EFFICIENT AND ROBUST DEEP LEARNING THROUGH APPROXIMATE COMPUTING

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Sanchari Sen

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2020

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Anand Raghunathan, Chair
School of Electrical and Computer Engineering

Dr. Kaushik Roy
School of Electrical and Computer Engineering

Dr. Sumeet K. Gupta
School of Electrical and Computer Engineering

Dr. Vijay Raghunathan
School of Electrical and Computer Engineering

Approved by:

Dr. Dimitrios Peroulis
Head of the School Graduate Program

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Anand Raghunathan, for his constant guidance and support throughout the last five years. Our regular discussions have helped me grow immensely not only as a researcher, but also as a person. He has constantly encouraged me to pursue new ideas and acquire new skills. His persistence on improving my writing and presentation skills has allowed me to conceptualize and communicate better. Throughout my graduate life, I have been trying to imbibe his impeccable research acumen, work ethic and thirst for knowledge, and will continue to do so in the next phase of my life.

I would like to thank the members of my advisory committee, Prof. Kaushik Roy, Prof. Vijay Raghunathan and Prof. Sumeet Gupta for their insightful comments and constructive feedback. Their thoughts and suggestions have helped to better shape my research. I am grateful for their time and effort on the same.

Next, I want to thank the past and current members of the Integrated Systems Laboratory (ISL) at Purdue, including Dr. Swagath Venkataramani, Dr. Ashish Ranjan, Dr. Shubham Jain, Younghoon Kim, Jacob Stevens, Sarada Krithivasan, Vinod Ganesan, Abinand Nallathambi, Manik Singhal, Sourjya Roy, Shrihari Sridharan, Reena Elangovan and Amrit Nagarajan. Their willingness to help and share their expertise have allowed me to overcome numerous obstacles encountered in my work. I have thoroughly enjoyed our conversations on a variety of academic and non-academic topics, both inside and outside the confinements of the lab. I am grateful for having the opportunity to closely collaborate with some of the members, which has helped expand my research horizon. I also enjoyed my interactions with Viji Srinivasan, Kailash Gopalakrishnan, Derrick Aguren and Joseph Greathouse as part of my internships at IBM T. J. Watson Research Center and AMD Research.

My stay at Purdue would have been incomplete without the amazing time I spent outside work. I am lucky to have been part of the Purdue University Tagore Society (PUTS) and attend their numerous events. I will greatly cherish the memorable moments spent with my friends here and would specially like to thank Sreya Sarkar, Esha Chatterjee, Indrani Biswas, Somrita Chatterjee, Srishti Chakravorty, Sayan Choudhury, Aritra Mitra, Arindam Nandi and Sayantan Bhattacharya for their invaluable friendships. I am deeply indebted to my boyfriend, Prabudhya Roy Chowdhury, for constantly supporting me and accompanying me throughout the past 8 years of my life.

Finally, I would like to thank my parents, Dr. Siddhartha Sen and Sreeparna Sen, for their unparalleled love, care and support; for always believing in me and encouraging me to pursue my dreams. I would like to thank my brother, Dr. Sambudhha Sen, for being someone I could always look up to, share my concerns and ask for advice. The following pages of the dissertation are dedicated to them.

TABLE OF CONTENTS

Page

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Sen, Sanchari PhD, Purdue University, August 2020. Efficient and Robust Deep Learning Through Approximate Computing. Major Professor: Anand Raghunathan.

Deep Neural Networks (DNNs) have greatly advanced the state-of-the-art in a wide range of machine learning tasks involving image, video, speech and text analytics, and are deployed in numerous widely-used products and services. Improvements in the capabilities of hardware platforms such as Graphics Processing Units (GPUs) and specialized accelerators have been instrumental in enabling these advances as they have allowed more complex and accurate networks to be trained and deployed. However, the enormous computational and memory demands of DNNs continue to increase with growing data size and network complexity, posing a continuing challenge to computing system designers. For instance, state-of-the-art image recognition DNNs require hundreds of millions of parameters and hundreds of billions of multiply-accumulate operations while state-of-the-art language models require hundreds of billions of parameters and several trillion operations to process a single input instance. Another major obstacle in the adoption of DNNs, despite their impressive accuracies on a range of datasets, has been their lack of robustness. Specifically, recent efforts have demonstrated that small, carefully-introduced input perturbations can force a DNN to behave in unexpected and erroneous ways, which can have to severe consequences in several safety-critical DNN applications like healthcare and autonomous vehicles. In this dissertation, we explore approximate computing as an avenue to improve the speed and energy efficiency of DNNs, as well as their robustness to input perturbations.

Approximate computing involves executing selected computations of an application in an approximate manner, while generating favorable trade-offs between computational efficiency and output quality. The intrinsic error resilience of machine learning applications makes them excellent candidates for approximate computing, allowing us to achieve execution time and energy reductions with minimal effect on the quality of outputs. This dissertation performs a comprehensive analysis of different approximate computing techniques for improving the execution efficiency of DNNs. Complementary to generic approximation techniques like quantization, it identifies approximation opportunities based on the specific characteristics of three popular classes of networks - Feed-forward Neural Networks (FFNNs), Recurrent Neural Networks (RNNs) and Spiking Neural Networks (SNNs), which vary considerably in their network structure and computational patterns.

First, in the context of feed-forward neural networks, we identify sparsity, or the presence of zero values in the data structures (activations, weights, gradients and errors), to be a major source of redundancy and therefore, an easy target for approximations. We develop lightweight micro-architectural and instruction set extensions to a general-purpose processor core that enable it to dynamically detect zero values when they are loaded and skip future instructions that are rendered redundant by them. Next, we explore LSTMs (the most widely used class of RNNs), which map sequences from an input space to an output space. We propose hardware-agnostic approximations that dynamically skip redundant symbols in the input sequence and discard redundant elements in the state vector to achieve execution time benefits. Following that, we consider SNNs, which are an emerging class of neural networks that represent and process information in the form of sequences of binary spikes. Observing that spike-triggered updates along synaptic connections are the dominant operation in SNNs, we propose hardware and software techniques to identify connections that can be minimally impact the output quality and deactivate them dynamically, skipping any associated updates.

The dissertation also delves into the efficacy of combining multiple approximate computing techniques to improve the execution efficiency of DNNs. In particular, we focus on the combination of quantization, which reduces the precision of DNN data-structures, and pruning, which introduces sparsity in them. We observe that the ability of pruning to reduce the memory demands of quantized DNNs decreases with precision as the overhead of storing non-zero locations alongside the values starts to dominate in different sparse encoding schemes. We analyze this overhead and the overall compression of three different sparse formats across a range of sparsity and precision values and propose a hybrid compression scheme that identifies that optimal sparse format for a pruned low-precision DNN.

Along with improved execution efficiency of DNNs, the dissertation explores an additional advantage of approximate computing in the form of improved robustness. We propose ensembles of quantized DNN models with different numerical precisions as a new approach to increase robustness against adversarial attacks. It is based on the observation that quantized neural networks often demonstrate much higher robustness to adversarial attacks than full precision networks, but at the cost of a substantial loss in accuracy on the original (unperturbed) inputs. We overcome this limitation to achieve the best of both worlds, i.e., the higher unperturbed accuracies of the full precision models combined with the higher robustness of the low precision models, by composing them in an ensemble.

In summary, this dissertation establishes approximate computing as a promising direction to improve the performance, energy efficiency and robustness of neural networks.

1. INTRODUCTION

Deep Neural Networks (DNNs) have transformed the field of machine learning by achieving state-of-the-art results on a wide range of tasks like image classification, speech recognition, object detection and machine translation [1–7]. Today, DNNs are deployed in a spectrum of real-world products and services including Google Translate [8], Google Maps [9], Apple's Siri [10], Google's voice and image search [11], Netflix and Amazon's recommendation engines [12]. Although neural networks have a rich history dating back to the 1950s, their phenomenal growth and expansion into nearly all fields of machine learning is relatively recent and can be attributed to a favorable confluence of various factors. A major factor on that front has been the improvement in the capabilities of hardware platforms like Graphics Processing Units (GPUs) and specialized accelerators. For instance, there was a 65× increase in the performance of different GPUs proposed between 2013 and 2016 [13]. The evolution of these hardware platforms, along with the availability of larger datasets, has been instrumental in allowing the development of more accurate (and more complex) networks over the years. For example, AmoebaNet, a state-of-the-art image-recognition DNN demonstrates an impressive 83.9% accuracy on the ImageNet dataset but requires 469 million parameters and 104 billion multiply-accumulate operations to classify a single image [14]. Thus, in order to continue the development of increasingly complex DNNs with better accuracies, computing system designers need to pursue innovative ways of meeting their high computational demands.

Prior research efforts have explored a few key directions to address the computational challenges posed by DNNs. The first direction explores efficient parallelization techniques of DNNs on programmable platforms such as multi-cores and GPUs [15–21]. The second direction develops specialized accelerators to match the compute and data access patterns of different kinds of DNNs [22–30]. Complemen-

tary to these directions, we explore approximate computing as a promising approach to improve the execution efficiency of DNNs.

Along with their high computational and memory demands, recent efforts have highlighted another key limitation of DNNs, namely, their lack of robustness [31]. Specifically, DNN outputs have been observed to be severely affected by small, carefully-introduced input perturbations, which can have drastic consequences in different safety-critical applications like autonomous driving and healthcare. These adversarial input perturbations can be systematically generated through a range of adversarial attack techniques, including Fast Gradient Sign Method [31], Projected Gradient Descent [32] and Basic Iterative Method [33]. Therefore, it is imperative to improve the robustness of DNNs for ushering in the next realm of deep learning applications, beyond the primitive tasks of image, speech and text analytics.

Realizing this need, previous efforts have looked into various means for boosting robustness, including training with adversarial or noisy inputs [31, 32, 34], defensive distillation [35] and regularizing input gradients [36]. We, on the other hand, focus on improving robustness through approximate computing.

Approximate computing is a computing paradigm that allows certain computations in an application to be performed in an inexact or approximate manner while preserving the output quality. It leverages the intrinsic error resilience of applications to introduce approximations that can achieve maximum benefits for a given quality constraint. In general, neural networks have been observed to be highly resilient to approximations in a significant fraction of their computations. For example, different quantized neural networks [37–39] utilizing low-precision data-structures have succeeded in achieving high accuracy levels, in spite of the errors introduced during the quantization process. This intrinsic error resilience makes DNNs attractive candidates for approximate computing.

A key challenge in approximate computing is to identify which computations to approximate and by how much. In order to achieve a favorable energy or execution time versus quality tradeoff, we primarily target our approximations on computations

that do not affect the network numerically, as well as, computations that do affect the intermediate values of the network such that the effects don't propagate to the final output quality significantly. In this dissertation, we perform a comprehensive analysis of these computations in DNNs and propose efficient software and hardware techniques for introducing approximations in them. We specifically illustrate the benefits of approximations in three different types of neural networks, namely, Feed-Forward Neural Networks (FFNNs), Recurrent Neural Networks (RNNs) and Spiking Neural Networks (SNNs), which have widely varying computational characteristics and network structures. We further explore the challenges in combining multiple approximate computing techniques for improving the efficiency of these DNNs, specially in the context of FFNNs and RNNs.

While a majority of previous efforts in approximate computing focus on improvements in execution time and energy consumption, a few recent efforts have also highlighted a previously unexplored advantage of approximate computing, in the form of improving robustness of DNNs [40–42]. We explore this advantage even further to develop DNNs robust to a range of adversarial attacks, while maintaining their output quality on the original unperturbed inputs.

In the following sections, we first discuss the computational challenges of DNNs. Next, we discuss the robustness of DNNs. Following that, we present the contributions of this dissertation in terms of developing approximate computing techniques for improving the efficiency and robustness of DNNs. Finally, we outline the remaining chapters of this dissertation.

1.1 Computational Challenges of DNNs

The rise of DNNs in the last few years has allowed us to achieve major breakthroughs in several machine learning tasks. Their success can most readily be exemplified by their achievements in the ImageNet challenge. Figure 1.1(a) illustrates the classification accuracies of the top entrants in the challenge over different years. The accuracies

are quantified in terms of the top-5 errors which represent the fraction of images for which the correct class did not appear in the top five categories determined by the algorithm. DNNs entered the challenge for the first time in 2012 and immediately caused the error rate to decrease by almost 10%. For the next few years, the development of other forms of DNNs allowed the error rate to quickly decrease to below 5% in 2017, which even exceeds human accuracy levels.

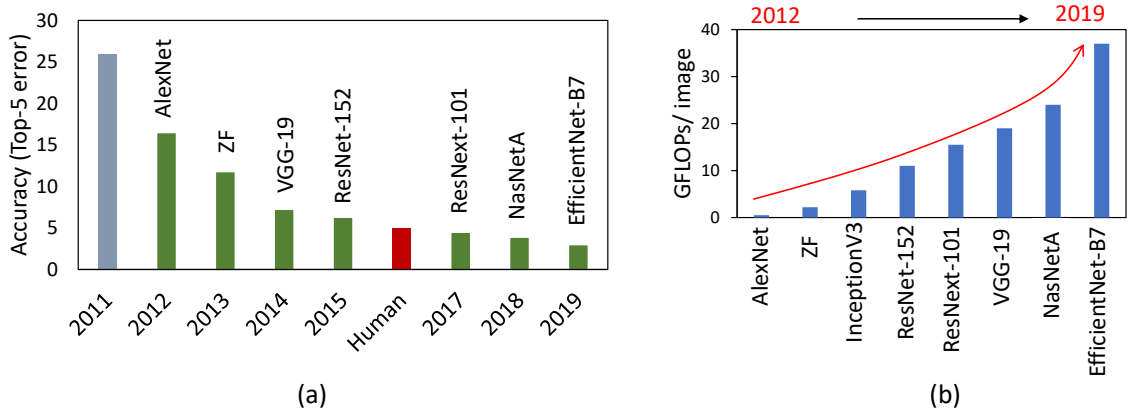

Fig. 1.1.: (a) Results from the ImageNet challenge (b) Number of FLOPs required to evaluate different DNNs from the ImageNet challenge

The dramatic accuracy improvements on the ImageNet dataset were mainly possible through the design of deeper and increasingly complex networks. Obviously, this came at the expense of increased computational and storage requirements of DNNs. Figure 1.1(b) quantifies the computational demands of different top DNN entries in the ImageNet challenge over the years. It shows that there was a 10× increase in the number of scalar floating point operations (FLOPs) required to evaluate different networks proposed between 2012 to 2019. A significant number of these DNNs relied on GPUs to meet their high computational demands in both training and inference. This was further facilitated by a 65× increase in GPU performance from 2013 to 2016 [13].

As DNNs get deployed in increasingly complex tasks on bigger datasets, their computational demands are expected to further increase in the future. In addition, this

growth can potentially surpass the growth in performance of future GPUs, thereby ceasing any possibility of addressing the challenge by simply executing them on increasingly powerful GPUs. Realizing this, existing research efforts have mainly tried to improve the computational efficiency of DNNs by designing accelerators that can mimic the compute and data access patterns of DNNs [22–30].

Complementary to the above approaches, this dissertation focuses on the use of approximate computing for improving the computational efficiency of DNNs. Previous efforts in that context have explored the use of network pruning [43–45], quantized data-structures [39, 46, 47] and approximate arithmetic units [37, 48, 49]. However, these approaches are often restricted to feed-forward neural networks and can primarily provide benefits on only specialized hardware architectures. In contrast, we consider different classes of DNNs and identify approximations that exploit their unique computational characteristics. Further, we don't restrict these approximations to accelerator platforms and demonstrate savings on different general-purpose platforms as well.

1.2 Robustness of DNNs

The remarkable success of DNNs on different machine learning tasks, as illustrated in the previous section for the ImageNet challenge, has naturally led to a significant interest in deploying them in various real-world applications. A large number of these real-world applications, including autonomous driving, healthcare, financial risk management, *etc.*, are safety-critical in nature. Accordingly, robustness, *i.e.*, the ability to cope with erroneous or malicious inputs fed to an application, is emerging as an important requirement for DNNs, along with accuracy.

Recent studies on robustness of DNNs have in fact revealed that DNNs are susceptible to errors under the presence of small input perturbations, imperceptible to humans [31]. A range of adversarial attack techniques, proposed in previous works, provide systematic methodologies for generating perturbations that can cause a DNN

to fail [31, 33, 50, 51]. These perturbations can even be targeted towards a particular output class [51] and transferred across different models [52], increasing their strength even further. Fig 1.2 presents some original and adversarial images from the MNIST dataset.

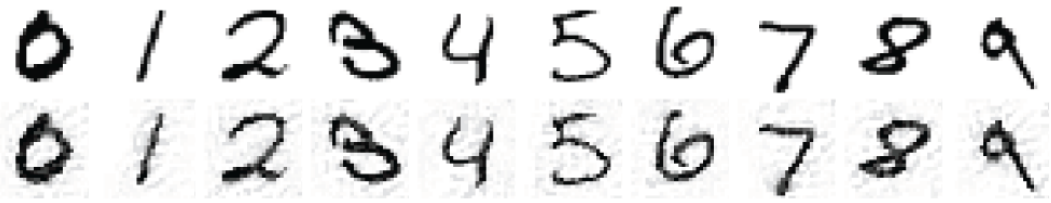

Fig. 1.2.: Original and adversarial images from the MNIST dataset [53]

To overcome this limitation, prior efforts have proposed a range of techniques for boosting robustness. These efforts can be grouped into two broad classes. The first class of efforts involve modifying the training process of DNNs to perform adversarial training [31, 32], noise-based training [34] or defensive distillation [35]. The second class of efforts focuses on different input pre-processing techniques to reduce the effect of perturbations [54, 55].

Complementary to these efforts, we focus on the use of approximate computing for boosting robustness. Earlier efforts in this respect have specifically studied the effect of quantization and observed that it can improve robustness [40–42], but at the cost of some loss in the original unperturbed accuracy. We, on the other hand, overcome this limitation by forming ensembles of both full-precision and low-precision DNNs, to increase the robustness of DNNs while maintaining their accuracy on the original unperturbed inputs.

1.3 Thesis contributions

In this dissertation, we explore approximate computing as an avenue to improve the speed, energy efficiency and robustness of both software and hardware implementations of DNNs.

The unique computational and data characteristics of different classes of DNNs give rise to distinct forms of computations that can be approximated. We first explore these approximations for improving the speed and energy efficiency of three different classes of DNNs, namely, Feed-forward Neural Networks (FFNNs), Recurrent Neural Networks (RNNs) and Spiking Neural Networks (SNNs). Next, we explore the opportunities for combining two different forms of approximations, namely, quantization and pruning, in DNNs. Finally, we explore approximations for improving the robustness of DNNs.

1.3.1 Improving Efficiency of Feed-Forward Neural Networks

Feed-Forward neural networks (FFNNs) are a class of DNNs characterized by the flow of information strictly in the forward direction from the input layer to the output layer. These include Convolutional Neural Networks (CNNs) and Multi-Layer Perceptrons (MLPs) deployed in tasks like image recognition and object detection. They lack any memory of previous inputs and are thus more suitable for tasks with fixed-length, temporally independent inputs and outputs. We identify sparsity, or the presence of zero values in different data-structures, as a significant source of redundancy in these CNNs. Specifically, multiply-accumulate (MAC) operations, which are the primitive unit of computations in these networks, become redundant when any of the input operands are zeros. Across 6 image-recognition feed-forward networks, we observe that sparsity results in $\sim 45.1\%$ of the computations being rendered redundant, presenting a significant opportunity for improving performance. Therefore, we target our approximations on these redundant MAC computations.

Sparsity in the FFNN data-structures like activations, weights and backpropagated errors arises from the presence of ReLU (Rectified Linear Unit) layers as well as from the application of pruning techniques for model size reductions. This sparsity can be both static or dynamic, depending on whether the zero values remain constant or vary across different inputs to the network. Sparsity in weights, intro-

duced by pruning connections in the network after training, is static in nature. In contrast, activation and error sparsities, caused by the thresholding nature of the ReLU activation functions, are dynamic in nature. In this dissertation, we propose lightweights extensions to general-purpose cores in the form of Sparsity-aware Core Extensions (SparCE) for exploiting both static and dynamic sparsities by dynamically detecting whether an operand (e.g., the result of a load instruction) is zero and subsequently skipping a set of future instructions that use it. SparCE consists of 2 key micro-architectural enhancements. First, a Sparsity Register File (SpRF) is utilized to track registers that are zero. Next, a Sparsity-Aware Skip Address (SASA) Table is used to indicate instruction sequences that can be skipped, and to specify conditions on SpRF registers that trigger instruction skipping. When an instruction is fetched, SparCE dynamically pre-identifies whether the following instruction(s) can be skipped, and if so appropriately modifies the program counter, thereby skipping the redundant instructions and improving performance. We model SparCE using the gem5 architectural simulator, and evaluate our approach on 6 state-of-the-art image-recognition CNNs (the most commonly used type of FFNNs) in the context of both training and inference using the Caffe deep learning framework. On a scalar microprocessor, SparCE achieves 1.11×-1.96× speedups across both convolutional and fully-connected layers that exhibit 10%-90% sparsity. These speedups translate to 19%-31% reduction in execution time at the overall application-level. We also evaluate SparCE on a 4-way SIMD ARMv8 processor using the OpenBLAS library, and demonstrate that SparCE achieves 8%-15% reduction in the application-level execution time.

In addition to the feed-forward networks described above, there exists a different class of neural networks referred to as Recurrent Neural Networks (RNNs), which can handle variable-length inputs and outputs forming sequences. The following subsection will briefly discuss our proposed approximations in RNNs.

1.3.2 Improving Efficiency of Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a class of DNNs with applications in text and handwriting synthesis [56], speech recognition [57], neural machine translation [58] and image and video captioning [3]. These networks have a cyclic structure, allowing information to persist temporally in the form of memory (state) in the network. The computation of an RNN can be thought of as proceeding in timesteps with a new element of the input sequence being fed to the network at each timestep.

In this dissertation, we propose hardware-agnostic approximate computing techniques for accelerating RNNs. We specifically consider Long Short Term Memory networks (LSTMs), the most popular class of RNNs. The proposed AxLSTM consists of two techniques Dynamic Timestep Skipping (DTS) and Dynamic State Reduction (DSR). Dynamic Timestep Skipping identifies, at runtime, input symbols that are likely to have little or no impact on the cell state and skips evaluating the corresponding timesteps. In contrast, Dynamic State Reduction reduces the size of the cell state in accordance with the complexity of the input sequence, leading to a reduced number of computations per timestep. We describe how AxLSTM can be applied to the most common application of LSTMs, viz., sequence-to-sequence learning. We implement AxLSTM within the TensorFlow deep learning framework and evaluate it on 3 state-of-the-art sequence-to-sequence models. On a 2.7 GHz Intel Xeon server with 128 GB memory and 32 processor cores, AxLSTM achieves 1.08×-1.31× speedups with minimal loss in quality, and 1.12×- 1.37× speedups when moderate reductions in quality are acceptable.

1.3.3 Improving Efficiency of Spiking Neural Networks

Apart from the conventional CNNs and RNNs discussed above, which consist of neurons communicating through continuous activation values, there also exists an emerging class of neural networks often referred to as the third generation of neural networks, namely, Spiking Neural Networks (SNNs). SNNs mimic the spiking be-

havior of biological neurons and accordingly, represent and process information in the form of binary valued spikes. They well-suited to applications that operate on temporal streams of data (*e.g.* outputs of event-driven cameras).

This dissertation proposes AxSNN, a set of approximate computing techniques exploiting the spiking behavior of SNNs to improve their computational efficiency. SNNs are composed of neurons associated with membrane potentials and these neurons generate spikes whenever the potential exceeds a threshold. The spikes generated by neurons can be viewed as events that trigger updates to the membrane potentials of all outgoing neurons. These spike-triggered updates thus form the fundamental compute primitives in SNNs. AxSNN determines spike-triggered neuron updates that can be skipped with little or no impact on output quality and selectively skips them to improve both compute and memory energy. Neurons that can be approximated are identified by utilizing various static and dynamic parameters such as the average spiking rates and current potentials of neurons, and the weights of synaptic connections. Such a neuron is placed into one of many approximation modes, wherein the neuron is sensitive only to a subset of its inputs and sends spikes only to a subset of its outputs. We apply AxSNN to both hardware and software implementations of SNNs. For hardware evaluation, we designed SNNAP, a Spiking Neural Network Approximate Processor that embodies the proposed approximation strategy, and synthesized it to 45nm technology. The software implementation of AxSNN was evaluated on a 2.7 GHz Intel Xeon server with 128 GB memory. Across a suite of 6 image recognition benchmarks, AxSNN achieves 1.4-5.5× reduction in scalar operations for network evaluation, which translates to 1.2-3.62× and 1.26-3.9× improvements in hardware and software energies respectively, for no loss in application quality.

1.3.4 Exploring Opportunities for Combining Multiple Approximate Computing Approaches

Along with the above approaches, a wide spectrum of other approximate computing techniques have also been proposed over the years for improving the efficiency of DNNs. Pruning and quantization have emerged as two of the most popular approaches among them, specially for reducing the memory requirements of DNNs. Pruning zeros out different weight values by removing redundant connections in the network and allows weights to be stored compactly in memory using different sparse formats. Quantization, on the other hand, converts DNN weights to low-precision values, which can be represented and stored in memory using a smaller number of bits. Both pruning and quantization have primarily been explored as independent approaches with ongoing efforts focusing on advancing their individual limits even further. We investigate the opportunities for combining them, specially as we go into the regime of ultra-low precision DNNs (sub-8 bits of precision).

In this dissertation, we systematically evaluate the effectiveness of pruning ultra-low precision DNNs, in terms of the memory reductions achieved by storing these pruned DNNs in different sparse formats. We consider two sparse formats widely used in compressing DNNs, *viz.*, Compressed Sparse Column (CSC) and Sparsity Map (Smap). We also propose a new format, namely, compressed Sparsity Map (cSmap), that improves upon the Smap format by replacing the sparsity map matrix with its compressed version. We realize the cSmap format in our implementation by re-purposing the test pattern compression tools widely used in manufacturing tests. We discover that the memory compression benefits of all three sparse formats reduce with decreasing precision because of the increasing overhead of storing the location of non-zero weights along with their values. Across several pruned and quantized versions of 6 state-of-the-art DNNs, we observe that the compression ratio achieved by each format is a strong function of both sparsity and precision, even leading to values <1 in certain cases. Based on this observation, we propose a new hybrid

compression scheme that compresses different networks, and individual layers within them, in different sparse formats — identified to be best-suited for their precision and sparsity levels. We demonstrate that such a hybrid scheme can improve the average compression ratio of a 2-bit DNN by 18.3% - 34.7% over homogeneous compression schemes.

1.3.5 Improving Robustness of DNNs

In addition to reducing the computational and memory demands of DNNs, a few recent efforts have highlighted that quantization can also be used as an approach to improve robustness, as low-precision DNNs exhibit higher adversarial accuracies than full-precision DNNs [40–42]. However, the loss in information associated with the quantization process often makes these low-precision models perform significantly worse than their full-precision counterparts while classifying the original unperturbed inputs.

This dissertation proposes EMPIR, ensembles of mixed-precision DNNs, as a new approach for improving robustness. It combines the higher robustness of low-precision models with the higher unperturbed accuracy of full-precision models by composing them in an ensemble. In the general case, it is composed of M full-precision models and N low-precision models with the outputs of the individual models combined through ensembling techniques that count the number of predictions for each class or average the probabilities of the models. We study the effect of ensemble size and ensembling technique on the overall robustness of the model and observe that $M = 1$ and $N = 2$ or 3 provides significant improvement in robustness with minimal compute and memory overheads. We implemented EMPIR within the TensorFlow framework and measured the adversarial accuracies under a range of adversarial attacks within the Cleverhans library. Our results indicate that EMPIR boosts the average adversarial accuracies by 42.6%, 15.2% and 10.5% for the DNN models trained on the MNIST, CIFAR-10 and ImageNet datasets respectively, when compared to single full-precision

models, without sacrificing accuracy on the unperturbed inputs. Further, these EMPIR models only incur modest compute and memory overheads compared to a single full-precision model (<25% in our evaluations).

1.4 Thesis outline

The rest of the thesis is organized as follows. Chapter 2 details the related research efforts in accelerating different types of DNNs and improving their robustness. Chapter 3 provides the necessary background on FFNNs, LSTMs and SNNs, along with the preliminaries of pruning, quantization and adversarial attacks on DNNs. Chapter 4 proposes techniques to accelerate FFNNs on general-purpose platforms by exploiting sparsity. Chapter 5 proposes hardware-agnostic approximate computing techniques for accelerating LSTMs. Chapter 6 discusses another application of approximate computing for accelerating SNNs. Chapter 7 investigates the opportunities for combining two popular approximate computing techniques of pruning and quantization in DNNs. Chapter 8 proposes an approximate computing technique for boosting robustness of DNNs. Finally, Chapter 9 concludes this dissertation.

2. RELATED WORK

Deep neural networks have successfully achieved state-of-the-art results in a wide range of machine learning tasks like image, video, text and speech processing. However, the pursuit of larger networks with better accuracies has also led to increasing concerns about their high computational and memory demands, which far outstrips the capabilities of modern computing platforms. Multiple research efforts have accordingly tried to address the computational challenge posed by DNNs by proposing techniques to improve their computational efficiency on different platforms. In this chapter, we discuss some of these efforts, particularly in the context of convolutional neural networks, long short term memory neural networks and spiking neural networks, and highlight the unique aspects of our work.

In addition to their high computational and memory demands, recent efforts have also highlighted another notable limitation of DNNs, namely, their lack of robustness. Various forms of adversarial attacks have successfully fooled DNNs through small, carefully-introduced input perturbations that cause large misclassifications. To overcome this limitation, prior research efforts have proposed different techniques for increasing robustness of DNNs. We present the details of some of these efforts in this chapter and contrast them with our approach.

Finally, as this dissertation is built on the principles of approximate computing, we also present previous research efforts in applying approximate computing techniques to improve the computational efficiency of different applications.

2.1 Improving Efficiency of DNNs

Prior research efforts that target improving the computational efficiency of DNNs can be grouped into the four broad classes which are discussed below in more detail.

2.1.1 Software parallelization on multi-cores and GPUs

A large number of previous efforts have been directed towards developing techniques for efficient parallelization of DNNs on programmable platforms such as multi-core servers and GPUs [15–20, 59–63]. These techniques primarily exploit two different forms of parallelism, namely, model parallelism and data parallelism [15]. Model parallelism refers to the parallelization of different parts of the same model across different threads or cores. On the other hand, data parallelism refers to the parallelization of different inputs to the same model across different threads or cores. The efficiency of model or data parallelism largely depends on the characteristics of a particular DNN. Model parallelism is generally observed to be more efficient for scenarios with large amounts of compute operations per neuron while data parallelism is observed to be more efficient for DNNs with larger batch sizes of inputs. Some of the approaches also employ other optimizations on top of model and data parallelism, exploiting the computational pattens of specific classes of DNNs like LSTMs [21] or SNNs [63]. Overall, the scalability of the techniques is often limited by synchronization and communication bottlenecks. Specially the dynamic and event-driven nature of SNNs makes parallelization challenging, as it leads to irregular and unpredictable compute and memory access patterns.

2.1.2 Specialized accelerators

Developing specialized hardware architectures has been an attractive approach to improve the computational efficiency of DNNs. These accelerators utilize specialized processing cores, interconnect networks and other hardware-software co-design methodologies to leverage the different forms of compute and data reuse patterns in DNNs. They differ on the kind of parallelism exploited, the DNN operating phase targeted for acceleration — training or inference phase, as well as, the area, power and energy constraints considered in the designs. Some of the designed accelerators

can support a range of neural networks [23, 24, 26] while others are more targeted towards a particular type of network [22, 29, 64–66].

2.1.3 Model compression

DNNs require high storage and memory bandwidths to efficiently store and transfer the large number of parameters involved in their computations. Existing efforts in pruning [43–45] and quantization [37,39,47,67] address this challenge by reducing the number of parameters and the number of bits used in representing them, respectively. This is specially advantageous in various resource-constrained embedded platforms which cannot support DNNs of large model sizes.

2.1.4 Exploiting sparsity in DNNs

Prior efforts that exploit sparsity to improve DNN efficiency can be grouped into two classes based on whether they exploit static sparsity or dynamic sparsity. Specialized sparse architectures that are capable of exploiting static or dynamic sparsity incorporate a variety of compression techniques and zero-skipping schemes to reduce storage requirements and avoid redundant multiplications in accelerators [26, 68–72].

On the other hand, software approaches that exploit static weight sparsity on GPPs take advantage of sparse matrix libraries. These sparse libraries usually yield performance improvements only under extreme levels of sparsity (>95%). Since DNNs naturally exhibit sparsity in the range of 40%-70%, a few research efforts force more weight sparsity into DNNs using sparse decomposition methods, *etc.* [73, 74] or customize the pruning to match the underlying hardware organization [75]. These invariably come at the cost of training overhead or loss of functional accuracy.

2.2 Improving Robustness of DNNs

Popular approaches for improving robustness of DNNs can be grouped into the following three classes.

2.2.1 Modified DNN training

The first class of efforts includes popular techniques like adversarial training and noise-based training which augment the training dataset with adversarial inputs [31,32] or noisy inputs [34] to increase robustness. Although they do improve the robustness of DNNs, the increase in training time can be prohibitively high. Another approach, defensive distillation [35], involves training networks on the output probabilities of classes instead of the conventional approach of training on hard output class labels. It is based on the technique of distillation that was originally proposed to efficiently transfer knowledge across different DNN models. Other approaches have also explored input gradient regularization [36] during the training process for enforcing smooth input gradients in the network.

2.2.2 Input pre-processing

The second class of efforts for increasing robustness performs input pre-processing for reducing the effect of perturbations. Popular pre-processing techniques include input discretization and quantization [54, 55]. However, modifying the attack generation process to include the discretization process is can break the defense mechanism [55].

2.2.3 Use of specialized DNN models

The third class of efforts focus on the use of specialized forms of DNN models for increasing robustness. Some efforts have advocated the use of ensembles of full precision models [53,76–78]. In addition to the above efforts, there have been a parallel set of efforts studying the robustness of low-precision or quantized DNNs. For example,

binary neural networks with single bit precisions for weights and activations have been shown to exhibit higher adversarial robustness than their full-precision counterparts on different white-box attacks [40, 41]. Stochastic quantization of activations has also been proposed as an approach to make DNNs more robust [42]. However, the quantized models in these efforts are often in the sub-8 bit domain and thus demonstrate lower accuracies on unperturbed or clean examples due to the loss in information associated with the quantization process.

2.3 Approximate computing

Approximate computing refers to an emerging computing paradigm that improves the performance and energy efficiency of computing platforms by exploiting the intrinsic error resilience of different applications. Intrinsic error resilience is defined as the ability of certain applications to produce outputs of satisfactory quality even when some of the underlying computations are approximated. Prior research efforts have proposed a multitude of approximate computing techniques over the years, applicable to different levels of the hardware stack, from circuits to architecture and software [79]. These include voltage overscaling techniques at the circuit level [80, 81], precision scaling techniques at the architectural level [82, 83] as well as iteration skipping and dependency relaxation techniques at the software level [84, 85].

Various machine learning applications in domains like recognition, vision, search and data analytics have been observed to exhibit significant amounts of intrinsic error resiliency and thereby be amenable to approximate computing. Accordingly, in the context of NNs, approximate computing has been previously applied to feed-forward neural networks [37,48,49]. In addition, one of the prior approaches have also explored approximate computing for RNNs [86]. These approaches utilize backpropagation, one of the key steps involved in training NNs, to characterize the criticality of neurons in the network, and correspondingly subject them to varying levels of approximation.

These approximations are subsequently realized by utilizing reduced precision and approximate arithmetic units.

2.4 Thesis contributions

The primary contributions of this dissertation are different or complementary to the prior efforts in the following aspects:

Sparsity aware general-purpose core extensions to accelerate FFNNs. Our work differs from specialized accelerator based approaches to accelerate FFNNs [22–26] by focusing on different size and cost-constrained systems like wearables and sensors where the use of accelerators employing large numbers of processing elements and considerable on-chip memory is prohibitive. We specifically accelerate FFNNs on these systems by developing lightweight extensions to the general-purpose processors (GPPs) already present in them. The proposed extensions exploit a key attribute of FFNNs, i.e., sparsity of weights and activation values. Unlike software based approaches that can only exploit the static sparsity in weights [73–75], our proposed extensions, in the form of SparCE (Sparsity-aware Core Extensions), can exploit both dynamic sparsity in activations and static sparsity in weights, while being effective even under intermediate levels. Further, these extensions account for just 1.04% area overhead over an ARM Cortex A35 core of $0.4mm^2$ area, as opposed to area overheads of $4 - 16mm^2$ of different accelerators [22, 25, 26].

Hardware-agnostic approximate computing for LSTMs and SNNs. Our work complements other efforts that accelerate SNNs or LSTMs through the use of parallel software [19–21, 59–63] or specialized hardwares [27–30, 64–66, 87, 88] by proposing the use of approximate computing to improve their computational efficiency. Prior efforts that apply approximate computing to neural networks [37, 48, 49, 86] utilize reduced precision and approximate arithmetic units. As a result, they often require specialized hardware implementations and do not benefit software implementations. In contrast, our work proposes new approximate computing techniques

that are generic in nature and are applicable to both software and hardware implementations. These techniques also take advantage of the unique structure and computational characteristics of SNNs and LSTMs. For instance, our proposed AxSNN (Approximate computing for SNN) identifies spike-triggered neuron updates to be the primitive unit of computations in SNNs and determines updates with little or no impact on output quality to selectively skip them at runtime. On the other hand, the proposed AxLSTM (Approximate computing for LSTM) takes advantage of LSTM's computational pattern of updating its entire memory state vector after processing a new symbol in each timestep. It accordingly skips the evaluation of input symbols that are likely to have little or no impact on memory state and reduces the size of the cell state in accordance with the complexity of the input sequence.

Investigating opportunities for combining pruning and quantization techniques. Prior efforts have primarily explored pruning [43–45, 89] and quantization [39, 46, 47, 67] as independent approaches for compressing DNN models. In contrast, we focus on the intersection of the two techniques and identify the opportunities for combining both. We demonstrate that existing sparse storage formats, which store non-zero locations along with non-zero values in pruned DNNs, suffer from inefficiencies in the ultra-low precision (<8 bits) regime. Specifically, the overhead of storing non-zero locations starts to dominate in this regime and can even force the overall compression ratio to drop below 1 in the worst case, when the sparse format ends up consuming higher memory than the dense format. Our work is also complementary to a few previous efforts that propose training frameworks to perform pruning and quantization in parallel [90]. These efforts solely on simultaneously maximizing the achieved sparsity levels and minimizing the weight precisions without considering the implications of exploiting sparsity in the ultra-low precision regime.

Ensembles of mixed-precision DNNs for increased robustness. Our work differs from existing ensembling approaches for increasing robustness [53, 76–78] by considering both low-precision and full-precision DNNs in the constituent models. The presence of all full-precision models in the previously proposed ensembles in-

creases their compute and memory requirements significantly (10× for an ensemble with 10 models in [53]), which prevents the application of this approach to larger state-of-the art models. In contrast, as low precision DNN models have significantly lower computational and storage requirements than full precision models, the proposed EMPIR ensemble incurs only modest compute and memory overheads compared to a single full-precision model (<25% in our evaluations). Further, unlike the low unperturbed accuracies of the low-precision models proposed for increasing robustness [40–42], EMPIR maintains high unperturbed accuracy by combining higher robustness of the low-precision models with the higher unperturbed accuracies of the full-precision models in the ensemble.

3. BACKGROUND

In this chapter, we first provide a brief background on the network architectures and computational patterns of FFNNs, RNNs and SNNs. Next, we present the basics of pruning and quantizing DNNs. Following that, we present a brief background on adversarial attacks on DNNs. Finally, we discuss the basics of test pattern compression mechanisms.

3.1 Feed-Forward Neural Networks

In this section, we first provide a brief background on Feed-Forward Neural Networks (FFNNs). We then explain the various static and dynamic sources of sparsity in the different data-structures of FFNNs and quantify the opportunity for performance improvement afforded by sparsity.

FFNNs are networks of primitive compute units called *neurons*, organized into layers like convolutional layer, pooling layer and fully-connected layer. Each layer is associated with a set of parameters called weights. Just like any other neural network, FFNNs operate in two phases *viz.* training and inference. During the training phase, a labeled training dataset is used to iteratively refine the weights of the FFNN. In the inference phase, the trained FFNN is used to classify new inputs.

Computationally, FFNN executions iteratively perform three key steps *viz.* Forward Propagation (FP), Backpropagation (BP), and Weight Gradient and Update (WG). These steps operate on four primary data-structures, *viz.*, activations, weights, errors and gradients. All three steps (FP, BP, WG) are performed during the training phase, while inference involves only the FP step. In FP, inputs to the FFNN are propagated through its layers to produce the FFNN outputs. In each layer, the input activations are operated on with its weights to produce its output activations, which

are then fed to the next layer and so on. In BP, errors observed at the output of the FFNN are propagated backwards through each layer of the FFNN. In this case, the error at the output of a layer is operated on with its weights to compute the error at its inputs. In WG, the input activations and the output error of each layer are used to refine its weights.

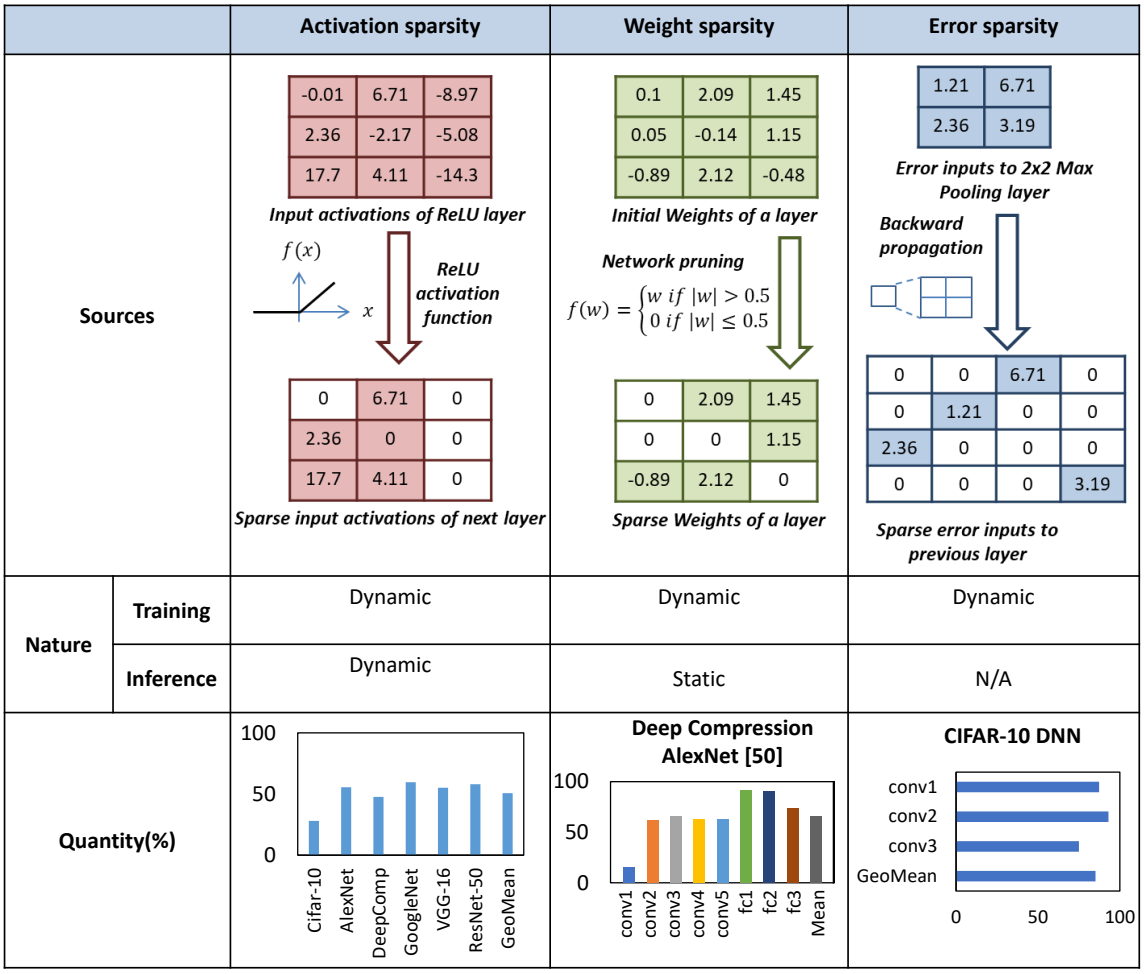

Fig. 3.1.: Different forms of sparsity in FFNNs

3.1.1 Sources of Sparsity in FFNNs

In practice, all major FFNN data-structures - activations, weights, errors and gradients - exhibit significant levels of sparsity, which can be exploited for computational savings. Three of the four data-structures (all except weight gradients) are used as inputs to multiply-and-accumulate operations in the different steps (FP/BP/WG), which become redundant when one of the input operands is zero. Among these three sparse multiply-and-accumulate operands, weights exhibit static sparsity that remains constant across different inputs while the remaining two data-structures (activations and errors) exhibit dynamic sparsity that varies dynamically across different inputs. Figure 3.1 summarizes the sparsity in the different FFNN data-structures, which we describe in the remainder of this subsection.

Static Sparsity

Weight Sparsity. Sparsity in weights occurs during the inference phase of the FFNN. As shown in Figure 3.1, after training, connections whose weights are close to zero are pruned to compress the model size [25, 43, 44]. The last row in Figure 3.1 shows the fraction of zero weights in the different layers of the AlexNet model trained using deep compression [43]. We find the sparsity to vary between 18%-85% across the different layers. Weight sparsity is static in nature because of the fact that zero weights are identified before the inference phase.

Dynamic Sparsity

Activation sparsity. Sparsity in activations stems from the thresholding nature of the activation function present at the output of each layer. As shown in Figure 3.1, the predominantly used ReLU (Rectified Linear Unit) activation function clips negative inputs to zero. Figure 3.1 also shows the average activation sparsity exhibited by various FFNN benchmarks, which ranges between $\sim$25%-60%. Activation sparsity

has a *dynamic* nature *i.e.,* because neurons whose outputs are zero vary considerably across inputs. Figure 3.2 illustrates this property for the activations produced by the conv3 layer of AlexNet. The activations are evaluated for two different inputs from the ImageNet dataset, and presented as black-and-white images where white pixels represent zero values. The example clearly illustrates that the fraction and locations of zero elements varies considerably across inputs.

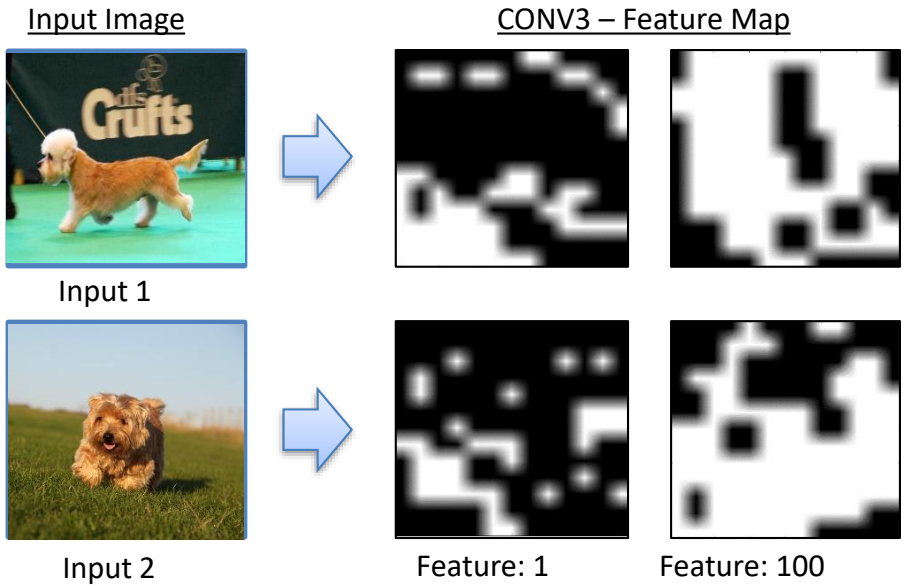

Fig. 3.2.: Variation in activation sparsity of AlexNet CONV3 layer across two different input images

Error Sparsity. Sparsity in the error data-structure originates from two sources. First, the derivative of the activation function, such as ReLU, is zero when the error at the output of the layer is negative. Next, when errors are propagated back through a max-pooling layer, as shown in Equation 3.1, only one input of each pooling window is set a non-zero error value.

$$\frac{\partial E}{\partial y_l}(x+p, y+q) = \begin{cases} 0, & \text{if } y_{l+1}(x,y) \neq y_l(x+p, y+q) \\ \frac{\partial E}{\partial y_{l+1}}, & \text{otherwise} \end{cases} \tag{3.1}$$

For example, if a pooling window of size 2×2 is used, at least three quarters of the error values are sparse. Similar to activation sparsity, the error sparsity is also dynamic. The average error sparsity in different layers of a FFNN trained for the CIFAR-10 dataset is shown in Figure 3.1. The error sparsity is considerable and varies from 75%-93%.

3.1.2 Opportunity for Computational Savings

Figure 3.3(a) shows the fraction of multiply-accumulate (MAC) computations that are rendered redundant for each FFNN benchmark due to dynamic sparsity in activations during inference. We find that between 25%-60% (average: 45%) of the computations can be skipped, underscoring the substantial opportunity improvement.

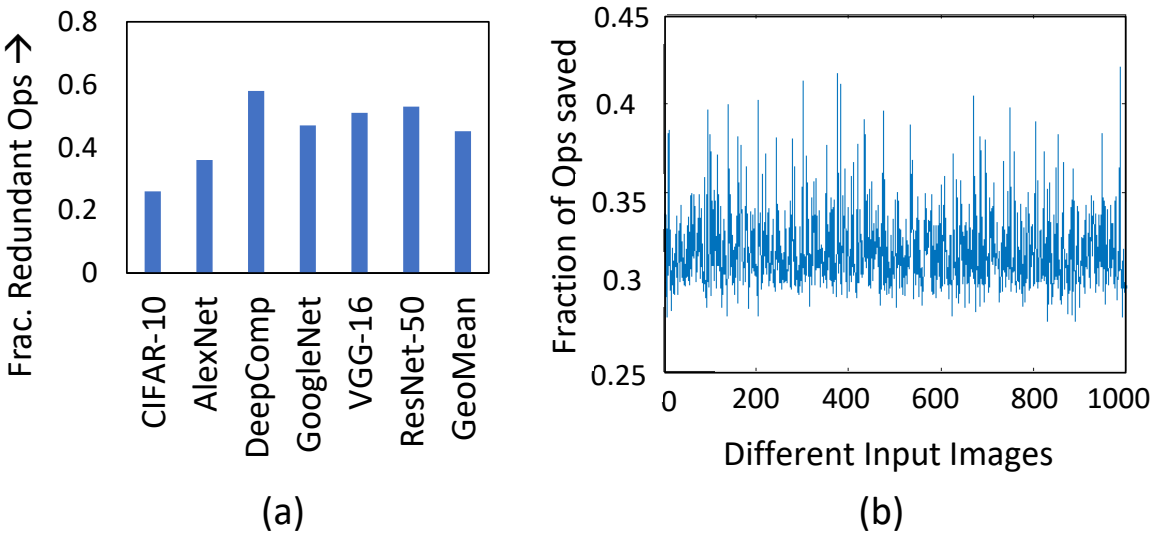

Fig. 3.3.: (a) Average fraction of redundant ops across benchmarks (b) Variation in fraction of redundant ops across different inputs of AlexNet

Figure 3.3(b) shows how the fraction of redundant operations varies across 1000 different inputs for the AlexNet CNN. We observe ∼14% variation across inputs,

although every input shows considerable opportunity for reduction in execution time (minimum: 28%).

In summary, the dynamic sparsity present in the activation and error data-structures offers a substantial opportunity to accelerate FFNNs. However, the levels of sparsity are not extreme enough to completely exploit them in software, and this coupled with their dynamic nature necessitates hardware solutions to realize benefits in the context of general purpose processors.

3.2 Long Short Term Memory Neural Networks

This section provides a brief review of Recurrent Neural Networks, Long Short Term Memory networks and sequence-to-sequence models. Recurrent Neural Networks are a class of neural networks designed to process sequential information with the help of memory or state. Individual symbols of an input sequence are presented to the RNN at each processing *timestep*. These inputs are used to modify the network state, thereby accumulating information from the past.

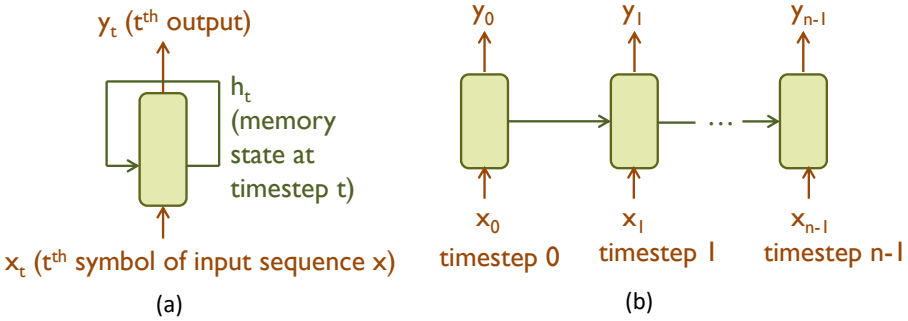

Fig. 3.4.: (a) Basic RNN (b) Time unrolled RNN

The basic structure of an RNN is shown in Figure 3.4(a). It operates on the t^{th} symbol, x_t, of the input sequence, x, at timestep t and modifies the state h_t, before feeding it back to the network at time $t+1$. The network is trained through the Backpropagation Through Time (BPTT) algorithm, performing backpropagation on an RNN unrolled into multiple timesteps (Figure 3.4(b)). Several RNN models, with

varying levels of ability to model sequences, have been proposed over the years [91–93]. We focus on the most commonly used model, called Long Short Term Memory networks (LSTMs). However, our proposed approximations can be applied to any RNN model that has clearly identifiable state and timesteps.

3.2.1 Long Short Term Memory networks (LSTMs)

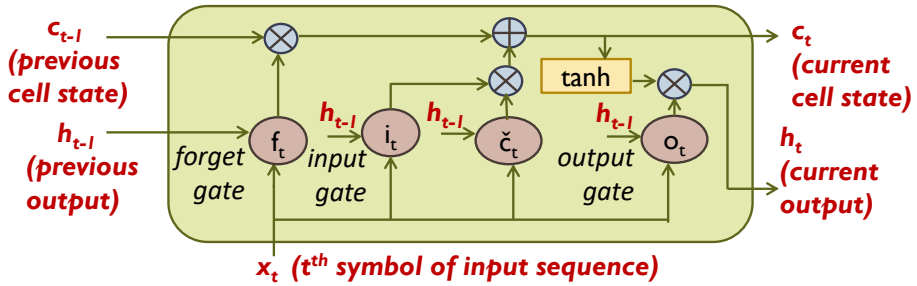

Fig. 3.5.: Long Short Term Memory cell

LSTMs [91] represent a special class of RNNs known for their ability to effectively learn long-term dependencies in sequences. Recent interest in RNNs has been largely fueled by LSTMs, and LSTMs and their variants account for most practical applications of RNNs. An LSTM is composed of cells. The structure of a cell is illustrated in Figure 3.5. Each cell has an associated state referred to as the cell state, c_t. Carefully regulated structures, called gates, control the addition and removal of information from the cell state. A gate is a neural network layer with a sigmoid activation function, followed by pointwise multiplication. The sigmoid output, with values between 0 and 1, dictates how much of each component should be let through for pointwise multiplication with the cell state. The forget gate, f_t, determines how much of the previous cell state, c_{t-1} should be passed on to the current time step. A new set of candidate values for the cell state, $\hat{c}_t$, is produced by a tanh layer. Subsequently, the input gate i_t determines the fraction of new candidate values to be added to the current cell state. Finally, the output gate, o_t, controls which parts of the cell state

go to the output, h_t, produced at timestep t. Mathematically, the computations in an LSTM can be represented by the following equations.

$$\begin{aligned} f_t &= \sigma(W_f \times [h_{t-1}, x_t] + b_f) \\ \hat{c}_t &= \tanh(W_c \times [h_{t-1}, x_t] + b_c) \\ i_t &= \sigma(W_i \times [h_{t-1}, x_t] + b_i) \\ c_t &= f_t * c_{t-1} + i_t * \hat{c}_t \\ o_t &= \sigma(W_o \times [h_{t-1}, x_t] + b_o) \\ h_t &= o_t * \tanh(c_t) \end{aligned} \tag{3.2}$$

Here, W_f, W_c, W_i and W_o are the matrices storing the weights of different gates; b_f, b_c, b_i and b_o are the biases of the different gates; c_t is the current cell state and h_t is the current output. Matrix-vector multiplications are indicated by $\times$ and element wise multiplications are indicated by $*$.

LSTMs are most commonly used for sequence-to-sequence learning, which we discuss next.

3.2.2 Sequence-to-Sequence Learning

Sequence-to-sequence learning refers to the ability to learn a mapping from input sequences in one domain to output sequences in a different domain. Sequence-to-sequence models are deployed in a wide range of tasks including neural machine translation [58], speech recognition [57] and video captioning [3]. As shown in Figure 3.6, a sequence-to-sequence model comprises of two structures — an encoder and a decoder. The encoder converts the input sequence into a fixed dimensional context vector, which is then used by the decoder to generate the output sequence. For example, in a machine translation task, the encoder utilizes the words of the source sentence to produce a context vector that summarizes the semantics of the sentence. The decoder subsequently operates on this semantic vector to generate translated words in the target language. Advanced sequence-to-sequence models employ en-

coder and decoder networks with multiple layers and residual connections between layers to enable effective learning of more complicated sequences.

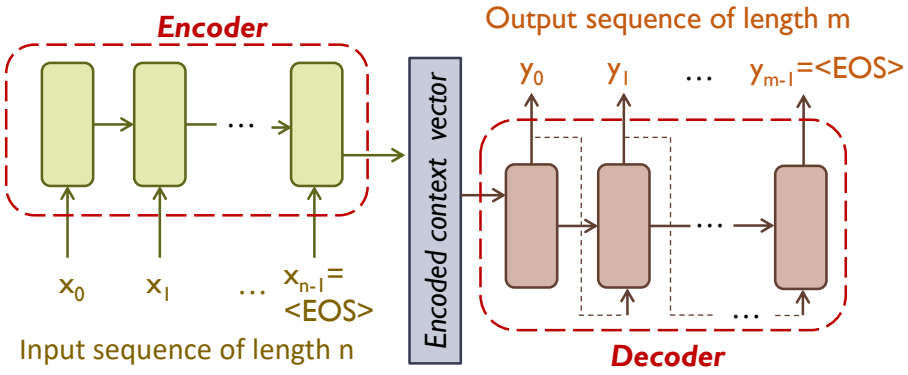

Fig. 3.6.: Sequence-to-sequence model

Evaluating sequence-to-sequence models. Sequence-to-sequence models are evaluated by determining the quality of output sequences generated by a model. Unlike image recognition tasks, which usually have a unique golden output, most tasks performed by sequence-to-sequence models may allow multiple correct outputs for a given input. For example, there could be multiple correct translations for a given input sentence, which vary in the choice of words or phrases, as well as the ordering thereof. Human evaluations of these output sequences can be expensive. Multiple automatic evaluation methods have been proposed for sequence-to-sequence models that generate natural language as their output. A few examples are BLEU [94], METEOR [95], ROUGE [96] and CIDEr [97]. We utilize the most popular metric BLEU (Bilingual Evaluation Understudy) to evaluate the quality of our sequence-to-sequence models. BLEU scores indicate how similar the generated sentence is to the reference sentences, with higher values representing more similar sentences. A BLEU score of 100 indicates that the generated sentence is identical to one of the reference sentences.

Execution time breakdown for sequence-to-sequence models. The overall execution time of a sequence-to-sequence model is a function of 4 major parameters as shown in equation 3.3.

$$\begin{aligned} &InputSeqLen \times ComputeTimePerInputSymbol + \\ &OutputSeqLen \times ComputeTimePerOutputSymbol \end{aligned} \tag{3.3}$$

where $InputSeqLen$ and $OutputSeqLen$ denote the lengths of the input and output sequences and $ComputeTimePerInputSymbol$ and $ComputeTimePer\ OutputSymbol$ denote the time required to process a single symbol of the input and output sequence. Overall, the first product term represents the encoding time while the second product term represents the decoding time. Three of the four parameters, namely, $InputSeqLen$, $ComputeTimePerInputSymbol$ and $ComputeTimePerOutputSymbol$ are deterministic in nature and depend on the number of input symbols processed and the amount of computation in the encoder and decoder, respectively. The remaining parameter, $OutputSeqLen$, is non-deterministic and is influenced by the mapping performed by the sequence-to-sequence model. Reducing the execution time of sequence-to-sequence models amounts to reducing one or more of the four parameters mentioned above, without adversely affecting the overall quality of the model.

In summary, LSTMs and sequence-to-sequence models in particular have structures that are significantly different from feedforward neural networks. The exploration of techniques exploiting their specific characteristics is key to improving their execution efficiency.

3.3 Spiking Neural Networks

Just like any other neural networks, SNNs are interconnected networks of primitive compute units, called neurons, that are organized in layers, with neurons in each layer connected to those in the layer succeeding it. The junction between connected neurons is called a synapse, which is associated with a parameter, the weight, that signifies the strength of the connection. The synaptic weights are learnt during the

training process. In SNNs, as shown in Figure 3.7, information is represented and processed using *spikes*, which take a binary value 0 or 1. Inputs are presented to the neurons in the first layer as a time series of spikes. The spike trains propagate through the network until the output layer is reached. Each neuron in the output layer is associated with a class label, and the input is assigned the class corresponding to the output neuron that spiked the largest number of times. The number of time steps for which the SNN is evaluated is a key network parameter, and is determined during training.

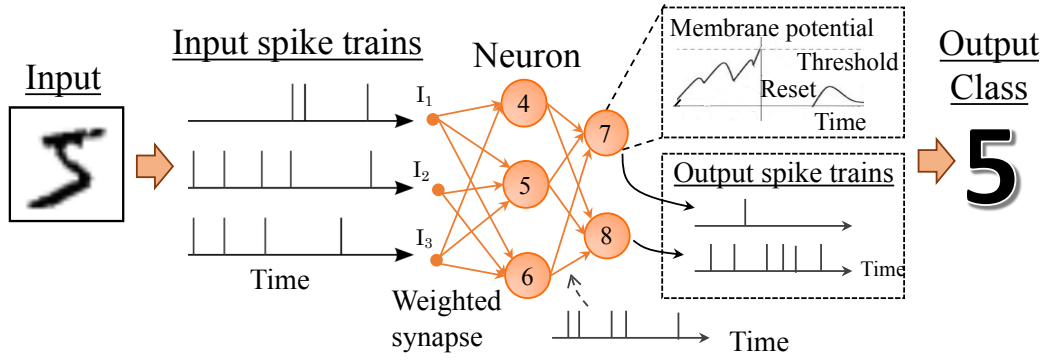

Fig. 3.7.: Spiking neural network preliminaries

Several spiking neuron models, with varying levels of biological fidelity, have been proposed [98]. In this dissertation, we consider the most commonly used model, called Leaky-Integrate-and-Fire (LIF); however, our proposed approximation method is independent of the neuron model used in the SNN. The LIF neuron has 3 key parameters *viz.,* the membrane, threshold and reset potentials. At the start of evaluation, the membrane potential is initialized to the reset value. Whenever a spike is observed at an input, the membrane potential is updated by the weight of the connection. In addition, in each time step, the membrane potential leaks (is decremented) by a fixed value. Mathematically, the LIF neuron is represented by Equation 3.4.

$$V_i(t) = V_i(t-1) - V_{leak} + \sum_j w_{ji} A_j(t) \tag{3.4}$$

where $V_i(t)$ is the membrane potential of neuron i at time t, V_{leak} is the leakage potential, w_{ji} is the weight of the synapse connecting neuron i and its input j, and A_j is a binary variable indicating whether input j spiked at time t. The LIF neuron produces an output spike whenever its membrane potential exceeds the threshold, following which the potential is re-initialized to the reset value.

The key compute primitive in SNNs is the set of updates triggered by a neuron spike. In this case, the potentials of all of its fanouts are updated by the respective synaptic weights. These updates may in turn cause new spikes to be generated and subsequently processed.

In summary, SNNs are event-driven workloads, wherein work is dynamically generated as neurons spike in the network. The set of updates triggered by each spike is the basic unit of work.

3.4 Pruning in DNNs

Pruning refers to the process of removing insignificant connections and neurons in a DNN and setting the corresponding weight values to zero. The benefits of pruning, and the weight sparsity borne out of it, are two-fold. First, as discussed in Section 3.1 in greater detail, the zero weight values lead to redundant MAC operations which can be skipped to obtain computational savings. Second, the pruned weights can be compressed and stored compactly in memory by utilizing different sparse formats.

A wide variety of pruning techniques have been proposed over the years, which can be grouped into two broad categories based on the granularity in which they introduce zeros. Fine-grained or element-wise pruning zeros out individual weights in the network [43–45], while coarse-grained or structured pruning zeros out well-defined groups of weights [99, 100]. Coarse-grained pruning leads to regular compute and data access patterns in DNNs, enabling easy exploitation of weight sparsity. It can also directly reduce the memory requirements without requiring sparse storage formats, specially in the extreme forms of channel and filter pruning. The main ad-

vantage of fine-grained pruning, on the other hand, is its ability to achieve higher sparsity levels in the lack of any shape or size constraints. It can also significantly prune hand-designed compact DNN models, like Mobilenet [101], as well as, compact models designed through neural architecture search (NAS) techniques, like NasNet-Mobile [101], which have reduced opportunities for coarse-grained pruning.

As part of our work on studying the efficacy of pruning in ultra-low precision DNNs, we focus on fine-grained pruning and consider the technique of Automated Gradual Pruning (AGP) [102] which can successfully realize a DNN with any target sparsity level. It gradually increases the sparsity level, and the number of pruned weights, in each pruning step or epoch until the target sparsity level is achieved. As expected, higher target sparsity levels can affect network accuracies significantly. In order to minimize this, the weights are pruned on the basis of their magnitudes, with all weights below a particular magnitude pruned in each step.

3.5 Quantization in DNNs

Quantization in DNNs reduces the number of bits used to represent the DNN data-structures. It is based on the observation that DNN data-structures dont require the full representations of double (64-bit) or single (32-bit) floating-point numbers. Instead, DNNs can produce correct outputs and achieve iso-accuracy levels even when their data-structures are represented with just 8 bits or less [103]. Quantization provides both computational and memory benefits in DNNs as the low-precision operations can be performed with arithmetic units of lower complexity, and the low-precision data-structures can be stored using less number of total bits. A broad range of DNN quantization approaches have been proposed over the past few years [38,103–105]. These approaches explore the use of different number formats (floating-point or fixed-point), uniform and non-uniform quantization techniques, as well as, post-training quantization and quantization-aware training processes. We

consider post-training quantization of DNNs to fixed-precision representations in the ultra-low precision DNNs explored in our work.

3.6 Adversarial Attacks on DNNs

Adversarial attacks modify inputs in a manner that force a DNN model to misclassify, while ensuring that the input changes are small and imperceptible to human eyes. In the context of DNNs that operate on images, which are the focus of most prior work, various attack methods have been proposed to systematically modify pixel values in the input image so as to result in a mis-classification. A few such methods are described below.

Fast Gradient Sign Method (FGSM) [31]. FGSM is a single-step attack that operates by calculating the gradient of the loss function with respect to the input pixels ($\nabla_x L(\theta, X, Y)$). Based on the sign of the loss, the input pixels are increased or decreased by a small constant, ϵ, to help move the image towards the direction of increased loss. The adversarial input, X_{adv} can be computed as:

$$X_{adv} = X + \epsilon Sign(\nabla_x L(\theta, X, Y)) \tag{3.5}$$

Here, X is the original input image associated with an output Y and θ refers to the weights of the network.

Basic Iterative Method (BIM) [33]. BIM is an iterative version of the FGSM attack which performs a finer optimization by modifying pixels by small values in each iteration. Further, the image generated in each iteration has its pixel values clipped to ensure minimal distortion. Mathematically, this attack can be described as:

$$X^0_{adv} = X, \quad X^{N+1}_{adv} = Clip_{X,\epsilon}\{X^N_{adv} + \alpha Sign(\nabla_x L(\theta, X^N_{adv}, y))\} \tag{3.6}$$

Here, the terms X, Y, θ and ϵ have the same meaning as in Equation 3.5 and X^N_{adv} refers to the adversarial input generated at the N^{th} iteration and α is the step size in each iteration.

Carlini-Wagner (CW) [51]. CW is another iterative attack that employs optimizers to create strong adversarial inputs by simultaneously minimizing the input distortion and maximizing the misclassification error. It can be described mathematically as:

$$\begin{gathered}\min_{\delta}\|\delta\|_2^2 + c \cdot f(X+\delta) \text{ such that } (X+\delta) \in [0,1]^n \\ f(X) = \max(\max_{i \neq t}\{Z(X)_i\} - Z(X)_t, 0) \\ X_{adv} = X + \delta\end{gathered} \tag{3.7}$$

where δ is the input distortion, c is the Lagrangian multiplier, $Z(X)$ is the logit output for the input X, t is the target class and $f(X)$ is an objective function that satisfies the condition $f(X+\delta) \leq 0$ for all misclassifications.

Projected Gradient Descent (PGD) [32]. PGD is a third type of iterative attack very similar in nature to the BIM attack. Unlike BIM, which starts with the original image itself, PGD starts with a random perturbation of the original input image. PGD can be described by the following equations:

$$\begin{aligned}X_{adv}^0 &= X + randomUniform(shape(X), \{-\epsilon, \epsilon\}) \\ X_{adv}^{N+1} &= Clip_{X,\epsilon}\{X_{adv}^N + \alpha Sign(\nabla_x L(\theta, X_{adv}^N, y))\}\end{aligned} \tag{3.8}$$

Here, the terms X, Y, X_{adv}^N, θ, ϵ and α have the same meaning as in Equation 3.6.

To summarize, different adversarial attacks have been proposed that expose the lack of robustness in current DNN models by constructing adversarial inputs that force a misclassification. Developing defenses to these adversarial attacks is critical to enable the deployment of DNNs in safety-critical systems.

3.7 Test Pattern Compression

Test pattern compression is widely used to reduce the size of tester memories in IC testing by exploiting the redundancies in test patterns. Various test pattern compression techniques have been proposed over the years utilizing reseedable-LFSR-codes [106], XOR networks [107] and Golomb codes [108]. We utilize Embedded

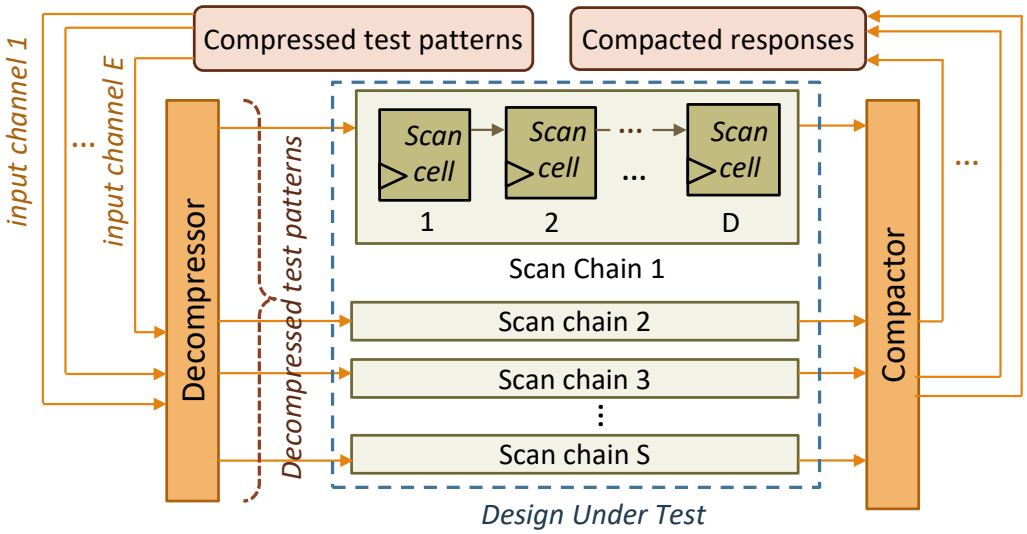

Fig. 3.8.: Test pattern compression architecture

Deterministic Test (EDT) [109] within Mentor Graphics Tessent tool to perform test pattern compression in our work. The overall test compression architecture in EDT is shown in Figure 3.8. It consists of a design-under-test (DUT) with multiple scan chains, each of which has several scan cells. The compressed test patterns are stored in the external test memory and fed to the decompressor to obtain the individual test patterns fed to the scan chains. At every test cycle, E bits fed to the decompressor are converted to S bits, where E and S are the number of external input channels and the number of internal scan chains, respectively. Thus, the achieved test pattern compression ratio achieved equals S/E. The compactor performs the inverse operation of the decompressor and compacts the outputs generated by the scan chains to store them externally as compacted responses, which are subsequently compared with the golden test results.

4. SPARCE: SPARSITY AWARE GENERAL-PURPOSE CORE EXTENSIONS TO ACCELERATE FEED-FORWARD NEURAL NETWORKS

Deep Neural Networks (DNNs) have transformed the field of machine learning and have greatly advanced the state-of-the-art in image, video, text and speech processing [1–3, 6, 110]. As mentioned in Chapter 1, their success in the domain of computer vision applications can primarily be attributed to the development of Feed-Forward Neural Networks (FFNNs), specially, Convolutional Neural Networks (CNNs). As FFNNs get pervasively deployed, a key challenge that arises is their computational demand, which far outstrips the capabilities of modern computing platforms. For example, ResNet-152 [1], a state-of-the-art CNN for image recognition, requires ∼11.3 giga operations to classify a single 224×224 image.

The computational demands of FFNNs have most commonly been addressed through the design of specialized accelerator architectures that exploit the unique compute and communication characteristics of FFNNs. However, we focus on deeply embedded systems such as wearables and IoT edge devices where the additional area and cost imposed by custom accelerators is prohibitive. For example, even a low power accelerator such as Eyeriss [26] occupies $12.25mm^2$ area which is 30× larger than the $0.4mm^2$ occupied by an ARM Cortex A35 core [111]. Thus, in highly area-constrained platforms, FFNNs are typically executed on the general-purpose processor (GPP) cores already present in them. We focus on improving the execution time of FFNNs on GPPs by leveraging *sparsity* in different FFNN data-structures, *viz.*, activations, weights and backpropagated errors. Sparsity in FFNNs can be both static or dynamic, depending on whether the zero values remain constant or vary across different inputs to the network. Sparsity in weights, introduced by pruning connections in the network after training, is static in nature. In contrast, activation

and error sparsities, caused by the thresholding nature of the ReLU (Rectified Linear Unit) activation functions, are dynamic in nature. We note that sparsity leads to truly redundant MAC operations in FFNNs which can be skipped with no effect on the output quality (*e.g.* classification accuracy).

	Static sparsity	Dynamic sparsity
Accelerators	Eyeriss[18], EIE[27], SCNN[28], Cambricon-x[29], GPU ZeroSkip[30]	Eyeriss[18], Cnvlutin[26], EIE[27], SCNN[28]
General Purpose Processors	SparseCNN[31], SSL[32], Scalpel[33], SPARCE	SPARCE

Fig. 4.1.: Related work: Exploiting sparsity in FFNNs

Prior efforts that exploit sparsity to accelerate FFNNs can be grouped into two classes based on whether they exploit static or dynamic sparsity, as shown in Figure 4.1. Specialized accelerators [26, 68–72] have been proposed to exploit both forms of sparsity. As mentioned previously, the area and cost overhead of these accelerators is prohibitively high for many IoT edge devices. Software-only approaches [73–75] allow static sparsity to be exploited on GPPs through sparse encodings and sparse matrix multiplication routines. However, these techniques are unable to exploit dynamic sparsity because of the encoding overhead involved at runtime. We observe across 6 image-recognition FFNNs that dynamic sparsity in features ranges from 40%-70%, resulting in 45.1% of the computations being rendered redundant during inference, presenting a significant opportunity for improving performance. We believe our effort, SPARCE, is the first to successfully exploit dynamic sparsity to accelerate FFNNs on GPPs. Moreover, since static sparsity is a special case of dynamic sparsity where the location of zeros can be identified a priori, SPARCE is a single solution that exploit both forms of sparsity.

To exploit dynamic sparsity, GPPs need to be equipped to dynamically detect if the result of an instruction (*e.g.*, a load from a sparse data structure) is zero and if so, skip a set of future instructions that use its result. However, there are three key challenges: (i) the instructions to be skipped often do not immediately follow the instruction that determines whether they are redundant; moreover, the instructions to be skipped may be non-contiguous in the program, (ii) to maximize performance benefits, the instructions to be skipped should be *prevented from even being fetched*, as squashing the instruction in-flight would diminish the benefits by introducing a pipeline stall, and (iii) GPPs often utilize SIMD (Single Instruction, Multiple Data) execution units which allow the instructions to be skipped only if the computations performed in all the SIMD lanes are redundant.

Sparsity aware Core Extensions (SparCE). To overcome the aforementioned challenges, Sparsity aware Core Extensions (SparCE) proposes two key micro-architectural enhancements to GPPs. First, SparCE contains a *Sparsity Register File* (SpRF) to track general purpose registers that contain zero values. We achieve this by augmenting the writeback stage of the processor to check if the update to a register is zero and appropriately modify the corresponding SpRF entry. Next, a *Sparsity-Aware Skip Address* (SASA) Table is used to store instruction sequences and the conditions under which they can be skipped *i.e.,* the registers in the SpRF that need to be zero for the instructions to become redundant. Whenever SparCE fetches an instruction, it uses the SASA Table and the SpRF to *pre-identify* whether the following instruction(s) can be skipped. If so, the program counter is modified to directly fetch the next irredundant instruction. In order to suitably leverage these micro-architectural extensions, we also propose a code generation process for SparCE that allows successful identification of redundant instruction sequences and subsequent programming of the SASA Table.

In summary, the key contributions of this work are:

- We propose Sparsity aware Core Extensions (SparCE) to accelerate FFNNs on general purpose processors by skipping redundant computations borne out of sparsity in the different FFNN data-structures.
- SparCE comprises of micro-architectural enhancements to dynamically track when the result of an instruction is zero, pre-identify future instructions that are rendered redundant, and prevent them from being fetched and executed, thereby improving performance.
- We evaluate SparCE on a suite of 6 state-of-the-art image-recognition FFNN benchmarks using the Caffe deep learning framework. We achieve application-level speedups of 19%-31% on a scalar ARMv8 microprocessor. We also achieve speedups of 8%-15% over highly optimized baseline implementations that use OpenBLAS on an ARMv8 processor with 4-way SIMD and prefetching support.

The rest of the chapter is organized as follows. Section 4.1 details the key design principles of SparCE and demonstrates them in the context of an in-order pipelined processor. Section 4.2 describes the code generation process and shows SparCE in action using the ARM-BLAS GEMM routine as a case study. Section 4.3 describes the experimental methodology and the results are presented in Section 4.4. Finally, Section 4.5 summarizes the chapter.

4.1 SparCE: Sparsity Aware General Purpose Core Extensions

To exploit the different forms of sparsity and improve FFNN performance on GPPs, we propose Sparsity aware Core Extensions (SparCE), a set of micro-architectural and ISA extensions that are general-purpose, minimally intrusive and low-overhead. In this section, we present the key ideas behind SparCE and describe how they can be integrated within an in-order processor pipeline.

4.1.1 Challenges

The key challenge in exploiting sparsity is to equip the processor with the ability to dynamically detect if the result of an instruction is zero and if so, skip a list of future instructions that are rendered redundant. We illustrate this challenge using the assembly code snippet shown in Figure 4.2, which computes the dot-product of two vectors, INP and KER, each of size N, to produce a scalar OUT. Registers $r0$, $r1$ and $r2$ hold the data operands, while $p0$, $p1$ and $p2$ are pointers that hold their respective memory locations. For each instruction in the program, Figure 4.2 shows the instructions that can be skipped when its result is zero. For example, when the INP load returns a zero (Inst. 2), the subsequent KER load (Inst. 4), and the multiply and add instructions can be skipped (Insts. 6-7). It is noteworthy that the computational savings is a weighted sum of the number of instructions skipped and the cycles taken by each instruction. For instance, floating point multiply and add instructions may take 3-5 cycles to execute, while a load incurs variable cycles depending on the level of cache hierarchy accessed.

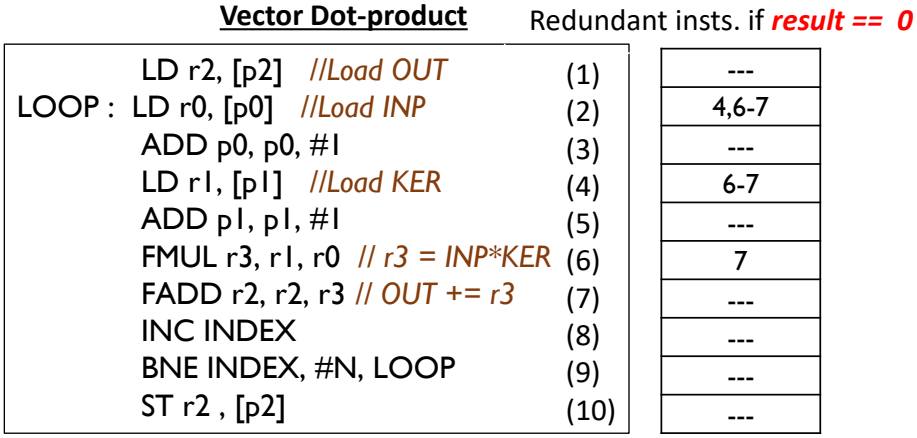

Fig. 4.2.: Redundant instructions due to sparsity in vector dot-product evaluation

The following observations highlight the challenges in detecting and benefiting from sparsity.

- **Location of redundant instructions.** In a program, the instructions that can be skipped may not immediately follow the instruction producing the zero result. Worse, redundant instruction sequences may be scattered non-contiguously through the program. For instance, in the program shown in Figure 4.2, when inst. 2 returns a zero, 2 non-contiguous instruction sequences (Inst. 4 and Insts. 6-7) need to skipped. Hence, efficient ways to *capture which instructions can be skipped* is key to leveraging sparsity.

- **Avoiding redundant instruction fetches.** To maximize performance, the *instructions that can be skipped need to be prevented from even being fetched.* This is especially critical in the context of in-order pipelines, where even if the instruction is squashed after being fetched, it would introduce a bubble in the pipeline. For example, if the condition for $r0$ or $r1$ being zero is checked after the $FMUL$ instruction is fetched (Inst. 6), it would result in a bubble flowing through the pipeline in place of Inst. 6. It is worth noting that, in the context of multi-cycle instructions, squashing the instruction after it is fetched can still improve performance.

- **Use of SIMD instructions.** GPPs use vector units with SIMD (Single Instruction, Multiple Data) execution engines to exploit fine-grained data parallelism in workloads. SIMD instructions can be skipped only if computations performed on all the vector lanes are redundant. This constrains the sparsity to be relatively coarse grained, as irregularly scattered zero values are not beneficial.

4.1.2 SparCE: Overview

Figure 4.3 shows an overview of the micro-architectural and ISA enhancements proposed in SparCE. We describe these extensions in detail, and demonstrate how they address the aforementioned challenges to leverage sparsity in CNNs.

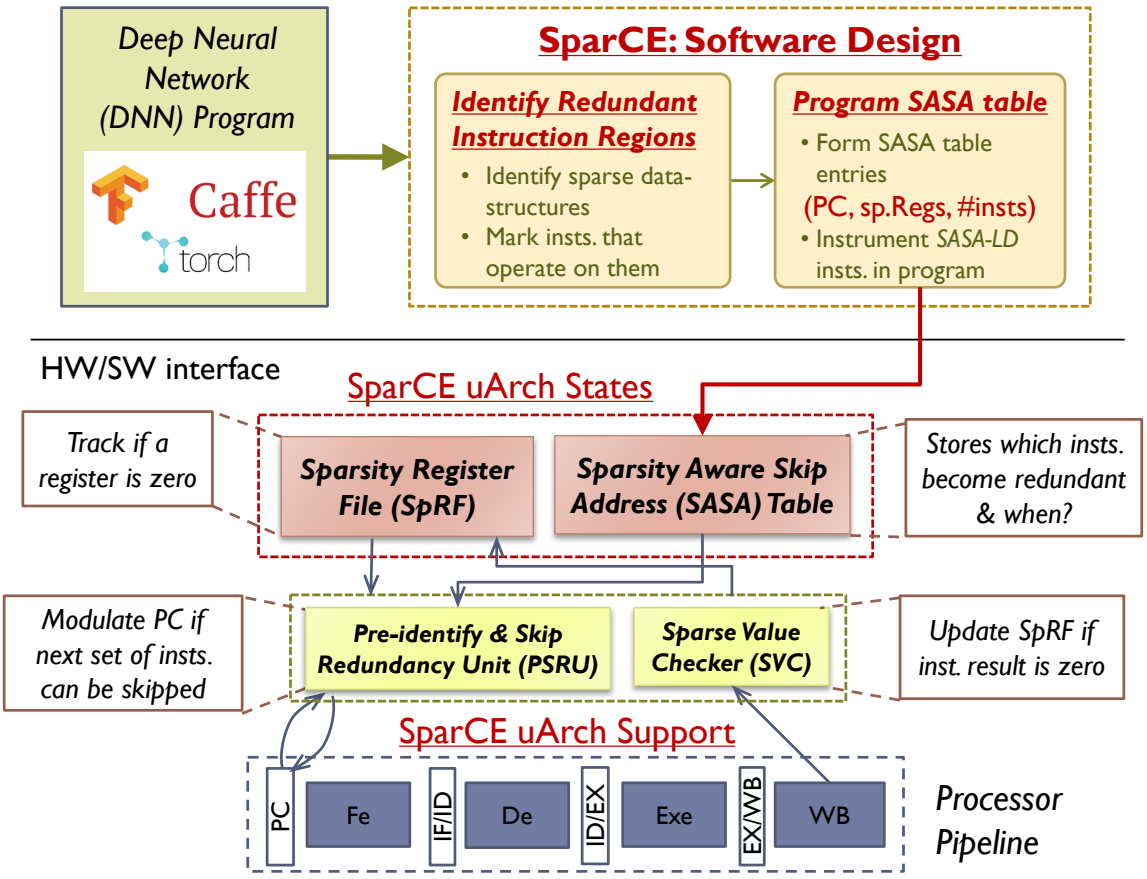

Fig. 4.3.: SparCE: Design Overview

Micro-architectural states and ISA Extension

In SparCE, we augment the processor with two new micro-architectural states *viz.* Sparsity Register File (SpRF) and Sparsity-Aware Skip Address (SASA) table. The SpRF is used to dynamically track which registers in the processor's register file

contain zero values. The SpRF contains one entry (few bits) corresponding to each register in the register file. When an instruction that writes to a register retires, the SpRF is updated if the result is zero. The SASA table stores information about which instructions can be skipped and under what conditions. Specifically, each entry stores the program counter (PC) of the instruction preceding a redundant instruction sequence, the index of the register that determines redundancy and the length of the sequence. Storing the PC of the instruction preceding a redundant instruction sequence allows SparCE to *pre-identify whether the next set of instructions can be skipped and if so, skip them before the instructions are fetched.*

SASA-LD Instruction. SparCE enables software to explicitly identify potentially redundant instruction regions by pre-loading the SASA table at program startup, or before the program execution enters a given code region. To this end, we extend the ISA with a new instruction *viz.* $SASA\text{-}LD$, which loads a given region of memory into the SASA table. As shown in Equation 4.1, the $SASA\text{-}LD$ instruction takes a register operand (Rn) that points to the SASA table's location in memory and an immediate operand ($size$) that denotes the size of the SASA table.

$$SASA\text{-}LD \quad [Rn], \#size \tag{4.1}$$

At a given point in the program execution, the number of entries in the SASA table limits the number of redundant instruction regions that can be skipped by SparCE. However, the SASA table can be periodically refreshed from memory as the program execution progresses. In the context of CNNs, we found that 20 entries in the SASA table suffice to capture all redundant instruction sequences, since the computational kernels are captured by a small number of library (*e.g.*, BLAS) functions.

Tracking, Pre-identifying, and Skipping Redundant Instructions

SparCE utilizes the SpRF and the SASA table to dynamically skip redundant instructions borne out of sparsity in the input data-structures. As shown in Figure 4.3, the micro-architecture of SparCE is extended to support the following functions.

Track Sparse Registers. SparCE contains a Sparse Value Checker (SVC), which in the processor's writeback stage compares the result of each instruction to zero and if so, updates the entry corresponding to the instruction's destination register in the SpRF.

Pre-identify & Skip Redundant Instructions. SparCE is equipped with a Pre-identify and Skip Redundancy Unit (PSRU) that utilizes the SASA table to identify and skip redundant instruction regions. For each instruction, we check if its PC contains an entry in the SASA table. An entry in the SASA table indicates that the instruction following the current instruction is the start of a potentially redundant instruction sequence. In this case, the PSRU checks the SpRF to identify if the registers indicated in the SASA table entry are currently zero. If so, it increments the PC to the end of the redundant instruction sequence, thereby skipping instructions to benefit performance. If not, SparCE proceeds to execute instructions in program order.

In summary, SparCE uses the SpRF and the SASA table to seamlessly track sparse registers, pre-identify instruction sequences that are redundant and dynamically skip them before they are even fetched to improve performance.

4.1.3 In-order SparCE Processor Pipeline

We now explain how SparCE can be integrated into an in-order processor. Figure 4.4 shows the block diagram of the overall SparCE processor architecture. We start with a conventional 4-stage (fetch, decode, execute/memory, and writeback) pipelined processor architecture implementing a RISC-style instruction set with at most 2 source register operands and one destination register operand. Although the SparCE architecture is described in this section with a scalar execution unit for ease of illustration, it is directly applicable to vector processors with any number of SIMD execution lanes. We augment the processor with the following structures.

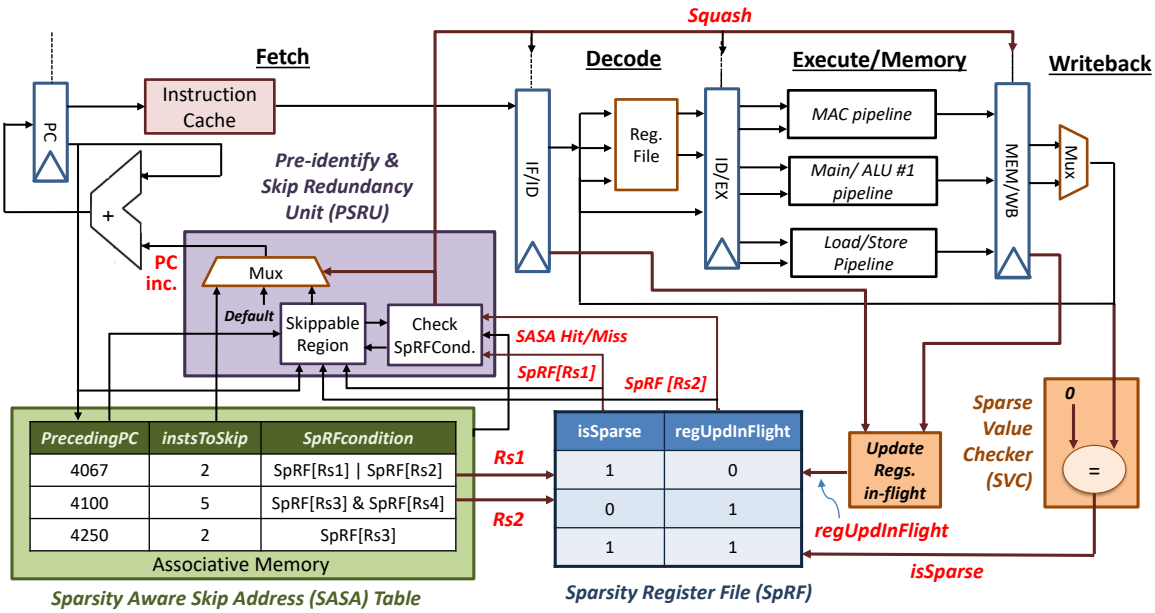

Fig. 4.4.: Block diagram of SparCE in-order processor architecture

Sparsity Register File (SpRF). The SpRF is located at the fetch stage of the SparCE processor. It is a multi-ported register file, with one entry corresponding to each register in the processor's register file. Each entry in the SpRF contains only two single-bit fields - *isSparse* and *regUpdInFlight*. The *isSparse* bit is set to 1 for registers containing zeros, and reset otherwise. The *regUpdInFlight* bit indicates whether an instruction modifying the register is in flight in any stage of the pipeline. For instance, an instruction modifying register Rd, sets the $SpRF[Rd][regUpdInFlight]$ field when it enters the decode stage and resets it after committing its result in the writeback stage. In determining whether a future instruction is redundant, the *regUpdInFlight* field ensures that we do not use a stale *isSparse* value when a more recent instruction updating this register is under execution.

Sparse Value Checker (SVC). The SVC is added to the writeback stage of the SparCE processor. It contains a comparator that checks if the output of each instruction that updates a register is zero. It then correspondingly updates the SpRF.

For example, when the output of an instruction with destination register Rd is zero, then $SpRF[Rd][isSparse]$ is set and $SpRF[Rd][regUpdInFlight]$ is reset.

Sparsity-Aware Skip Address (SASA) table. The SASA table is also present in the fetch stage of the SPARCE processor. As shown in Figure 4.4, the SASA table is an associative memory structure with three fields: (i) $precedingPC$, which stores the PC of the instruction prior to the redundant instruction sequence, (ii) $SpRFCondition$ field which stores a Boolean combination of 2 register indices in the SpRF that should be satisfied for the instruction region to be skipped, and (iii) $instsToSkip$, which contains the length of the redundant instruction sequence. As an example, for the code in Figure 4.2, the SASA table entry to skip instructions 6-7 would be $\{precedingPC{=}5,\ SpRFCondition{=}r0|r1,\ instsToSkip{=}2\}$.

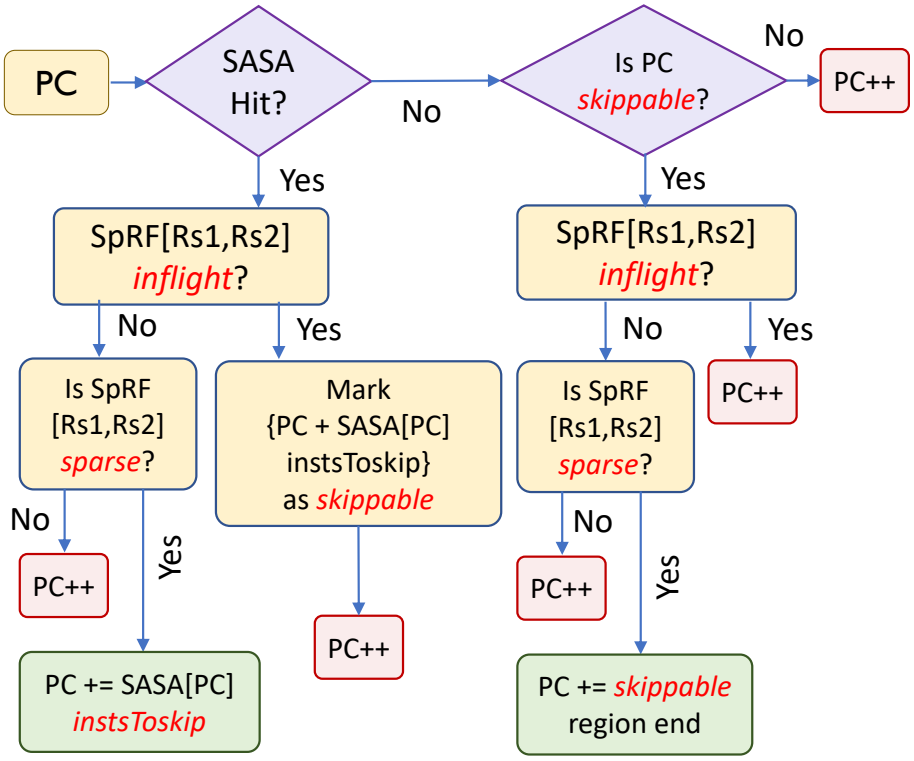

Fig. 4.5.: Flowchart for pre-identifying and skipping redundancy

Pre-identify and Skip Redundancy Unit (PSRU) The PSRU is also added to the fetch stage of SPARCE processor. The PSRU determines whether an instruction region specified in the SASA table can be skipped, and appropriately modulates the PC. Figure 4.5 shows the flowchart depicting the operation of PSRU. Given a PC, an associative lookup is performed on the $preceedingPC$ field of the SASA table. If the PC is a hit in the SASA table, then the next instruction marks the beginning of a potentially redundant instruction sequence. To ascertain if the instruction sequence can be skipped, the PSRU reads the register indices specified in the $SpRFCondition$ field of the SASA table (say R_{s1} and R_{s2}) from the SpRF. If neither $SpRF[R_{s1}]$ nor $SpRF[R_{s2}]$ have their $regUpdInFlight$ field set, then PSRU computes the Boolean condition (specified in the $SpRFCondition$ field) on their respective $isSparse$ fields. If the condition is satisfied, then the instruction region is deemed redundant and the PC is incremented by $instsToSkip$ to point to the instruction immediately following the end of the redundant region. If not, PC is incremented by 1 and the instructions are executed in program order. However, if the $regUpdInFlight$ field is set for either $SpRF[R_{s1}]$ or $SpRF[R_{s2}]$ and the Boolean condition in $SpRFCondition$ cannot be definitively evaluated, then the instruction region is temporarily marked as a *skippable* region within the PSRU. The PC is incremented by 1 and the program execution is continued. It is worth noting that continuing or aborting execution of a skippable region will *not* affect program functionality.

In the case when a PC is not present in the SASA table, the PSRU checks if the PC is part of an active skippable region. For the registers present in the $SpRFCondition$, the $regUpdInFlight$ fields are re-checked from the SpRF. If they are reset, the Boolean condition in $SpRFCondition$ is evaluated. If the skippable region is determined to be redundant, then the remaining instructions in the region are skipped by appropriately modifying the PC. Further, instructions belonging to the skippable region prior to the current PC are squashed if they are still in flight. If the Boolean condition evaluates to a *false*, then the remaining instructions in the skippable region are executed. Finally, for a PC that misses the SASA table and is

not part of any active skippable region, the PC is incremented by 1 to fetch the next instruction.

In general, branches can also be a part of redundant instruction regions if the branches are guaranteed to be always not-taken when the associated registers are sparse. In the absence of such a guarantee, the redundant instruction regions are fragmented into multiple segments without the branches. This ensures that the branches are always evaluated and the control flow of the program remains unaffected. Irredundant branch instructions preceding a redundant instruction region can potentially pose a challenge as the result of its execution (whether the branch will be taken or not) is unknown. We address this challenge by changing the preceding PC in the SASA Table from the branch instruction to the following redundant instruction. This ensures that we skip the redundant instructions only if the branch takes the control flow to that region. However, the observed benefits may be lower due to the inability to skip the first instruction in the redundant region.

We note that the logic introduced in SparCE to pre-identify redundant instructions executes in parallel with the instruction cache (ICache) access. The additional logic does not impact the latency of the fetch stage, as both the SASA table and the SpRF are significantly smaller structures compared to the ICache.

In summary, by using the SpRF and the SASA table, SparCE dynamically tracks sparse registers, pre-identifies if an instruction region is redundant and skips instructions before they are even fetched to improve performance. Thus SparCE enables CNN acceleration on GPPs by exploiting the sparsity resident in their data-structures.

4.2 Software for SparCE Processsors

To extract maximum performance from SparCE, the software needs to suitably leverage the sparsity-aware micro-architectural extensions. In this section, we outline the key principles behind software design for SparCE, and demonstrate them in the

context of a highly optimized implementation of matrix multiplication (GEMM) from the OpenBLAS library.

4.2.1 Code Generation for SparCE

Figure 4.6 presents an overview of the code generation process for SparCE. The individual steps are subsequently described in more detail.

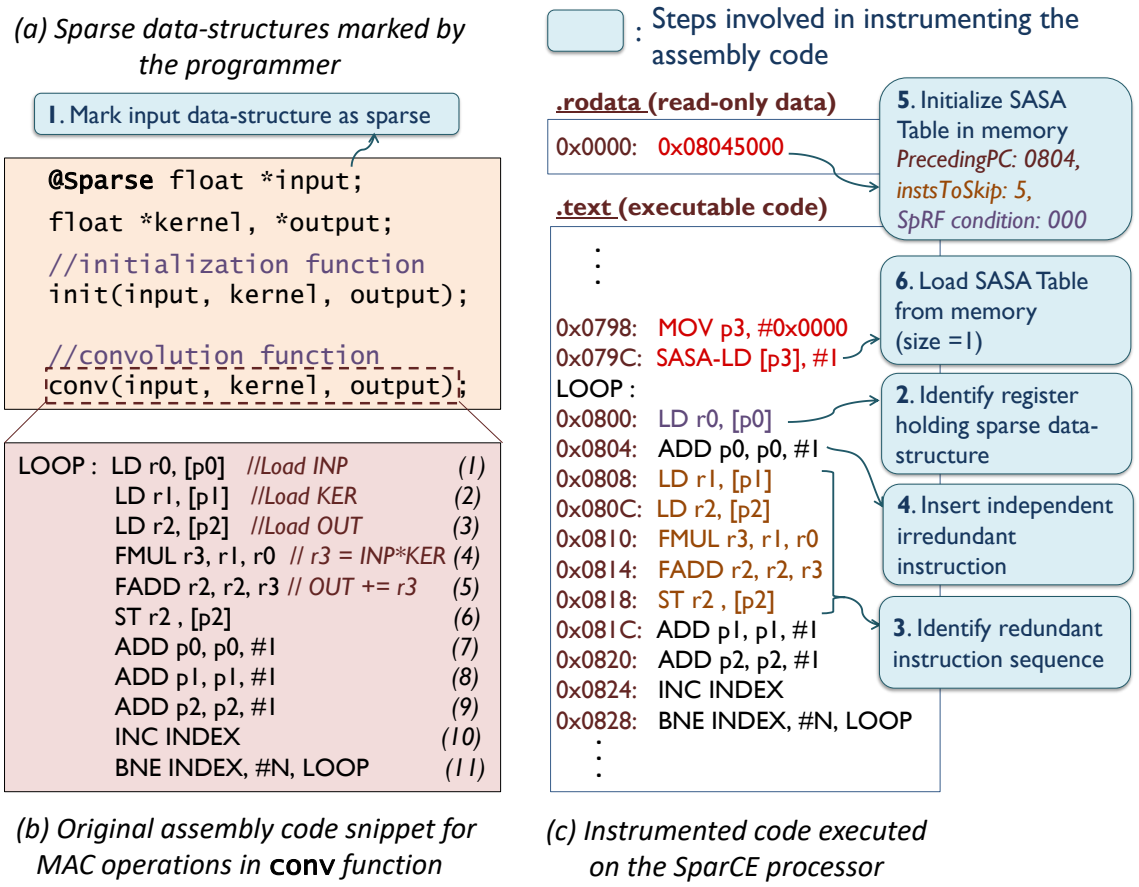

Fig. 4.6.: Code generation for SparCE

Identifying sparse data-structures and redundant instructions. One of the key requirements on software is to identify which instruction regions are potentially redundant due to sparsity. To that end, the programmer first annotates sparse data-types with `@Sparse` qualifiers in a high-level program. In the example shown in

Figure 4.6(a), the data-structure named `input` is marked as sparse. Next, when the application is compiled, the registers into which the sparse data-structures are loaded from memory are identified along with the corresponding load instructions (step 2 in Figure 4.6(c)). Then, a static dependency analysis of the instruction stream reveals the instructions that become redundant with zero values in the sparse data-structures. These are marked as potentially redundant instruction sequences, as mentioned in step 3 of Figure 4.6(c).

Reordering instructions to improve benefits. To completely skip a redundant instruction region, the instruction whose result makes them redundant should have completed execution. Therefore, in the context of in-order processors, they should ideally be spaced at least few instructions apart (3 in our as case as the SVC is located in writeback stage) in the program to prevent squashing and subsequent introduction of pipeline bubbles. To this end, the instruction stream is re-ordered by inserting independent irredundant instructions wherever possible to ensure maximum benefits. For example, instruction (7) in Figure 4.6(b) is reordered to appear earlier in the final code depicted in Figure 4.6(c).

Programming the SASA Table. The individual entries of the SASA Table contain information about the registers identified as sparse and the corresponding redundant instruction sequences. These entries are stored in memory as read-only data during the initialization phase of the program (Step 5). Finally, the executable code is instrumented with appropriate *SASA-LD* instructions before the execution of the program region containing the redundant instruction sequences (Step 6).

Mapping sparse data-structures to vector processors. In vector processors, typically one of the input operands is shared by all SIMD lanes, while the other is different across lanes. A vector instruction can be skipped only if computations performed on all SIMD lanes are redundant. Hence, it is better to map a sparse data-structure as the shared input operand, since the likelihood of all non-shared inputs being zero is low. If both data-structures are sparse, then the data-structure that exhibits the most block-wise sparsity is mapped as the non-shared input operand.

We present more details about this step in the following subsection where we discuss the details behind executing BLAS based GEMM routines on SparCE.

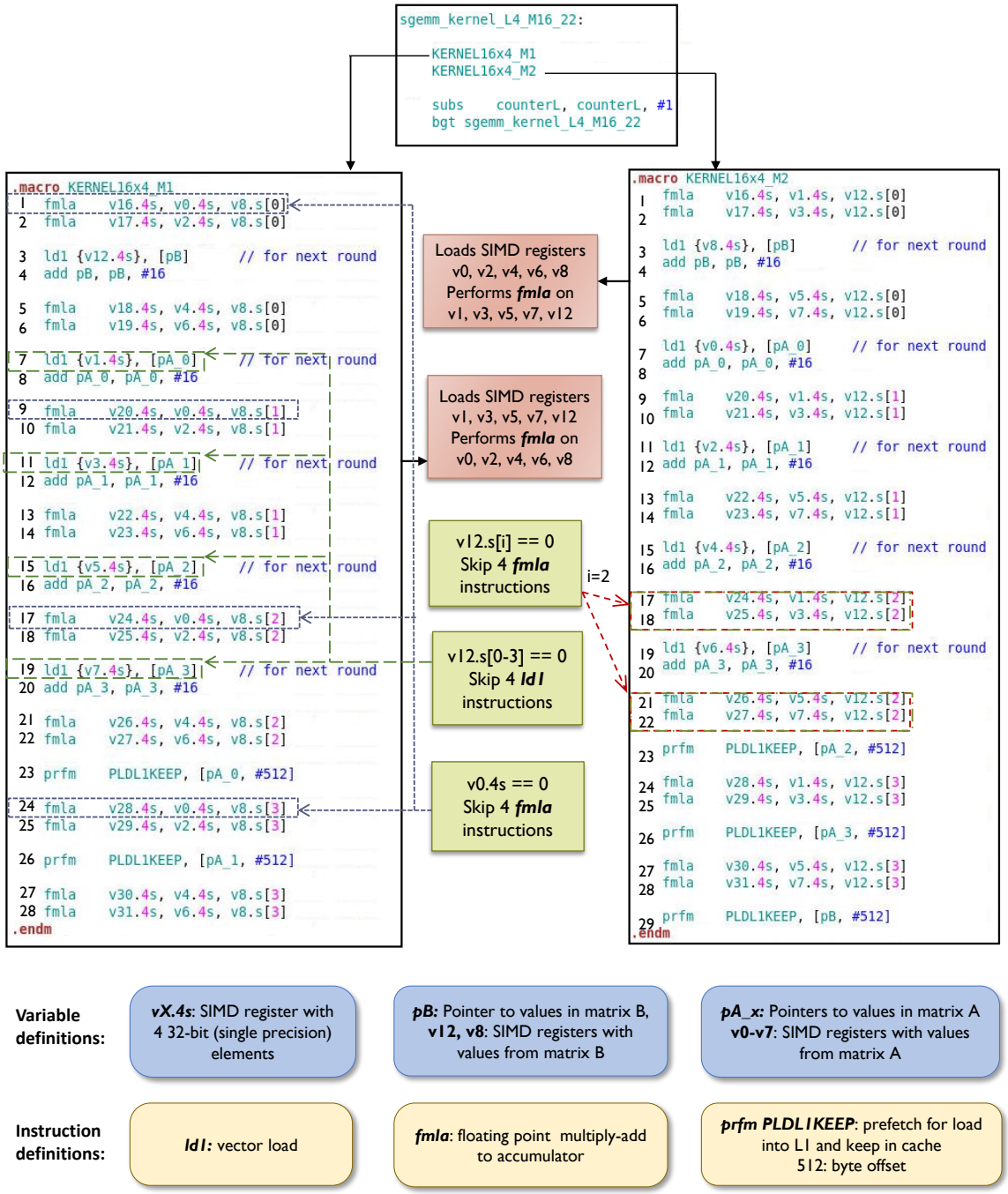

Fig. 4.7.: Zero skipping for *sgemm_kernel* subroutine in BLAS

4.2.2 Case Study: Executing GEMM Routine on SparCE

Popular deep learning frameworks, such as Caffe [112], TensorFlow [113], *etc.,* execute FFNNs as a series of matrix multiply operations, and leverage highly optimized software libraries such as BLAS (Basic Linear Algebra Subprograms) to realize them. Therefore, we evaluate the benefits of SparCE by executing the GEMM (Generalized Matrix Multiply) routine from the OpenBLAS library [114] for an ARM v8 processor with 4-width SIMD. The corresponding implementation is referred to as the OpenBLAS-SIMD4 implementation in Sections 4.3 and 4.4.

Figure 4.7 shows the original assembly program for the *sgemm_kernel_l1_M16_22* subroutine, which performs single-precision floating point matrix multiplication ($B \times A = C$). The *sgemm* routine executes two smaller subroutines *viz. kernel16x4_M1* (which we abbreviate as $M1$) and *kernel16x4_M2* ($M2$) sequentially in a loop. The subroutines utilize vector registers $v8$ and $v12$ to hold the first input operand (B) and registers $v0$-$v7$ to hold the second operand (A). The intermediate results are computed in registers $v16$-$v31$. To optimize performance, each subroutine prefetches data for the other. For example, $M1$ fetches data for operand A into the *odd* registers, which are then used by $M2$ in the subsequent iteration. The memory addresses are provided by scalar registers pB and pA_0-pA_3. The subroutines also prefetch data in the L1-cache using the *prfm* instruction.

We identify redundant instruction sequences in the *sgemm_kernel_l1_M16_22* subroutine assuming one of the input operands (say B) is sparse. The analysis can be easily extended to cover the scenario when either A or both A and B are sparse. Since B is sparse, it is beneficial to map it as the operand shared across the SIMD lanes, which is already the case in Figure 4.7. Even when one of the words in a vector register containing B ($v8$ or $v12$) is zero, 4 $fmla$ instructions can be skipped. For instance, when $v12.s[1]$ equals 0, instructions 9,10,13,14 in $M2$ can be skipped. This forms 2 redundant instruction sequences (9-10 and 13-14), each of size 2. This amounts to 16 $fmla$s being skipped when the entire vector register ($v8$ or $v12$) is

zero. It is worth noting that, had the sparse data-structure (B) been mapped as the non-shared SIMD operand in the program, only 4 $fmla$s could be skipped even when the entire vector register is zero.

In addition to the $fmla$ instructions being skipped, when the vector register is fully zero, the load instructions for the second operand can also be skipped. For example, when $v12$ is zero, $ld1$ instructions (7, 11, 15 and 19) for operand A can be skipped in $M1$. Also, note that we did not re-order any instruction in the *sgemm* routine, as the redundant instruction sequences were sufficiently spaced apart from the instruction that triggers their skipping. In the context of $fmla$ instructions, the vector loads happen in a different subroutine owing to pre-fetching, and in the case of the $ld1$ instructions, they were naturally spaced >3 instructions apart.

PrecedingPC	instsToSkip	SpRFcondition
Bgt inst. in sgemm	2	SpRF[v8[0]]
M1.4	2	SpRF[v8[0]]
M1.6	1	SpRF[v12]
M1.8	2	SpRF[v8[1]]
M1.10	1	SpRF[v12]
M1.12	2	SpRF[v8[1]]
M1.14	1	SpRF[v12]
M1.16	2	SpRF[v8[2]]
M1.18	1	SpRF[v12]
M1.20	2	SpRF[v8[2]]
M1.23	2	SpRF[v8[3]]
M1.26	2	SpRF[v8[3]]

Fig. 4.8.: SASA table entries for *kernel16x4_M1* subroutine

Based on the above analysis, Figure 4.8 shows the SASA table corresponding to the $M1$ subroutine. We find a total of 12 entries in the SASA table, 8 entries of size 2 for the 16 $fmla$s and 4 entries of size 1 for the $ld1$ instructions. The final GEMM routine executed on SparCE has this information about the contents of the SASA table built in it along with explicit SASA-LD instructions to load the table from memory.

SparCE **in action.** Figure 4.9 depicts the sequence of events that leads to instructions being skipped by SparCE. First, when register $v12$ is loaded by the $M1$ subroutine, its SpRF entry is updated. Note that the $isSparse$ field is a bit vector, one bit for each word in the vector register. Next, when instruction $M2.12$ is fetched, it sees a hit in the SASA table and the reads $SpRF[12]$ to ascertain if the bit corresponding to $SpRF[12][1]$ is sparse. Since that is the case, the PC is incremented to directly fetch $M2.15$.

In summary, with minimal changes to software, SparCE processors can leverage sparsity to improve performance.

4.3 Experimental Methodology

In this section, we present the methodology adopted in our experiments to evaluate SparCE.

4.3.1 Performance Evaluation

We modeled the micro-architectural extensions proposed in SparCE using the cycle-accurate *gem5* architectural simulator [115]. The gem5 simulator was tightly integrated with the popular Caffe [112] deep learning framework, wherein the matrices corresponding to each layer and each input batch was formed in Caffe and fed into the gem5 simulator to perform the matrix computations. The results were fed back to Caffe to form the inputs for the next layer (or input batch), and so on. Table 4.1(a) shows the gem5 system configuration used in our experiments. All experiments were

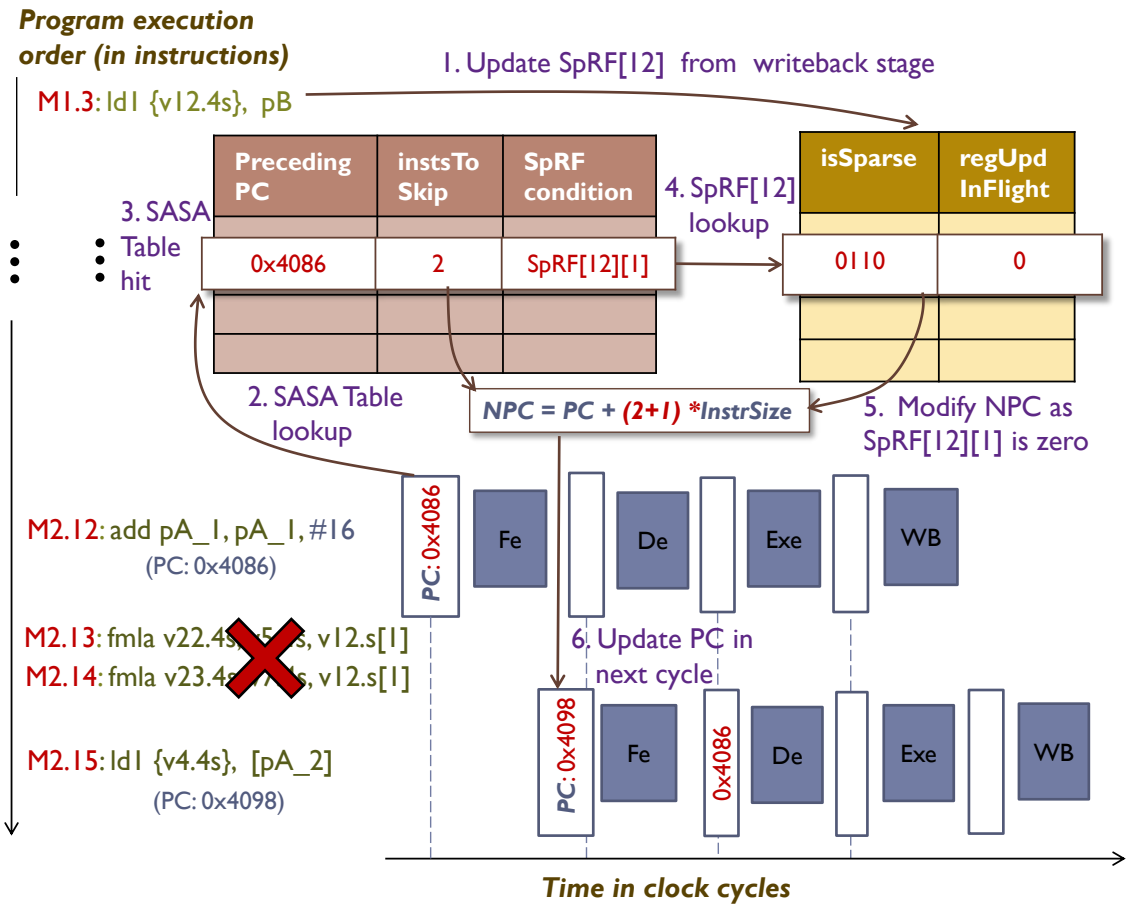

Fig. 4.9.: SparCE in action for *sgemm* routine

run in full-system mode. We evaluate SparCE by measuring application level execution times under two scenarios. The first scenario targets embedded scalar processors that are present in ultra-low power edge/IoT devices and lack support for high performance libraries. We chose a scalar ARM v8 in-order processor architecture as the baseline. We leverage the modular nature of gem5 to cater to this scenario, wherein we disable support for advanced architectural features such as SIMD processing and prefetching in the ARM v8 processor. We then prototyped a direct convolution routine (which we call *Dir-Conv-Scalar*) that does not utilize the disabled features and used it in our experiments. The second scenario targets a reasonably sophisticated mobile processor for which we chose the ARM v8 in-order processor architecture with 4-way

SIMD as the baseline. In this case, the matrix computations were realized using the highly-optimized OpenBLAS [114] based GEMM routines described in Section 4.2.2. We refer to this implementation as *OpenBLAS-SIMD4*.

4.3.2 Power and Area Evaluation

We implemented the hardware extensions of SparCE at the Register Transfer Level (RTL) using Verilog HDL and synthesized them to IBM 45nm technology using Synopsys Design Compiler to measure its power and area overheads. For the configuration shown in Table 4.1(a), the area overhead is 1.04% of the ARM Cortex A35 core [111]. Thus the area overhead of SparCE is quite minimal allowing its deployment in the resource-constrained embedded platforms.

Table 4.1.: (a) Gem5 simulation parameters (b) Application benchmarks

Processor config.	ARMv8-A, In-order
SparCE config.	20 SASA table entries, 32 SpRF entries
L1 Cache	Split I&D, 32KB I cache, 64KB D cache, 2-way set associative I & D cache, 64B line, 3-cycles/access
L2 Cache	Unified 2MB 8-way set associative, 64B line, 12 cycles/ access

(a)

Benchmark	Dataset	# Layers	#Ops
CIFAR-10	CIFAR-10	5	0.01B
AlexNet	ImageNet	8	0.72B
VGG-16	ImageNet	16	15.4B
ResNet-50	ImageNet	50	3.86B
GoogleNet	ImageNet	22	1.59 B
DeepComp.	ImageNet	8	0.72B

(b)

4.3.3 Benchmarks

As CNNs are currently the most widely used FFNNs, we evaluate the benefits of SparCE on CNNs. Our benchmark suite, listed in Table 4.1(b), consists of 6 state-of-the-art image-recognition CNNs *viz.* CIFAR-Caffe CNN using the CIFAR-10 dataset [116], and AlexNet [117], VGG-16 [110], ResNet-50 [1], GoogleNet [2] and Deep Compression-AlexNet [43] using the ImageNet dataset. These benchmarks contained 5-50 layers and took 0.01-15.4 Billion scalar operations to classify an image. We utilized pre-trained models from the Caffe Model Zoo to evaluate SparCE in the context of inference. For training, we utilized only the smaller CIFAR-10 benchmark, as training ImageNet models on gem5 was prohibitively time consuming. It is noteworthy that all benchmarks exhibited dynamic sparsity in features and errors, while only Deep Compression-AlexNet exhibited static sparsity in weights.

4.4 Results

In this section, we present the results of our experiments that highlight the advantages of SparCE.

4.4.1 Performance and Energy Improvement

Figure 4.10 shows the normalized execution time benefits of SparCE over the baseline processor for both inference and training. In the context of Dir-Conv-Scalar, the reduction in *application runtime* ranges between 19%-31% across the benchmarks. In contrast, OpenBLAS-SIMD4 demonstrates benefits in the range of 8%-15% reduction in runtime. This is because $fmla$ instructions occupy a much smaller fraction of their runtime, as their execution engines are more sophisticated - multiple SIMD lanes, low floating point instruction latency *etc.* Also, since they support features such as prefetching where the data is already fetched into the higher levels of the

memory subsystem, avoiding redundant data fetches has a less prominent impact on performance.

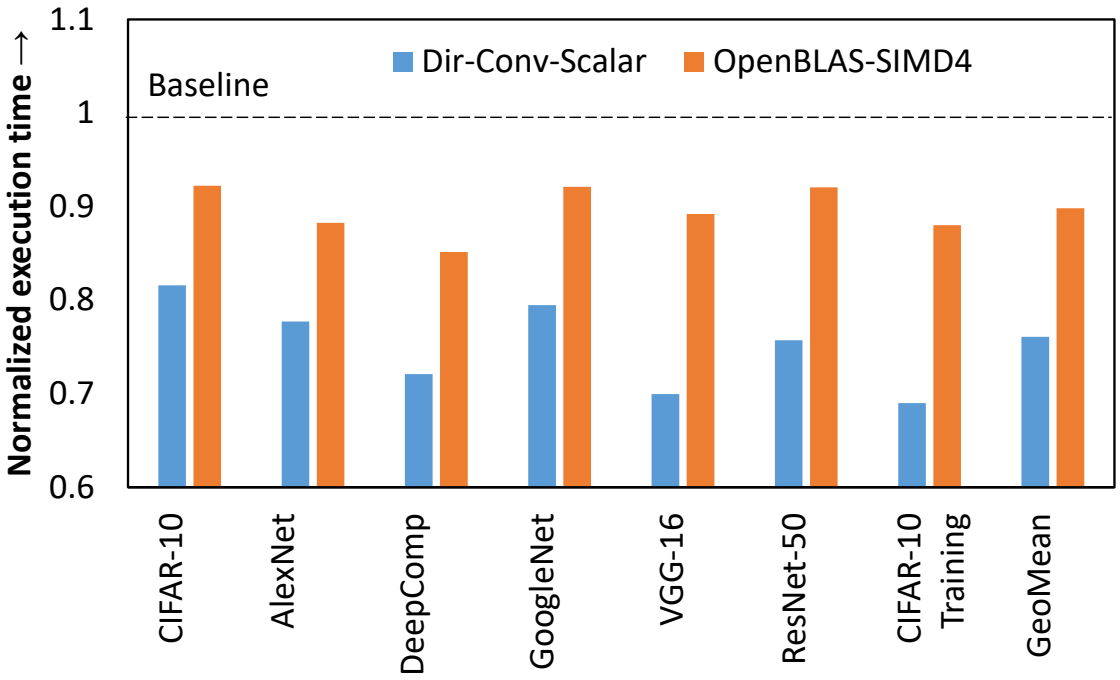

Fig. 4.10.: Improvement in execution time at the application level

Among the benchmarks, the execution time benefits are largely proportional to the amount of sparsity that they exhibit (Figure 3.1 in Section 3.1). The Deep Compression-AlexNet benchmark achieves the most benefits because both its activation and weight data-structures are sparse, as opposed to other benchmarks whose weight data-structure is dense. In the context of training, the backpropagation step achieves more improvement compared to forward propagation. This stems from the fact that the error data-structure is more sparse compared to activations.

Execution Time Breakdown. To better appreciate the improvements achieved by SparCE, Figure 4.11 hierarchically breaks down the application execution time (in the context of both Dir-Conv-Scalar and OpenBlas-SIMD4 implementations) into components that can and cannot be impacted by SparCE. At the top level, the solid yellow and gray colors represent the execution time fraction that cannot be

improved by SparCE. This is primarily constituted by auxiliary CNN operations such as activation functions, subsampling and others, which represent 1.9% and 12.2% of the runtime for Dir-Conv-Scalar and OpenBlas-SIMD4 implementations, respectively. It is noteworthy that although these operations occupy <1% of the total CNN FLOPs, they occupy a substantially larger fraction of the runtime for the OpenBLAS-SIMD4 implementation. This is owed to the fact that they are typically memory-bound (higher Bytes/FLOP ratio), which is further amplified as matrix multiply operations are significantly optimized by the GEMM subroutine. Also, since the inputs to the CNN are typically dense, the first CNN layer exhibits little redundancy. This occupies 14.3% and 16.9% of the total runtimes of AlexNet for Dir-Conv-Scalar and OpenBLAS-SIMD4 implementations, respectively. The fraction grows smaller in deeper networks such as ResNet and VGG.

In Figure 4.11 the green color bars denote the computations that can accelerated by leveraging sparsity (∼71%). For Dir-Conv-Scalar implementation, this is limited to 83.6% of the baseline AlexNet runtime. Since AlexNet contains ∼36% redundant computations (Figure 3.3 (b) in Section 3.1), *the best case benefits are limited to 29.8%*, of which SparCE achieves 22.3%. For the OpenBLAS-SIMD4 implementation, the underlying GEMM involves supplementary operations like memory allocate, copy and free operations, which as marked by the vertically hatched regions consume 27% of the total execution time. This constraints the opportunity for SparCE to ∼44% of AlexNet runtime as shown by the diagonally hatched portions in Figure 4.11. Since AlexNet contains ∼36% redundant computations (Figure 3.3 (b) in Section 3.1), *the best case benefits are limited to ∼16%*, of which SparCE achieves 12% improvement as other control operations such as pointer arithmetic, prefetching *etc.* present within the loop body cannot be avoided.

Layer-wise Benefits. We now present the layer-wise breakdown of the benefits quantified in terms of the execution time, instructions and data cache (D-Cache) accesses skipped for the convolutional layers of AlexNet. Figure 4.12 shows the benefits achieved with SparCE in the context of both SIMD and scalar processor implemen-

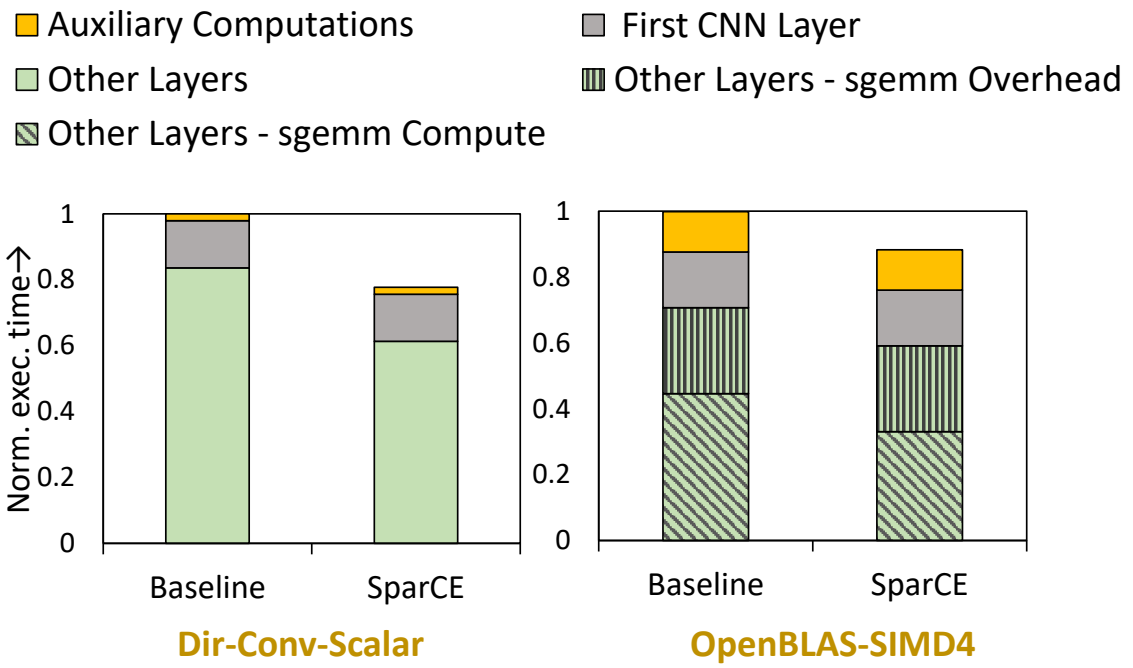

Fig. 4.11.: Execution time breakdown for AlexNet

tations. The values have been normalized with respect to the execution time, instructions and data cache accesses observed in the baseline. As shown in Figure 4.12, we are able to achieve, on an average, 39.4% reduction in instruction count and 35.1% reduction in D-Cache accesses for Dir-Conv implementations on low-power embedded scalar processors. The reduction in instruction count and D-Cache accesses amount to 30.5% and 14% respectively for OpenBLAS-SIMD4 implementations. We also observe the benefits are typically larger for layers deeper in the CNN, as they typically exhibit more sparsity.

Energy Benefits. Power evaluation of the SparCE (Section 4.3.2) reveals that it consumes 1.74 mW at 1 GHz, which amounts to 1.9% of the 90 mW power consumed by even the most power-efficient baseline ARM v8 processor, the Cortex A35 processor [111]. Accordingly, the execution time benefits translate to benefits in the range of 16.9%-28.7% reduction in application-level energy for a Dir-Conv-Scalar im-

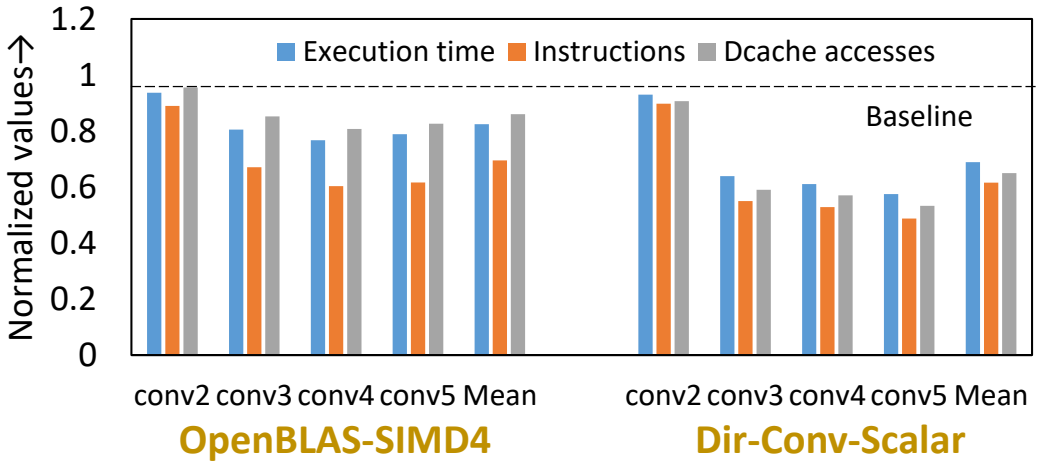

Fig. 4.12.: Layer-wise benefits breakdown for AlexNet

plementation. In the context of OpenBLAS-SIMD4 implementations, the reduction in energy ranges between 6.1%-13.2% across the benchmarks.

4.4.2 Performance Scaling with Sparsity

We now study how the performance of SparCE scales with increasing levels of sparsity. To this end, we consider the implementation of *conv4* layer (4th convolutional layer) of AlexNet which convolves input activations of size 384×13×13 with kernels of size 384×384×3×3 to produce output activations of size 384×13×13. We varied the sparsity of input activations by constraining the number of zero entries. The location of the zeros and other entries of the matrices were chosen at random. Figure 4.13 shows how the execution time and the fraction of instructions executed varies with sparsity in the context of both Dir-Conv-Scalar and OpenBLAS-SIMD4 implementations. The values have been normalized with respect to the those observed in the baseline processor without SparCE extensions. We find that both implementations exhibit strong performance scaling with sparsity, outlining the ability of SparCE to efficiently skip computations. We find the number of instructions executed to be larger than ideal (dotted line in Figure 4.13) due to the presence of control

instructions for pointer arithmetic, loop counts *etc.,* in the program, which cannot be skipped. Also, the disparity in the fraction of instructions executed and the resultant execution time benefits is more pronounced for the OpenBLAS-SIMD4 implementation. We attribute this to the intelligent instruction ordering in the GEMM routine utilized in the implementation, wherein computations are aggressively overlapped with data-fetches. Therefore, even if computations are skipped, the improvement in performance is limited by the time taken for the data-fetches. The performance improvements can potentially be boosted with advanced compiler optimizations capable of generating differently ordered GEMM instructions for data-structures with different amounts of sparsity. However, in this work, we have focused on minimal software changes. Another important property that can be observed in Figure 4.13 is that SparCE does not cause any slowdown at 0% sparsity in both OpenBLAS-SIMD4 and Dir-Conv-Scalar implementations. This can be attributed to the fact that the overhead associated with programming the SASA Table is negligible when compared to the total number of computations.

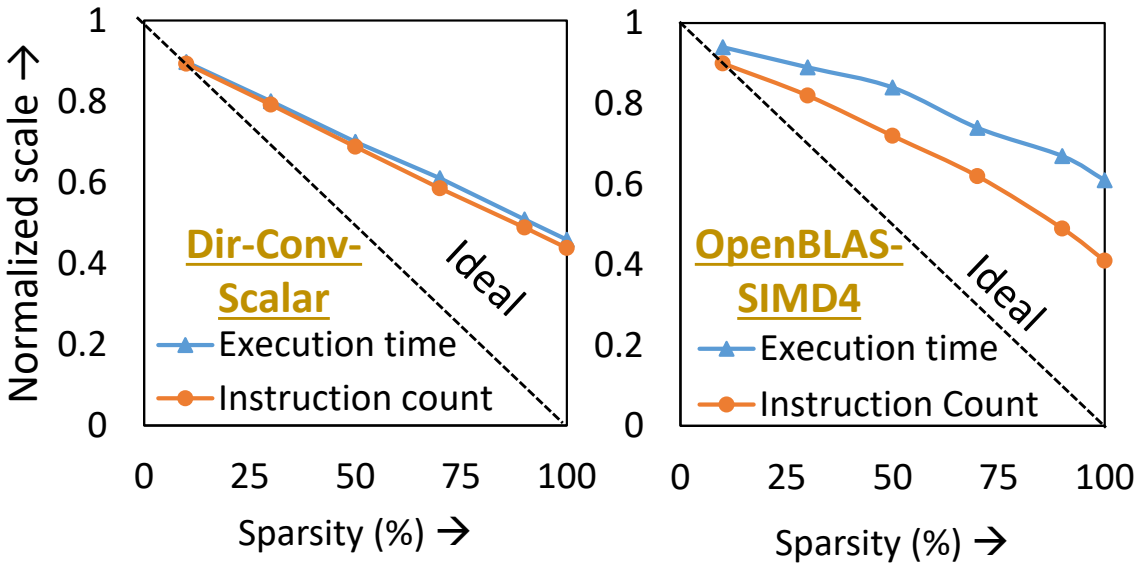

Fig. 4.13.: SparCE performance scaling with sparsity

4.4.3 Operand Ordering in SparCE OpenBLAS-SIMD4 Implementations

As described in Section 4.2, based on the amount of sparsity exhibited by the data-structures, mapping the right data-structure as the shared-SIMD operand can have a considerable impact on performance. In the case of all benchmarks other than Deep Compression-AlexNet, only the activation data-structure is sparse. Therefore, mapping it as the shared-SIMD operand (Matrix B in the *sgemm* subroutine) would yield the best benefits.

Figure 4.14 shows the performance improvement achieved when operands are ordered in both ways *viz.* Activations×Weights and Weights×Activations. In the context of AlexNet, we find that mapping activations as the shared-SIMD operand yields 1.86× better benefits (∼12% *vs.* 6.5%) compared to mapping the non-sparse weight

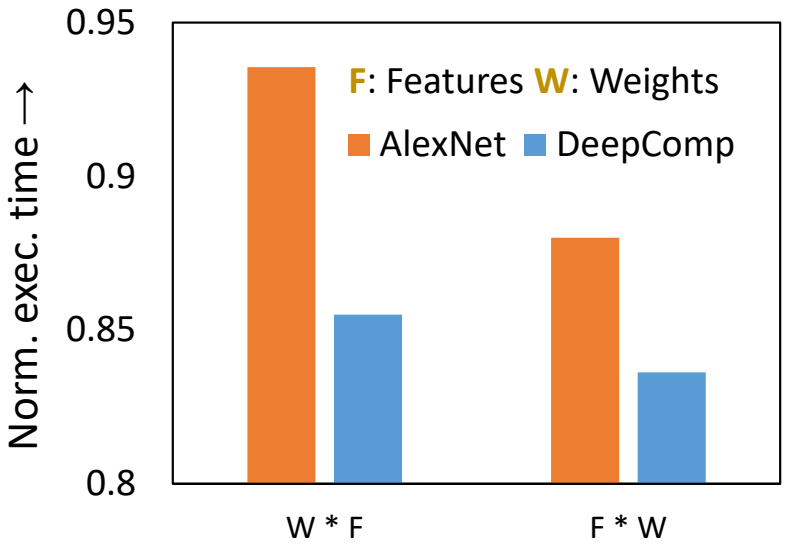

Fig. 4.14.: Impact of operand ordering on performance

data-structure. For the Deep Compression-AlexNet network, since both activation and weight data-structures are sparse, the disparity in performance due to operand ordering is relatively small (<2%). Even in this case, we find that choosing activations as the shared-SIMD operand is beneficial. This is attributed to the fact that some of the weight matrices have high degree of sparsity, and their zero entries are

typically clustered. Therefore, in this case, using weights as the non-shared SIMD operand has less of an adverse impact on performance compared to using activations.

Overall, the programmer is responsible for making the choice of which sparse data-structure should be mapped as a non-shared SIMD operand. For applications with both data-structures as sparse, higher benefits are obtained when the operand exhibiting a higher sparsity value is chosen as the non-shared SIMD operand. In contrast, for applications with only one sparse data-structure, mapping the same as the shared SIMD operand yields higher benefits.

4.5 Summary

As FFNNs pervade the spectrum of computing devices, new approaches to improve their computational efficiency on resource-constrained IoT/ edge devices becomes critical. In this work, we accelerate FFNNs on GPPs, which are an indispensable part of IoT/ edge devices, by exploiting sparsity in the different FFNN data-structures. To this end, we propose Sparsity-aware general purpose Core Extensions (SparCE) that enable GPPs to efficiently leverage sparsity, while being minimally intrusive and low-overhead. SparCE comprises of two key micro-architectural enhancements. First, a Sparsity Register File (SpRF) dynamically tracks zero-valued registers in the processor. Next, a Sparsity-Aware Skip Address (SASA) Table indicates potentially redundant instruction sequences and the conditions under which they can be skipped. A Pre-identify and Skip Redundancy Unit (PSRU) combines the information from the SpRF and the SASA table to dynamically *pre-identify* if an instruction sequence can be skipped, and if so masks it from being fetched and executed. We take advantage of these micro-architectural extensions by suitably modifying the code generation process to allow successful identification of redundant instruction sequences and subsequent programming of the SASA Table. We evaluate SparCE on 6 image-recognition FFNNs in the context of both training and inference. Our evaluations

reveal that SparCE is a promising design that allows us to exploit all forms of static and dynamic sparsity to accelerate FFNNs on GPPs.

5. APPROXIMATE COMPUTING FOR LONG SHORT TERM MEMORY (LSTM) NEURAL NETWORKS

As discussed in Chapter 1, DNNs can broadly be classified into feed-forward neural networks and recurrent neural networks. Feed-forward networks like Convolutional Neural Networks (CNNs) and Multi-Layer Perceptrons (MLPs) are suitable for tasks with fixed-length, temporally independent inputs and outputs, such as image classification. These networks lack any feedback connections and are characterized by a strict forward flow of information from the input layer to the output layer.

Recurrent Neural Networks (RNNs), on the other hand, can handle variable-length inputs and outputs that form sequences. Text and handwriting synthesis [56], speech recognition [57], neural machine translation [58] and image and video captioning [3] are a few examples of applications that deploy RNNs. These networks have a cyclic structure, allowing information to persist temporally in the form of memory in the network. The computation of an RNN can be thought of as proceeding in timesteps with a new element of the input sequence being fed to the network at each timestep. Recent interest in RNNs has been largely fueled by the success of Long Short Term Memory networks (LSTMs) [91]. LSTMs and their derivatives have demonstrated an ability to learn long-term dependencies in sequences and are currently deployed in numerous real-world applications. An LSTM comprises of cells, each of which is associated with a cell state and multiple gate units that control the flow of information in and out of the cell.

State-of-the-art LSTMs are highly compute-intensive. For example, Neural Machine Translation (NMT) of the WMT development set From English to French, using a quantized version of Google's NMT, takes 1322 seconds on a pair of Intel Haswell CPUs [20]. Previous efforts to improve the execution efficiency of LSTMs have explored three distinct directions. The first direction is to efficiently parallelize LSTMs

on CPUs and GPUs [19–21]. The second direction develops accelerators that exploit the data access and compute patterns of LSTMs [24, 27–30]. Finally, the last set of efforts applies model pruning and quantization to reduce the model size for execution on resource-constrained platforms [45, 67, 89].

We explore *approximate computing* techniques to accelerate the execution of LSTMs. The intrinsic error resilience of machine learning applications makes them excellent candidates for approximate computing, which attempts to reduce execution time and energy with minimal effect on the quality of outputs. Previous efforts have proposed approximate computing techniques for neural networks by utilizing variable bit precision and approximate arithmetic units [37, 48, 49, 86]. Other works have also proposed approximate computing techniques for biologically inspired spiking neural networks [118]. However, these techniques provide benefits only on specialized hardware architectures or do not specifically exploit the unique characteristics of LSTMs. To the best of our knowledge, we are the first to propose hardware-agnostic approximate computing techniques for LSTMs.

We present AxLSTM, a general approach to approximate LSTMs [1]. AxLSTM consists of two techniques — *Dynamic Timestep Skipping (DTS)* and *Dynamic State Reduction (DSR)*. DTS is based on the observation that some elements of the input sequence have little or no impact on the state of the LSTM (and hence the output produced). Therefore, identifying such unimportant elements of the input sequence allows the LSTM to skip evaluating them at runtime. On the other hand, DSR is based on the observation that, since the LSTM state is provisioned to capture all possible input sequences, many inputs sequences do not require the full dimensionality of the LSTM state. Thus, the computational complexity of an LSTM can be reduced by dynamically reducing the size of the LSTM state based on the input sequence.

[1] Although we focus on LSTMs due to their widespread use, the proposed techniques are applicable to any RNN.

In summary, the key contributions of this work are:

- We propose AxLSTM, an approximate computing framework for LSTMs, which combines *Dynamic Timestep Skipping (DTS)* and *Dynamic State Reduction (DSR)*.
- We develop heuristics to dynamically modulate the number of timesteps in DTS, as well as the number of computations per timestep in DSR, based on the input sequence.
- We implement AxLSTM within the TensorFlow deep learning framework and apply it to state-of-the-art sequence-to-sequence models for neural machine translation and video caption generation. We achieve speedups of 1.08× - 1.31× with minimal loss in quality, and 1.12× - 1.37× when moderate reductions in quality are acceptable.

The rest of the chapter is organized as follows. Section 5.1 presents the AxLSTM approach and attendant details, followed by the experimental methodology in Section 5.2. Section 5.3 presents the results of our experiments. Section 5.4 concludes the chapter.

5.1 AxLSTM: Design Approach and Methodology

To improve the execution efficiency of LSTMs, we propose AxLSTM, a set of approximate computing techniques that exploit the key characteristics of LSTMs. In this section, we present the salient features of AxLSTM and describe its details in the context of sequence-to-sequence models.

5.1.1 AxLSTM: Overview

Figure 8.2 outlines the approximation strategies adopted by AxLSTM and the targeted benefits. AxLSTM consists of two techniques — Dynamic Timestep Skipping (DTS) and Dynamic State Reduction (DSR). The motivation behind Dynamic

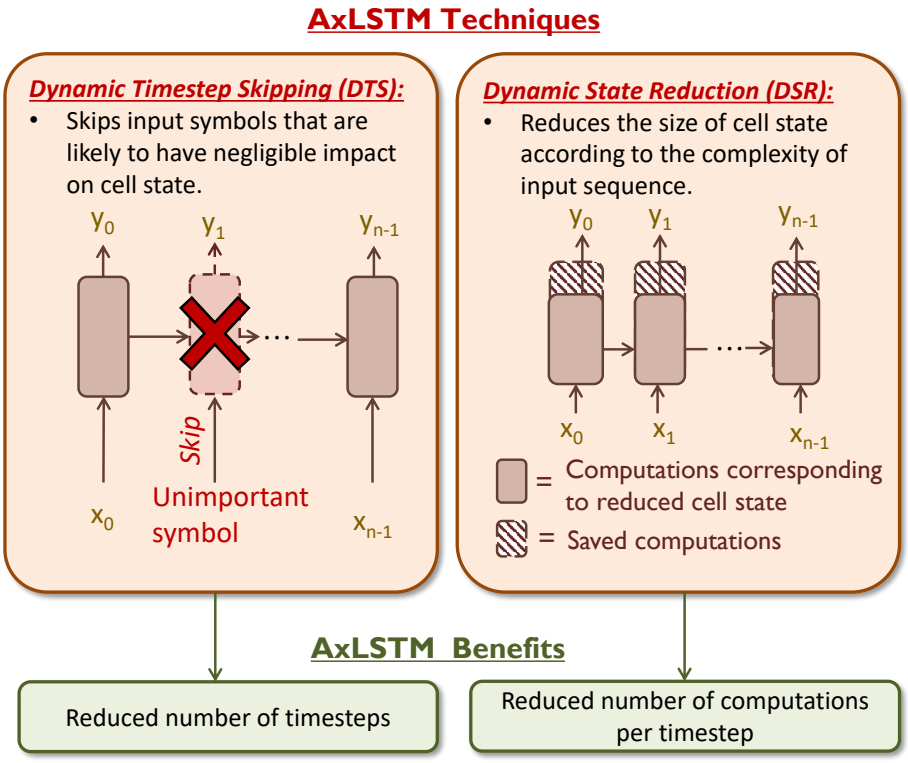

Fig. 5.1.: Overview of AxLSTM approximation strategies

Timestep Skipping is the fact that all symbols in an input sequence do not have an equal impact on the LSTM state. Consequently, the LSTM can skip evaluating symbols that are likely to have negligible effects on its state and thereby save execution time. For example, not all frames in a video sequence are equally important for an LSTM to generate an appropriate caption for the video. Similarly, some words of a sentence may turn out to be redundant in machine translation. Skipping the evaluation of any unimportant frame (or word) can help save one entire timestep worth of computation, which involves multiple matrix multiplications for calculating the different gates of an LSTM.

In contrast, Dynamic State Reduction is based on the principle that all input sequences are not semantically complex enough to require the entire dimensionality of the cell state. As a result, the state size can be dynamically modulated at runtime. A smaller state size reduces the sizes of matrices and thereby, the complexity of matrix

multiplications involved in calculating the different gates of an LSTM. This, in turn, significantly decreases the time spent in evaluating each timestep of an LSTM.

In summary, AxLSTM reduces the execution time of LSTMs by taking advantage of the two techniques, DTS and DSR, to reduce the number of timesteps as well as the number of computations per timestep.

5.1.2 Dynamic Timestep Skipping (DTS)

The key challenge involved in DTS lies in developing an accurate, yet low overhead mechanism to identify unimportant symbols in an input sequence. To ensure savings, the time expended in evaluating this dynamic mechanism should be significantly less than that expended towards evaluating the LSTM itself.

We propose to augment an LSTM with a new structure, the *Input Analyzer (IA)*. The IA acts as a filter to the LSTM and feeds only important input symbols of a sequence to it. It substitutes all unimportant symbols with a special symbol called the *Skip Symbol (SS)*. Unlike other input symbols, the LSTM doesn't perform any computations on the Skip Symbol. Instead, it simply moves on to processing the next timestep. The application of DTS to the encoder of a sequence-to-sequence model is shown in Figure 5.2. In this case, DTS targets the encoding time and reduces the effective $InputSeqLen$ observed by the encoder.

We propose two heuristics for the IA to dynamically identify unimportant input symbols. These heuristics, which are named *StateEffect* and *InputDiff*, are illustrated in Figure 5.2. The StateEffect heuristic utilizes knowledge of each input symbol's expected or average effect on the encoder state. This effect is quantified as the L2 norm of the difference between the encoder state before and after processing a symbol. We rank the input symbols in increasing order of their effect on the encoder state for all examples in the training set. Next, we skip each symbol and observe the effect on overall quality. Finally, we program the IA to store a list of input symbols that can be skipped for a given quality requirement. The size of the corresponding

StateEffect table is determined by the number of available quality levels, the number of insignificant symbols per quality level and the encoding size for each symbol. In our evaluations, we observed that a table of size 12 kB is sufficient.

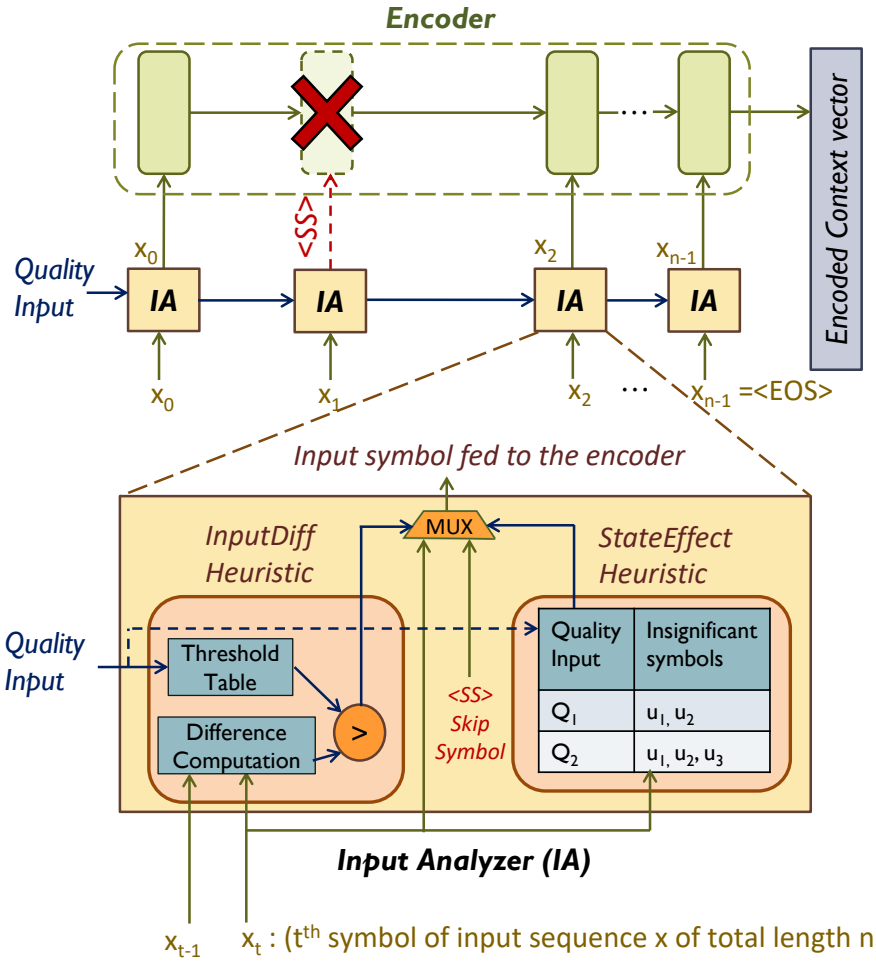

Fig. 5.2.: Dynamic Timestep Skipping in sequence-to-sequence models

The InputDiff heuristic is based on the fact that an input symbol that is very similar to the previous input symbol does not convey any new information to the encoder. Accordingly, skipping it's evaluation will minimally affect the overall output quality. An IA executing the InputDiff heuristic dynamically measures the difference between two successive input symbols. It is programmed to have a *Threshold Table* containing *DiffThres* values corresponding to each quality requirement. The IA decides to skip

an input symbol if its difference from the previous symbol is less than DiffThres. In general, the suitability of the StateEffect or the InputDiff heuristic can vary across different applications and the IA can be programmed to execute either or both of them. The values in the Threshold Table are determined using the methodology described in 5.1.4.

In summary, by using the IA, the encoder of a sequence-to-sequence model is able to dynamically identify unimportant symbols of an input sequence and skip processing them, thereby reducing the execution time.

5.1.3 Dynamic State Reduction (DSR)

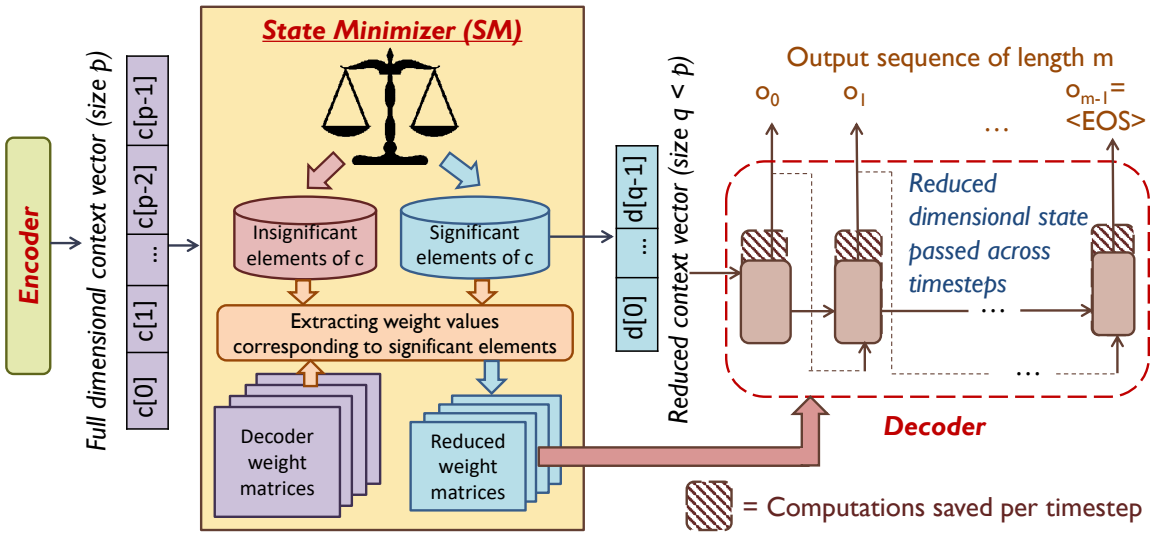

Fig. 5.3.: Dynamic State reduction in sequence-to-sequence models

In Dynamic State Reduction, the original full dimensional state is converted to a smaller state by identifying and retaining only the significant state elements for a given input sequence. Correspondingly, the original weight matrices are sliced into smaller matrices by extracting only the values corresponding to the significant state elements.

In the context of sequence-to-sequence models, we apply DSR to the decoder and successfully reduce the contribution of $ComputeTimePerOutputSymbol$ in Equation 3.3. As shown in Figure 5.3, we achieve this by adding a new structure called the State Minimizer (SM). In general, the SM can be invoked anywhere during the encoding or decoding process. However, invoking the SM at an intermediate timestep of the encoder and limiting the size of the encoder state size thereafter may lead to significant error since the subsequent input symbols have not yet been reflected in the encoder state. In addition, determining significant elements too frequently using the SM can turn out to be counter-productive due to the overhead of the significant index identification and matrix slicing steps. In order to avoid these challenges in our implementation, we invoke the SM only once at the well-defined boundary between the encoder and decoder network to identify a fixed set of significant indices before the decoding process starts. Specifically, the size of the hidden state in the encoder stays at the maximum value throughout the encoding process, producing a complete context vector. The SM analyzes the context vector and strips it down to a smaller vector containing only state elements that are deemed to be significant. The decoder weight matrices are also sliced by the SM based on the indices of the significant elements of the context vector. Each timestep of the decoder consumes and produces a context vector of this reduced size.

The SM examines the content of the encoded context vector to determine the significant elements. Two heuristics for identifying these significant elements are described below.

- **Relative thresholding:** The relative thresholding approach discards elements that are numerically far less than the maximum value in the context vector. For a given threshold $relThres$, an element i of the context vector C is identified to be insignificant if

$$C[i] < relThres \times max_i C[i] \tag{5.1}$$

- **Absolute thresholding:** The absolute thresholding approach discards elements whose magnitude is less than an absolute constant. For a given threshold of $absThres$, an element i of the context vector C is identified to be insignificant if

$$|C[i]| < absThres \tag{5.2}$$

The $relThres$ and $absThres$ values in both criteria play an important role in achieving a favourable quality versus savings tradeoff. Lower values of $relThres$ and $absThres$ lead to smaller states as most of the context vector elements are discarded by the SM. However, this also leads to a potential loss of useful information and subsequently, a drop in quality. For a given user-defined quality constraint, the threshold values are determined using the methodology described in 5.1.4 and are programmed into the SM before the AxLSTM model is deployed for inference.

For sequence-to-sequence models that perform inference after batching multiple input sequences, the SM adopts a conservative approach by forming a union of significant elements across all input sequences in a batch. This leads to some amount of inefficiency as the final state size, determined by the number of significant elements, can eventually be higher than that required for a single sequence. However, batching of input sequences also provides an advantage by ensuring that the overhead of slicing states and weights is amortized over multiple sentences in the batch. A large number of state-of-the-art sequence-to-sequence models have multiple layers in the encoder and decoder. The SM is sensitive to the difference in the behavior of different layers and accordingly outputs different sets of important indices for different layers.

In summary, the SM equips an LSTM with the ability to reduce the state size dynamically at runtime based on the complexity of the input sequence. This leads to a reduction in the amount of computation per timestep.

5.1.4 AxLSTM: Design Methodology

Algorithm 1 presents a method to automatically design sequence-to-sequence models of specified quality levels with AxLSTM. For brevity of discussion, we restrict ourselves to the StateEffect heuristic of DTS and the absThres criterion for DSR. However, the method can be applied to other heuristics and criteria with minor modifications. The inputs to the method are a sequence-to-sequence model trained with conventional LSTMs ($Seq2Seq$), the training dataset ($TrainData$), the list of input symbols in the training dataset ($InputSym$) and the quality constraint (Q). The output is the approximate version of the Seq2Seq model ($AxSeq2Seq$), the list of unimportant input sequence symbols for DTS (DTS_{list}) and the DSR threshold for identifying unimportant elements of the context vector C for layer Li (DSR^{Li}_{thres}). The algorithm builds $AxSeq2Seq$ by successively adding more input symbols to DTS_{list}, reducing DSR^{Li}_{thres} and retraining $AxSeq2Seq$ until the quality drops below the specified threshold.

We first measure the average effect on encoder state for all input symbols in the training dataset (lines 1-2) by calculating the L2 difference between the state vector before and after processing the symbol. Next, we sort the input symbols in order of their increasing effect on state (line 3) and store them in $SortedSym$. The DSR_{thres} for each layer is initialized to the maximum absolute value of the corresponding context vector (lines 4-6). The DTS_{list} is initialized to be a null list (line 7) and the weight and bias values of $AxSeqtoSeq$ are initialized to be equal to that of the original $Seq2Seq$. At each iteration of the algorithm, new symbols are copied from the $SortedSym$ list to DTS_{list} (line 11). The DSR_{thres} values for each layer of the network are reduced by a small constant Δ (line 13). Finally, the network is retrained for T training iterations to further improve its quality level (line 14). This process is stopped when the quality level $Q_{AxSeq2Seq}$ of the $AxSeqtoSeq$ drops below the specified quality level Q.

Algorithm 1 Designing sequence-to-sequence models with AxLSTM

Input: $Seq2Seq$: Trained sequence-to-sequence model,

$TrainData$: Training dataset,

$InputSym$: List of input sequence symbols,

Q: Quality constraint

Output: $AxSeq2Seq$: Approximate sequence-to-sequence model,

DTS_{list}: List of unimportant input sequence symbols,

DSR_{thres}^{Li}: Threshold for identifying unimportant elements of context vector C for layer Li

1: **For** each symbol $S \in InputSym$:
2: Compute average effect on encoder state, $Effect[S]$
3: Sort $InputSym$ in increasing order of $Effect$, $SortedSym$
4: **For** each Layer Li:
5: Compute Max. Abs(Context vector C), C_{max}^{Li}
6: Initialize $DSR_{thres}^{Li} = C_{max}^{Li}$
7: $DTS_{list} = \emptyset$
8: $AxSeq2Seq = Seq2Seq$
9: $numSkipped = 0$
10: **while** (1) **do**
11: $DTS_{list} = DTS_{list} \cup SortedSym[numSkipped]$
12: **For** each Layer Li:
13: Set DSR_{thres}^{Li} -= Δ
14: $AxSeq2Seq$ = Retrain($AxSeq2Seq$, DTS_{list}, DSR_{thres}, $TrainData$, T training iterations)
15: **if**($Q_{AxSeq2Seq} < Q$) **break**
16: $numSkipped = numSkipped+1$
17: **end while**
18: **return** $AxSeq2Seq$, DTS_{list} **and** DSR_{thres}^{Li}

In summary, by selectively skipping unimportant symbols in the encoder and reducing the decoder state size, AxLSTM improves the execution efficiency of LSTMs.

5.2 Experimental Methodology

In this section, we present the methodology utilized in our experiments to evaluate AxLSTM.

5.2.1 Performance Evaluation

We implemented AxLSTM within TensorFlow [113], a popular deep learning framework. The files used by the Python API of TensorFlow were modified to incorporate the DTS and DSR techniques. We evaluated the performance benefits of AxLSTM in software by measuring application level runtimes with and without the proposed techniques on a 2.7GHz Intel Xeon server with 128GB memory and 32 processor cores.

Table 5.1.: Application benchmarks

Model	**NMT1**	**NMT2**	**S2VT**
Task	Neural Machine Translation	Neural Machine Translation	Video Captioning
Dataset	IWSLT15	WMT16	Youtube Corpus (MSVD)
Input Sequence	English sentence	German sentence	Video
Output Sequence	Vietnamese sentence	English sentence	English sentence
No. of layers	2	4	2
LSTM state size	512	1024	256

5.2.2 Application benchmarks

Our benchmark suite, listed in Table 5.1, consists of three sequence-to-sequence models with applications in Neural Machine Translation (NMT) and video captioning. These models vary considerably in the nature of their input sequences, output sequences and their computational complexity, as indicated by the number of layers and LSTM state size. The two NMT models are based on the models available in [119]. The first NMT model performs English-Vietnamese translation and is trained on a parallel corpus of TED talks (133K sentence pairs) provided by the IWSLT Evaluation Campaign. The next NMT model performs German-English translation and is trained on the German-English parallel corpus (4.5M sentence pairs) provided by the WMT Evaluation Campaign. On the other hand, the video captioning model is based on [3], trained on the Microsoft Video Description Corpus [120] and used to generate captions on Youtube clips collected on Amazon Mechanical Turk. We utilize BLEU scores, a method for automatically evaluating machine translations, as a metric to evaluate application quality. The baseline implementations were trained on the training datasets using conventional LSTM networks, whereas the AxLSTM implementations were obtained using the methodology mentioned in Section 5.1.4 with the DTS and DSR techniques in place. The application level runtimes and BLEU scores of both implementations were measured on a separate validation dataset.

5.3 Results

In this section, we present the results of our experiments that evaluate the benefits of AxLSTM, and analyze the sources of these benefits.

5.3.1 Performance Benefits Versus Accuracy

Figure 5.4(a) shows the normalized execution time benefits achieved by AxLSTM across all benchmarks. For each benchmark, models with different quality lev-

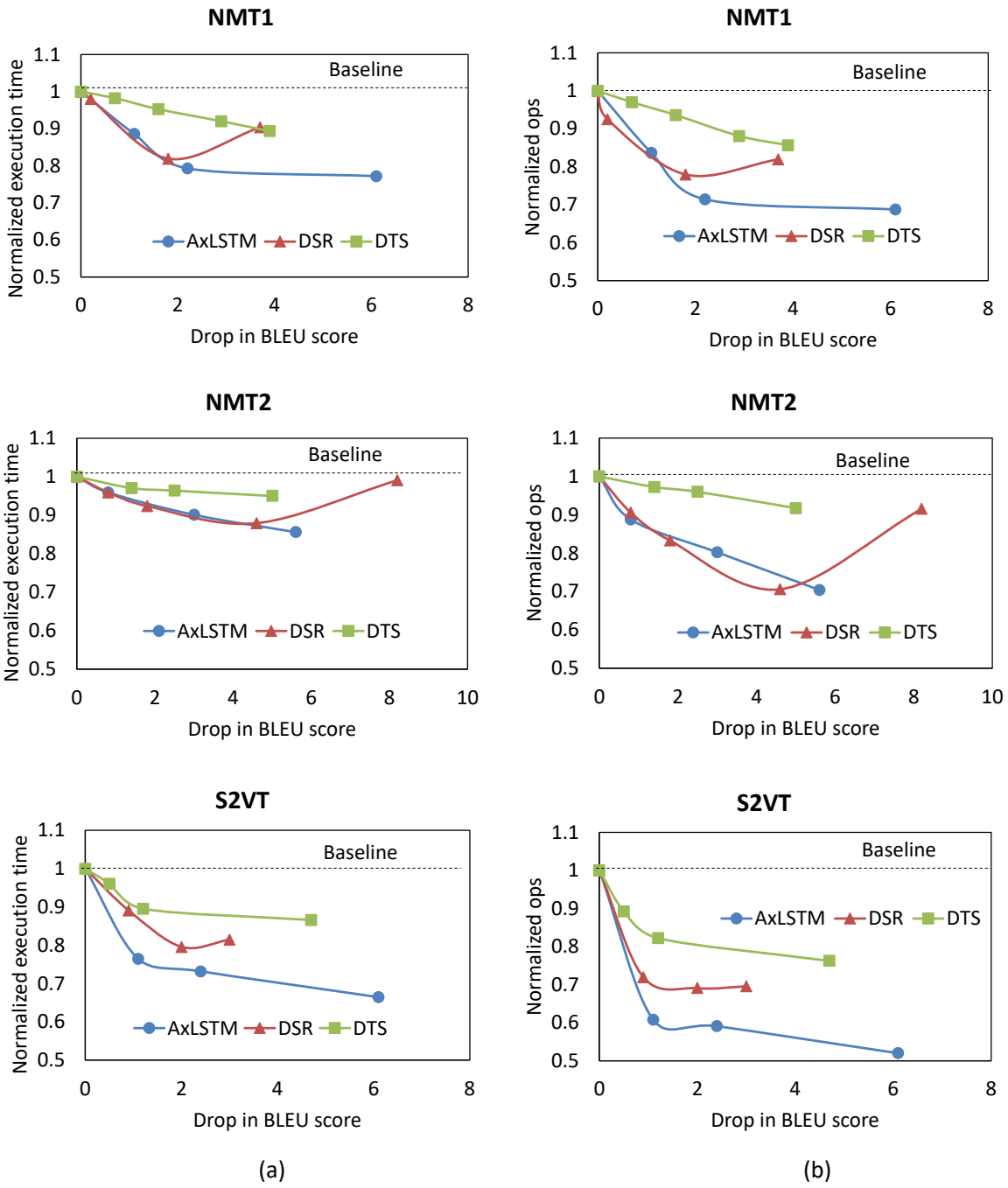

Fig. 5.4.: (a) Normalized execution time and (b) Normalized compute operations versus drop in quality using AxLSTM for sequence-to-sequence models

els, or equivalently, different BLEU scores, are obtained by varying how aggressively input symbols are skipped in DTS and how aggressively the state size is reduced using DSR. AxLSTM reduces the execution time by 13%, 19.1% and 23.5% for average drops in BLEU scores of 0.9, 2.5 and 5.8, respectively. This models were obtained by applying quality constraints of <1, <3 and <6 drops in BLEU scores respectively in the methodology described in Section 5.1.4. For each of these cases, we also calculated the theoretical reduction in compute operations by taking into account the number of skipped symbols and the reduction in state size. The corresponding values are shown in Figure 5.4(b). We observe that AxLSTM achieves 22.2%, 29.8% and 36.29% reduction in operations for average drops in BLEU scores of 0.9, 2.5 and 5.8, respectively. The reduction in compute operations don't translate entirely into the execution time benefits because of the overheads associated with DTS and DSR, which are discussed in more detail in Section 5.3.2.

Figures 5.4(a) and 5.4(b) also show the individual execution time versus quality tradeoffs observed with DTS and DSR. The speedups and compute reductions observed with DTS depend on the amount of redundancy present in the input sequence. In general, frames in a video have more redundancy than words in a sentence. Specifically, consecutive frames in a video have a higher chance of conveying similar redundant information than consecutive words in a sentence. As a result, DTS can demonstrate higher benefits on sequence-to-sequence models with videos as input sequences. In Figure 5.4(a), DTS achieves 10.5% reduction in execution time on the S2VT model as opposed to 4.7% and 2.9% reductions on NMT1 and NMT2, respectively, for similar drops in BLEU scores. This corresponds to 17.8%, 6.4% and 4.1% reductions in compute operations in the S2VT, NMT1 and NMT2, respectively, as shown in Figure 5.4(b).

In contrast, the benefits observed with DSR depend primarily on the fraction of semantically complex input sequences in a dataset. A higher proportion of semantically simpler inputs allows S2VT to derive higher DSR benefits than the other two models. As shown in Figure 5.4, S2VT experiences 20.4% reduction in execution

time with DSR whereas NMT1 and NMT2 experience reductions of 18.1% and 7.6% respectively, for similar drops in BLEU scores. The reduction in compute operations follows a similar trend as illustrated in Figure 5.4(b). DSR achieves 30.9% reduction in compute operations on S2VT as opposed to 22.1% and 16.7% reductions on NMT1 and NMT2 models, respectively.

Figures 5.4(a) and 5.4(b) present an interesting behavior exhibited by models with DSR-based approximations. We observe that the execution time and compute operations initially drop with a reduction in BLEU score but subsequently increase with further reduction in BLEU score. This counterintuitive behavior can be attributed to the fact that these inferior quality models suffer from the problem of lack of coverage, in which the smaller state vector leads to loss of useful information. This results in over-translation, where some words are unnecessarily translated multiple times [121]. This causes $OutputSeqLen$ in Equation 3.3 to increase, outweighing the reduction in $ComputeTimePerOutputSymbol$, resulting in a net increase in the compute operations and decoding time.

5.3.2 Benefits Breakdown and Overhead analysis

In this subsection, we analyze the benefits observed with AxLSTM by providing a breakdown in terms of the individual execution time benefits observed during the encoding and decoding process. Figure 7.4 shows this breakdown across the three benchmarks for an average of 0.9 drop in BLEU score. On average, AxLSTM reduces the encoding and decoding time by 9.8% and 14.6% respectively, which translates to 13.9% reduction in overall execution time. It is important to note here that the decoding process accounts for a larger fraction of the overall execution time in both the baseline and the AxLSTM based implementations of all three models.

The figure also highlights the overheads encountered with AxLSTM. We observe an average of 6.4% and 2.08% overheads during the encoding and decoding process respectively, which in turn leads to an overall overhead of 3.5%. The encoding and

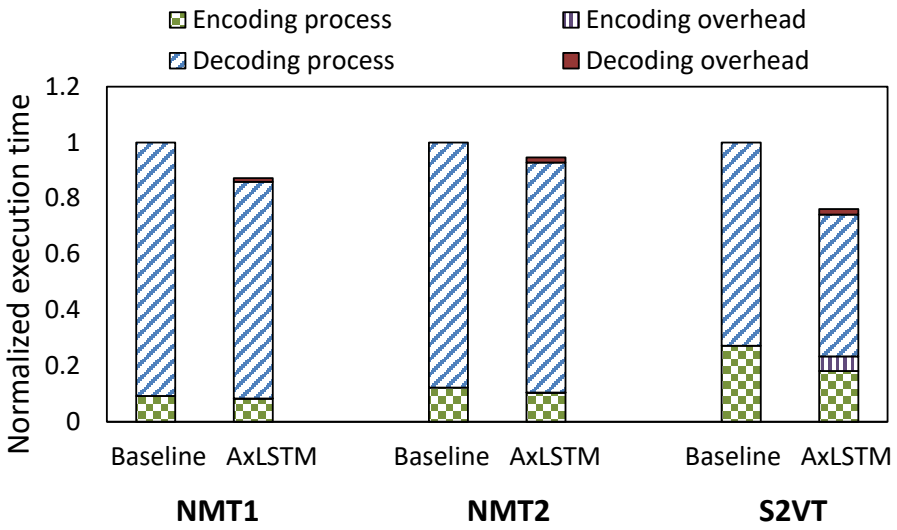

Fig. 5.5.: Execution time benefits breakdown with AxLSTM

decoding overheads are directly attributable to DTS and DSR, respectively. The encoding time overhead is equal to the time taken by the IA in DTS to determine the unimportant symbols on the fly. We observe that this overhead is a strong function of the heuristic adopted by IA. In general, the time consumed by the StateEffect heuristic is substantially less than that consumed by the InputDiff heuristic. Consequently, the encoding overhead for NMT1 and NMT2, which utilize the StateEffect heuristic, is $<0.1\%$ as opposed to an encoding overhead of 19.2% in the S2VT model, which utilizes the InputDiff heuristic. This difference stems from the fact that the computations involved in calculating the sum of absolute difference between pixels of consecutive video frames in S2VT are significantly more expensive than the quality table lookup performed in the other two models. On the other hand, the decoding overhead is proportional to the time expended by the SM to extract important elements and slice matrices as part of the DSR process. The observed values vary according to the number of important elements and the matrix sizes involved.

In summary, the individual benefits and overheads of DTS and DSR combine to determine the overall speedups observed with AxLSTM.

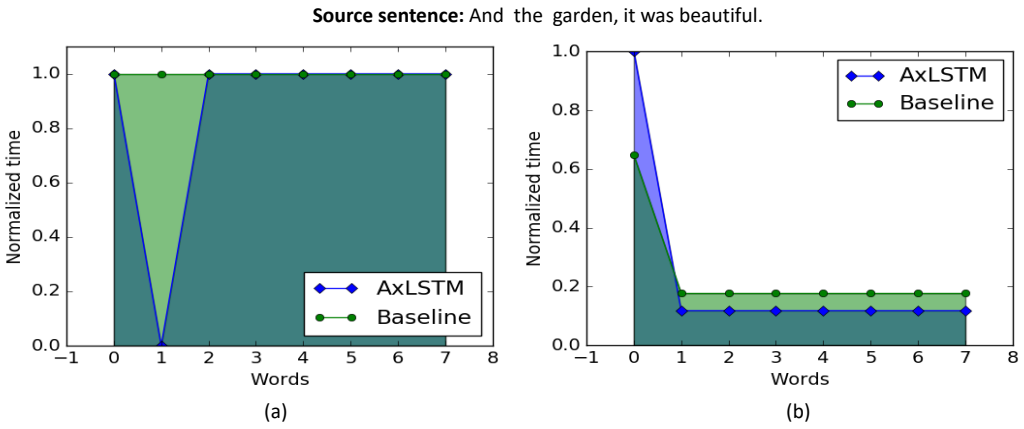

Fig. 5.6.: (a) Normalized encoding time per input word and (b) Normalized decoding time per output word for a semantically simple sentence with and without AxLSTM

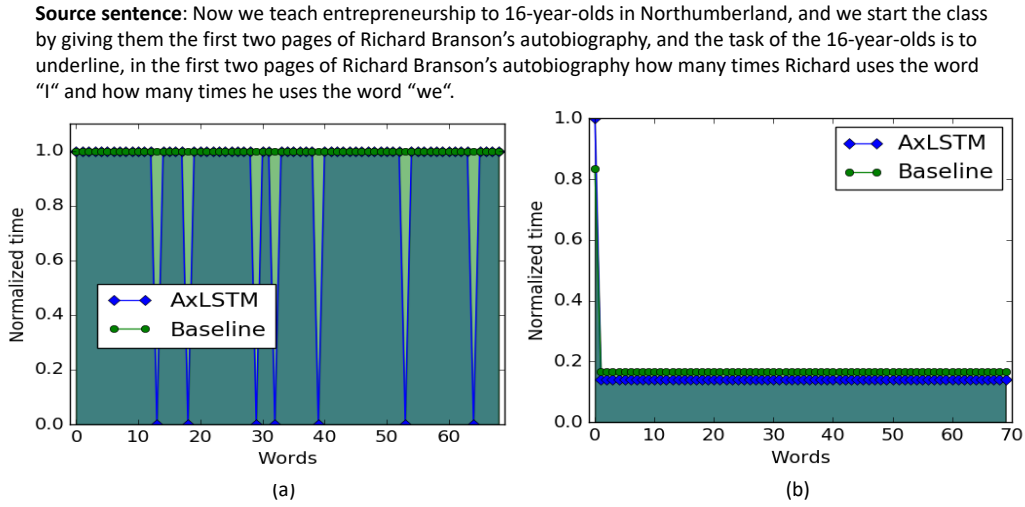

Fig. 5.7.: (a) Normalized encoding time per input word and (b) Normalized decoding time per output word for a semantically complex sentence with and without AxLSTM

5.3.3 Input Adaptive Approximations in Action

As described in section 5.1, the two techniques of AxLSTM, DTS and DSR, reduce two different components of Equation 3.3, *viz.*, $InputSeqLen$ and $ComputeTimePer$-

OutputSymbol. We illustrate these reductions in the NMT1 model with the help of two different sentences from the IWSLT dataset. For both sentences AxLSTM produces translations that are identical to the baseline LSTM, *i.e.*, there is no drop in output quality.

Figure 5.6(a) shows the *ComputeTimePerInputSymbol* with and without AxLSTM for each input symbol in a semantically simple source sentence comprising of 8 words, or equivalently, 8 input symbols. The specified quality constraint allows DTS to skip all instances of 'the' present in the sentence, leading to zero processing time for the second word in the case of AxLSTM. The *ComputeTimePerInputSymbol* for the remaining words is not affected by AxLSTM because the quality constraint doesn't allow DTS to skip them. Overall, the area under the curves denote the total encoding time and we observe that AxLSTM successfully reduces the encoding time from the baseline implementation.

Figure 5.6(b) shows the *ComputeTimePerOutputSymbol* with and without AxLSTM for each symbol during the decoding process of the same sentence. For all output symbols other than the first symbol, the *ComputeTimePerOutputSymbol* observed with AxLSTM is less than that observed with LSTM because of the reduced state size with DSR. The increase in time for the first symbol with AxLSTM can be attributed to the overhead associated with the matrix slicing process in DSR. However, the total area under the AxLSTM curve is less than that under the baseline curve, indicating that AxLSTM reduces the decoding time for this sentence.

The *ComputeTimePerInputSymbol* and *ComputeTimePerOutputSymbol* for a semantically complex source sentence with 69 words are shown in Figures 5.7(a) and (b). We observe that DTS is able to extract higher benefits in this case because of a more frequent occurrence of unimportant words. Specifically, the compute time reduces to zero for 7 words. However, the complex nature of this sentence dictates the use of a larger decoder state in DSR and accordingly, we observe a lower reduction in time during the decoding process. Nevertheless, the area under the AxLSTM curves

during both the encoding and decoding process are still less than the baseline, indicating that AxLSTM is beneficial.

In summary, AxLSTM is able to successfully extract execution time benefits in the form of reduced encoding and decoding times for input sequences of widely varying complexity.

5.4 Summary

Long Short Term Memory networks (LSTMs), a special class of RNNs, have attracted widespread attention due to their success in a range of machine learning applications involving sequences, including text generation, machine translation and speech recognition. We address the computational challenges posed by LSTMs by proposing AxLSTM, an application of approximate computing to improve the execution efficiency of LSTMs. AxLSTM comprises of two techniques — Dynamic Timestep Skipping (DTS) and Dynamic State Reduction (DSR) — that exploit the unique structure and computational characteristics of LSTMs. DTS reduces the effective number of timesteps of an LSTM by dynamically identifying input symbols that have little or no impact on the LSTM state and skipping them. DSR reduces the computations in each timestep of an LSTM by dynamically reducing the size of the LSTM state. Both these techniques are intrinsically input-adaptive, *i.e.*, they modulate the overall computational effort of an LSTM based on the complexity of the input sequences. We describe how AxLSTM can be applied in the context of sequence-to-sequence models. We implement AxLSTM within the TensorFlow deep learning framework and evaluate its benefits on 3 state-of-the-art sequence-to-sequence benchmarks. Our evaluations on an Intel Xeon Server reveal that AxLSTM achieves speedups of 1.08× - 1.31× with minimal loss in quality, and 1.12× - 1.37× when moderate reductions in quality are acceptable.

6. APPROXIMATE COMPUTING FOR SPIKING NEURAL NETWORKS

In this chapter, we focus on an emerging class of NNs, called Spiking Neural Networks (SNNs), which are often referred to as the 3^{rd} generation NNs. Compared to prior generations of NNs, SNNs exhibit higher biological fidelity *i.e.,* they mimic the spiking behavior of biological neurons. Therefore, SNNs have the potential to achieve better algorithmic performance with lower network complexity, especially in applications where temporal streams of data are processed [122]. SNNs are an active area of research, and in the recent past, SNNs have demonstrated state-of-the-art recognition performance on popular datasets such as MNIST [123] and CIFAR-10 [124].

Computational Challenges. SNNs are compute and data intensive workloads. For example, spiking networks emulating the functionality of the visual cortex may contain over a million neurons and a billion synapses [63]. When used to process an image of size 256×256, this translates to ∼2 giga scalar operations per frame, and over 4 GB memory. With growth in data and the need for higher accuracy, these requirements are only expected to increase further in the future. Hence, exploring avenues to improve the energy efficiency of SNNs is fundamental to their adoption.

Realizing this need, prior approaches have explored three key directions for efficient realization of SNNs. The first is software parallelization on commercial platforms such as multi-cores and GPUs [59–62]. The event driven nature of SNNs makes software parallelization quite challenging. In SNNs, work is generated dynamically as neurons spike, which renders the control flow and memory access patterns irregular. In contrast, commercial platforms, such as GPUs, are optimized for regular memory access patterns and fine-grained SIMD parallelism. The second direction is to build hardware architectures specialized for SNNs. A range of architectures, from low-

power IP cores [64] to large-scale systems [65, 66], have been proposed. The final set of efforts investigate alternate device technologies, such as memristor ans spintronics to realize SNNs [125–127].

Approximate SNNs. In this work, we explore a new direction - *approximate computing* - to improve the efficiency of SNNs. Due to their large-scale structure and the application context in which they are deployed, NNs are highly resilient to approximations in a significant fraction of their computations. Approximate computing has been applied to prior generations of (non-spiking) NNs [37, 48, 49]. However, due to the unique characteristics of SNNs (described below), such methodologies are not directly applicable. We believe ours to be the first effort to explore approximate computing for SNNs.

In SNNs, information is encoded and processed using trains of spikes. Each neuron is associated with a *membrane potential*, and a spike is dynamically generated when the potential goes above a specified threshold. When a neuron spikes, the potentials of all its fanout neurons are incremented by the weights of the respective connections. Thus *spike-triggered neuron updates* are the fundamental compute kernel in SNNs. To approximate SNNs, we develop a methodology, called AxSNN, to identify the criticality of spike-triggered updates and skip a subset of them to lower computational requirements thereby energy. AxSNN associates an approximation level with each neuron. The approximation level determines which fraction of a neurons successors will be updated when it spikes, as well as which fraction of its inputs it is sensitive to. All update operations are carried out when a neuron at the most accurate (or least approximate) level spikes. Progressively fewer update operations are performed as the approximation level of the neuron is increased. Spikes are entirely skipped when the neuron is in its most approximate state. To determine the approximation level of the neuron at runtime, AxSNN estimates the probability of the neuron spiking, and the significance of its spikes. For this purpose, it utilizes static network-level characteristics such as the number of fanout paths from the neuron to SNN outputs

and their average path weights, as well as dynamic local characteristics such as spike rate and current membrane potential of the neuron.

SNNs of any desired output quality can be realized using the above approach. We develop a framework that automatically tunes how aggressively neurons transition between approximation levels, thereby yielding an efficiency *vs.* quality trade-off. It is worth noting that the proposed approach is intrinsically input-adaptive *i.e.,* the approximate SNN modulates its computational effort across input samples, based on how often the neurons spike and the significance of the spikes to the eventual output.

In summary, the key contributions of this work are:

- We propose approximate computing as a new approach to improve the efficiency of SNNs.
- We develop a systematic approach to identify the criticality of spikes generated by each neuron at runtime. We correspondingly skip some or all updates due to the spike, thereby improving both compute and memory energy for a minimal loss in quality.
- We evaluate our approach in both software and hardware. For software, we utilize a C++ implementation of SNNs on a commodity server. In the case of hardware, we develop SNNAP, a Spiking Neural Network Approximate Processor. We achieve 1.2×-3.9× improvement in energy across a suite of 6 image recognition SNNs that contain ∼3K-14K neurons and ∼1.3M-48.8M connections.

The rest of the chapter is organized as follows. Sections 6.1 describes the design approach and methodology. Section 6.2 details the SNNAP architecture. The experimental methodology is presented in Section 6.3 followed by the results in Section 6.4. Section 6.5 concludes the chapter.

6.1 AxSNN: Design Approach and Methodology

To address the computational challenges imposed by SNNs, we propose AxSNN, a new design approach that leverages approximate computing to improve their efficiency. In this section, we present the key concepts behind AxSNN and describe the design methodology in detail.

6.1.1 Approximating Spike-triggered Updates

As described in Chapter 3, spike triggered neuron updates form the key compute primitive in SNNs. In our benchmark suite comprising of 6 image recognition SNNs, spike-triggered updates accounted for ∼97% of the overall operations, and consumed ∼93% of the total software execution time on a commodity Intel Xeon server. Therefore, we target these operations for approximation.

We associate an approximation level (α) with each neuron in the SNN. The approximation level (α) takes a value between 0 and 1, and is modulated dynamically during network evaluation. Based on α, we approximate the update operations associated with the neuron, as shown in Figure 6.1. An α of 1 indicates that the neuron is at its highest accuracy level, in which case all its fan-in and fan-out connections are active. As α is reduced, the neuron is progressively made more approximate by only enabling a fraction α of its fan-in and fan-out connections to be active. In this case, when the neuron spikes, only its active fan-out connections are updated, while the rest are skipped. Similarly, its membrane potential is updated only when one of its active fan-in connections spike. By dynamically deactivating input and output connections of a neuron, we reduce computation and save energy.

An important aspect of our approach is that we eliminate connections in a significance-aware manner, *i.e.,* based on synaptic weights. Once the SNN is trained, we pre-sort input and output connections to each neuron in increasing order of weight magnitude and deactivate the ones with lower magnitude first. For ease of implementation, we restrict the number of number of approximation levels to 5 *viz.* 1, 0.5,

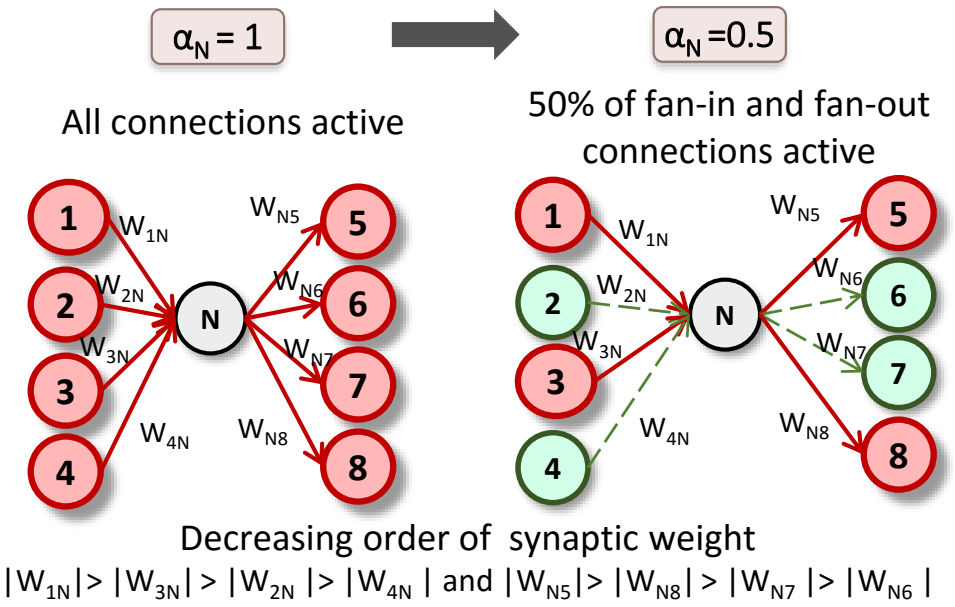

Fig. 6.1.: Neuron approximation mechanism

0.25, 0.125 and 0. Note that α of 0 indicates that the neuron is completely removed from the network as none of its fan-in and fan-out connections are active.

6.1.2 AxSNN: Overview

With the aforementioned approximation mechanism in place, we now present the overall approximation strategy adopted in AxSNN, as illustrated in Figure 6.2. The proposed strategy is dynamic, *i.e.,* the approximation levels of different neurons are determined at runtime in the course of evaluating an input. As shown in Figure 6.2, at the start of evaluation (t = 0), all neurons are set to their most accurate level (α = 1). We augment the SNN with an *AxSNN controller*, which is invoked periodically after every λ time steps. The AxSNN controller loops through each neuron in the network and determines the approximation level with which it should be executed for the next λ time steps. To make this decision, the AxSNN controller considers several key factors as described below.

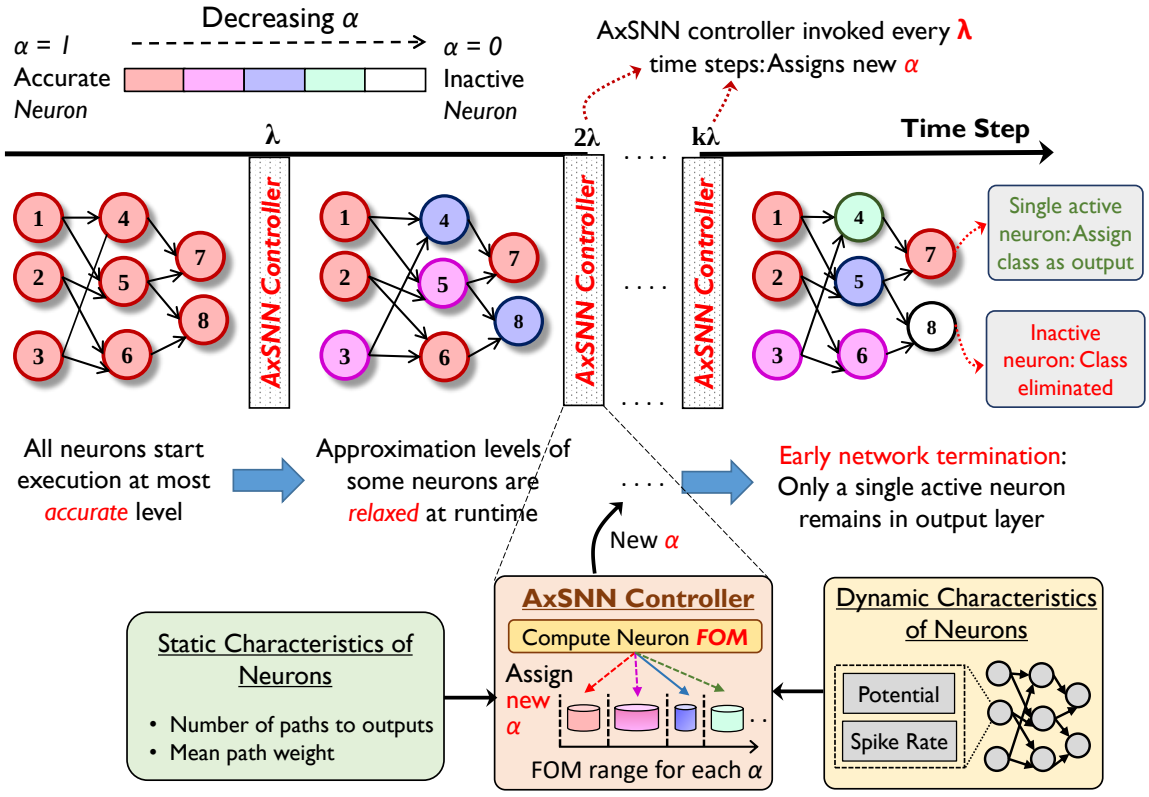

Fig. 6.2.: Overview of approximation strategy in AxSNN

Determining Approximation Levels

In order to determine the approximation level of a neuron, the AxSNN controller estimates the potential impact of approximations on the overall output quality using two key factors: (i) Spike probability, which captures the probability of the neuron spiking in the next λ time steps, and (ii) Spike significance, which denotes the relative importance of the neuron's spike to the overall network. Since the AxSNN controller is invoked periodically during execution, the above factors need to be estimated with very low overhead, as they directly offsets the benefits derived from approximate computing.

The AxSNN controller utilizes a mix of static and dynamic parameters to determine the spike probability ($SpikeProb$) and significance ($SpikeSig$) of each neuron. For a neuron to spike, its membrane potential should exceed its threshold value.

Therefore, to estimate the *SpikeProb* at runtime, as shown in Equation 6.1, we first identify how far the neuron's current potential is from its threshold, normalized to the reset value. We then divide the rate at which the neuron spiked in the past by the normalized potential difference to compute *SpikeProb*.

$$SpikeProb = \frac{SpikeRate}{(Thresh. - Potential)/(Thresh. - Reset)} \tag{6.1}$$

Intuitively, from Equation 6.1, the spike probability of a neuron is higher if it has spiked frequently in the past, or if its potential is close to the threshold value. *SpikeSig*, as shown in Equation 6.2, is computed as the product of the number of paths connecting the neuron to the network outputs and the mean of all path weights.

$$SpikeSig = NumPathsToOutputs * MeanPathWeight \tag{6.2}$$

We note that, since the number of paths and the mean path weight remain constant across all time steps, we can pre-calculate *SpikeSig* for each neuron once the SNN is trained. *SpikeProb* and *SpikeSig* are combined into a single Figure-Of-Merit (FOM), as shown in Equation 6.3.

$$FOM = SpikeProb * SpikeSig \tag{6.3}$$

The AxSNN controller contains a set of pre-defined FOM ranges for each approximation level. Based on the range in which the FOM of a neuron falls, the AxSNN controller assigns it the corresponding approximation level. Given an output quality requirement, the methodology used to obtain the FOM ranges is described in Section 6.1.3.

Early Network Termination

Another key aspect of the proposed approximation strategy is that it enables the SNN to classify an input even before all the time steps are complete. We note that, in the most approximate level, the neuron is completely disconnected from the network

and no further spike activity can occur at its output. Therefore, when all but one neuron in the output layer reach an α of 0, the execution is terminated and the input is assigned the class of the neuron that is active.

6.1.3 AxSNN: Design Methodology

We now describe how the FOM range for each approximation level is obtained. The FOM ranges determine how aggressively neurons transition across approximation levels, leading to different points in the efficiency *vs.* quality space. At the finest granularity, the FOM ranges can be defined individually for each neuron in the network. However, this leads to a prohibitively large design space, and further incurs significant overhead to store the FOM ranges in the AxSNN controller. We address this challenge by leveraging the fact that neurons in a layer are computationally similar, and constrain them to utilize the same FOM ranges. In other words, the FOM range for each approximation level is defined layer-wise.

We constrain the search space further by imposing a constraint that the FOM range endpoints for the different approximation levels are spaced in the proportion to the value of α. For example, the threshold to transition from $\alpha : 0.5 \rightarrow 0.25$ is constrained to be half of the threshold to transition from $\alpha : 1 \rightarrow 0.5$ and so on. This simplifies the search to finding one parameter per layer, which represents the threshold to transition from the most accurate level to the first approximate level ($FOM_{\alpha:1\rightarrow a1}$).

Algorithm 2 presents the pseudocode to find $FOM_{\alpha:1\rightarrow a1}$ for each layer. A trained SNN, the training dataset and the output quality constraint are provided as inputs. We first identify the maximum FOM for each layer, by setting its $SpikeProb$ to 1 (Line 2), and initialize its $FOM_{\alpha:1\rightarrow a1}$ to this value (Line 3). We then iteratively search the space of $FOM_{\alpha:1\rightarrow a1}$ as follows (Lines 4-9). For each layer, the corresponding $FOM^{Li}_{\alpha:1\rightarrow a1}$ is decreased by a small constant Δ, and the energy (E_{Li}) and quality (Q_{Li}) of the resultant AxSNN is computed by evaluating it on the training dataset

(Lines 5-6). Amongst these, we commit to the change in $FOM^{Li}_{\alpha:1\rightarrow a1}$ for the layer with the minimum E_{Li}/Q_{Li} ratio (Line 8). This process is repeated until Q_{Li} for none of the layers meet the specified quality constraint (Line 7).

Algorithm 2 Identifying FOM transition thresholds

Input: SNN: Trained spiking network, $TrData$: Training dataset, Q: Quality constraint

Output: $FOM^{Li}_{\alpha:1\rightarrow a1}$: Threshold to transition from most accurate to first approximate level for each layer

1: **For** each Layer Li:
2: Compute Max. FOM FOM^{Li}_{max}
3: Initialize $FOM^{Li}_{\alpha:1\rightarrow a1} = FOM^{Li}_{max}$
4: **while** (1) **do**
5: **For** each Layer Li:
6: Set $FOM^{Li}_{\alpha:1\rightarrow a1}$ -= Δ and compute E_{Li}, Q_{Li}
7: **if** $(Q_{Li} < Q \ \forall \ Li)$ **break**
8: Commit $FOM^{Li}_{\alpha:1\rightarrow a1}$ in layer with min. (E_{Li}/Q_{Li}) and $Q_{Li} > Q$
9: **end while**

In summary, by selectively skipping updates triggered by spikes, AxSNN achieves significant improvements in the implementation efficiency of SNNs.

6.2 SNNAP: Architecture

In this work, we evaluate AxSNN, using both hardware and software SNN implementations. To demonstrate the benefits in hardware, we propose Spiking Neural Network Approximate Processor (SNNAP), a new hardware architecture for SNNs, enhanced to support the proposed approximation mechanism. Figure 6.3 shows the block diagram of SNNAP. It consists of two types of processing units: (i) a scalar Leak-and-Spike (LnS) unit, and (ii) a 1D array of Spiking Neuron Processing Elements

(SNPEs). The architecture also contains two memory banks, the State Memory (SM) and the Weight Memory (WM), which store the neuron potentials and the weights respectively. A global controller orchestrates the overall execution.

We now describe how SNNs are realized in SNNAP. The LnS unit loops over all neurons in all time steps, leaks its membrane potential and checks if the potential is above its threshold. If not, the execution moves on to the next neuron. In the case the neuron spiked, the SNPE array is activated, which reads the SM and WM banks and updates the potentials of the fan-out neurons. The network state and weights are statically partitioned across the SM and WM banks, such that the compute load to each SNPE is roughly balanced.

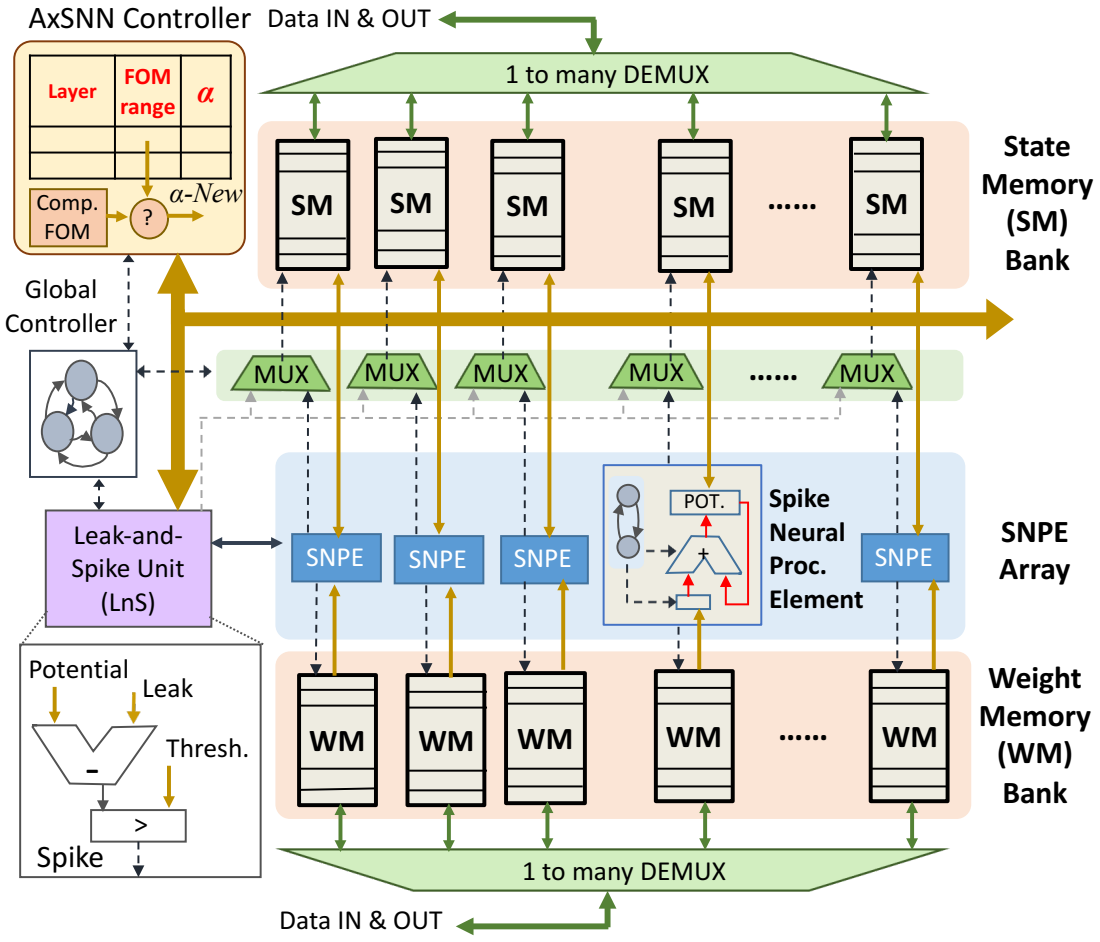

Fig. 6.3.: Block diagram of SNNAP

To support approximate operation, SNNAP is enhanced with an AxSNN controller that is periodically invoked by the global controller. We also add 3 bits to the neuron states in the SM banks to store their approximation levels. Further, the weights in WM are sorted in accordance to their magnitude, such that updates can be skipped with no overhead when neurons transition approximation levels. Overall, SNNAP incurs ∼3.5% area overhead to support approximate operation.

6.3 Experimental Methodology

In this section, we describe the methodology used in our experiments to evaluate AxSNN.

6.3.1 Runtime and Energy Evaluation

We evaluate AxSNN in both hardware and software. The software was implemented in C++ and run on a 2.7GHz Intel Xeon server with 128GB memory. In the case of hardware, the SNNAP architecture was implemented at the Register Transfer Level (RTL) using Verilog HDL and synthesized to IBM 45nm technology using Synopsys Design Compiler. The micro-architectural and circuit level parameters of the implementation are shown in Figure 6.1a. We measured performance through cycle-accurate simulations using ModelSim, and the switching activity traces were fed to Synopsys Power Compiler to estimate (static and dynamic) logic power at the gate-level. Energy was computed as the product of power and execution time. CACTI was used to model the memory structures. The memory energy was computed by profiling the number of memory reads and writes and multiplying them with the corresponding energy values obtained from CACTI.

Table 6.1.: (a) SNNAP parameters (b) Application benchmarks

Metric	Value
Feature Size	45nm
Area	0.34 mm^2
Power	175.74 mW
Gate Count	193723
Frequency	1 GHz
No. of SNPE lanes	64

(a)

Dataset	Type	Layers	Neurons	Connections
MNIST	F.C	4	3194	2392800
	Conv.	6	13594	6527300
NORB	F.C	4	3053	1276500
	Conv.	6	89806	4012960
SVHN	F.C.	5	7582	11149000
CIFAR-10	F.C.	4	14082	48854400

(b)

6.3.2 Application Benchmarks

Our benchmark suite, listed in Table 6.1b, comprises of 6 image recognition SNNs, of which two are convolutional networks and the rest are fully-connected networks. Table 4.1b also lists the number of layers, neurons and connections in each benchmark. The networks were trained using the methods described in [124] and [123]. We utilized classification accuracy, *i.e.,* fraction of inputs classified correctly, as our metric to evaluate application quality.

6.4 Results

This section presents the results of our experiments that demonstrate the benefits of AxSNN in both HW and SW.

6.4.1 Energy Benefits at Iso-Accuracy

Figure 6.4 shows the normalized benefits using AxSNN with no loss in accuracy across all benchmarks. The improvements are quantified using three metrics: (i) number of spike update operations, (ii) hardware energy, and (iii) software energy. We

achieve 1.4×-5.5× reduction in spike update operations across the benchmarks, which translates to 1.2×-3.62× and 1.26×-3.9× improvement in hardware and software energies respectively, at iso-accuracy. The benefits largely depend on the difficulty of classifying the inputs in each dataset. For example, in the case of MNIST where we observe the most improvements, almost 99% of the inputs can be terminated early. In contrast, only 63% and 52% of inputs are amenable to early termination in CIFAR and SVHN respectively. The hardware and software energies include the energy spent in performing other control and compute operations involved in SNN evaluation in addition to the spike update operations. Further, they also reflect the overheads associated with realizing the approximation methodology. Due to the above reasons, the reduction in the spike update operations don't translate entirely into the hardware and software benefits.

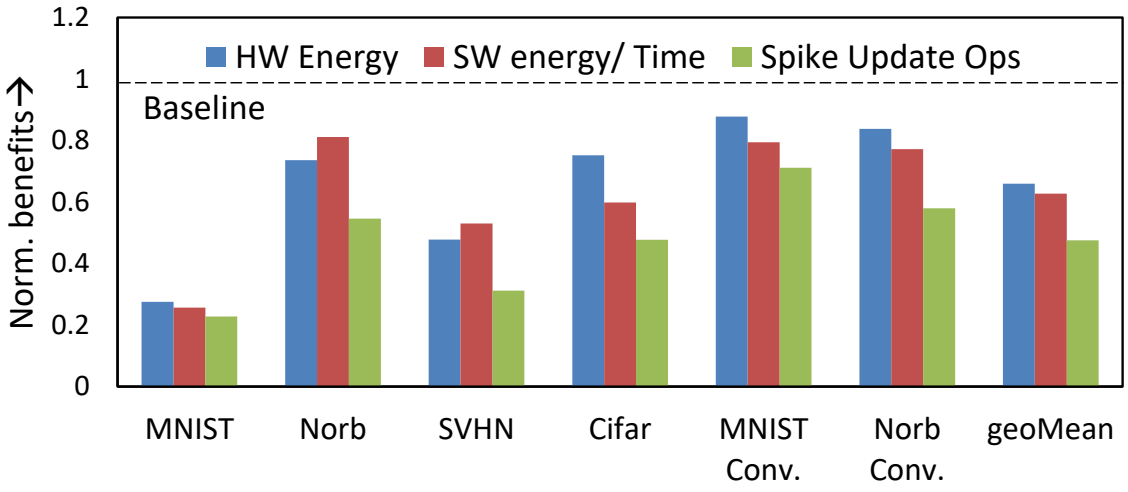

Fig. 6.4.: Normalized OPS and energy benefits for different applications

6.4.2 Energy vs. Accuracy Tradeoff

By modulating how aggressively the neurons transition to higher approximation levels, different application-level qualities can be achieved in AxSNN with corresponding benefits in energy. Figure 6.5 shows the energy *vs.* quality trade-off curves thus

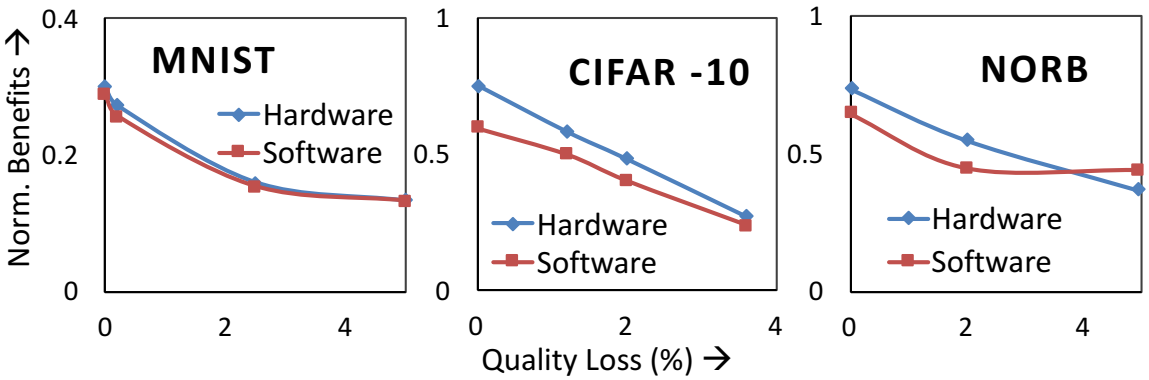

Fig. 6.5.: Normalized energy *vs.* accuracy trade-off for 3 SNN benchmarks

achieved for 3 benchmarks. On an average, we obtain 1.86×, 2.51× and 3.37× energy improvement for a quality loss of 1%, 2% and 5% respectively. Since modulating application quality only requires a change in the FOM ranges set in the AxSNN controller, the same implementation can easily support multiple application quality levels and can switch between them at runtime.

6.4.3 Input Adaptive Approximations: Easy vs. Hard Inputs

An important aspect of AxSNN is that the approximations are intrinsically input adaptive. Inputs that are easy-to-classify are approximated more than the harder inputs, thereby scaling computational effort in proportion to input difficulty. We illustrate this in Figure 6.6, using 2 examples from the MNIST handwritten digit recognition dataset. Figure 6.6 plots the approximation level of each neuron in the SNN at each time step. Red denotes the most accurate level, while white indicates the neuron is inactive. The AxSNN controller is invoked after every 5 time steps, and the neurons may switch approximation levels after this interval. For ease of understanding, we plot the approximation level for the output neurons separately (graph on the right for both inputs).

We observe that all neurons start execution (t=0) in their most accurate state. However, after the AxSNN controller is first invoked (t=5), a substantial fraction of

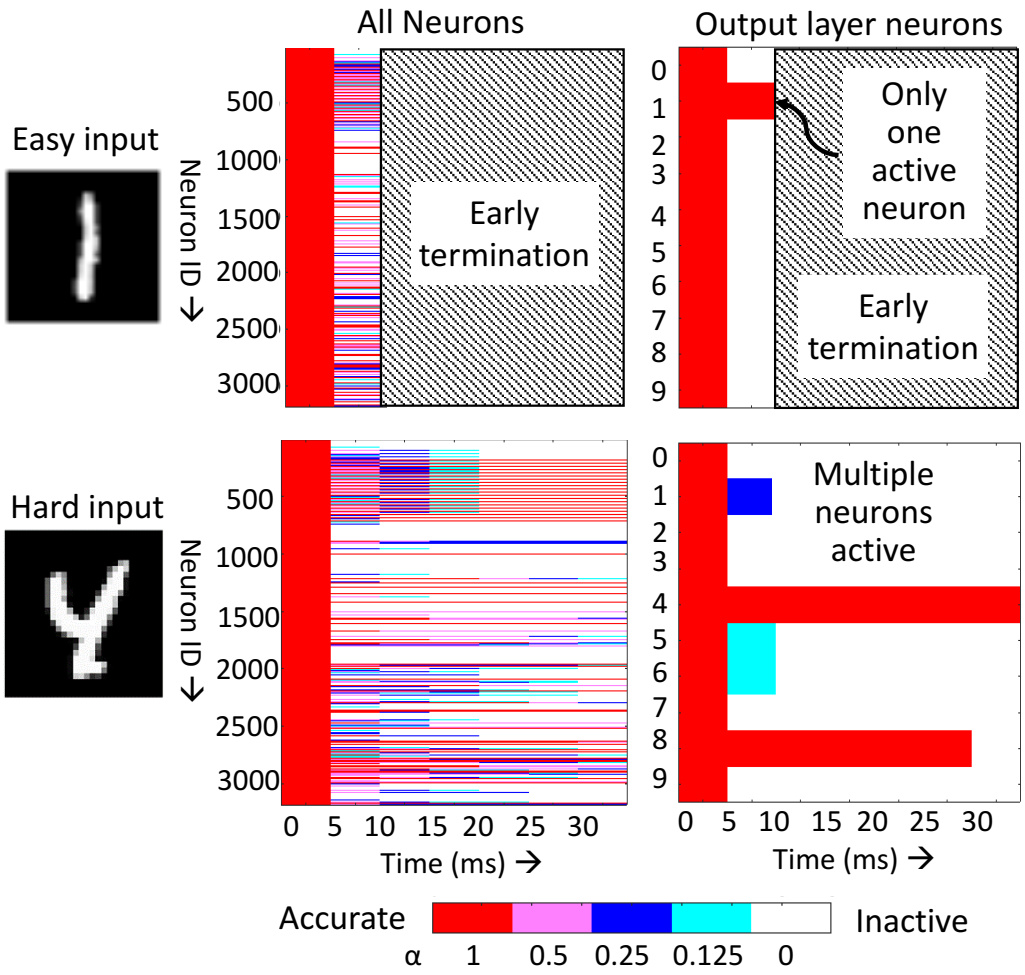

Fig. 6.6.: Approximation levels of neurons at each time step

the neurons change approximation levels. Specifically, we observe that several output neurons have been rendered inactive, effectively eliminating the respective class labels from consideration. For example, in the case of the easy input, except for the class '1', neurons corresponding to all other classes are inactive. Therefore, the execution is terminated and the input is classified. For the hard input, 5 classes are in contention, albeit with their neurons at different approximation levels. This reduces to 2 class labels ('4' and '8') at t=10, and the input is classified after 35 time steps. We still achieve significant benefits in the case of the hard input, as neurons in other layers are progressively approximated.

6.5 Summary

Spiking neural networks (SNNs) are an emerging class of neural networks that have demonstrated significant promise in realizing several recognition, data analytics and computer vision applications. To improve the implementation efficiency of SNNs, we utilize approximate computing, a design paradigm in which selected computations are performed in an approximate manner, saving energy with minimal loss in quality. We identified updates triggered by a neuron spike as the key compute primitive in SNNs. We target our approximations at this primitive by skipping some or all updates caused by a spike. We develop a methodology, AxSNN, to identify the spike triggered neuron updates that can be skipped while meeting the specified output quality requirement. Across a suite of 6 image recognition SNN benchmarks, we demonstrate significant benefits in energy for both hardware and software implementations.

7. EFFICACY OF PRUNING IN ULTRA-LOW PRECISION DNNS

Among different approximate computing approaches for improving the efficiency of DNNs, pruning and quantization have emerged as two of the most popular techniques that reduce both DNN model sizes and computational complexities. But they have been primarily explored as independent techniques, exploiting different forms of redundancies in DNNs. In this chapter, we investigate the opportunities for combining them, especially as they are driven to their individual limits.

Quantization refers to the conversion of DNN data-structures, like weights and activations, from full-precision to low-precision values. It allows these data-structures to be represented and stored using fewer bits, reducing their memory requirements. It also reduces the computational demands of DNNs as the low-precision values can be processed by simpler arithmetic units. Previous research efforts in quantizing DNNs have succeeded in pushing the limits of quantization to perform DNN training and inference with 8 or lower bits of precision while maintaining accuracy levels close to full-precision implementations [103,105]. This has further motivated different computing platforms, such as, CPUs [128], GPUs [13] and DNN accelerators [129] to natively support low-precision operations in hardware. As these ultra-low precision (sub-8 bits) DNNs become increasingly mainstream, we analyze the efficacy of pruning them to extract additional benefits in terms of model size during inference.

Pruning removes redundant weights and neurons in a DNN, setting their values to zeros. It produces sparse models that are both compute and memory-efficient compared to their dense counterparts. The memory benefits are obtained by utilizing different sparse coding schemes, that store only non-zero values, along with information about their locations. However, we observe that these sparse storage formats suffer from inefficiencies in the ultra-low precision regime as the overhead of storing

the non-zero locations starts to dominate. For instance, the memory compression benefits of a 70% pruned model, with respect to a dense model, reduces from 2.21× to 1.1× when the weight precision is decreased from 8 to 2 bits when using the Compressed Sparse Column (CSC) sparse storage format [130]. Thus, there is a need to re-evaluate the benefits of pruning as we go into the domain of ultra-low precision DNNs.

In this work, we examine the efficacy of pruning in ultra-low precision DNNs by measuring the compression benefits achieved by storing pruned DNNs in different sparse formats. We consider two sparse formats widely used for DNNs, namely, Compressed Sparse Column (CSC) and Sparsity map (Smap) format. We also propose a new format, compressed Sparsity map (cSmap), that compresses the binary sparsity map in the Smap format to reduce the overhead of storing non-zero value locations. We illustrate a mechanism for realizing this format by re-purposing test pattern compression methods widely used in manufacturing test. For each of the formats, we formulate analytical expressions for the achieved compression ratios and identify their variation with precision and sparsity levels. Across pruned and quantized versions of 6 state-of-the-art DNNs, we observe that all three formats suffer from low compression ratios in the ultra-low precision regime. However, the compression ratios of the individual formats vary differently with sparsity and precision, leading to different formats achieving highest compressions in different scenarios. Accordingly, we further propose the use of a hybrid compression scheme that selects the optimal sparse format at a network or even layer granularity, to increase the compression benefits across a range of DNNs.

The key contributions of this work are summarized below.

- We evaluate the efficacy of pruning in ultra-low precision DNNs.
- We develop analytical expressions for the compression benefits of two popular sparse formats, CSC and Smap format, and analyze the effects of different parameters on them.

- We propose a new format, compressed Sparsity map (cSmap), to reduce the overhead for storing non-zero locations in the Smap format and realize it using test pattern compression methods employed in manufacturing test.

- We evaluate the compression benefits of all three formats across 6 state-of-the-art Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) with varying sparsity and precision levels. We further propose a hybrid compression scheme that dynamically identifies the best sparse format in a given scenario and improves the average compression ratio by 18.3% - 34.7% over homogeneous compression schemes for a 2-bit DNN.

7.1 Sparse Storage Formats for Pruned DNNs

In this section, we describe the sparse formats considered in our analysis and formulate their compression ratios and overheads. We also identify the impact of varying different parameters on them. Finally, we detail the implementation of a new format proposed in this work, using test pattern compression techniques.

The first sparse format examined in this work is the Compressed Sparse Column (CSC) format, which is well-suited for storing irregular sparse matrices. We specially consider a small variant of the basic format explored in sparse DNN accelerators like Eyeriss [131] and EIE [69] to maximize compression benefits on DNNs. The second sparse format that we consider is the sparsity map (Smap) format which has also been explored by previous efforts in compressing sparse DNNs [132, 133]. The final sparse format explored in our analysis is referred to as the compressed Sparsity Map (cSmap) format. It is a new sparse format proposed in this work, targeted towards reducing the overhead of non-zero locations in the Smap format.

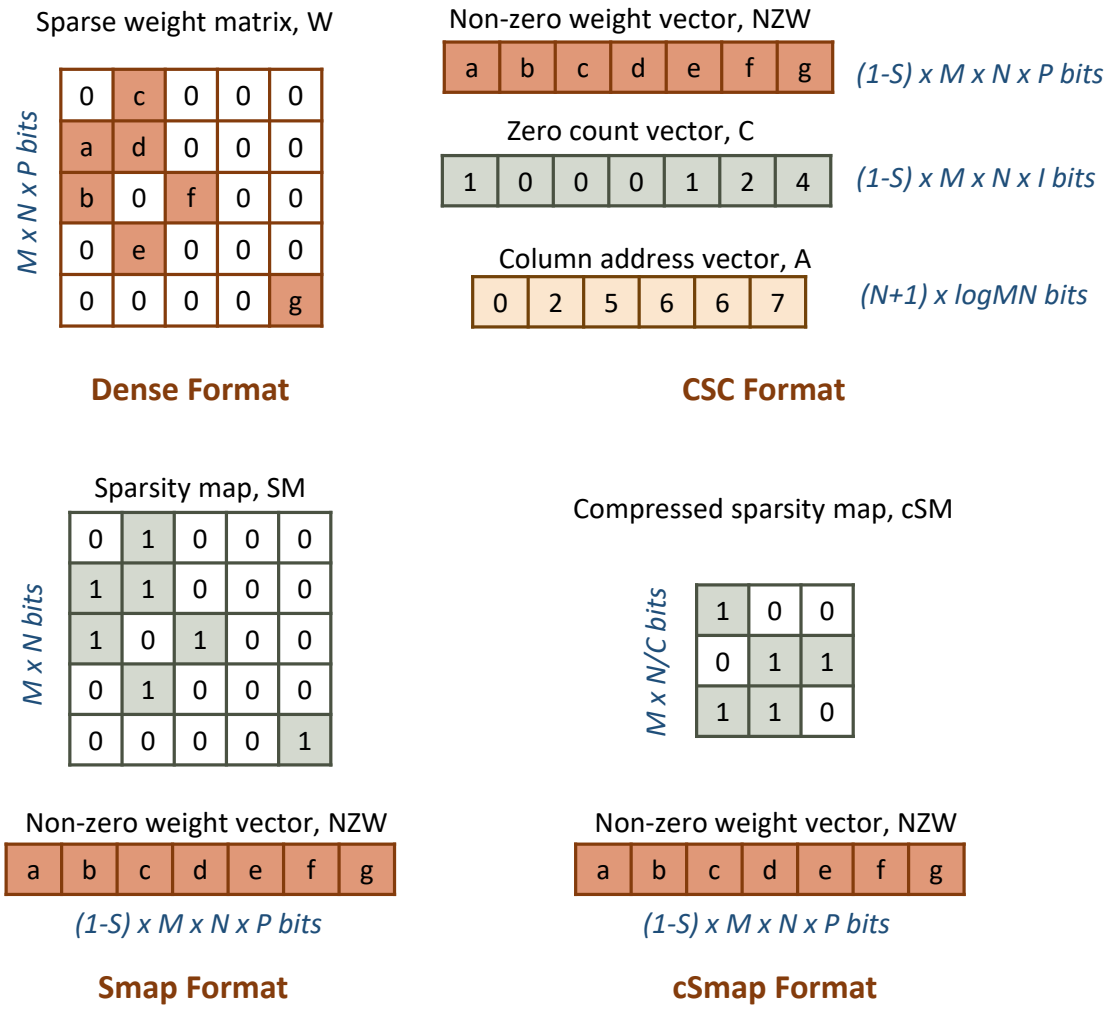

Fig. 7.1.: Data-structures and memory requirements of different storage formats

7.1.1 Compression Ratios

Compressed Sparse Column (CSC) Format. Fig 7.1 illustrates an example sparse weight matrix, W, stored in the CSC format. It consists of 3 different data-structures — the non-zero weight vector, NZW, zero count vector, C, and column address vector, A. NZW stores the non-zero weight values of W while C stores the number of zeros between them. The values are ordered in a column-major manner with the A vector storing the starting index for each column in NZW and C. Thus,

all non-zero elements in column p of W lies between indices $A[p]$ and $A[p+1]$ in NZW.

The storage requirements of the individual data-structures in CSC are also shown in Fig 7.1. The original $M \times N$ weight matrix, W, requires $M \times N \times P$ bits ($= Dense_{bits}$), when stored in a dense format with a weight precision of P. For S amount of sparsity, or fraction of zeros, in W, the NZW vector requires $(1-S) \times M \times N \times P$ bits of storage to store $(1-S) \times M \times N$ number of non-zeros. Similarly, the C vector requires $(1-S) \times M \times N \times I$ bits of storage, where I equals the precision of count values. On the other hand, the A vector requires $(N+1) \times \log_2 MN$ for storing $(N+1)$ entries in $\log_2 MN$ bits of precision. $\log_2 MN$ bits is the minimum precision required to support the worst-case scenario of $S = 0$, when the last value of A equals MN. The overall storage requirements for the CSC sparse format can be expressed by the following equation.

$$CSC_{bits} = (1-S) \times M \times N \times (P+I) + (N+1) \times \log_2 MN \tag{7.1}$$

Accordingly, the compression ratio achieved by the CSC format can be expressed as

$$C_{CSC} = \frac{Dense_{bits}}{CSC_{bits}} = \frac{P}{\frac{(N+1) \times log_2 MN}{M \times N} + (1-S) \times (I+P)} \tag{7.2}$$

Overall, the A and C vectors form the overhead for specifying the location of non-zeros in this format. Clearly, the compression ratio decreases when the overhead becomes significant for (i) smaller values of P (lower precision) and (ii) smaller values of S (lower sparsity).

Sparsity map (Smap) Format. An example of a sparse matrix representation using the Smap format is shown in Figure 7.1. It consists of two data-structures, NZW and SM. The NZW vector resembles that in the CSC format, while the Sparsity Map (SM) matrix, is a binary matrix indicating the location of non-zeros in W. Specifically, '1' values in SM correspond to non-zeros while '0' values correspond to zeros. Assuming the same sparse weight matrix W as above, the NZW vector again requires $(1-S) \times M \times N \times P$ bits, as shown in Figure 7.1. The SM matrix

requires $M \times N$ bits since it stores a single bit for each element in W. The following equation formulates the overall storage requirements for the Smap format.

$$Smap_{bits} = (1 - S) \times M \times N \times P + M \times N \tag{7.3}$$

The compression ratio, C_{Smap}, achieved by the Smap format amounts to

$$C_{Smap} = \frac{Dense_{bits}}{Smap_{bits}} = \frac{P}{1 + (1 - S) \times P} \tag{7.4}$$

The SM matrix forms the overhead in this format and it can dominate the overall storage requirements for (i) smaller values of P (lower precision) and (ii) smaller values of S (lower sparsity), leading to lower compression ratios in these scenarios.

Compressed Sparsity map (cSmap) Format. The cSmap format is a new sparse format proposed in this work, which improves upon the Smap format by compressing the SM matrix into a cSM matrix, as shown in Figure 7.1. Considering a compression ratio of C, the memory footprint of the cSM matrix equals $(M \times N)/C$ bits. The memory footprint of the NZW vector, storing the non-zero values, remains same as the previous two formats. The total storage requirements for the cSmap format can be expressed by the following equation.

$$cSmap_{bits} = (1 - S) \times M \times N \times P + \frac{M \times N}{C} \tag{7.5}$$

This leads to an overall compression ratio of

$$C_{cSmap} = \frac{Dense_{bits}}{cSmap_{bits}} = \frac{P}{\frac{1}{C} + (1 - S) \times P} \tag{7.6}$$

The value of the compression ratio decreases for (i) smaller values of C (lower SM compression), (ii) lower values of S (lower sparsity) and (iii) smaller values of P (lower precision). In this work, we illustrate a mechanism for compressing the SM matrix by utilizing test pattern compression tools. The details of the mechanism will be presented in the next subsection.

Overall, we clearly see that all three sparse formats suffer from low compression ratios in the ultra-low precision regime considered in this work. But the exact variation in the value of the compression ratio, with different precision and sparsity levels, depends on the format, and will be discussed in greater detail in Section 7.3.

7.1.2 Realizing the cSmap Format using Test Pattern Compression

We now present the details for realizing the cSmap format with the help of the test pattern compression tools discussed in Section 3.7. The advantages of utilizing these test pattern compression tools are three-fold. First, these tools can efficiently compress binary test vectors and should accordingly be able to compress the binary matrix, SM, into its compressed form, cSM. Second, the decompression overhead is very low, allowing cSM to be decompressed into SM at runtime. Third, it allows us to re-use the existing test units in hardware.

However, commercial test pattern compression tools (for *e.g.*, Mentor Graphics' Tessent TestKompress [134]) lack the flexibility to compress an arbitrary binary matrix, SM. They are engineered to compress only test patterns generated by ATPG (Automatic Test Pattern Generation) tools, and are unable to compress any user-defined test patterns. As a result, we modify the DUT and the ATPG process to generate patterns resembling the values in SM. The overall process consists of three distinct steps, namely, creation of DUT, generation of constrained patterns and compression of these patterns.

First, we create a simple flip-flop (FF) based design consisting of rows of FFs connected in chains. Each FF gets converted to a scan cell and rows of FFs get converted to scan chains during the test insertion process, to yield a design similar to the DUT in Figure 3.8. The size of the design is set to be a function of the size of the SM matrix that we want to compress. Specifically, the number of FF rows is set to be αM and the number of FF in each row is set to be βN, where M and N have the same meaning as in the previous subsection. Further, each bit in the SM matrix mapped to a unique FF in the design. For α and β values of 1, this mapping is straightforward with each matrix element mapping to the FF with the same index. For higher values of α and β, we adopt a strided and staggered mapping strategy, where element (i, j) in the matrix maps to scan cell $(\alpha \times i,\ \beta \times j + i\ \%\ \alpha)$.

Next, we force the ATPG tool to generate patterns resembling the SM matrix by constraining the FFs to have values equal to their corresponding values in SM. The strided and staggered mapping strategy ensures that the constraints are sufficiently spread out and the ATPG is able to generate the constrained patterns efficiently.

Finally, we compress these patterns to form cSM. For DNNs with larger weight matrices, we further partition SM into smaller submatrices and perform the design creation, test pattern generation and compression process iteratively on individual submatrices.

Lossy scheme to improve SM compression. In order to improve the compression of the SM matrix even further, we also allow the compression mechanism to be lossy in nature. Specifically, during the ATPG process, we constrain all cells corresponding to '1' valued bits but only a fraction, f_c, of cells corresponding to '0' valued bits in SM. The test pattern bits corresponding to the remaining $(1 - f_c)$ fraction of '0' elements in SM can potentially end up having a value of '1'. The compressed version of these test patterns, forming the cSM matrix, thus decompresses to form dSM, which is a small variant of the SM matrix with some additional 1' values, as shown in Figure 7.2. Clearly, the NZW vector now has to store zero values corresponding

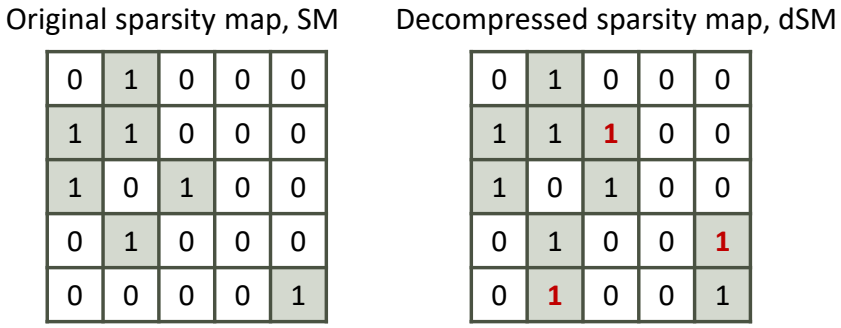

Fig. 7.2.: Lossy SM compression

to these extra '1's, increasing its memory footprint to $(1 - S_{dSM}) \times M \times N \times P$ bits where S_{dSM} is the sparsity fraction in the dSM matrix. Equation 7.6, describing the

compression ratio achieved by the cSmap format, thus gets modified to the following form.

$$C_{cSmap} = \frac{Dense_{bits}}{cSmap_{bits}} = \frac{P}{\frac{1}{C} + (1 - S_{dSM}) \times P} \tag{7.7}$$

7.2 Experimental Methodology

This section describes the methodology adopted in our experiments to study the efficacy of pruning in ultra-low precision DNNs.

7.2.1 Benchmarks

Our benchmark suite consists of Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) for image-recognition and language modeling tasks, respectively. The details of the benchmarks are listed in Table 7.1. The number of weights, or connections, in the networks varies between 0.43 million and 135.83 million. We prune each of the networks to different sparsity levels using AGP [102] and quantize them to four different ultra-low weight precision levels — 2, 4, 6 and 8 bits. As the quantization is performed after pruning, the sparsity fraction and distribution remains constant across precision levels.

Table 7.1.: Benchmarks

Name	MNISTconv	AlexNet	ResNet18	ResNet50	MobileNetv2	WordLangModel
Dataset	MNIST	ImageNet	ImageNet	ImageNet	ImageNet	ImageNet
Type	CNN	CNN	CNN	CNN	CNN	RNN
Accuracy (%)	99.09	56.52	69.76	76.13	71.88	84.23 (Ppl: Perplexity)
# Weights (M)	0.43	61.09	22.73	11.51	3.49	135.83

7.2.2 Compression Evaluation

For each of the pruned networks, we calculate the compression benefits of the three sparse formats with respect to a dense network of the same precision. The individual weight memory requirements of a layer, as expressed by equations 7.1, 7.3 and 7.5, are summed up to yield network-level compression benefits. For the CSC format, we consider $I = 4$, similar to [69]. For the cSmap format, we perform a grid search on the values of α, β and f_c, and select the values yielding the highest compression ratio for each network.

7.3 Results

In this section, we present the results of our experiments on exploring the efficacy of pruning in ultra-low precision DNNs.

7.3.1 Network-level Compression Ratios

Figure 7.3 shows the network-level compression ratios achieved by the 3 sparse formats across different pruned and quantized versions of the benchmarks. The individual networks, along with their weight sparsity levels after pruning, are listed horizontally. The sparse formats and weight precision values are listed vertically. The colors in the figure represent the compression ratios achieved by a format when the corresponding sparse network is stored with a specific weight precision. All compression ratios below 1 are shown in red color and it highlights the scenarios where using a sparse format hurts us by increasing the memory requirements to more than that of a dense network with the same precision. On the other hand, all compression ratios above 1 are shown in green color, corresponding to scenarios where we can benefit from pruning by utilizing a sparse format. Overall, the compression ratio varies between 0.65×-5.27× across different formats, networks, precisions and sparsity levels. Individually, it varies between 0.65×-5.27×, 0.99×-4.44× and 0.99×-3.43× for the CSC, Smap and

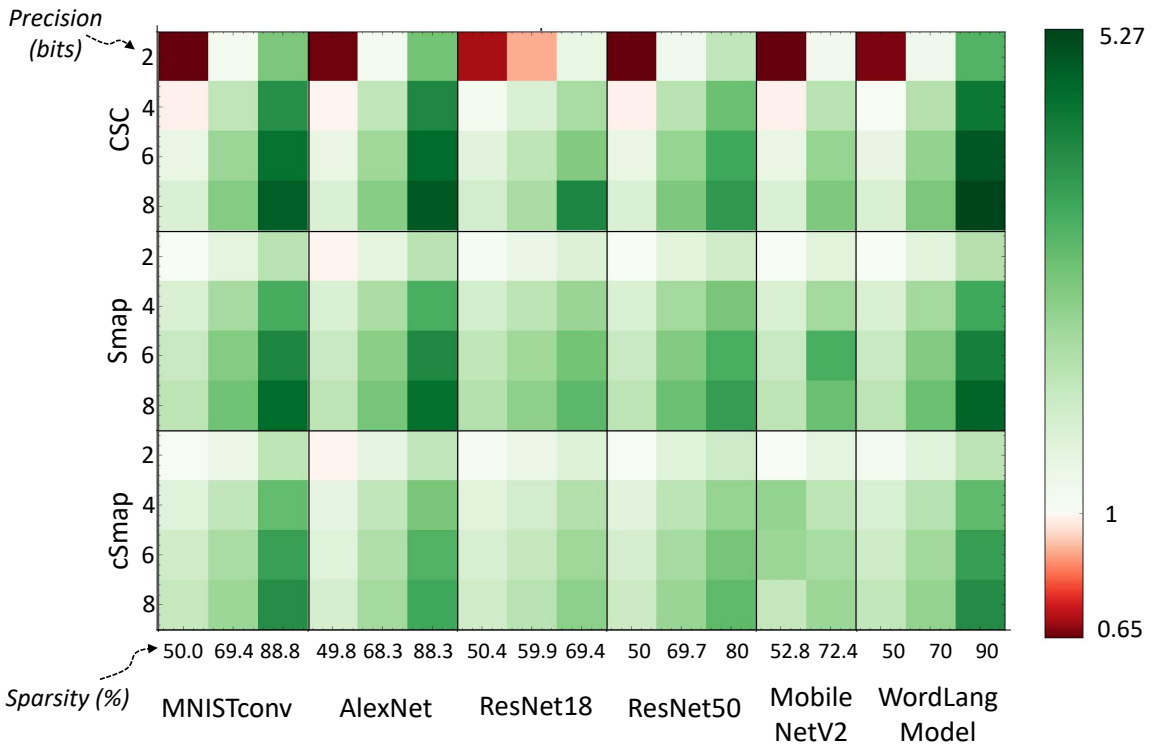

Fig. 7.3.: Network-level compression

cSmap format, respectively. It decreases with decreasing precision and sparsity levels, as evident from the shift in darker green shades in the bottom right corner to lighter green or red shades in top left corner for all formats and networks. For instance, the top left corner illustrates the compression ratio of 0.65× achieved by the CSC format for the MNISTconv network with 50% weight sparsity and 2 bits of weight precision. The low or no compression benefits at intermediate sparsity levels (50-70%) are specially concerning for compact networks like MobileNet [101] which cannot be pruned beyond those levels without causing a huge drop in accuracy.

7.3.2 Storage Breakdown in Sparse Formats

Next, we study the variation in the storage requirements of the individual components, *i.e.*, the non-zero locations and non-zero values, in the different sparse formats.

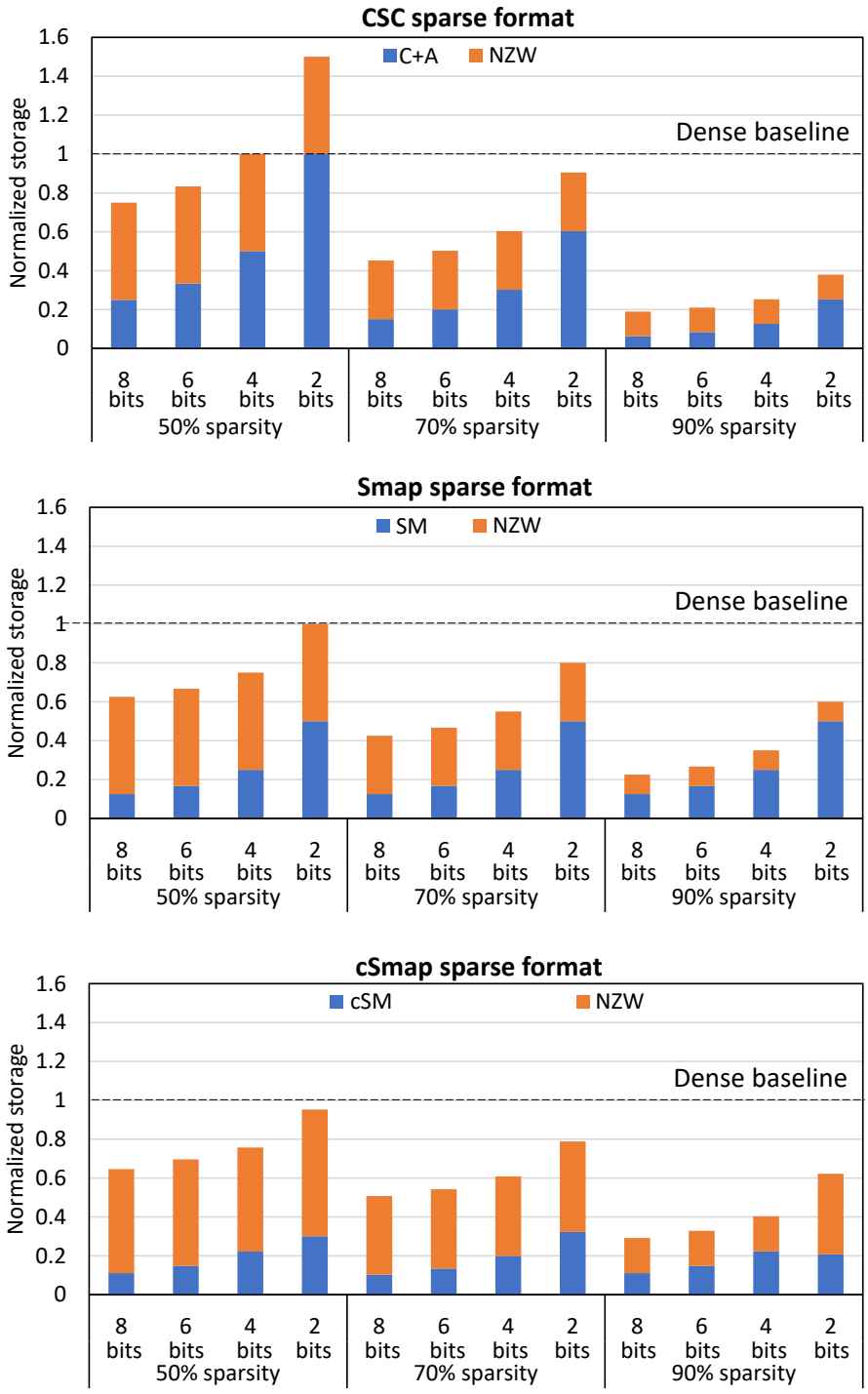

Fig. 7.4.: Storage breakdown in different sparse formats

Figure 7.4 plots the normalized storage fraction of these components with respect to a dense baseline of the same weight precision across different pruned versions of the WordLangModel benchmark. The non-zero location component, shown in blue, corresponds to different data-structures in each format. Across all formats, we observe that the location storage fraction increases with decreasing precision for a fixed sparsity level. However, the individual values of this fraction, as well as its rate of increase, is a strong function of the sparse format, with the cSmap format exhibiting the minimum value. For the same precision level, the location fraction decreases with increasing sparsity in the CSC format but remains constant in the Smap format. It also remains constant in the cSmap format for identical hyperparameters in the test compression process. For the non-zero value component, we observe that its fraction in both the CSC and Smap formats remains constant across all precisions for a constant sparsity level, while decreasing with increasing sparsity levels because of lower number of non-zeros. The value fraction in the cSmap format, on the other hand, is higher than that in the other two formats across all sparsity levels and precisions due to the introduction of some additional non-zeros in the lossy SM compression with the test pattern compression tool.

7.3.3 Benefits of a Hybrid Compression Scheme

As evident from the previous subsection, the compression ratios of the 3 sparse formats exhibit different trends across sparsity and precision levels. This in turn leads to a variation in the best performing sparse format across networks, as well as, layers within a network. We thus propose a hybrid compression scheme to overcome this limitation by identifying the optimal format at a layer or network granularity based on its sparsity and precision level. In this subsection, we discuss the advantages of such a hybrid scheme.

Figure 7.5 illustrates the inter-layer variation in the best performing sparse format for the AlexNet network exhibiting an average sparsity of 90%. The individual

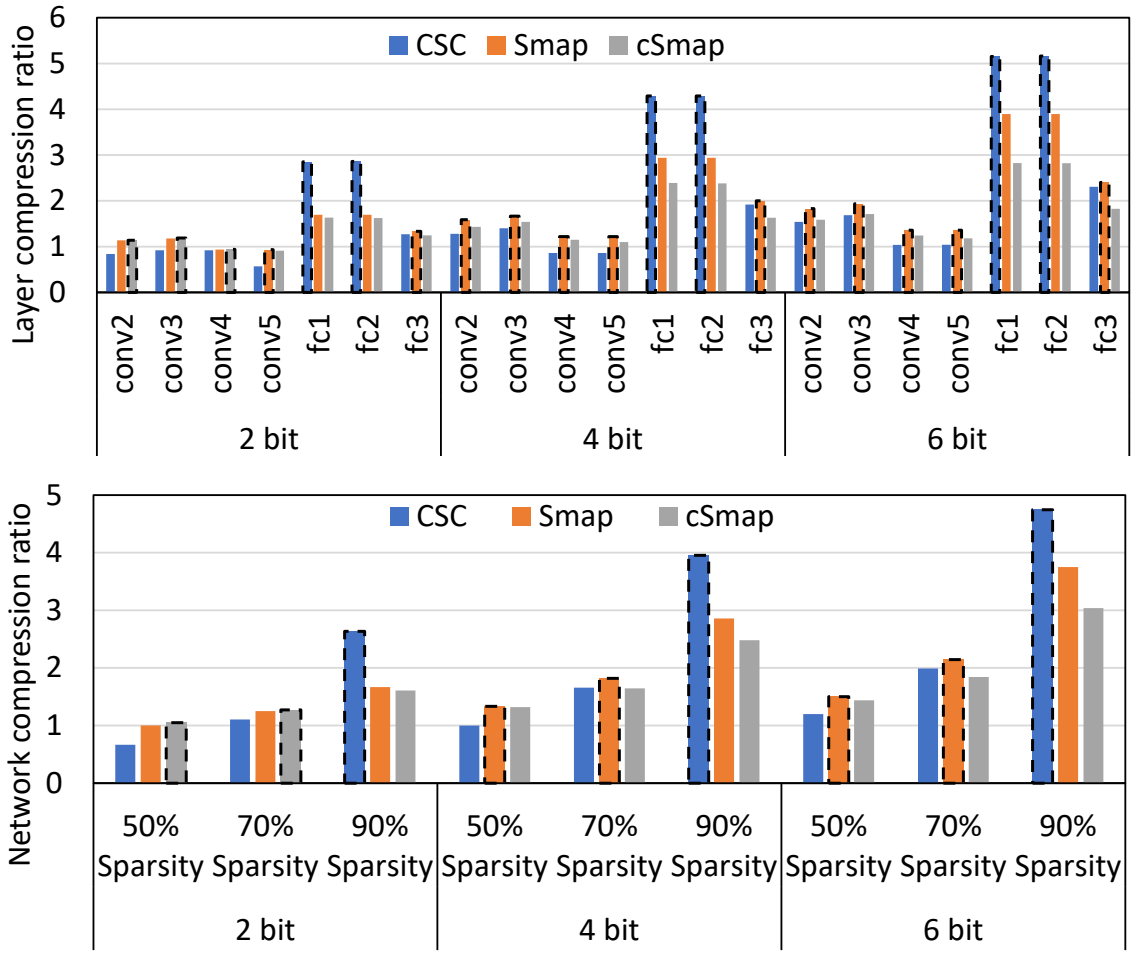

Fig. 7.5.: Layer-level and network-level variation in best performing sparse formats

layers in this network are pruned to different sparsity levels based on their relative importance in the overall recognition task. At 2 bit precision, we observe that the cSmap format achieves the highest compression for the conv2, conv3 and conv4 layers, while the Smap format achieves the highest compression for the conv5 and fc3 layers, and the CSC format achieves the highest compression for the fc1 and fc2 layers. By selecting these best performing formats in each layer, the hybrid scheme increases the average compression ratio by 14.5%-36.6% over homogeneous schemes. Figure 7.5 highlights the inter-network variation in the best performing sparse format across different pruned versions of the WordLangModel benchmark. We again observe that

different formats among Smap, CSC and cSmap, emerge as the best performing format in different scenarios. The hybrid scheme selects the best format in each of these scenarios, increasing the average compression ratio across 2-bit networks by 18.3%, 34.7% and 34.2% over the CSC, Smap and cSmap format, respectively.

7.4 Summary

The high computational and memory demands of DNNs impede their efficient execution on different resource-constrained platforms. Quantization and pruning have emerged as two of the most widely-used approaches to address these demands. In this work, we identify the challenges in effectively combining them, specially as each of them gets pushed to its limits. We observe that as we go into the domain of ultra-low precision (sub-8 bit) DNNs, the efficacy of pruning reduces due to the increasing overhead of storing non-zero locations in different sparse encoding schemes. Our results across multiple pruned and quantized versions of 6 state-of-the-art DNNs reveal that the compression benefits are a strong function of the pruning level and the sparse encoding scheme, and can even drop below 1 in the worst-case, rendering pruning ineffective. However, a hybrid compression scheme, proposed in this work, can exploit the variation across sparse encodings to identify the optimal format in a given scenario and improve the average compression benefits over homogeneous encoding schemes, across a range of DNNs.

8. EMPIR: ENSEMBLES OF MIXED PRECISION DEEP NETWORKS FOR INCREASED ROBUSTNESS AGAINST ADVERSARIAL ATTACKS

As discussed in Chapter 1, the concerns about the high computational and memory demands of DNNs has also been accompanied by concerns regarding their lack of robustness. Robustness, i.e., the ability to cope with erroneous or malicious inputs fed to an application, is a key requirement in safety-critical applications of DNNs like autonomous cars, unmanned aerial vehicles and healthcare, wherein errors (mis-classifications) made by DNNs can lead to severe — in the extreme case, fatal — consequences.

Several efforts have in fact shown that DNNs behave in unexpected and incorrect ways for small, specifically designed input perturbations [31]. An attacker can take advantage of this behavior to intentionally modify the inputs in a manner that forces the DNN model to mis-classify, and the overall system that uses the DNN to fail. A variety of methods for launching adversarial attacks on DNNs have been proposed over the years. These adversarial attacks systematically modify a given original input to cause a misclassification while keeping the input distortion minimal. A few examples of adversarial attacks that have been successfully applied to various DNN models are the Fast Gradient Sign Method (FGSM) [31], Jacobian-based Saliency Map Attack (JSMA) [50], Carlini-Wagner (CW) [51] and the Basic Iterative Method (BIM) [33], which have been discussed in greater detail in Chapter 3.

Prior works have tried to overcome these vulnerabilities by proposing various defense mechanisms against adversarial attacks. Adversarial training [31], defensive distillation [35] and input gradient regularization [36] are a few representative defense techniques. Each of these approaches, albeit promising, has limitations with respect to the kind of attacks they can defend against, the increase in training complexity,

as well as their effect on the model's accuracy on the original unperturbed inputs. To address these shortcomings, we propose EMPIR, an ensemble of mixed precision DNN models, as a new form of defense against adversarial attacks and demonstrate that it can significantly improve the robustness of a variety of DNN models across a wide range of adversarial attacks.

Ensembles have been widely explored as an approach to improve the performance of machine learning models and classifiers [135]. Examples of various successful ensembling methods include averaging, bagging [136], boosting [137], *etc.* Recently, it has also been suggested that ensembles may help boost the robustness of DNNs [53,76–78]. The individual models in these ensembles are restricted to full precision DNN models, *i.e.*, models utilizing 32 bits of numerical precision to represent different data-structures. Such ensembles are very expensive in terms of the computational and memory overhead (*e.g.*, 10× the baseline for an ensemble with 10 models [53]). In contrast, the use of quantized models in EMPIR, which entail the use of significantly lower number of bits in storage and compute, ensures that the overhead is modest (less than 25% in our evaluations).

Quantized DNNs are characterized by the use of lower numbers of bits to represent DNN data-structures like weights and activations [37–39,104]. They have been widely explored as an approach to reduce the high computational and memory demands of DNNs. Recent studies have also observed that these quantized models demonstrate higher robustness to adversarial attacks [40–42]. However, the loss in information associated with the quantization process often makes these quantized models perform significantly worse than their full-precision counterparts while classifying the original unperturbed inputs. This motivates the design of EMPIR, which successfully combines the higher robustness of low-precision models with the higher unperturbed accuracy of the full-precision models. In the general case, EMPIR comprises of M full-precision models and N low-precision models with the final prediction determined by an ensembling technique such as averaging the probabilities or counting the num-

ber of predictions for each class. In practice, we find that $M = 1$ and $N = 2$ or 3 provides a significant improvement in adversarial accuracy with small overheads.

In summary, the key contributions of this work are

- We propose the use of ensembles of mixed precision models as a defense against adversarial attacks on DNNs.
- We analyze the effect of ensemble size and ensembling techniques on the overall robustness as well as the computational and storage overheads of the ensemble.
- Across a suite of 3 different DNN models under 4 different adversarial attacks, we demonstrate that EMPIR exhibits significantly higher robustness when compared to individual models as well as ensembles of full-precision models.

8.1 EMPIR: Ensembles of Mixed Precision Deep Networks for Increased Robustness against Adversarial Attacks

To improve the robustness of DNN models, we propose EMPIR, or ensembles of mixed precision models. In this section, we will detail the design of EMPIR models and discuss the overheads associated with them.

8.1.1 Adversarial Robustness of Low-Precision Networks

DNNs have conventionally been designed as full precision models utilizing 32-bit floating point numbers to represent different data-structures like weights, activations and errors. However, the high compute and memory demands of these full-precision models have driven efforts to move towards quantized or low-precision DNNs [37–39, 103, 104]. A multitude of quantization schemes have been proposed to minimize the loss of information associated with the quantization process. While our proposal is agnostic to the quantization method used, for the purpose of demonstration we adopt the quantization scheme proposed in DoReFaNet [38], which has been shown

to produce low-precision models with competitive accuracy values. The quantization scheme can be described by Equation 8.1.

$$\begin{aligned} quantize_k(x) &= \frac{1}{2^k - 1} round((2^k - 1) \cdot x) \\ w_k = 2 \cdot quantize_k(&\frac{\tanh(w)}{2 \cdot max(|\tanh(w)|)} + \frac{1}{2}) - 1, \quad a_k = quantize_k(a) \end{aligned} \tag{8.1}$$

where k refers to the number of quantization bits in the low precision network, w and w_k refer to weight values before and after quantization, and a and a_k refer to activation values before and after quantization.

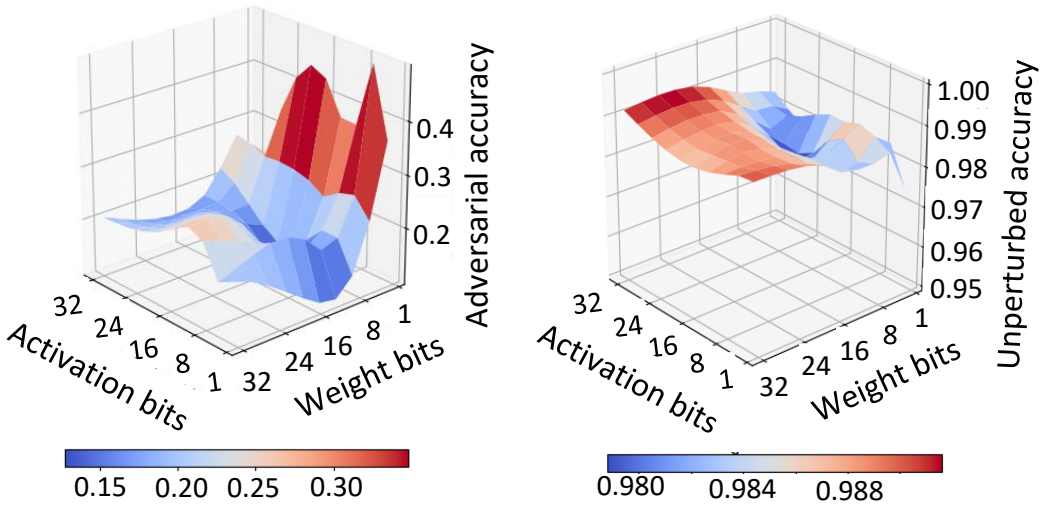

Fig. 8.1.: Unperturbed accuracies and adversarial accuracies of low-precision models trained for the MNIST dataset

In addition to the widely known advantages of reduced model size and reduced complexity of arithmetic computations, recent research efforts have also brought to light another lesser known advantage of low-precision models in the form of increased robustness to adversarial attacks. It has been observed that low-precision models in general exhibit higher values of adversarial accuracy than full-precision models with identical network structures [40, 41]. One possible explanation for this property is that higher quantization introduces higher amounts of non-linearity, which prevents

small changes in the input from drastically altering the output and forcing a misclassification [40]. Figure 8.1 shows the adversarial accuracies of different low precision models trained on the MNIST dataset under the FGSM attack. Unlike the activations and weights, the gradients utilized in the attack generation process were not quantized, allowing the adversary to launch a stronger attack. From the figure, it is apparent that models with lower numbers of bits used for representing weights and activations exhibit significantly higher levels of adversarial accuracy.

However, increasing the robustness of a system by simply replacing the full-precision model with its low-precision variant can negatively impact its accuracy on the original unperturbed inputs (unperturbed accuracy). In other words, the model may now start to mis-classify inputs that were not adversarially perturbed. Figure 8.1 also shows the unperturbed accuracies of low-precision models. As expected, models with weights and activations represented using lower numbers of bits exhibit lower unperturbed accuracies.

Based on the above observations, we propose the use of ensembles of mixed-precision models to achieve the best of both worlds, *i.e.*, increase robustness against adversarial attacks without sacrificing the accuracy on unperturbed inputs.

8.1.2 EMPIR: Overview

Figure 8.2 presents an overview of EMPIR. In the general case, an EMPIR model comprises of M full-precision (FP) models and N low-precision (LP) models. The full-precision models help in boosting the unperturbed accuracy of the overall model, while the low-precision models contribute towards higher robustness. All the individual models are fed the same input and their predicted classes or probabilities are combined with the help of an ensembling technique at the end to determine the final prediction of the EMPIR model. In practice, we found that a single full-precision model (M=1) and a small number of low-precision models (N=2 or 3) are sufficient to achieve

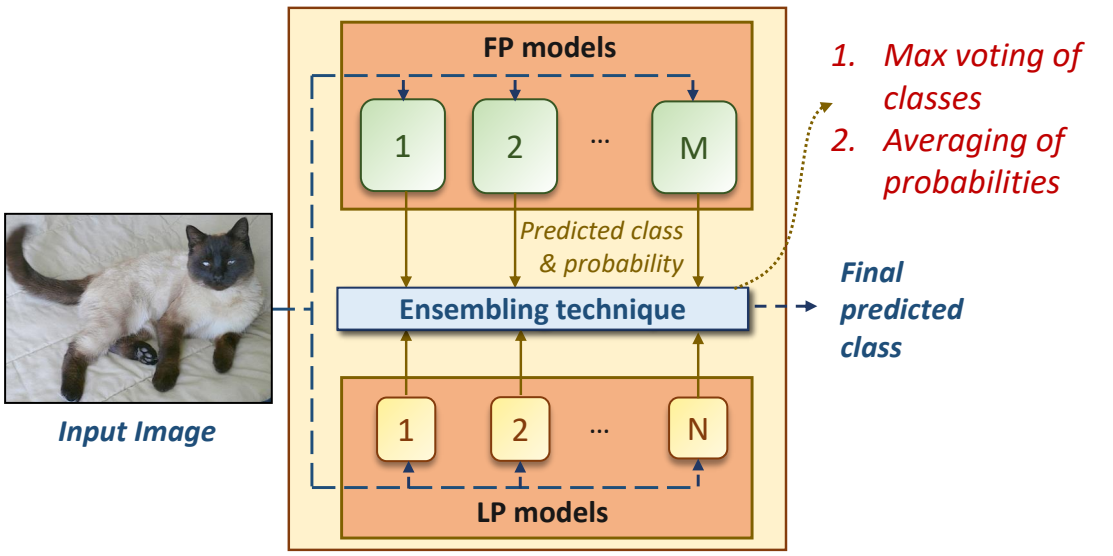

Fig. 8.2.: Overview of EMPIR

high adversarial accuracies without any noticeable compromises in the unperturbed accuracies.

The ensembling function plays a vital role in the overall performance of the model as it determines the final classification boundary. In this work, we consider two of the most commonly used ensembling functions, namely, averaging and max voting. The averaging function averages the output probabilities of each model and identifies the class with the maximum average probability as the final predicted class. On the other hand, max voting considers the predictions of each model as votes and determines the class with the maximum number of votes to be the final class. In our experiments, we found that averaging achieves better adversarial accuracies on ensembles of size 2 while max voting achieves better adversarial accuracies on ensembles of size greater than 2.

In order to allow an ensemble model to work better than a single model, the individual models should also be designed to be diverse [135]. This ensures that the models dont produce similar errors and hence, that the probability of two models

misclassifying the same input is lower. We introduce diversity in the individual models of EMPIR by training them with different random initializations of weights.

8.1.3 Computational and Memory Complexity of EMPIR

The ensembling of multiple full-precision and low-precision models in EMPIR increases its computational and storage requirements as these models need to be stored and evaluated. In this work, we keep these memory and computational complexities within reasonable limits by restricting the precision of weights and activations in the low-precision models of EMPIR to a maximum of 4 bits.

The increasing popularity of low-precision DNN models has prompted recent hardware platforms including GPUs and neural network accelerators to add native hardware support for operating on low precision data [13, 129]. These hardware platforms reconfigure a common datapath to perform computations on full-precision data (32 or 64 bits) as well as low-precision data (4, 8 or 16 bits). Low-precision operations can achieve higher throughputs than full-precision operations on these platforms as the same number of compute elements and the same amount of memory bandwidth can support a larger number of concurrent operations. Consequently, the additional execution time required to evaluate the low-precision models in EMPIR is much less than that of a full-precision model. Overall, we quantify the execution time and storage overhead of an EMPIR model using the formula described by Equation 8.2.

$$\begin{aligned} TimeOverhead\{EMPIR(M,N)\} &= M + \sum_{i=1}^{N} \frac{Ops\_per\_sec(FP)}{Ops\_per\_sec(k_i)} \\ StorageOverhead\{EMPIR(M,N)\} &= M + \sum_{i=1}^{N} \frac{k_i}{FP} \end{aligned} \tag{8.2}$$

where k_i is the precision of the i^{th} low-precision model, FP is the precision of the full-precision models, and $Ops\_per\_sec(b)$ is the throughput of b bit operations on the underlying hardware platform.

8.2 Experiments

In this section, we describe the experiments performed to evaluate the advantages of EMPIR models over baseline full-precision models.

8.2.1 Benchmarks

We studied the robustness of EMPIR models across three different image recognition DNNs, namely, MNISTconv, CIFARconv and AlexNet. The individual full-precision and-low precision networks within the EMPIR models were designed to have identical network topologies. The details of the individual networks in these benchmarks are listed in Table 8.1. The benchmarks differ in the number of convolutional layers, fully connected layers as well as the datasets. We consider three different datasets, namely, MNIST [138], CIFAR-10 [116] and ImageNet [139] which vary significantly in their complexity. The low precision networks were obtained using the quantization scheme proposed in DoReFa-Net [38]. The full precision models were trained using 32 bit floating point representations for all data-structures.

8.2.2 Evaluation of robustness

We implemented EMPIR within TensorFlow [140] and have released the source code for our implementation [1]. The robustness of the EMPIR models was measured in terms of their adversarial accuracies under a variety of white-box attacks within the Cleverhans library [141]. We specifically consider the four adversarial attacks described in Section 3.6. The adversarial parameters for the attacks on the different benchmarks are presented in Table 8.2. The attacks were generated on the entire test dataset for each of the benchmarks. Generating these white-box attacks involves computation of the gradient $\nabla_x L(\theta, X, Y)$ (Section 3.6), which is not directly defined for ensembles. For the EMPIR models, we compute this gradient as an average over

[1] `https://github.com/sancharisen/EMPIR`

Table 8.1.: Benchmarks

Network	Dataset	Configuration
MNISTconv	MNIST	Conv(8×8×64), ReLU, Conv(6×6×128), ReLU, Conv(5×5×128), ReLU, Fully_Connected(10), SoftMax
CIFARconv	CIFAR-10	Conv(5×5×32), ReLU, MaxPool(3×3), Conv(8×8×64), ReLU, AvgPool(3×3), Conv(8×8×64), ReLU, AvgPool(3×3), Fully_Connected(64), Fully_Connected(10), SoftMax
AlexNet	ImageNet	Conv(12×12×96), ReLU, Conv(5×5×256), BatchNorm, ReLU, MaxPool(3×3), Conv(3×3×384), BatchNorm, ReLU, MaxPool(3×3), Conv(3×3×384), BatchNorm, ReLU, Conv(3×3×256), BatchNorm, ReLU, MaxPool(3×3), Fully_Conn(4096), BatchNorm, ReLU, Fully_Conn(4096), BatchNorm, ReLU, Fully_Conn(1000), SoftMax

Table 8.2.: Attack parameters

Network	CW	FGSM	BIM	PGD
MNISTconv	Attack iterations = 50	ϵ= 0.3	ϵ= 0.3, α=0.01 No. of iterations = 40	ϵ= 0.3, α=0.01 No. of iterations = 40
CIFARconv	Attack iterations = 50	ϵ= 0.1	ϵ= 0.1, α=0.01 No. of iterations = 40	ϵ= 0.1, α=0.01 No. of iterations = 40
AlexNet	Attack iterations = 50	ϵ= 0.1	ϵ= 0.1, α=0.01 No. of iterations = 40	ϵ= 0.1, α=0.01 No. of iterations = 5

all individual models for an averaging ensemble and as an average over the individual models that voted for the final identified class for a max-voting ensemble.

8.3 Results

In this section, we present the results of our experiments highlighting the advantages of EMPIR models.

8.3.1 Robustness of EMPIR models across all attacks

Tables 8.3, 8.4 and 8.5 present the results of our experiments across the MNIST-conv, CIFARconv and AlexNet benchmarks, respectively. The EMPIR models presented in the tables are the ones exhibiting highest average adversarial accuracies under the constraints of <25% compute and memory overhead and <2% loss in unper-

Table 8.3.: MNISTconv: Unperturbed and adversarial accuracies of the baseline and EMPIR models across different attacks

	Unperturbed	**Adversarial Accuracy (%)**				
Approach	**Accuracy (%)**	**CW**	**FGSM**	**BIM**	**PGD**	**Average**
Baseline FP	98.87	3.69	14.32	0.9	0.77	4.92
EMPIR	**98.89**	**86.73**	**67.06**	**18.61**	**17.51**	**47.48**
Defensive Distill.	98.12	2.34	40.22	7.61	3.28	13.36
Inp. Grad. Reg.	99.01	6.83	30.15	1.14	1.20	9.83
FGSM Adv. Train	99.06	3.09	76.56	0.87	0.39	20.23
EMPIR (FGSM Adv. Train)	**99.09**	**90.54**	**75.98**	**33.16**	**5.17**	**51.21**

turbed accuracy. We observed that across all the benchmarks, ensembles comprised of two low-precision and one full-precision model combined with the max-voting ensembling technique satisfy these constraints. However, the individual configurations of the low-precision models, *i.e.*, the precisions of weights and activations in the ensembles, differ across the benchmarks. For example, both low-precision models in the EMPIR model for MNISTconv have weight precisions of 2 bits and activation precisions of 4 bits. On the other hand, the two low-precision models in the AlexNet EMPIR model have {weight,activation} bit-precisions of {2,2} and {4,4}, respectively. In general, we observe that the EMPIR models exhibit substantially higher adversarial accuracies across all attacks for the three benchmarks.

We also compare the benefits of EMPIR with four other popular approaches for increasing robustness, namely, defensive distillation [35], input gradient regularization [36] , FGSM based adversarial training [31] and PGD based adversarial train-

Table 8.4.: CIFARconv: Unperturbed and adversarial accuracies of the baseline and EMPIR models across different attacks

	Unperturbed	**Adversarial Accuracy (%)**				
Approach	**Accuracy (%)**	**CW**	**FGSM**	**BIM**	**PGD**	**Average**
Baseline FP	74.54	13.38	10.28	11.97	10.69	11.58
EMPIR	**72.56**	**48.51**	**20.45**	**24.59**	**13.55**	**26.78**
FGSM Adv. Train	72.36	14.36	41.58	12.92	11.24	20.03
EMPIR (FGSM Adv. Train)	**73.62**	**45.73**	**31.67**	**29.55**	**14.74**	**30.42**
PGD Adv. Train	73.55	12.62	12.45	10.97	8.52	11.14

Table 8.5.: AlexNet: Unperturbed and adversarial accuracies of the baseline and EMPIR models across different attacks

	Unperturbed	**Adversarial Accuracy (%)**				
Approach	**Accuracy (%)**	**CW**	**FGSM**	**BIM**	**PGD**	**Average**
Baseline FP	53.23	9.94	10.29	10.81	10.30	10.34
EMPIR	**55.09**	**29.36**	**21.65**	**20.67**	**11.76**	**20.86**

ing [32]. The distillation process was implemented with a softmax temperature of $T = 100$, the gradient regularization was realized with a regularization penalty of $\lambda = 100$, while the adversarial training mechanisms utilized adversarial examples generated with a maximum possible perturbation of $\epsilon = 0.3$. Tables 8.3 and 8.4 present the results for the approaches that were able to achieve <5% loss in unper-

turbed accuracy for the MNISTconv and CIFARconv benchmarks respectively. We observe that FGSM based adversarial training significantly boosts the adversarial accuracies of the MNISTconv and CIFARconv models under the FGSM attack but is unable to increase the accuracies under the other three attacks, often hurting them in the process. A similar result is observed for the MNISTconv model trained with defensive distillation and gradient regularization. In contrast, EMPIR successfully increases the robustness of the models under all four attacks. In fact, it can even be combined with the other approaches to further boost the robustness, as evident from the adversarial accuracies of an EMPIR model comprising of adversarially trained models for the MNISTconv and CIFARconv benchmarks. EMPIR also achieves a higher adversarial accuracy than PGD based adversarial training for the CIFARconv benchmark. Overall, EMPIR increases robustness with zero training overhead, as opposed to considerable training overheads associated with the other defense strategies like adversarial training, defensive distillation and input gradient regularization.

8.3.2 Comparison with individual models

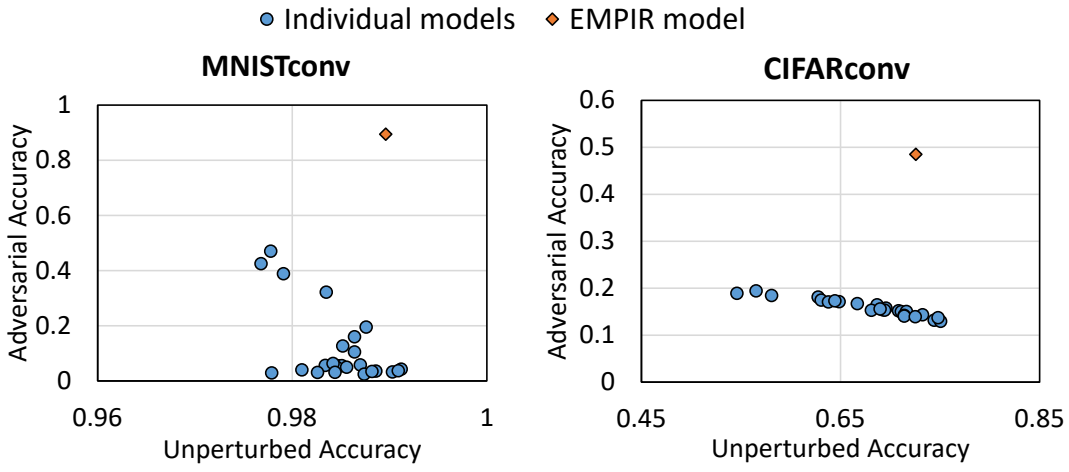

Fig. 8.3.: Tradeoff between unperturbed and adversarial accuracies of the individual and EMPIR models across 2 benchmarks.

Figure 8.3 illustrates the tradeoff between the adversarial and unperturbed accuracies of the individual DNN models and EMPIR models for two of the benchmarks under the CW attack. The circular blue points correspond to individual models with varying weight and activation precisions while the red diamond points correspond to the EMPIR models presented in Section 8.3.1. The figure clearly indicates that the EMPIR models in both the benchmarks are notably closer to the desirable top right corner with high unperturbed as well as high adversarial accuracies. Among the individual models, the ones demonstrating higher adversarial accuracies but lower unperturbed accuracies (towards the top left corner) correspond to lower activation and weight precisions while those demonstrating lower adversarial accuracies and higher unperturbed accuracies (towards the bottom right corner) correspond to higher activation and weight precisions.

8.3.3 Analysis of confusion matrices

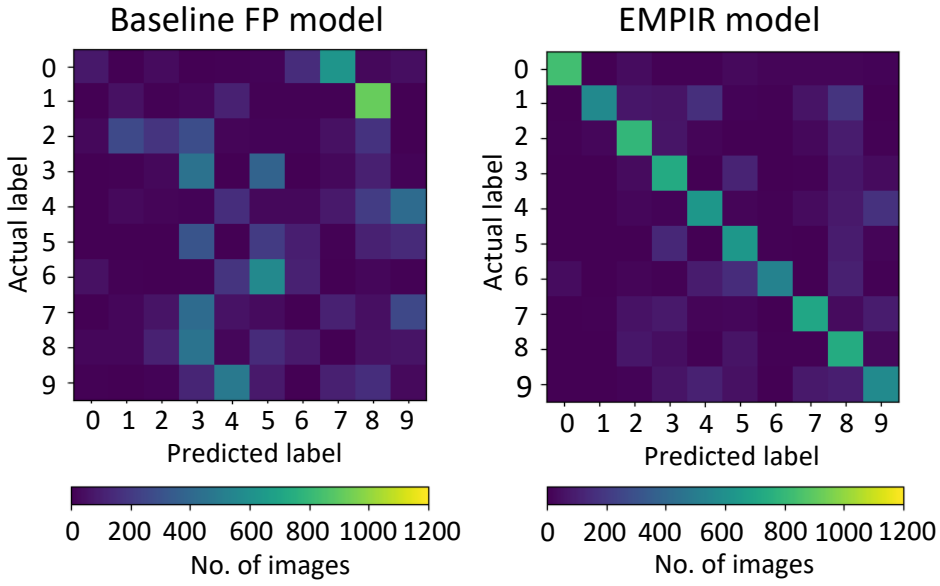

Fig. 8.4.: Confusion matrices of the baseline FP and EMPIR model for the MNISTconv benchmark.

Figure 8.4 presents the confusion matrices of the baseline FP model and the EMPIR model for the MNISTconv benchmark under the FGSM attack. The actual ground truth class labels are listed vertically while the predicted labels are listed horizontally. The colors represent the number of images in the test dataset that correspond to the combination of actual and predicted class labels. The diagonal nature of EMPIR's confusion matrix clearly illustrates its superiority over the FP model, which frequently misclassifies the generated adversarial images.

8.3.4 Impact of varying the number of low-precision and full-precision models

In this subsection, we vary the number of low-precision and full-precision models in EMPIR between 0 and 3 to observe its effect on the unperturbed and adversarial accuracies of the MNISTconv benchmark under the FGSM attack. We also measure the execution time and memory footprint of the EMPIR models to quantify their overheads with respect to a baseline single full-precision model. We restrict the low-precision models to have weight and activation precisions between 2 and 4 bits and choose the configurations that maximize the adversarial accuracies of the EMPIR models while introducing <1% drop in unperturbed accuracies.

Figure 8.5 presents the results of this experiment. Figure 8.5(a) and (b) clearly indicates that a higher number of low-precision models in EMPIR helps in boosting the adversarial accuracies while a higher number of full-precision models help in boosting the unperturbed accuracies. For instance, an EMPIR model comprising of only three low-precision models demonstrates unperturbed and adversarial accuracies of 98.8% and 56.9% respectively while an EMPIR model comprising of only three full-precision models demonstrates unperturbed and adversarial accuracies of 99.2% and 31%, respectively. The execution time and memory footprint associated with the former are only 0.38× and 0.25× over the baseline, as opposed to 3× in case of the latter. Overall, we observe that an EMPIR model comprising of a single full-precision model and two

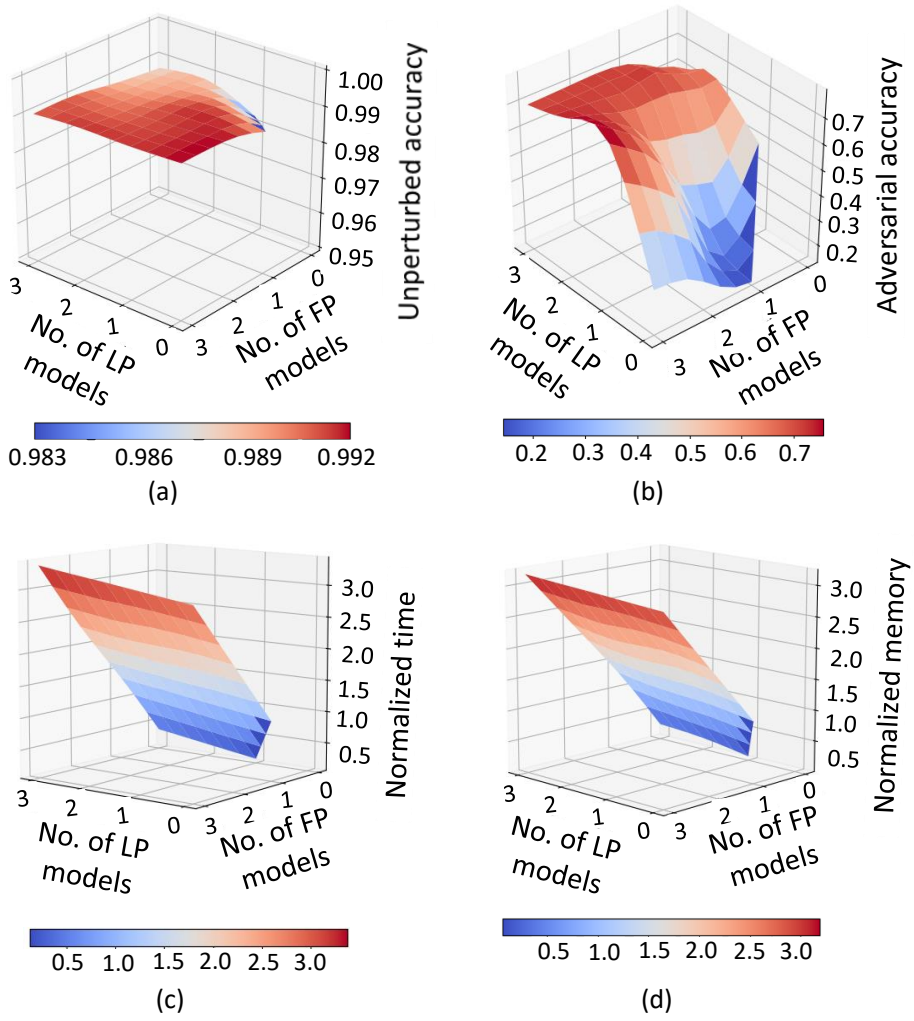

Fig. 8.5.: Effects of varying the number of LP and FP models in EMPIR (a) Unperturbed accuracies, (b) Adversarial accuracies, (c) Execution time overheads and (d) Storage overheads

low-precision models (configuration presented in Tables 8.3, 8.4 and 8.5) achieves a good balance between adversarial and unperturbed accuracies with modest execution time and storage overheads.

8.4 Summary

As deep neural networks get deployed in applications with stricter safety requirements, there is a dire need to identify new approaches that make them more robust to adversarial attacks. In this work, we boost the robustness of DNNs by designing ensembles of mixed-precision DNNs. In its most generic form, EMPIR comprises of M full-precision DNNs and N low-precision DNNs combined through ensembling techniques like max voting or averaging. EMPIR combines the higher robustness of low-precision DNNs with the higher unperturbed accuracies of the full-precision models. Our experiments on 3 different image recognition benchmarks under 4 different adversarial attacks reveal that EMPIR is able to significantly increase the robustness of DNNs without sacrificing the accuracies of the models on unperturbed inputs.

9. CONCLUSION

Deep neural networks have succeeded in greatly advancing the state-of-the-art in a large number of machine-learning tasks. Their success has in turn contributed to a growing interest in deploying them in different real-world applications and services. However, any major initiative on that front has been plagued by two notable limitations of DNNs, namely, their high computational and memory demands, and their lack of robustness.

The high computational and memory demands of neural networks was one of the factors responsible for their limited success even after multiple decades of research efforts, post their conception in the 1950s. The development of powerful computational platforms like Graphics Processing Units (GPUs) and specialized accelerators, over the past decade, has helped alleviate a part of the challenge. But the strive for continuously improving DNN accuracies has propelled the design of even larger networks with higher compute and memory footprints. Thus, there is a pressing need to come up with alternative techniques for addressing the computational challenge of DNNs.

The lack of robustness in DNNs, on the other hand, has been highlighted by numerous efforts on adversarial attacks which force DNN mis-classifications through small, human-imperceptible input perturbations. The success of adversarial attacks has raised concerns on the deployment of DNNs in different safety-critical applications where such erroneous or malicious inputs need to be processed safely. Thus, there is also an urgent need to develop techniques for boosting the robustness of DNNs.

9.1 Thesis Summary

In this dissertation, we propose the use of approximate computing for improving both the efficiency and robustness of DNNs. We identify opportunities for approximations in multiple classes of DNNs that exploit their unique computational and connectivity patterns to achieve energy and execution time savings. We also develop approximate computing approaches for boosting robustness of DNNs without sacrificing their accuracy on the unperturbed inputs. The primary contributions of the dissertation are summarized below.

- The thesis explores micro-architectural and instruction set extensions extensions in the form of SparCE for accelerating FFNNs on general-purpose processor cores. The proposed sparsity-aware core extensions exploit a key attribute of FFNNs, *viz.*, sparsity, by dynamically identifying zero values loaded into the processor pipeline and skipping subsequent instructions rendered redundant by them. The lightweight and minimally intrusive nature of the extensions establishes SparCE as a promising approach for accelerating DNNs on various area and cost-constrained devices that can't accommodate specialized DNN accelerators.

- Next, the thesis explores hardware-agnostic approximations for long-short term memory networks. The proposed AxLSTM technique exploits the timestep-driven computation in LSTMs to dynamically skip input symbols with little or no impact on the cell state and modulate the size of the cell state based on the complexity of the input sequence. We apply AxLSTM to sequence-to-sequence learning, one of the most common application of LSTMs, and demonstrate execution time savings on a general-purpose processor.

- The thesis also develops AxSNN, a set of approximate computing techniques for spiking neural networks that take advantage of the spike-triggered nature of computations in SNNs. AxSNN identifies appropriate approximation modes

for each neuron in the network based on its static and dynamic properties, and determines a suitable subset of spike-triggered updates that can skipped in each mode. We apply AxSNN to both software and hardware implementations of SNNs and demonstrate significant energy and execution time savings with minimal loss in quality.

- The thesis explores the opportunities for combining two popular approximate computing techniques — pruning and quantization — in DNNs, particularly as each gets pushed to its limits. It specifically finds that the efficacy of pruning, in reducing memory requirements of DNNs, diminishes in the ultra-low precision regime. This is because of the increasing overhead of non-zero locations in different sparse coding schemes utilized for storing the pruned networks. By observing the variation in compression ratios of three different sparse formats, we further propose a hybrid compression scheme that identifies the optimal sparse format for a network or a layer based on its sparsity and precision value.

- Finally, the thesis proposes ensembles of mixed-precision DNNs, EMPIR, as a new approach for improving the robustness of DNNs to different adversarial attacks. EMPIR combines the higher adversarial accuracies of low-precision models with the higher unperturbed accuracies of the full-precision models to achieve the "best of the both worlds". By utilizing low-precision models in the ensemble, EMPIR also ensures that the memory and compute overheads of the ensemble are modest ($<25\%$ in our evaluations).

REFERENCES

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[2] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. [Online]. Available: http://arxiv.org/abs/1409.4842

[3] S. Venugopalan, M. Rohrbach, J. Donahue, R. J. Mooney, T. Darrell, and K. Saenko, "Sequence to sequence - video to text," in *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, 2015, pp. 4534–4542. [Online]. Available: http://dx.doi.org/10.1109/ICCV.2015.515

[4] A. Y. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, and A. Y. Ng, "Deep speech: Scaling up end-to-end speech recognition," *CoRR*, vol. abs/1412.5567, 2014. [Online]. Available: http://arxiv.org/abs/1412.5567

[5] X. Zhang and Y. LeCun, "Text understanding from scratch," *CoRR*, vol. abs/1502.01710, 2015. [Online]. Available: http://arxiv.org/abs/1502.01710

[6] G. Hinton, L. Deng, D. Yu, G. Dahl, A. rahman Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition," *Signal Processing Magazine*, 2012.

[7] J. Li, M. Galley, C. Brockett, G. P. Spithourakis, J. Gao, and B. Dolan, "A persona-based neural conversation model," March 2016. [Online]. Available: https://www.microsoft.com/en-us/research/publication/persona-based-neural-conversation-model/

[8] "Deep learning boosts google translate tool," *Nature News*, 2016. [Online]. Available: https://www.nature.com/news/deep-learning-boosts-google-translate-tool-1.20696

[9] "Updating google maps with deep learning and street view," *[online] Google AI Blog*, May 2017. [Online]. Available: https://ai.googleblog.com/2017/05/updating-google-maps-with-deep-learning.html

[10] "Apple is turning siri into a next-level artificial intelligence," *Mashable*, 2016. [Online]. Available: http://mashable.com/2016/06/13/siri-sirikit-wwdc2016-analysis/hLMSxZKVnEqO

[11] "Improving photo search: A step across the semantic gap," *Google Research blog*, 2013.

[12] "How amazon and netflix are winning the personalization battle," *MarTech Advisor*, 2016. [Online]. Available: https://www.martechadvisor.com/articles/customer-experience-2/recommendation-engines-how-amazon-and-netflix-are-winning-the-personalization-battle/

[13] "The Intelligent Industrial Revolution — NVIDIA Blog." *[online] NVIDIA Blog*, 2017. [Online]. Available: https://blogs.nvidia.com/blog/2016/10/24/intelligent-industrial-revolution/

[14] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *AAAI*, 2019.

[15] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *CoRR*, vol. abs/1404.5997, 2014. [Online]. Available: http://arxiv.org/abs/1404.5997

[16] F. N. Iandola, K. Ashraf, M. W. Moskewicz, and K. Keutzer, "FireCaffe: near-linear acceleration of deep neural network training on compute clusters," *CoRR*, vol. abs/1511.00175, 2015. [Online]. Available: http://arxiv.org/abs/1511.00175

[17] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems 25*, P. Bartlett, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., 2012, pp. 1232–1240.

[18] "Neon, nervana systems: http://neon.nervanasys.com/docs/latest/index.html," 2016.

[19] B. Li, E. Zhou, B. Huang, J. Duan, Y. Wang, N. Xu, J. Zhang, and H. Yang, "Large scale recurrent neural network on GPU," in *2014 International Joint Conference on Neural Networks (IJCNN)*, July 2014, pp. 4062–4069.

[20] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," *CoRR*, vol. abs/1609.08144, 2016. [Online]. Available: http://arxiv.org/abs/1609.08144

[21] J. Devlin, "Sharp models on dull hardware: Fast and accurate neural machine translation decoding on the CPU," *CoRR*, vol. abs/1705.01991, 2017. [Online]. Available: http://arxiv.org/abs/1705.01991

[22] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 609–622. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.58

[23] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "ScaleDeep: A scalable compute architecture for learning and evaluating deep networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 13–26. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080244

[24] N. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080246

[25] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 267–278. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2016.32

[26] 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*, vol. 52, no. 1, pp. 127–138, Jan 2017.

[27] A. X. M. Chang, B. Martini, and E. Culurciello, "Recurrent neural networks hardware implementation on FPGA," *CoRR*, vol. abs/1511.05552, 2015. [Online]. Available: http://arxiv.org/abs/1511.05552

[28] M. Lee, K. Hwang, J. Park, S. Choi, S. Shin, and W. Sung, "FPGA-based low-power speech recognition with recurrent neural networks," in *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*, Oct 2016, pp. 230–235.

[29] S. Li, C. Wu, H. Li, B. Li, Y. Wang, and Q. Qiu, "FPGA acceleration of recurrent neural network based language model," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2015, pp. 111–118.

[30] Y. Guan, Z. Yuan, G. Sun, and J. Cong, "Fpga-based accelerator for long short-term memory recurrent neural networks," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2017, pp. 629–634.

[31] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and Harnessing Adversarial Examples," *arXiv e-prints*, p. arXiv:1412.6572, Dec 2014.

[32] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards Deep Learning Models Resistant to Adversarial Attacks," *arXiv e-prints*, p. arXiv:1706.06083, Jun 2017.

[33] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," *CoRR*, vol. abs/1607.02533, 2016. [Online]. Available: http://arxiv.org/abs/1607.02533

[34] A. Liu, X. Liu, C. Zhang, H. Yu, Q. Liu, and J. He, "Training robust deep neural networks via adversarial noise propagation," *ArXiv*, vol. abs/1909.09034, 2019.

[35] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a Defense to Adversarial Perturbations against Deep Neural Networks," *arXiv e-prints*, p. arXiv:1511.04508, Nov 2015.

[36] A. S. Ross and F. Doshi-Velez, "Improving the adversarial robustness and interpretability of deep neural networks by regularizing their input gradients," in *AAAI*, 2017.

[37] 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*, ser. ISLPED '14. New York, NY, USA: ACM, 2014, pp. 27–32. [Online]. Available: http://doi.acm.org/10.1145/2627369.2627613

[38] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *CoRR*, vol. abs/1606.06160, 2016. [Online]. Available: http://arxiv.org/abs/1606.06160

[39] M. Courbariaux, Y. Bengio, and J. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," *CoRR*, vol. abs/1511.00363, 2015. [Online]. Available: http://arxiv.org/abs/1511.00363

[40] A. Galloway, G. W. Taylor, and M. Moussa, "Attacking Binarized Neural Networks," *arXiv e-prints*, p. arXiv:1711.00449, Nov 2017.

[41] P. Panda, I. Chakraborty, and K. Roy, "Discretization based solutions for secure machine learning against adversarial attacks," *CoRR*, vol. abs/1902.03151, 2019. [Online]. Available: http://arxiv.org/abs/1902.03151

[42] A. Siraj Rakin, J. Yi, B. Gong, and D. Fan, "Defend Deep Neural Networks Against Adversarial Examples via Fixed and Dynamic Quantized Activation Functions," *arXiv e-prints*, p. arXiv:1807.06714, Jul 2018.

[43] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2015. [Online]. Available: http://arxiv.org/abs/1510.00149

[44] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," *CoRR*, vol. abs/1405.3866, 2014. [Online]. Available: http://arxiv.org/abs/1405.3866

[45] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. B. J. Dally, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. (FPGA), 2017, pp. 75–84. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021745

[46] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, pp. 1737–1746.

[47] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *CoRR*, vol. abs/1612.01064, 2016. [Online]. Available: http://arxiv.org/abs/1612.01064

[48] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu, "ApproxANN: An approximate computing framework for artificial neural network," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2015, pp. 701–706.

[49] Z. Du, A. Lingamneni, Y. Chen, K. V. Palem, O. Temam, and C. Wu, "Leveraging the error resilience of neural networks for designing highly energy efficient accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 8, pp. 1223–1235, Aug 2015.

[50] N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," *CoRR*, vol. abs/1511.07528, 2015. [Online]. Available: http://arxiv.org/abs/1511.07528

[51] N. Carlini and D. A. Wagner, "Towards evaluating the robustness of neural networks," *CoRR*, vol. abs/1608.04644, 2016. [Online]. Available: http://arxiv.org/abs/1608.04644

[52] N. Papernot, P. D. McDaniel, and I. J. Goodfellow, "Transferability in machine learning: from phenomena to black-box attacks using adversarial samples," *CoRR*, vol. abs/1605.07277, 2016. [Online]. Available: http://arxiv.org/abs/1605.07277

[53] T. Strauss, M. Hanselmann, A. Junginger, and H. Ulmer, "Ensemble Methods as a Defense to Adversarial Perturbations Against Deep Neural Networks," *arXiv e-prints*, p. arXiv:1709.03423, Sep 2017.

[54] C. Guo, M. Rana, M. Cissé, and L. van der Maaten, "Countering adversarial images using input transformations," *CoRR*, vol. abs/1711.00117, 2017. [Online]. Available: http://arxiv.org/abs/1711.00117

[55] J. Buckman, A. Roy, C. Raffel, and I. Goodfellow, "Thermometer encoding: One hot way to resist adversarial examples," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=S18Su--CW

[56] A. Graves, "Generating sequences with recurrent neural networks," *CoRR*, vol. abs/1308.0850, 2013. [Online]. Available: http://arxiv.org/abs/1308.0850

[57] A. Graves, A. Mohamed, and G. E. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2013, pp. 6645–6649.

[58] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14, Cambridge, MA, USA, 2014, pp. 3104–3112. [Online]. Available: http://dl.acm.org/citation.cfm?id=2969033.2969173

[59] R. Ananthanarayanan and D. S. Modha, "Anatomy of a cortical simulator," in *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, Nov 2007, pp. 1–12.

[60] A. Morrison, C. Mehring, T. Geisel, A. Aertsen, and M. Diesmann, "Advancing the boundaries of high-connectivity network simulation with distributed computing," *Neural Computation*, vol. 17, pp. 1776–1801, 2005.

[61] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau, and A. Veidenbaum, "Efficient simulation of large-scale spiking neural networks using cuda graphics processors," in *2009 International Joint Conference on Neural Networks*, June 2009, pp. 2145–2152.

[62] A. K. Fidjeland and M. P. Shanahan, "Accelerated simulation of spiking neural networks using gpus," in *The 2010 International Joint Conference on Neural Networks (IJCNN)*, July 2010, pp. 1–8.

[63] J. L. Krichmar, P. Coussy, and N. Dutt, "Large-scale spiking neural networks using neuromorphic hardware compatible models," *J. Emerg. Technol. Comput. Syst.*, vol. 11, no. 4, pp. 36:1–36:18, Apr. 2015. [Online]. Available: http://doi.acm.org/10.1145/2629509

[64] T. Schoenauer, S. Atasoy, N. Mehrtash, and H. Klar, "Neuropipe-chip: A digital neuro-processor for spiking neural networks," *IEEE Transactions on Neural Networks*, vol. 13, no. 1, pp. 205–213, Jan 2002.

[65] S. B. Furber, D. R. Lester, L. A. Plana, J. D. Garside, E. Painkras, S. Temple, and A. D. Brown, "Overview of the spinnaker system architecture," *IEEE Transactions on Computers*, vol. 62, no. 12, pp. 2454–2467, Dec 2013.

[66] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014. [Online]. Available: http://science.sciencemag.org/content/345/6197/668

[67] S. Shin, K. Hwang, and W. Sung, "Fixed-point performance analysis of recurrent neural networks," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, March 2016, pp. 976–980.

[68] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 1–13.

[69] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," *CoRR*, vol. abs/1602.01528, 2016. [Online]. Available: http://arxiv.org/abs/1602.01528

[70] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: an accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 27–40. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080254

[71] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–12.

[72] H. Park, D. Kim, J. Ahn, and S. Yoo, "Zero and data reuse-aware fast convolution for deep neural networks on GPU," in *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES '16. New York, NY, USA: ACM, 2016, pp. 33:1–33:10. [Online]. Available: http://doi.acm.org/10.1145/2968456.2968476

[73] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Penksy, "Sparse convolutional neural networks," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 806–814.

[74] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds. Curran Associates, Inc., 2016, pp. 2074–2082. [Online]. Available: http://papers.nips.cc/paper/6504-learning-structured-sparsity-in-deep-neural-networks.pdf

[75] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing DNN pruning to the underlying hardware parallelism," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 548–560. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080215

[76] T. Pang, K. Xu, C. Du, N. Chen, and J. Zhu, "Improving adversarial robustness via promoting ensemble diversity," in *ICML*, 2019.

[77] W. He, J. Wei, X. Chen, N. Carlini, and D. Song, "Adversarial example defenses: Ensembles of weak defenses are not strong," in *Proceedings of the 11th USENIX Conference on Offensive Technologies*, ser. WOOT'17. Berkeley, CA, USA: USENIX Association, 2017, pp. 15–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=3154768.3154783

[78] F. Tramèr, A. Kurakin, N. Papernot, D. Boneh, and P. D. McDaniel, "Ensemble adversarial training: Attacks and defenses," *ArXiv*, vol. abs/1705.07204, 2017.

[79] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing and the quest for computing efficiency," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[80] R. Hegde and N. R. Shanbhag, "Soft digital signal processing," *Very Large Scale Integration (VLSI) Systems, IEEE Trans. on*, vol. 9, no. 6, Dec. 2001. [Online]. Available: http://dx.doi.org/10.1109/92.974895

[81] D. Mohapatra, V. K. Chippa, A. Raghunathan, and K. Roy, "Design of voltage-scalable meta-functions for approximate computing," in *2011 Design, Automation Test in Europe*, March 2011, pp. 1–6.

[82] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality programmable vector processors for approximate computing," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2013, pp. 1–12.

[83] V. K. Chippa, D. Mohapatra, K. Roy, S. T. Chakradhar, and A. Raghunathan, "Scalable effort hardware design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 9, pp. 2004–2016, Sep. 2014.

[84] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 124–134. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025133

[85] J. Meng, S. Chakradhar, and A. Raghunathan, "Best-effort parallel execution framework for recognition and mining applications," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, ser. IPDPS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12. [Online]. Available: https://doi.org/10.1109/IPDPS.2009.5160991

[86] J. Kung, D. Kim, and S. Mukhopadhyay, "Dynamic approximation with feedback control for energy-efficient recurrent neural network hardware," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ser. (ISLPED), 2016, pp. 168–173. [Online]. Available: http://doi.acm.org/10.1145/2934583.2934626

[87] A. S. Cassidy, J. Georgiou, and A. G. Andreou, "Design of silicon brains in the nano-cmos era: Spiking neurons, learning synapses and neural architecture optimization," *Neural Networks*, vol. 45, pp. 4 – 26, 2013, neuromorphic Engineering: From Neural Systems to Brain-Like Engineered Systems. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608013001597

[88] B. V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J. Bussat, R. Alvarez-Icaza, J. V. Arthur, P. A. Merolla, and K. Boahen, "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 699–716, May 2014.

[89] A. See, M.-T. Luong, and C. D. Manning, "Compression of neural machine translation models via pruning," in *CoNLL*, 2016.

[90] F. Tung and G. Mori, "Clip-q: Deep network compression learning by in-parallel pruning-quantization," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 7873–7882.

[91] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[92] K. Greff, R. K. Srivastava, J. Koutnik, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A search space odyssey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2222–2232, Oct 2017.

[93] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *CoRR*, vol. abs/1412.3555, 2014. [Online]. Available: http://arxiv.org/abs/1412.3555

[94] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL '02. Stroudsburg, PA, USA: Association for Computational Linguistics, 2002, pp. 311–318. [Online]. Available: https://doi.org/10.3115/1073083.1073135

[95] M. Denkowski and A. Lavie, "Meteor universal: Language specific translation evaluation for any target language," in *In Proceedings of the Ninth Workshop on Statistical Machine Translation*, 2014.

[96] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out: Proceedings of the ACL-04 Workshop*, S. S. Marie-Francine Moens, Ed. Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 74–81.

[97] R. Vedantam, C. L. Zitnick, and D. Parikh, "CIDEr: Consensus-based image description evaluation," *CoRR*, vol. abs/1411.5726, 2014. [Online]. Available: http://arxiv.org/abs/1411.5726

[98] E. M. Izhikevich, "Which model to use for cortical spiking neurons?" *Neural Networks, IEEE Trans. on*, vol. 15, no. 5, Sep. 2004.

[99] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[100] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *ArXiv*, vol. abs/1608.08710, 2016.

[101] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *ArXiv*, vol. abs/1704.04861, 2017.

[102] M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression," *ArXiv*, vol. abs/1710.01878, 2018.

[103] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *NeurIPS*, 2018.

[104] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *J. Mach. Learn. Res.*, vol. 18, no. 1, pp. 6869–6898, Jan. 2017.

[105] A. Mishra, E. Nurvitadhi, J. J. Cook, and D. Marr, "Wrpn: wide reduced-precision networks," *arXiv preprint arXiv:1709.01134*, 2017.

[106] S. Hellebrand, J. Rajski, S. Tarnick, S. Venkataraman, and B. Courtois, "Built-in test for circuits with scan based on reseeding of multiple-polynomial linear feedback shift registers," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 223–233, 1995.

[107] I. Bayraktaroglu and A. Orailoglu, "Test volume and application time reduction through scan chain concealment," in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, 2001, pp. 151–155.

[108] A. Chandra and K. Chakrabarty, "Test data compression for system-on-a-chip using golomb codes," in *Proceedings 18th IEEE VLSI Test Symposium*, 2000, pp. 113–120.

[109] J. Rajski, J. Tyszer, M. Kassab, and N. Mukherjee, "Embedded deterministic test," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 5, pp. 776–792, 2004.

[110] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[111] "ARM Announces New Cortex-A35 CPU - Ultra-High Efficiency For Wearables & More: https://www.anandtech.com/show/9769/arm-announces-cortex-a35," *Anandtech*, 2015.

[112] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22Nd ACM International Conference on Multimedia*, ser. MM '14. New York, NY, USA: ACM, 2014, pp. 675–678. [Online]. Available: http://doi.acm.org/10.1145/2647868.2654889

[113] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[114] "OpenBLAS: An optimized BLAS library: http://www.openblas.net/," 2017. [Online]. Available: http://www.openblas.net/

[115] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[116] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[117] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[118] S. Sen, S. Venkataramani, and A. Raghunathan, "Approximate computing for spiking neural networks," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, March 2017, pp. 193–198.

[119] M. Luong, E. Brevdo, and R. Zhao, "Neural machine translation (seq2seq) tutorial," *https://github.com/tensorflow/nmt*, 2017.

[120] *Collecting Highly Parallel Data for Paraphrase Evaluation*. Association for Computational Linguistics, January 2011. [Online]. Available: https://www.microsoft.com/en-us/research/publication/collecting-highly-parallel-data-for-paraphrase-evaluation/

[121] Z. Tu, Z. Lu, Y. Liu, X. Liu, and H. Li, "Coverage-based neural machine translation," *CoRR*, vol. abs/1601.04811, 2016. [Online]. Available: http://arxiv.org/abs/1601.04811

[122] J. Vreeken, "Spiking neural networks, an introduction," Tech. Rep., 2003.

[123] P. U. Diehl, D. Neil, J. Binas, M. Cook, S. Liu, and M. Pfeiffer, "Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing," in *2015 International Joint Conference on Neural Networks (IJCNN)*, July 2015, pp. 1–8.

[124] Y. Cao, Y. Chen, and D. Khosla, "Spiking deep convolutional neural networks for energy-efficient object recognition," *International Journal of Computer Vision*, vol. 113, no. 1, pp. 54–66, May 2015. [Online]. Available: https://doi.org/10.1007/s11263-014-0788-3

[125] D. Kuzum, S. Yu, and H.-S. P. Wong, "Synaptic electronics: materials, devices and applications," *Nanotechnology*, vol. 24, no. 38, p. 382001, sep 2013. [Online]. Available: https://doi.org/10.1088%2F0957-4484%2F24%2F38%2F382001

[126] X. Liu, M. Mao, B. Liu, B. Li, Y. Wang, H. Jiang, M. Barnell, Q. Wu, J. Yang, H. Li, and Y. Chen, "Harmonica: A framework of heterogeneous computing systems with memristor-based neuromorphic computing accelerators," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 63, no. 5, pp. 617–628, May 2016.

[127] A. Sengupta, P. Panda, P. Wijesinghe, Y. Kim, and K. Roy, "Magnetic tunnel junction mimics stochastic cortical spiking neurons," *Scientific Reports*, vol. 6, July 2016.

[128] A. Rodriguez, "Lowering Numerical Precision to Increase Deep Learning Performance," *[online] Intel*, 2018. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/lower-numerical-precision-deep-learning-inference-and-training.html

[129] B. M. Fleischer, S. Shukla, M. M. Ziegler, J. Silberman, J. Oh, V. Srinivasan, J. Choi, S. Mueller, A. Agrawal, T. Babinsky, N. Cao, C.-Y. Chen, P. Chuang, T. W. Fox, G. Gristede, M. Guillorn, H. Haynie, M. Klaiber, D. Lee, S.-H. Lo, G. W. Maier, M. Scheuermann, S. Venkataramani, C. Vezyrtzis, N. Wang, F. Yee, C. Zhou, P.-F. Lu, B. W. Curran, L. Chang, and K. Gopalakrishnan, "A Scalable Multi- TeraOPS Deep Learning Processor Core for AI Trainina and Inference," *2018 IEEE Symposium on VLSI Circuits*, pp. 35–36, 2018.

[130] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[131] Y. Chen, T. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[132] U. Gupta, B. Reagen, L. Pentecost, M. Donato, T. Tambe, A. M. Rush, G. Wei, and D. Brooks, "Masr: A modular accelerator for sparse rnns," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019, pp. 1–14.

[133] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "Sparten: A sparse tensor accelerator for convolutional neural networks," *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[134] M. Graphics, "Tessent TestKompress www.mentor.com/products/silicon-yield/multimedia/overview/tessent-testkompress-1849a6d7-9f9d-404a-9c19-f12d945c3baf," *[online]*.

[135] L. K. Hansen and P. Salamon, "Neural network ensembles," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 10, pp. 993–1001, Oct 1990.

[136] L. Breiman, "Bagging predictors," *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, Aug. 1996. [Online]. Available: http://dx.doi.org/10.1023/A:1018054314350

[137] T. G. Dietterich, "Ensemble methods in machine learning," in *Multiple Classifier Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 1–15.

[138] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[139] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[140] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," *ArXiv e-prints*, Mar. 2016.

[141] N. Papernot, F. Faghri, N. Carlini, I. Goodfellow, R. Feinman, A. Kurakin, C. Xie, Y. Sharma, T. Brown, A. Roy, A. Matyasko, V. Behzadan, K. Hambardzumyan, Z. Zhang, Y.-L. Juang, Z. Li, R. Sheatsley, A. Garg, J. Uesato, W. Gierke, Y. Dong, D. Berthelot, P. Hendricks, J. Rauber, and R. Long, "Technical report on the cleverhans v2.1.0 adversarial examples library," *arXiv preprint arXiv:1610.00768*, 2018.

VITA

VITA

Sanchari Sen received the B.Tech (Hons.) degree in Electronics and Electrical Communication Engineering from the Indian Institute of Technology, Kharagpur, India. She is pursuing a Ph.D. degree in the School of Electrical and Computer Engineering, Purdue University, West Lafayette, Indiana.

She worked as a summer intern at AMD Research, Austin, Texas in 2018. She was also an intern at the IBM T. J. Watson Research Center, Yorktown Heights, New York, during the summer of 2019. Her current research interests include software and hardware techniques for improving the execution efficiency and robustness of deep neural networks on different platforms.

She was awarded the Institute Silver medal for her academic performance in IIT Kharagpur. As part of the Ph.D. program at Purdue University, she received the Ross Fellowship and the Bilsland Dissertation Fellowship in 2015 and 2019, respectively. She also received the Richard Newton Young Student Fellowship Award for the Design Automation Conference (DAC) in 2016.