compression rates than existing solutions and comparable network performance.

Method	Network	N_W	N_A	s_W	s_A	Baseline Acc	Accuracy	Weights (Mb)	Activations (Mb)	PD (Acc/Mb)
[29]	ResNet-32	8	–	77.8%	–	92.58	92.64 (+0.06)	0.83	20.19	4.41
[3]	DenseNet-76	2	–	54%	–	92.19	91.17 (-1.02)	0.68	141.26	0.64
[105]	VGG-19	–	–	95%	–	93.50	93.34 (-0.16)	16.03	19.40	2.63
	PreResNet-110	–	–	95%	–	95.04	92.35 (-2.69)	1.38	67.50	1.34
	DenseNet-100	–	–	95%	–	95.24	94.19 (-1.05)	0.97	213.37	0.44
	VGG-19	–	–	70%	70%	93.5	93.60 (+0.1)	70.70	5.31	1.23
	PreResNet-164	–	–	60%	60%	95.04	94.23 (-0.81)	16.67	40.21	1.66
	DenseNet-40	–	–	60%	60%	94.10	93.87 (-0.23)	1.60	22.97	3.82
[187]	DenseNet-40	–	–	60%	60%	94.11	93.16 (-0.95)	1.60	22.97	3.79
	ResNet-20	–	–	38%	38%	92.01	91.66 (-0.35)	2.70	7.96	8.59
	ResNet-56	–	–	45%	45%	93.04	92.26 (-0.78)	7.53	19.18	3.45
	ResNet-110	–	–	63%	63%	93.21	92.96 (-0.25)	10.25	25.12	2.63
[169]	VGG-16	–	–	78.8%	78.8%	93.40	91.50 (-1.90)	49.96	3.75	1.70
[43]	VGG16-C	–	–	95%	–	93.51	93.00 (-0.51)	11.78	17.70	3.15
	WRN-22-8	–	–	95%	–	95.74	95.07 (-0.67)	13.73	115.34	0.74
[175]	ResNet-20	1.9	–	54%	–	91.29	91.15 (-0.14)	0.24	12.85	6.97
[156]	VGG-7	4.8	5.4	–	–	93.05	93.23 (+0.18)	43.85	3.27	1.98
[128]	MobileNet	8	8	–	–	91.31	90.59 (-0.72)	25.74	13.17	2.33
[30]	ResNet-32	8	–	87.5%	–	92.58	92.57 (-0.01)	0.47	20.19	4.48
Ours	ResNet-32	8	8	25%	25%	92.58	92.53 (-0.05)	2.80	7.57	**8.92**
		8	8	40%	40%		91.77 (-0.81)	2.24	6.06	**11.06**
		8	8	50%	50%		90.16 (-2.42)	1.87	5.05	**13.04**
		4	4	50%	50%		87.30 (-5.28)	0.93	2.52	**25.25**

uniform quantization. As demonstrated in Section 3.5, our joint layerwise channel pruning and uniform quantization framework yields increased performance per memory footprint over existing solutions across a range of discriminative and generative networks. The resulting neural networks can be efficiently accelerated by resource-constrained hardware platforms such as edge GPUs and FPGAs. In future work, we aim to focus more closely on applying our framework to generative models and explore advanced quantization-aware training techniques to work jointly with our non-parametric greedy layerwise channel pruning algorithm.

Acknowledgements: This chapter is based on material as it appears in the 2022 MPDI Applied Sciences Special Issue on Hardware-Aware Deep Learning (Xinyu Zhang, Ian Colbert, and

Srinjoy Das, “Learning Low-Precision Structured Subnetworks Using Joint Layerwise Channel Pruning and Uniform Quantization”). The dissertation author was a co-primary investigator and author of this paper. This chapter was supported in part by NSF awards CNS1730158, ACI-1540112, ACI-1541349, OAC-1826967, the University of California Office of the President, and the California Institute for Telecommunications and Information Technology’s Qualcomm Institute (Calit2-QI).

Chapter 4

Unified Quantization and Pruning

As discussed thus far, the power budget for embedded hardware implementations of deep learning algorithms can be extremely tight. To address implementation challenges in such domains, new design paradigms such as approximate computing have drawn significant attention. Approximate computing techniques exploit the innate error-resilience of deep learning algorithms, a property that makes them amenable for deployment on low-power computing platforms. This chapter introduces an approximate computing framework that we refer to as AX-DBN, which is designed for an architecture belonging to the class of stochastic deep learning algorithms known as deep belief networks (DBNs). Specifically, we consider procedures for efficiently implementing the discriminative deep belief network (DDBN), a stochastic neural network that is used for classification tasks, extending approximation computing from the analysis of deterministic to stochastic neural networks. For the purpose of optimizing the DDBN for hardware implementations, we explore the use of: (a) low-precision representations of neurons and functional approximations of activation functions; (b) statistical criticality analysis to identify the nodes in the network which can operate at reduced precision while allowing the network to maintain target accuracy levels; and (c) a greedy search methodology with incremental retraining to determine the optimal reduction in precision for all neurons to maximize power savings. To represent neurons at low precision, our framework jointly considers neuron pruning by interpreting a pruned neuron as being represented with 0 bits. We refer to such a formulation

as a *unified quantization and pruning formulation*. Using the AX-DBN methodology proposed in this chapter, we present experimental results across several network architectures that show significant power savings under a user-specified accuracy loss constraint with respect to ideal full-precision implementations.

4.1 Introduction

In recent years, there has been a significant increase in the use of deep learning algorithms for a variety of applications such as image recognition, text retrieval, and pattern completion [73, 96, 140]. Enabling such algorithms to operate on low-power, real-time platforms such as mobile phones and edge devices is an area of critical interest. On such platforms, a training procedure is typically performed on a cloud server. This training involves optimizing the parameters of a cloud-based neural network using data presented to the device and uploaded to the cloud. Thereafter, the cloud performs classification when requested by the device, and subsequently, the device communicates with the cloud each time an inference task is requested. This purely cloud-based approach to performing inference has a number of drawbacks including high power consumption, latency, security, and reliance on stable and fast internet connectivity, and is therefore not suitable for use on ultra-low-power devices with battery life constraints. Implementing inference locally on embedded hardware is perceived to be a desirable solution to this problem and is the focus of current research [27, 39, 157, 182].

We adopt a shared approach where the cloud performs neural network training while a low-power implementation of the neural network on a local device is used for performing inference, as shown in Figure 4.1. Neural networks are inherently error-resilient [16] and, therefore, high-precision representations and operations are not necessary for sufficiently accurate performance of such algorithms. This property is exploited by approximate computing techniques based on the use of limited precision representations and algorithm-level approximations to achieve energy-efficient implementations with negligible performance loss [157, 182]. In this chapter,

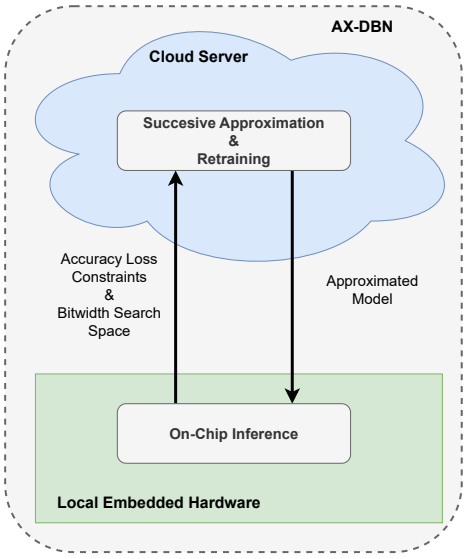


Figure 4.1. The macro-level design of our framework leaves all training, approximation, and retraining to the cloud server. The hardware client sends a request to the cloud server giving it an accuracy loss constraint and a fixed bit width search space. The resulting approximated model is then used for energy-efficient inference on embedded hardware.

we propose AX-DBN, an approximate computing methodology for a class of stochastic neural networks known as discriminative deep belief networks (DDBNs). These are multi-layered extensions of the discriminative restricted Boltzmann machine (DRBM) [96], which can be used for classification in both supervised and semi-supervised settings. Our proposed AX-DBN framework extends the analysis of deterministic networks [157, 182] to the domain of stochastic neural networks. AX-DBN involves training and approximating on the cloud and performing classification on embedded hardware, as depicted in Figure 4.1. In this approach, efficiency arises from exploiting arithmetical and functional approximations subject to a user-specified accuracy loss constraint relative to an ideal full-precision implementation.

Related Prior Work. Previous work for neural network implementations in hardware using approximate computing has focused on feedforward deterministic networks solving classification tasks [157, 182]. Venkataramani *et al.* [157] propose a design-space exploration framework, AxNN, that systematically applies approximation techniques to deterministic neural networks. Using "resilience characterization," their approach identifies and applies limited preci-

sion approximation on neurons whose impact will be the lowest on overall network performance to enable power-optimized classification. Zhang *et al.* [182] follows a similar approach where the formulation of their neuron criticality analysis is applied to identify neurons where approximate multiplier circuits are instantiated to reduce power during classification in feedforward neural networks. Their approximate computing framework ApproxANN, along with their power model, addresses energy savings for memory accesses and computation workloads based on network structures and hardware characteristics.

Contributions of this Paper. Extending the work of [157] and [182] into the stochastic domain, we develop a methodology allowing for the design of an energy-efficient implementation of a stochastic discriminative DBN (DDBN) to balance the trade-off between performance and power.[1] Power efficiency arises from selectively exploiting arithmetical and functional approximations subject to a user-specified accuracy loss constraint. To accomplish this, two different levels of approximation are considered in the hardware implementation of a DDBN: (1) limited precision representation of hidden neurons; and (2) functional approximation of activation functions. The main contributions of our work are as follows:

- The use of criticality analysis to rank order neurons based on their contribution to DDBN network performance. This work carries criticality analysis to the domain of stochastic deep learning algorithms implemented on finite precision digital hardware. Criticality metric driven approximation using cross entropy is compared against random ordering using Monte Carlo simulations to gauge their effectiveness in limited precision network approximation with variable bit widths for individual neurons.

- The use of a greedy retraining procedure to optimize neuron bit widths under given accuracy loss constraints with respect to ideal full-precision implementations.

- The use of a generalized power model for both computation workloads and memory

[1] Although the stochastic DBN can perform both classification and generation, in this paper we analyze its classification properties, leaving an analysis of its generation properties for future work.

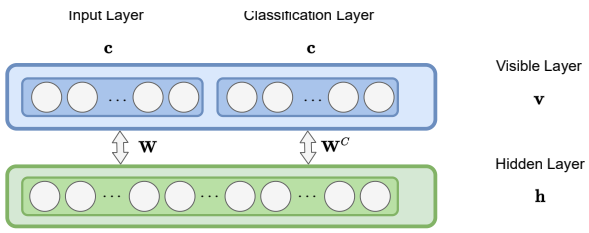


Figure 4.2. The Discriminative RBM is a bipartite graphical model with visible neurons $\boldsymbol{v}$, which consist of input neurons $\boldsymbol{x}$ and classification neurons $\boldsymbol{c}$, that are connected to the hidden neurons $\boldsymbol{h}$ by weights $\mathbf{W}$ and $\mathbf{W}_c$, respectively.

accesses based on fixed point representations of individual neurons, number of samples, and hardware characteristics.

Outline of the Chapter. In Section 4.2 we review discriminative RBMs (DRBMs) and describe how they are stacked to form discriminative DBNs (DDBNs). We also present the network structures that are implemented in this paper. Section 4.3 describes how limited precision and activation function approximation will be performed for the purpose of implementing such networks on hardware. In Section 4.4 we present the mathematics for gradient-based neuron criticality analysis in stochastic neural networks such as DRBMs and DDBNs. In Section 4.5 we outline our AX-DBN hardware design methodology[2]. Section 4.6 outlines the power model used for this work. Section 4.7 describes the accuracy and power results for different DRBM and DDBN architectures using our design methodology and resulting conclusions.

4.2 Discriminative Deep Belief Networks (DDBNs)

Here, we review the mathematical basics of the DRBM, the building block of DDBNs, and present the DRBM and DDBN configurations used for this work.

4.2.1 Discriminative Restricted Boltzmann Machines

A restricted Boltzmann machine (RBM) is a generative stochastic artificial neural network that can learn a probability distribution over its set of inputs. The RBM is a bipartite graph

[2]We note that this method is a significant extension and improvement from our preliminary results reported in [173].

that consists of two layers - one visible layer $\boldsymbol{v}$ where the states of the units in this layer can be observed, and one hidden layer $\boldsymbol{h}$. The probability distribution implemented by the RBM is given by the Boltzmann distribution defined by:

$$P(\boldsymbol{v},\boldsymbol{h}) = \frac{e^{-E(\boldsymbol{v},\boldsymbol{h})}}{\sum\limits_{\boldsymbol{v},\boldsymbol{h}} e^{-E(\boldsymbol{v},\boldsymbol{h})}} \tag{4.1}$$

where the energy function $E(\boldsymbol{v},\boldsymbol{h})$ is given below:

$$E(\boldsymbol{v},\boldsymbol{h}) = -\boldsymbol{v}^T\boldsymbol{b}^v - \boldsymbol{h}^T\boldsymbol{b} - \boldsymbol{v}^T\boldsymbol{W}\boldsymbol{h} \tag{4.2}$$

Here, $\boldsymbol{b}^v$ and $\boldsymbol{b}$ are biases for the visible and hidden layers, respectively, and $\boldsymbol{W}$ is the weight matrix between them [71]. From Eq. 4.1, one can derive the conditional distributions:

$$P(h_j|\boldsymbol{v}) = \sigma(b_j + \sum_i v_i W_{ij}) \tag{4.3}$$

$$P(v_i|\boldsymbol{h}) = \sigma(b_i^v + \sum_j h_j W_{ij}) \tag{4.4}$$

where $\sigma(x)$ is the logistic sigmoid function $1/(1+\exp(-x))$. The training method we use is described in [71], where the contrastive divergence (CD) method is applied to provide weight and bias updates.

To use RBMs as classifiers, Larochelle *et al.* [96] proposed the use of discriminative RBMs (DRBMs), which jointly concatenate the input $\boldsymbol{x}$ and its associated "one-hot" target class $\boldsymbol{c}$ to form the single visible layer $\boldsymbol{v}$, as illustrated in Figure 4.2. The energy function defined by [96] is given below:

$$E(\boldsymbol{x},\boldsymbol{h},\boldsymbol{c}) = -\boldsymbol{x}^T\boldsymbol{b}^x - \boldsymbol{h}^T\boldsymbol{b} - \boldsymbol{c}^T\boldsymbol{b}^c - \boldsymbol{x}^T\boldsymbol{W}\boldsymbol{h} - \boldsymbol{c}^T\boldsymbol{W}^c\boldsymbol{h} \tag{4.5}$$

Here, $\boldsymbol{b}^x$, $\boldsymbol{b}$ and $\boldsymbol{b}^c$ are biases of the visible, hidden, and classification layers, respectively.

$\boldsymbol{W}$, $\boldsymbol{W}^c$ are the weight matrices between layers as shown in Figure 4.2. The inputs $\boldsymbol{x}$ to the DRBM can be either binary or continuous. In the scope of this work, we consider binary inputs $\boldsymbol{x}$ as they can be used for efficient implementations on finite precision hardware with low power and area.

Similar to the RBM, the following conditional distributions can be derived for a DRBM with V visible, H hidden and C class units [96]:

$$P(h_j|\boldsymbol{v}) = P(h_j|\boldsymbol{x},\boldsymbol{c}) = \sigma(b_j + \sum_{i=1}^{V} x_i W_{ij} + \sum_{i=1}^{C} c_i W_{ij}^c) \tag{4.6}$$

$$P(x_i|\boldsymbol{h}) = \sigma(b_i^x + \sum_{j=1}^{H} h_j W_{ij}) \tag{4.7}$$

$$P(c_i|\boldsymbol{h}) = \text{softmax}(b_i^c + \sum_{j=1}^{H} h_j W_{ij}^c) \tag{4.8}$$

where $\sigma(x)$ is the logistic sigmoid function $1/(1+\exp(-x))$ and $\text{softmax}(x_i)$ is given by Eq. 4.9, where $i \in \{1,\cdots,k\}$.

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{k}\exp(x_j)} \tag{4.9}$$

Similar to the RBM, the DRBM can also be trained using contrastive divergence to estimate the network weight and bias values [96]. Using the notation of Larochelle *et al.* [96], the DRBM conditional probability of a class label c_i given input $\boldsymbol{x}$ is

$$P(c_i|\boldsymbol{x}) = \frac{e^{-F(\boldsymbol{x},c_i)}}{\sum_{j=1}^{C} e^{-F(\boldsymbol{x},c_j)}} \tag{4.10}$$

where $F(\boldsymbol{x},c_i)$ denotes the free energy of the DRBM given input $\boldsymbol{x}$ and class label c_i, as defined below:

$$F(\boldsymbol{x},c_i) = -b_{c_i}^c - \sum_{j=1}^{H} \log\left(1+\exp(b_j + W_{jc_i}^c + \sum_{i=1}^{V} W_{ji}x_i)\right) \tag{4.11}$$

A trained DRBM can be used to perform classification using two equivalent methods [72]:

- *Free Energy:* From Eq. 4.10, it can be seen that for a given visible vector $\boldsymbol{x}$ the class $\boldsymbol{c}$ that

has the highest probability of activation corresponds to the minimum value of $F(\boldsymbol{x}, c_i)$. Therefore, in this method, the free energy for a given $\boldsymbol{x}$ is calculated for all labels c_i. The classification result is the class c_i that corresponds to the minimum free energy.

- *Gibbs Sampling:* The binary activation state of each class can also be found by repeatedly sampling all class neurons of the DRBM for a given visible vector. The class with the highest activation frequency given a sufficient number of sampling iterations is the classification result.

The Gibbs sampling method, with suitable approximations for the sigmoid function, is more amenable to a realization on digital hardware than free energy due to the highly nonlinear dependence of free energy on its input values. Therefore in this work, we use Gibbs sampling for inference in our hardware implementations of DRBMs. Details of the sigmoid function approximation are outlined in Section 4.3.

4.2.2 Discriminative Deep Belief Networks

Deep belief networks (DBNs) are probabilistic generative models which learn to extract a deep hierarchical representation from the training data. Hinton *et al.* [72] show that DBNs are stacked RBMs and can be learned in a greedy manner by sequentially learning RBMs. Using contrastive divergence, the first layer is trained as an RBM with the input of the DBN as its input layer and, after the first RBM is trained, the weights $\boldsymbol{W}^1$ are fixed, as well as representations of $\boldsymbol{h}^1$. The binary states of the first hidden unit layer are then used as inputs training the second RBM for $\boldsymbol{W}^2$. Iterating this way layer-by-layer, the DBN can be trained by greedily training RBMs.

Similar to the DRBM, a DBN can also be used as a discriminative model by concatenating a classification layer to the final hidden layer, as shown in Figure 4.3. Discriminative DBNs (DDBNs) can be used to perform classification using either free energy or Gibbs sampling, as described for DRBMs. With the free energy method, we compute the activation probability of all

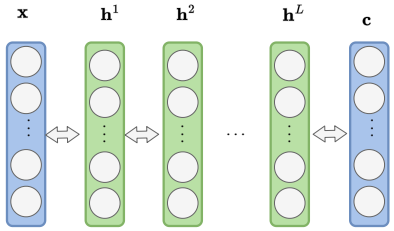


Figure 4.3. The DDBN architecture consists of stacked RBMs with a DRBM at the classification layer. Similar to the DRBM, the visible neurons $\boldsymbol{v}$ are split into input neurons $\boldsymbol{x}$ and class neurons $\boldsymbol{c}$. Each hidden layer $\boldsymbol{h}^l$, where $l \in \{1, \cdots, L\}$, is greedily trained layer by layer [72].

hidden units across layers. Following this, the classification result is given by the class $\boldsymbol{c}$ that has the minimum free energy across all classes based on the activation values from the last hidden layer L. Using Gibbs sampling, the binary activation state of each class given a visible vector $\boldsymbol{x}$ is found by repeated sampling of all hidden neurons across all layers and class neurons. The correct classification result is given by the class with the highest activation frequency. Similar to the DRBM, our hardware implementations of DDBNs use Gibbs sampling for on-chip classification.

4.2.3 Network Architecture

We use the Discriminative DBN to perform classification on the MNIST dataset [98], which contains 60k training samples and 10k test samples of 28x28 gray-scale handwritten digits. We binarize the images using a fixed threshold of 0.5. Throughout this paper, our DRBM and DDBN naming conventions denote the number of neurons in each hidden layer bottom-up, *e.g.*, DDBN-100-200 denotes a 2-layer DDBN with 100 neurons in the first hidden layer and 200 neurons in the second hidden layer. Our analysis in this paper is based on the following architectures:

- 300-neuron budget architectures: DRBM-300, DDBN-100-200
- 600-neuron budget architectures: DRBM-600, DDBN-100-200-300

4.3 Limited Precision and Function Approximation

Discriminative DBNs implemented with full-precision weights and biases and nonlinear sigmoidal activation functions cannot be directly implemented for inference on finite-precision digital hardware platforms. Therefore, we study the effect of fixed-point representations and approximate sigmoid functions on DDBN classification performance. These approximations form the basis of our AX-DBN framework used for the implementation of DDBNs in finite precision embedded hardware.

4.3.1 Limited Precision Approximation

Fixed-point is preferred to floating-point representation because the latter requires extensive area and power for digital hardware implementations [109]. As shown in Figure 4.4, there are three places in the computation where fixed-point variables can be used to map the network onto its limited precision (LP) version:

- LP1 limits the precision of weight $\boldsymbol{W}$ and bias $\boldsymbol{b}$
- LP2 limits the precision of the result $\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}$
- LP3 limits the precision of $\sigma(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b})$

4.3.2 Sigmoid Function Approximation

Approximate implementations and comparisons of sigmoid activation functions designed for digital hardware exist in the literature [9, 151]. Here we use PLAN, a piecewise linear approximation of the sigmoid function proposed by Amin *et al.* [9] in our design

$$y = \begin{cases} f(x) & x > 0 \\ 1 - f(|x|) & x \leq 0 \end{cases} \tag{4.12}$$

where

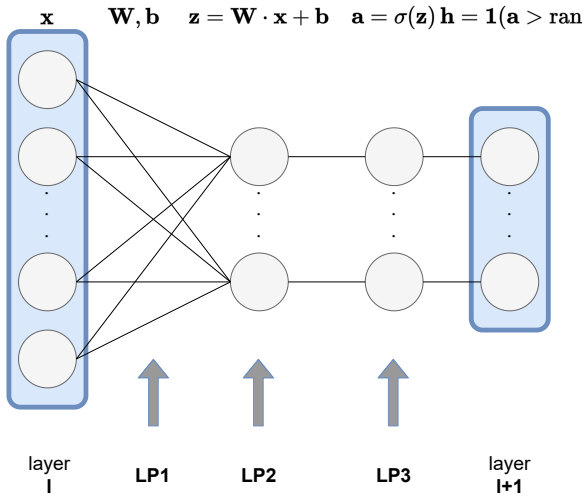


Figure 4.4. We exploit limited-precision (LP) approximations at various stages in the DBN.

$$f(x) = \begin{cases} 1 & x > 5 \\ 0.03125x + 0.84375 & 2.375 < x \leq 5 \\ 0.125x + 0.625 & 1 < x \leq 2.375 \\ 0.25x + 0.5 & 0 \leq x \leq 1 \end{cases}$$

Only addition and shift operations are involved in the approximation given by Eq. 4.12, which allows for a relatively inexpensive implementation in digital hardware with minimal accuracy loss. Therefore, we use the PLAN approximation of the sigmoid function in our work. To motivate this decision, we show that the PLAN approximation achieves almost the same performance as the original sigmoid function.

Table 4.1. The classification error at various approximation levels.

bits	full	64	32	16	8	4
Original	95.56	95.58	94.9	94.7	94.1	8.9
PLAN	95.55	94.7	94.8	94.7	93.9	10.3

4.4 Neuron Ordering using Criticality Analysis

A neuron is said to be *critical* if a network's performance is significantly degraded by random noise injected on said neuron, otherwise, it is said to be *resilient* [182]. The performance of a neural network is invariant to lowering the precision of, or even removing, resilient neurons, whereas it is significantly degraded otherwise. In the AxNN [157] and ApproxANN [182] design methodologies, which use deterministic feedforward networks, the Euclidean distance between the classification output and true label is used as the loss function for both training and criticality analysis. The magnitude of the average loss over all training samples is used to characterize the criticality of each neuron.

For the case of stochastic neural networks, such as DRBMs and DDBNs, in this paper we perform the following steps for criticality analysis:

- After performing stochastic learning, we estimate the criticality of individual neurons using a gradient-based backpropagation approach as outlined in [157] and [182]. However, unlike deterministic models, DRBMs and DDBNs are fundamentally stochastic models which can in principle be used for both discrimination and generation tasks. Keeping this perspective of a fully probabilistic framework, we propose the use of cross entropy as a loss function for determining critical neurons using gradient-based backpropagation.

- Inference in DRBMs and DDBNs is performed by Gibbs sampling as described in Section 4.2 for ease of implementation in digital hardware. However, for criticality analysis we represent the binary states of the neurons in these networks with their activation probabilities, which are continuous between 0 and 1.

In the cross entropy (CE) loss function, y_j and a_j^c denote the one-hot ground truth and softmax class prediction probabilities, respectively.

$$\mathbb{L}_{\text{CE}} = -\sum_j y_j \ln\left(a_j^c\right)$$

The derivative of a given loss function with respect to the value of a hidden neuron relates that neuron's contribution to classification error caused by bit width and functional approximations at that neuron. Using CE, the error sensitivity of loss due to corruption of neuron j in layer $\boldsymbol{h}^L$ (denoted as h_j^L) of a DDBN with L layers is defined in Eq. 4.13, where a_i^c is defined by the softmax output in Eq. 4.8 and $z_i^L = b_i^c + \sum_j h_j^L W_{ij}^c$.

$$\begin{aligned}\frac{\partial \mathbb{L}_{\text{CE}}}{\partial h_j^L} &= \sum_i \frac{\partial \mathbb{L}}{\partial a_i^c}\frac{\partial a_i^c}{\partial z_i^L}\frac{\partial z_i^L}{\partial h_j^L} \\ &= \sum_i (-\frac{y_i}{a_i^c}) \cdot \left(\frac{\partial \text{softmax}(z_i^c)}{\partial z_i^c} \frac{\partial \sum_k (h_k^L W_{ik}^c + b_l^c)}{\partial h_j^L} \right) \\ &= \sum_i (-\frac{y_i}{a_i^c}) \cdot a_i^c (\mathbb{1}_{i=j} - a_j^c) W_{ij}^c\end{aligned} \tag{4.13}$$

The error sensitivity of neuron j in hidden layers $\boldsymbol{h}^l$, where $l < L$, is computed by Eq. 4.14. The binary states of individual neurons h_i^l are approximated by their sigmoid output a_i^l. Here, $\sigma(z_i^l) = 1/(1+\exp(-z_i^l))$.

$$\begin{aligned}\frac{\partial \mathbb{L}_{\text{CE}}}{\partial h_j^l} &= \sum_i \frac{\partial \mathbb{L}}{\partial a_i^{l+1}}\frac{\partial a_i^{l+1}}{\partial h_j^l} \\ &= \sum_i \frac{\partial \mathbb{L}}{\partial a_i^{l+1}} \cdot \sigma(z_j^{l+1})(1-\sigma(z_j^{l+1}))W_{ij}^{l+1}\end{aligned} \tag{4.14}$$

For k samples of training data, the criticality score for neuron j in hidden layer $\boldsymbol{h}^l$ with $N_{\boldsymbol{h}^l}$ neurons, where $l \in \{1, \cdots, L\}$, is given by:

$$\text{criticality}(h_j^l) = \frac{\mathbb{E}[h_j^l]}{\frac{1}{N_{\boldsymbol{h}^l}} \sum_{i=1}^{N_{\mathbf{h}^l}} \mathbb{E}[h_j^l]} \tag{4.15}$$

where $\mathbb{E}[h_j^l] = \frac{1}{k} \sum_{\mathbf{x},\mathbf{y} \in \text{data}} \left| \frac{\partial \mathbb{L}_{\text{CE}}}{\partial h_j^l} \right|$.

The hidden neurons are then ranked in order from least to most critical by the magnitude of these obtained scores.

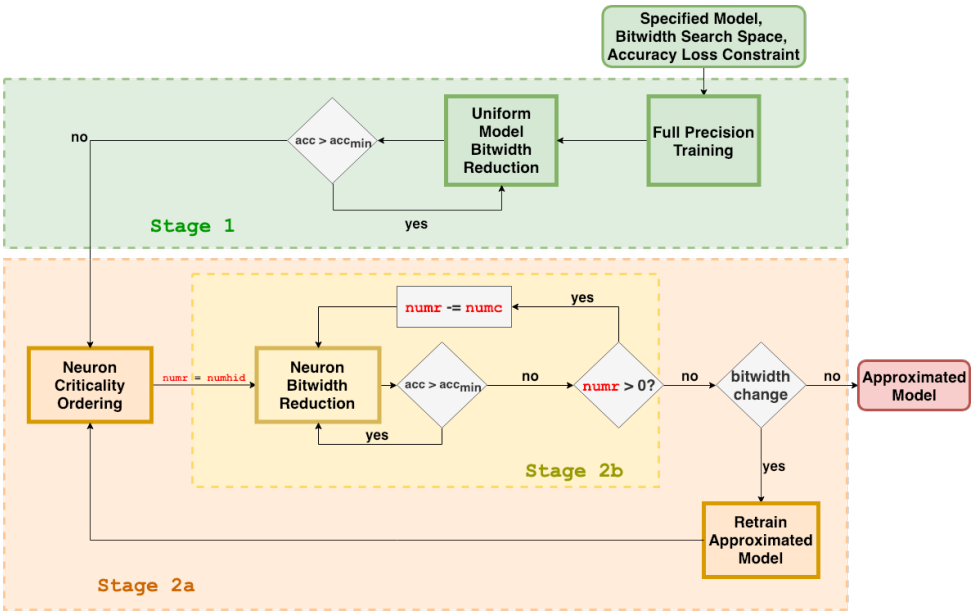


Figure 4.5. The approximation algorithm can be broken into two stages: (1) uniform bit width reduction and (2) neuron criticality analysis with systematic bit width reduction and retraining. The algorithm takes in a specified model, a set bit width search space, and an accuracy loss constraint as inputs, then returns an approximated energy-efficient model. All algorithm hyperparameters are shown in red. The number of hidden neurons is given by **numhid**, the number of resilient neurons to approximate is given by **numr**, and the search step size is given by **numc**.

4.5 AX-DBN Design Methodology

The design methodology illustrated by Figure 4.1 is composed of two parts: (1) a cloud-based training and approximation process that optimizes the precision of individual neurons for inference on hardware; (2) inference performed locally on embedded hardware.

4.5.1 Cloud-based Model Training and Approximation

The approximation algorithm is a scalable framework that can be applied to any fully connected network of variable width and depth. Figure 4.5 illustrates the approximation flow chart, starting with a user-specified accuracy loss constraint with respect to a full-precision implementation and a set of allowed bit widths for a specified DRBM/DDBN model then ending with its approximated counterpart. Algorithm 4 formally describes this approximation procedure

and can be summarized by two stages:

1) Full-precision training and uniform bit width reduction

2) Neuron bit width reduction using (a) neuron criticality analysis and retraining and (b) limited precision neuron approximation

Stage 1 first trains the specified model at full precision then subsequently reduces bit width uniformly across all weights and biases until it can no longer maintain the accuracy loss constraint. Stage 2 first analyzes and rank orders hidden neurons according to a given criticality metric, then individually approximates these neurons based on the criticality ranking and, finally retrains the limited precision model to allow for further network approximation. In our experiments, hidden neurons can be represented at 64-bit $Q8.56$, 16-bit $Q8.8$, 12-bit $Q6.6$, 8-bit $Q4.4$, or 4-bit $Q1.3$ precision.[3] Here, we use $Qm.n$ to denote the fixed-point precision format with m integral bits (including sign bit), and n fractional bits. Additionally, the algorithm can prune the neuron completely by representing the bit width as 0-bit $Q0.0$ precision. Class neurons are represented at 16-bit $Q8.8$ precision and not considered in our optimization due to their relatively minor impact on power savings when compared to that arising from approximating the larger number of hidden neurons considered in our architectures.

4.5.2 Inference on Embedded Hardware

After the cloud-based approximation procedure is completed, we download the limited precision weights and biases onto embedded hardware and perform Gibbs sampling for local inference, as described in Section 4.2. As depicted in Figure 4.6, our hardware implementations use a pipelined architecture consisting of $L+1$ stages with one stage for each of the L hidden layers and a classification layer, respectively.

[3]We found that these $Qm.n$ bit widths gave the best accuracy when approximating the models uniformly.

Algorithm 4. The AX-DBN approximation algorithm greedily explores neuron bit width distributions given a specified model (model_0), an accuracy constraint (acc_{min}), and a set bit width search space (bw). Algorithm hyper-parameters include the desired neuron step size (numc), the number of approximated resilient neurons (numr), and the total number of hidden neurons (numhid). The summed difference in all neuron bit widths between successive criticality-retraining iterations is given by Δsumbit.

Input: model_0 := specified model, acc_{min} := min accuracy, bw := bit width search space
Output: model_{ax} := approximated model

```
model ← fullprecisionTraining(model_0)
model_ax ← uniformApprox(model)
while Δsumbit > 0 do
    order ← criticalityScore(model_ax)
    numr ← numhid
    while numr > 0 and acc > acc_min do
        model_ax ← neuronApprox(model_ax, numr, order)
        numr ← numr − numc
    model_ax ← retrain(model_ax)
```

4.6 Power Model for Compute and Memory Requirements

To perform inference on a DDBN with x visible units, c class units, k data samples, and L hidden layers each with N_{h_l} hidden neurons, a chip needs to read network parameters and input data from memory and to write classification results to memory. Each weight or bias is represented by an q-bit fixed-point number, each input sample is a x-dim binary vector, and each classification result is represented by a c-dim binary vector. In this model, q refers to $m+n$ in a $Qm.n$ representation and the number of neurons at q-bit precision in hidden layer $\boldsymbol{h}^l$ is given by $N_{h_l}^q$.

The power for off-chip to on-chip data transfer DT is given by Eq. 4.16, where the power consumption for reading and writing 1 bit of data is given by A. The computation workload CW is defined by Eq. 4.17 in terms of multiply-accumulate operations, where power consumption for a q-bit ALU operation is given by B_q. A is obtained by modeling in CACTI [95]. We calculate B_q as the average accumulator power consumption using Verilog implementations of our specified DRBM and DDBN architectures at q-bit precision with Synopsys EDA tools using a 65 nm

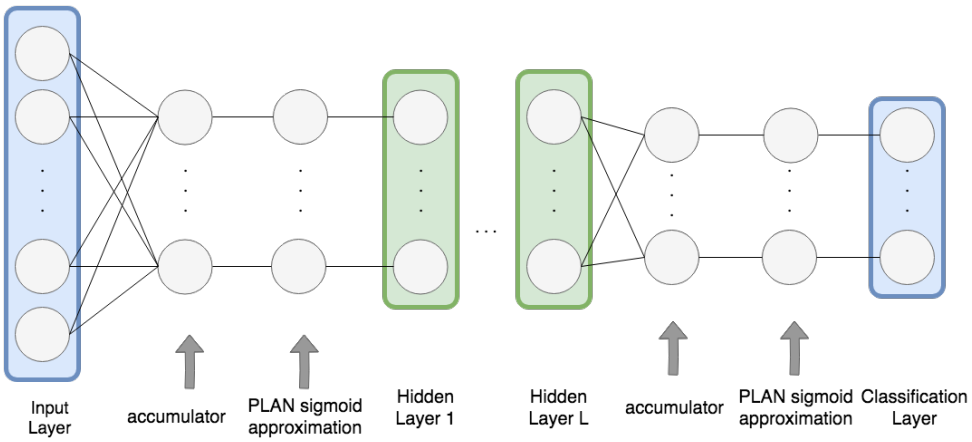


Figure 4.6. In our hardware implementations, we use Gibbs sampling to propagate binary activation states from the input layer $\boldsymbol{x}$ to the classification layer $\boldsymbol{c}$ using the PLAN approximation for each sigmoid output. Each hidden layer performs random sampling using a pseudo-random pattern generator (PRPG) implemented in hardware.

standard cell library. We estimate the total power consumed by an AX-DBN approximated model to be the combination of its data transfer and computation workload, i.e $DT + CW$. In our implementation, q takes the following values: $\{4, 8, 12, 16, 64\}$.

4.7 Experimental Results

An effective criticality metric should be able to accurately determine hidden neurons that can be approximated using lower precision representations while maintaining specified classification accuracy above a desired threshold. Our simulations show that the criticality measure based on cross entropy (CE) achieves this objective. Owing to the stochasticity of our models, we run 200 Monte Carlo iterations to obtain distributions of power savings and average neuron bit width. A different random seed is used in each Monte Carlo iteration to initialize the weights and biases of a new model and train it using contrastive divergence (CD) on the MNIST dataset. We then compare the performance of criticality metric-driven realizations versus that of random ordering.

We explore the effectiveness of our AX-DBN framework across the DRBM and DDBN architectures presented in Section 4.2.3. For an even comparison, we fix the total number

of neurons as we increase the depth of the network (*i.e.*, DDBN-100-200 and DRBM-300) will both have a fixed budget of 300 neurons. In our experiments, we first train each network using contrastive divergence on the MNIST dataset [98], we then perform our criticality-driven approximations on each network, and finally evaluate the approximated model's performance. Reducing the bit width of all neurons uniformly results in significant accuracy degradation, as shown in Table 4.2. With each median and mean, we provide the inter-quartile range (IQR) and standard deviation in parenthesis, respectively. Using criticality-driven neuron approximation and pruning, we are able to significantly reduce the average neuron bit width while maintaining user-specified accuracy loss constraints with respect to ideal full-precision implementations, as shown in Tables 4.3 and 4.4.

Figures 4.7, 4.8, 4.9, and 4.10 visualize the power savings with respect to a 64-bit implementation, the distribution of neuron bit widths, and average neuron bit width for 1% and 5% accuracy loss constraints for different network architectures. We observe that approximated networks realized using CE-driven critical neuron determination yield higher average power savings and lower average neuron bit widths when compared to those realized using random neuron ordering. Overall, CE is an effective metric for hidden neuron criticality determination across accuracy loss constraints and stochastic network architectures.

$$DT = A\left((x+1)\sum_{q=1}^{64} qN_{h_1}^q + \sum_{l=1}^{L-1}(N_{h_l}+1)\sum_{q=1}^{64}(qN_{h_{(l+1)}}^q) + 16(N_{h_L}+1)c + k(x+c)\right) \quad (4.16)$$

$$CW = k\left((x+1)\sum_{q=1}^{64}(B_q N_{h_1}^q) + \sum_{l=1}^{L-1}(N_{h_l}+1)\sum_{q=1}^{64} B_q N_{h_{(l+1)}}^q + B_{16}(N_{h_L}+1)c\right) \quad (4.17)$$

4.8 Conclusions

In this paper, we propose a systematic approximation methodology for stochastic discriminative restricted Boltzmann machines (DRBMs) and discriminative deep belief networks (DDBNs) to optimize power consumption subject to the constraint of maintaining user-specified

classification accuracy. This work extends criticality analysis from the domain of deterministic neural networks to the realm of stochastic networks and introduces a unified formulation of quantization and pruning by representing a pruned neuron as a 0-bit limited-precision approximation. Our procedure involves two key steps: (1) The use of criticality analysis to rank order neurons based on their contribution to network performance; (2) The use of greedy retraining to optimize neuron bit widths under accuracy constraints. Our results show that cross entropy can be used as an effective metric for determining neuron criticality in stochastic network approximation and yields lower average neuron bit width representations as well as higher savings in power consumption when compared to random criticality ordering of neurons. Our future research will extend our methodology to optimize the power consumption of hardware implementations of generative neural networks for purposes such as image generation, denoising, and infilling.

Acknowledgements: This chaper is based on material as it appears in the 2019 International Joint Conference on Neural Networks (Ian Colbert, Kenneth Kreutz-Delgado, and Srinjoy Das, "AX-DBN: An approximate computing framework for the design of low-power discriminative deep belief networks"). The dissertation author was the primary investigator and author of this paper. This chapter was supported in part by NSF awards CNS- 1730158, ACI-1540112, ACI-1541349, OAC-1826967, the University of California Office of the President, and the California Institute for Telecommunications and Information Technology's Qualcomm Institute (Calit2-QI). Thanks to CENIC for the 100Gpbs networks. We would also like to thank Professor Andrew Kahng, Jonas Chan, and Chih-Yin Kan at UC San Diego for providing assistance with Verilog implementations and power measurements.

Table 4.2. We run 200 Monte Carlo iterations to measure the median (top) and mean (bottom) test accuracy of each architecture when reducing the bit width of each neuron uniformly. With each median and mean, we provide the IQR and standard deviation in parenthesis, respectively.

Architecture	4-bit	8-bit	12-bit	16-bit	64-bit
DRBM-300	54.5 (5.3)	88.6 (1.1)	94.0 (0.3)	94.3 (0.2)	94.3 (0.3)
DDBN-100-200	73.7 (4.7)	94.3 (0.5)	95.9 (0.2)	95.9 (0.2)	96.0 (0.2)
DRBM-600	54.2 (4.2)	82.7 (3.0)	95.0 (0.2)	95.3 (0.2)	95.3 (0.2)
DDBN-100-200-300	58.8 (7.4)	93.0 (0.8)	95.6 (0.2)	95.6 (0.2)	95.9 (0.2)
DRBM-300	54.6 (3.8)	88.5 (0.8)	94.0 (0.2)	94.3 (0.2)	94.3 (0.2)
DDBN-100-200	74.0 (3.8)	94.3 (0.4)	95.9 (0.1)	95.9 (0.2)	95.9 (0.1)
DRBM-600	54.0 (3.0)	82.5 (2.3)	95.0 (0.2)	95.3 (0.2)	95.3 (0.2)
DDBN-100-200-300	58.7 (5.3)	92.8 (0.7)	95.6 (0.2)	95.6 (0.2)	95.9 (0.2)

Table 4.3. We run 200 Monte Carlo iterations to measure the median (top) and mean (bottom) test accuracy and average neuron bit width of each architecture when approximating with AX-DBN given a 1% accuracy loss constraint. With each median and mean, we provide the IQR and standard deviation in parenthesis, respectively. The classification accuracy of the full-precision (FP) model on the test dataset is given in the first column. FP denotes the full-precision 64-bit floating-point model.

Architecture	FP	Random	CE
DRBM-300	94.3 (0.3)	93.7 (0.7) | 11.1-bit (0.4)	93.6 (0.5) | 10.2-bit (0.6)
DDBN-100-200	96.0 (0.2)	95.0 (0.2) | 9.63-bit (0.9)	94.8 (0.2) | 8.72-bit (0.9)
DRBM-600	95.3 (0.2)	94.7 (0.8) | 11.3-bit (0.3)	94.5 (0.6) | 9.68-bit (0.7)
DDBN-100-200-300	95.9 (0.2)	94.8 (0.4) | 10.3-bit (0.6)	94.8 (0.3) | 8.72-bit (0.7)
DRBM-300	94.3 (0.2)	93.8 (0.5) | 11.1-bit (0.3)	93.7 (0.4) | 10.2-bit (0.4)
DDBN-100-200	95.9 (0.1)	95.0 (0.2) | 9.57-bit (0.7)	94.8 (0.2) | 8.71-bit (0.7)
DRBM-600	95.3 (0.2)	94.9 (0.5) | 11.2-bit (0.3)	94.6 (0.4) | 9.66-bit (0.5)
DDBN-100-200-300	95.9 (0.2)	94.8 (0.2) | 10.3-bit (0.5)	94.8 (0.2) | 8.71-bit (0.5)

Table 4.4. We run 200 Monte Carlo iterations to measure the median (top) and mean (bottom) test accuracy and average neuron bit width of each architecture when approximating with AX-DBN given a 5% accuracy loss constraint. With each median and mean, we provide the IQR and standard deviation in parenthesis, respectively. The classification accuracy of the full-precision (FP) model on the test dataset is given in the first column. FP denotes the full-precision 64-bit floating-point model.

Architecture	FP	Random	CE
DRBM-300	94.3 (0.3)	91.8 (1.5) | 8.00-bit (0.6)	92.4 (1.0) | 6.46-bit (1.0)
DDBN-100-200	96.0 (0.2)	92.4 (1.0) | 6.91-bit (0.4)	92.5 (0.8) | 5.92-bit (0.5)
DRBM-600	95.3 (0.2)	93.5 (1.1) | 9.05-bit (0.6)	94.4 (0.7) | 6.86-bit (1.1)
DDBN-100-200-300	95.6 (0.2)	92.4 (0.9) | 7.36-bit (0.4)	92.3 (0.8) | 6.13-bit (0.4)
DRBM-300	94.3 (0.2)	91.7 (0.9) | 8.09-bit (0.4)	92.2 (0.8) | 6.52-bit (0.7)
DDBN-100-200	95.9 (0.1)	92.4 (0.6) | 6.89-bit (0.3)	92.4 (0.6) | 5.89-bit (0.3)
DRBM-600	95.3 (0.2)	93.7 (0.7) | 9.10-bit (0.5)	94.3 (0.6) | 6.81-bit (0.7)
DDBN-100-200-300	95.9 (0.2)	92.3 (0.7) | 7.38-bit (0.3)	92.2 (0.6) | 6.15-bit (0.3)

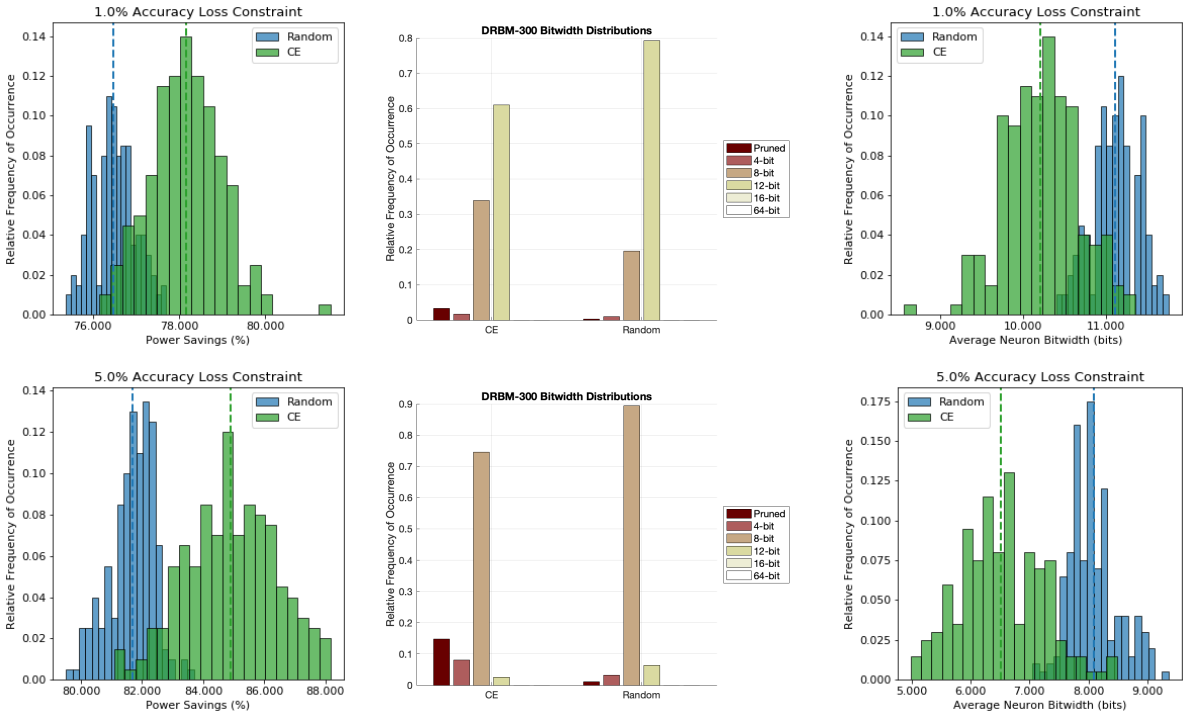


Figure 4.7. DRBM-300: Relative power savings (left), neuron bit width distribution (middle), and average neuron bit widths (right).

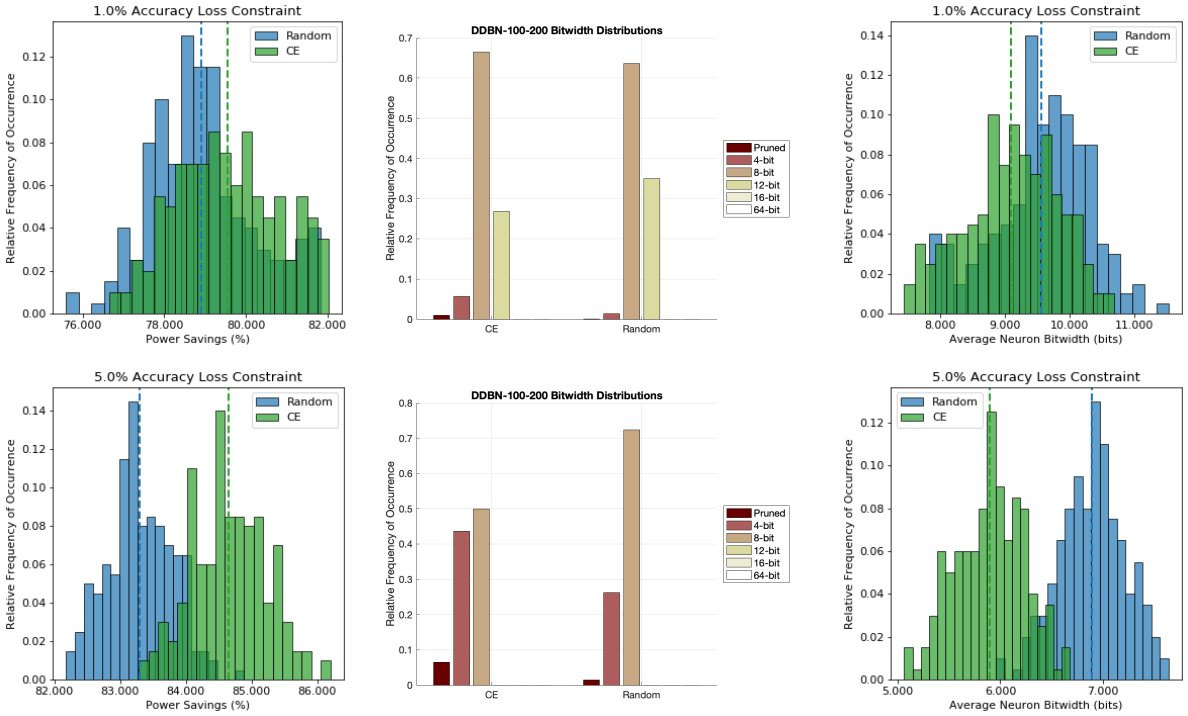


Figure 4.8. DDBN-100-200: Relative power savings (left), neuron bit width distribution (middle), and average neuron bit widths (right).

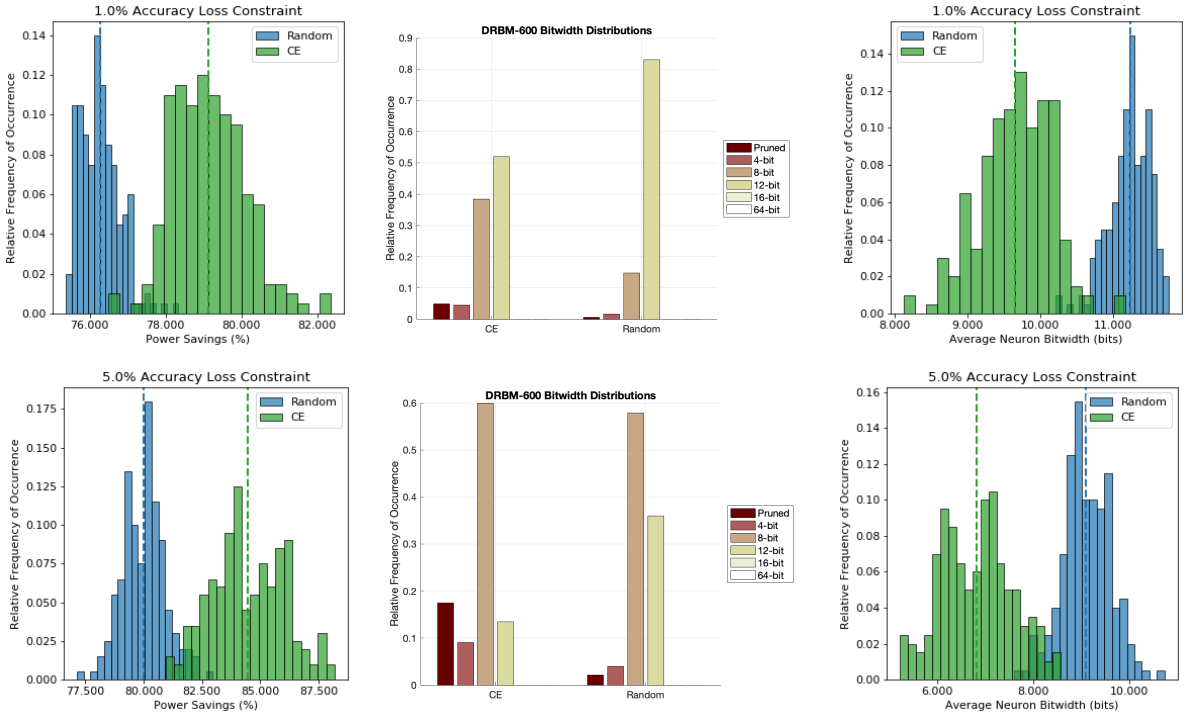


Figure 4.9. DRBM-600: Relative power savings (left), neuron bit width distribution (middle), and average neuron bit widths (right).

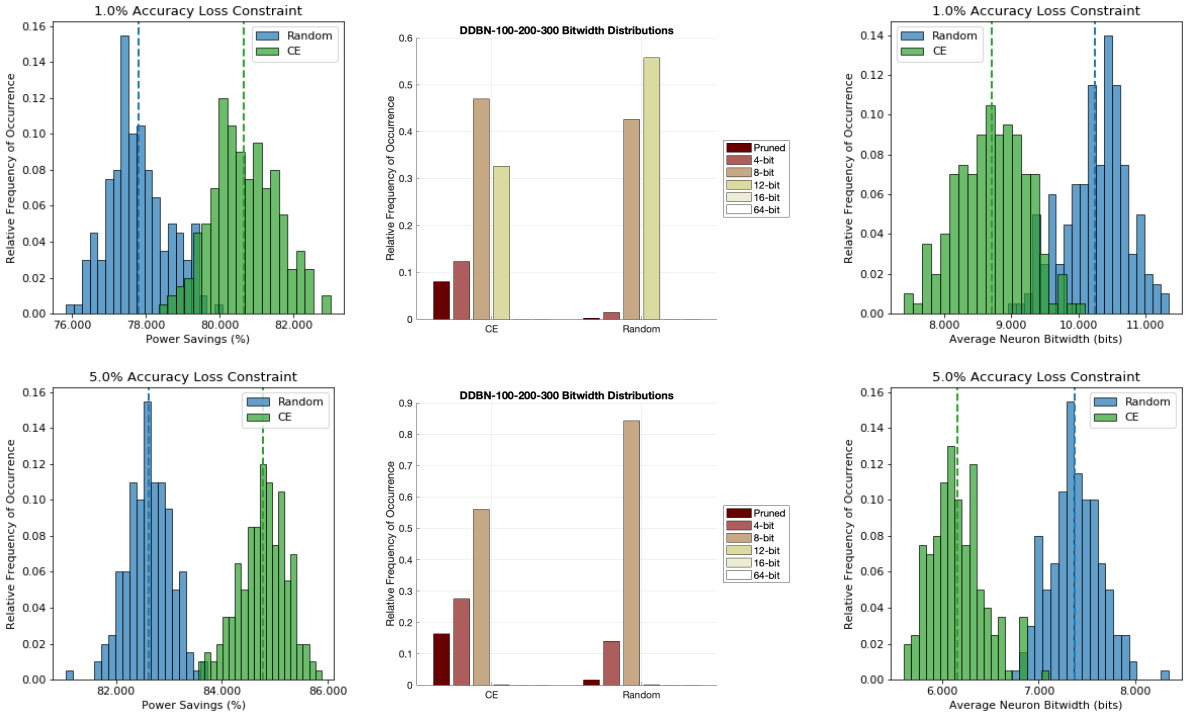


Figure 4.10. DDBN-100-200-300: Relative power savings (left), neuron bit width distribution (middle), and average neuron bit widths (right).

Chapter 5

Improving Inference Algorithms for Image Upsampling

Many prominent computer vision problems involve image upsampling (*e.g.*, super resolution [144], scene segmentation [106], pose estimation [147], and image generation [57]). In this chapter, our focus is on the core algorithms used to build deep learning solutions to such problems, namely deconvolution [48, 180], resize convolution [45, 124], and sub-pixel convolution [6, 144]. Previous works comparing these algorithms focus on improving training by increasing image fidelity and optimizing sample efficiency [6, 124, 144, 145]. In contrast, this chapter is focused on comparing the *inference* properties of these algorithms. Our analysis is followed by a novel deconvolution inference algorithm designed to optimize image upsampling latency and energy efficiency without sacrificing the image fidelity obtained from training. We first provide a comprehensive survey and analysis of these state-of-the-art deep learning-based image upsampling algorithms. We then introduce a novel deconvolution inference algorithm that exposes more opportunities for concurrent execution and improves its adaptability to limited resource availability as is common of edge computing platforms.

5.1 Introduction

Many solutions to important computer vision and image processing applications require increasing the number of pixels per unit area (or resolution) by inferring values in high-

dimensional image spaces from low-dimensional representations. This process, commonly referred to as image upsampling, is a one-to-many mapping used to predict, generate, or recover information to increase dimensionality [144, 164]. In contrast, downsampling is a many-to-one mapping used to encode features and reduce dimensionality [164]. Image upsampling frameworks based on deep learning typically rely on convolution-based and/or interpolation-based algorithms to increase resolution:

- **Interpolation-based upsampling algorithms** infer the value of pixels for which there are no sample points using only local information such as nearest pixel value [120]. This class of techniques includes algorithms such as nearest neighbor and bilinear interpolation.
- **Convolution-based upsampling algorithms** also infer the value of pixels for which there are no sample points but predict, generate, or recover high-frequency information by learning spatial correlations through training strategies [144]. This class of techniques includes algorithms such as deconvolution, sub-pixel convolution, and resize convolution, which are commonly used in end-to-end deep learning solutions [164, 166].

The majority of state-of-the-art deep learning solutions for image upsampling rely on either sub-pixel or resize convolution [6, 38, 45, 124, 144, 186]. However, these algorithms require memory-intensive feature map transformations at each inference pass. In this chapter, we review commonly used convolution-based image upsampling algorithms. We provide insights into the pros and cons of each algorithm and introduce a novel deconvolution inference algorithm that further improves concurrency. Our study motivates the contributions later detailed in Chapter 6, where we characterize the data movement costs of these algorithms and introduce kernel transformations that exploit the functional equivalence of sub-pixel and resize convolutions to deconvolution.

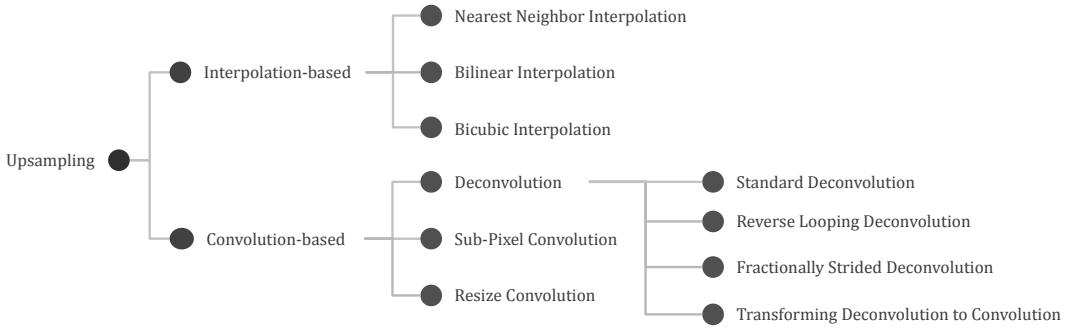


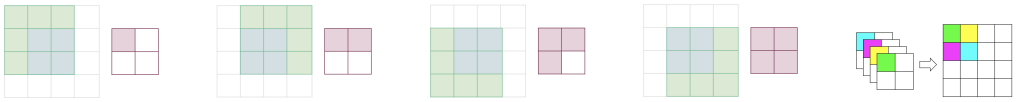
Figure 5.1. We provide an "image upsampling taxonomy." State-of-the-art deep learning solutions for image upsampling rely on convolution-based and/or interpolation-based algorithms to increase the resolution of images; however, only the convolution-based algorithms are learnable.

5.2 Sub-Pixel Convolution

The sub-pixel convolution was introduced by Shi *et al.* [144] to upsample images using a fully convolutional neural network and is used as the standard method for upsampling images in deep learning solutions [164, 166]. As shown in Figure 5.2, the algorithm is executed as two serialized operations: (1) a same-padded convolution followed by (2) a pixel shuffle. However, convolution (see Algorithm 5) is inherently a downsampling operation. To upsample by a factor of r, the sub-pixel convolution first generates r^2 more output channels using a same-padded convolution to then feed the resulting output into the pixel shuffle (see Algorithm 6) to be reshaped. Aitken *et al.* [6] show that, when properly initialized, a network trained using the sub-pixel convolution converges faster with lower image reconstruction error than other convolution-based upsampling algorithms. However, the sub-pixel convolution requires pixel shuffle post-processing for every inference pass. As further discussed in Chapter 6, this memory-intensive feature map transformation severely limits time and energy efficiency at inference.

5.3 Resize Convolution

The resize convolution was introduced by Dong *et al.* [45] and popularized by Odena *et al.* [124] to address checkerboard artifacts that can arise during training. As depicted in Figure 5.3,



(a) t_0 **(b)** t_1 **(c)** t_2 **(d)** t_3 **(e)** PS

Figure 5.2. We visualize an example sub-pixel convolution, where a 2×2 input (in blue) is convolved with a 3×3 kernel (in green) using a stride $S = 1$ and padding $P = 1$ to create the 2×2 output (in red) before upsampling the image by a factor of 2 using the pixel shuffle (PS).

Algorithm 5. Standard Convolution

for $o_c = 0$, o_c++, while $o_c < O_C$ **do**
  for $o_h = 0$, o_h++, while $o_h < O_H$ **do**
    for $o_w = 0$, o_w++, while $o_w < O_W$ **do**
      for $i_c = 0$, i_c++, while $i_c < I_C$ **do**
        for $k_h = 0$, k_h++, while $k_h < K$ **do**
          for $k_w = 0$, k_w++, while $k_w < K$ **do**
            $i_h \leftarrow S \times o_h + k_h - P$
            $i_w \leftarrow S \times o_w + k_w - P$
            $x \leftarrow \text{input}[i_c, i_h, i_w]$
            $w \leftarrow \text{kernel}[o_c, i_c, k_h, k_w]$
            $\text{output}[o_c, o_h, o_w] \leftarrow x \times w$

the resize convolution is executed as two serialized operations, similar to the sub-pixel convolution. To upsample by a factor of r, the resize convolution first uses (1) an interpolation-based upsampling algorithm before applying (2) a same-padded convolution in the higher resolution space [124]. Standard implementations for resize convolution rely on nearest neighbor interpolation (NN-interpolation) as opposed to bilinear or bicubic [89, 124, 164]. Similar to sub-pixel convolution's pixel shuffle, the interpolation-based pre-processing is a memory-dominated algorithm required for every inference pass. However, unlike the sub-pixel convolution, the same-padded convolution is executed in a higher dimensional space where the time and energy costs are much higher [46, 144]. Even so, the sub-pixel and resize convolutions have the same compute requirements when upsampling an image by a factor of r. As further discussed in Chapter 6, the interpolation-based pre-processing is the bottleneck that severely limits time and energy efficiency at inference.

Algorithm 6. Pixel Shuffle

for $o_c = 0$, $o_c{+}{+}$, while $o_c < O_C$ **do**
  for $o_w = 0$, $o_w{+}{+}$, while $o_w < O_W$ **do**
    for $o_h = 0$, $o_h{+}{+}$, while $o_h < O_H$ **do**
      $i_h \leftarrow \lfloor \frac{o_h}{r} \rfloor$
      $i_w \leftarrow \lfloor \frac{o_w}{r} \rfloor$
      $i_c \leftarrow r^2 \cdot o_c + r \cdot \textbf{mod}(o_h, r) + \textbf{mod}(o_w, r)$
      $\text{output}[o_c, o_h, o_w] \leftarrow \text{input}[i_c, i_h, i_w]$

(a) NN **(b)** t_0 **(c)** t_1 **(d)** t_2 **(e)** t_3

Figure 5.3. We visualize an example nearest neighbor resize convolution, where a 2×2 input is first upsampled by a factor of 2 using NN interpolation, then convolved with a 3×3 kernel (in green) in a higher dimensional space to create the 4×4 output (in red).

5.4 Deconvolution

Deconvolution (also referred to as transpose convolution) is an end-to-end learnable upsampling algorithm that, in contrast to the sub-pixel and resize convolutions, inherently increases the resolution of an input image [48, 180]. Deconvolution was first popularized by Zeiler *et al.* [180] to visualize the latent representations of convolutional neural networks and has since gained popularity in deep learning-based image upsampling solutions [164]. Unlike the other convolution-based upsampling algorithms, deconvolution upsamples the image directly in one operation. A common misconception of the deconvolution operation is that it *requires* inserting zeros to perform fractionally strided transposed convolutions. While it is possible to execute deconvolution this way, it greatly increases the input feature map size by adding redundant zero-valued operations, resulting in a much less efficient implementation [48]. We discuss this formulation further in Section 5.4.2. As shown in Figure 5.4, the standard deconvolution algorithm given by Algorithm 7 strides over the input space, creating overlapping sums in the output space when the stride (S) is smaller than kernel (K) [24, 31, 48, 185]. In the context of training, these overlapping sums have been shown to introduce checkerboard

artifacts as a result of gradient updates [124]. In the context of inference, accumulating over these overlapping regions requires storing partial sums when $K > S$ and leads to communication overhead, complex dataflow, and increased resource utilization via on-chip buffering [24, 31, 104, 185]. Key algorithmic approaches that work around this *overlapping sums* problem can be divided into the following three categories: reverse looping deconvolution, fractionally strided deconvolution, and transforming deconvolution to convolution. We describe each below.

(a) t_0 **(b)** t_1 **(c)** t_2 **(d)** t_3 **(e)** final

Figure 5.4. The standard deconvolution algorithm strides over the input space, creating overlapping sums in the output space when the stride S is smaller than kernels size K [24, 31, 48, 185]. In this example, the 2×2 input (blue) is deconvolved with a 4×4 kernel (green) using a stride $S = 2$ and a padding $P = 1$ to create the 4×4 output (red).

Algorithm 7. Standard Deconvolution

for $o_c = 0$, o_c++, while $o_c < O_C$ **do**
  for $i_w = 0$, i_w++, while $i_w < I_W$ **do**
    for $i_h = 0$, i_h++, while $i_h < I_H$ **do**
      for $i_c = 0$, i_c++, while $i_c < I_C$ **do**
        for $k_h = 0$, k_h++, while $k_h < K$ **do**
          for $k_w = 0$, k_w++, while $k_w < K$ **do**
            $o_h \leftarrow S \times i_h + k_h - P$
            $o_w \leftarrow S \times i_w + k_w - P$
            $x \leftarrow \text{input}[i_c, i_h, i_w]$
            $w \leftarrow \text{kernel}[i_c, o_c, k_h, k_w]$
            $\text{output}[o_c, o_h, o_w] \leftarrow x \times w$

5.4.1 Reverse Looping Deconvolution

To avoid partial summations in the output space, Zhang *et al.* [185] introduce a technique they refer to as *reverse looping* in which they switch the looping criteria from the *input* space (see Algorithm 7) to the *output* space (see Algorithm 13). By doing so, they enable a one-shot write to off-chip memory for each output pixel and expose opportunities for concurrent execution. To

ensure functional correctness, they also introduce a technique they refer to as *stride-hole skipping* in which the looping criteria skips over the output space in S^2 tiles and an offset based on modulo arithmetic is added to each output pixel address. The resulting reverse looping deconvolution algorithm (REVD) enables the concurrent execution of these independent tiles but relies on expensive modulo arithmetic for address calculations.

Observing that the modulo arithmetic needed to calculate the output pixel address (o_h) is only dependent on the kernel address (k_h), Colbert *et al.* [31] minimize its impact by pre-computing and caching these offsets for each value of k_h. Assuming square kernels, the process reduces the number of modulo operations to $2K$, which minimizes resource utilization and on-chip memory requirements as K tends to be small. In Section 5.5, we propose a further improved version of this algorithm to expose additional opportunities for concurrent execution without the use of stride-hole skipping.

Algorithm 8. Reverse Looping Deconvolution (REVD)

for $o_c = 0$, o_c++, while $o_c < O_C$ **do**
  for $\hat{o}_w = 0$, $\hat{o}_w$+=S, while $\hat{o}_w < O_W$ **do**
    for $\hat{o}_h = 0$, $\hat{o}_h$+=S, while $\hat{o}_h < O_H$ **do**
      for $k_h = 0$, k_h++, while $k_h < K$ **do**
        for $k_w = 0$, k_w++, while $k_w < K$ **do**
          for $i_c = 0$, i_c++, while $i_c < I_C$ **do**
            $f_h = \textbf{mod}(S - \textbf{mod}(P - k_h, S), S)$
            $f_w = \textbf{mod}(S - \textbf{mod}(P - k_w, S), S)$
            $o_h = \hat{o}_h + f_h$
            $o_w = \hat{o}_w + f_w$
            $i_h = (o_h + P - k_h)/S$
            $i_w = (o_w + P - k_w)/S$
            $x \leftarrow \text{input}[i_c, i_h, i_w]$
            $w \leftarrow \text{kernel}[i_c, o_c, k_h, k_w]$
            $\text{output}[o_c, o_h, o_w] \leftarrow x \times w$

5.4.2 Fractionally Strided Deconvolution

The fractionally strided deconvolution (STRD) can be implemented using unmodified convolution accelerators and is most commonly used by machine learning frameworks such as

PyTorch [127] and TensorFlow [1]. It avoids the overlapping sum problem by padding each input pixel by $S-1$ zeros before executing a standard convolution using transposed kernels [48]. It can be viewed as two serialized operations: (1) a zero-insertion feature map transformation followed by (2) a same-padded convolution with transposed kernels. As shown in Figure 5.5, to upsample by a factor of r, the fractionally strided convolution first inserts $H-1$ rows and $W-1$ columns of $S-1$ zeros into the input feature map [48] before transposing the deconvolution kernels to execute a same-padded convolution. While this works around the overlapping sum problem, it introduces massive redundancies as the input feature map grows [48]. Figure 5.6 shows how the percentage of zero-valued input pixels increases with upsampling factor. Note that, even when upsampling by a factor of 2, the majority of the input pixels are redundant computations.

(a) t_0 **(b)** t_1 **(c)** t_2 **(d)** t_3 **(e)** final

Figure 5.5. For the fractionally strided deconvolution to be equivalent to the example in Figure 5.4, $S-1$ zeros are first inserted in between the input pixels. Then, the transpose of the 4×4 deconvolution kernels (green) is convolved over the 3×3 input (blue) with a stride of $S=1$ and padding $P=2$ to create the 4×4 output (red).

5.4.3 Transforming Deconvolution to Convolution

Chang *et al.* [24] avoid the overlapping sum problem by transforming deconvolution into S^2 tiled convolutions to compute each region of the output feature map independently. They refer to this formulation as transforming deconvolution to convolution (TDC). To split a deconvolution into S^2 convolutions, the TDC algorithm first uses the transformation given by Algorithm 9 to split the deconvolution kernels into S^2 tiles of size $K_T\times K_T$, where $K_T=\lceil\frac{K}{S}\rceil$. When K is not evenly divisible by S, this transformation requires padding each kernel tile by P_K, where $P_K=(S\times K_T)-K$. Figure 5.6 shows how the percentage of zero-valued weights increases with upsampling factor (r). Given the transformed kernels, the output pixels of each of the S^2 tiles

can then be computed concurrently using a same-padded convolution. However, similar to the sub-pixel convolution, the transformation process requires expensive post-processing to stitch the resulting tiles back together [22, 24, 171]. Xu *et al.* [171] introduce a variant of this algorithm that directly stitches these output tiles together during computation by merging the modulo-based arithmetic into the core algorithm. This optimization reduces penalties from data transfers at the cost of increased latency and additional hardware resources. In our algorithm comparisons, we focus on this variant of TDC.

Algorithm 9. TDC Kernel Transformation

for $o_c = 0$, o_c++, while $o_c < O_C$ **do**
  for $i_c = 0$, i_c++, while $i_c < I_C$ **do**
    for $k_h = 0$, k_h++, while $k_h < K_H + P_K$ **do**
      for $k_w = 0$, k_w++, while $k_w < K_W + P_K$ **do**
        $n = S \times \textbf{mod}(k_h, S) + \textbf{mod}(k_w, S)$
        $\hat{k}_h = K_T - \lceil \frac{k_h+1}{S} \rceil$
        $\hat{k}_w = K_T - \lceil \frac{k_w+1}{S} \rceil$
        $w_T[o_c, i_c, n, \hat{k}_h, \hat{k}_w] = w[i_c, o_c, k_h, k_w]$

5.4.4 Comparison of Compute and Memory Requirements

Table 5.1 summarizes the compute and memory requirements for each of these deconvolution algorithms. Compute requirements (C) are measured by the number of multiply-accumulates (MACs). Memory requirements (M) are measured by the number of weights (W) and activations[1] (A) such that $M \equiv W + A$.

Table 5.1. We provide the compute and memory requirements for each deconvolution algorithm. For simplicity, we assume square kernels K, square inputs H, and equal input/output channels C. Recall that the upsampling factor is denoted by r. We define $P_H = (H-1)(r-1)$ as the zeros inserted to pad each input pixel for the fractionally strided deconvolution (STRD).

	#MACs (C)	**# Parameters (W)**	**# Activations (A)**
REVD	$r^2 \times K^2 \times H^2 \times C^2$	$r^2 \times K^2 \times C^2$	$(1+r^2) \times H^2 \times C$
STRD	$r^4 \times K^2 \times H^2 \times C^2$	$r^2 \times K^2 \times C^2$	$\left(r^2 \times H^2 + (H+P_H)^2\right) \times C$
TDC	$r^2 \times K^2 \times H^2 \times C^2$	$r^2 \times K^2 \times C^2$	$(1+r^2) \times H^2 \times C$

[1] We refer to *activations* as the total input and output feature maps

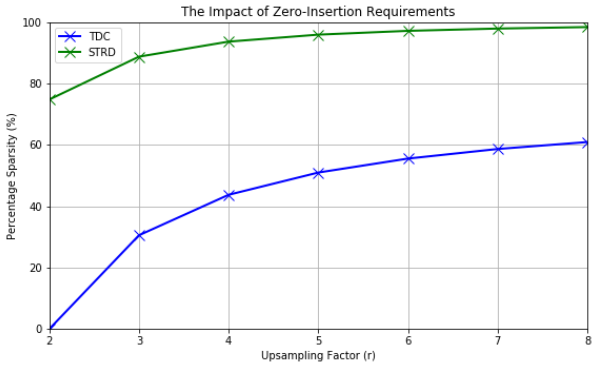


Figure 5.6. We visualize the impact of zero-insertion requirements for STRD features and TDC weights.

In Figure 5.6, we visualize how the percentage of sparse features or weights increase for STRD or TDC, respectfully, as the upsampling factor (r) increases. As shown in **green**, implementing a fractionally strided deconvolution (STRD), as is common practice, greatly increases the input feature map size by adding redundant zero-valued operations [48]. Assuming a square input feature map, even upsampling by a factor of 2 requires 75% of input pixels to be zero. As shown in **blue**, transforming deconvolution to a convolution (TDC) requires padding the transformed kernels by P_K when the kernel size (K) is not evenly divisible by the stride (S). Here, we assume $P_K = 2$ to demonstrate how the percentage of zero-valued weights increases with r, but at a lesser rate than STRD. Note that if K is evenly divisible by S, then no zeros are required in the TDC kernel transformation.

5.5 Improving Reverse Looping Deconvolution

When the memory requirements of a deep learning model exceed the resources available on an edge device, the inference pass is typically divided into smaller workloads using tiling [110, 181]. These tiled workloads are data-independent if they each write to distinct memory locations without overlap. Algorithms that can be divided into smaller tiles have higher degrees of parallelism as data-independent workloads can be executed concurrently across single-instruction, multiple-data (SIMD) lanes on parallel platforms such as GPUs and FPGAs. When data-

independent workloads are evenly balanced, tiling optimizations can increase hardware utilization and lead to improved energy efficiency. On the other hand, the presence of imbalanced workloads can force all live processes to wait for an overloaded lane to finish before synchronization, which increases the penalty as the number of concurrent processes grows [129]. Algorithms with higher degrees of parallelism expose more opportunities for efficient concurrent execution as workloads are more easily balanced across SIMD lanes.

To understand the impact of data independence and load balance on inference acceleration, consider a processor with 16 SIMD lanes designed to accelerate a deconvolution workload. When used to upsample a 14×14 image by a factor of 2 using a stride of 2 such that $r = 2$, $S = 2$, and $O_H \times O_W = 28 \times 28$, a designer could use a tile size of 7 to divide the total work into 16 data-independent workloads to be dispatched across each lane and achieve 100% load balance. However, both the reverse looping deconvolution (REVD) and transforming deconvolution to convolution (TDC) algorithms discussed in Section 5.4 traverse the output space in S^2 tiles [24, 185]. As a consequence, functional correctness breaks down when the tiling along the output space is not perfectly divisible by the stride S. This constrains the degree to which data independence can be exploited. A tiling factor of 7 is not perfectly divisible by the stride of 2 so a designer is left with options 6 and 8. Using a tiling factor of 8 requires zero-padding each workload, effectively increasing the data movement by 30%[2]. This 1.3x increase is detrimental to the system's energy efficiency as energy consumption is dominated by data movement [76]. Using a tiling factor of 6 requires multiplexing through the 16 SIMD lanes twice, reducing hardware utilization to 78%, increasing system latency, and introducing imbalanced workloads that would need to wait for each lane to finish[3]. To both circumvent the overlapping sums problem and fully exploit data-independent concurrent execution, a deconvolution algorithm needs to sequentially traverse the output space. To address this, we propose an improved reverse

[2]Moving 16 tiles of 8×8 pixels is 1.3x that of moving 16 tiles of 7×7.

[3]Executing 16 tiles of 6×6 pixels only covers 73% of the total compute work of a 28×28 image. To complete the final 27%, the hardware would need to multiplex through the SIMD lanes a second time with 6 tiles of 6×6 pixels. In this pass, only 36% of the hardware is utilized. Over both passes, only 78% of the hardware is being used.

looping deconvolution algorithm, which we refer to as REVD2.

As shown in Algorithm 10, REVD2 uses stride-hole skipping along the *weight* space rather than the output space to both avoid the overlapping sums problem and fully expose the data independence of output pixels for more effective load balancing. Unlike TDC or REVD, REVD2 supports a tile size of 7 in the example described above. It also reduces the cost of modulo arithmetic. Furthermore, we can fully remove any dependence of REVD2 on modulo arithmetic by leveraging the data independence of each output pixel. When dispatching each tiled workload for concurrent execution, $\mathbf{mod}(o_h + P, S)$ can be replaced by a simple counter (j) initialized to P. The counter is incremented for each workload and reset to zero when $j \geq S$, where S is the stride[4].

Algorithm 10. Improved Reverse Looping Deconvolution

for $o_c = 0, o_c{+}{+}$, while $o_c < O_C$ **do**
  for $o_h = 0, o_h{+}{+}$, while $o_h < O_H$ **do**
    for $o_w = 0, o_w{+}{+}$, while $o_w < O_W$ **do**
      for $i_c = 0, i_c{+}{+}$, while $i_c < I_C$ **do**
        for $\hat{k}_h = 0, \hat{k}_h{+}{=}S$, while $\hat{k}_h < K$ **do**
          for $\hat{k}_w = 0, \hat{k}_w{+}{=}S$, while $\hat{k}_w < K$ **do**
            $k_h = \hat{k}_h + \mathbf{mod}(o_h + P, S)$
            $k_w = \hat{k}_w + \mathbf{mod}(o_w + P, S)$
            $i_h = (o_h + P - k_h)/S$
            $i_w = (o_w + P - k_w)/S$
            $x \leftarrow \text{input}[i_c, i_h, i_w]$
            $w \leftarrow \text{kernel}[i_c, o_c, k_h, k_w]$
            $\text{output}[o_c, o_h, o_w] \leftarrow x \times w$

5.6 Conclusions

In this chapter, we introduce a novel deconvolution algorithm that exposes further opportunities for parallelization. This algorithm builds on top of the reverse looping deconvolution [31, 185] to apply stride-hole skipping over the weight space rather than the output space. Doing so fully exposes per-pixel parallelization. We also characterize the difference of compute

[4]We have implemented this at https://github.com/icolbert/upsampling

workloads when using other proposed deconvolution inference algorithms. We show that our algorithm (REVD2) requires far less padding, which reduces both the compute and memory requirements. In Chapter 6, we will study the time and energy costs of these algorithms using theoretical quantitative models.

Acknowledgements: This chapter is based on material as it appears in the 2021 IEEE Access Journal (Ian Colbert, Kenneth Kreutz-Delgado, and Srinjoy Das, "An Energy-Efficient Edge Computing Paradigm for Convolution-Based Image Upsampling"). The dissertation author was the primary investigator and author of this paper. The authors would like to thank Xinyu Zhang for his insightful discussions. The constructive comments of the anonymous reviewers and the associate editor are also acknowledged.

Chapter 6

Kernel Transforms, Layer Fusions, and Time-Energy Analysis

State-of-the-art deep learning solutions for image upsampling are currently trained using either resize or sub-pixel convolution to learn kernels that generate high-fidelity images with minimal artifacts. However, performing inference with these learned convolution kernels requires memory-intensive feature map transformations that dominate time and energy costs in real-time applications. To alleviate this pressure on memory bandwidth, we propose a novel energy-efficient edge computing paradigm that confines the use of resize or sub-pixel convolution to training in the cloud by transforming learned convolution kernels to deconvolution kernels before deploying them for inference as functionally equivalent deconvolutions. These kernel transformations, intended as a one-time cost when shifting from training to inference, enable a systems designer to use each algorithm in their optimal context by preserving the image fidelity learned when training in the cloud while minimizing data transfer penalties during inference at the edge. We then compare the inference properties of these convolution-based image upsampling algorithms. To demonstrate the benefits of our approach, we upsample images selected from the BSD300 dataset using a pre-trained single-image super resolution network provided by the PyTorch model zoo. Using quantitative models of incurred time and energy costs to analyze this deep neural network, we estimate that using REVD2 for inference at the edge improves the latency of the final layer by up to 2.1x or 2.8x and energy efficiency by up to 2.1x or 2.7x when

respectively compared to sub-pixel or resize convolution counterparts.

6.1 Introduction

As discussed in Chapter 5, state-of-the-art deep learning solutions most often use resize or sub-pixel convolution when building deep neural networks for image upsampling [164]. However, both require memory-intensive feature map transformations that can bottleneck latency and energy efficiency when deployed for inference in real-time applications. This forces systems designers to balance trade-offs between image fidelity and hardware performance. Models trained using resize convolution learn to generate high-fidelity images without introducing checkerboard artifacts; however, the algorithm relies on memory-intensive pre-processing to resize the image before inefficiently executing compute operations in a higher dimensional space where the cost is greater [46, 124, 144]. When properly initialized, models trained with sub-pixel convolution converge faster with less test reconstruction error [6, 144]; however, the core algorithm requires memory-intensive post-processing to shuffle pixels every inference pass. Alternatively, a model can use deconvolution to efficiently upsample images without any additional pre- or post-processing, thereby reducing data transfer penalties; however, gradient updates during training can introduce checkerboard artifacts during inference [6, 124]. To ease the selection between these convolution-based image upsampling algorithms, we propose a novel edge computing paradigm that enables the use of each algorithm in their optimal context[1]. In doing so, we avoid the data transfer penalties incurred by image resize or pixel shuffle without sacrificing the image quality obtained when training a network using either resize or sub-pixel convolution, respectively.

As depicted in Figure 6.1, our framework confines the use of sub-pixel or resize convolution to training in the cloud, where the cost of a pixel shuffle or image resize is less severe. Given a trained network, the learned convolution kernels are converted to deconvolution kernels

[1]We use *convolution-based upsampling* to denote algorithms relying on either convolution or transposed convolution, commonly known as deconvolution.

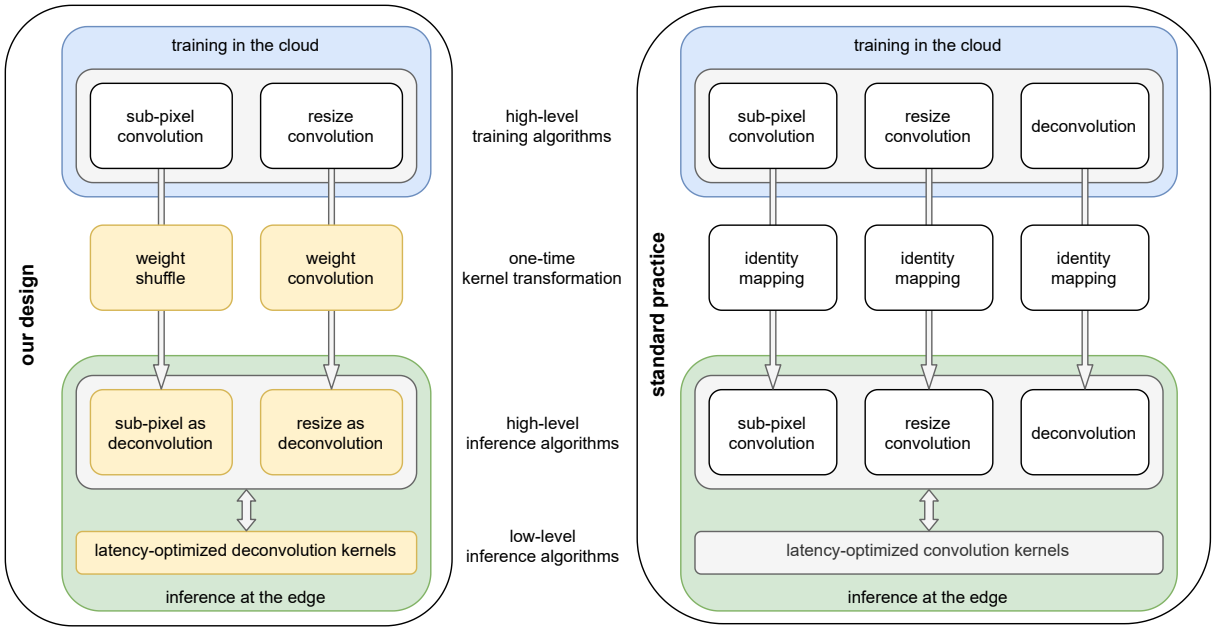


Figure 6.1. In our design paradigm (left), we introduce the blocks highlighted in yellow to use convolution-based image upsampling algorithms in their optimal context. Standard edge computing frameworks (right) deploy pre-trained networks to run inference locally on edge devices using the same high-level algorithms selected for training (*i.e.*, an identity mapping from training to inference). At runtime, these high-level algorithms (*e.g.*, convolution) are executed directly as latency-optimized compute kernels (*e.g.*, general matrix-multiply) [86,97]. We refer to these compute kernels as *low-level algorithms* so as to not confuse them with learned convolution weights, which are also referred to as kernels. Note that the high-level algorithms used for training are not the same as those used for inference under our paradigm further discussed in Section 6.3.

using the one-time transformations discussed in Section 6.4. The transformed kernels are then executed as a functionally equivalent deconvolution using our improved reverse looping deconvolution algorithm, which we refer to as REVD2, for inference at the edge. Thus, the high-level algorithms used for training are not the same as those used for inference. When avoiding the memory-intensive feature map transformations required for sub-pixel or resize convolution-based inference, we significantly reduce data transfer penalties to improve both latency and energy efficiency without sacrificing image fidelity. Below, we summarize our key contributions:

1. As shown in Figure 6.1, we enable the physical separation of high-level training and inference algorithms by introducing a set of single-use kernel transformations that translate

sub-pixel and resize convolutions to functionally equivalent deconvolutions (Section 6.4).

2. We analyze and compare the properties of these algorithms under a quantitative model to verify our design paradigm and show that translating to deconvolution for inference from sub-pixel or resize convolution reduces time and energy costs (Section 6.5).

3. We summarize the implications of these experiments and provide recommendations for system designers to support real-time, energy-efficient convolution-based image upsampling (Section 6.6).

As computations move from high-performance cloud servers to resource-constrained edge devices, the physical separation of training and inference exposes opportunities for specialized optimizations in either context. Although our proposal is targeted at edge computing in computer vision applications, we believe the algorithms and conclusions discussed in this paper have a broader impact. By demonstrating that our kernel transformations improve both latency and energy efficiency during inference, we hope to exploit this knowledge to support real-time deep learning applications, guide system designers to build inference-aware deep neural network architectures, and improve our understanding of the core algorithms used for image upsampling.

6.2 Related Works

This paper synthesizes a collection of previous works into the foundation of our edge computing design methodology. We provide an overview of prior works related to compute kernel fusion and quantitative models of time and energy.

Compute Kernel Fusion. To mask the costs of memory traffic at runtime, previous works have used *compute kernel fusion* - a software optimization technique that combines two or more functions into one equivalent, unified compute kernel [10,58,102,132,158,189]. The fused kernel uses the same compute resources as its original components, but with reduced penalties incurred

by data transfers [102]. In deep learning, this is often referred to as *layer fusion* as the compute kernels being fused are often pre-optimized neural network primitives. Similar to our work, layer fusion techniques are intended as inference optimizations capable of efficiently combining operators to reduce penalties incurred by memory accesses and context switching [132, 189]. Previous works aim to either vertically fuse layers (*e.g.*, fusing batch normalization and activations functions into a convolution) or horizontally fuse layers (*e.g.*, combining parallel executing modules) while preserving the underlying algorithms [10, 58, 132, 189]. These fusions do not alter the underlying computations; instead, they restructure the layers to execute faster and more efficiently [132]. The key difference in our research from that of layer fusion is that our kernel transformations *do* alter the underlying computations such that the high-level algorithms used for training are not the same as those used for inference. Unlike previous works, our approach does not simply concatenate compute kernels together, but rather we execute mathematical transformations directly on the learned convolution weights. In this way, our techniques can be more aptly seen as *layer translations* enabled by our novel transformations, which are discussed further in Section 6.4. Note that, after applying our transformations, a systems designer can additionally leverage layer fusion techniques to further reduce penalties incurred by memory accesses and context switching.

Quantitative Models of Time and Energy. As further discussed in Section 6.5, we adopt existing quantitative models of time and energy to validate our proposed paradigm based on the properties of each algorithm. To analyze the performance bounds of each algorithm in terms of time, we use the framework proposed by Williams *et al.* [167], commonly referred to as the roofline model. For a given hardware target characterized by its peak computational performance and peak memory bandwidth, this model estimates the performance bounds of an algorithm as a function of its compute and memory operations [28, 167]. Inspired by this work, Choi *et al.* [28] introduce an energy-based analog of the time-based roofline model. Similar to [167], this roofline model of energy connects properties of an algorithm (expressed in terms of

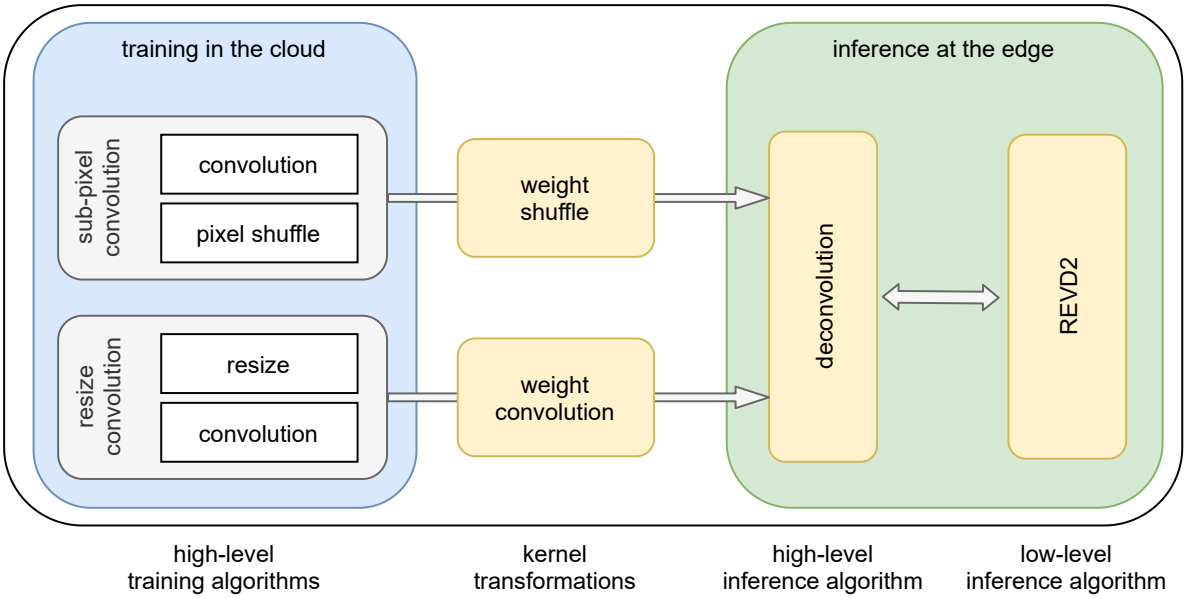


Figure 6.2. We confine the use of sub-pixel or resize convolution to training in the cloud, where the cost of their additional data transfers is less severe. Once trained, the learned convolution kernels are transformed into deconvolution kernels and then deployed for energy-efficient inference using REVD2.

operations, concurrency, and memory traffic) with the time and energy costs of a target hardware architecture [28]. We use this framework to analyze the performance bounds of each algorithm in terms of energy. To further analyze energy efficiency, we use the methods introduced by Jha *et al.* [84], which provide another means of estimating the energy efficiency of a given convolution-based image upsampling algorithm by its data requirements. We discuss each of these quantitative models further in Section 6.5.

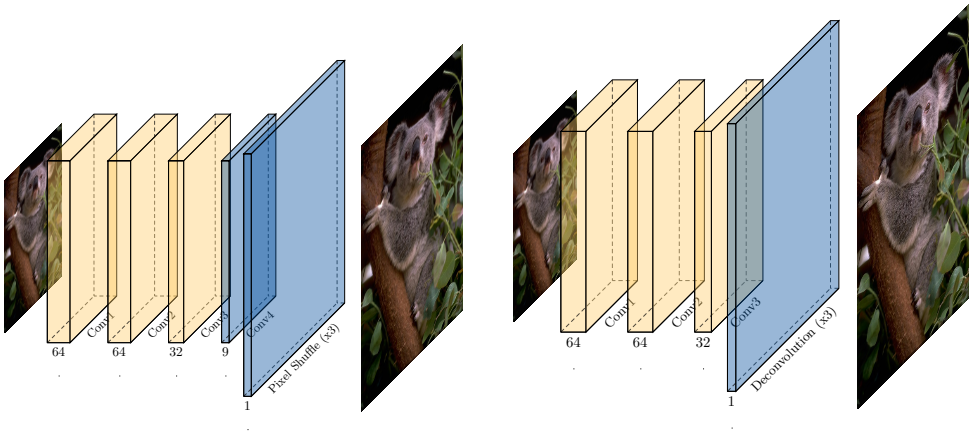
6.3 Deep Learning-based Image Upsampling at the Edge

As edge devices have become more powerful, deep learning computations have moved increasingly closer to data sources to support real-time applications [44]. In such paradigms, inference at the edge is physically separated from cloud-based training. This *cloud-to-edge* separation reduces strain on network bandwidth, decreases system latency, removes reliance on ubiquitous internet connectivity, and improves overall security [44]. Standard edge computing

frameworks deploy pre-trained neural networks to run locally on edge devices and the high-level algorithms used during training are the same as those used during inference. At runtime, these high-level algorithms are executed directly as compute kernels often selected by an optimizer without an explicit directive from a systems designer [132]; for example, the standard convolution algorithm can be executed using general matrix-multiply (GEMM) or fast Fourier transform (FFT) compute kernels [86, 97]. We refer to these compute kernels as *low-level algorithms* so as to not confuse them with learned convolution weights, which are also referred to as kernels.

As discussed in Chapter 5, state-of-the-art image upsampling neural networks are currently trained to generate high-fidelity images with minimal artifacts using either sub-pixel or resize convolution. However, convolution is inherently a downsampling algorithm; to upsample an image requires memory-intensive feature map transformations with each inference pass that severely limits both latency and energy efficiency. Alternatively, deconvolution is inherently an upsampling algorithm; the core arithmetic increases the resolution of an image directly, without reliance on any pre- or post-processing. Thus, our proposed edge computing paradigm confines the use of sub-pixel or resize convolution to training in the cloud, where the cost of a pixel shuffle or interpolation-based image resize is less severe. To alleviate memory pressure at inference, we convert the learned convolution kernels into deconvolution kernels using the transformations introduced in Section 6.4. As shown in Figure 6.2, these kernel transformations translate the sub-pixel or resize convolutions into a functionally equivalent deconvolution to be executed using our novel inference algorithm, which we refer to as REVD2 (see Chapter 5). Thus, the high-level algorithms used for training are not the same as those used for inference under our paradigm. By doing so, we significantly reduce data transfer penalties to improve both latency and energy efficiency during inference at the edge without sacrificing the image fidelity of the pre-trained model. Figure 6.3 shows an example of this optimization applied to the single-image super resolution network provided by PyTorch [127]. Note that the images generated before and after the kernel transformation are identical.

As discussed in Chapter 5, standard deconvolution arithmetic traverses the input space,



(a) Training Neural Network Architecture **(b)** Inference Neural Network Architecture

Figure 6.3. We provide an example use case of our proposed methodology using the single-image super resolution network example provided by PyTorch [127]. Using the techniques detailed in Section 6.4.1, we avoid the sub-pixel convolution layer (*i.e.*, conv4 + pixel shuffle in **blue** on the top) by using the weight shuffle algorithm to translate the learned kernels to deconvolution kernels. The transformed kernels are then executed as a functionally equivalent deconvolution at inference (*i.e.*, deconvolution in **blue** on the bottom). The demonstration is performed using an image selected from the BSD300 [114] dataset.

creating regions of overlapping summations in the output space that introduce difficulties for efficient inference acceleration. Previous works propose solutions to work around this *overlapping sums problem*, but none efficiently expose opportunities to compute each individual output pixel concurrently. To address these shortcomings, we build on the reverse looping deconvolution (REVD) algorithm to introduce a novel variation that improves its adaptability to the limited resource availability common to edge devices. Introduced in Chapter 5, REVD2 exposes more opportunities for concurrent execution and further reduces the cost of modulo arithmetic. Using the quantitative models of time and energy discussed in Section 6.5, we analyze the benefits of using REVD2 as the low-level inference algorithm under our paradigm depicted by Figure 6.2.

6.4 Kernel Transformations

To preserve the image fidelity of a pre-trained neural network while avoiding the data transfer penalties during inference, we introduce two novel kernel transformations that exploit

the functional equivalence of deconvolution to the sub-pixel and resize convolution algorithms. As opposed to the feature map transformations required for each sub-pixel or resize convolution inference pass, these kernel transformations are intended as a one-time sunk cost in software *before* deploying the trained model for inference. Once deployed, the functionally equivalent deconvolution can be executed using any of the formulations described in Section 5.4[2].

Given that functions f and g are respectively parameterized by θ and β, then g_β is functionally equivalent to f_θ if $f_\theta(x) = g_\beta(x)$ for all valid x. As described in Chapter 5, both the sub-pixel and resize convolution rely on same-padded convolutions, which use a stride of 1. As shown below, this restricts the valid convolution kernel sizes to be odd as $K = 2P + 1$. When transforming learned convolution kernels to deconvolution kernels, the functional equivalence holds for all valid kernel sizes (*e.g.*, 1, 3, 5, 7, *etc.*).

$$O_H = (I_H - K + 2P)/S + 1 \tag{6.1}$$

$$H = (H - K + 2P)/1 + 1 \tag{6.2}$$

$$K = 2P + 1 \tag{6.3}$$

6.4.1 Sub-Pixel Convolution to Deconvolution

To upsample by a factor of r, the sub-pixel convolution generates r^2 more output channels before applying the pixel shuffle algorithm over the output space. As shown in Figure 5.2, this results in a unique pattern of r^2 output pixels, each originating from independent channels. To replicate this pattern, a functionally equivalent deconvolution uses a stride $S = r$ with a kernel size $K^D = rK$. The deconvolution padding (P^D) is calculated below where K and P are respectively the convolution kernel size and padding given in Eq. 6.3, I_H is the input height, K^D

[2]Although TDC translates deconvolution to convolution, it does not undo the proposed kernel transformations. The algorithm splits the resulting deconvolution into S^2 tiles to be executed concurrently as same-padded convolutions but the outputs need to be stitched back together either during [24] or after [171] the computation. While the algorithm is functionally equivalent to deconvolution, the resulting S^2 tiles may require zero-padding of the learned kernels under the conditions described in Section 5.4.

is the deconvolution kernel size, S is the stride, and O_H is the output height [48].

$$O_H = S \times (I_H - 1) + K^D - 2P^D \tag{6.4}$$

$$rH = r(H-1) + rK - 2P^D \tag{6.5}$$

$$2P^D = r(K-1) \tag{6.6}$$

$$P^D = rP \tag{6.7}$$

Building from the qualitative analysis of Shi *et al.* [145], we introduce the weight shuffle algorithm, given by Algorithm 11. Given a sub-pixel convolution with a valid kernel size, this algorithm transforms the $K \times K$ learned convolution kernels into $rK \times rK$ deconvolution counterparts to be executed as a functionally equivalent deconvolution. As shown in Figure 6.4, the weight shuffle algorithm moves the learned parameters of the convolution kernels in a similar way to the pixel shuffle but also reverses element indices as a 2D matrix transpose. Unlike the pixel shuffle algorithm, which requires a memory-intensive feature map transformation at each inference pass, the weight shuffle algorithm transforms the kernels of a pre-trained network. It is a one-time sunk cost that can be done in software *before* deploying a trained model for inference.

Algorithm 11. Weight Shuffle

for $i_c = 0$, i_c++, while $i_c < I_C$ **do**
  for $o_c^d = 0$, o_c^d++, while $o_c^d < O_C^D$ **do**
    for $k_h^d = 0$, k_h^d++, while $k_k^d < K_H^D$ **do**
      for $k_w^d = 0$, k_w^d++, while $k_w^d < K_W^D$ **do**
        $k_h^c \leftarrow K_H^C - \lfloor \frac{k_h^d}{r} \rfloor - 1$
        $k_w^c \leftarrow K_W^C - \lfloor \frac{k_w^d}{r} \rfloor - 1$
        $o_c^c \leftarrow r^2 * o_c^d + r * \mathbf{mod}(k_h^d, r) + \mathbf{mod}(k_w^d, r)$
        $w_D[i_c, o_c^d, k_h^d, k_w^d] \leftarrow w_C[i_c, o_c^c, k_h^c, h_w^c]$

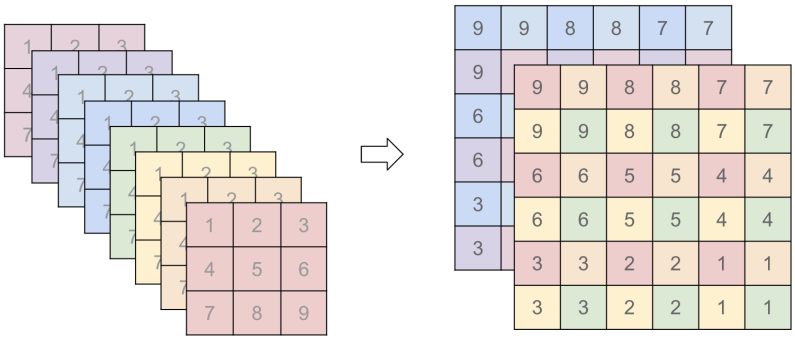


Figure 6.4. Our weight shuffle algorithm moves the learned parameters of the sub-pixel convolution kernels in a similar way to the pixel shuffle algorithm but also reverses the element indices. Unlike the pixel shuffle, this is a one-time cost in software *before* deploying a trained model for inference.

6.4.2 Resize Convolution to Deconvolution

To upsample by a factor of r, the nearest neighbor (NN) resize convolution first uses NN-interpolation to increase the resolution of the image before applying a same-padded convolution over the higher dimensional output space. As shown below to reflect Figure 5.3, the replication of each input pixel results in a unique pattern of r^2 identical output pixels. Formulating the ensuing same-padded convolution as a matrix multiplication enables a significant reduction in operations.

$$\mathbf{y}_{0,0} = \mathbf{x}_{0,0} * (\mathbf{w}_{1,1} + \mathbf{w}_{2,1} + \mathbf{w}_{1,2} + \mathbf{w}_{2,2}) \tag{6.8}$$

$$\mathbf{y}_{0,1} = \mathbf{x}_{0,0} * (\mathbf{w}_{1,0} + \mathbf{w}_{1,1} + \mathbf{w}_{2,0} + \mathbf{w}_{2,1}) + \tag{6.9}$$

$$\mathbf{x}_{0,1} * (\mathbf{w}_{1,2} + \mathbf{w}_{2,2}) \tag{6.10}$$

$$\mathbf{y}_{0,2} = \mathbf{x}_{0,1} * (\mathbf{w}_{1,1} + \mathbf{w}_{2,1} + \mathbf{w}_{1,2} + \mathbf{w}_{2,2}) \tag{6.11}$$

$$\mathbf{x}_{0,0} * (\mathbf{w}_{1,0} + \mathbf{w}_{2,0}) + \tag{6.12}$$

$$\mathbf{y}_{0,3} = \mathbf{x}_{0,1} * (\mathbf{w}_{1,0} + \mathbf{w}_{1,1} + \mathbf{w}_{2,0} + \mathbf{w}_{2,1}) \tag{6.13}$$

The emerging pattern matches the deconvolution arithmetic shown in Figure 5.4.

$$\mathbf{y}_{0,0} = \mathbf{x}_{0,0}\mathbf{w}^D_{1,1} \tag{6.14}$$

$$\mathbf{y}_{0,1} = \mathbf{x}_{0,0}\mathbf{w}^D_{1,2} + \mathbf{x}_{0,1}\mathbf{w}^D_{1,0} \tag{6.15}$$

$$\mathbf{y}_{0,2} = \mathbf{x}_{0,0}\mathbf{w}^D_{1,3} + \mathbf{x}_{0,1}\mathbf{w}^D_{1,1} \tag{6.16}$$

$$\mathbf{y}_{0,3} = \mathbf{x}_{0,1}\mathbf{w}^D_{1,2} \tag{6.17}$$

Solving for both sets of equations, we find that the linear combination follows a locally connected pattern similar to a convolution with transposed kernels, as shown below.

$$\mathbf{w}^D_{1,0} = \mathbf{w}_{1,2} + \mathbf{w}_{2,2} \tag{6.18}$$

$$\mathbf{w}^D_{1,1} = \mathbf{w}_{1,1} + \mathbf{w}_{2,1} + \mathbf{w}_{1,2} + \mathbf{w}_{2,2} \tag{6.19}$$

$$\mathbf{w}^D_{1,2} = \mathbf{w}_{1,0} + \mathbf{w}_{1,1} + \mathbf{w}_{2,0} + \mathbf{w}_{2,1} \tag{6.20}$$

$$\mathbf{w}^D_{1,3} = \mathbf{w}_{1,0} + \mathbf{w}_{2,0} \tag{6.21}$$

To account for the redundant replication of input pixels, a functionally equivalent deconvolution uses a stride $S = r$ and maintains padding such that $P^D = P = \frac{K-1}{2}$. The resulting kernel size K^D is calculated below where K is the convolution kernel size given by Eq. 6.3, I_H is the input height, S is the stride, and O^H is the output height [48].

$$O_H = S \times (I_H - 1) + K^D - 2P^D \tag{6.22}$$

$$rH = r \times (H - 1) + K^D - K - 1 \tag{6.23}$$

$$K^D = K + r - 1 \tag{6.24}$$

Building from this algebraic reduction, we introduce the weight convolution, given by Algorithm 12. Given a NN resize convolution with a valid kernel size, this algorithm transforms

$K \times K$ learned convolution kernels into $(r+K-1) \times (r+K-1)$ deconvolution kernels to be executed as functionally equivalent deconvolutions. To stride over the kernel space, the inequality $i+K-1 < K^D$ must hold such that $i < r$. As shown in Figure 6.5, the weight convolution transposes the learned kernels before convolving over the weight space with a stride of 1.

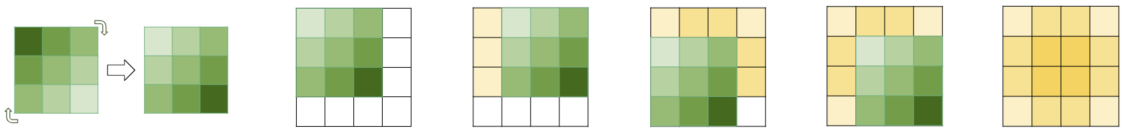
Algorithm 12. Weight Convolution

```
for i_c = 0, i_c++, while i_c < I_C do
    for o_c = 0, o_c++, while o_c < O_C do
        for i = 0, i++, while i < r do
            for j = 0, j++, while j < r do
                w_D[i_c, o_c, i : i+K, j : j+K] ← w_C[i_c, o_c]^T
```

Similar to the weight shuffle, this algorithm is a one-time sunk cost that is done *before* deploying a trained model for inference. However, unlike the weight shuffle, the weight convolution drastically lowers the total compute work as a consequence of the algebraic reductions. The resulting deconvolution requires only $H \times W \times C^2 \times (r+K-1)^2$ multiply-accumulate (MAC) operations. When compared to the resize convolution, which requires $H \times W \times C^2 \times K^2 \times r^2$, this is a significant reduction that scales with upsampling factor r. Using the standard kernel size of 3, the resulting ratio of deconvolution MACs to NN resize convolution MACs becomes $\frac{(r+2)^2}{(9r^2)}$. When upsampling by a factor of 2, the resulting deconvolution only requires 44% of the MAC operations used by its NN resize convolution counterpart to generate the same image. When upsampling by a factor of 3, the resulting deconvolution only requires 30% of the MACs to generate the same image. It is important to note that other forms of resize convolution (e.g., bilinear or bicubic) can be derived here as well, but we focus our attention on nearest neighbor resize convolution because of its prevalence in recent models [124, 136].

6.5 Time and Energy Analysis at Inference

In our proposed edge computing paradigm depicted in Figure 6.2, a deep learning model is first trained in the cloud using either sub-pixel convolution (C-SP) or nearest neighbor resize



(a) Transpose **(b)** t_0 **(c)** t_1 **(d)** t_2 **(e)** t_3 **(f)** t_N

Figure 6.5. We visualize an example application of the weight convolution algorithm. We can equate the operations depicted in Figures 5.3 and 5.4 from Chapter 5 by first rotating the kernels to reverse the indices, the convolving the reversed 3×3 convolution kernel (in green) over the 4×4 deconvolution kernel (in yellow) with a stride of 1.

convolution (C-NN). The learned weights are then converted to deconvolution kernels using the transformations discussed in Section 6.4. Finally, the resulting kernels are deployed for inference as a functionally equivalent deconvolution. In Chapter 5, we discussed the various formulations of deconvolution that avoid the overlapping summation problem observed when the kernel size (K) is greater than the stride (S). We consider and compare the following deconvolution variants as low-level inference algorithms - improved reverse looping deconvolution (REVD2), transforming deconvolution to convolution (TDC), and fractionally strided deconvolution (STRD). Here, we discuss a quantitative analysis framework to validate our proposed paradigm based on the properties of each algorithm. We refer to deconvolution as translated from sub-pixel convolution as D-SP and deconvolution as translated from nearest neighbor resize convolution as D-NN. The results and conclusions are discussed in Section 6.6.

6.5.1 Quantitative Models for Time and Energy

We use the simplified, optimistic quantitative model detailed by [28] to compare the time and energy costs of algorithms characterized by their compute and memory requirements. The assumptions held by this model yield a best-case analysis that is only valid when algorithms are sufficiently parallelizable [28]. As discussed in Chapter 5, each of the algorithms considered (*i.e.*, REVD2, TDC, STRD) are data-independent along the output space and can therefore be executed with high degrees of concurrency.

The time cost model given by Eq. 6.25 assumes an idealized hardware design that

perfectly masks the communication overhead with computation work, where T_{comp} and T_{mem} are the total time to execute all compute and memory operations, respectively [28]. Here, a higher computation-to-communication ratio would better hide memory bottlenecks.

$$T \equiv \max(T_{\text{comp}}, T_{\text{mem}}) \tag{6.25}$$

Similarly, let E_{comp} and E_{mem} be the total energy to execute all compute and memory operations, respectively, and let $E_0(T)$ be the cost of constant energy expended while executing the algorithm. Unlike time cost, the energy cost model given by Eq. 6.26 does not overlap computation and communication costs and has an additional penalty for increased latency [28].

$$E \equiv E_{\text{comp}} + E_{\text{mem}} + E_0(T) \tag{6.26}$$

For a fixed hardware architecture, let τ_{comp} and τ_{mem} be the time cost per compute and memory operation, respectively. For a given algorithm, let C be the total number of computation operations and M the total number of memory operations required. Under the optimistic assumption of hiding memory latency with perfect overlap, the total time cost of an algorithm becomes Eq. 6.27, where $T_{\text{comp}} \equiv C\tau_{\text{comp}}$ and $T_{\text{mem}} \equiv M\tau_{\text{mem}}$ [28].

$$T = \max(C\tau_{\text{comp}}, M\tau_{\text{mem}}) \tag{6.27}$$

Similarly, let $\varepsilon_{\text{comp}}$ and ε_{mem} be the energy cost per compute and memory operation, respectively. The total energy cost of an algorithm then becomes Eq. 6.28, where $E_{\text{comp}} \equiv C\varepsilon_{\text{comp}}$, $E_{\text{mem}} \equiv M\varepsilon_{\text{mem}}$, and the constant energy cost is assumed to be linear in time with a fixed constant power defined by π_0 such that $E_0(T) \equiv \pi_0 T$ [28].

$$E \equiv C\varepsilon_{\text{comp}} + M\varepsilon_{\text{mem}} + \pi_0 T \tag{6.28}$$

Given the compute and memory requirements of an algorithm, we use Eq. 6.27 and Eq. 6.28 to estimate time and energy costs using this idealized abstraction of hardware performance. In all of our experiments, we assume a fixed graphics processing unit (GPU) with the capabilities summarized below in Table 6.1, where time costs for compute and memory operations are measured in picoseconds per operation (ps/op) and energy costs for compute and memory operations are measured in picojoules per operation (pJ/op).

Table 6.1. For each experiment, we use the specifications provided by [28], which are based on the best-case capabilities of the assumed hardware.

τ_{comp}	0.3 ps / op
τ_{mem}	5.2 ps / op
$\varepsilon_{\text{comp}}$	43.2 pJ / op
ε_{mem}	437.5 pJ / op
π_0	66.4 W

Deconvolution Time and Energy Costs

Unlike REVD2, both STRD and TDC require the insertion of zeros to upsample an image by a factor of r. As discussed in Chapter 5, the presence of these redundant zero-valued computations increases with the upsampling factor and cannot be ignored when analyzing data movement patterns[3].

Figure 6.6 shows the relative increase in time and energy costs as a function of upsampling factor (r) for each low-level deconvolution algorithm using either the D-SP or D-NN formulation. In each experiment, we consider the case of upsampling a $1024 \times 1024 \times 3$ image by a factor of r using 3×3 kernels[4]. We assume a GPU with the capabilities summarized in Table 6.1 where all pixel values and network parameters are executed and stored at 32-bit precision (*i.e.*, 4 bytes). Because memory requirements are dominated by activations rather than weights, the impact of

[3]Zero-skipping techniques can lessen the impact of increased sparsity. However, it requires control logic that can introduce overhead and, if not properly balanced across concurrent processes, it can also introduce synchronization issues in multi-threaded hardware [15, 122]. We ignore the impact of these optimizations in our quantitative analysis.

[4]Note that most practical computer vision applications deal with 1K, 2K, or 4K images. To simplify analyses, we alter the standard 1K RGB image resolution, which is $1024 \times 768 \times 3$, to be square (*i.e.*, $1024 \times 1024 \times 3$).

the zero-insertion requirements for STRD is massive while that of TDC is negligible. As such, the fractionally strided deconvolution (STRD) formulation does not scale as well as REVD2 and TDC with workload size. Additionally, we observe the exacerbation of this trend when using D-SP rather than D-NN; for example, the time cost of upsampling an image by a factor of 5 using STRD for D-SP is 5x that of D-NN. This is due to the significant reduction in compute operations resulting from the weight convolution algorithm, as discussed in Section 6.4.2. We discuss this further in Section 6.6.

Convolution-based Upsampling Time and Energy Costs

Whereas deconvolution directly upsamples an image in one operation, both the sub-pixel convolution (C-SP) and nearest neighbor resize convolution (C-NN) rely on memory-dominated operations to move data for post- and pre-processing, respectively. These operations are required for every inference pass and cannot be ignored when analyzing data movement patterns, nor trivially fused with preceding or succeeding layers.

Table 6.2 summarizes the compute and memory requirements for each of the high-level convolution-based image upsampling algorithms. Note that the compute (C) and activation (A) requirements of C-SP and C-NN are equal. The sub-pixel convolution performs computations in low-resolution (LR) space but generates r^2 more output channels and requires expensive post-processing in high-resolution (HR) space. The resize convolution requires less-expensive pre-processing in LR space but performs its computations in HR space. Using kernel transformations to enable deconvolution for inference at the edge avoids any penalties for expensive data pre- or post-processing while maintaining the image fidelity learned through training in the cloud. We separately consider deconvolution as translated from sub-pixel convolution (D-SP) and deconvolution as translated from nearest neighbor resize convolution (D-NN).

Figure 6.7 shows the relative increase in time and energy costs as a function of upsampling factor (r) for each high-level convolution-based image upsampling algorithm. Again, we consider the case of upsampling a $1024 \times 1024 \times 3$ image by a factor of r using standard kernels of size

Table 6.2. We provide the compute and memory requirements for each high-level convolution-based image upsampling algorithm. For simplicity, we assume square kernels K, square inputs H, and equal input/output channels C. Note that both C-SP and C-NN activations include the pixel shuffle and NN-interpolation, respectively, as they are required for every inference pass. We assume REVD2 as the low-level deconvolution algorithm for D-SP and D-NN.

	#MACs (C)	**# Parameters (W)**	**# Activations (A)**
C-SP	$r^2 \times K^2 \times H^2 \times C^2$	$r^2 \times K^2 \times C^2$	$(1+3r^2) \times H^2 \times C$
C-NN	$r^2 \times K^2 \times H^2 \times C^2$	$K^2 \times C^2$	$(1+3r^2) \times H^2 \times C$
D-SP	$r^2 \times K^2 \times H^2 \times C^2$	$r^2 \times K^2 \times C^2$	$(1+r^2) \times H^2 \times C$
D-NN	$r^2 \times \lceil \frac{r+K-1}{r} \rceil^2 \times H^2 \times C^2$	$(r+K-1)^2 \times C^2$	$(1+r^2) \times H^2 \times C$

3×3. We assume fixed hardware with the capabilities summarized in Table 6.1 where all pixel values and network parameters are executed and stored at 32-bit precision. For both D-SP and D-NN, we assume REVD2 as the low-level deconvolution algorithm. With activations dominating memory requirements, the deviation in weight requirements (W) between C-NN and C-SP has minimal impact on time and energy costs. As such, the algorithms scale equivalently with workload size; however, neither scale as well as D-SP and D-NN. As further discussed in Section 6.6, avoiding the memory-intensive feature map transformations required of C-SP and C-NN significantly reduces both time and energy costs.

6.5.2 Estimating Energy Efficiency by Data Reuse

The efficiency of an algorithm is typically described by data reuse and measured using arithmetic intensity - the number of *useful* compute operations for every byte of data accessed. Algorithms with high data reuse are more likely to yield performance improvements with an increase in compute resources because computations dominate the communication overhead. Algorithms with low data reuse put more strain on a system's memory bandwidth as each compute operation requires more off-chip memory accesses. While the value of this ratio implies the scalability and locality of an algorithm, it fails to reliably estimate the energy efficiency of convolution-based deep learning algorithms [84]. By separately considering weight and activation reuse, Jha *et al.* [84] show that, for convolution-based deep learning algorithms, the

variation in arithmetic intensity is attributed to the variation in activation reuse, which in turn is highly correlated to variations in energy efficiency. As such, we use the compute and memory requirements discussed in Section 6.5.1 to estimate the energy efficiency of convolution-based upsampling algorithms by activation reuse.

Energy consumption is dominated by data movement rather than computation [76]. When estimating the energy efficiency using activation reuse, we define *useful* compute operations as those contributing to output pixel values. As such, we define activation reuse as the number of non-zero-valued computations for every byte of activation data accessed when upsampling an image by a factor of r. Figure 6.8 shows how activation reuse increases with upsampling factor r for each convolution-based upsampling algorithm. For D-SP and D-NN, we assume REVD2 as the low-level deconvolution algorithm. Again, we consider the case of upsampling a $1024 \times 1024 \times 3$ image by a factor of r using standard kernels of size 3×3 where all pixel values and network parameters are executed and stored at 32-bit precision.

6.5.3 Roofline Models of Time and Energy

The hardware architecture analog to arithmetic intensity is time-balance [28]. For a fixed machine, its time-balance point (B_τ) is defined as the ratio of its time cost per memory operation (τ_{mem}) to its time cost per compute operation (τ_{comp}) such that $B_\tau \equiv \tau_{\text{mem}}/\tau_{\text{comp}}$ [28]. Similarly, the energy-balance point (B_ε) of a machine is defined as the ratio of its energy cost per memory operation (ε_{mem}) to its energy cost per compute operation ($\varepsilon_{\text{comp}}$) such that $B_\varepsilon \equiv \varepsilon_{\text{mem}}/\varepsilon_{\text{comp}}$ [28].

When the arithmetic intensity of an algorithm is equal to the balance of a machine such that $\text{AI} = B$, the cost of the algorithm's compute operations is equal to that of its memory operations. We can visualize these balance principles using roofline models of time [167] and energy [28], which provide an upper bound on the attainable performance of an algorithm for fixed hardware as a function of that algorithm's data reuse patterns. Figure 6.9 shows these roofline models using the balance points provided by [28]. The roofs of each model are normalized to peak performance and data reuse is measured by arithmetic intensity for time and

activation reuse for energy. For each experiment, we consider the single-image super resolution network provided by PyTorch [127] shown in Figure 6.3. We use a 320x480 image selected from the BSD300 [114] dataset and assume the GPU capabilities summarized in Table 6.1. The model is pre-trained using sub-pixel convolution (*e.g.*, conv4 + pixel shuffle) to upsample a single-channel image by a factor of 3 such that the output image is 960x1440. As such, we use this deep neural network to analyze C-SP and D-SP. To analyze C-NN and D-NN, we design an analogous deep neural network architecture in which the sub-pixel convolution is substituted with NN resize convolution (*e.g.*, NN-interpolation + conv4).

6.6 Results of Quantitative Analysis

Local edge devices are often limited by battery life and hardware area which constrains the power budget and on-chip resources available for inference [44, 148]. To support real-time deep learning applications in the landscape of rapidly evolving edge devices, low latency, high energy efficiency, and algorithm scalability become critical. When considering such applications, we define latency as the time cost incurred by upsampling a single image as real-time image upsampling applications typically process images sequentially in batch sizes of 1. Furthermore, we evaluate energy efficiency under two metrics. First, we use energy per pixel to evaluate the energy cost incurred by upsampling a single image [84]. This is defined as the total energy cost divided by the total output pixels generated and is measured in units of Joules/pixel [84]. Second, we use performance per energy to evaluate the rate of computation for every unit of energy consumed. This is defined as the total useful computations divided by the total energy cost and, as the energy analog of throughput, is measured in units of MACs/Joule [28]. To analyze algorithm scalability, we use activation reuse since an increased computation-to-communication ratio implies a greater opportunity for performance improvements when compute and memory resources increase [28, 84, 167].

As shown in Table 6.2, the compute (C) and activation requirements (A) of the sub-pixel

convolution (C-SP) and nearest neighbor resize convolution (C-NN) are equal[5]. With activations dominating memory requirements (M) and, therefore, data transfer penalties, the variations in weight requirements (W) have negligible impact on time and energy costs. By translating learned convolution kernels for deconvolution inference, the kernel transformations introduced in Section 6.4 remove the reliance of C-SP and C-NN on the memory-intensive feature map transformations that increase activate requirements. We highlight the following implications of this reduction in memory accesses, assuming REVD2 as the low-level deconvolution inference algorithm. Using the quantitative models detailed in Section 6.5, we analyze and compare the properties of convolution-based image upsampling algorithms using metrics of time and energy.

As in Section 6.5.3, we again consider the single-image super resolution network provided by PyTorch [127] and shown in Figure 6.3 to upsampling the 320x480 image selected from the BSD300 [114] dataset by a factor of 3. Note that, because the network is trained to do so using the sub-pixel convolution (*e.g.*, conv4 + pixel shuffle), we use this deep neural network to analyze C-SP and C-NN. To analyze C-NN and D-NN, we design an analogous neural network architecture in which the sub-pixel convolution is substituted with NN resize convolution (*e.g.*, NN-interpolation + conv4).

1. **Latency.** As shown in Figure 6.7a, the detrimental impact of memory-intensive feature map transformations on C-SP and C-NN increases with upsampling factor (r). Translating C-SP to D-SP removes its reliance on the pixel shuffle post-processing. Translating C-NN to D-NN not only avoids NN-interpolation pre-processing but also significantly reduces the total MACs required to upsample an image, as discussed in Section 6.4. To analyze the impact these kernel transformations have on system latency, we use the time cost model detailed in Section 6.5.1 and consider the deep neural network depicted by Figure 6.3. Upsampling the selected image using D-SP decreases system latency by 2.1x when compared to C-SP and using D-NN decreases system latency by 2.8x when compared

[5]For simplicity of analysis, we ignore the impact of address calculations and modulo arithmetic, which are often long-latency instructions.

to C-NN.

2. **Energy per Pixel.** Energy consumption is dominated by data movement [76]. With activation requirements dominating memory accesses, removing the memory-intensive feature map transformations of C-SP and C-NN reduces the energy cost of upsampling an image. Figure 6.7b shows how the energy cost of each algorithm increases with upsampling factor (r). As each algorithm is generating an $rH \times rH \times C$ output image, this is also the relative increase in energy per pixel. To analyze the impact these kernel transformations have on energy per pixel, we use the energy cost model detailed in Section 6.5.1 and consider the neural network depicted by Figure 6.3. Upsampling the selected image using D-SP decreases energy per pixel by 2.1x when compared to C-SP and using D-NN decreases energy per pixel by 2.7x when compared to C-NN.

3. **Performance per Energy.** As discussed in Section 6.5.2, activation reuse is defined as the ratio of useful compute work to activation requirements. The activation reuse of convolution-based upsampling algorithms is tightly correlated with performance per energy (PPE) [84]. Figure 6.8 shows how the activation reuse of each convolution-based upsampling algorithm increases with upsampling factor (r). While removing the reliance on memory-intensive feature map transformations improves the PPE of D-SP, it reduces the PPE of D-NN as the reduction in MACs significantly outweighs the reduction in memory accesses. This imbalanced reduction ultimately renders D-NN memory-bound as the amount of compute work grows slower than the amount of memory accessed.

4. **Scalability.** The ratio of useful computations to memory accesses (*i.e.*, arithmetic intensity) implies the scalability of an algorithm. As described in Section 6.5.3, algorithms with an arithmetic intensity lower than the machine balance point are ultimately bound by memory bandwidth [167]. Algorithms with an arithmetic intensity higher than the machine balance point are ultimately bound by compute resources and are more likely to see performance gains as resources scale [28]. For each convolution-based upsampling algorithm, we use

the roofline models depicted in Figure 6.9 to visualize the relationships of their arithmetic intensities to the time and energy machine balance points. We further show how these relationships change with upsampling factor in Figure 6.10. Unlike C-SP, C-NN, and even D-NN, D-SP remains compute-bound in both time and energy as upsampling factor increases. With compute work dominating memory accesses, the increased arithmetic intensity of D-SP implies increased time and energy efficiency as compute resources scale [28].

These trends do not hold for all selections of deconvolution formulations. Deep learning frameworks commonly use the fractionally strided deconvolution (STRD) formulation to leverage unmodified convolution accelerators [1, 127]. However, the zero-insertion requirements on the input feature maps exponentially increase the data transfer penalties. Figure 6.6 shows how time and energy costs increase with upsampling factor. Once again considering the deep neural network depicted in Figure 6.3, using REVD2 as the low-level deconvolution algorithm in place of STRD reduces latency by 19x and reduces energy per pixel by 6.8x, as estimated using the time and energy cost models detailed in Section 6.5.1. As shown in Table 5.1, the compute and activation requirements of transforming deconvolution to convolution (TDC) are the same as REVD2. As such, the TDC zero-insertion requirements on the learned kernels have negligible impact on high-resolution images. However, as discussed in Section 5.5, the functional correctness of TDC breaks down when attempting to tile the workloads in sizes not evenly divisible by the stride (S). Additionally, the address calculations are heavily modulo-based when minimizing the cost of stitching together the data-independent output tiles from TDC. These penalties respectively lead to under-utilized hardware resources and longer latency but do not show in this simplified quantitative model of hardware performance. We aim to quantify this impact in future work.

6.7 Conclusions and Future Work

Cloud computing systems can have nearly limitless resources, making them ideal for resource-intensive tasks such as data storage, data processing, and model training. However, real-time deep learning applications often require edge computing frameworks to improve system latency and energy efficiency by executing inference locally on edge devices without reliance on a stable internet connection [44, 148].

We propose a novel edge computing paradigm for real-time convolution-based image upsampling applications that separately considers algorithms for training in the cloud and inference at the edge. The use of sub-pixel or resize convolution is confined to training in the cloud to minimize the data transfer penalties incurred by the memory-intensive feature map transformations they require for inference. The learned convolution kernels are then transformed into deconvolution kernels without sacrificing the image fidelity learned in training. These deconvolution kernels are then deployed for inference at the edge using our improved reverse looping deconvolution algorithm, which we refer to as REVD2. We summarize our results as follows:

1. Using quantitative models of time and energy, we show that executing deconvolution inference at the edge with REVD2 improves both system latency and energy efficiency when compared to sub-pixel or resize convolution counterparts (Section 6.5).

2. When optimizing for energy efficiency and scalability, we show that training with sub-pixel convolution in the cloud and then transforming the learned kernels using the weight shuffle for deconvolution inference at the edge minimizes the pressure on memory bandwidth and maximizes energy efficiency (Section 6.6).

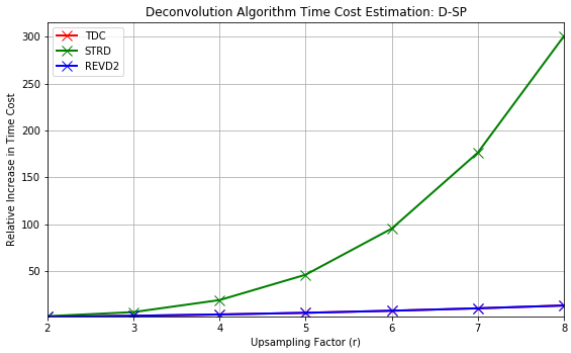
3. When optimizing for latency and energy consumption, we show that training with nearest neighbor resize convolution in the cloud and then transforming the learned kernels using the weight convolution for deconvolution inference at the edge minimizes the time and

energy costs incurred by upsampling an image (Section 6.6).

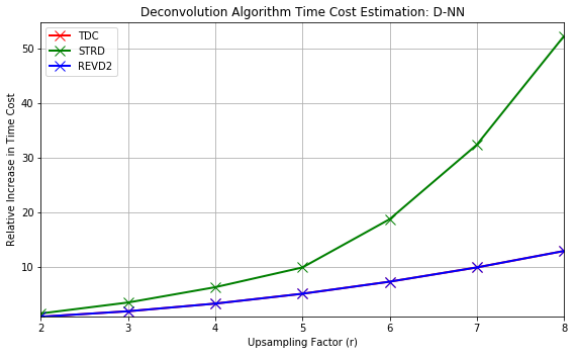
4. To upsample images selected from the BSD300 [114] dataset, we consider the pre-trained single-image super resolution network provided by PyTorch [127] and estimate that using REVD2 for deconvolution inference at the edge improves system latency by 2.1x or 2.8x and energy efficiency by 2.1x or 2.7x when compared to sub-pixel or resize convolution counterparts, respectively (Section 6.6).

In future work, we aim to extend our analyses to further quantify each algorithm's adaptability to the underlying constraints of a target hardware architecture. Additionally, we hope to build an energy-efficient accelerator designed to exploit the parallelism that is exposed from REVD2. Code for each algorithm discussed in this paper can be found at https://github.com/icolbert/upsampling.

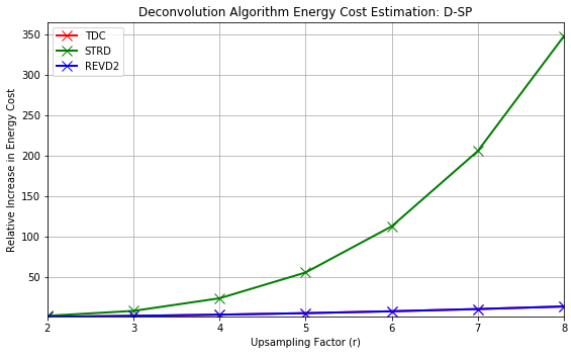
Acknowledgements: This chapter is based on material as it appears in the 2021 IEEE Access Journal (Ian Colbert, Kenneth Kreutz-Delgado, and Srinjoy Das, "An Energy-Efficient Edge Computing Paradigm for Convolution-Based Image Upsampling"). The dissertation author was the primary investigator and author of this paper. The authors would like to thank Xinyu Zhang for his insightful discussions. The constructive comments of the anonymous reviewers and the associate editor are also acknowledged.



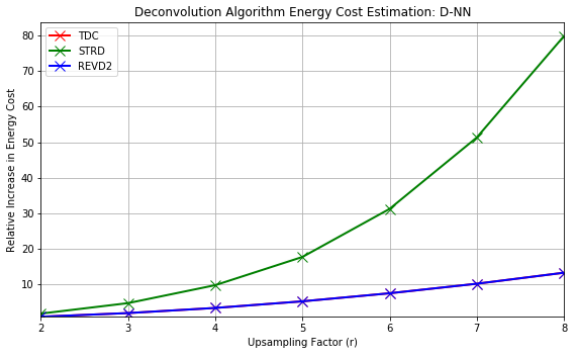
(a) D-SP: Time Cost



(b) D-NN: Time Cost

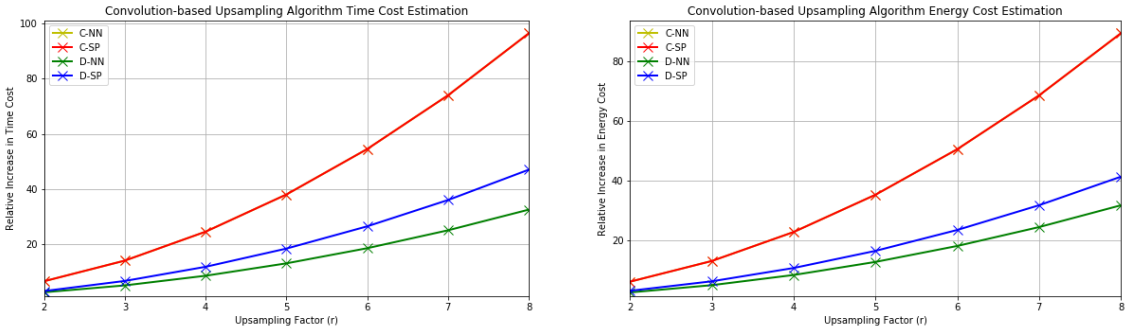


(c) D-SP: Energy Cost



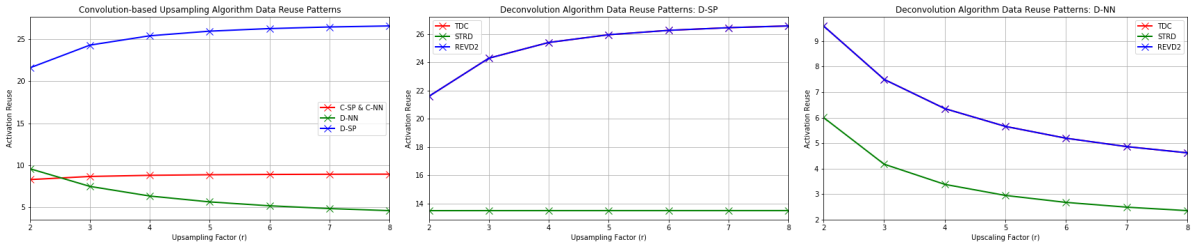
(d) D-NN: Energy Cost

Figure 6.6. We analyze the effect of workload size on time and energy costs for each low-level deconvolution algorithm. Here, we show how these costs increase as a function of upsampling factor (r). For clarity, we normalize all estimates to the cost of using REVD2 when $r = 1$ such that $I_H = O_H = H$. Intuitively, as r increases, the workload increases, thus increasing the time and energy costs of each algorithm. The left column, (*i.e.*, (a) and (c)) shows how time and energy costs increase for each of the three low-level deconvolution algorithms under consideration (*i.e.*, REVD2, TDC, and STRD) assuming the high-level algorithm was derived from sub-pixel convolution (D-SP). Similarly, the right column (*i.e.*, (b) and (d)) shows how time and energy costs increase for deconvolution as derived from nearest neighbor resize convolution (D-NN). As further discussed in Section 6.5.1, fractionally strided deconvolution (STRD) formulation does not scale as well as REVD2 and TDC, whose costs are nearly identical as the upsampling factor (r) increases.



(a) Time Cost **(b)** Energy Cost

Figure 6.7. We analyze the effect of workload size on time and energy costs for each high-level convolution-based image upsampling algorithm. For clarity, we normalize all estimates to the cost of using deconvolution as derived from sub-pixel convolution (D-SP) when $r = 1$ such that $I_H = O_H = H$ and assume REVD2 as the low-level deconvolution algorithm. Again, each experiment assumes a $1024 \times 1024 \times 3$ input image upsampled using a standard 3×3 kernel. As shown in Table 6.2, the compute (C) and activation (A) requirements of C-SP and C-NN are equal. With data movement dominated by activations, any variance in weights (W) is minimally impactful, making their costs nearly identical as the upsampling factor (r) increases.



(a) **(b)** D-SP **(c)** D-NN

Figure 6.8. We use the compute and memory requirements from Tables 5.1 and 6.2 to calculate the activation reuse for each convolution-based image upsampling algorithm. Following the work of Jha *et al.* [84], we use this metric to estimate the energy efficiency of each algorithm as a function of upsampling factor (r). Each experiment assumes a $1024 \times 1024 \times 3$ input image upsampled using a standard 3×3 kernel. Note that the compute (C) and activation (A) requirements of the high-level algorithms C-SP and C-NN are equal as are those of the low-level algorithms REVD2 and TDC. Thus, their respective activation reuse, measured as the ratio of compute operations to bytes of activations transferred, are nearly identical as upsampling factor (r) increases.

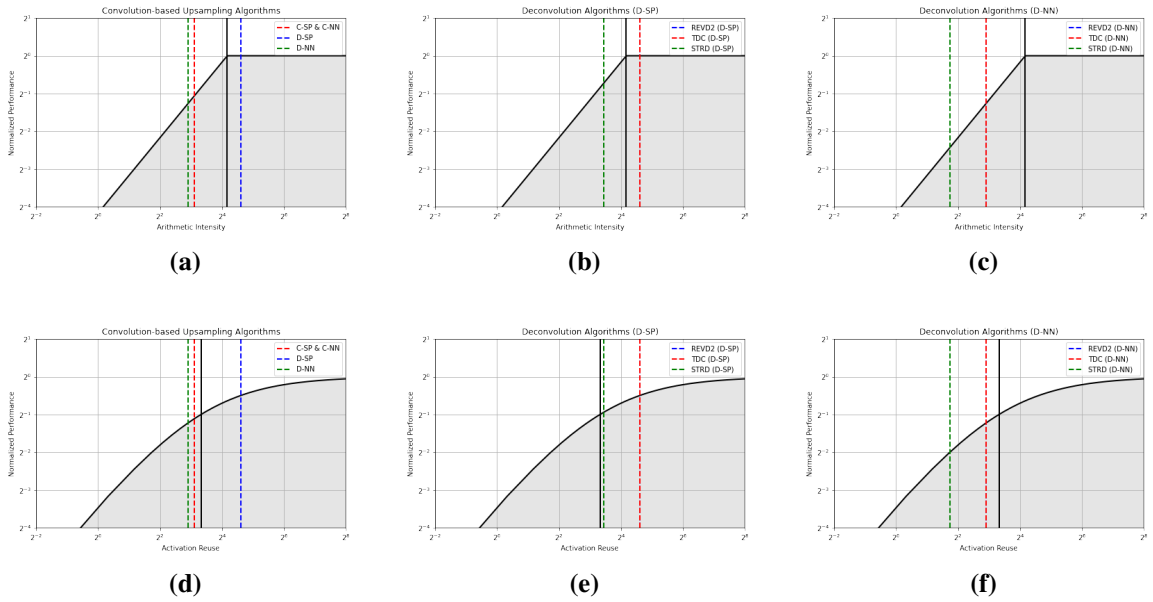
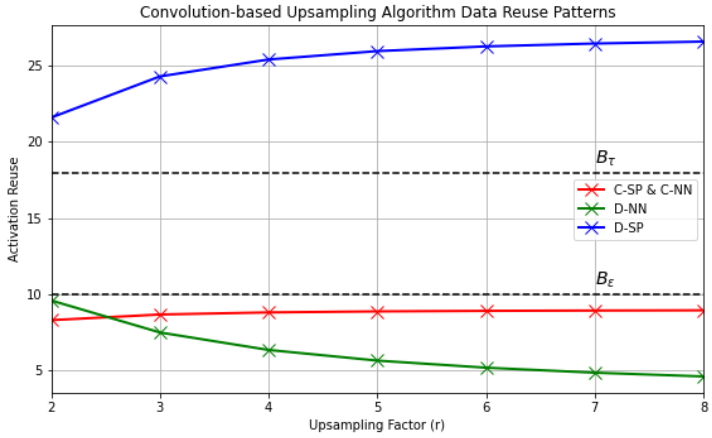


Figure 6.9. We use the roofline models of time (top row) and energy (bottom row) to visualize the performance of the pre-trained single-image super resolution example shown in Figure 6.3 provided by PyTorch [127]. Because the network is trained to upsampling an image by a factor of 3 using the sub-pixel convolution (*e.g.*, conv4 + pixel shuffle), we use this deep neural network to analyze C-SP and C-NN. To analyze C-NN and D-NN, we design an analogous neural network architecture in which the sub-pixel convolution is substituted with NN resize convolution (*e.g.*, NN-interpolation + conv4). For each experiment, we use the 320x480 image selected from the BSD300 [114] dataset and assume the GPU summarized in Table 6.1. In each plot, the black curve is the roofline model and the vertical black line is its respective balance point using values provided by [28]. The roofs of each model are normalized to peak performance. As discussed in Section 6.5.2, we use arithmetic intensity for the time roofline model [167] and activation reuse for the energy roofline model [28, 84]. Note that REVD2 and TDC have nearly the same activation reuse or arithmetic intensity.



(a)

Figure 6.10. To visualize algorithm scalability, we add the time balance point (B_τ) and energy balance point (B_ε) of the GPU summarized in Table 6.1 to the data reuse patterns of Figure 6.8a. The memory-intensive feature map transformations of C-SP and C-NN render each algorithm memory-bound in both time and energy. Avoiding the pixel shuffle post-processing of C-SP drastically increases the efficiency of D-SP for inference. The increased activation reuse implies increased time and energy efficiency as compute resources scale because the overwhelming majority of work is dedicated to computations rather than memory accesses [28]. For D-NN, the significant reduction in MACs outweighs the reduced memory accesses, ultimately rendering it memory-bound in both time and energy as the memory pressure increases faster than the compute workload as upsampling factor increases. Unlike C-SP, C-NN, and even D-NN, D-SP remains compute-bound in both time and energy as upsampling factor (r) increases.

Chapter 7

Hardware Acceleration for Deconvolution Inference

When trained as generative models, deep learning algorithms have shown exceptional performance on tasks involving high-dimensional data such as image denoising and super resolution. In an increasingly connected world dominated by mobile and edge devices, there is surging demand for these algorithms to run locally on embedded platforms. FPGAs, by virtue of their re-programmability and low-power characteristics, are ideal candidates for these edge computing applications. As such, we design a spatiotemporally parallelized hardware architecture capable of accelerating a deconvolution algorithm optimized for power-efficient inference on a resource-limited FPGA. We propose this FPGA-based accelerator to be used for deconvolutional neural network (DCNN) inference in low-power edge computing applications. To this end, we develop methods that systematically exploit micro-architectural innovations, design space exploration, and statistical analysis. Using an AMD-Xilinx PYNQ-Z2 FPGA, we leverage our architecture to accelerate inference for two DCNNs trained on the MNIST and CelebA datasets using the Wasserstein GAN framework. On these networks, our FPGA design achieves a higher throughput-to-power ratio with lower run-to-run variation when compared to the NVIDIA Jetson TX1 edge computing GPU.

7.1 Introduction

Generative models are widely used as a means of parameterizing distributions of high-dimensional signals and structures. Among the various types of generative models, the generative adversarial network (GAN) first proposed by Goodfellow *et al.* [57] yields superior performance on applications such as image generation, super resolution, and language modeling [125]. The learning strategy of the GAN jointly optimizes a generator G and a discriminator D. While the generator G is trained to minimize the distance between the ground truth distribution P_g and the model-parameterized distribution P_θ, the discriminator D is trained to separate P_g from P_θ. Although training optimizes both G and D, only the generator G is needed for inference when drawing samples from P_θ.

The typical GAN framework shown in Fig. 7.1 involves convolution layers, where D is a convolutional neural network (CNN) and G is a deconvolutional neural network (DCNN). Traditionally, these networks are deployed on CPUs and GPUs using cloud computing infrastructures. However, the proliferation of applications for mobile and edge computing has created new opportunities to deploy these models on embedded hardware for local inference. In contrast to CPUs and GPUs, FPGAs offer large-scale fine-grained parallelism and provide consistent power-efficient throughput, making them well-suited for these edge computing applications [15].

In this paper, we consider DCNN inference acceleration using a resource-limited AMD-Xilinx PYNQ-Z2 FPGA. We benchmark our implementation against the NVIDIA Jetson TX1 GPU, a processor heavily optimized for edge computing applications, and achieve a superior throughput-to-power ratio. The contributions of this paper are as follows:

- Significant enhancements over the algorithm proposed by [185] that reduce resource utilization, improve dataflow, and exploit memory hierarchy
- A spatiotemporally parallelized hardware architecture specifically designed to exploit these algorithmic innovations for power-efficient acceleration of DCNN inference

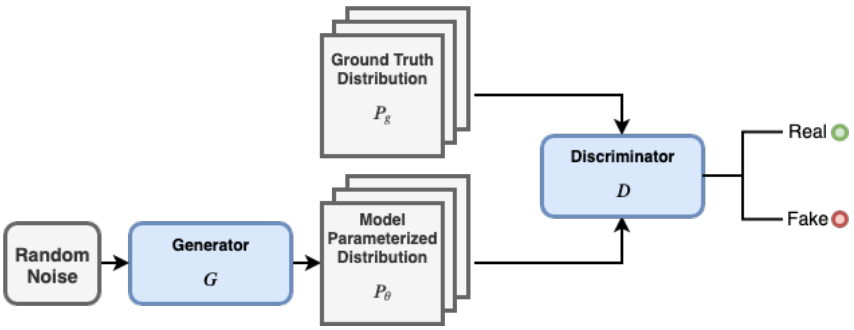


Figure 7.1. We visualize the generative adversarial network architecture [57]. After training on the cloud, we map generator G onto local hardware for low-power inference at the edge.

- An application of high-dimensional statistical analyses to balance the trade-off between hardware performance and generative quality when exploring network sparsity

7.2 Related Works

Previous works take architectural and algorithmic approaches to accelerate deconvolution workloads. The authors in [176] and [177] reformulate the deconvolution operation as a sparse convolution and build complex architectures that unify SIMD and MIMD execution models. Wang *et al.* [159] also use the zero-insertion deconvolution algorithm, approaching the problem by parallelizing over a uniform 2D systolic array hardware architecture to accelerate both 2D and 3D DCNNs. Liu *et al.* [104] propose a tiling method with a memory-efficient architecture that limits off-chip memory accesses at the cost of increased resource utilization via on-chip buffering. Chang *et al*. [21, 22] propose an accelerator that transforms the deconvolution operation into a convolution (TDC). Given a stride of S, this operation requires S^2 as many filters and potentially zero-padding both the learned weights. To improve dataflow, Tu *et al.* [152] explore the on-chip re-stitching of the disjoint output feature maps resulting from the TDC method. Mao *et al.* [112] adapt this method in a piecewise manner to handle the load imbalance resulting from zero-padding at the cost of increased hardware complexity. The algorithm first proposed by Zhang *et al.* [185] avoids the zero-insertion and zero-padding requirements of the methods outlined above. We adapt this algorithm to a parallel hardware architecture as described in Sections 7.3 and 7.4.

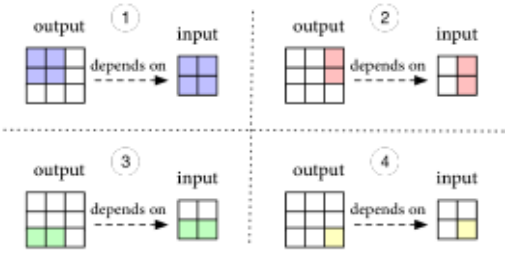


Figure 7.2. Visualization from [185] for mapping input and output blocks in reverse looping deconvolution algorithm.

7.3 Deconvolution Algorithm

Standard deconvolution arithmetic traverses the input space, which requires a summation of regions that overlap in the output space [48]. When realized in hardware, accumulating over these overlapping regions can require complex dataflow and increase resource utilization via on-chip buffering [22, 104, 185]. To circumvent this, Zhang *et al.* [185] redesign the deconvolution algorithm to directly loop over the output space at the cost of the expensive modulo arithmetic required to calculate dependent input pixels. We propose the following three enhancements to adapt this reverse looping algorithm to a spatiotemporally parallelized hardware architecture.

(1) Preprocessing modulo arithmetic. Standard deconvolution arithmetic calculates the indices of dependent output pixels o_h from input index i_h using weight index k_h, stride S, and padding P, as shown in Eq. 7.1. Here, tiling along the input space leads to overlapping blocks in the output space, creating communication overhead [22, 152, 185].

$$o_h = i_h \times S + k_h - P \tag{7.1}$$

To avoid this, Zhang *et al.* [185] use the mapping in Fig. 7.2 to loop over the output space and determine i_h using Eq. 7.2.

$$i_h = \frac{o_h + P - k_h}{S} \tag{7.2}$$

When $S > 1$, Eq. 7.2 yields fractional values. To ensure functional correctness, Zhang *et al.* [185]

Algorithm 13. We provide pseudocode for our reverse looping deconvolution dataflow, where each kernel loads inputs, weights, and offsets into local memory before computing each output block.

$\mathbf{y} \leftarrow$ initializeToBias()
for $i_c = 0$, i_c++, while $i_c < I_C$ **do**
  $\mathbf{x} \leftarrow$ loadInputBlock()
  $\mathbf{w} \leftarrow$ loadWeightBlock()
  for $k_h = 0$, k_h++, while $k_h < K$ **do**
    for $k_w = 0$, k_w++, while $k_w < K$ **do**
      $w = \mathbf{w}[k_h, k_w]$
      $f_h = \text{loadOffset}(k_h)$
      $f_w = \text{loadOffset}(k_w)$
      for $\widehat{o}_h = 0$, $\widehat{o}_h$+ =S, while $\widehat{o}_h < T_{O_H}$ **do**
        for $\widehat{o}_w = 0$, $\widehat{o}_w$+ =S, while $\widehat{o}_w < T_{O_W}$ **do**
          $o_h = \widehat{o}_h + f_h$
          $o_w = \widehat{o}_w + f_w$
          $i_h = (o_h + P - k_h)/S$
          $i_w = (o_w + P - k_w)/S$
          $\mathbf{y}[o_h, o_w] \leftarrow w \times \mathbf{x}[i_h, i_w]$
pushOutputBlock($\mathbf{y}$)

propose a stride hole skipping technique, adding an offset value f_h given by Eq. 7.3.

$$f_h = \textbf{mod}(S - \textbf{mod}(P - k_h, S), S) \tag{7.3}$$

However, the resulting input pixel calculation given by Eq. 7.4 relies on modulo arithmetic which increases resource utilization and power consumption when implemented in hardware.

$$i_h = \frac{o_h + P - k_h + f_h}{S} \tag{7.4}$$

Observing that, in Eq. 7.3, f_h is only dependent on k_h, we pre-compute and cache these offsets for each value of k_h. This process reduces the number of modulo operations to $2K$, where K is the weight filter size. This minimizes resource utilization and on-chip memory as K tends to be small.

(2) Dataflow Optimization. Loop interchange is an algorithm-level optimization that can be applied to improve the sequential computation order of operations [110]. We reorder the loops of the deconvolution arithmetic in [185] to sequentially traverse the weight space and maximize

data reuse. Increasing weight-level data reuse also increases the impact of zero-skipping - a conditional execution paradigm that eliminates redundant operations by only processing non-zero elements.

Additionally, we exploit the opportunities for data-level parallelism exposed by directly looping over the output space. Unlike the standard deconvolution algorithm, which suffers from the overlapping sum problem, the output space of the reverse looping deconvolution can be tiled into smaller batches to execute concurrently on a parallelized hardware architecture. When the size of the output feature space increases owing to the upsampling nature of deconvolution operations, the workloads and memory requirements remain constant, simplifying hardware design requirements.

(3) Decoupling external memory accesses from compute operations. Reverse looping deconvolution arithmetic using [185] produces a non-sequential external memory access pattern over the input space. To mask any resulting overhead, we decouple all external memory accesses from compute operations to allow for the cascaded execution of these sub-tasks on a pipelined hardware architecture and restrict non-sequential memory access patterns to faster on-chip memory. This is done by first computing the pixel addresses of an input block using Eq. 7.4, then sequentially reading these addresses from external memory, and finally caching the data on-chip to be distributed. To do this, we determine the tile size T_{I_H} of the input block needed for each output block from the output tiling factor T_{O_H} and the layer parameters using Eq. 7.5. The resulting deconvolution kernel given by Algorithm 13 can then continuously compute $T_{O_H} \times T_{O_W}$ output blocks with a non-sequential access pattern over locally cached $T_{I_H} \times T_{I_W}$ input blocks using $K \times K$ weight blocks as the next set of inputs are fetched from external memory using sequential reads.

$$T_{I_H} = \max(i_h) - \min(i_h) = \left\lceil \frac{T_{O_H}}{S} \right\rceil + \left\lceil \frac{K}{S} \right\rceil \tag{7.5}$$

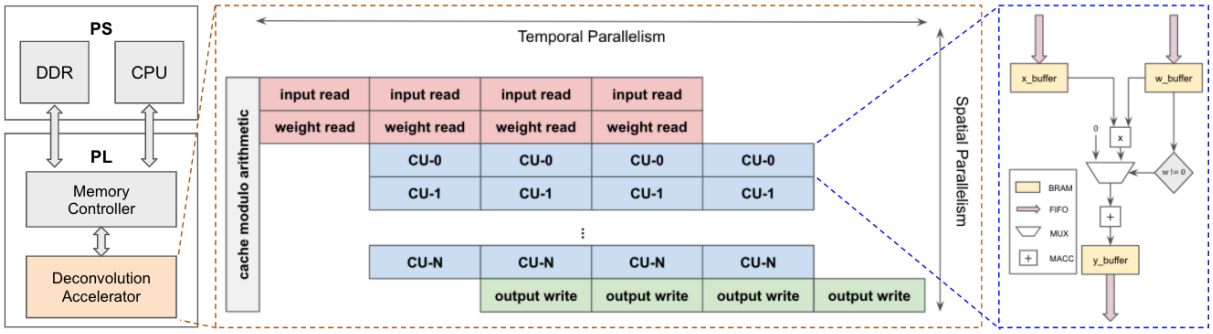


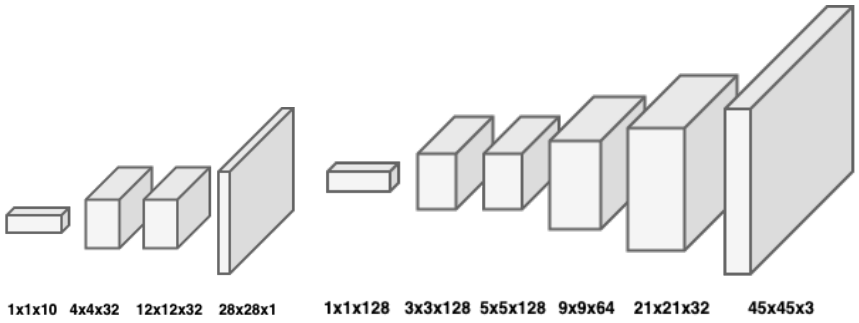
Figure 7.3. We design a spatiotemporally parallelized hardware architecture customized to accelerate the deconvolution algorithm proposed in Section 7.3 for low-power DCNN inference at the edge.

7.4 FPGA Hardware Architecture

To accelerate DCNN inference on an FPGA, we design a SIMD (single-instruction multiple-data) hardware architecture with replicable compute units (CUs) that exploits the opportunities for both spatial and temporal data-level parallelism that arise from the optimizations discussed in Section 7.3. As depicted in Figure 7.3, the dataflow of the deconvolution accelerator IP block is split into the three pipelined stages outlined below.

(1) Reading Inputs and Weights. The limited amount of on-chip memory is a bottleneck when accelerating large networks on a resource-limited FPGA. As such, the input feature maps and network weights are stored in off-chip DDR memory and fetched using AXI interconnects. As described in Section 7.3, decoupling external memory accesses masks the communication overhead when executed in a pipelined architecture. We separate input and weight external memory accesses into dedicated hardware blocks to concurrently read from DDR memory and stream to CUs through on-chip FIFOs. This efficient memory hierarchy is realized by on-chip buffers using BRAMs to store tiled input and weight blocks to be processed by CUs.

(2) Spatially Parallelized Compute Units. Looping over the output feature map enables



(a) MNIST DCNN **(b)** CelebA DCNN

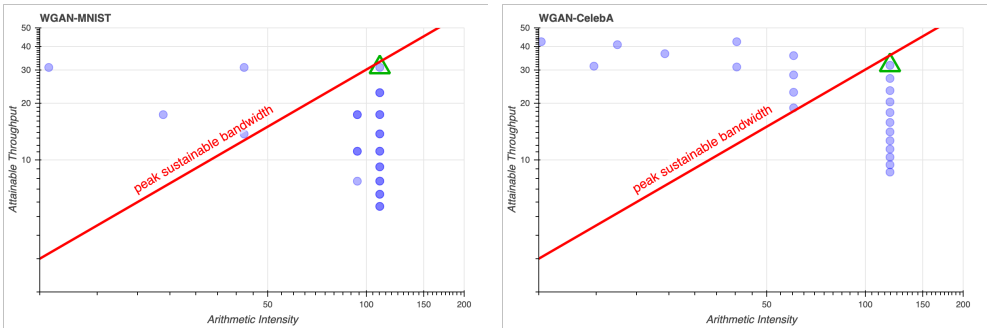
Figure 7.4. We consider the network architectures shown above for inference acceleration on low-power hardware.

partitioning deconvolution arithmetic into tiled batches that can execute concurrently across an array of CUs. The CUs follow a SIMD execution model, where each workload is dependent on blocks of inputs and weights that are sequentially streamed in through FIFOs and accumulated. The CUs each perform the deconvolution arithmetic outlined in Algorithm 13 using on-chip DSP units and the resulting $T_{O_H} \times T_{O_W}$ output block is streamed out to be written to off-chip memory. To maximize the occupancy of these CUs, we explore the design space as outlined in Section 7.5.1 to optimize the output tiling factor.

(3) Writing Output Pixels. Traversing the output space and avoiding the overlapping sum problem allows for a one-shot write to external memory for each output block computed by a CU. We dedicate a hardware block to stream the outputs from each element in the CU array to be written to external DDR memory. This minimizes communication overhead with DDR and on-chip BRAM memory requirements.

7.5 Experimental Results

We implement our architecture on an AMD-Xilinx PYNQ-Z2 board at 32-bit fixed-point precision using the Vivado Design Suite. With the available hardware resources, we synthesize the



(a) MNIST DCNN **(b)** CelebA DCNN

Figure 7.5. The optimal tiling factor T_{O_H} maximizes attainable throughput while satisfying the peak sustainable bandwidth constraint as measured by the STREAM benchmark [115].

design with 16 CUs at 125MHz in Vivado HLS using HLSLIB [41] and benchmark performance on the two DCNNs depicted in Figure 7.4. Each DCNN is trained on the MNIST and CelebA datasets using the WGAN-GP [60] framework.

7.5.1 Design Space Exploration

In this work, we explore square tiling factors over the output space such that $T_{O_H} = T_{O_W}$ and use the design space exploration methodology proposed by Zhang *et al.* [181] to optimize T_{O_H}. Because our accelerator multiplexes through the DCNN layers, we optimize T_{O_H} globally across all layers for each network architecture as a unified hardware design parameter as in [181]. Fig. 7.5 depicts all legal solutions for both the MNIST and CelebA DCNNs. Any solution to the left of the peak sustainable bandwidth slope requires a higher bandwidth than the FPGA can sustain [181]. The optimal T_{O_H} (indicated in green) maximizes attainable throughput while satisfying the hardware constraints. Table 7.1 provides the values used in this work and the resulting FPGA resource utilization. Note that the PYNQ-Z2 board is extremely resource-constrained, using only 9% of the DSP blocks used in [176] and 5% of that used in [159] and [21].

Table 7.1. AMD-Xilinx PYNQ-Z2 Resource Utilization

	T_{O_H}	DSP48s	BRAMs	Flip-Flops	LUTs
MNIST	12	134	50	43218	36469
CelebA	24	134	74	48938	40923

7.5.2 Performance-per-Watt Comparison with Edge GPU

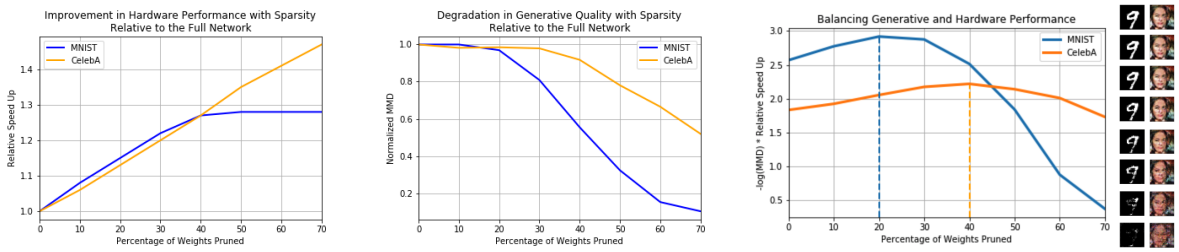
GPUs are power-hungry processors heavily optimized for large batch processing of on-chip memory [64]. Unlike the FPGA, which has been shown to provide workload-insensitive throughput with better power efficiency, the time-varying optimizations leveraged by modern GPUs give rise to a non-deterministic execution model that can rarely provide the consistent performance that is required by edge computing applications [15, 87]. Additionally, modern GPUs use hardware throttling (*i.e.*, reducing clock frequency) to lower power and cool the chip when it gets hot, further increasing run-to-run variation [123]. This makes FPGAs the more suitable choice for edge computing applications when consistent throughput and power efficiency are key requirements [15].

In our experiments, we compare the throughput to power ratio of our AMD-Xilinx PYNQ-Z2 FPGA design against the heavily optimized NVIDIA Jetson TX1 edge computing GPU. As in [122], we evaluate the GPU with PyTorch [127] using nvprof to collect performance and power numbers for each layer in each DCNN. We measure FPGA power using a USB Power

Table 7.2. We measure the mean and standard deviation (in parenthesis) of the throughput-to-power ratio (GOps/second/Watt) of each layer in each DCNN on each processor over 50 runs.

MNIST	L1	L2	L3	Total
FPGA	**2.4 (0.02)**	**3.0 (0.01)**	**2.8 (0.01)**	**2.9 (0.01)**
GPU	1.3 (0.17)	2.7 (0.42)	1.8 (0.25)	2.1 (0.18)

CelebA	L1	L2	L3	L4	L5	Total
FPGA	**4.0 (0.00)**	4.0 (0.00)	**4.0 (0.00)**	2.3 (0.00)	1.2 (0.01)	**3.9 (0.00)**
GPU	3.2 (0.66)	**4.4 (0.81)**	3.9 (0.66)	**4.4 (0.69)**	**2.2 (0.40)**	3.6 (0.31)



(a) Hardware Performance **(b)** Generative Quality **(c)** Balancing Trade-Off

Figure 7.6. While unstructured sparsity leads to FPGA speed-ups when using conditional execution paradigms like zero-skipping, removing learned parameters from a network invariably leads to degradation in generative quality. We propose a design metric to balance this trade-off.

Meter Voltage Detector and collect performance numbers using hardware counters. We compute total network throughput as the sum of the arithmetic operations of all layers divided by the sum of the execution time of all layers. Our results provided in Table 7.2 show that our design yields a higher total network throughput-to-power ratio with lower run-to-run variation when compared to the GPU for both DCNNs. As noted in [181], unified design parameters such as T_{O_H} simplify implementation cost but may be sub-optimal for some layers. We observe this behavior for the CelebA DCNN as shown in Table 7.2. In future work, we will investigate dynamically reconfiguring tiling factors to optimize dataflow per layer.

7.5.3 Sparsity Experiments

Weight pruning is a widely studied technique used to reduce network power consumption and memory footprint on mobile and edge computing platforms [63]. It's difficult for GPUs to effectively accelerate this form of unstructured sparsity as they are highly sensitive to conditional execution paradigms such as zero-skipping [15, 122]. Alternatively, FPGA performance is stable under such paradigms and can yield significant speed-ups when only executing non-zero valued computations [15, 23]. Previous works optimizing DCNN dataflow for unstructured sparsity fail to account for this degradation [23].

In our experiments, we systematically prune DCNN weights by their magnitude as done in [63]. To visualize both hardware performance and generative quality, we analyze the rates of

change of both system latency and maximum mean discrepancy (MMD) distance, respectively. MMD distance is used to compute the dissimilarity between model-parameterized distribution P_θ and ground truth distribution P_g and, in practice, is empirically estimated by drawing independent samples $\{\boldsymbol{x_1}, \cdots, \boldsymbol{x_n}\} \sim \mu$ and $\{\boldsymbol{y_1}, \cdots, \boldsymbol{y_n}\} \sim \nu$ from distributions P_θ and P_g, respectively, where kernel k maps to a reproducing kernel Hilbert space [19, 59]. It follows that the MMD distance given by the equation below is zero if and only if the distributions are identical. Here, we explore the use of MMD with the standard Gaussian kernel $k(\boldsymbol{x}, \boldsymbol{x}') = \exp(\|\boldsymbol{x} - \boldsymbol{x}'\|^2)$ using the Euclidean distance, selecting the median euclidean distance between ground truth samples as the bandwidth [59].

$$\mathrm{MMD}_k(\mu, \nu) = \mathbb{E}_{\mu,\mu}[k(X, X')] + \mathbb{E}_{\nu,\nu}[k(Y, Y')] - 2\mathbb{E}_{\mu,\nu}[k(X, Y)]$$

Pruning more weights yields higher speed-ups when skipping computations with zero-valued weights, as shown in Fig 7.6-a. However, as shown in Fig 7.6-b, the generative quality decreases with added sparsity. To balance the trade-off between hardware performance and generative quality, we propose an optimization metric given by Eq. 7.6. Here, t_0 and d_0 denote the execution time and MMD distance with respect P_g using the full weight matrix θ_0 while t_p and d_p denote that of the sparse matrix θ_p where d_i is given by $\mathrm{MMD}(P_g, P_{\theta_i})$. Multiplying the rate of change of system latency and MMD distance leads to a concave optimization curve with a peak representing the sparsity level that balances image quality with execution time.

$$\frac{d_0}{d_p} \times \frac{t_0}{t_p} \tag{7.6}$$

7.6 Conclusions and Future Work

In this paper, we adapt the deconvolution algorithm first proposed in [185] to a parallelized execution model by reducing resource utilization, improving data flow, and exploiting memory hierarchy. We design a spatiotemporally parallelized hardware architecture to accelerate this algorithm for DCNN inference on an AMD-Xilinx PYNQ-Z2 FPGA. For edge computing applications when consistent throughput and power efficiency are key requirements, we show that

this resource-limited FPGA achieves a higher throughput-to-power ratio with lower run-to-run variation than the NVIDIA Jetson TX1 edge computing GPU. To balance the trade-off between generative quality and hardware performance, we propose an MMD-based optimization metric when exploring unstructured sparsity. In future work, we will adapt this architecture to other GANs and investigate the effect of bit width reduction on hardware performance and generative quality.

Acknowledgements: This chapter is based on unpublished material (Ian Colbert, Jake Daly, Kenneth Kreutz-Delgado, and Srinjoy Das, "A Competitive Edge: Can FPGAs Beat GPUs at DCNN Inference Acceleration in Resource-Limited Edge Computing Applications?"". The dissertation author is the primary investigator and author of this paper. This chapter was supported in part by NSF awards CNS1730158, ACI-1540112, ACI-1541349, OAC-1826967, the University of California Office of the President, and the California Institute for Telecommunications and Information Technology's Qualcomm Institute (Calit2-QI). We would also like to thank Parimal Patel and Stephen Neuendorffer at AMD-Xilinx and Byungheon Jeon at UC San Diego.

Chapter 8

Conclusion

In recent years, deep learning-based solutions have achieved state-of-the-art performance on image upsampling tasks by training deep neural networks (DNNs) on large annotated datasets (*e.g.*, super resolution [144, 149], image generation [57, 74, 89], and style transfer [134, 190]). However, the adoption of these solutions in real-time applications is limited by the deployment costs of the resulting models as end-user devices impose significant compute and memory constraints on inference pipelines. To address this challenge, researchers and practitioners have proposed methods to reduce inference costs without sacrificing model quality. While many of these works focus on DNNs designed for image downsampling, we show that specifically tailoring optimizations for image upsampling workloads can lead to more efficient and effective deployment of deep learning-based image upsampling networks.

This dissertation detailed specialized hardware-aware deep learning techniques, inference algorithms, and compute graph transformations designed for the core layers within deep learning-based image upsampling networks (*e.g.*, sub-pixel convolutions, resize convolutions, and deconvolutions). By integrating our proposed inference optimization techniques into an end-to-end pipeline, we can achieve significant improvements in hardware performance and efficiency without sacrificing model quality. Furthermore, by co-designing our software optimizations along with our specialized hardware architecture, we further reduce resource utilization and improve dataflow on parallelizable platforms such as FPGA-based inference accelerators,

leading to a more effective deployment on resource-constrained platforms such as mobile or edge devices.

In summary, our contributions address the deployment costs of deep neural networks (DNNs) designed for real-time computer vision applications while improving power efficiency, throughput, and resource utilization without adversely impacting model quality. As deep learning research continues to progress at an accelerating pace, the solution design space continues to expand. By focusing on improving the hardware performance and efficiency of the core image upsampling algorithms, we develop adaptable solutions that keep pace with this rapidly evolving landscape since most state-of-the-art computer vision models consist of the same fundamental building blocks [74, 134, 136]. In future work, we aim to explore the intersection of this work with automated design space exploration techniques such as neural architecture search (NAS) algorithms.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Kamel Abdelouahab, Maxime Pelcat, and François Berry. Accelerating the CNN inference on FPGAs. In Deep Learning in Computer Vision, pages 1–40. CRC Press, 2020.

[3] Jan Achterhold, Jan Mathias Koehler, Anke Schmeink, and Tim Genewein. Variational network quantization. In International Conference on Learning Representations, 2018.

[4] Eirikur Agustsson, Fabian Mentzer, Michael Tschannen, Lukas Cavigelli, Radu Timofte, Luca Benini, and Luc V Gool. Soft-to-hard vector quantization for end-to-end learning compressible representations. Advances in neural information processing systems, 30, 2017.

[5] Eirikur Agustsson and Radu Timofte. Ntire 2017 challenge on single image super-resolution: Dataset and study. In Proceedings of the IEEE conference on computer vision and pattern recognition workshops, pages 126–135, 2017.

[6] Andrew Aitken, Christian Ledig, Lucas Theis, Jose Caballero, Zehan Wang, and Wenzhe Shi. Checkerboard artifact free sub-pixel convolution: A note on sub-pixel convolution, resize convolution and convolution resize. arXiv preprint arXiv:1707.02937, 2017.

[7] AMD-Xilinx. FINN: Dataflow compiler for QNN inference on FPGAs. https://github.com/Xilinx/finn, 2023. Accessed: 2023-01-19.

[8] AMD-Xilinx. Vivado design suite user guide: Notes for higher performance FPGA design. https://docs.xilinx.com/r/en-US/ug897-vivado-sysgen-user/Use-Saturation-Arithmetic-and-Rounding-Only-When-Necessary, 2023. Accessed: 2023-01-19.

[9] H. Amin, K.M. Curtis, and B.R. Hayes-Gill. Piecewise linear approximation applied to nonlinear function of a neural network. In Circuits, Devices & Syst., IEE Proc., volume 144, pages 313–317. IET, 1997.

[10] Jeremy Appleyard, Tomas Kocisky, and Phil Blunsom. Optimizing the performance of recurrent neural networks on GPUs. arXiv preprint arXiv:1604.01946, 2016.

[11] Caglar Aytekin, Francesco Cricri, and Emre Aksu. Compressibility loss for neural network weights. arXiv preprint arXiv:1905.01044, 2019.

[12] Johan Barthélemy, Nicolas Verstaevel, Hugh Forehead, and Pascal Perez. Edge-computing video analytics for real-time traffic monitoring in a smart city. Sensors, 19(9):2048, 2019.

[13] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. arXiv preprint arXiv:1308.3432, 2013.

[14] Marco Bevilacqua, Aline Roumy, Christine Guillemot, and Marie Line Alberi-Morel. Low-complexity single-image super-resolution based on nonnegative neighbor embedding. Proceedings of the 23rd British Machine Vision Conference (BMVC), 2012.

[15] Saman Biookaghazadeh, Ming Zhao, and Fengbo Ren. Are FPGAs suitable for edge computing? In USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18), 2018.

[16] C.M. Bishop. Neural networks for pattern recognition. Oxford university press, 1995.

[17] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? Proceedings of machine learning and systems, 2:129–146, 2020.

[18] Michaela Blott, Thomas B Preusser, Nicholas J Fraser, Giulio Gambardella, Kenneth O'Brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. ACM Transactions on Reconfigurable Technology and Systems (TRETS), 11(3):1–23, 2018.

[19] Ali Borji. Pros and cons of GAN evaluation measures. Computer Vision and Image Understanding, 179:41–65, 2019.

[20] Wen-Pu Cai and Wu-Jun Li. Weight normalization based quantization for deep neural network compression. arXiv preprint arXiv:1907.00593, 2019.

[21] Jung-Woo Chang, Saehyun Ahn, Keon-Woo Kang, and Suk-Ju Kang. Towards design methodology of efficient fast algorithms for accelerating generative adversarial networks on FPGAs. In 2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC), pages 283–288. IEEE, 2020.

[22] Jung-Woo Chang, Keon-Woo Kang, and Suk-Ju Kang. An energy-efficient FPGA-based deconvolutional neural networks accelerator for single image super-resolution. IEEE Transactions on Circuits and Systems for Video Technology, 2018.

[23] Jung-Woo Chang, Keon-Woo Kang, and Suk-Ju Kang. SDCNN: An efficient sparse deconvolutional neural network accelerator on FPGA. In 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 968–971. IEEE, 2019.

[24] Jung-Woo Chang and Suk-Ju Kang. Optimizing FPGA-based convolutional neural networks accelerator for image super-resolution. In Proceedings of the 23rd Asia and South Pacific Design Automation Conference, pages 343–348. IEEE Press, 2018.

[25] Shih-Kang Chao, Zhanyu Wang, Yue Xing, and Guang Cheng. Directional pruning of deep neural networks. Advances in Neural Information Processing Systems, 33:13986–13998, 2020.

[26] Tianlong Chen, Xuxi Chen, Xiaolong Ma, Yanzhi Wang, and Zhangyang Wang. Coarsening the granularity: Towards structurally sparse lottery tickets. In International Conference on Machine Learning, 2022.

[27] Y.-H. Chen, T. Krishna, J.S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. IEEE Journal of Solid-State Circuits, 52(1):127–138, 2017.

[28] Jee Whan Choi, Daniel Bedard, Robert Fowler, and Richard Vuduc. A roofline model of energy. In 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, pages 661–672. IEEE, 2013.

[29] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. Towards the limit of network quantization. In International Conference on Learning Representations, 2017.

[30] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. Universal deep neural network compression. IEEE Journal of Selected Topics in Signal Processing, 14(4):715–726, 2020.

[31] Ian Colbert, Jake Daly, Ken Kreutz-Delgado, and Srinjoy Das. A competitive edge: Can FPGAs beat GPUs at DCNN inference acceleration in resource-limited edge computing applications? arXiv preprint arXiv:2102.00294, 2021.

[32] Ian Colbert, Ken Kreutz-Delgado, and Srinjoy Das. AX-DBN: An approximate computing framework for the design of low-power discriminative deep belief networks. In 2019 International Joint Conference on Neural Networks (IJCNN), pages 1–9. IEEE, 2019.

[33] Ian Colbert, Kenneth Kreutz-Delgado, and Srinjoy Das. An energy-efficient edge computing paradigm for convolution-based image upsampling. IEEE Access, 9:147967–147984, 2021.

[34] Ian Colbert, Alessandro Pappalardo, and Jakoba Petri-Koenig. Quantized neural networks for low-precision accumulation with guaranteed overflow avoidance. arXiv preprint arXiv:2301.13376, 2023.

[35] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. Understanding performance differences of FPGAs and GPUs. In 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 93–96. IEEE, 2018.

[36] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[37] Bin Dai, Chen Zhu, Baining Guo, and David Wipf. Compressing neural networks using the variational information bottleneck. In International Conference on Machine Learning, pages 1135–1144. PMLR, 2018.

[38] Tao Dai, Jianrui Cai, Yongbing Zhang, Shu-Tao Xia, and Lei Zhang. Second-order attention network for single image super-resolution. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 11065–11074, 2019.

[39] S. Das, B.U. Pedroni, P. Merolla, J. Arthur, A.S. Cassidy, B.L. Jackson, D. Modha, G. Cauwenberghs, and K. Kreutz-Delgado. Gibbs sampling with low-power spiking digital neurons. In 2015 IEEE Int. Symp. Circuits & Systems (ISCAS), pages 2704–2707. IEEE, 2015.

[40] Barry de Bruin, Zoran Zivkovic, and Henk Corporaal. Quantization of deep neural networks for accumulator-constrained processors. Microprocessors and Microsystems, 72:102872, 2020.

[41] Johannes de Fine Licht and Torsten Hoefler. hlslib: Software engineering for hardware design. arXiv preprint arXiv:1910.04436, 2019.

[42] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In 2009 IEEE conference on computer vision and pattern recognition, pages 248–255. Ieee, 2009.

[43] Tim Dettmers and Luke Zettlemoyer. Sparse networks from scratch: Faster training without losing performance. arXiv preprint arXiv:1907.04840, 2019.

[44] Sauptik Dhar, Junyao Guo, Jiayi Liu, Samarth Tripathi, Unmesh Kurup, and Mohak Shah. On-device machine learning: An algorithms and learning theory perspective. arXiv preprint arXiv:1911.00623, 2019.

[45] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Image super-resolution using deep convolutional networks. IEEE transactions on pattern analysis and machine intelligence, 38(2):295–307, 2015.

[46] Chao Dong, Chen Change Loy, and Xiaoou Tang. Accelerating the super-resolution convolutional neural network. In European conference on computer vision, pages 391–407. Springer, 2016.

[47] Xuanyi Dong and Yi Yang. Network pruning via transformable architecture search. Advances in Neural Information Processing Systems, 32, 2019.

[48] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. arXiv preprint arXiv:1603.07285, 2016.

[49] Andre Esteva, Katherine Chou, Serena Yeung, Nikhil Naik, Ali Madani, Ali Mottaghi, Yun Liu, Eric Topol, Jeff Dean, and Richard Socher. Deep learning-enabled medical computer vision. NPJ digital medicine, 4(1):5, 2021.

[50] Shaoxia Fang, Lu Tian, Junbin Wang, Shuang Liang, Dongliang Xie, Zhongmin Chen, Lingzhi Sui, Qian Yu, Xiaoming Sun, Yi Shan, et al. Real-time object detection and semantic segmentation hardware system with deep learning networks. In 2018 International Conference on Field-Programmable Technology (FPT), pages 389–392. IEEE, 2018.

[51] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In International Conference on Learning Representations, 2019.

[52] Jun Fu, Jing Liu, Haijie Tian, Yong Li, Yongjun Bao, Zhiwei Fang, and Hanqing Lu. Dual attention network for scene segmentation. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, pages 3146–3154, 2019.

[53] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. arXiv preprint arXiv:1902.09574, 2019.

[54] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse GPU kernels for deep learning. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–14. IEEE, 2020.

[55] Jianfeng Gao and Joshua Goodman. Language model size reduction by pruning and clustering. In Proceedings International Conference on Spoken Language Processing, 2000.

[56] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. arXiv preprint arXiv:2103.13630, 2021.

[57] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Advances in neural information processing systems, pages 2672–2680, 2014.

[58] Allison Gray, Chris Gottbrath, Ryan Olson, and Shashank Prasanna. Deploying deep neural networks with NVIDIA TensorRT, 2017.

[59] Arthur Gretton, Karsten M Borgwardt, Malte J Rasch, Bernhard Schölkopf, and Alexander Smola. A kernel two-sample test. The Journal of Machine Learning Research, 13(1):723–773, 2012.

[60] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of Wasserstein GANs. In Advances in neural information processing systems, pages 5767–5777, 2017.

[61] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. A survey of FPGA-based neural network inference accelerators. ACM Transactions on Reconfigurable Technology and Systems (TRETS), 12(1):1–26, 2019.

[62] Gousia Habib and Shaima Qureshi. Optimization and acceleration of convolutional neural networks: A survey. Journal of King Saud University-Computer and Information Sciences, 34(7):4244–4268, 2022.

[63] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding. In International Conference on Learning Representations, 2016.

[64] Mark Harris. Mapping computational concepts to GPUs. In ACM SIGGRAPH 2005 Courses, pages 50–es. ACM, 2005.

[65] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask R-CNN. In Proceedings of the IEEE international conference on computer vision, pages 2961–2969, 2017.

[66] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.

[67] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In European conference on computer vision, pages 630–645. Springer, 2016.

[68] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In Proceedings of the IEEE international conference on computer vision, pages 1389–1397, 2017.

[69] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md Patwary, Mostofa Ali, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically. arXiv preprint arXiv:1712.00409, 2017.

[70] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. GANs trained by a two time-scale update rule converge to a local Nash equilibrium. Advances in neural information processing systems, 30, 2017.

[71] G.E. Hinton. A practical guide to training restricted Boltzmann machines. Momentum, 9(1):926, 2010.

[72] G.E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. Neural computation, 18(7):1527–1554, 2006.

[73] G.E. Hinton and R.R. Salakhutdinov. Reducing the dimensionality of data with neural networks. Science, 313(5786):504–507, 2006.

[74] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. Advances in Neural Information Processing Systems, 33:6840–6851, 2020.

[75] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. The Journal of Machine Learning Research, 22(1):10882–11005, 2021.

[76] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), pages 10–14. IEEE, 2014.

[77] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861, 2017.

[78] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. arXiv preprint arXiv:1607.03250, 2016.

[79] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 4700–4708, 2017.

[80] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. The Journal of Machine Learning Research, 18(1):6869–6898, 2017.

[81] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 2704–2713, 2018.

[82] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 2704–2713, 2018.

[83] Sambhav Jain, Albert Gural, Michael Wu, and Chris Dick. Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks. Proceedings of Machine Learning and Systems, 2:112–128, 2020.

[84] Nandan Kumar Jha. Hardware-Aware Co-Optimization of Deep Convolutional Neural Networks. PhD thesis, Indian Institute of Technology Hyderabad, 2020.

[85] Nandan Kumar Jha, Sparsh Mittal, and Sasikanth Avancha. Data-type aware arithmetic intensity for deep neural networks. Energy, 120:x109, 2021.

[86] Marc Jorda, Pedro Valero-Lara, and Antonio J Peña. Performance evaluation of cuDNN convolution algorithms on Nvidia Volta GPUs. IEEE Access, 7:70461–70473, 2019.

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

[88] Ketan Kapse. An overview of current deep learned rendering technologies. In 2021 7th International Conference on Advanced Computing and Communication Systems (ICACCS), volume 1, pages 1404–1409. IEEE, 2021.

[89] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of GANs for improved quality, stability, and variation. arXiv preprint arXiv:1710.10196, 2017.

[90] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.

[91] William R Knight. A computer method for calculating Kendall's tau with ungrouped data. Journal of the American Statistical Association, 61(314):436–439, 1966.

[92] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. arXiv preprint arXiv:1806.08342, 2018.

[93] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[94] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. Communications of the ACM, 60(6):84–90, 2017.

[95] H. Labs. http://www.hpl.hp.com/research/cacti/.

[96] H. Larochelle, M. Mandel, R. Pascanu, and Y. Bengio. Learning algorithms for the classification restricted Boltzmann machine. Journal of Machine Learning Research, 13(Mar):643–669, 2012.

[97] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 4013–4021, 2016.

[98] Yann LeCun. The MNIST database of handwritten digits. http://yann. lecun. com/exdb/mnist/, 1998.

[99] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. nature, 521(7553):436–444, 2015.

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

[101] Vladimir Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. In Soviet physics doklady, volume 10, pages 707–710. Soviet Union, 1966.

[102] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. Automatic horizontal fusion for GPU kernels. arXiv preprint arXiv:2007.01277, 2020.

[103] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. Pruning and quantization for deep neural network acceleration: A survey. Neurocomputing, 461:370–403, 2021.

[104] Shuanglong Liu, Chenglong Zeng, Hongxiang Fan, Ho-Cheung Ng, Jiuxi Meng, Zhiqiang Que, Xinyu Niu, and Wayne Luk. Memory-efficient architecture for accelerating generative networks on FPGA. In 2018 International Conference on Field-Programmable Technology (FPT), pages 30–37. IEEE, 2018.

[105] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. In International Conference on Learning Representations, 2019.

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

[107] Qian Lou and Lei Jiang. SHE: A fast and accurate deep neural network for encrypted data. Advances in Neural Information Processing Systems, 32, 2019.

[108] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In Proceedings of the IEEE international conference on computer vision, pages 5058–5066, 2017.

[109] D.L. Ly and P. Chow. A high-performance FPGA architecture for restricted Boltzmann machines. In Proc. ACM/SIGDA int. symp. Field programmable gate arrays, pages 73–82. ACM, 2009.

[110] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pages 45–54, 2017.

[111] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J. Dally. Exploring the granularity of sparsity in convolutional neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops, July 2017.

[112] Wendong Mao, Jun Lin, and Zhongfeng Wang. F-DNA: Fast convolution architecture for deconvolutional network acceleration. In 2019 IEEE Transactions On Very Large Scale Integration (VLSI) Systems., volume 28. IEEE, 2020.

[113] D. Martin, C. Fowlkes, D. Tal, and J. Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In Proc. 8th Int'l Conf. Computer Vision, volume 2, pages 416–423, July 2001.

[114] David Martin, Charless Fowlkes, Doron Tal, and Jitendra Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001, volume 2, pages 416–423. IEEE, 2001.

[115] John D McCalpin. STREAM benchmark. Link: www. cs. virginia. edu/stream/ref. html# what, 22, 1995.

[116] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 11264–11272, 2019.

[117] Diksha Moolchandani, Anshul Kumar, and Smruti R Sarangi. Accelerating CNN inference on ASICs: A survey. Journal of Systems Architecture, 113:101887, 2021.

[118] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. Data-free quantization through weight equalization and bias correction. In Proceedings of the IEEE/CVF International Conference on Computer Vision, pages 1325–1334, 2019.

[119] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization. arXiv preprint arXiv:2106.08295, 2021.

[120] Kamal Nasrollahi and Thomas B Moeslund. Super-resolution: A comprehensive survey. Machine vision and applications, 25(6):1423–1468, 2014.

[121] Renkun Ni, Hong-min Chu, Oscar Castañeda Fernández, Ping-yeh Chiang, Christoph Studer, and Tom Goldstein. Wrapnet: Neural net inference with ultra-low-precision arithmetic. In International Conference on Learning Representations ICLR 2021. OpenReview, 2021.

[122] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, et al. Can FPGAs beat GPUs in accelerating next-generation deep neural networks? In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pages 5–14, 2017.

[123] NVIDIA. NVIDIA Jetson Linux Developer Guide. NVIDIA.

[124] Augustus Odena, Vincent Dumoulin, and Chris Olah. Deconvolution and checkerboard artifacts. Distill, 1(10):e3, 2016.

[125] Zhaoqing Pan, Weijie Yu, Xiaokai Yi, Asifullah Khan, Feng Yuan, and Yuhui Zheng. Recent progress on generative adversarial networks (GANs): A survey. IEEE Access, 7:36322–36333, 2019.

[126] Alessandro Pappalardo. Xilinx/brevitas, 2021.

[127] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, Advances in Neural Information Processing Systems 32, pages 8024–8035. Curran Associates, Inc., 2019.

[128] Kimessha Paupamah, Steven James, and Richard Klein. Quantisation and pruning for neural network compression and regularisation. In 2020 International SAUPEC/RobMech/PRASA Conference, pages 1–6. IEEE, 2020.

[129] Olga Pearce, Todd Gamblin, Bronis R De Supinski, Martin Schulz, and Nancy M Amato. Quantifying the effectiveness of load balance algorithms. In Proceedings of the 26th ACM international conference on Supercomputing, pages 185–194, 2012.

[130] Tobias Pohlen, Alexander Hermans, Markus Mathias, and Bastian Leibe. Full-resolution residual networks for semantic segmentation in street scenes. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 4151–4160, 2017.

[131] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. In International Conference on Learning Representations, 2018.

[132] Shashank Prasanna, Prethvi Kashinkunti, and Fausto Milletari. TensorRT 3: Faster tensorflow inference and volta support, Aug 2020.

[133] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. arXiv preprint arXiv:1804.02767, 2018.

[134] Zhonghe Ren, Fengzhou Fang, Ning Yan, and You Wu. State of the art in defect detection based on machine vision. International Journal of Precision Engineering and Manufacturing-Green Technology, 9(2):661–691, 2022.

[135] Nesma M Rezk, Tomas Nordström, and Zain Ul-Abdin. Shrink and eliminate: A study of post-training quantization and repeated operations elimination in rnn models. Information, 13(4):176, 2022.

[136] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In International Conference on Medical image computing and computer-assisted intervention, pages 234–241. Springer, 2015.

[137] Julien Roy, Roger Girgis, Joshua Romoff, Pierre-Luc Bacon, and Christopher Pal. Direct behavior specification via constrained reinforcement learning. arXiv preprint arXiv:2112.12228, 2021.

[138] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. International journal of computer vision, 115:211–252, 2015.

[139] Charbel Sakr, Naigang Wang, Chia-Yu Chen, Jungwook Choi, Ankur Agrawal, Naresh Shanbhag, and Kailash Gopalakrishnan. Accumulation bit-width scaling for ultra-low precision training of deep networks. arXiv preprint arXiv:1901.06588, 2019.

[140] R. Salakhutdinov, A. Mnih, and G. Hinton. Restricted Boltzmann machines for collaborative filtering. In Proc. 24th int. conf. Machine learning, pages 791–798. ACM, 2007.

[141] Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. Advances in neural information processing systems, 29, 2016.

[142] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobilenetV2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 4510–4520, 2018.

[143] Claude Elwood Shannon. A mathematical theory of communication. The Bell system technical journal, 27(3):379–423, 1948.

[144] Wenzhe Shi, Jose Caballero, Ferenc Huszár, Johannes Totz, Andrew P Aitken, Rob Bishop, Daniel Rueckert, and Zehan Wang. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 1874–1883, 2016.

[145] Wenzhe Shi, Jose Caballero, Lucas Theis, Ferenc Huszar, Andrew Aitken, Christian Ledig, and Zehan Wang. Is the deconvolution layer the same as a convolutional layer? arXiv preprint arXiv:1609.07009, 2016.

[146] Rajat Kumar Soni and Binoy B Nair. Deep learning-based approach to generate realistic data for ADAS applications. In 2021 5th international conference on computer, communication and signal processing (ICCCSP), pages 1–5. IEEE, 2021.

[147] Ke Sun, Bin Xiao, Dong Liu, and Jingdong Wang. Deep high-resolution representation learning for human pose estimation. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 5693–5703, 2019.

[148] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. Proceedings of the IEEE, 105(12):2295–2329, 2017.

[149] Manu Mathew Thomas, Karthik Vaidyanathan, Gabor Liktor, and Angus G Forbes. A reduced-precision network for image reconstruction. ACM Transactions on Graphics (TOG), 39(6):1–12, 2020.

[150] Naftali Tishby and Noga Zaslavsky. Deep learning and the information bottleneck principle. In 2015 ieee information theory workshop (itw), pages 1–5. IEEE, 2015.

[151] M.T. Tommiska. Efficient digital implementation of the sigmoid function for reprogrammable logic. In Computers and Digital Techniques, IEE Proceedings-, volume 150, pages 403–411. IET, 2003.

[152] Kaijie Tu. Accelerating deconvolution on unmodified cnn accelerators for generative adversarial networks–a software approach. arXiv preprint arXiv:1907.01773, 2019.

[153] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A framework for fast, scalable binarized neural network inference. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '17, pages 65–74. ACM, 2017.

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

[155] University of California, San Diego. Nautilus. https://ucsd-prp.gitlab.io/, 2022. (Accessed on 07/18/2021).

[156] Mart Van Baalen, Christos Louizos, Markus Nagel, Rana Ali Amjad, Ying Wang, Tijmen Blankevoort, and Max Welling. Bayesian bits: Unifying quantization and pruning. Advances in neural information processing systems, 33:5741–5752, 2020.

[157] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan. AxNN: energy-efficient neuromorphic systems using approximate computing. In Proceedings of the 2014 international symposium on Low power electronics and design, pages 27–32. ACM, 2014.

[158] Mohamed Wahib and Naoya Maruyama. Scalable kernel fusion for memory-bound gpu applications. In SC'14: Proceedings of the International Conference for High-Performance Computing, Networking, Storage and Analysis, pages 191–202. IEEE, 2014.

[159] Deguang Wang, Junzhong Shen, Mei Wen, and Chunyuan Zhang. Towards a uniform architecture for the efficient implementation of 2D and 3D deconvolutional neural networks on FPGAs. In 2019 IEEE International Symposium on Circuits and Systems (ISCAS), pages 1–5. IEEE, 2019.

[160] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. Advances in neural information processing systems, 31, 2018.

[161] Peiqi Wang, Dongsheng Wang, Yu Ji, Xinfeng Xie, Haoxuan Song, XuXin Liu, Yongqiang Lyu, and Yuan Xie. Qgan: Quantized generative adversarial networks. arXiv preprint arXiv:1901.08263, 2019.

[162] Tianzhe Wang, Kuan Wang, Han Cai, Ji Lin, Zhijian Liu, Hanrui Wang, Yujun Lin, and Song Han. APQ: Joint search for network architecture, pruning and quantization policy. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 2078–2087, 2020.

[163] Ying Wang, Yadong Lu, and Tijmen Blankevoort. Differentiable joint pruning and quantization for hardware efficiency. In European Conference on Computer Vision, pages 259–277. Springer, 2020.

[164] Zhihao Wang, Jian Chen, and Steven CH Hoi. Deep learning for image super-resolution: A survey. IEEE transactions on pattern analysis and machine intelligence, 2020.

[165] Ziheng Wang, Jeremy Wohlwend, and Tao Lei. Structured pruning of large language models. arXiv preprint arXiv:1910.04732, 2019.

[166] Alexander Watson. Deep learning techniques for super-resolution in video games. arXiv preprint arXiv:2012.09810, 2020.

[167] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM, 52(4):65–76, 2009.

[168] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. Integer quantization for deep learning inference: Principles and empirical evaluation. CoRR, abs/2004.09602, 2020.

[169] Xia Xiao and Zigeng Wang. Autoprune: Automatic network pruning by regularizing auxiliary parameters. Advances in Neural Information Processing Systems 32 (NeurIPS 2019), 32, 2019.

[170] Hongwei Xie, Yafei Song, Ling Cai, and Mingyang Li. Overflow aware quantization: Accelerating neural network inference by low-bit multiply-accumulate operations. In Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence, pages 868–875, 2021.

[171] Dawen Xu, Cheng Liu, Ying Wang, Kaijie Tu, Bingsheng He, and Lei Zhang. Accelerating generative neural networks on unmodified deep learning processors—a software approach. IEEE Transactions on Computers, 69(8):1172–1184, 2020.

[172] Shuyuan Xu, Jun Wang, Wenchi Shou, Tuan Ngo, Abdul-Manan Sadick, and Xiangyu Wang. Computer vision techniques in construction: a critical review. Archives of Computational Methods in Engineering, 28:3383–3397, 2021.

[173] X. Xu, S. Das, and K. Kreutz-Delgado. Approxdbn: Approximate computing for discriminative deep belief networks. arXiv preprint arXiv:1704.03993, 2017.

[174] Chen Yang, Zhenghong Yang, Abdul Mateen Khattak, Liu Yang, Wenxin Zhang, Wanlin Gao, and Minjuan Wang. Structured pruning of convolutional neural networks via l1 regularization. IEEE Access, 7:106385–106394, 2019.

[175] Haichuan Yang, Shupeng Gui, Yuhao Zhu, and Ji Liu. Automatic neural network compression by sparsity-quantization joint learning: A constrained optimization-based approach. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 2178–2188, 2020.

[176] Amir Yazdanbakhsh, Michael Brzozowski, Behnam Khaleghi, Soroush Ghodrati, Kambiz Samadi, Nam Sung Kim, and Hadi Esmaeilzadeh. Flexigan: An end-to-end solution for FPGA acceleration of generative adversarial networks. In 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 65–72. IEEE, 2018.

[177] Amir Yazdanbakhsh, Kambiz Samadi, Nam Sung Kim, and Hadi Esmaeilzadeh. GANAX: A unified mimd-simd acceleration for generative adversarial networks. In Proceedings of the 45th Annual International Symposium on Computer Architecture, pages 650–661. IEEE Press, 2018.

[178] Po-Hsiang Yu, Sih-Sian Wu, Jan P Klopp, Liang-Gee Chen, and Shao-Yi Chien. Joint pruning & quantization for extremely sparse neural networks. arXiv preprint arXiv:2010.01892, 2020.

[179] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8bert: Quantized 8bit bert. In 2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS), pages 36–39. IEEE, 2019.

[180] Matthew D Zeiler, Dilip Krishnan, Graham W Taylor, and Rob Fergus. Deconvolutional networks. In 2010 IEEE Computer Society Conference on computer vision and pattern recognition, pages 2528–2535. IEEE, 2010.

[181] C Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pages 161–170. ACM, 2015.

[182] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu. ApproxANN: an approximate computing framework for artificial neural network. In Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, pages 701–706. EDA Consortium, 2015.

[183] Xinyu Zhang, Ian Colbert, and Srinjoy Das. Learning low-precision structured subnetworks using joint layerwise channel pruning and uniform quantization. Applied Sciences, 12(15):7829, 2022.

[184] Xinyu Zhang, Ian Colbert, Ken Kreutz-Delgado, and Srinjoy Das. Training deep neural networks with joint quantization and pruning of weights and activations. arXiv preprint arXiv:2110.08271, 2021.

[185] Xinyu Zhang, Srinjoy Das, Ojash Neopane, and Ken Kreutz-Delgado. A design methodology for efficient implementation of deconvolutional neural networks on an FPGA. arXiv preprint arXiv:1705.02583, 2017.

[186] Yulun Zhang, Kunpeng Li, Kai Li, Bineng Zhong, and Yun Fu. Residual non-local attention networks for image restoration. arXiv preprint arXiv:1903.10082, 2019.

[187] Chenglong Zhao, Bingbing Ni, Jian Zhang, Qiwei Zhao, Wenjun Zhang, and Qi Tian. Variational convolutional neural network pruning. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 2780–2789, 2019.

[188] Yiren Zhao, Xitong Gao, Daniel Bates, Robert Mullins, and Cheng-Zhong Xu. Focused quantization for sparse CNNs. Advances in Neural Information Processing Systems, 32, 2019.

[189] Zhen Zheng, Pengzhan Zhao, Guoping Long, Feiwen Zhu, Kai Zhu, Wenyi Zhao, Lansong Diao, Jun Yang, and Wei Lin. Fusionstitching: boosting memory intensive computations for deep learning workloads. arXiv preprint arXiv:2009.10924, 2020.

[190] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In Proceedings of the IEEE international conference on computer vision, pages 2223–2232, 2017.

[191] Michael Zhu and Suyog Gupta. To prune, or not to prune: Exploring the efficacy of pruning for model compression. In International Conference on Learning Representations, 2018.