Accelerator Architecture for Secure and Energy Efficient Machine Learning

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Mohammad Hossein Samavatian,

Graduate Program in Department of Computer Science and Engineering

The Ohio State University

2022

Dissertation Committee:

Radu Teoderescu, Advisor

Wei-Lun Chao

Yang Wang

Abstract

ML applications are driving the next computing revolution. In this context both performance and security are crucial. We propose hardware/software co-design solutions for addressing both. First, we propose RNNFast, an accelerator for Recurrent Neural Networks (RNNs). RNNs are particularly well suited for machine learning problems in which context is important, such as language translation. RNNFast leverages an emerging class of non-volatile memory called domain-wall memory (DWM). We show that DWM is very well suited for RNN acceleration due to its very high density and low read/write energy. RNNFast is very efficient and highly scalable, with a flexible mapping of logical neurons to RNN hardware blocks. The accelerator is designed to minimize data movement by closely interleaving DWM storage and computation. We compare our design with a state-of-the-art GPGPU and find 21.8× higher performance with 70× lower energy.

Second, we brought ML security into ML accelerator design for more efficiency and robustness. Deep Neural Networks (DNNs) are employed in an increasing number of applications, some of which are safety-critical. Unfortunately, DNNs are known to be vulnerable to so-called adversarial attacks. In general, the proposed defenses have high overhead, some require attack-specific re-training of the model or careful tuning to adapt to different attacks. We show that these approaches, while successful

for a range of inputs, are insufficient to address stronger, high-confidence adversarial attacks. To address this, we propose HASI and DNNSHIELD, two hardware-accelerated defenses that adapt the strength of the response to the confidence of the adversarial input. Both techniques rely on approximation or random noise deliberately introduced into the model. HASI uses direct noise injection into the model at inference. DNNSHIELD uses approximation that relies on dynamic and random sparsification of the DNN model to achieve inference approximation efficiently and with fine-grain control over the approximation error. Both techniques use the output distribution characteristics of noisy/sparsified inference compared to a baseline output to detect adversarial inputs. We show an adversarial detection rate of 86% when applied to VGG16 and 88% when applied to ResNet50, which exceeds the detection rate of the state of the art approaches, with a much lower overhead. We demonstrate a software/hardware-accelerated FPGA prototype, which reduces the performance impact of HASI and DNNSHIELD relative to software-only CPU and GPU implementations.

Vita

August 4, 1989 ..... Born - Hamedan, Iran

2011 ..... B.S. Computer Engineering

2013 ..... M.S. Computer Science

2015-present ..... PhD Computer Science

Publications

Mohammad Hossein Samavatian, Saikat Majumdar, Kristin Barber, Radu Teodorescu "HASI: Hardware-Accelerated Stochastic Inference, A Defense Against Adversarial Machine Learning Attacks". *Secure and Private Systems for Machine Learning Workshop* 2021

Mohammad Hossein Samavatian, Anys Bacha, Li Zhou, Radu Teodorescu "RNNFast: An Accelerator for Recurrent Neural Networks Using Domain-Wall Memory". *ACM Journal on Emerging Technologies in Computing Systems*, Article No.: 38, September 2020

Collaborative publications:
Saikat Majumdar, Mohammad Hossein Samavatian, Kristin Barber, Radu Teodorescu "Using Undervolting as an on-Device Defense Against Adversarial Machine Learning Attacks". *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)* 2021

Fields of Study

Major Field: Department of Computer Science and Engineering

Table of Contents

Page

List of Tables

Table **Page**

List of Figures

Chapter 1: Introduction

Deep learning is transforming the way we approach everyday computing. Deep neural networks (DNNs) are rapidly becoming indispensable tools for solving an increasingly diverse set of complex problems, including computer vision [70], natural language processing [28], machine translation [13], and many others. In this context both performance and security are crucial. Different applications in deep learning have different requirements that may not necessarily be aligned. Some applications need more memory or compute resources depending on the underlying neural network while in other applications privacy and security concerns have higher priority. However, compute efficiency is not necessarily unrelated to security and privacy. For instance, techniques designed to improve energy efficiency can be used indirectly for ML privacy purposes. Mobile devices, while energy constrained, are sufficiently powerful to allow complex ML computation to be performed locally, avoiding the transfer of potentially sensitive user data to the cloud. This does however require energy efficient ML solutions. This is especially important for memory-intensive ML models such as recurrent neural networks.

Applications like speech recognition empower today's digital assistants, business intelligence applications fueled by the analysis of social media postings, etc. For these applications, processing information in a way that preserves the correct context

is crucial. For instance, The sentences "white blood cells destroying an infection" and "an infection destroying white blood cells" have very different meanings even though they contain the same words. Machine learning models such as Convolutional Neural Networks (CNNs) do not consider context and are therefore not well suited for solving such problems. Recurrent Neural Networks (RNNs) are a powerful class of networks designed to consider context by retaining and using information from previously processed inputs. RNNs can learn sequences and can be applied to any problems that require context that needs to be remembered. The popularity of RNN networks in production applications was highlighted by Google in a recent paper [65], which reports that RNN workloads represent almost 30% of the workloads on Google's TPU datacenters. This is in contrast to only 5% for CNN workloads. RNNs have been shown to scale better than CNNs to large input sizes even for applications for which CNNs are generally well suited, such as image processing [12]. Enabling mobile devices to run such workloads on device benefits from preserving users data privacy.

At the same time, these and other application domains, such as medical, self-driving cars, face recognition, etc. require high accuracy outputs to gain public trust and widespread commercial adoption. Unfortunately, DNNs are known to be vulnerable to so-called "adversarial attacks" that purposefully compel classification algorithms to produce erroneous results. For example, in the computer vision domain, a large number of attacks [18, 92, 93, 46, 20, 73, 103, 88] have demonstrated the ability to force state-of-the-art classifiers such as ResNet[54], AlexNet[71], VGG[121], etc. to misclassify inputs that are carefully manipulated by an attacker. In most of the attacks, input images are only slightly altered such that they appear to the casual observer to be unchanged. However, the alterations are made with sophisticated

attacks that, in spite of the imperceptible changes to the input, resulting in reliable misclassification. Running models on local devices also cannot prevents such attacks, therefore security of the ML models is another issue that needs to be addressed directly.

Advancements in commercial hardware accelerators (e.g. Google's TPUs[66], NVIDIA's Volta and Turing architectures, or Microsoft's Brainwave [39]), and new frameworks [4] are making the deployment of machine learning algorithms to increasingly diverse domains possible. Emerging accelerator architectures [23, 21, 36, 81, 36, 52, 22, 118, 24, 69, 6, 79, 113, 82, 26] hold the promise of continued performance gains, driving new applications. Most of these accelerators target specific types of neural network or design accelerators for general deep learning algorithms. More importantly, almost none of them address security issues.

In this work, we introduce three hardware accelerators, RNNFast, HASI and DNNShield. RNNFast is an accelerator for RNN, while HASI and DNNShield are hardware accelerators for CNNs that are robust against adversarial attacks. RNNFast leverages domain-wall memory (DWM), an emerging non-volatile memory technology, to provide high-density on-chip storage as well as energy-efficient computation. DWM [107, 139, 140, 147, 60, 26, 153] is a magnetic spin-based memory technology, which stores information by setting the spin orientation of so-called magnetic domains in a ferromagnetic wire. Multiple magnetic domains can occupy a single wire (referred to as "racetrack") allowing up to 64 bits to be represented.

HASI and DNNShield are hardware/software co-designed defense that relies on a novel stochastic inference process to effectively defend against state-of-the-art

adversarial attacks. We show that HASI and DNNSHIELD can easily generalize to different attacks while requiring no changes to the DNN models.

1.1 Organization of this Thesis

The rest of this thesis is organized as follows.

Chapter 2 will introduce RNNFast architecture. In this chapter, we first provide background and related works around neural network acceleration. Then we present RNNFast architecture and its detailed design. We conclude the chapter with the evaluation of RNNFast and comparing with other accelerators.

Chapter 3 discusses the security issues of neural networks within the scope of image classification. We first introduce adversarial attacks for image classification and then discuss limitations of existing defenses. Next we introduce HASI and DNNSHIELD, hardware/software co-designed defense methods, and compare them with the state of the art defenses.

Chapter 2: RNNFast: An Accelerator for Recurrent Neural Networks Using Domain Wall Memory

Recurrent Neural Networks (RNNs) are a powerful class of networks designed to consider context by retaining and using information from previously processed inputs. RNNs have the ability to learn sequences and can be applied to any problems that require context that needs to be remembered. RNNs are used across a wide range of applications that include speech recognition for digital assistants such as Siri and Google Now, sentiment analysis for classifying social media postings, and language translation.

However, RNN workloads are data-intensive because they store a partial history of the output sequence and perform computations on that history along with the current input. As a result, RNNs require both vast amounts of storage and increased processing power. For example, the RNN neuron requires 8× the number of weights and multiply-accumulate (MAC) operations of a typical FC neuron. RNN networks are also generally quite large. For instance, Amodei et al. [7] developed a network for performing speech recognition that utilized seven recurrent layers and a total of 35 million parameters. At this scale, RNNs with large input sets are susceptible to memory bottlenecks when running on existing accelerators such as GPUs [49] or FPGAs [49, 76, 38, 91, 11, 135, 77, 136]. In addition, the fundamentally different

design of the RNN cell makes previously proposed custom CNN accelerators [131, 120, 105, 23, 21, 81, 36, 52, 22, 19, 118, 24, 69, 6, 137, 64, 72, 79, 113, 82] not directly applicable to RNN workloads.

In this chapter we introduce RNNFast, a hardware accelerator for RNN networks. RNNFast leverages domain-wall memory (DWM), an emerging non-volatile memory technology, to provide high density on-chip storage as well as energy efficient computation. DWM has many attractive characteristics. It has read and write latencies that are close to SRAM and write performance and energy that are substantially lower than STT-RAM and other non-volatile memories [133]. Perhaps more importantly, DWM is expected to have 30× higher density than SRAM and 10× higher than DRAM or STT-RAM. The technology would therefore allow dramatically higher storage capacity in the same chip area. While the technology is still in the early stages of development, prototypes have yielded encouraging results [10]. We show that DWM is very well suited for RNN acceleration due to its very high density, linear access pattern, and low read/write energy.

The RNNFast architecture is modular and highly scalable forgoing the need for long communication buses despite the high output fanout of typical RNN networks. RNNFast allows flexible mapping of logic neurons to RNN hardware blocks. The accelerator is designed to minimize data movement by closely interleaving DWM storage and computation. The basic hardware primitive, the RNN processing element (PE) includes custom DWM-based multiplication and custom nonlinear functional units for high performance and low-energy. RNNFast also includes an error mitigation mechanism for position errors, expected to be relatively common in DWM. The error mitigation is tailored to the RNNFast data access pattern to minimize overhead. We

compare RNNFast with a state-of-the art NVIDIA P100 GPGPU and find RNNFast improves performance by 21.8× while reducing energy 70×.

We also compare with two alternative RNNFast designs. 1) a CMOS-based RNNFast design in which both memories and logic use traditional CMOS. We find the RNNFast design to be up to 2× more energy efficient than the CMOS version, in a much smaller chip area. 2) a memristor-based implementation that uses an analog dot-product engine, a state-of-the-art design that has been shown to be very efficient for CNNs [24, 9]. RNNFast shows better performance, energy and area than the memristor-based design. Qualitative comparisons with FPGA-based RNN accelerators, Google's TPU and Microsoft's Brainwave [39] also indicate RNNFast has better performance and lower energy for similar workloads.

The rest of this chapter is organized as follows: Section 2.1 provides background information. Section 2.2 discusses related work. Section 2.3 details the RNNFast architecture. Section 2.4 presents the error mitigation aspects of the design. Sections 2.5 and 2.6 describe the evaluation and Section 2.7 concludes this chapter.

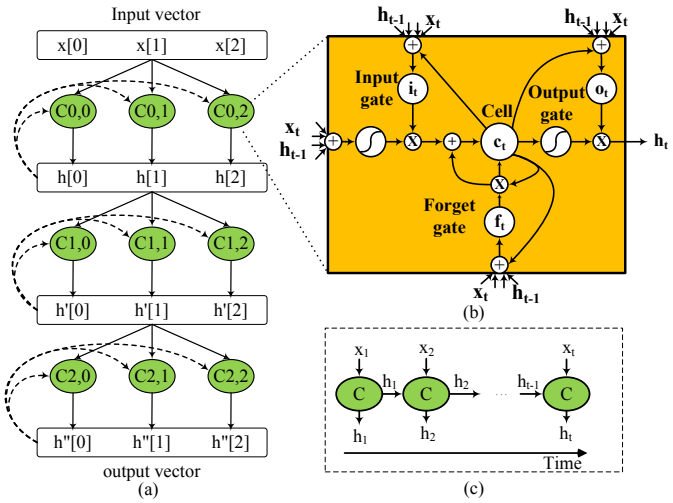

Figure 2.1: (a) 3-layer RNN with 3 LSTM cells/layer, (b) LSTM cell, (c) an LSTM cell unrolled over time

2.1 Background

Recurrent neural networks (RNN) are a powerful class of networks that have the ability to learn sequences. They are applicable to anything with a sense of order that needs to be remembered. RNNs are used across a wide range of applications that includes speech recognition for enabling today's digital assistants, sentiment analysis for analyzing posts (text and video) and classifying them as positive or negative, and machine translation for sequence to sequence translation between languages.

2.1.1 The Long Short-Term Memory Cell

Most recurrent neural networks make use of special "neurons" called Long Short-Term Memory (LSTM) cells [56, 48]. LSTMs are designed to process and remember prior inputs and factor them into their outputs over time. Figure 2.1 shows an example of a very simple 3-layer RNN with 3 LSTM cells/layer. The output of each

layer is a vector that is supplied as the input to the following layer. In addition to those inputs, a feedback loop takes the output vector of each layer and feeds it back as an additional input to each LSTM neuron. An illustration of the inputs and outputs of a single LSTM cell C unrolled over time is shown in Figure 2.1(c). An input x_0 into neuron C at time step $t = 0$, will generate an output h_0 that is propagated downstream to the next layer. In addition, h_0 is saved within the neuron's memory cell for use in the next time step. At time step $t = 1$, the same neuron C will process input x_1, but also use the previously stored output h_0 to generate the new output h_1.

A detailed look inside the LSTM neuron (Figure 2.1(b)) reveals a significantly more complex operation compared to CNN neurons. The strength of the LSTM lies in the way it regulates the fraction of information it recalls from its embedded memory and the fraction of input it processes for generating outputs over time. In other words, the LSTM cell progressively memorizes and forgets contextual information as it processes more inputs. This is achieved through special gates that are controlled through a set of mathematical functions [47] governed by equations 2.1–2.5.

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \tag{2.1}$$

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \tag{2.2}$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \tag{2.3}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \tag{2.4}$$

$$h_t = o_t \odot tanh(c_t) \tag{2.5}$$

The input gate i_t receives the input to be written into a neuron's memory cell at time step t. The forget gate f_t controls what information should be erased from a neuron's memory cell at time step t. The cell c_t represents the content of the neuron's

memory cell. The output gate o_t controls the amount of information read from the neuron's cell and how much of it contributes to the output. The output h_t represents the output of the cell to the next layer at time step t. This output is also fed back into the input gate i_{t+1} of the same LSTM cell at time step $t+1$. The Ws and bs represent the weights and biases, respectively.

Note that $\odot$ used in equations 2.4 and 2.5 represents the dot product operator. In addition, equations 2.1–2.5 represent neurons for an entire layer within a network. Therefore, i_t, f_t , o_t, c_t, h_t, h_{t-1}, and x_t are vectors and all Ws are matrices. As such, if we augment a given matrix W to include the weights for both x and h such that its dimensions are $n \times m$, then each row in W^l for hidden layer l would be mapped to neuron j where $j \in [1, n]$. The value m is the size of input vector.

Because of the complex design, LSTM cells require substantially more storage and computation relative to their CNN counterparts. Moreover, RNN networks are also generally fully-connected, further increasing the data movement overhead.

$$W^l = \begin{bmatrix} W^l_{11} & \dots & W^l_{1m} \\ \vdots & \ddots & \vdots \\ W^l_{n1} & \dots & W^l_{nm} \end{bmatrix} \tag{2.6}$$

The *tanh* and σ activation functions are also outlined in equations (2.7) and (2.8) for clarity. These functions are applied as elementwise operations on the resulting vectors.

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{2.7}$$

$$tanh(z) = 2\sigma(2z) - 1 \tag{2.8}$$

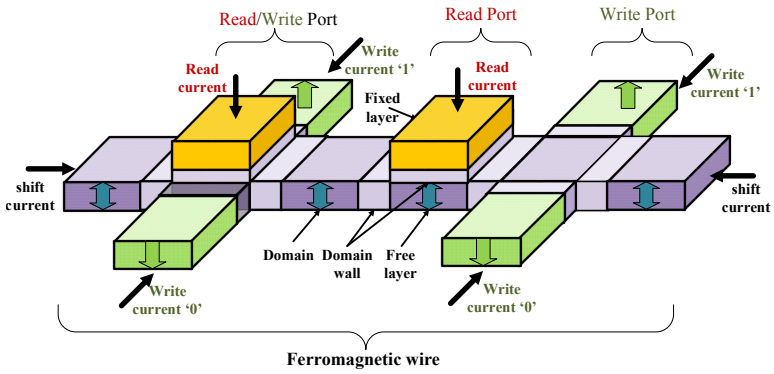

Figure 2.2: DWM device structure.

2.1.2 Domain-wall Memory

Domain wall (a.k.a. racetrack) memory was first proposed by Parkin et al. [107] from IBM in 2008. In 2011, Annunziata et al.[10] demonstrated the first 200mm DWM wafer, fabricated with IBM 90nm CMOS technology. Each die contained 256 racetrack cells, proving the feasibility of DWM fabrication. A large body of research has since sought to improve and optimize the technology at device and circuit levels [125, 43, 123, 151, 94, 148, 132] and find solutions to improve its reliability [149].

Domain wall (racetrack) memory represents information using the spin orientation of magnetic domains in a ferromagnetic wire, as shown in Figure 2.2. Each of these domains can be independently set to an up-spin or down-spin to represent the value of a single bit. Since multiple magnetic domains can reside on a single wire, multiple bits (32-64) of data can be packed in a single DWM device, resulting in a very high density. Three basic operations can be performed on a DWM device: read, write and shift. A magnetic tunnel junction (MTJ) [145, 122] structure is used to read data from the DWM cell (read port in Figure 2.2).

In a DWM device, all the magnetic domains share a single read MTJ (generally referred-to as a read head or port). The bit to be read needs to be aligned with the MTJ before it can be accessed. This is accomplished using a property that is unique to DWM, called domain wall motion, which refers to the shifting of magnetic domains down the ferromagnetic wire. When a current pulse of a suitable magnitude is applied through the ferromagnetic wire, the magnetic spins of all domains "move" across the wire in a direction opposite to the direction of the current. The number of bit positions in a shift motion is controlled by the duration of the shift current. Additional blank domains are included at the ends of each racetrack to allow all data domains to be shifted to the read head without data loss at the ends of the wire [112].

Formerly, the write operation was also performed with the MTJ similar to STT-RAM. This write operation is highly energy consuming. However, a recent development in DWM has eliminated this inefficiency. It has been experimentally shown that domain wall motion (originally intended to realize shifts) can also be used to perform fast, energy-efficient writes in DWMs. This operation, often referred as shift-based writes, is demonstrated in Fig. 2.2-b. The structure for the write operation consists of a ferromagnetic wire with three domains: two fixed domains and a free domain. The magnetization of the two fixed domains are hardwired to up-spin and down-spin during fabrication. However, the magnetization of the free domain, which is sandwiched between the fixed domains, can be varied by shifting the magnetization of one of the fixed domains by applying a current pulse in the appropriate direction. The spin of either of the fixed domains can be shifted into the free domain through the domain motion process by applying a current pulse in the appropriate direction. The latency and energy of shift-based writes are equivalent to those of simple shifts.

The main challenge of racetrack memory is the access latency to data stored in a DWM tape which is variable depending upon the number of shifts required to align the accessed bit with the read or write heads. RNNFast mitigates this disadvantage by optimizing data placement for sequential access such that most accesses only require a single shift.

Reliability

DWM technology also presents reliability challenges including possible misalignment of the data domains leading to erroneous reads and/or writes [61, 149]. Prior work [149] has classified DWM errors into two main types: "stop-in-the-middle" and "out-of-step" errors. The first class of errors is caused when data domains are not aligned with the read/write heads, leading to invalid accesses. The second class of errors is caused when the incorrect domain is aligned with the read/write head which causes the wrong bit in the track to be accessed. The errors are generally caused by variability in the magnitude or duration of the current pulse applied during the domain shift operation. Zhang et al.[149] has developed a technique for eliminating "stop-in-the-middle" errors that relies on the application of a short subthreshold shift current to nudge the misaligned domain back into alignment. They also demonstrate that the subthreshold pulse is small enough that it cannot misalign a correctly aligned domain. As a result, sub-threshold shifts can virtually eliminate "stop-in-the-middle" errors, at the cost of increasing the number of "out-of-step" errors.

While subthreshold shifts can be applied in both directions, we choose to apply them in the shift direction. As a result, all "out-of-step" errors will be converted into overshift errors by 1 or more positions in the shift direction. For a single-position shift, which represents virtually all shifts in RNNFast, the probability of single-bit

overshift is on the order of 10^{-5} [149], which is quite high. However, the probability of multibit overshift is about 10^{-21}, which is negligible. As a result, RNNFast implements mitigation for single-bit overshift errors.

2.2 Related Work

Many customized accelerators for machines learning algorithms and DNNs have been proposed recently [23, 21, 36, 81, 36, 52, 22, 118, 24, 69, 6, 79, 113, 82, 26]. The majority of this work focuses on improving the performance of CNNs, exploring the potential for resources sharing, leveraging emerging memory technologies, optimizing basic operations, and developing domain specific methods.

Han et al. [52] used compression of the network model to reduce the memory footprint and accelerate real-time networks in which batching cannot be employed to improve data reuse. Eyeriss [22] explored local data reuse of filter weights and activations in high-dimensional convolutions in order to minimize the energy of data movement.

Emerging memory technologies and in-memory processing have been leveraged for CNN designs to address memory latency limitations and to implement custom logic. PRIME [24] combined processor-in-memory architecture and ReRAM-based neural network computation. The crossbar array structure in ReRAM can be used to perform matrix-vector multiplication as well as regular memory to increase memory space. PUMA [8], a recently proposed general-purpose and ISA-programmable accelerator built with ReRAM. It has a spatial architecture organized in cores, tiles, and nodes. PUMA features a microarchitecture, ISA, and compiler co-designed to optimize data movement and maximize energy and area efficiency. The PUMA design is more general than ISAAC [118], and, as a result, it generally performs worse in terms of throughput and energy efficiency. ReRAM-based DNN accelerators benefit

from the speed and efficiency of the memristor crossbar; however the need for additional peripheral circuits such as ADCs and DACs, and other components, reduce the benefits of crossbar-based computation.

Neurocube [69] proposed a programmable and scalable digital neuromorphic architecture based on 3D high-density memory integrated with a logic tier for efficient neural computing. The design in [84] also used ReRAM cross bar for RNN acceleration for a case of human activity detection with a small network size of 100 and simple vanilla RNN. CNV [6] accelerates DNNs in hardware by eliminating a large fraction of ineffectual zero-valued operand multiplications. It improves the performance and energy using data-parallel units and a co-designed data storage format without losing accuracy. RedEye [79] reduces analog readout and computational burden by moving convolutional processing into an image sensor's domain. Minerva [29] automates the co-design flow by optimizing across the algorithm, architecture and circuit levels. In details, it aggressively optimizes data types, selectively pruning operations, and reduces SRAM voltages safely with novel fault mitigation techniques. Cambricon [82] propose a novel domain-specific Instruction Set Architecture (ISA) for neural network accelerators which is a load-store architecture that integrates scalar, vector, matrix, logical, data transfer, and control instructions.

PuDianNao [81] focuses on a range of popular machine learning algorithms. However all these optimizations are CNNs/DNNs specific. Chung et. al [26] used DWM for CNN computations as well. They proposed a new design that replaces the ReRAM cross bar with a DWM-based CNN layer for dot product. However, they still use costly ADC/DAC circuits and also did not address DWM shift errors in their design.

Brainwave [39] proposed a single threaded SIMD architecture for CNN/RNN. It expands the compound SIMD operations into thousands fixed vector size operations which form primitives that are fanned out to compute units. These parallelized vector operations that are mapped to one-dimensional flat functional units, connected in a way that allows vectors to flow through the pipeline without any bubbles.

Relatively little work has focused on the acceleration of RNNs. Nurvitadhi et al. [100] performed a comprehensive study of multiple accelarators for Gated Recurrent Units (GRUs), an LSTM variant. They compare CPU/GPU, to FPGAs and their own ASIC design. GRUs have a simpler design than LSTM, with only two control gates (and associated weights) vs four for LSTMs.

On the other hand, some works focused on FPGAs in order to boost up the performance of Neural Networks [49, 76]. Guan et.al focused on LSTM accelaration over FPGA and read 20× speed up regarding CPU implementation for 3 layer 250 neuron per layer network. However, FPGA design cannot compete with ASIC design. [100] performed a comprehensive study over all type of RNN accelarators from CPU-GPU to FPGAs and ASIC. The study shows that FPGA accelators are 7× less efficient than the ASIC designs in the best case. They used external DRAM to store the parameters, inputs and outputs and load the data into FPGA for computation. They reshape parameters off-line in the external DRAM to insure that they can be accessed sequentially regarding the irregular data access in LSTMs. They benefit from data buffers for incoming and outgoing inputs to overlap communication with inference computation.

They configure the FPGA in order to have 4 main LSTM gates (Forget Gate, Cell Gate, Input Gate and Output Gate) and get vetor data from input buffer groups

through a crossbar. These four gate modules perform LSTM-RNN inference, and transport results to LSTM Functional Logic to perform the remaining computation (element-wise multiplication and addition of gate vectors, activations, etc.). Then, the final results are loaded to output buffer groups through a crossbar. The current state of the LSTM cell is stored in am on-chip buffer, called Cell Buffer.

Inside each gate module, gate vector is calculated in a tiling scheme. Tiled input vectors and the correspond-ing parameters are transferred into the LSTM gate module in parallel to perform inference. Inside each LSTM gate module, all multiplications between input elements and parameters are performed in parallel. The results are then summed up through a addition tree to minimize latency. The whole architecture is also pipelined to further improve throughput. The outputs are fed into activation nodes to generate the final output vectors of each gate.

Domain-Wall Memory is an increasingly popular candidate for replacing conventional memories such as Flash, DRAM and SRAM, and there are prior work utilizing DWM in reconfigurable computing and machine learning architectures designs [139, 140, 147, 60, 26, 153]. Zhao et al. [153] employed racetrack memory for reconfigurable computing to achieve high density and low energy compared with SRAM . Chung et al. [26] proposed a DWM dot product engine using a DWM-based analog design, which requires ADCs.Yu et al. [147] designed data intensive machine learning image-processing into in-memory DWM. The high storage density offered by racetrack memory makes it a promising candidate for the data-intensive machine learning applications.

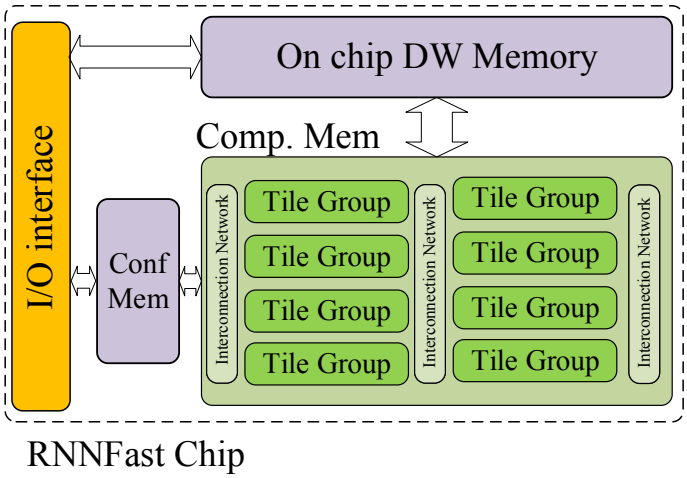

Figure 2.3: RNNFast architecture overview at chip level.

2.3 RNNFast Architecture

We present RNNFast, a new architecture that leverages domain wall memory for accelerating recurrent neural networks. Figure 2.3 shows a comprehensive view of the RNNFast architecture.

At a high level the RNNFast chip consists of Global Memory, a Computational Memory array, Configuration Memory and I/O interface as shown in Figure 2.3. The Global Memory is a dense memory block implemented using DWM. This is the main memory of the accelerator and is used to store inputs and results. The Computational Memory is the compute engine and is implemented primarily using DWM elements augmented with CMOS logic where appropriate. The compute array is organized as a pool of highly reconfigurable and tightly interconnected tile groups.

One or more multi-layer RNN networks can be mapped to multiple tile groups, in a weight-stationary design (weights are stored locally in the Computational Memory).

The Configuration Memory holds the runtime configuration settings for the chip. RNNFast is optimized to deliver low latency without batching, and it is also efficient for batch workloads.

2.3.1 Compute Tiles

A compute tile consists of multiple LSTM hardware units that share a single input and a single output racetrack. They are interconnected with their nearest horizontal and vertical neighbors through racetrack memories. Figure 2.4 shows the tile design and layout. The results of the computation within each tile are written directly onto the input track of the tile belonging to the next layer in the network. Tiles are organized in tile groups, which are connected to each other through traditional wired interconnection networks.

Inter-tile Communication

RNNs are typically fully connected networks requiring all inputs to be delivered to all the neurons in a given layer. The high degree of connectivity that has to be supported by the hardware can lead to substantial energy and area overheads when traditional wired interconnects are used. To address this challenge, we leverage the shifting mechanism of DWM racetracks for communication both within and across tiles.

Within a tile, inputs are read sequentially from the tile's input racetrack and broadcast to all LSTM units across a locally-shared bus. Each read is followed by a shift of the input track to align the next input element with the read head. Figure 2.4 (b) illustrates two timesteps in this process. In addition to the tile-local broadcast, each input is also sent to the neighboring tile on the left for addition to its input

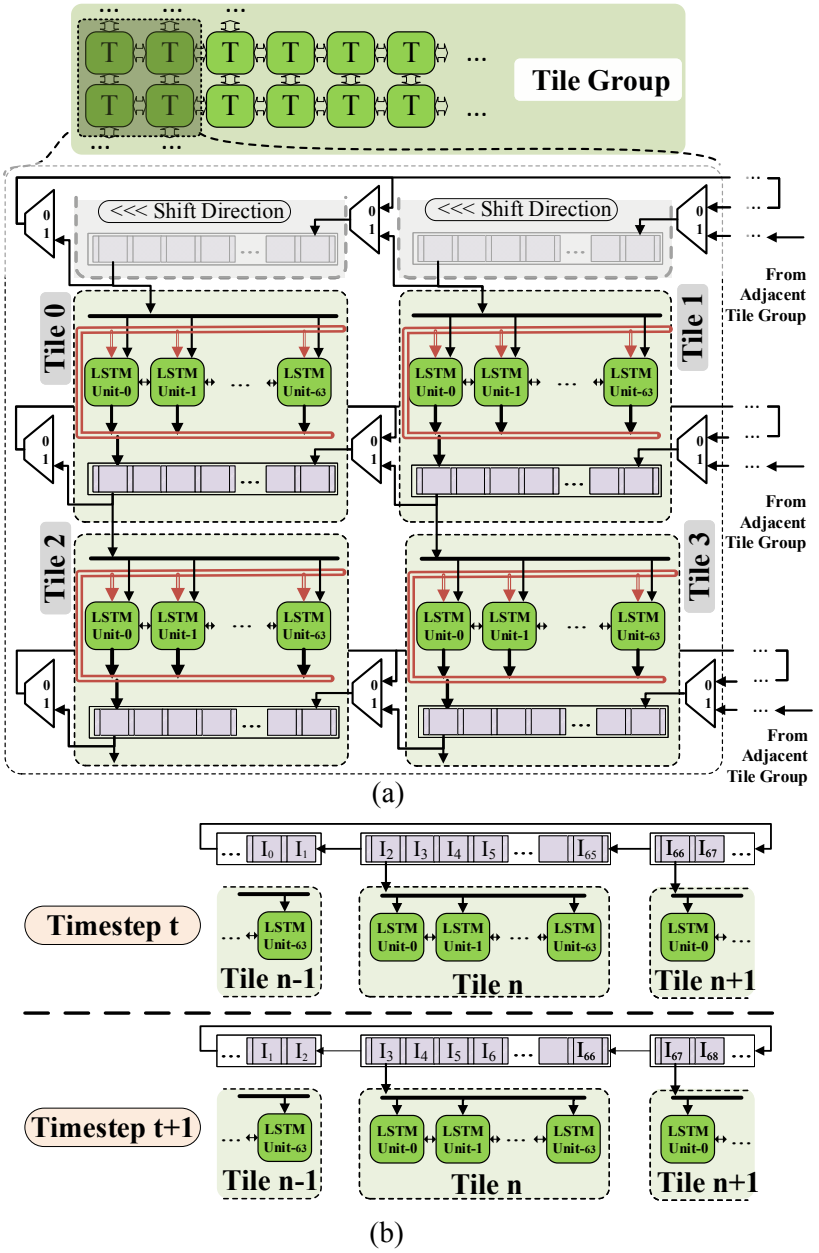

Figure 2.4: (a) Compute tile layout, internal design and interconnection through racetrack chains. (b) Reading inputs into tiles in two consecutive timesteps.

track. We call this process "chaining". Chains are essentially circular buffers that circulate all inputs to all tiles that are mapped to the same layer of the NN. Chains of different lengths can be configured depending on the number of neurons in each layer of the network. Racetracks are connected through MUXs (Figure 2.4 (a)) that enable different chain lengths. A variable number of tracks can be included in a chain by simply setting the right most track MUX to 0 and the rest to 1.

2.3.2 LSTM Units

Each tile consists of multiple LSTM compute units (64 in our design). RNNFast is a weight-stationary design, with fixed capacity for weight storage in each LSTM unit. A logical neuron can be mapped to one or more LSTM compute units depending on the number of weights it requires. We expect a 1-to-1 mapping between logical neurons and hardware LSTM units for most networks. However, when a logical neuron requires more weights than a single LSTM unit can store, it is mapped to multiple LSTM units. Figure 2.5 (a) shows three mapping examples for a single logical LSTM cell: 1 LSTM unit (top), 2 LSTM units (middle) and 4 LSTM units (bottom).

Processing Elements

The architecture of an LSTM cell is shown in Figure 2.5 (b). Each cell is subdivided into four processing elements (PEs) ①. Per equations (2.1) – (2.5), each input X_t is multiplied with four different sets of weights. A single PE can be assigned to any one of the weight sets (known as gates), e.g. I_G, F_G, O_G or C_G. However, an LSTM cell gate can be mapped to one or more PEs across LSTM units depending on its storage requirements and input/output fanout. Allocating four hardware PEs

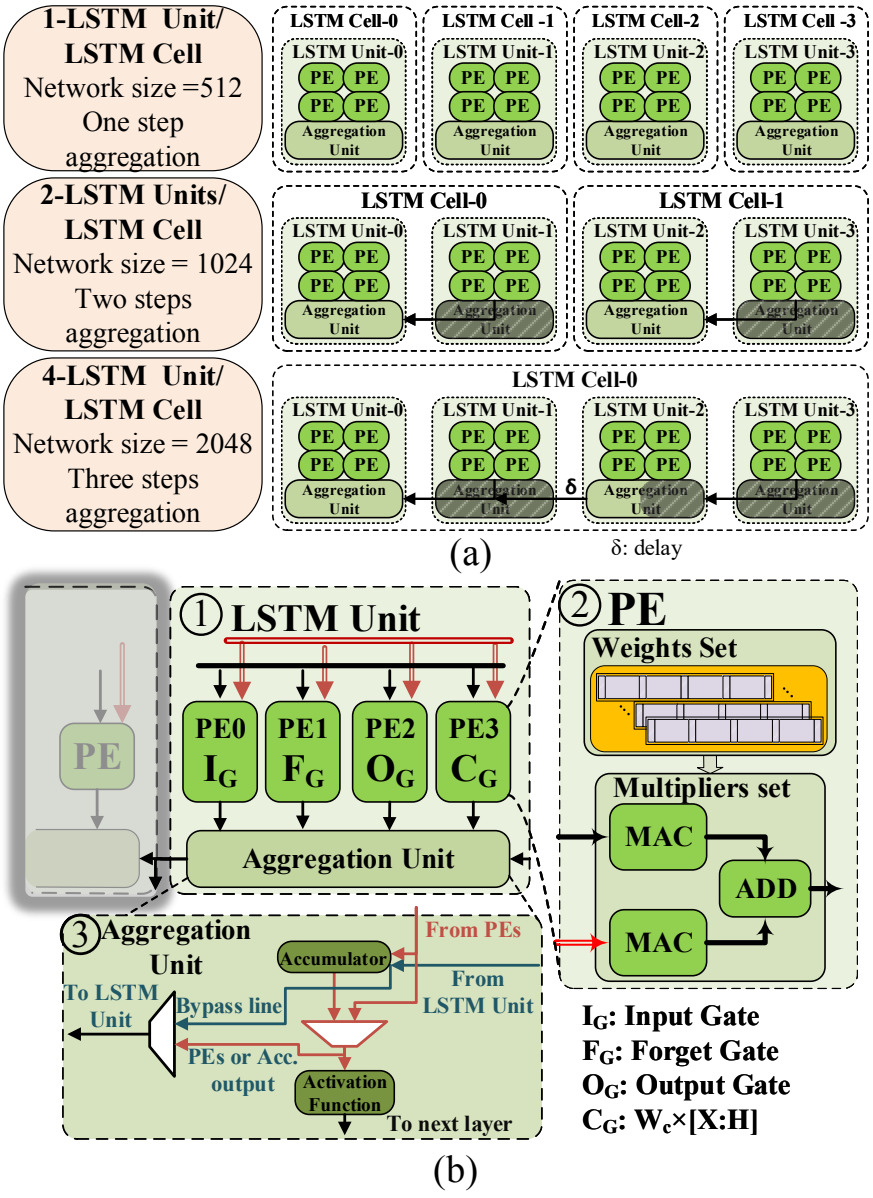

Figure 2.5: (a) Three mapping examples of logical LSTM cells to LSTM units. (b) LSTM unit design.

to each LSTM unit allows RNNFast to accommodate different RNN variants (see Section 2.3.4).

PEs have racetrack-based storage for weights and racetrack-based compute units, including multiply accumulator (MAC) engines for matrix multiplication. The MAC engine is composed of 256+16 DWM based full adders. The MAC unit is deeply pipelined into 48 stages. In order to increase parallelism, each PE uses two MAC engines, one for the main input X_t and one for the feedback input h_{t-1}.

Each PE unit holds a set of weights and performs the dot product on the corresponding subset of inputs. Each PE only consumes inputs corresponding to the weights it stores. Each input to a PE is multiplied by its weight and accumulated with the result of the previous multiplication (2). Each PE stores the result of the accumulation in its own output racetrack.

As in the LSTM neural networks different set of inputs and weights are matrix multiplied, in order to maximize parallelism, the appropriate number of cheap matrix multiplication are implemented inside the memory for each of input-weight sets. Specifically for LSTM networks, the incoming input is multiplied by 4 different set of weights. Thus, the same input with different weight will go to different MAC engine. This is also true for the feedback loop of a LSTM cell which needs same resources in PE unit. The controller inside PE takes advantage of a bitmap and a counter to select the correct weight to use with incoming input.

Input and Weight Mapping

The input and weight assignment to racetracks is a trade-off between access latency and hardware overhead. In RNNFast, inputs are spread across multiple racetracks with 1 bit per track. This allows an entire input word to be read in a single

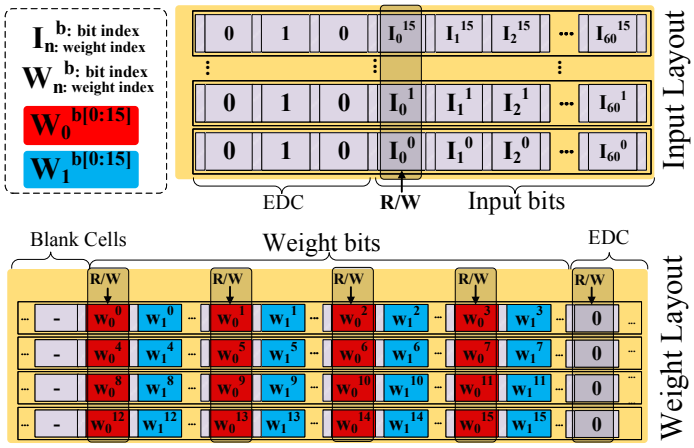

Figure 2.6: Mapping of inputs and weights to racetracks.

cycle, as the top half of Figure 2.6 illustrates. Error detection bits are also included in the tracks and their role will be detailed in Section 2.4. Note that the input tracks do not require dummy domains (Figure 2.4-b). Values at the end of the track are read and sent to the neighboring track.

Unlike inputs, which move from track to track along the chain, weights are stationary at the PE level and are reused multiple times. This means that after scanning all weights, the tracks need to be returned to the initial weight. To minimize the number of shifts, weight values are distributed both within and across multiple racetracks. Weight racetracks are provisioned with multiple read/write heads (5 in our design) which divide the racetrack into 6 10-bit segments. The left-most segment domains are used as dummy domains and the rest of the segments are used to store weight values. Data layout is such that all read heads across all tracks can access all the bits of a single weight simultaneously. Racetracks are grouped in sets of 4, with each set storing 10 weights. The bottom of Figure 2.6 illustrates this layout.

Weight W_0 (red) is currently aligned with the read heads. A single-position shift to the left will align the next weight W_1 (blue) with all the read heads. Access to each set of weight racetracks is pipelined. When all 10 weights are read from the current set of racetracks, the next set of weights will be read from the next set. While the new weights are accessed, the weights in the previous set are shifted back to their initials positions. This takes place when the racetrack set is not being accessed and is therefore off the timing critical path.

Result Aggregation

If more than one LSTM unit is mapped to a neuron, the partial results of the individual LSTMs have to be combined to form the neuron's output. Aggregation units ③ in each LSTM are used to sum up partial results in that LSTM block. In addition, the aggregation units apply the sigmoid and tanh functions and perform the multiplication and accumulation operations in order to generate the final output of the cell.

For cases in which neurons span multiple LSTM blocks, aggregation units in those blocks are linked to produce the final result. This is achieved by collecting all the partial results computed by each LSTM unit (mapped to the same neuron) to a single aggregation unit. Aggregation units are also chained through adjacent LSTM units. Each aggregation unit sends out its final result to the adjacent aggregation unit to its left. The adjacent unit will use the incoming result to either accumulate or bypass it to the next unit (Figure 2.5-③). Even-indexed aggregation units consume and odd-indexed aggregation units forward the incoming result. The leftmost LSTM in a neuron will be responsible for the final aggregation and will apply the sigmoid and tanh. Aggregation time is a logarithmic function in the number of LSTM cells mapped

to a single neuron. This is also done by setting multiplexers in the aggregation unit and power gating the inactive units in output generators at odd indexed LSTM units.

The design tradeoff for LSTM units is driven by the need to support networks that are both large and small. If LSTM units and PEs are too large, storage space will be wasted when small networks are mapped. If they are too small, large networks will require several LSTM units per neuron, increasing the aggregation time.

2.3.3 Nonlinear Functions

The nonlinear functions are an important component of the RNN cells and are used for output activation. As is shown in prior work [14], LUT is an efficient digital implementation of the sigmoid and tanh function for re-programmable logics. RNNFast uses hardware acceleration for the sigmoid and tanh nonlinear functions. The hardware is included in each Aggregation Unit (Figure 2.5). We propose an area efficient approximate logic function-based unit implemented using DWM for the nonlinear functions.

The approximation has been proposed by prior work [127] as an alternative to the standard sigmoid and follows Equation 2.9:

$$\sigma(z) = \begin{cases} \frac{\frac{1}{2} + \frac{\hat{z}}{4}}{2^{|(z)|}} & if z < 0 \\ \\ 1 - \sigma(-z) & if z > 0 \end{cases} \tag{2.9}$$

This approximation has the advantage of being easier to implement in hardware. As Equation 2.9 shows, the hardware has to support division by 2^n numbers. This can be implemented using shift operations which are a feature of racetrack memories. The tanh approximation function can be computed from the sigmoid function through two multiplications and a subtraction. Note that $\hat{z} = z + \mid (z) \mid$, where (z) is the

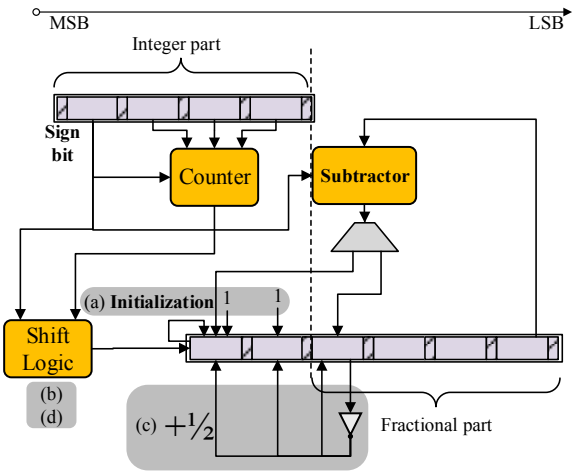

Figure 2.7: DW based implementation of sigmoid/tanh.

integer part of z. Figure 2.7 shows our DWM-based implementation of the sigmoid approximation. The sigmoid for a negative value will be computed as follows: a) the output integer part is initialized with binary '1'; b) two right shifts are performed to compute $\hat{z}/4$; c) $+1/2$ is applied to the result; d) the final result is shifted right $\mid (z) \mid$ times. For a positive number two subtraction steps are added in the beginning and end of above steps. To compute the tanh approximation, a right shift $(2 \times z)$ and a subtraction will be applied in the first and last steps respectively. This design is very area and energy efficient utilizing only a 16 bit racetrack memory, along with some simple subtraction and counting logic. Deep RNNs have a very large number of neurons. This implies dedication of high number of LUT for sigmoid and tanh functions in each LSTM cells which leads to massive storage overhead. Section 2.6 evaluates the relative merits of the approximate designs regarding LUTs.

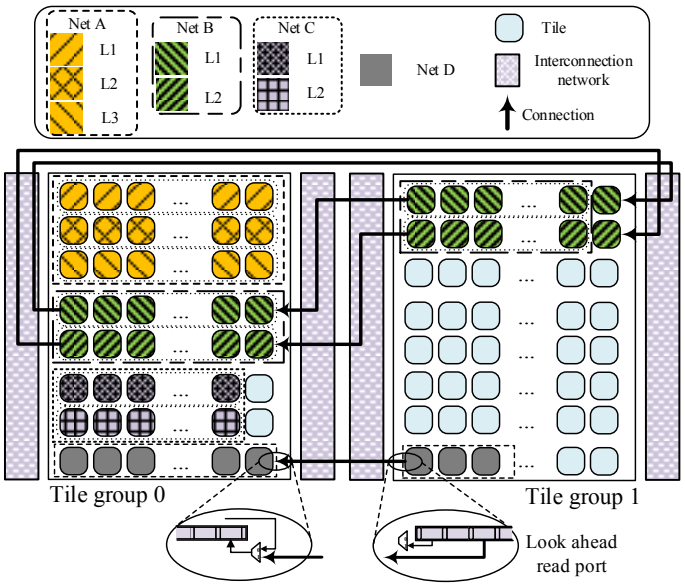

Figure 2.8: Mapping multiple LSTM networks to RNNFast. Interconnection network helps extend racetrack chains beyond tile groups for large networks.

2.3.4 RNNFast Mapping and Configuration

The RNNFast hardware can be configured to implement different network sizes and topologies. Moreover, multiple distinct neural networks can be mapped to the same chip. Outputs from one network can be delivered directly to the following network or stored in the on-chip memory for further processing, if needed. Figure 2.8 illustrates an example of four networks A, B, C and D mapped to two tile groups. Tile groups are connected through a wired interconnect. The racetrack chains for each row of tiles have additional read/write heads to provide access to the inter-tile network.

Multilayer networks span multiple rows with different layers mapped to consecutive rows. Tile groups are designed with wide rows to accommodate most network sizes (e.g. Nets A and C). However, when a network layer cannot fit in a single row,

RNNFast supports splitting it across tile groups (e.g. Nets B and D). This is achieved by extending the input/output racetrack chains to neighboring tile groups using the inter-group wire interconnect. We chose to split layers across tile groups (as opposed to within a tile group) in order to allow consecutive network layers to continue to be mapped to adjacent rows, preserving inter-layer communication.

One important design constraint was to enable the extension of the racetrack chains across tile groups without adding to the track chain shift latency. This is accomplished by implementing a look-ahead read port at the end of the track that reads inputs several cycles ahead of the end of the track, as illustrated for Net D in Figure 2.8. This allows the input to reach the destination row in the neighboring tile through the higher latency interconnect by the time the same input reaches the end of the source track.

Other LSTM Variants

RNNFast is designed for the more demanding LSTM design. However it is also compatible with LSTM variants like Gated Recurrent Unit (GRU) and Vanilla RNN, which require fewer compute resources. Unlike LSTM, the GRU unit does not use a memory element to control the flow of information and are useful when input sequences are not very long. Figure 2.9 shows how a GRU cell can be mapped to a RNNFast LSTM unit. The shaded areas represent unutilized components. A GRU utilizes 75% of the MAC resources. Simpler RNNs like *Vanilla RNN*, only utilize a single PE per neuron and do not need the aggregation unit. As a result, RNNFast can map four *Vanilla RNN* neurons in each LSTM unit.

Moreover, RNNFast allows the mapping of other network types such as Bidirectional RNNs (BiRNN). A BiRNN consists essentially of two RNNs stacked on top of

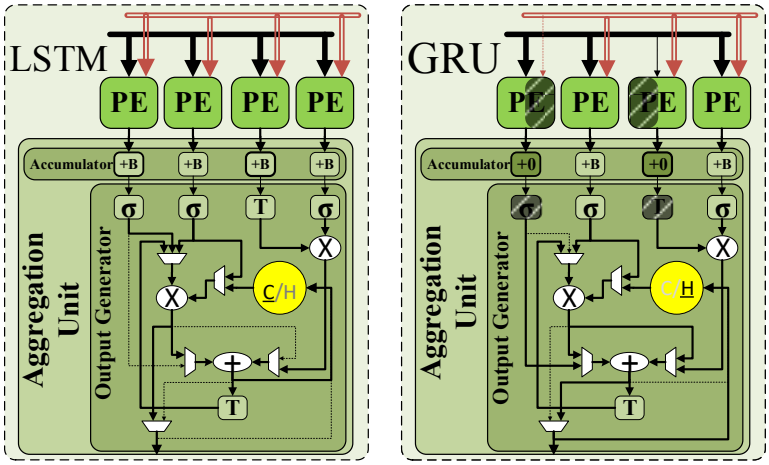

Figure 2.9: LSTM vs GRU cell configuration on RNNFast

each other. The output is computed based on the hidden state of both networks. In our design, the two networks are mapped on the hardware in an interleaved fashion. The aggregation hardware is used to link the two networks. The input data is also duplicated and interleaved in reverse order $(x_1, x_n, x_2, x_{n-1}, x_3, x_{n-2}, ..., x_n, x_1)$.

RNNFast Configuration

The RNNFast configuration is programmed through configuration registers that control input assignment at the PE level, input track chaining, result aggregation setup, etc. A configuration file with the LSTM network(s) specifications is loaded into the device driver of the accelerator and propagated to the appropriate registers.

A config file with LSTM network specifications is fed to device driver config file contains the number of LSTM cells, layers, input size, bit precision, etc. Then device driver decides on whether the RNNFast is capable of placing Network. In the case that a network cannot fit fully into the architecture, the device driver breaks the

LSTM network down into multiple networks in order to place the part of the network into the architecture.

2.4 Error Mitigation Design

As mentioned earlier, DWM technology presents reliability challenges including possible misalignment of the data domains leading to erroneous reads and/or writes known as shift errors. In this section we address shift errors in DWM and its effect on RNNFast.

2.4.1 DWM Position Errors

Out-of-step shift errors, in which the wrong bit is aligned with the read/write heads, are a significant reliability challenge for DWM. Since RNNFast accesses data sequentially, that means virtually all accesses require only single-position bit shifts. While prior work [149] has shown that single-bit shifts are less likely to result in an error compared to multi-bit shifts, these errors are still significant and have to be addressed. For single-position shifts, the probability of single-bit overshift errors is on the order of 10^{-5} [149], which is quite high. However, the probability of multibit overshift is about 10^{-21}, which is negligible. We therefore focus only on single-bit overshift errors, which are expected to occur with a relatively high probability (10^{-5}).

While prior work [113] has shown that neural networks are quite resilient to errors, we find that error rates on the order of DWM overshift errors can degrade output accuracy substantially. Figure 2.10 shows the accuracy of the output for two benchmarks, measured by the BLEU (bilingual evaluation understudy) metric [104], relative to an error-free baseline. BLEU is an algorithm for evaluating the quality of text which has been machine-translated from one natural language to another. Quality is considered to be the correspondence between a machine's output and that of a human. The models that we used have reported very close BLEU scores to the

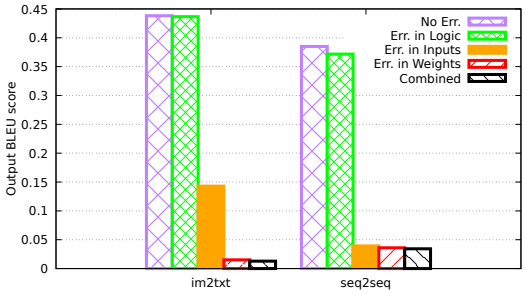

Figure 2.10: Output accuracy (BLEU score) for logic, inputs and weights components.

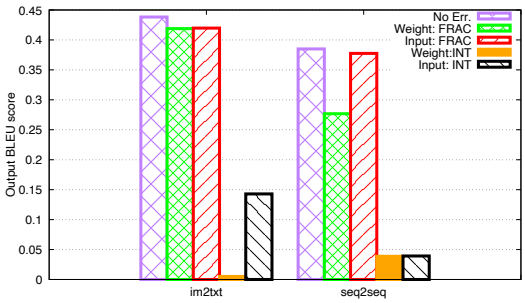

Figure 2.11: Output accuracy (BLEU score) for integer and fraction components.

state of the art models [128]. We inject single-bit overshift errors in different DWM components of RNNFast: the racetrack chains used to hold inputs and outputs for each NN layer, the weights associated with all PEs, the DWM components of the logic functions (MAC units and the nonlinear functions). Shift errors are modeled as a uniform distribution with an overshift probability of 4.55×10^{-5} [149].

Figure 2.10 shows that when errors are injected only in the logic, the drop in output accuracy is very low: ¡1% for *im2txt* and 3% for *seq2seq*, two of the benchmarks we run. This is because overshift off-by-one errors in the MAC and nonlinear functions tend to produce results that are relatively close to the correct value. As a result, the accuracy of the output is very high. However, when errors are injected into the input chains and the weight arrays, the output accuracy drops dramatically to between 10% and 35% of the original. When errors are injected uniformly in all DWM tracks, the output accuracy drops below 5% for *im2txt* and below 10% for *seq2seq*, meaning that the results are essentially useless. This data highlights that mitigation solutions for errors in the inputs as well as weights are essential.

To better understand which errors have the worst effect on output quality, we selectively inject errors into different bits of data words. RNNFast uses 2's complement fixed point representation for both inputs and weights. We inject errors separately into the integer and the fraction portions of the word. Figure 2.11 shows the results of this experiment. When errors are injected only in the fraction, the drop in accuracy is less than 3% for both inputs and weights in *im2txt*. For *seq2seq*, the accuracy degradation is worse when errors are injected in the weights compared to inputs, but the overall output quality is still reasonably high.

Injecting errors with the same probability in the integer portion of the data words has a much more dramatic effect, leading to a drop in output accuracy of between 35% and 10%. The large effect is due to the fact that in these workloads both inputs and weights are represented with small fractional numbers. A single bit flip in the integer fraction can turn a small number into a much larger value, which has a disproportionate effect on the rest of the network.

The large effect on output accuracy is due to the 2's complement representation. This is because a single shift error in a data word that stores a small value can cause that value to be interpreted as a large value with the opposite sign. For example the binary "00000011.10000010" (3.5078125 in decimal) would flip into "00100011.10000010" (35.5078125) or "10000011.10000010" (-124.492188) when a non-sign or sign bit in the integer part is inverted, respectively. This is also true for a negative number. The value "11111111.00101010" (-0.8359375) turns into "01111111.00101010" (127.1640625) after the sign bit is flipped.

2.4.2 RNNFast Error Mitigation

RNNFast addresses overshift errors by implementing an efficient error mitigation mechanism that considers the sensitivity of RNN workloads to errors that result in very large values. We implement different error detection and mitigation mechanisms for input/output racetrack chains and for weight arrays. We take advantage of their design characteristics to implement a more efficient single error detect, single error correct (SEDSEC) design that has lower area overhead and requires fewer extra domains and access ports compared to prior DWM EDC solutions such as [149].

Input Errors

In order to detect overshit errors in the input tracks, we append a 3-bit pattern to the left side of each track, as shown in the example in Figure 2.12. The figure shows a single track that stores bit n for multiple inputs $I_1 - I_7$. In the initial state, the Error Detection Code (EDC) "101" is stored in the leftmost bits of the track. Input I_1 is read in the current cycle. At time t_1 the track is shifted left by 1 to access the next input. If the shift is correct, the leading (check) bit should be a "1". Input I_2 is read and sent to the LSTM units. A new EDC code is written at cycle t_3 in the first three bits of the track using three parallel write ports. Note that updating the EDC does not introduce any time overhead since a write cycle already exists following each read to allow data to be written into the next track in the chain.

At cycle t_4 we show an overshift error. The track has incorrectly shifted left two positions instead of one. This means that I_3 (instead of I_2) is now aligned with the read head. The check bit is now "0" indicating a shift error. To recover from this error we use an additional read head to also read I_2. The outputs of the two read

heads are connected to a multiplexer. The check bit value selects the multiplexer output (shown in blue in Figure 2.12). A "1" selects the error-free output and a "0" selects the overshifted output. A similar mechanism selects the correct location for writing the input coming from the previous track in the chain. If an overshift error occurs, the write location is also shifted to the left, as the right hand side of Figure 2.12 shows.

At t_6 the EDC code is updated again. Following an overshift error, the shift controller will not issue a shift command for the following cycle (t_7) since the track is already properly aligned to access the next input (I_4) during that cycle. Note that, since individual words are stored across multiple tracks to enable single-cycle access, an overshift error will affect all inputs that share that track (up to 60 in our design). It is therefore important to detect and correct these errors.

Errors in Weight Arrays

A similar mechanism is deployed to detect and mitigate errors in weight arrays associated with each PE. However, because the access timing to the weights array is more critical and weights are stored in a more compact representation, the detection and mitigation steps are implemented differently. Unlike inputs, which move from track to track along the chain, weights are stationary at PE level and are reused multiple times. This means that after scanning all weights, the tracks need to be returned to the initial weight. To minimize the number of shifts, weight values are distributed both within and across multiple racetracks. Weight racetracks are provisioned with multiple read/write heads (5 in our design). Data layout is such that all read heads across all tracks can access all the bits of a single weight simultaneously.

Similarly, unlike the input racetrack chain, access to the weight arrays does not require a write cycle, so an update to the EDC code is not feasible. We instead store a fixed EDC pattern of "01010" at the rightmost edge of the weight tracks as shown in Figure 2.13. Error detection logic detects an overshift error when the current EDC bit does not match the expected value. For instance, in the initial state, the read heads are aligned with bits from weight W_0 and the error detection logic expects to read "0" from the EDC.

At time t_1 a correct shift takes place and W_1 can be read. At time t_2 an overshift error occurs and weight W_3 is read instead of W_2. A recovery mechanism similar to the one for inputs could be employed. This would require doubling the number of read heads in each track and extra logic. Since weight storage in RNNFast is substantial, the overhead would be nontrivial. We can, however, avoid this extra overhead by leveraging the observation that replacing the incorrect weight with "zero" yields very little loss in output accuracy compared to error-free execution. This is in contrast with using the erroneous weight, which can be a large value. The following cycle at t_3, the shift controller will not shift because the track is already aligned for accessing the next weight.

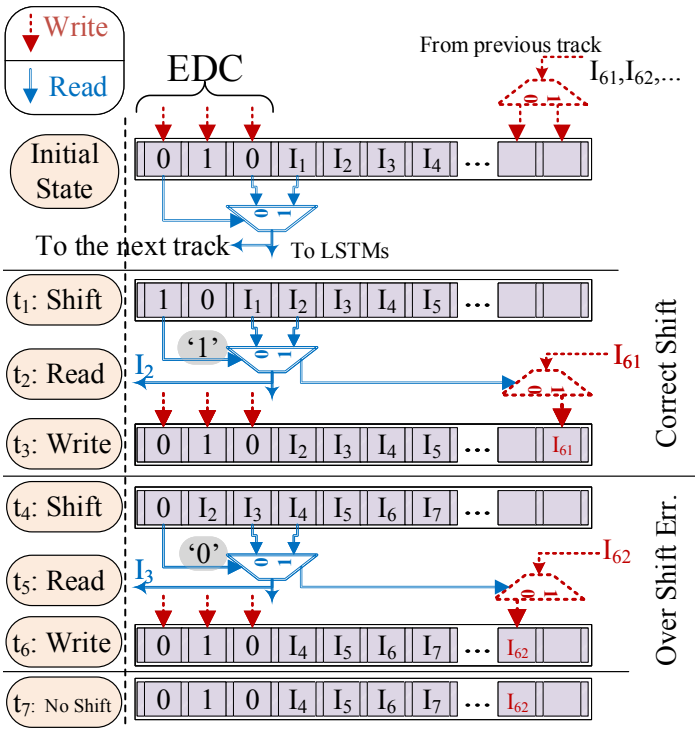

Figure 2.12: Mitigation mechanism for overshift errors in the input track chains.

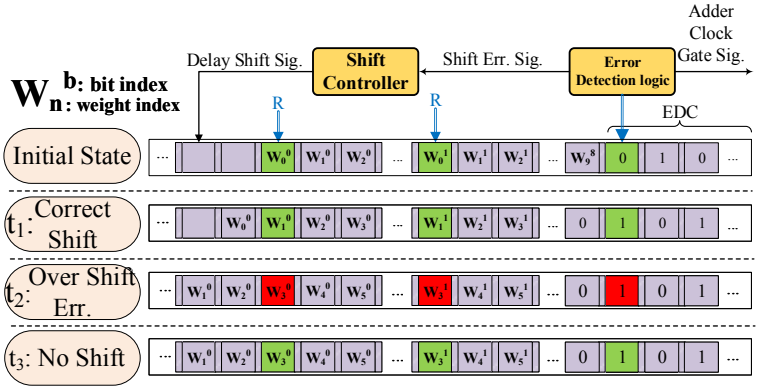

Figure 2.13: Mitigation mechanism for overshift errors in the weight track chains.

2.5 Evaluation Methodology

In this section we describe the methodology for evaluating the RNNFast, discuss RNNFast parameters and design variations and introduce the benhcmarks and baseline architecture that we will compare RNNFast against them.

2.5.1 RNNFast Modeling Infrastructure

We implemented a detailed behavioral model to evaluate performance, chip area and energy consumption of the RNNFast design. A cycle-level model that accounts for the latency of each component in the design is used for the timing simulation. The simulated hardware is configured for each neural network in our benchmark set, by enabling the appropriate number of hardware tiles, LSTMs and PEs. Since all LSTM units execute independently and in parallel, only a single LSTM per tile is simulated to speed up simulation time. For the energy evaluation, we include the number of reads, writes, shifts as well as decoder, adder/multiplier and LUT accesses for all the units in the design.

To understand the energy consumption, an electrical model for the shift and write latency of the Domain Wall Memory (DWM) is necessary. To this end, a Verilog-A based SPICE model for DWM from [95, 97, 96] was simulated on Cadence Virtuoso. The DWM model estimates the effective resistance as a function of the length of the track and uses the width and thickness of the strip to calculate current density and position shift. A Cadence component was created for the DWM model and a test-bench was setup to stimulate the device. A sensitivity analysis was conducted to study the effect of track length on the shift latency and energy. Table 2.1 shows the characteristics of the DWM we model.

Description	Default Value
racetrack width/length/thickness	1F / 64F / 3nm
domain length	1F
number of bits per track	64
read/shift/write energy	0.39pJ/0.24pJ/9.6fJ
read/shift/write latency	1ns/0.5ns/0.5ns
Effective cell size	$2.56F^2$
Technology node	32nm

Table 2.1: Summary of racetrack memory parameters.

Table 2.2 shows the architectural parameters for RNNFast and the power/area breakdown for the different components. Since the weight values use 16-bit precision, each four set of racetracks stores 10 weights. Therefore, storing 512 weights requires each PE to have 205 recetracks. We performed an energy analysis on the number of LSTMs per tile and chose the number of LSTMs per tile to be 64. A more detailed discussion on parameter tuning is included in section 2.6.4. The number of accumulator, multiplier, sigmoid and tanh units in the Aggregation unit (figures 2.1 and 2.9) is optimized for energy and performance. We select the smallest number of units that allows the LSTM to operate without stall cycles.

RNNFast Design Variations

We compare our design with two alternative RNNFast architectures that use CMOS and Memristor technologies. We call them RNNFast-CMOS and ISAAC-RNN, respectively. For RNNFast-CMOS, we used SRAM buffers for both LSTM inputs and weight storage within PEs. MAC units are also implemented with CMOS logic. We used SRAM-based LUTs for the nonlinear functions. Input SRAM buffers

Tile properties				
Component	Configuration	Specification	Power(mW)	area(μm^2)
Input buffer	1 track/tile with EDC	16 stripes/track 64 cell/stripe	2.59	2.68
LSTM unit	64 per tile	4 PEs/LSTM 1 Aggre./LSTM	9.74	2046
Total tile		**256 PEs 64 Aggre. Unit**	**626**	**0.130**mm^2
PE properties				
MAC	2/PE	272 Adder	2.43	422
Weight array	2 track/PE with EDC	205 stripes/track 64 cell/stripe		
Aggregation Unit properties				
Accumulator	4/LSTM	-	0.004	356
Multiplier	2/LSTM	-		
sigmoid	3/LSTM	Approx. nonlinear func. design		
tanh	2/LSTM	Approx. nonlinear func. design		
On-chip DW Memory				
Size: 128MB, 4R/W ports, Area: 6.2mm^2**, Acc. Eng.: 0.89nJ, Acc. lat.: 1.69ns, Leakage 24.3mW**				

Table 2.2: RNNFast design parameters with associated power and area overheads.

are also chained like racetrack memories in order to deliver all the inputs to the LSTM units.

ISAAC-RNN is an ISAAC [118]-like design for RNN that stores inputs in eDRAM and is entirely CMOS and memristor-based. ISAAC-RNN uses 128x128 2-bit memristor crossbars, similar to what was used in ISAAC, for the dot product engine. We kept the input buffer and aggregation unit designs the same as RNNFast in order to observe the effect of memristor in the design and have a more fair comparison since eDRAM and CMOS logic has higher energy consumption than DWM. Each memristor dot product engine is capable of 128×16 multiplications in parallel (128 inputs by 16 weights). Within an LSTM neuron, each input is multiplied by 4 different weight sets. Thus, each memristor dot product engine can handle 4 neurons, making each crossbar in the ISAAC-RNN computationally equivalent to 4 LSTMs in RNNFast. Thus, there are 16 LSTM units per tile for ISAAC-RNN instead of 64 per tile in RNNFast. Inputs are delivered bit by bit to the memristor crossbars.

Therefore, a chuck of 128 inputs needs to be supplied in a single cycle. Memristor based dot product engine performs on single bit of 128 set of inputs simultaneously. For a fair comparison, we changed the input layout to maximize the performance of ISAAC-RNN.

GPU Baseline

We choose as a baseline system for our evaluation a GPGPU optimized for machine learning: the NVIDIA Tesla P100 (Pascal architecture) with 16GB of CoWoS-HBM2 memory. All of our benchmarks use the DNN-optimized cuDNN NVIDIA libraries version 7 [2], which deliver roughly 6× performance improvement relative to a standard GPU implementation for LSTM on Torch [3]. We measure the runtime of the forward passes through the LSTM layers using instrumentation in Deepbench. We measure power consumption using the NVIDIA SMI profiler. Since the SMI profiler provides the total board power, we subtract the power measured at idle in order to isolate the active power of the GPU. Since the board components are less energy proportional with activity compared to the GPU, they account for most of the idle power.

PUMA

We also compared our design with PUMA [8], a recently proposed DNN accelerator built with ReRAM. The authors of PUMA released a simulator and toolchain that we use to compile and run our benchmarks. We used the PUMA compiler to find the number of tiles required for each benchmark. We then set the simulator configuration file to inference mode and used the PUMA simulator to measure runtime and energy consumption.

2.5.2 Benchmarks

We used LSTM-based RNN workloads from the Deepbench [1] open source benchmark suite for DNNs, released by Baidu. For our experiments we used:

Bench.	Platform	Precision	Layers× Neurons	Time-step	Description
im2txt	DeepBench	16 bit	1×512	11	image caption
seq2seq	DeepBench	16 bit	3×1024	15	language translation
mach-tran	DeepBench	16 bit	1×512 1×1024 1×2048	25	Machine translation
lang-mod	DeepBench	16 bit	1×1536	50	language modeling
D-Speech	DeepBench	16 bit	1×2816	1500	Deep Speech

Table 2.3: Summary of the benchmarks evaluated.

Image Caption Generator: This benchmark is based on the "Show and Tell" Model [134], which is an encoder-decoder type neural network. The decoder is an LSTM RNN that generates captions from a fixed-length vector input.

Sequence-to-Sequence Model: This benchmark is based on the RNN encoder-decoder model by Cho et al. [25], which performs language translation. The encoder and decoder are 3-layer LSTM networks.

Machine Translation: also based on the RNN encoder-decoder model by Cho et al. [25].

Language Modeling: a probability distribution over sequences of words. It is used in speech recognition, sentiment analysis, information retrieval and other applications [108].

Deep Speech: a Speech-To-Text engine that uses a model trained by machine learning techniques, based on Baidu's Deep Speech research [53].

All benchmarks are run using 16-bit precision arithmetic.

2.6 Evaluation

We evaluate the RNNFast performance and energy consumption compared to the NVIDIA GPU, PUMA, the CMOS-based and the Memristor-based RNNFast design. We evaluate the reliability of the RNNFast error mitigation. We show an area utilization estimate for different benchmarks. We also include a high-level comparison to other RNN accelerators.

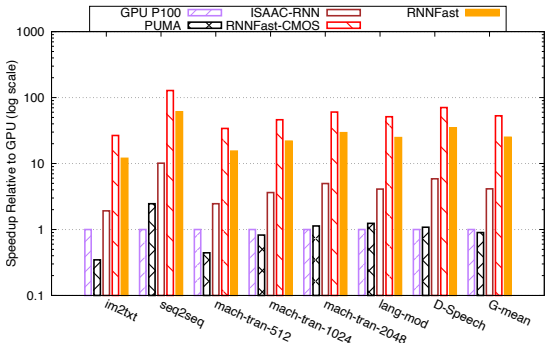

Figure 2.14: RNNFast, RNNFast-CMOS, ISAAC-RNN and PUMA runtime relative to the GPU P100 execution.

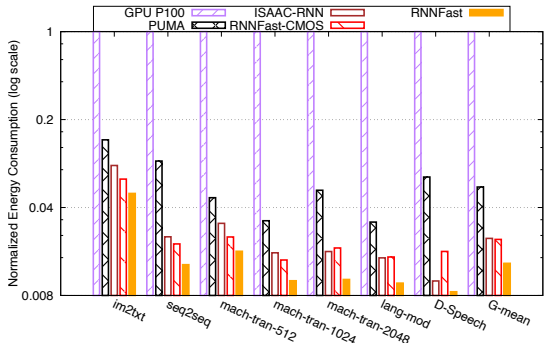

Figure 2.15: Energy consumption for RNNFast, RNNFast-CMOS, ISAAC-RNN and PUMA relative to the GPU P100.

2.6.1 Performance Improvements and Energy Savings

Figure 2.14 shows the execution time speedup for RNNFast, RNNFast-CMOS and ISAAC-RNN relative to the P100 GPU for the seven benchmarks we run. RNNFast speedup relative to the GPU varies between 12× for *im2txt* and 34.5× for *D-speech*, with an average speedup of 21.8×. RNNFast speedups increase with the network size, demonstrating the excellent scalability of the design. For instance, in *mach-trans* we test three different network sizes ranging from 512 to 2048, We observe speedups

increase from 15.4× to 29.3×. This is because the large number of threads required to handle the larger network becomes a bottleneck even for the GPU, whereas RNNFast scales much better.

ISAAC-RNN also brings a substantial speedup relative to the GPU ranging between 1.88× for *im2txt* and 5.8× for *D-speech*. Although this is significant, ISAAC-RNN is more than 6.1× slower than the DWM RNNFast implementation. This is primarily due to the higher latency of the LSTM unit in ISAAC-RNN, which is 7.3× higher than a RNNFast LSTM unit. The higher latency is due to the memristor array read latency (100ns) and overheads that stem from the ADC/DAC components. Even though a single memristor array can handle up to 4 neurons, which increases throughput, ISAAC-RNN is still fundamentally slower than RNNFast. RNNFast-CMOS shows 2.1× speedup compared to RNNFast. This is due to faster CMOS adders and random memory access instead of the shift-based access in RNNFast.

The PUMA ReRAM-based design is more general than ISSAC and RNNFast, supporting both CNNs and DNNs. However, its performance is lower than both ISAAC-RNN and RNNFast. In general, PUMA tends to have better performance than the GPU for larger networks, especially for multi-layer networks (seq2seq) where PUMA benefits from its pipelined architecture.

Figure 2.15 shows the energy consumption for RNNFast, RNNFast-CMOS and ISAAC-RNN relative to the GPU, on a log scale. RNNFast reduces energy consumption on average by 70×. This is due to a much faster execution time achieved with about 1/3 the power of a GPU. The RNNFast-CMOS design has 55% higher energy compared to RNNFast. This reaches a 100% increase for *D-speech* due to higher resource demand, which increases the leakage energy for both compute and memory

logic in CMOS. This causes the CMOS design to reach its maximum thermal design power (TDP) at smaller network sizes. ISAAC-RNN also has higher energy usage than RNNFast due to its ADC/DAC and CMOS logic. PUMA energy consumption is much lower than the GPU. However, as expected, it is not lower than ISAAC-RNN. RNNFast is much more energy efficient, using about 25% the energy of PUMA.

RNNFast offers a much more scalable design relative to a GPU due to its modularity and very high storage density of DWM. Figure 2.16 shows the log scale of execution time for the *mach-tran* benchmark as a function of problem (neural network) size ranging from 128 nodes to 16K nodes per layer in a single-layer configuration. For problem sizes larger then 16K, the GPU runs fail because the device runs out of memory. The GPU execution time exhibits a super-linear increase in execution time with problem size due to memory pressure. RNNFast is consistently faster than the GPU with an improvement that ranges from 13.9× (0.5K) to 156× (16K). RNNFast also scales better to very large problem sizes of 16K nodes and beyond. ISAAC-RNN also scales well, but it is 6.2× slower than RNNFast on average for *mach-tran*. RNNFast-CMOS shows almost 2× speedup over RNNFast. However, this speedup comes at the cost of a much higher energy.

Figure 2.17 shows a similar trend for *im2txt*. The GPU shows good performance up to 0.5K, but runtime increases exponentially beyond that.

2.6.2 Error Mitigation

We also evaluate RNNFast resilience to position errors. Figure 2.18 shows the accuracy of the output as evaluated by the BLEU metric [104], as a function of the probability of position errors. We can see that for a relatively low probability of errors

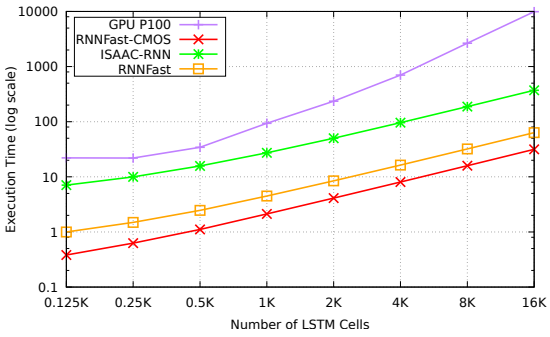

Figure 2.16: RNNFast, ISAAC-RNN and GPU execution times vs. network size for *mach-tran*, normalized to RNNFast 0.125K.

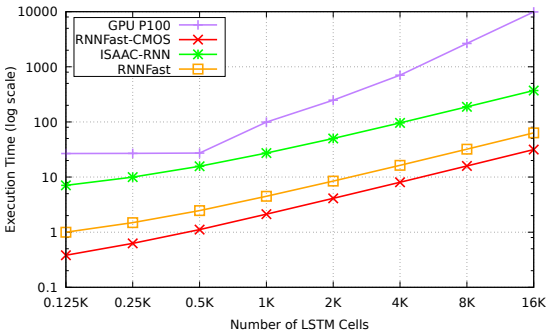

Figure 2.17: RNNFast, ISAAC-RNN and GPU execution times vs. network size for *im2txt*, normalized to RNNFast 0.125K.

of 4.5×10^{-7}, the output accuracy is virtually unaffected. This is primarily due to the inherent robustness of the RNN to errors. However, without error mitigation, the output accuracy degrades substantially at higher errors rates. In the region around 4.5×10^{-5} (highlighted region), which is the expected rate for single bit position errors, the output accuracy drops to 45% for *im2txt* and 10% for *seq2seq*, an unacceptable performance for most applications. When RNNFast error mitigation is enabled, the drop in output accuracy is negligible at less than 2%.

The RNNFast error mitigation produces outputs with less than 5% accuracy loss even for much higher error rates of 10^{-3} or around 20% accuracy loss for 10^{-2}. This shows that RNNFast EDC is robust to much higher error rates than what is expected for DWM technology.

It is also worth highlighting the fact that error mitigation incurs no performance penalty even when errors are detected. Correction or mitigation are performed without stalling the execution pipeline. This is an important design consideration because

of the highly synchronized nature of the design. A single stall to correct an error would result in lost cycles for thousands of functional units.

2.6.3 Nonlinear Function Hardware

We evaluate two designs for the nonlinear function hardware: a LUT-based implementation, and an approximate logic function-based unit. The function-based implementation is area efficient since it does not require as much storage as the LUT-based design. While the function-based implementation is slower than the simple lookup of the LUT version, the activation functions are not a significant latency bottleneck. The advantage for our design is the area reduction. At this scale we have thousands of nonlinear units on chip and reducing their area adds up to real savings.

Figure 2.19 shows the storage savings and performance degradation of the function-based sigmoid/tanh relative to the LUT design for multiple network sizes. The storage savings diminish as the network size increases because the storage space for the weights dominates. For large networks the storage savings are about 4%, which represents ¿1GB of DWM for a 16K network. As for the performance cost, it starts at about 9%, but falls below 1% for larger networks. The approximated nonlinear function does not result in loss of accuracy as measured by the BLEU score.

2.6.4 RNNFast Parameter Tuning

We also conduct a sensitivity analysis on the number of LSTM units per tile. Figure 2.20 illustrates the tile input buffer energy versus different number of LSTMs per tile for different network sizes. As the number of LSTMs per tile increases, the power/area overhead for the within tile bus increases super-linearly. The minimum

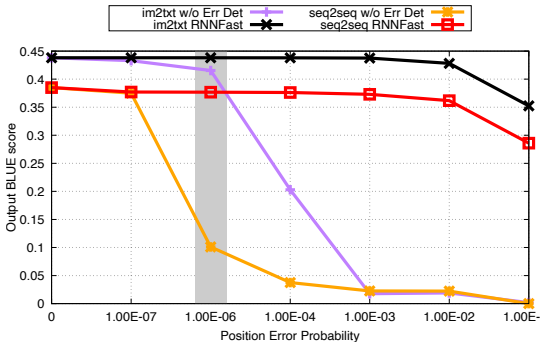

Figure 2.18: Output accuracy for benchmarks *im2txt* and *seq2seq* with and without RNNFast EDC.

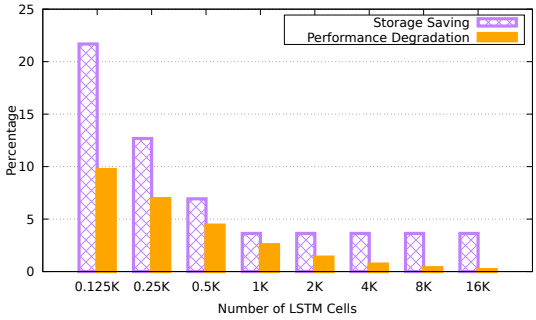

Figure 2.19: Storage saving and performance degradation for different network sizes for Approx. Function-based sigmoid design relative to LUT.

energy point is different depending on the size of the network. The 64 LSTM units per tile represents a reasonable compromise for medium-to-large networks.

2.6.5 Comparison to Other RNN Accelerators

Several recent papers have proposed FPGA-based accelerators for RNNs [124, 77, 75, 142, 152, 38, 39, 91, 51]. We provide a qualitative comparison with some of the more recent ones, for which runtime and energy numbers were available and similar applications were evaluated. Table 2.4 summarizes the energy and runtime for FPGA-based designs from [38, 91, 51, 39] as well as the energy and runtime of RNNFast while running networks of equivalent size.

The networks used in [38, 91, 51] vary from vary small to large. RNNFast shows from 4.7× to 64× speedup. Compared to [38] RNNFast has 19× less energy consumption.

Recently Fowers et al.[39] introduced Brainwave, an FPGA-based accelerator for RNN with no batching for real time AI. While a very efficient design, Brainwave has

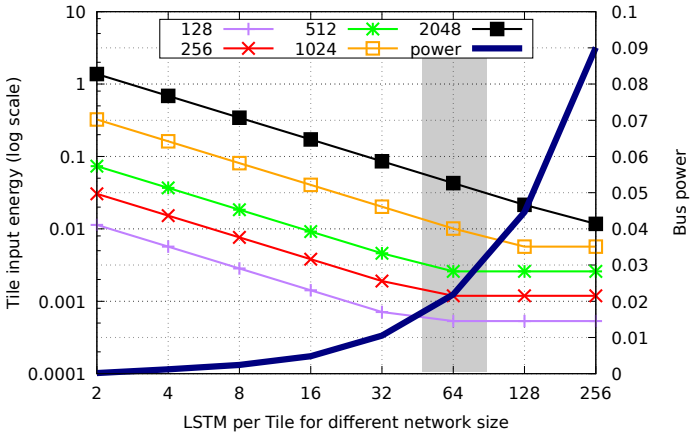

Figure 2.20: Sensitivity analysis for the number of LSTMs per tile.

50-70% higher energy energy than RNNFast. Brainwave also shows poorer performance for smaller networks, but slightly better performance for large ones, compared to RNNFast. Note that this is not a quantitative apples-to-apples comparison to our design given that Brainwave uses 8 bit precision (vs 16 bit for RNNFast) and a 14nm techology node (vs. 32nm for RNNFast).

FPGA Design	Net size	Timesteps	run time(μs)	energy (μJ)	RNNFast run time (μs)	RNNFast energy (μJ)
[38]	32	1	1.586	0.8	0.332	0.0419
[91]	256	7735	42.48E3	NA	2.13E3	1.28E3
[51]	1024	1	82.7	NA	1.29	12.8
[39]	256-1k-2K	150-25-25	425-74-74	Est.: 425-1091-4356	117-58-110.7	252-643-2575

Table 2.4: Energy and run time for FPGA-based RNNs.

The Google TPU is also capable of running RNN workloads efficiently. In [65] they report up to 8× better performance for LSTM workloads compared to NVIDIA

K80. RNNFast is up to 260× faster than the newer NVIDIA P100 for workloads of similar size.

2.7 Conclusion

The unprecedented growth of available data is accelerating the adoption of deep learning across a wide range of applications including speech recognition, machine translation, and language modeling. In this study, we present RNNFast, a novel accelerator designed for recurrent neural networks. Our design demonstrates that using domain wall memory is not only feasible, but also very efficient. We compare RNNFast with a state-of-the-art P100 NVIDIA GPU and find 21.8× better performance with 70× lower energy.

Chapter 3: Accelerator Architecture for ML Security

Convolutional neural networks have demonstrated high accuracy on various tasks in recent years. However the are extremely vulnerable to adversarial examples. For example, imperceptible perturbations added to clean images can cause convolutional network to fail. Figure 3.1 shows two examples of adversarial images generated using the state-of-the art CW-L_2 attack [18]. The leftmost images are benign, unmodified samples. They are correctly classified by a DNN model such as VGG16 with 99% and 87% confidence, respectively. The middle and rightmost pairs of images represent the output of two versions of the CW-L_2 attack, each resulting in misclassification. Note that all adversarial images are virtually indistinguishable from the original to the casual observer, even though the confidence of the classifier in all cases is very high.

Several defenses have been proposed to address adversarial attacks [86, 102, 146, 34, 15, 87]. Most rely on purely software implementations, with high overheads, limiting their utility to real-world applications. A recent line of research has explored hardware-assisted approximate computing to introduce controlled errors into the inference process, either through model quantization [41, 101] or approximate computation [50]. This inference approximation disrupts the effect of the adversarial

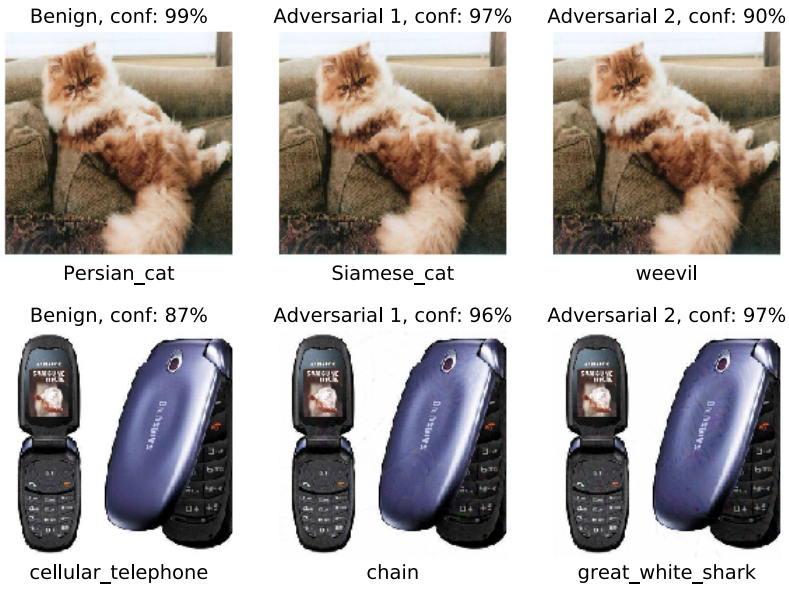

Figure 3.1: Benign vs Adversarial Images.

modifications, making the attacks less likely to succeed. At the same time, if the errors are kept small, approximate inference tends to have less effect on benign inputs' classification accuracy.

We investigate the scalability of noise based and defensive approximation approaches to a broader class of attacks. We find that, while approximation methods work well for some inputs, they do not scale well to strong adversarial attacks that are trained to have high classification confidence. This is because the noise introduced through approximation is insufficient to reverse the adversarial effects. We also show that, even if noise is increased, full recovery of strong adversarials is less likely. We therefore argue that defensive techniques should focus on detecting adversarial inputs, which has higher probability of success, rather than recovery of the original class. A key observation we make in this work is that **tailoring the approximation error**

rate to the confidence of the input classification dramatically increases the adversarial detection rate, while at the same time maintaining a low false positive rate for benign inputs. This is the first work to recognize the importance of this correlation for accurate adversarial detection.

In this chapter, first we present Hardware-Accelerated Stochastic Inference (HASI), a defense that relies on a novel stochastic inference process to effectively defend against state-of-the art adversarial attacks. We show that HASI can easily generalize to different attacks, while requiring no changes to the DNN models. We observed that, under certain conditions, adversarial inputs can be identified based on the DNN's response to the injection of random noise into the network. We show that by injecting small amounts of noise into the activation step of select convolution operations, we can identify adversarial inputs with high accuracy and low false negative rate.

While prior work has similarly explored ways of discriminating adversarial inputs by randomly perturbing either inputs or model[114, 27, 55, 74], our approach is different. The main idea behind HASI is to inject noise throughout the model and run multiple inference passes, each pass with a different noise distribution. We call this process *stochastic inference*. We then use the distribution of classification outputs to determine if the input is potentially adversarial.

HASI requires multiple inference passes, potentially increasing inference latency. To mitigate this overhead we augment a hardware accelerator design with hardware that speeds up the main HASI functions, including hardware support for random noise injection, the efficient reuse of intermediate results, custom adversarial detection function, etc.

Later in this chapter, inspired by HASI, we present DNNSHIELD, a hardware/software co-designed defense that takes a different approach to inference approximation and addresses some of the limitations of the previous approach to reduce the overhead more with less tuning parameters. DNNSHIELD is an online adversarial detection framework that uses the effects of model sparsification to discriminate between adversarial and benign inputs. Same as HASI, DNNSHIELD runs both precise and sparse inference passes for each input and compares their output. It then uses the deviation in output classification that is triggered by the sparsification to classify the inputs as benign or adversarial.

Unlike prior work, DNNSHIELD **dynamically** and **randomly** varies the approximation error and distribution. Dynamic approximation error is needed to adapt to the confidence of diverse inputs. Randomness in the error distribution is crucial in ensuring that adversaries cannot re-train to account for predictable inference noise.

To achieve these goals DNNSHIELD uses hardware-assisted dynamic and random model sparsification to implement approximate inference. Model sparsification involves dropping weights from the model, and has been used to improve performance and energy efficiency [31, 44]. DNNSHIELD controls the sparsification rate dynamically to enable flexible control over the approximation error. Sparsification is also random to make the noise input independent and consequently training defense-aware attacks difficult.

DNNSHIELD demonstrates robust detection across a broad set of attacks, with high accuracy and low false positive rate. We show an adversarial detection rate of 86% when applied to VGG16 and 88% when applied to ResNet50, which exceeds the

detection rate of the state of the art approaches. We also show that DNNSHIELD is robust against attacks that are aware of our defense and attempt to circumvent it.

The accelerator design builds explicit support for dynamic and random model sparsification. The DNNSHIELD accelerator is optimized for efficiently executing sparsified models in which the sparsification rate changes as a function of the input – which is more challenging compared to models for which weight sparsity is fixed. We show that the DNNSHIELD accelerator reduces the performance impact of approximate inference-based adversarial detection to $1.53\times - 2\times$ relative to the unprotected baseline, compared to 15×–25× overhead for a software-only GPU implementation.

The rest of this chapter is organized as follows: Section 3.1 provides background information. Section 3.2 discusses related work. Section 3.3 explains the threat model. Section 3.4 focuses on high confident adversarial examples and how prior approximate methods fail to detect such anomalies. Section 3.5 presents the fundamental of the detection mechanism used in our work. Section 3.9 and 3.7 present the details of the HASI and DNNSHIELD designs. Section 3.8 provide the information of the FPGA implementation of both designs. Finally section 3.9 and 3.10 shows the evaluation results and Section 3.11 concludes the chapter.

3.1 Background

3.1.1 Adversarial Attacks

Adversarial attacks were first introduced by Szegedy et al. in [126], which showed that despite the high accuracy of machine learning models, small perturbations to inputs can reliably force misclassifications–while the perturbed input remains indistinguishable from the original seed to the naked eye. This phenomena has other intriguing properties, for instance, carefully crafted perturbations may result in very high confidence for these coerced incorrect predictions; additionally, perturbed inputs, or *adversarial examples*, can be *transferable*, causing a deliberate misclassification across many different networks, even when these networks have different architectures and training sets.

The objective of an adversarial attack is to force the output classification for some maliciously crafted input $\mathbf{x}'$, based on a benign input $\mathbf{x}$, to be incorrect with respect to $\mathbf{x}$. Attacks can be *targeted*, where the adversary's goal is for $\mathbf{x}'$ to be misclassified as a particular class t, or *untargeted*, such that a misclassification of $\mathbf{x}'$ to any class other than the correct class of $\mathbf{x}$ (ground truth) is sufficient. Targeted attacks were formally defined by Szegedy et al. [126] as solving the following optimization problem:

$$\min \; d(x, x+\delta) \qquad \text{subject to: } \; C(x+\delta) = t \qquad x+\delta \in [0,1]^n, \tag{3.1}$$

where δ is the added noise, t is the desired target label for the adversarial example produced by $x+\delta$, and d is a metric to measure distance between the benign example and the adversarial one. Effective adversarial example generation is achieved by coercing prediction results while maintaining similarity to the original image, where similarity

is measured by a chosen distortion metric that is ideally capable of approximating human perception.

Early work on adversarial example generation for image classifiers proposed one-step methods, which make a single (relatively large) change to the input and consequently are only applicable in the untargeted setting. Fast Gradient Sign Method (FGSM) [46] is one such attack, which perturbs an image by maximizing the loss subject to some distortion constraints. Perturbations are calculated using the gradient vector of the training loss with respect to the benign input:

$$\Delta(x, x') = \epsilon \cdot \text{sign}(\nabla_x \, J(g(x), y)) \tag{3.2}$$

In Equation (3.2), ϵ specifies the allowable magnitude of distortion between $\mathbf{x}$ and $\mathbf{x}'$, and sign($\nabla \, J(\cdot, \cdot)$) is the sign of the gradient. $\Delta(\cdot)$ is typically an L_p-norm distance metric–the selection of optimal similarity metrics is still an open research problem. Three L_p norms have been consistently used in existing adversarial attacks: L_∞, L_2 and L_0. The L_∞ norm measures the maximum change in any dimension and will be applied uniformly to all pixels, the L_2 norm measures the Euclidean distance between $\mathbf{x}$ and $\mathbf{x}'$ and the L_0 norm measures the total number of pixels that may be altered, but not the magnitude of perturbation.

Subsequently, several attacks were proposed [46, 18, 88, 103, 93, 73] with variations on this theme, gradually becoming more iterative in nature, with increased sophistication and accuracy. Some examples include Basic Iterative Method (BIM) [73], proposes a refinement to FGSM suggesting that instead of taking a single step of size ϵ in the direction of the gradient-sign, to take multiple, smaller steps, where the result is clipped by ϵ. DeepFool [93] is an untargeted attack optimized for the L_2 distance metric which imagines that the network is linear, in which case a hyperplane

separates classes from one another. From this simplified formulation, the authors analytically derive the optimal solution, which is the minimum noise required to move the adversarial example to the linearized classification boundary. Jacobian-Based Saliency Map Attack (JSMA) [103] is a targeted attack optimized for the L_0 distance metric, using the gradient to compute a *saliency map* representing, for each pixel, the likelihood that changing it will result in the image being classified as a target class l. Given this saliency map, an algorithm modifies the most important pixels at each iteration to increase the probability that the input will be classified as l.

The Carlini-Wagner (CW) attack differs from prior attack formulations in several ways which allows it to finds adversarial examples with considerably smaller perturbation amounts and higher accuracy. CW has a variant for each of the popular L_p-norm distance metrics. The **CW-L_2** attack reformulates Equation (3.1), noticing that it is expressed in a way not easily amenable to optimization solvers,

$$\text{minimize} \;\; d(x, x+\delta) + c \cdot f(x+\delta) \text{ such that } \;\; x+\delta \in [0,1]^n \tag{3.3}$$

In Equation (3.3), c is a constant weighting the relative importance of the distance and loss terms and $f(\cdot)$ is the objective function. The authors found that a logits-based objective function resulted in superior adversarial examples compared to the commonly-used softmax-cross-entropy loss in previous optimization-based attacks. To ensure that modifications result in a legal image, constraints must be placed on δ, where $0 \leq x_i + \delta_i \leq 1$ for all i. To more easily optimize for these constraints, CW further proposes to employ the change-of-variables (COV) method with a *tanh* space transformation on $\mathbf{x}$ in order to remove the box-constraint. Now, instead of optimizing over the variable δ, optimization is done over a new variable w, setting

$$\delta_i = \frac{1}{2}(\tanh(w_i) + 1) - x_i. \tag{3.4}$$

Since -1 $\leq \tanh(w_i) \leq 1$, it follows that $0 \leq x_i + \delta_i \leq 1$, therefore the solution will automatically be valid. Combining these insights, CW is able to obtain a method for finding adversarial examples with competitively low distortion in the L_2 metric. Formally, the CW-L_2 attack can be expressed as the following optimization problem: Given an input x and a target class t, the attack searches for w that solves,

$$\begin{aligned} &\text{minimize} \quad \|\frac{1}{2}(\tanh(w)+1) - x\|_2^2 + c \cdot f(\frac{1}{2}(\tanh(w)+1) \\ &\text{with } f \text{ defined as:} \\ &f(x') = \max(\max\{Z(x')_i : i \neq t\} - Z(x')_t, -\kappa). \end{aligned} \tag{3.5}$$

The parameter κ in Equation (3.5) controls the confidence with which the misclassification occurs.

CW also has variants for the L_0 and L_∞ distortion metrics. The **CW-L_0** attack takes a different approach, as the L_0 distance metric is non-differentiable and ill-suited for gradient descent. CW-L_0 employs an iterative algorithm that, at each iteration, uses the CW-L_2 attack to identify pixels which do not contribute much to the classifier output and freeze those pixel values. At the end, by process of elimination, the minimum sets of pixels which require modification in order to generate an adversarial example have been identified. The **CW-L_∞** attack uses the L_∞ distance metric, which also is not fully differentiable. The authors found that it is difficult to optimize because only the maximum term is penalized during gradient descent. As a result, CW-L_∞ revises the objective function to limit perturbations to be less than a threshold. The attack iteratively searches for the smallest possible threshold which still produces a solution.

The **EAD** attack [20] formulates the objective function in terms of regularized elastic-nets, incorporating the L_1 distance metric for distortion, where elastic-net

regularization is a linear mixture of L_1 and L_2 penalty functions. EAD has two variants, where the optimization process can be tuned by two different decision rules: *EN*, the elastic-net loss or L_1, the least L_1 distortion.

3.1.2 Robustness in Neural Networks

Robustness can be informally defined as the measure of how difficult it is to find adversarial examples close to their original inputs. There have been many theoretical contributions in this area [35, 33, 117, 89, 35], which have studied the underlying data distributions on which networks are trained in order to determine intrinsic properties of robustness to adversarial examples. Several methods for designing robust neural networks to adversarial attacks have been proposed in the literature. These methods typically fall into four broad categories [5]:

1. **Hardening the model**, also known as *adversarial training*. In this case, the defense is limited by the samples used in training.

2. **Hardening the test inputs**, also known as applying *input transformations*, such as filtering or encoding the image.

3. **Adding a secondary, external network** to classify unseen examples or features of intermediate layers. This approach generally requires careful tuning that may affect generality.

4. **Modifying the network post-training**, such as removing/adding layers, tweaking activation functions, etc.

Approaches which modify or extend the network can be further categorized based on whether they aim to (a) provide a correction, that is, continue to provide the

correct classification in the presence of adversarial inputs; or (b) aim to provide detection, rejecting suspicious inputs. We will discuss some of the important works in section 3.2.

3.2 Related Work

3.2.1 Noise-Based Approaches

Prior work has similarly explored ways of discriminating adversarial inputs using noise. However, prior approaches have either proposed injecting noise into the input [27, 15] – with lower detection rate – or into the model during training [114, 55, 74, 116]. However, the challenge with training-based approaches such as [55] is that the noise parameters tend to converge to zero as training progresses, making the noise injection progressively less effective over time [63]. While training-based approaches have enabled "certified robust" inputs, that certification is generally limited to a very narrow set of inputs.

Region-based Classification (RC) [15] assembles information from the region around an example to predict its label. When predicting a label for an image, m data points are sampled uniformly at random from the hypercube centered at the testing example with distance r. The final classification is given by the majority vote of the outputs among the sampled data points. This method has high overhead and is focused on correction, with a lower success rate.

Recent works also created certified robust models by training models under Gaussian noise injection into the inputs [27] or the model [55, 74]. While these methods represent a systematic solution to adversarial attacks, they are limited to certain perturbation norms (e.g L_2) and do not scale for large-scale dataset like ImageNet.

Liu et al. [83] propose a solution for adversarial scene detection in robotics, based on a generative sampling-based search where each sample represents some possible distortion effects. The final classification is then determined based on the likelihood

of a particular object under these different effects, not simply from the traditional hard-threshold of the network from a single sample.

Stochastic Activation Pruning (SAP) [34] introduces randomness into the evaluation of a neural network to defend against adversarial examples. SAP is a complete defense tries to recover the accuracy of the model by classifying the adversarial correctly. SAP randomly drops some neurons of each layer to 0 with a probability proportional to their absolute value. Values which are retained are scaled up to retain accuracy. SAP is a single forward path execution and does not use any voting or averaging mechanism.

Hardening the test inputs, also known as applying *input transformations*, such as filtering or encoding the image. Input hardening methods require profiling to select appropriate parameters, such as Feature Squeezing [146] and Path Extraction [110, 42]. Xie et al. [143] propose to defend against adversarial examples by adding a randomization layer before the input to the classifier. The defense first randomly rescales the image, and then randomly zero-pads to fit the expected dimensions. [42] showed that adversarial inputs tend to activate distinctive paths on neurons from those of benign inputs. They proposed hardware accelerated adversarial sample detection, which uses canary paths from offline profiling.

Feature squeezing (FS) [146, 78] is a correction-detection mechanism that relies on the observation that the input feature space is typically unnecessarily large and provides ample opportunity for constructing adversarial examples. The strategy taken in this work is to limit the input space by removing features. Two feature reduction techniques are evaluated for "squeezing" images: reducing the color depth and smoothing to reduce the variation among pixels. Detection of adversarial images

is achieved by comparing the output of the network using the original image with the output of the squeezed image(s).

3.2.2 Approximation-Based Defenses

Some prior works have used approximate inference to improve model robustness against adversarial attacks using approximate compute logic [50] or quantization [101, 40, 111, 68, 80]

Recent work has proposed using hardware-based approximation methods as similar defenses. Guesmi et al. [50] proposed "Defensive Approximation" (*DA*) which used custom approximate multipliers, to introduce controlled errors into a CNN accelerator. Similarly, Fu et al. [41] used hardware-assisted parameter quantization as the approximation mechanism. Model quantization is the process of reducing the precision of the model parameter representation, and has been used to improve performance, energy and storage efficiency of DNNs. In [41] a *2-in-1* hardware accelerator dynamically chooses between 12 quantization levels to use at inference, introducing approximation into the model. While these approaches are effective and have low overheads, they use either fixed approximation error [50] or randomly-selected error from a limited set of up to 12 precision levels [41]. In addition, both techniques generate input-dependent noise, which an attacker could reproduce to circumvent the defense.

3.2.3 Other Defense Methods

Wang et al. [141] propose to interpret Neural Networks by identifying Critical Data Routing Paths (CDRP) which leverages class-level sparsity and can be used as adversarial detection mechanisms. CDRP requires retraining and thus is not able

to detect adversaries at inference-time. Deepfense [115] proposed by Rouhani et al. represents a class of detection mechanisms that use modular redundancy and which requires training a set of redundancy modules (checkpoints) to isolate potential adversarial sub-spaces in the Pre-processing phase. Deepfence employs multiple latent models as redundancies.

Recently [42] showed that adversarial inputs tend to activate distinctive paths on neurons from those of benign inputs. They proposed hardware accelerated adversarial sample detection framework, which uses canary paths generated from offline profiling to detect adversarial samples at runtime based on the work in [110]. Paths are unique per class and need to be profiled offline and stored in memory. Also switching to different models requires re-profiling. In contrast, HASI does not need profiling. Unlike HASI, [42] did not study high confidence adversarials.

Local Instrinsic Dimensionality (LID) [87] detects adversarial examples by comparing the expansion rate of local distance distributions. The expansion of adversarial subspaces was empirically found to be *higher* than normal data subspaces; in other words, adversarial images were found by the authors to have higher dimensionality.

Adding a secondary, external network solely responsible for adversarial detection and with a separate training phase, such as NIC [86]. DNNGuard [138] proposed an accelerator for such detection mechanism but has not evaluated a specific detection classifier. Secondary network detection-based methods are not as effective, and can be evaded by adaptive attacks [17].

3.3 Threat Model

We assume in this work that the adversary has complete access to the network, including the output prediction and logits, with full knowledge of the architecture and parameters, and is able to use this in a white-box manner. We focus mainly on state-of-the-art optimization-based attacks–CW and EAD–since it has been demonstrated that all earlier attacks can be overcome utilizing other methods, such as adversarial training [45] or defensive distillation [102], which could be used in combination with our approach. Additionally, we verify our evaluation includes high confidence adversarial examples, as some previously proposed defenses were later shown to perform poorly under a more holistic treatment which included these [85].

3.4 Motivation

Strong attacks such as CW and EAD can be tuned to produce a class of adversarial inputs that present a significant challenge to approximation-based defenses. Prior work has shown that adversarial inputs can be constructed to induce misclassification with very high confidence [119, 16, 18]. In other words, the victim model assigns a very high probability to the adversarial input belonging to the wrong class. These so-called "high confidence" adversarials can be constructed while minimizing the distortion to original input.

3.4.1 High-Confidence Attack Variants

Figure 3.2 shows an example of multiple adversarial samples for a benign image with different levels of classification confidence and the corresponding distortion. To measure classification confidence we used the Z-score (the number of standard deviations by which the value of a raw score is above or below the mean value) of the maximum logit value, which corresponds to the class with the highest confidence. Adv_1 is a low distortion adversarial of the benign with low classification confidence of 4.18. Adv_2 is a high confidence example of the same input with very high classification confidence of 12. While distortion is also higher, it is still imperceptible to the naked eye. We will show that existing defenses are ineffective against this type of adversarial. Increasing the confidence beyond 12 increased distortion significantly, as Adv_3 shows.

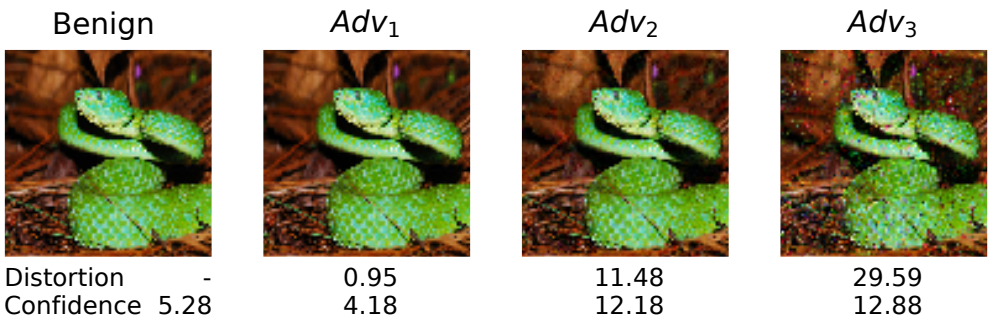

Figure 3.2: Benign and adversarial examples with various distortion and confidence values.

3.4.2 Classification Confidence and Approximation

Approximate computing defense methods introduce noise into the input and/or the model and often result in the recovery of the original class. Figure 3.3 schematically illustrates how approximation can recover adversarial inputs. Figure 3.3 illustrates the decision space of a classifier, with four output classes: C_1, C_2, C_3 and C_4, represented as regions with different colors. The darkness of the color represents the confidence of the classification. We consider a benign input X_1 classified with low in class C_1, and an adversarial sample $Adv_1(X_1)$ that is misclassified into class C_2 with low confidence. The dotted circles around each input represent the range of classification outcomes as a result of noise.

Prior work has observed that high-quality adversarial inputs occur with low probability, which means they reside in small and low-density pockets of the classification space. As a result, their output class distribution differs from that of their closest data submanifold [37]. We can see in Figure 3.3 (a) that $Adv_1(X_1)$ resides in a narrow cone of class C_2, where benign images do not generally exist. This means that

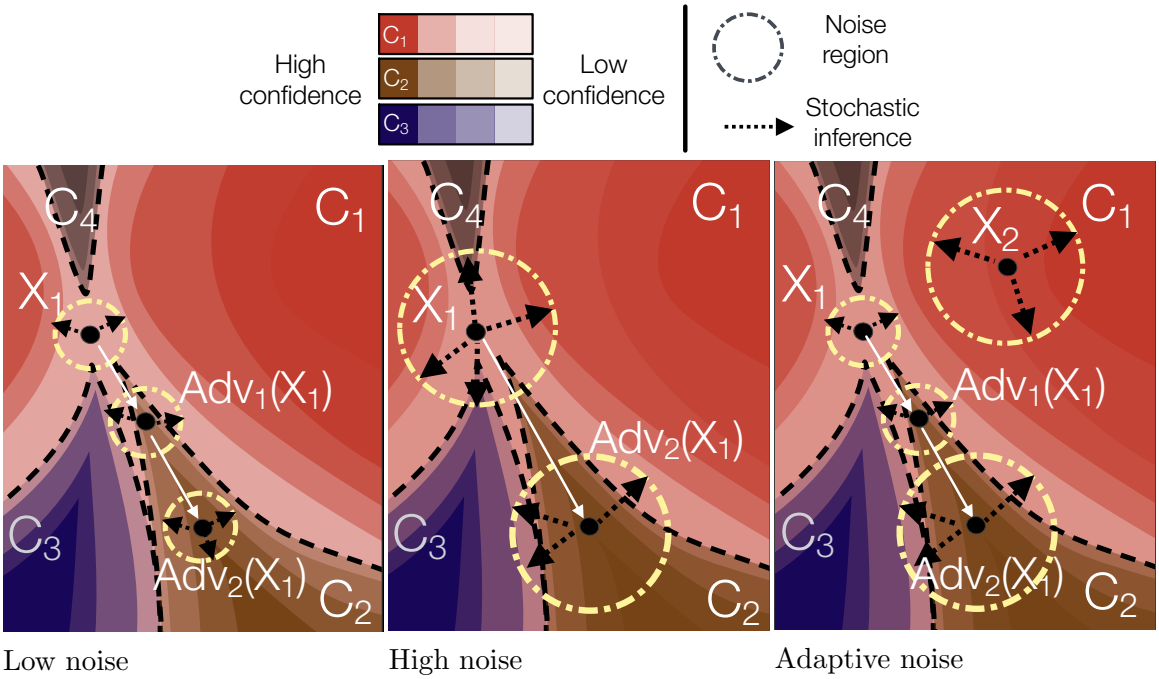

Figure 3.3: Schematic illustration of decision regions of a base classifier. Different classes are drawn in different colors with darker shades indicating higher confidence. X_1 and X_2 are benign inputs, $Adv_1(X_1)$ and $Adv_2(X_1)$ are low confidence and high confidence adversarial inputs generated from X_1. The circles shows range of classification deviation under a certain amount of noise.

approximation can easily change the classification of $Adv_1(X_1)$ and move it back into its original class, C_1.

In this work we show for the first time, that high-confidence adversarials do not respond in the same way to approximation errors. To illustrate why, let's consider $Adv_2(X_1)$, a high-confidence adversarial of X_1. As Figure 3.3 (a) shows, for the same error, the classification area of $Adv_2(X_1)$ lands within region C_2, failing to recover the original class C_1.

Figure 3.3 (b) shows that, if we increase the approximation error, the probability of recovering $Adv_2(X_1)$ increases. However, if the same approximation error is applied uniformly to X_1, there is an increased probability that X_1 will be misclassified, resulting in false positives.

In order to address this limitation, we propose *correlating the approximation error to the confidence of the classification*. Figure 3.3 (c) illustrates this with circles of different radii: small radius corresponding to lower approximation error for X_1 and $Adv_1(X_1)$ – which are low confidence classifications – and larger radius (approximation error) for $Adv_1(X_1)$ and a high confidence benign, X_2. We can see that the low error does not lead to misclassification for X_1, while recovering $Adv_1(X_1)$ with high probability. At the same time, a high error rate will not lead to misclassification for X_2, while increasing the probability of recovering $Adv_2(X_1)$.

3.4.3 Approximation-Based Defenses

Prior work has used input noise and approximate inference to improve model robustness against adversarial attacks. For example, [58] and [27] have shown that adding some amount of random noise to images can help DNNs correctly classify adversarial inputs.

Recent work has proposed using hardware-based approximation methods as similar defenses. Guesmi et al. [50] proposed "Defensive Approximation" (DA) which used custom approximate multipliers, to introduce controlled errors into a CNN accelerator. Similarly, Fu et al. [41] used hardware-assisted parameter quantization as the approximation mechanism. Model quantization is the process of reducing the

precision of the model parameter representation, and has been used to improve performance, energy and storage efficiency of DNNs. In [41] a *2-in-1* hardware accelerator dynamically chooses between 12 quantization levels to use at inference, introducing approximation into the model. While these approaches are effective and have low overheads, they use either fixed approximation error [50] or randomly-selected error from a limited set of up to 12 precision levels [41]. In addition, both techniques generate input-dependent noise, which an attacker could reproduce to circumvent the defense.

In order to study the response of these approximation techniques to high confidence adversarials, we re-implemented both the *DA* and the *2-in-1* defenses for VGG16 and ResNet50 models. The original DA defense uses a single AMA5 floating-point based approximate multiplier. In order to explore a wider range of approximation errors we use a set of seven Int8 approximate multipliers from the *Evoapproxlib* library [98]. We created approximated models using approximate convolution (*AxConv*) implementations from [130, 99]. We also used the quantized model from *2-in-1* [41] and generated high confidence adversarial samples targeting two fixed quantization levels of 16 and 4-bit precision. We then tested the 2-in-1 defense on these samples. We generated several sets of adversarial samples with different classification confidences by changing the k parameter in the CW and EAD attacks.

Figure 3.4 (a) shows correction rate (percentage of adversarial examples that are correctly classified by the approximate inference) vs. adversarial confidence for two approximate multiplier versions *DA(125K)* and *DA(KEM)*, and the two *2-in-1* adversarial variants. For reference, we also include two software-only approximation

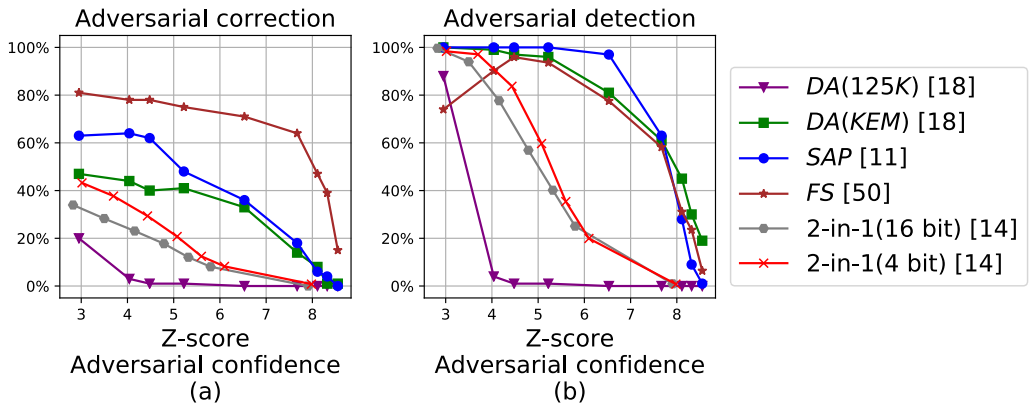

Figure 3.4: (a) Adversarial correction and (b) adversarial detection for different defense methods versus confidence of adversarial attacks on VGG16 and ResNet50.

techniques: Feature Squeezing (FS) [119] and SAP [34]. We can see that most approximation techniques perform relatively well with low and medium confidence samples. However, as adversarial confidence increases, correction rate drops below 20%. The software-only methods perform the best, but they also have the highest overhead. FS, which exhibits the highest correction rate, also has a 4× performance/energy overhead.

Given the low correction rate of defense methods for high confidence adversarial inputs, we also evaluated an adversarial detection approach. Except FS which has its own detection methodology, for detection we simply compared the classification outputs of the unprotected and protected models for the same input. If the outputs do not match, the input is classified as adversarial. The intuition behind this approach is that the classification of adversarial examples is more likely to change during approximate inference, although the output classification may not be the correct one. This is especially important for very strong adversarial examples which are far from decision boundaries (e.g. $Adv_2(X_1)$ in Figure 3.3), and approximation is unlikely to

recover the correct classification, as observed in [57]. However, as we will show in this paper, approximate inference is sufficient to change output distribution of all adversarials in a way that makes them detectable with high probability.

Figure 3.4 (b) shows adversarial detection for the same adversarial inputs and defense methods. We can see that the detection rate is much higher than correction for low-confidence adversarials, but still drops below 50% for adversarials with confidence greater than 8. FS performs better than other methods, but it is still inadequate for high confidence samples.

These results show that approximation methods used in prior work, which use fixed error rates are insufficient to detect high confidence adversarial examples.

Other Approximate Multipliers

We expand the study in [50] to include a range of approximate multipliers with different degrees of approximation errors. We aimed to study the potential of approximate multipliers as alternative sources of noise in an approximation-based defense. Figure 3.5 (a) shows correction rate (percentage of adversarial examples that are correctly classified by the approximate inference) vs. adversarial confidence. We show multiple approximated models using approximate multipliers with different error rates – measured using the mean square error (MSE). The MSE for each multiplier is shown in Figures 3.5 (b) and (d) on the left Y-axis on a log scale.

Figure 3.5 (a) shows that, as the adversarial confidence increases, the correction rate decreases for all the approximated multipliers. Although high MSE approximate multipliers have better correction rate (e.g. KEM, 14VP), all models have very low correction rate for high confidence adversarial examples. Note that increasing MSE

has the downside of degrading the accuracy of classification for benign examples, as Figure 3.5 (b) shows, and cannot therefore be increased further.

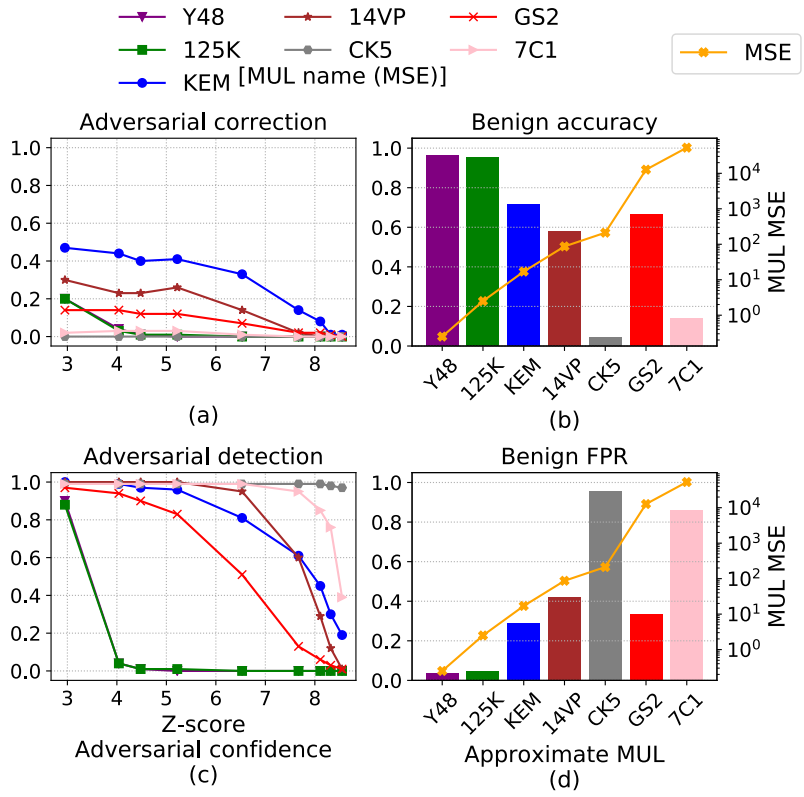

Figure 3.5: Adversarial correction (a) and detection rates (c) for different adversarial strengths. Benign accuracy (b) and FPR (d) for VGG16

Given the low correction ability of approximate multipliers for high confidence adversarial inputs, we also evaluated an adversarial detection approach. For detection we simply compared the classification outputs of non-approximated and approximated models for the same input. If the outputs do not match, the input is classified as adversarial. The intuition behind this approach is that the classification of adversarial examples is more likely to change during approximate inference, although the output

classification may not be the correct one. This is especially important for very strong adversarial examples which are far from decision boundaries, and approximation is unlikely to recover the correct classification, as observed in [57]. However, as we will show in this paper, approximate inference is sufficient to change output distribution sufficiently to be used in a detection mechanism.

3.4.4 The Case for Adaptive Approximation

The solution we propose in this work is to dynamically adapt the level of error/approximation to the confidence of the classification. To motivate an adaptive approximation over simply increasing the approximation error, we conduct an experiment in which errors are introduced directly into the model, in the activation layers. Figure 3.6 shows the adversarial detection rate and benign false positive rate (FPR) for fixed and adaptive error rates. We can see that fixed errors below 50% have very low detection rates. Increasing the injected error to 100% results in high detection rate, but at the cost of an unacceptable 80% false positive rate. The *Adaptive* error rate, correlated with the input confidence, achieves both high adversarial detection and low benign FPR.

The next research question we tackle is how to introduce a well controlled, variable and randomly distributed approximation error into the inference process in a performance-efficient way. Unfortunately, approximate computation using approximate functional units does not offer sufficient flexibility to tune the error rate since they are generally not easily tunable. The same is true for quantization methods, which do not provide sufficient granularity for the approximation errors. In addition,

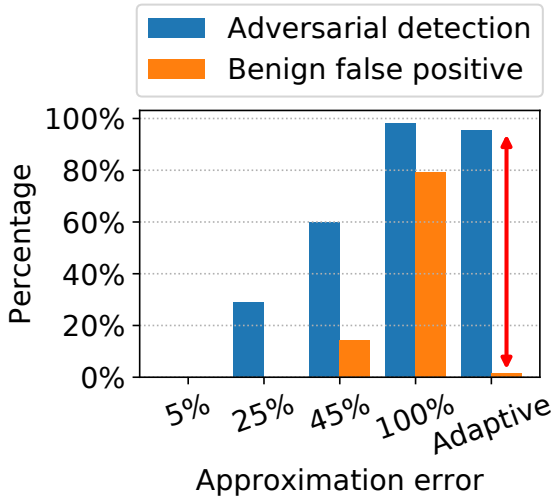

Figure 3.6: Adversarial detection and benign FPR for fixed versus adaptive inference errors in VGG16.

both quantization and approximate computation tends to be deterministic, producing predictable and reproducible error distributions that can be exploited by an attacker.

3.5 Dynamic Noise and Adversarial Detection

3.5.1 Impact of Noise Rate on Classification Output

In order to understand the impact of model approximation on classification output, we use the L_1-norm metric to measure the difference between noisy and non-noisy outputs for the same inputs. The L_1-norm (also known as Manhattan Distance) is the sum of the absolute pair-wise differences between elements of two vectors. We refer to this difference as the *Classification Probability Deviation under Noise* (CPDN).

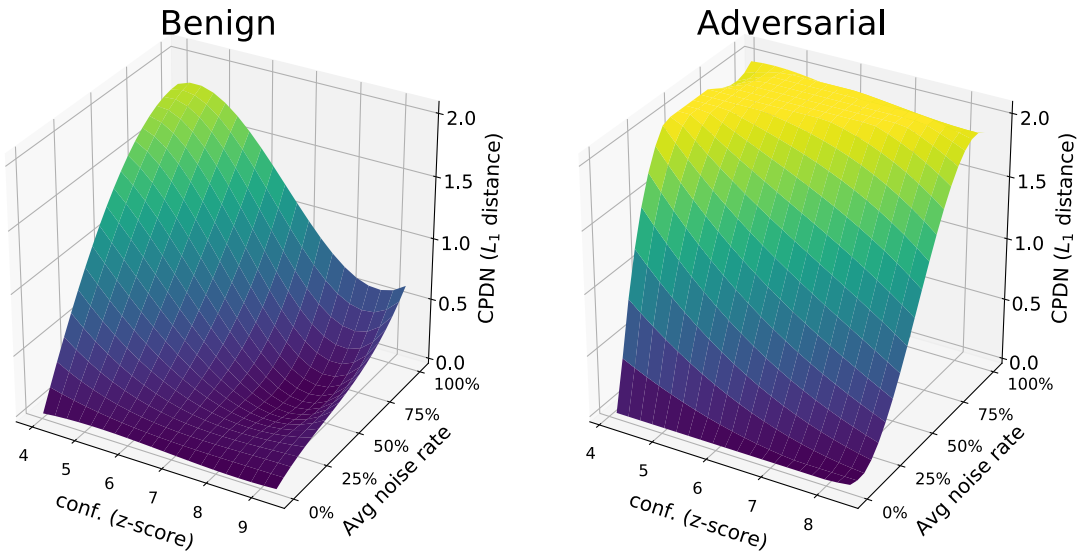

Figure 3.7: Classification Probability Deviation under Noise (CPDN) distribution under variable noise rate and classification confidence for (a) benign and (b) adversarial samples, for VGG16.

For this analysis we randomly select 1000 benign images from ImageNet. We further generate a total of 1000 adversarial images using 10 different attacks (or attack variants). We run each input 8 times with different noise distributions to

generate 8000 sample points for benign as well as adversarial inputs. We measure the CPDN for each input. Figure 3.7 shows a 3D representation of CPDN for varying classification confidence levels and average noise rates, for benign (a) and adversarial samples (b). Figure 3.7 shows that across the entire sample space of 8000 data points, CPDN is consistently higher for adversarials compared to benigns, confirming lower robustness to noise for adversarials.

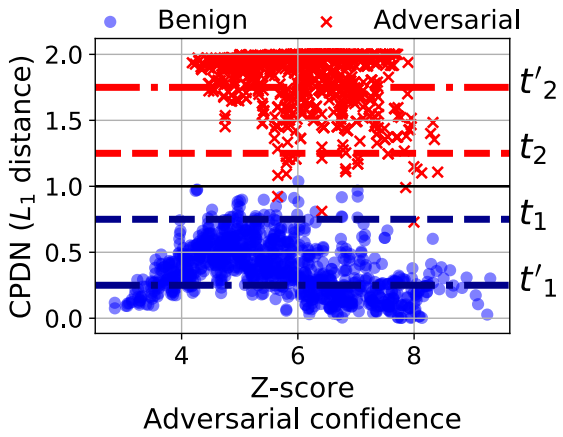

Figure 3.8: CPDN distributions for benign and adversarial images when noise is correlated to the confidence of the non-noisy classification for VGG16.

The data also shows that benign inputs, while more robust to noise than adversarials, are sensitive to high levels of noise if their classification confidence is low. This can be observed in Figure 3.7 (a) from the high CPDN at high noise and low Z-score. This suggests that low-confidence benign inputs should receive lower noise. Figure 3.7 (b) shows that low-confidence adversarials exhibit relatively high CPDN even at low noise levels. If we turn our attention to high confidence adversarial samples, we see that they require higher noise levels to exhibit high CPDN. The same high levels of

noise, however, when applied to high confidence benigns exhibit a much lower CPDN, indicating that they are robust to high noise. This data suggests that correlating the noise to the confidence of the classification is important to using CPDN in adversarial detection.

Figure 3.8 shows the CPDN distributions for benign and adversarial images when noise is correlated to the confidence of the non-noisy classification – higher noise for higher confidence. Figure 3.8 shows a clear separation between the CPDN distributions of adversarial and benign images, across the entire range of classification confidence that our attacks can generate. As expected, adversarial inputs exhibit higher CPDN deviation from non-noisy baseline compared to benign inputs. DNNSHIELD uses this separation with appropriate thresholds to detect adversarial inputs.

3.5.2 Robustness Analysis

In order to understand why it is important to correlate approximation noise to classification confidence, let us consider a model that classifies input X in the most probable class C_1 with probability P_1 and the runner-up class C_2 with probability P_2. Cohen et al. [27] showed that the distance between P_1 and P_2 has a direct correlation to the amount of the noise around X that can be tolerated by the classifier.

The L_2 radius R around X can be calculated by:

$$R = \frac{\sigma}{2}(\Phi^{-1}(P_1) - \Phi^{-1}(P_2)) \tag{3.6}$$

where Φ^{-1} is the inverse of standard Gaussian CDF and σ is the standard deviation of the noise. The higher the R value for an input X, the more noise the classifier can tolerate and still classify X correctly. According to Equation 3.6 the radius R is large when the probability of top class C_1 is high and the probability of the next class

is low, which corresponds to high confidence classification. This shows robustness to noise is correlated to classification confidence. We therefore expect that benign images will not suffer from high false positive rate regardless of confidence, if the approximation noise is scaled with the confidence.

3.5.3 Adversarial Detection

The detection framework relies on the CPDN metric to detect adversarial inputs. This process is depicted in Figure 3.9. Initially, a first inference pass through the network is performed without noise injection to establish a reference. The output classification is recorded as P^b. This is followed by another approximate inference pass. The confidence of the classification P^b is used to determine the amount of noise to be injected.

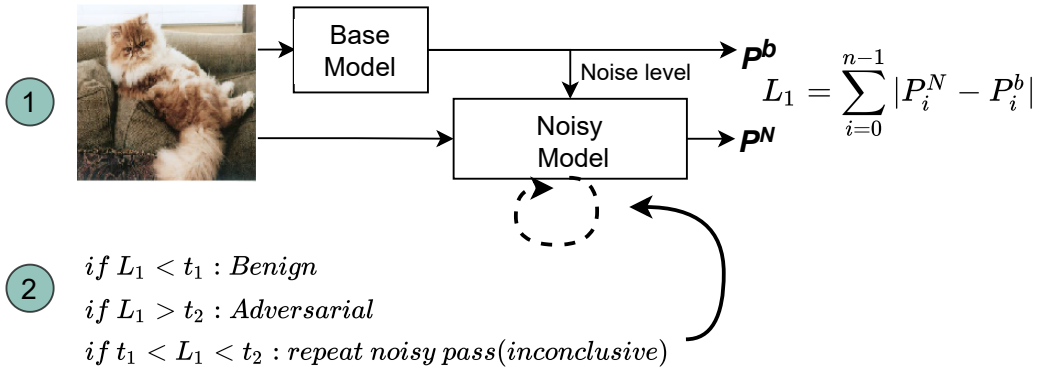

Figure 3.9: Adversarial detection using CPDN (L_1 distance).

The L_1 distance between the output vectors of the noisy (P^N) and non-noisy (P^b) inference passes is computed. The L_1 distance is then compared with different thresholds values. Depending on the outcome, subsequent inference passes may be required.

Figure 3.8 shows the 4 thresholds used by the detection mechanism overlaid on the L_1 distance distribution for *VGG16*. t_1 and t_2 represent the L_1 distance below/above which most benign/adversarial images fall, respectively. t'_1 and t'_2 represent tighter thresholds below/above which about 80% of the benign/adversarial images fall.

If the measured L_1 distance is either very high or very low ($< t'_1$ or $> t'_2$) the input image can immediately, and with high confidence, be classified as benign or adversarial, respectively. Most inputs ($>\approx 80\%$) from both our benign and adversarial test sets fall in this category. In this case the detection algorithm terminates and the outcome is reported.

Otherwise, detection mechanism cannot yet make a high-confidence detection, and another noisy inference pass is required. The average L_1 distance over all the previous noisy runs is computed and compared with the more conservative thresholds t_1 and t_2. The images with average $L_1 < t_1$ are classified as benign and those with average $L_1 > t_2$ are classified as adversarial. If $t_1 < L_1 < t_2$, a new noisy pass is performed and the average L_1 distance is recomputed. The algorithm repeats until a maximum M number of iterations is reached ($M = 4$ and $M = 8$ in our experiments). If a classification is still not possible, the algorithm defaults to Benign.

3.6 HASI: Hardware-Accelerated Stochastic Inference

The HASI architecture relies on a process of stochastic inference for adversarial detection. The stochastic inference relies on noisy activation function at the end of each layer in the network.

3.6.1 Noisy Activation

In our design and evaluation we use two popular image classification networks: VGG16 and ResNet50. Figure 3.10 shows the network architectures highlighting the layers in which we inject noise. VGG16 includes multiple convolution and fully connected (FC) layers. ResNet50 includes several Conv and Identity blocks each consisting of multiple convolution and activation layers, as illustrated in Figure 3.10. In VGG16 the Relu activation is fused to the convolution output while in ResNet50 there is batch normalization layer between convolution and activation.

HASI injects noise in multiple layers of both models. In both networks noise is injected in the output of the activation by multiplying a coefficient to the activation value. The magnitude of the noise is proportional to the activation value. The noise coefficient is randomly generated with a certain range. The range is set as a function of the classification confidence obtained from the non-noisy run. At runtime, noise injection can be enabled in some or all layers through software control.

Adaptive Noise Regulation

In Section 3.5 we showed that adapting the noise level to the confidence of the classification results in better separation of adversarial from benign distributions. Algorithm 1 shows the method used to determine the appropriate noise level for each

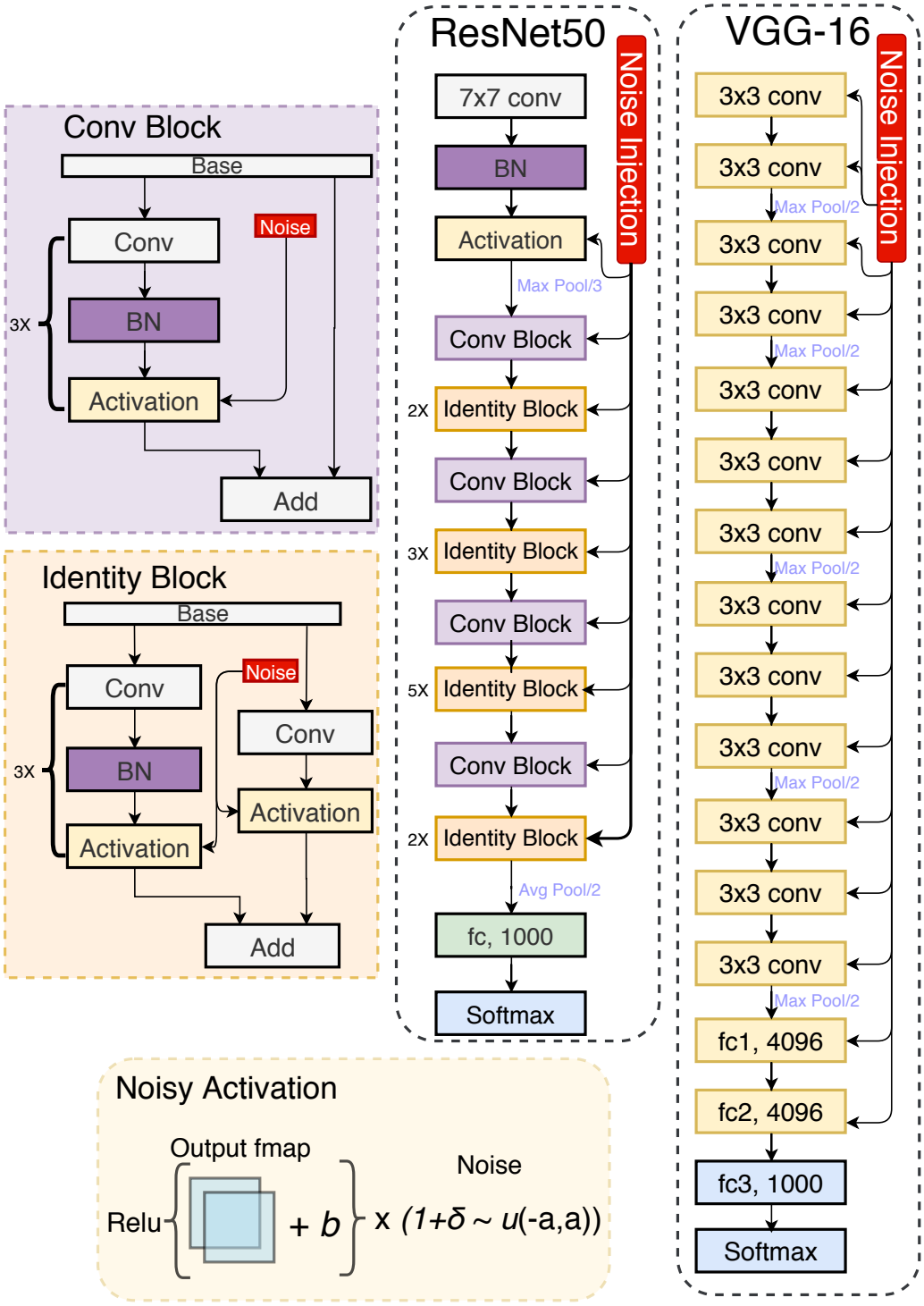

Figure 3.10: Noisy activation in the VGG-16 and ResNet50 models.

input. It computes a maximum range value (α) for the random noise generation function (δ). The maximum range value increases exponentially as a function of the classification confidence. The intuition behind this relationship is that high-confidence adversarials require high levels of noise to be detectable. At the same time, high confidence benign inputs are more robust to noise and are classified correctly in spite of the higher noise levels. This insight allows HASI to reliably detect adversarial inputs with a broad range of classification confidences.

Algorithm 1 Noise Regulation

function NOISE_RANGE(logit,α,β) ▷ Where logit - array, α,β - tuning parameters
  $\mu = mean(logit), \sigma = std(logit)$
  $sort\_logit = sort(logit)$
  $max_{1st} = sort\_logit[-1], max_{2nd} = sort\_logit[-2]$
  $z\_score_{1st} = \frac{max_{1st}-\mu}{\sigma}$, $z\_score_{2nd} = \frac{max_{2nd}-\mu}{\sigma}$
  $diff = z\_score_{1st} - z\_score_{2nd}$
  $step = z\_score_{1st} - z\_score_{2st}/diff$
  $range = \frac{\alpha \times e^{\beta \times step}}{\sum_{i=0}^{n} e^{logit[i]}}$
  return *range*
end function

3.6.2 Performance Optimizations

Running multiple noisy inference passes, although rare, can have a substantial performance impact. We make the observation that full re-execution of the model is not needed to obtain high adversarial detection rates. To reduce performance overhead, we can reuse some of the computation performed in previous passes. We refer to this approach as re-execution branching, illustrated in Figure 3.11. First, noise injection can skip the first few layers of the model. This allows the output of these first layers to be saved and reused for the execution of multiple noisy runs. Figure

3.11 (a) illustrates this approach. The initial m non-noisy layers are only executed once. The output of the last non-noisy convolution is saved and used by multiple noisy execution paths for the last n layers of the network. This can significantly reduce the performance overhead since initial layers are often large. However, this can also reduce the entropy of the noisy outputs.

A second optimization, which we call *tree branching* (TB), depicted in Figure 3.11 (b) allows the re-execution paths to diverge at different branching points in the network, forming a tree structure. This approach enables more computation reuse and lower overhead, while achieving slightly lower level of noise entropy, as some noisy activation outputs are re-used. For instance, Path 2 in Figure 3.11(b) starts from branching point ⓑ (re-executing only n' layers), instead of the original branching point ①. The branching factor b determines the frequency of branching. The higher the branching factor, the larger the computation reuse and lower the overhead.

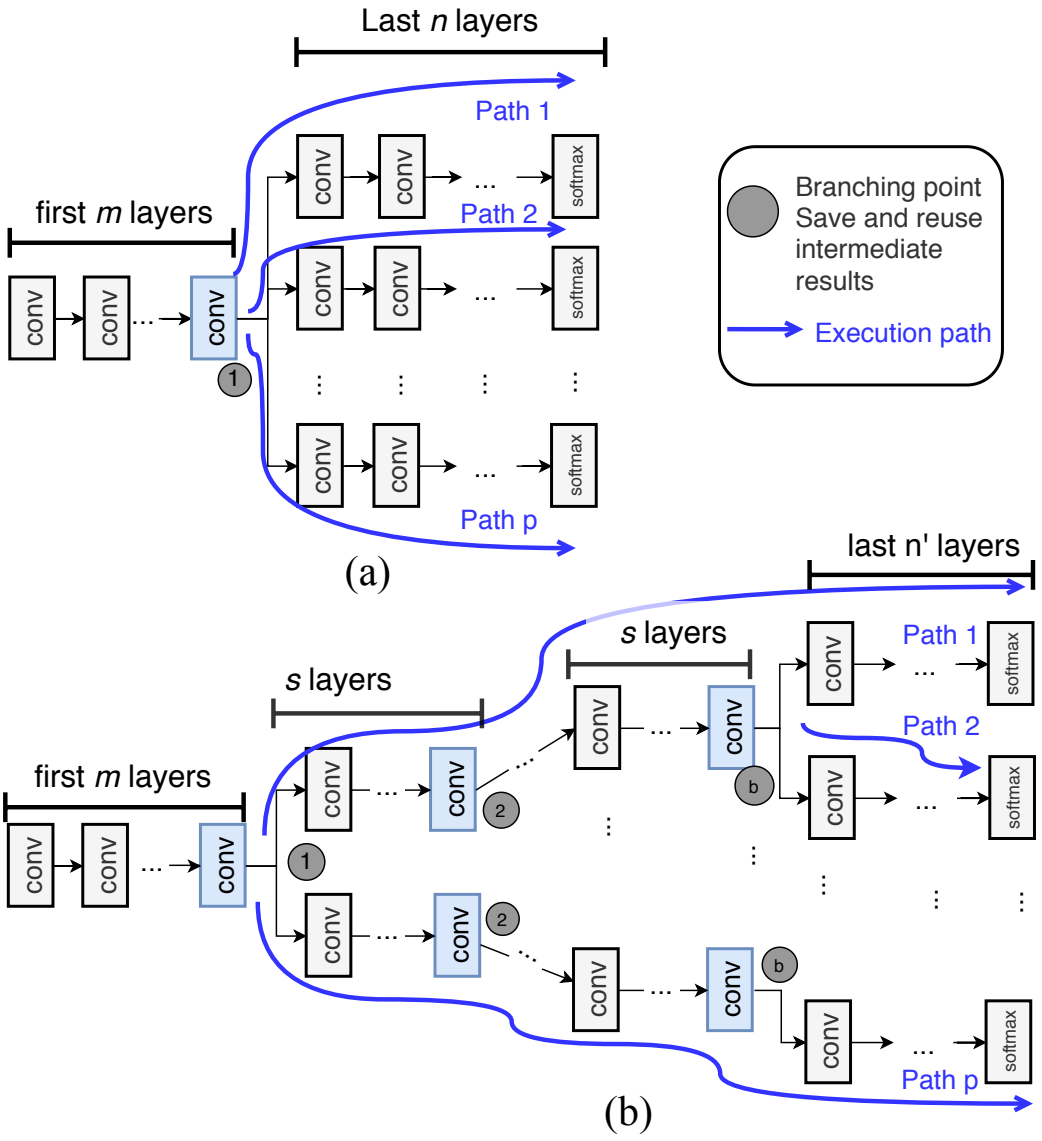

Figure 3.11: Different approaches for generating multiple stochastic inference outputs, with low overhead. a) Full Branching and b) Tree Branching b=s.

3.7 DNNShield: Dynamic Randomized Model Sparsification Defense Design

The research question we tackle is how to introduce a well controlled, variable and randomly distributed approximation error into the inference process in a performance-efficient way. Unfortunately approximate computation using approximate functional units does not offer sufficient flexibility to tune the error rate since they are generally not easily tunable. The same is true for quantization methods, which may not provide sufficient granularity for the approximation errors. In addition, both quantization and approximate computation tends to be deterministic, producing predictable results that may be exploited by an attacker. Although HASI porovides some optimizations as discussed earlier but it has higher overhead than approximation methods.

In order to address the aforementioned challenges we introduce the DNNSHIELD framework, which includes a new flexible and efficient mechanism to add controlled approximation into the model inference, a method to correlate the amount of error introduced into the model to the confidence of the non-noisy classification and a mechanism for using the approximate inference to detect adversarial inputs.

3.7.1 Dynamic Random Sparsification

We set a few important criteria for our approximate inference design: (1) flexibility to tune the error rate dynamically at runtime – to allow error rate to be correlated to input confidence, (2) randomness of the error distribution – to make defense-aware attacks less likely to succeed, and (3) low overhead.

In order to satisfy these criteria DNNSHIELD introduces noise into the DNN by randomly "dropping" (essentially ignoring) weights from the model, through a

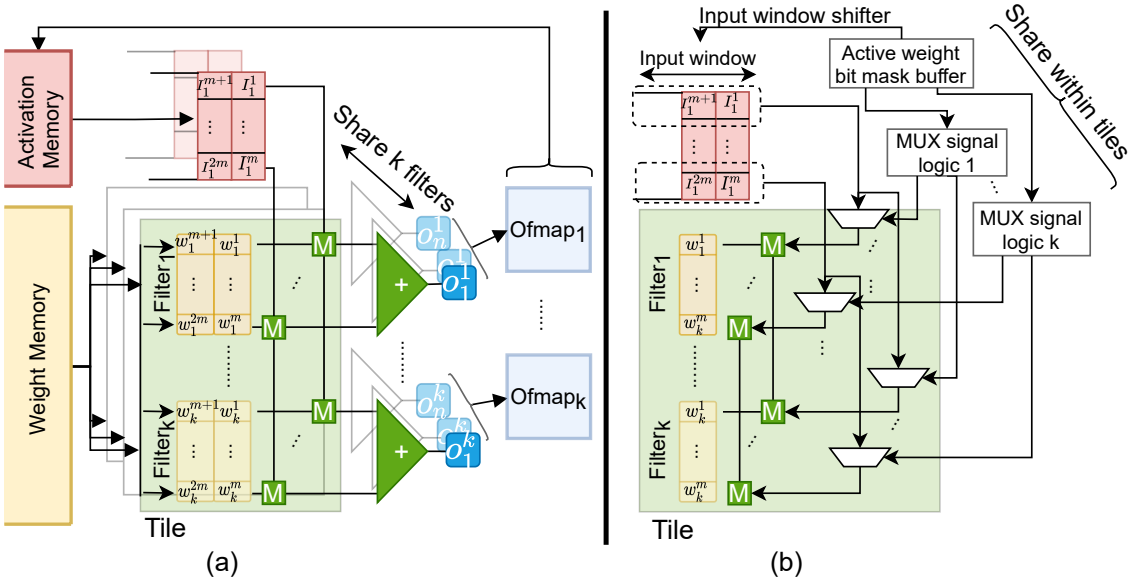

Figure 3.12: (a) Baseline accelerator, (b) DNNShield accelerator tile.

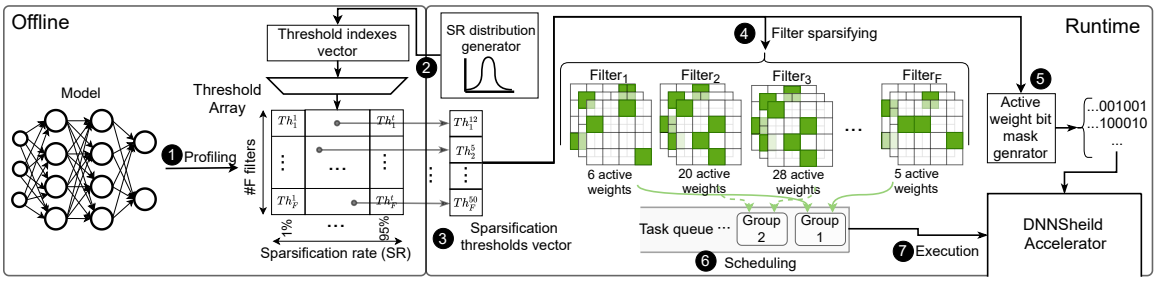

Figure 3.13: DNNShield software consisting of offline model profiling and runtime filter sparsification and scheduling.

process we call *dynamic random sparsification*. The fraction of dropped weights, or *sparsification rate* (SR) controls the amount of error in the model. The sparsification rate for each input is determined based on the classification confidence of the non-noisy run of that input.

The main advantage of dynamic random sparsification is the potential reduction in performance overhead. Prior work on sparse convolution accelerators [30, 150, 67, 106, 59, 32, 62, 109, 90] targeted statically weight sparsed models with the ultimate goal of improving performance/energy efficiency. However, exploiting sparse filters is more challenging in the case of DNNShield because the filter sparsity changes randomly from run to run. To address these new challenges we developed a hardware/software co-design that profiles the model first and performs scheduling for efficient resource utilization. The hardware accelerator supports dynamic-random sparse execution of the inference with minimum stalls with the help of the software scheduler and flexible control flow.

3.7.2 DNNShield Accelerator

Figure 3.12 shows the DNNShield accelerator tile (b) compared to a baseline Dense CNN accelerator (a). The baseline design consists of N tiles which share k filters. Each tile shares the same set of inputs and consists of $k \times m$ MAC units which perform $k \times m$ 8-bit multiplications per cycle. After a total of $k \times M$ ($M =$ filter size) MAC operations, tree-adders accumulate m partial results per output and generate k outputs per tile. The baseline accelerator processes all available weights uniformly, assuming no sparsity. Since the DNNShield random sparsification is a dynamic process, using conventional sparse accelerators is not practical since they

require deterministic offline preprocessing of the statically sparsed model to utilize the resources efficiently at runtime. Our DNNShield accelerator consists of two components: 1) software scheduler and 2) hardware accelerator.

DNNShield Software

The DNNShield software handles two main tasks: 1) one-time offline profiling and 2) runtime scheduling (Figure 3.13). In order to reduce the overhead of online sparsification, the DNNShield software parses the model offline and generates a table of threshold values for each filter, corresponding to different sparsification rates ❶.

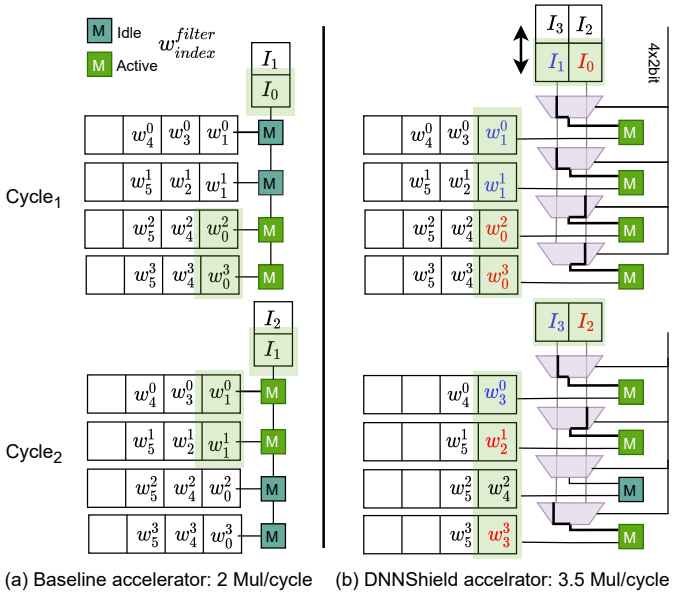

Figure 3.14: DNNShield sparse weights dataflow example through dense baseline (a), DNNShield accelerator (b).

At runtime, the *SR distribution generator* generates random sparsification rates for each filter ②, extracts the corresponding threshold value for each filter from the threshold array, and creates a threshold vector ③. The threshold values are used by the filter sparsifier to drop/ignore weight values below the thresholds assigned to each filter ④. At the same time, a bit mask of active weights is generated, and will be used by the hardware to map the correct inputs to the active weights ⑤.

Finally, the scheduler groups the filters with roughly similar number of active weights and places them in the task queue ⑥. This will increase the efficiency of the inference run since the filters scheduled in the same group will require similar numbers of multiply-accumulate operations, reducing load imbalance and the number of idle cycles. Finally each group is sent to the accelerator together with the active weight bit mask ⑦.

DNNShield Hardware

The DNNShield accelerator modifies the baseline dense accelerator to leverage the dynamically sparsified model. The DNNShield scheduler attempts to schedule the k filters with approximately the same number of active weights. However, DNNShield needs to make sure that different weights in each kernel can get access to their corresponding input with minimum stalling. For this purpose we used a look-ahead mechanism similar to that in [90] to match inputs and weights. Figure 3.12-b shows the DNNShield accelerator tile. The MUXes are added per each MAC unit to deliver the correct inputs to the active weights in each filter. The accelerator uses the active bit mask to generate the MUX select signals and also identify the offset by which the input window will be shifted every cycle.

Figure 3.14 shows an example of how sparse weights and their corresponding inputs flow through for baseline accelerator (Figure 3.14 (a)) and through DNNShield (Figure 3.14 (b)). Figure 3.14 shows 4 filters sharing the input line within the tile. At cycle 1 I_0 is ready to be used by the MAC units sharing the input, however, only two filters have corresponding weights active (w_0^2 and w_0^3). While the baseline accelerator leaves two MAC units underutilized, DNNShield fully utilizes the MAC units by performing 4 multiplication at cycle 1 due to the flexibility of selecting appropriate input through MUXes on top of each MAC unit. Therefore w_1^0 and w_1^1 are consumed by the MAC unit in the same cycle. Since no filter needs I_0 or I_1 the next two inputs I_2 and I_3 will be loaded into the input buffer in cycle 2. The non-deterministic scheduling of filter groups at runtime prevents pre-generating the signals that drive the selection multiplexers shown in Figure 3.14 (b). DNNShield instead uses a "MUX signal logic" unit that uses the bit mask of active weights produced by DNNShield scheduler to dynamically generate the control signals.

3.8 Prototype Implementation

As a proof-of-concept, we implement both HASI and DNNShield in a FPGA-based DNN accelerator, the Xilinx CHaiDNN architecture [144]. Figure 3.15 shows a diagram of the design. The baseline includes dedicated hardware for Convolution, Pooling, and Deconvolution functions. All the compute elements are connected to a Memory Interface Block which allows access to the on-chip SRAM as well as the main system DRAM via a custom AXI Interconnect.

HASI augments the baseline accelerator with the programmable random noise generation unit highlighted in yellow. DNNShield augments the baseline accelerator with the modified convolution supporting noisy sparsification including MUX select generator logic, input window buffers and priority encoders, color coded blue. The common component for HASI an DNNShield is color coded green which is custom logic for computing CPDN, additional control logic for coordinating partial result reuse and early termination.

Random Noise Generation One of the more compute intensive functions that HASI needs is the random number generation for noise injection. We use custom hardware to implement a pseudo-random number generator that has low performance overhead. The design, detailed inside the Figure 3.15 diagram, uses a Linear Feedback Shift Register (LFSR). The traditional LFSR design only produces 1 random bit per cycle. We unroll the single-bit design to produce n-bits every cycle, where n is configurable in software. This allows the HASI API to control the level of random noise to be injected in each layer of the model. The higher the values of n, the larger the range of noise values that will be added to activation output.

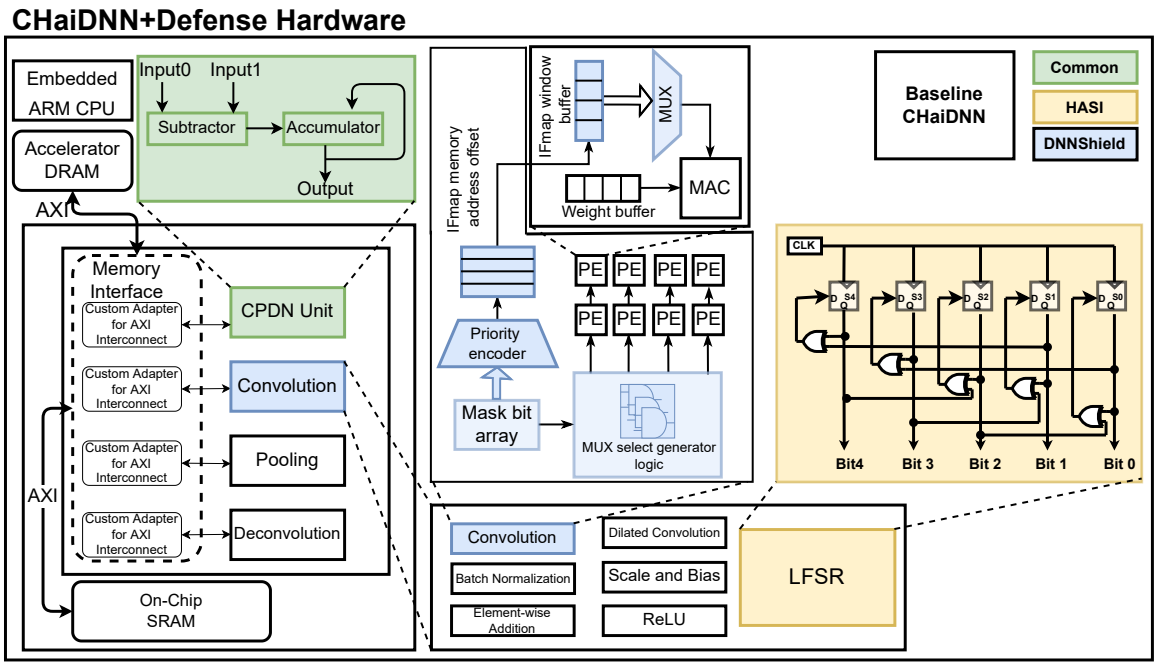

Figure 3.15: HASI and DNNShield hardware design based on the Xilinx CHaiDNN accelerator.

Convolution is the core of noisy sparsification with MUXes distributed through PEs for and MUX signal generators. PEs within the column share MUX signals. Also, priority encoders are used to determine the number of inputs that need to be bypassed regarding the ignored weights in sparsification. Then the address offset is calculated and the next set of inputs will be loaded in the input buffer.

CPDN Unit is used to compute the CPDN between a noisy and non-noisy output. It primarily consists of logic to subtract vector elements and accumulate the absolute value of the result, shown in Figure 3.15.

Detection API The CHaiDNN/HASI/DNNShield software stack as shown in Figure 3.16 that color-coded modifications based on different design. A Defense-specific

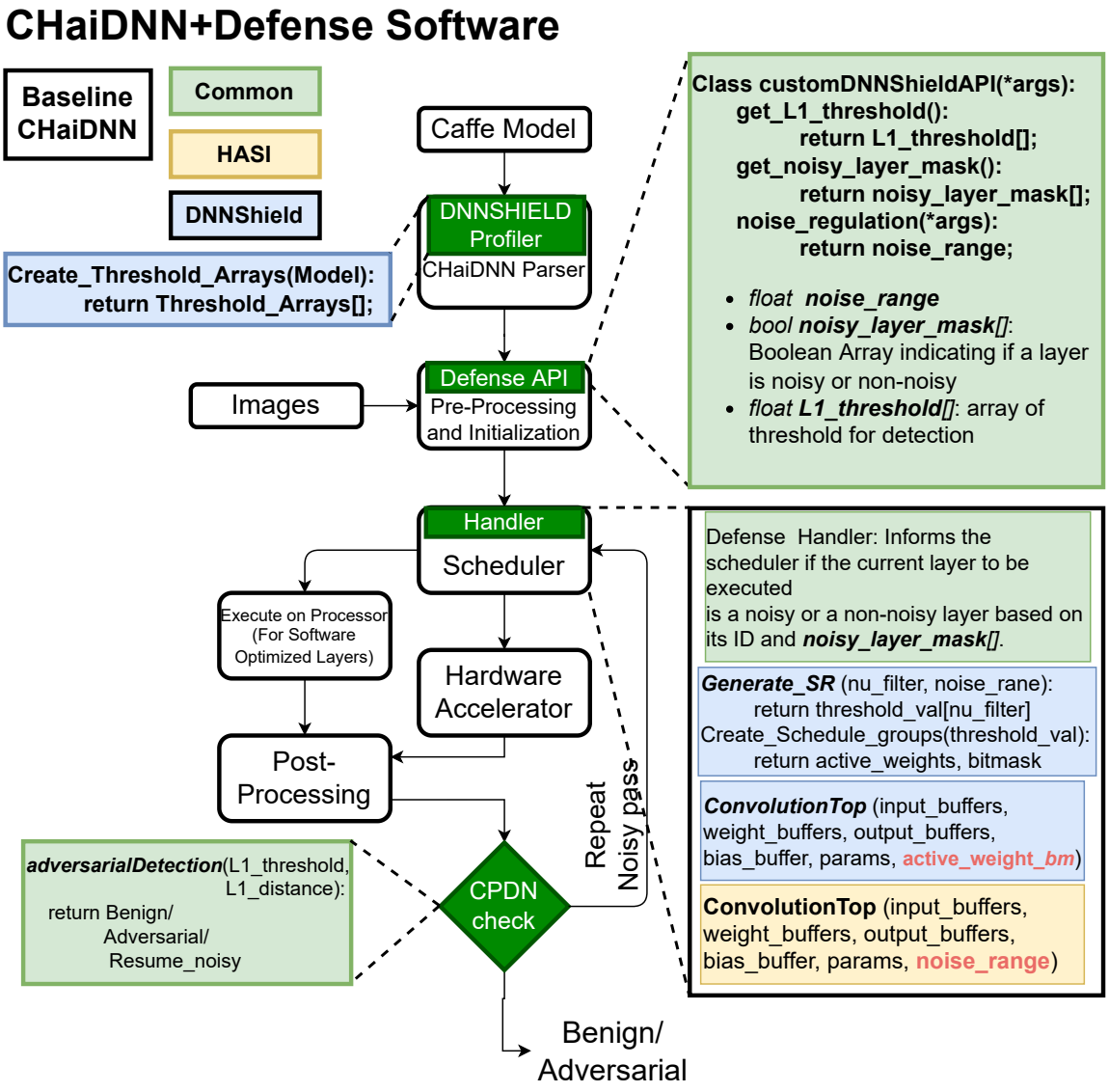

Figure 3.16: Defense API integrated with CHaiDNN software.

configuration file will set the noise mask, indicating which layers require noise injection. The Defense API is invoked by the pre-processor following the initial non-noisy run to determine the noise level to be injected for HASI design. This will be translated to sparsification rate for DNNSHIELD. A high-level scheduler schedules the execution of the model functions on the embedded processor and the hardware accelerators. The handler is designed to invoke the scheduler when the CPDN needs to be computed in the hardware. The CPDN Unit receives two input buffers containing the probabilities of the initial non-noisy run and the current noisy run and a *L1_Distance* buffer, where we maintain the running average of the L_1 distance, updated after every noisy run. After all the layers are processed, the value stored in *L1_Distance* is compared to the L1 threshold. If a benign/adversarial classification can be made, the response along with the classification output is returned. Otherwise, a new noisy run is scheduled until a decision is reached.

DNNSHIELD profiler is added to the ChaiDNN parser to create the threshold array that later will be used by DNNSHIELD. For DNNSHIELD, the handler also augments the scheduler with DNNSHIELD scheduler for grouping the balanced filter together for more efficient execution.

3.9 Methodology

3.9.1 Evaluation platform

We implement, synthesize and deploy both HASI and DNNShield on CHaiDNN running on a Xilinx Zynq UltraScale+ FPGA. The SoC associated with the board is ZU7EV which integrates a quad-core Arm Cortex-A53 processor. CHaiDNN is an open-source Deep Neural Network accelerator designed to run on Xilinx Ultrascale MPSoCs. CHaiDNN is built using Xilinx's Software-Defined System on Chip (SDSoC), which generates a synthesizable hardware design from a high-level description language. A quantizer is used to reduce weight representation precision from 32-bit float to 6-bit or 8-bit fixed point integers. We compare our FPGA accelerator with two software implementation of DNNShield on a CPU and GPU. We implemented the software HASI and DNNShield using TensorFlow2 [4]. We run our software DNNShield on Intel Core-i7 Ivy Bridge CPU@3.40GHz and NVIDIA RTX-2060 Turing GPU.

3.9.2 DNN Models, Input Dataset, Attacks

We used two networks VGG16 [121] and ResNet50 [54] trained on ImageNet [70] for running attacks and generating adversarial images. We randomly selected 100 images from the ImageNet dataset and generated 100 adversarial images for each attack we evaluate (for a total of 4600). All attacks are orchestrated in order to make high confidence adversarial images with minimum modifications. Targeted attacks, which aim to misclassify an input into a target class, use two types of targets called *Next* and *LL*. *Next* corresponds to choosing the target class $t = L + 1 \ mod \ \#classes$ where L is the sorted index of top ground truth classes. For *LL* the target class t

is chosen as least likely class ($t = min(\hat{y})$) where $\hat{y}$ is the prediction vector of an input image. Table 3.1 summarizes the adversarial attacks alongside their detailed parameters, success rate, average confidence and average distortion with different metrics per model.

3.9.3 Comparison with Existing Defenses

We compared our detection rate of adversarial as well as True Positive rate (TPR) of benign images with two state-of-the-art post-training defense mechanisms, detailed below. **Stochastic Activation Pruning (SAP)** [34] introduces randomness into the evaluation of a neural network to defend against adversarial examples. SAP randomly drops some neurons of each layer to 0 with a probability proportional to their absolute value. Values which are retained are scaled up to retain accuracy.

Feature squeezing (FS) [146, 78] is a correction-detection mechanism that relies on the observation that the input feature space is typically unnecessarily large and provides ample opportunity for constructing adversarial examples. The strategy taken in this work is to limit the input space by removing features. Two feature reduction techniques are evaluated for "squeezing" images: reducing the color depth and smoothing to reduce the variation among pixels. Detection of adversarial images is achieved by comparing the output of the network using the original image with the output of the squeezed image(s). FS requires off-line profiling and training to find the best squeezer and corresponding thresholds for each pair of data-set and attack, making it less practical to deploy in real-world applications. FS also requires at least 3 squeezers, resulting in at least 4× performance overhead.

Table 3.1: Attack parameters for multiple variants of CW and EAD attacks. Original attack success rate, confidence, and distortion. Dataset from ImageNet, on VGG16 and ResNet50.

Attack	Target	Param k		Mean Confidence		Distortion							
						L_0		L_1		L_2		L_∞	
		VGG	RNet	VGG	RNet	VGG	RNet	VGG	RNet	VGG	RNet	VGG	RNet
CW_{L0}	Next	5	5	92.9%	94.5%	42.2%	42.0%	232	105	10.44	6.26	0.94	0.88
	LL			84.8%	87.9%	42.4%	42.1%	382	185	13.65	8.33	0.96	0.92
CW_{L2}	Next	10	30	100.0%	100.0%	100.0%	100.0%	412	411	1.69	1.63	0.07	0.06
	LL			100.0%	100.0%	100.0%	100.0%	555	531	2.24	2.07	0.08	0.06
	Next	70	140	100.0%	100.0%	99.9%	100.0%	2,015	2,609	7.43	9.22	0.24	0.2
	LL			100.0%	100.0%	99.9%	100.0%	2,176	3,267	8.07	11.56	0.25	0.24
$CW_{L\infty}$	Next	5	5	94.7%	95.3%	100.0%	100.0%	735	523	2.27	1.6	0.01	0.01
	LL			91.8%	93.4%	99.9%	100.0%	1,023	699	3.05	2.12	0.01	0.01
EAD_{L1}	Next	10	30	100.0%	100.0%	52.4%	58.3%	173	205	2.69	2.79	0.24	0.22
	LL			100.0%	100.0%	54.7%	59.4%	269	276	3.56	3.47	0.29	0.26
	Next	70	140	100.0%	100.0%	78.8%	80.5%	1,400	1,734	9.92	12.34	0.54	0.49
	LL			100.0%	100.0%	78.3%	85.5%	1,510	2,782	10.65	17.36	0.55	0.58
EAD_{EN}	Next	10	30	100.0%	100.0%	47.9%	52.4%	191	421	4.36	6.73	0.6	0.73
	LL			100.0%	100.0%	48.0%	57.4%	252	596	5.9	8.59	0.72	0.8
RNet: ResNet50, VGG: VGG16													

For a fair comparison we retrained FS on our set of benign and Adversarial images for both VGG16 and ResNet50 separately. We also used the best three FS squeezers for ImageNet reported in the paper.

3.10 Evaluation

We evaluate HASI and DNNShield adversarial detection rate, robustness to defense-aware attacks, and performance and area overheads. We also conduct a number of sensitivity studies for the main design parameters.

3.10.1 Adversarial Detection

We first look at DNNShield's ability to identify adversarial images. We measure the detection rate for adversarial inputs as well as the false positive rate (FPR) for benign inputs. We compare HASI and DNNShield with Feature Squeezing (FS) and SAP for multiple configurations of CW and EAD. Table 3.2 lists the detection rate for all the attack variants we evaluate, for both VGG16 and ResNet50.

The results show that both DNNShield significantly outperforms both FS and SAP on average. DNNShield shows an average detection rate of 86% and 88% for VGG16 and ResNet50, respectively These numbers are 86% and 93% for HASI. HASi and DNNShield also significantly outperform the state of the art defense, FS which averages 55% and 79% for VGG and ResNet, respectively. This is especially true for high-confidence attack variants, for which FS does not work as well. For instance, under the EAD_{L1} attack with $k = 70$ we see 93% detection rate for DNNShield vs. 4% for FS (VGG16). This shows that HASI and DNNShield are resilient to very strong attacks. Since DNNShield is more efficient design that HASI, we will only show results for DNNShield in the rest of this section.

Figure 3.17 shows detection rate versus adversarial confidence for DNNShield, FS and the *Approx. mul8u_KEM* as a function of classification confidence. Both FS and *Approx. mul8u_KEM* detection rates fall steeply as confidence increases while

Table 3.2: Detection rates for 4 defenses: SAP, FS, Approximate-MUL, HASI and DNNShield. Dataset from ImageNet, on VGG16 and ResNet50.

Attacks				**Defenses**									
Attack	Target	Param k		SAP*		FS+		Approximate mul8u_KEM		HASI		DNNShield	
		VGG	RNet	VGG	RNet	VGG	RNet	VGG	RNet	VGG	RNet	VGG	RNet
CW_{L0}	Next	5	5	35%	42%	**100%**	**100%**	34%	59%	90%	95%	67%	82%
	LL			29%	43%	100%	100%	56%	96%	**100%**	**100%**	**100%**	**100%**
CW_{L2}	Next	10	30	45%	45%	84%	89%	68%	11%	86%	99%	**91%**	**100%**
	LL			36%	46%	100%	100%	81%	28%	**100%**	**100%**	**100%**	**100%**
	Next	70	140	0%	21%	6%	48%	7%	0%	78%	97%	**84%**	**89%**
	LL			0%	17%	9%	67%	19%	0%	92%	**99%**	**96%**	98%
$CW_{L\infty}$	Next	5	5	42%	47%	**91%**	96%	69%	71%	89%	**97%**	83%	89%
	LL			45%	45%	100%	100%	94%	98%	99%	**100%**	**100%**	**100%**
EAD_{L1}	Next	10	30	27%	52%	78%	98%	41%	2%	84%	**99%**	**91%**	98%
	LL			34%	41%	100%	98%	59%	0%	**100%**	**100%**	**100%**	**100%**
	Next	70	140	0%	16%	4%	46%	2%	0%	62%	**94%**	**78%**	88%
	LL			0%	18%	4%	81%	4%	0%	87%	**90%**	**93%**	84%
EAD_{EN}	Next	10	30	17%	40%	63%	89%	27%	0%	**82%**	93%	80%	**94%**
	LL			12%	26%	98%	**98%**	36%	2%	**100%**	97%	99%	97%
RNet: ResNet50			AVG	16.6%	33.2%	55%	79%	36%	15%	**86%**	**93%**	**86%**	88%
VGG: VGG16			FPR	58%	37%	6%	3%	29%	19%	5%	5%	6%	6%

+FS threshold: VGG16: 1.022, ResNet50: 1.229, *Values for SAP are accuracy

DNNShield detection rate remains high. These results re-emphasize the importance of adapting the approximation error to the confidence of the classification.

3.10.2 Defense-Aware Attacks

In order to investigate the robustness of the HASI/DNNShield defense, we construct a set of attacks tailored specifically to defeat it. These attacks assume full knowledge of the HASI/DNNShield design. In theory, HASI/DNNShield could be defeated by an attack that generates adversarial examples for which the model's robustness to approximate inference is similar to that of benign examples. In order to attempt to generate such adversarial examples, we used the approach suggested in [129] to generate adversarial examples that target the probability vector of an arbitrary benign example from another class. The idea is to create an adversarial example that mimics the response of benign images under noise. Hence, for sample

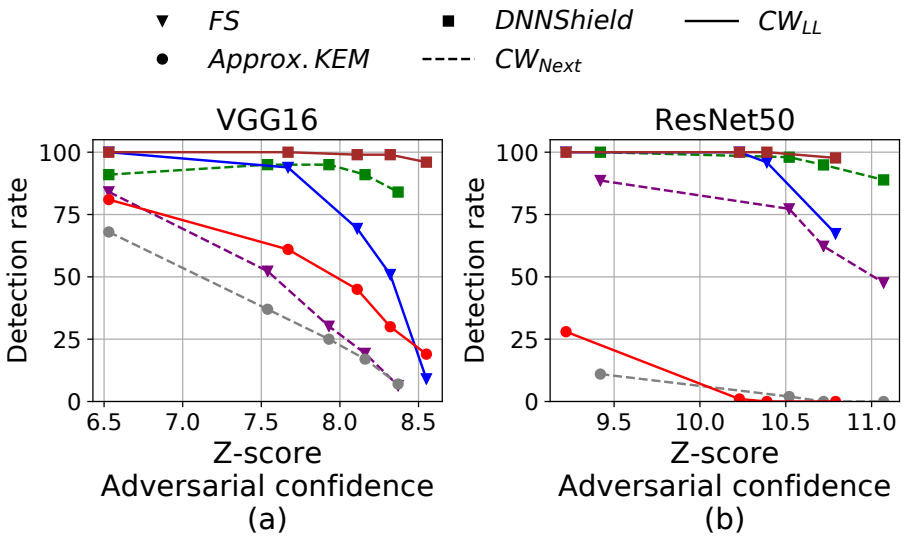

Figure 3.17: Detection rate for different adversarial confidence generated by CW_{L2} attack, (a) VGG16 and (b) ResNet50

.

x of class y, we pick a target $t \neq y$ and create adversarial example x' that minimizes the objective:

$$minimize \quad ||y(x') - y(x_t)||_1 \tag{3.7}$$

where $y(x')$ and $y(x_t)$ are the probability vector of the adversarial and target inputs respectively. While we try to minimize the L_1 distance between adversarial and the benign target, we need to also minimize the adversarial perturbation under the L_2 distortion metric. The final objective function is:

$$\begin{aligned} minimize_x \quad & cf(x,t) + \beta||y(x') - y(x_t)||_1 + ||x - x_0||_2^2 \\ & \text{such that} \quad x \in [0,1]^n \end{aligned} \tag{3.8}$$

where $f(x,t)$ denotes the loss function and β is the regularization parameter for L_1 penalty. Increasing β forces a lower L_1 distance between the adversarial and target benign and could evade HASI/DNNSHIELD detection.

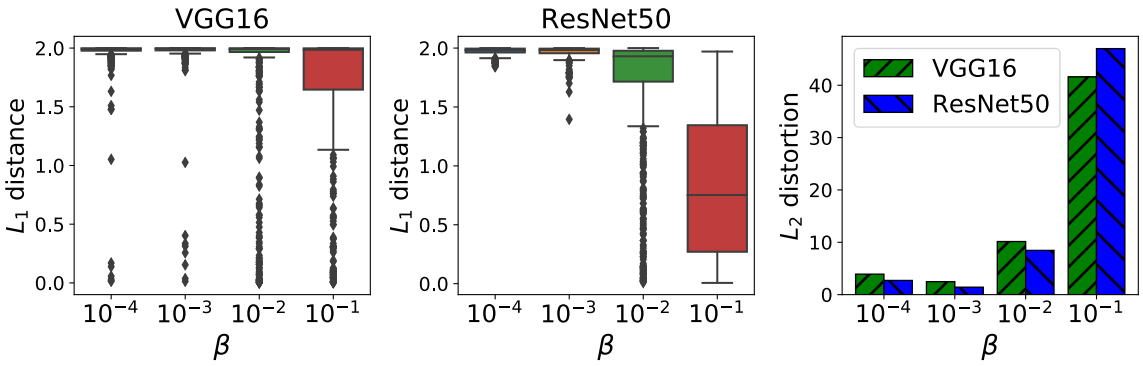

Figure 3.18: L_1 distance vs. L_2 distortion for different β values.

Table 3.3 summarizes the adaptive attack parameters and detection rates under HASI/DNNSHIELD. We can see that for low-β attacks, HASI/DNNSHIELD detection rate is very high (A_1-A_3). For A_4, with $\beta = 10^{-1}$ the detection rate is lower. However, in order to generate A_4, the L_2 distortion has to be increased by 4-5× relative to A_3. To understand why, Figure 3.18 shows the effect of β on the L_1 distance of probability distribution and L_2 distortion. Optimizing for both low L_2 distortion and L_1 distance are competing objectives. Increasing β will decrease the L_1 distance, making the adversarial harder to detect, but it also increases L_2 distortion. The target benign input, which the adversarial sample is trying to mimic, is chosen randomly from 1000 images in the adversarial targeted class. While a few of these targets do lead to lower distortion, the average distortion, for high β (10^{-1}), is very high. Another popular approach is using an EoT attack in which noise (transformation) was applied during adversarial generation. We injected variable noise correlated to the confidence of the classification in each training iteration, as in HASI/DNNSHIELD. The result was that, because of the variable noise, the attack could not converge on

a successful adversarial. Using fixed noise as in traditional EoT did not work either because of the adaptive HASI/DNNSHIELD response.

Table 3.3: HASI/DNNSHIELD aware adaptive attacks

Attack	β	Success rate		Mean Confidence		L_2 Distortion		HASI/DNNSHIELD det.	
		VGG	RNet	VGG	RNet	VGG	RNet	VGG	RNet
A_1	10^{-4}	100%	100%	94.3%	95.9%	3.91	2.71	99%	100%
A_2	10^{-3}	100%	100%	92.3%	94.0%	2.48	1.42	99%	100%
A_3	10^{-2}	100%	100%	96.1%	96.9%	10.14	8.45	95%	84%
A_4	10^{-1}	58%	31%	97.7%	97.9%	41.58	46.99	81%	39%

Figure 3.19 shows two examples of adversarial inputs generated with different β values. We can see that distortion artifacts are clearly visible for $\beta = 10^{-1}$, and can be detected through other means.

At high β values, the attack is also less likely to succeed. For $\beta = 10^{-1}$, only 58% (VGG) and 31% (ResNet) of examples can be converted into adversarials that defeat the unprotected baseline. HASI/DNNSHIELD is still able to detect 81% and 39% of the VGG16 and ResNet50 ones, respectively.

This shows that HASI/DNNSHIELD is robust to defense-aware attacks that optimize for low L_1 distance.

3.10.3 Performance, Area and Power Overheads

We next examine the performance, area and overheads of the HASI and DNNSHIELD framework. Figure 3.20 shows the average normalized run time of HASI and DNNSHIELD on the GPU, CPU and DNNSHIELD accelerator. The runtime overhead of software DNNSHIELD on GPU is 15× to 25× higher than the baseline. This high overhead is

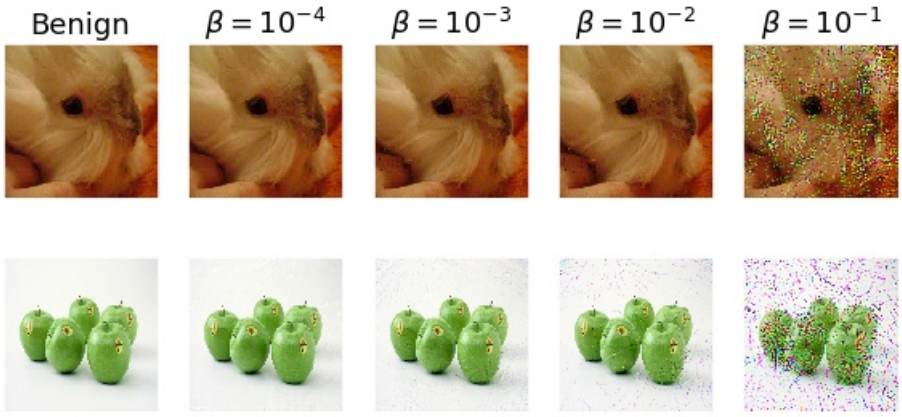

Figure 3.19: Adversarials generated by HASI/DNNSHIELD-aware attacks.

primarily due to the random number generation function used by the dynamic sparsification algorithm – which does not appear to be optimized on the GPU – and is called when sparsifying each filter. This overhead is highlighted by the "DNNSHIELD overhead", shown as a pattern in Figure 3.20.

In contrast, the overhead of the DNNSHIELD accelerator implementation is much lower at 1.53× and 2× for ResNet50 and VGG16, respectively. Unlike the GPU, the DNNSHIELD accelerator performance overhead is primarily due to re-execution of the approximate inference. While not trivial, the DNNSHIELD performance overhead compares favorably with that of FS which exceeds 4×. For software-DNNSHIELD on the CPU the overhead ranges from 2.43× to 4.47× which is again, higher that for DNNSHIELD. In addition, the total runtime of the models on the CPU is dramatically longer than the FPGA. Very slow runtime of convolutional and FC layers on CPU dominate execution time. Hardware support for dynamic sparsification reduces

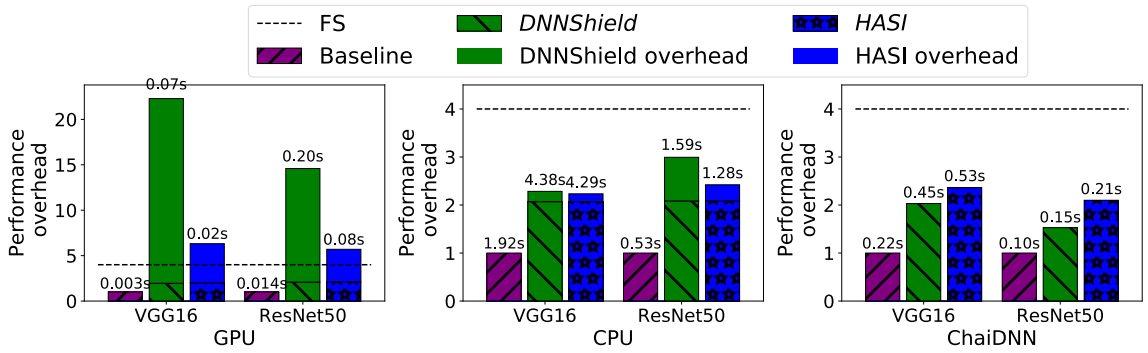

Figure 3.20: HASI/DNNShield runtime on (a) GPU, (b) CPU and (c) HASI/DNNShield accelerator for VGG16 and ResNet50.

overhead by 15% and 30% relative to the DNNShield without sparsification support. DNNShield reduced the overhead of HASI from 2.4× to 2× for VGG16 and from 2.1× to 1.5× for ResNet50.

Table 3.4 summarizes the area and power overhead of the combined DNNShield hardware relative to the baseline CHaiDNN accelerator. We can see that the total overhead is low, with FPGA resource utilization increasing by at most 2.56%. Power overhead is higher, but still small at 4.5% dynamic.

3.10.4 Sensitivity Studies

The DNNShield design spans a broad design space that affects performance overhead for adversarial detection accuracy.

Sparsification Approaches

We evaluate multiple approaches for dynamic sparsification. The naive approach of randomly dropping any weight subject to the sparsification rate (SR) results in, as Figure 3.21 shows, a very high (> 90%) false positive rate (FPR) for benign inputs,

Table 3.4: FPGA resources and power overhead of DNNSHIELD over baseline CHaiDNN accelerator.

Resource	Baseline	DNNSHIELD	Overhead%
BRAM	202.5	204	0.75%
DSP	696	696	0.0
FF	112501	113630	1.1%
LUT	158060	159381	0.8%
URAM	80	80	0.00
BUFG	3	3	0.00
PLL	1	1	0.00
Power	Baseline	DNNSHIELD	Overhead%
Static	0.721W	0.726	0.6
Dynamic	5.567W	5.822W	4.5

indicating that random weight sparsification results cannot be used to discriminate adversarial inputs. This is because random sparsification can result in the dropping of large weight values, with large impact on classification output. To address this issue, in DNNSHIELD we drop a random number of weights between 0 and SR from each filter, in ascending order of their values. This results in high adversarial detection, with low benign FPR. This is mostly due to the fact that dropping weights in ascending order enables more precise control over the approximation error. We also show that adapting the SR to classification confidence is very important. The High SR and Low SR experiments in Figure 3.21 show the effects of weight dropping at fixed rates of up to 80% and 20% respectively. The Low SR is insufficient to achieve adversarial detection, while fixed 80% results in very high benign FPR.

Detection Convergence

Figure 3.22 shows the attack success rate as a function of the number of runs with inference approximation. More runs should ensure higher detection accuracy

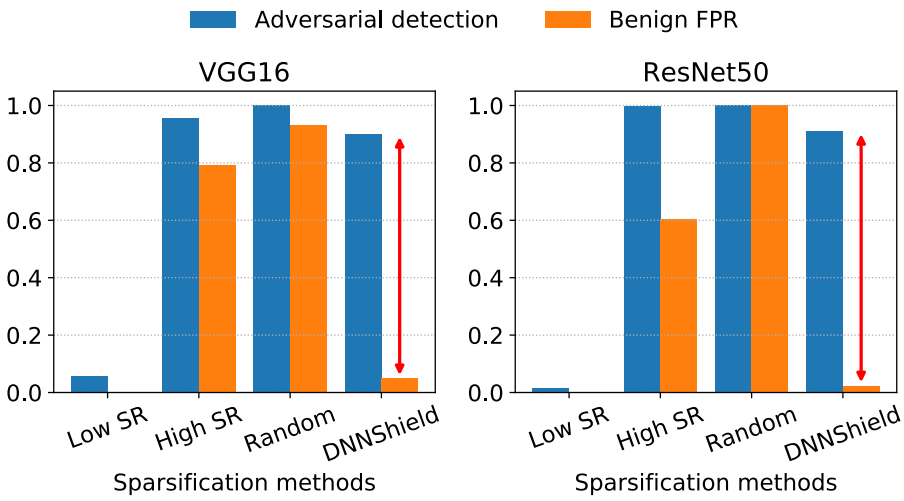

Figure 3.21: Adversarial detection and benign FPR with different sparsification approaches.

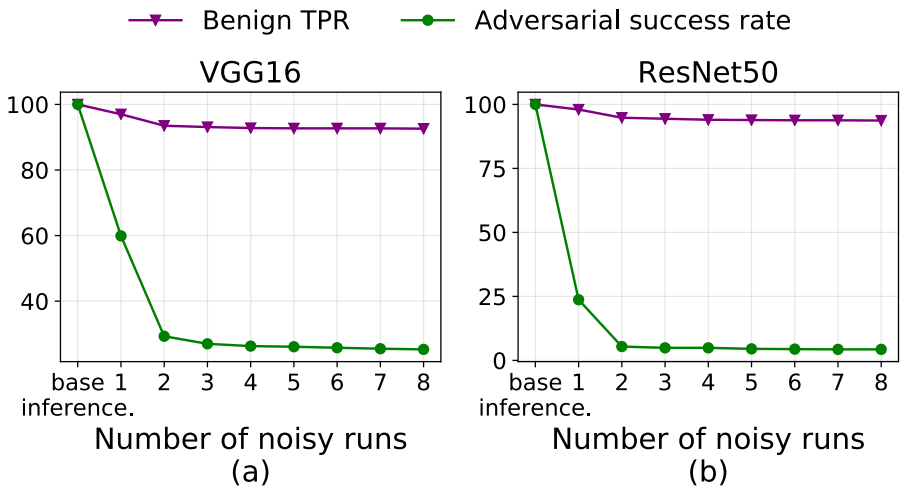

Figure 3.22: Adversarial attack success rate for multiple attacks as a function of the number of noisy runs in DNNShield.

by generating more samples for the L_1 distance average. We can see that the attack success rate drops rapidly after 1-2 noisy runs, and remains mostly constant after that. This translates in DNNSHIELD converging rapidly on a detection decision. Figure 3.23 shows the number of runs required to make a decision (on the X-axis) for benign and adversarial inputs. We can see that a single run is sufficient for ¿80% of the benign inputs, and less than 10% require more than 2. Only the benign re-execution rate is relevant to the overhead since adversarials are rare events.

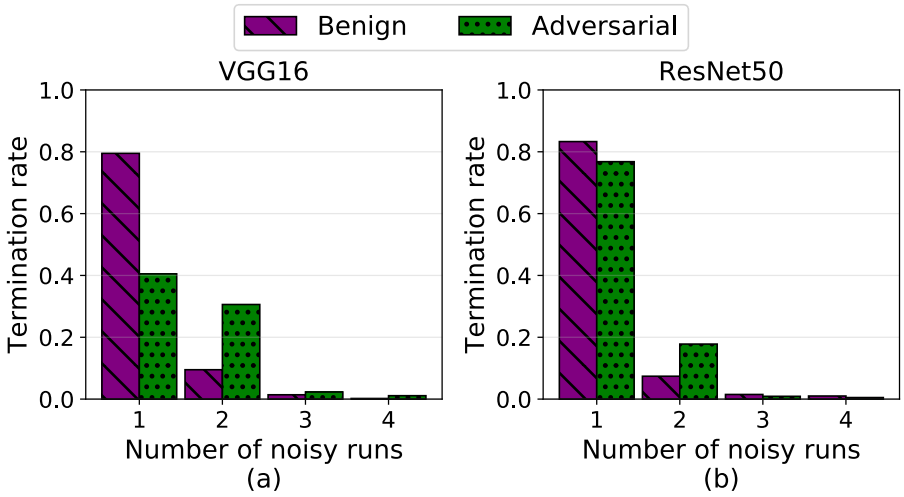

Figure 3.23: Number of noisy runs required by DNNSHIELD to make a classification decision for multiple adversarial data sets and benign inputs.

Detection Thresholds

Finally, we performed a sensitivity analysis on the threshold parameters used for adversarial detection. To study the effect of detection thresholds, we varied t'_1 in the $[0.05, t_1]$ range in 0.1 increments. Then, for each value of t'_1 we varied t'_2 in the $[t_2, 1.95]$ range and computed the average false positive rate (FPR). Figure 3.24 shows

the average FPR for benign and adversarial inputs for different values of t'_1. ResNet50 exhibits a tighter distribution for L_1 distance under approximation and is therefore not sensitive to the threshold values. VGG16 on the other hand is more sensitive due to its wider distribution The threshold value allows a small tradeoff between FPRs for benigns vs. adversarials.

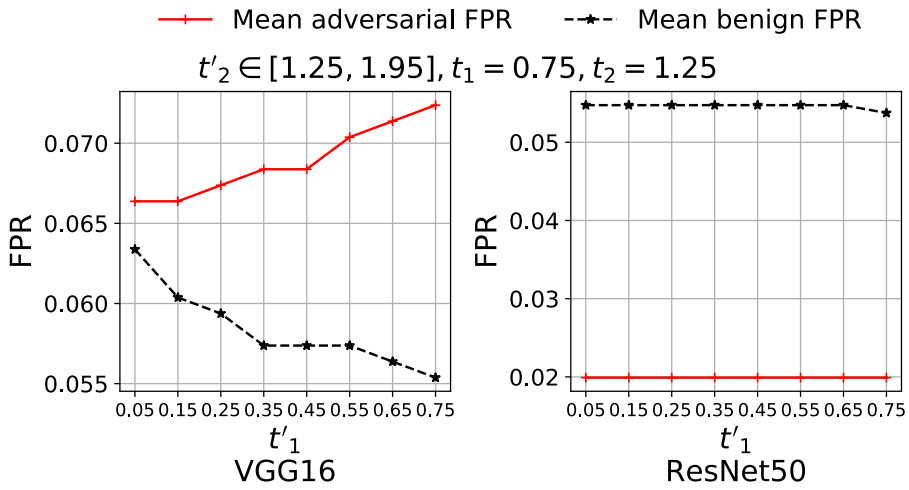

Figure 3.24: Benign and adversarial FPR for different threshold values, left VGG16 and right ResNet50.

3.11 Conclusion

In conclusion, this chapter showed that adaptive noise injection in DNN models enables robust $> 90\%$ adversarial detection across multiple strong attacks, for different image classifiers. We also showed that a hardware/software co-design of DNNSHIELD reduces performance overhead relative to a software-only GPU implementation to $1.5\times - 2.\times$ relative to an unprotected baseline running on an FPGA, with $< 3\%$ hardware overhead.

Bibliography

[1] Deepbench. https://svail.github.io/DeepBench/.

[2] Nvidia cuda deep neural network library. https://developer.nvidia.com/cudnn.

[3] Optimizing recurrent neural networks in cudnn 5. https://devblogs.nvidia.com/parallelforall/optimizing-recurrent-neural-networks-cudnn-5/.

[4] Tensorflow. https://www.tensorflow.org/.

[5] Naveed Akhtar and Ajmal Mian. Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access*, 6:14410–14430, 2018.

[6] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: ineffectual-neuron-free deep neural network computing. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 1–13. IEEE, 2016.

[7] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Awni Y. Hannun, Billy Jun, Tony Han, Patrick LeGresley, Xiangang Li, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Sheng Qian, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Chong Wang, Yi Wang, Zhiqian Wang, Bo Xiao, Yan Xie, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. Deep speech 2 : End-to-end speech recognition in english and mandarin. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 173–182, 2016.

[8] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R Stanley Williams, Paolo Faraboschi, Wen-mei W Hwu, John Paul Strachan, Kaushik Roy, et al. Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 715–731, 2019.

[9] Aayush Ankit, Abhronil Sengupta, Priyadarshini Panda, and Kaushik Roy. Resparc: A reconfigurable and energy-efficient architecture with memristive crossbars for deep spiking neural networks. *arXiv preprint arXiv:1702.06064*, 2017.

[10] A. J. Annunziata, M. C. Gaidis, L. Thomas, C. W. Chien, C. C. Hung, P. Chevalier, E. J. O'Sullivan, J. P. Hummel, E. A. Joseph, Y. Zhu, T. Topuria, E. Delenia, P. M. Rice, S. S. P. Parkin, and W. J. Gallagher. Racetrack memory cell array with integrated magnetic tunnel junction readout. In *2011 International Electron Devices Meeting*, pages 24.3.1–24.3.4, Dec 2011.

[11] Elham Azari, Aykut Dengi, and Sarma Vrudhula. An energy-efficient fpga implementation of an lstm network using approximate computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, pages 305–306, New York, NY, USA, 2019. ACM.

[12] Jimmy Ba, Volodymyr Mnih, and Koray Kavukcuoglu. Multiple object recognition with visual attention. *CoRR*, abs/1412.7755, 2014.

[13] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations (ICLR)*, 2015.

[14] Mahdi Nazm Bojnordi and Engin Ipek. Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning. In *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*, pages 1–13, 2016.

[15] Xiaoyu Cao and Neil Zhenqiang Gong. Mitigating evasion attacks to deep neural networks via region-based classification. In *Proceedings of the 33rd Annual Computer Security Applications Conference on*, pages 278–287, 2017.

[16] Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian Goodfellow, Aleksander Madry, and Alexey Kurakin. On evaluating adversarial robustness. *arXiv preprint arXiv:1902.06705*, 2019.

[17] Nicholas Carlini and David Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM workshop on artificial intelligence and security*, pages 3–14, 2017.

[18] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy (SP)*, 2017.

[19] K. Chang and T. Chang. Vscnn: Convolution neural network accelerator with vector sparsity. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, May 2019.

[20] Pin-Yu Chen, Yash Sharma, Huan Zhang, Jinfeng Yi, and Cho-Jui Hsieh. Ead: elastic-net attacks to deep neural networks via adversarial examples. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

[21] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 269–284, 2014.

[22] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, pages 367–379, 2016.

[23] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, pages 609–622, 2014.

[24] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. PRIME: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, pages 27–39, 2016.

[25] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[26] Jinil Chung, Jongsun Park, and Swaroop Ghosh. Domain wall memory based convolutional neural networks for bit-width extendability and energy-efficiency. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design, ISLPED 2016, San Francisco Airport, CA, USA, August 08 - 10, 2016*, pages 332–337, 2016.

[27] Jeremy M Cohen, Elan Rosenfeld, and J Zico Kolter. Certified adversarial robustness via randomized smoothing. *arXiv preprint arXiv:1902.02918*, 2019.

[28] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning (ICLR)*, pages 160–167, 2008.

[29] Arup De, Maya Gokhale, Rajesh Gupta, and Steven Swanson. Minerva: Accelerating data analysis in next-generation ssds. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 9–16. IEEE, 2013.

[30] Alberto Delmas Lascorz, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, Kevin Siu, and Andreas Moshovos. Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 749–763, 2019.

[31] Chunhua Deng, Yang Sui, Siyu Liao, Xuehai Qian, and Bo Yuan. Gospa: An energy-efficient high-performance globally optimized sparse convolutional neural network accelerator. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1110–1123. IEEE, 2021.

[32] Chunhua Deng, Yang Sui, Siyu Liao, Xuehai Qian, and Bo Yuan. Gospa: An energy-efficient high-performance globally optimized sparse convolutional neural network accelerator. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1110–1123, 2021.

[33] Kedar Dhamdhere, Mukund Sundararajan, and Qiqi Yan. How important is a neuron. In *7th International Conference on Learning Representations*, 2019.

[34] Guneet S. Dhillon, Kamyar Azizzadenesheli, Jeremy D. Bernstein, Jean Kossaifi, Aran Khanna, Zachary C. Lipton, and Animashree Anandkumar. Stochastic activation pruning for robust adversarial defense. In *International Conference on Learning Representations (ICLR)*, 2018.

[35] Gavin Weiguang Ding, Kry Yik Chau Lui, Xiaomeng Jin, Luyu Wang, and Ruitong Huang. On the sensitivity of adversarial robustness to input data distributions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPR)*, pages 13–16, 2019.

[36] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, pages 92–104, 2015.

[37] Reuben Feinman, Ryan R Curtin, Saurabh Shintre, and Andrew B Gardner. Detecting adversarial samples from artifacts. *arXiv preprint arXiv:1703.00410*, 2017.

[38] J. C. Ferreira and J. Fonseca. An fpga implementation of a long short-term memory neural network. In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, Nov 2016.

[39] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, June 2018.

[40] Yonggan Fu, Qixuan Yu, Meng Li, Vikas Chandra, and Yingyan Lin. Double-win quant: Aggressively winning robustness of quantized deep neural networks via random precision training and inference. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 3492–3504. PMLR, 18–24 Jul 2021.

[41] Yonggan Fu, Yang Zhao, Qixuan Yu, Chaojian Li, and Yingyan Lin. 2-in-1 accelerator: Enabling random precision switch for winning both adversarial robustness and efficiency. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 225–237, 2021.

[42] Y. Gan, Y. Qiu, J. Leng, M. Guo, and Y. Zhu. Ptolemy: Architecture support for robust deep learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 241–255, 2020.

[43] S. Ghosh. Design methodologies for high density domain wall memory. In *2013 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pages 30–31, July 2013.

[44] Zhangxiaowen Gong, Houxiang Ji, Christopher W Fletcher, Christopher J Hughes, Sara Baghsorkhi, and Josep Torrellas. Save: Sparsity-aware vector engine for accelerating dnn training and inference on cpus. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 796–810. IEEE, 2020.

[45] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[46] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *ICLR 2015 : International Conference on Learning Representations 2015*, 2015.

[47] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. Speech recognition with deep recurrent neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*, pages 6645–6649, 2013.

[48] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 2016.

[49] 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)*, pages 629–634, Jan 2017.

[50] Amira Guesmi, Ihsen Alouani, Khaled N Khasawneh, Mouna Baklouti, Tarek Frikha, Mohamed Abid, and Nael Abu-Ghazaleh. Defensive approximation: securing cnns using approximate computing. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 990–1003, 2021.

[51] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 75–84. ACM, 2017.

[52] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: efficient inference engine on compressed deep neural network. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, pages 243–254, 2016.

[53] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep speech: Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567, 2014.

[54] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[55] Zhezhi He, Adnan Siraj Rakin, and Deliang Fan. Parametric noise injection: Trainable randomness to improve deep neural network robustness against adversarial attack. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 588–597, 2019.

[56] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[57] Hossein Hosseini, Sreeram Kannan, and Radha Poovendran. Are odds really odd? bypassing statistical detection of adversarial examples. *arXiv preprint arXiv:1907.12138*, 2019.

[58] Shengyuan Hu, Tao Yu, Chuan Guo, Wei-Lun Chao, and Kilian Weinberger. A new defense against adversarial images: Turning a weakness into a strength. In *NeurIPS 2019 : Thirty-third Conference on Neural Information Processing Systems*, pages 1633–1644, 2019.

[59] Chao-Tsung Huang. Ringcnn: Exploiting algebraically-sparse ring tensors for energy-efficient cnn-based computational imaging. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1096–1109, 2021.

[60] Kejie Huang, Rong Zhao, and Yong Lian. Racetrack memory-based nonvolatile storage elements for multicontext fpgas. *IEEE Trans. VLSI Syst.*, 24(5):1885–1894, 2016.

[61] Anirudh Iyengar and Swaroop Ghosh. Modeling and analysis of domain wall dynamics for robust and low-power embedded memory. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 65:1–65:6, New York, NY, USA, 2014. ACM.

[62] Jun-Woo Jang, Sehwan Lee, Dongyoung Kim, Hyunsun Park, Ali Shafiee Ardestani, Yeongjae Choi, Channoh Kim, Yoojin Kim, Hyeongseok Yu, Hamzah Abdel-Aziz, Jun-Seok Park, Heonsoo Lee, Dongwoo Lee, Myeong Woo Kim, Hanwoong Jung, Heewoo Nam, Dongguen Lim, Seungwon Lee, Joon-Ho Song, Suknam Kwon, Joseph Hassoun, SukHwan Lim, and Changkyu Choi. Sparsity-aware and re-configurable npu architecture for samsung flagship mobile soc. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 15–28, 2021.

[63] Ahmadreza Jeddi, Mohammad Javad Shafiee, Michelle Karg, Christian Scharfenberger, and Alexander Wong. Learn2perturb: an end-to-end feature perturbation learning to improve adversarial robustness. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1241–1250, 2020.

[64] Tian Jin and Seokin Hong. Split-cnn: Splitting window-based operations in convolutional neural networks for memory system optimization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 835–847, New York, NY, USA, 2019. ACM.

[65] Norman P Jouppi and et. al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[66] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. Ten lessons from three generations shaped google's tpuv4i : Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2021.

[67] Hyeong-Ju Kang. Accelerator-aware pruning for convolutional neural networks. *IEEE Transactions on Circuits and Systems for Video Technology*, 30(7):2093–2103, 2019.

[68] Faiq Khalid, Hassan Ali, Hammad Tariq, Muhammad Abdullah Hanif, Semeen Rehman, Rehan Ahmed, and Muhammad Shafique. Qusecnets: Quantization-based defense mechanism for securing deep neural network against adversarial attacks. In *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 182–187. IEEE, 2019.

[69] Duckhwan Kim, Jaeha Kung, Sek M. Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, pages 380–392, 2016.

[70] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[71] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[72] H.T. Kung, Bradley McDanel, and Sai Qian Zhang. Packing sparse convolutional neural networks for efficient systolic array implementations: Column

combining under joint optimization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 821–834, New York, NY, USA, 2019. ACM.

[73] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial examples in the physical world. In *International Conference on Learning Representations (ICLR)*, 2017.

[74] Mathias Lecuyer, Vaggelis Atlidakis, Roxana Geambasu, Daniel Hsu, and Suman Jana. Certified robustness to adversarial examples with differential privacy. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 656–672. IEEE, 2019.

[75] Chen-Lu Li, Yu-Jie Huang, Yu-Jie Cai, Jun Han, and Xiao-Yang Zeng. Fpga implementation of lstm based on automatic speech recognition. In *2018 14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, pages 1–3. IEEE, 2018.

[76] 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*, pages 111–118, May 2015.

[77] Z. Li, C. Ding, S. Wang, W. Wen, Y. Zhuo, C. Liu, Q. Qiu, W. Xu, X. Lin, X. Qian, and Y. Wang. E-rnn: Design optimization for efficient recurrent neural networks in fpgas. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 69–80, Feb 2019.

[78] Bin Liang, Hongcheng Li, Miaoqiang Su, Xirong Li, Wenchang Shi, and Xiaofeng Wang. Detecting adversarial image examples in deep networks with adaptive noise reduction. *arXiv preprint arXiv:1705.08378*, 2017.

[79] Robert LiKamWa, Yunhui Hou, Yuan Gao, Mia Polansky, and Lin Zhong. Redeye: Analog convnet image sensor architecture for continuous mobile vision. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, pages 255–266, 2016.

[80] Ji Lin, Chuang Gan, and Song Han. Defensive quantization: When efficiency meets robustness. *arXiv preprint arXiv:1904.08444*, 2019.

[81] Dao-Fu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Temam, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. Pudiannao: A polyvalent machine learning accelerator. In *Proceedings of the Twentieth International*

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015, pages 369–381, 2015.

[82] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. Cambricon: An instruction set architecture for neural networks. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, pages 393–405, 2016.

[83] Yanqi Liu, Alessandro Costantini, R. Iris Bahar, Zhiqiang Sui, Zhefan Ye, Shiyang Lu, and Odest Chadwicke Jenkins. Robust object estimation using generative-discriminative inference for secure robotics applications. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, page 75, 2018.

[84] Y. Long, E. M. Jung, J. Kung, and S. Mukhopadhyay. Reram crossbar based recurrent neural network for human activity detection. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 939–946, July 2016.

[85] Pei-Hsuan Lu, Pin-Yu Chen, and Chia-Mu Yu. On the limitation of local intrinsic dimensionality for characterizing the subspaces of adversarial examples. In *ICLR (Workshop)*, 2018.

[86] Shiqing Ma, Yingqi Liu, Guanhong Tao, Wen-Chuan Lee, and Xiangyu Zhang. Nic: Detecting adversarial samples with neural network invariant checking. In *NDSS*, 2019.

[87] Xingjun Ma, Bo Li, Yisen Wang, Sarah M. Erfani, Sudanthi Wijewickrema, Michael E. Houle, Grant Schoenebeck, Dawn Song, and James Bailey. Characterizing adversarial subspaces using local intrinsic dimensionality. *arXiv preprint arXiv:1801.02613*, 2018.

[88] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations (ICLR)*, 2018.

[89] Saeed Mahloujifar, Xiao Zhang, Mohammad Mahmoody, and David Evans. Empirically measuring concentration: Fundamental limits on intrinsic robustness. In *33rd Conference on Neural Information Processing Systems (NeurIPS)*, pages 5210–5221, 2019.

[90] Mostafa Mahmoud, Isak Edo, Ali Hadi Zadeh, Omar Mohamed Awad, Gennady Pekhimenko, Jorge Albericio, and Andreas Moshovos. Tensordash: Exploiting sparsity to accelerate deep neural network training. In *2020 53rd Annual*

IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 781–795. IEEE, 2020.

[91] Thomas Mealey and Tarek M Taha. Accelerating inference in long short-term memory neural networks. In *NAECON 2018-IEEE National Aerospace and Electronics Conference*, pages 382–390. IEEE, 2018.

[92] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 86–94, 2017.

[93] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: A simple and accurate method to fool deep neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[94] S. Motaman, A. S. Iyengar, and S. Ghosh. Domain wall memory-layout, circuit and synergistic systems. *IEEE Transactions on Nanotechnology*, 14(2):282–291, March 2015.

[95] Seyedhamidreza Motaman and Swaroop Ghosh. Adaptive write and shift current modulation for process variation tolerance in domain wall caches. *IEEE Trans. VLSI Syst.*, 24(3):944–953, 2016.

[96] Seyedhamidreza Motaman, Anirudh Iyengar, and Swaroop Ghosh. Synergistic circuit and system design for energy-efficient and robust domain wall caches. In *International Symposium on Low Power Electronics and Design, ISLPED'14, La Jolla, CA, USA - August 11 - 13, 2014*, pages 195–200, 2014.

[97] Seyedhamidreza Motaman, Anirudh Srikant Iyengar, and Swaroop Ghosh. Domain wall memory-layout, circuit and synergistic systems. *IEEE Transactions on Nanotechnology*, 14(2):282–291, 2015.

[98] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina. Evoapprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 258–261, March 2017.

[99] V. Mrazek, Z. Vasicek, L. Sekanina, M. A. Hanif, and M. Shafique. Alwann: Automatic layer-wise approximation of deep neural network accelerators without retraining. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, Nov 2019.

[100] E. Nurvitadhi, Jaewoong Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr. Accelerating recurrent neural networks in analytics servers: Comparison of fpga, cpu, gpu, and asic. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Aug 2016.

[101] Priyadarshini Panda. Quanos: adversarial noise sensitivity driven hybrid quantization of neural networks. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 187–192, 2020.

[102] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 582–597, 2016.

[103] Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *IEEE European Symposium Security and Privacy (EuroS&P)*, 2016.

[104] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pages 311–318, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[105] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 27–40, New York, NY, USA, 2017. ACM.

[106] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 45(2):27–40, 2017.

[107] Stuart S. P. Parkin, Masamitsu Hayashi, and Luc Thomas. Magnetic domain-wall racetrack memory. *Science*, 320(5873):190–194, 2008.

[108] Jay M. Ponte and W. Bruce Croft. A language modeling approach to information retrieval. In *SIGIR '98: Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, August 24-28 1998, Melbourne, Australia*, pages 275–281, 1998.

[109] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020*

IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 58–70, 2020.

[110] Yuxian Qiu, Jingwen Leng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. Adversarial defense through network profiling based path extraction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4777–4786, 2019.

[111] Adnan Siraj Rakin, Jinfeng Yi, Boqing Gong, and Deliang Fan. Defend deep neural networks against adversarial examples via fixed and dynamic quantized activation functions. *arXiv preprint arXiv:1807.06714*, 2018.

[112] A. Ranjan, S. G. Ramasubramanian, R. Venkatesan, V. Pai, K. Roy, and A. Raghunathan. Dyrectape: A dynamically reconfigurable cache using domain wall memory tapes. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 181–186, March 2015.

[113] Brandon Reagen, Paul N. Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David M. Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *isca*, 2016.

[114] Kevin Roth, Yannic Kilcher, and Thomas Hofmann. The odds are odd: A statistical test for detecting adversarial examples. In *ICML*, volume 97 of *Proceedings of Machine Learning Research*, pages 5498–5507, 2019.

[115] Bita Darvish Rouhani, Mohammad Samragh, Mojan Javaheripi, Tara Javidi, and Farinaz Koushanfar. Deepfense: Online accelerated defense against adversarial deep learning. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[116] Hadi Salman, Mingjie Sun, Greg Yang, Ashish Kapoor, and J Zico Kolter. Denoised smoothing: A provable defense for pretrained classifiers. *arXiv preprint arXiv:2003.01908*, 2020.

[117] Ali Shafahi, Ronny Huang, Christoph Studer, Soheil Feizi, and Tom Goldstein. Are adversarial examples inevitable. In *7th International Conference on Learning Representations (ICLR)*, 2019.

[118] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*, pages 14–26, 2016.

[119] Yash Sharma and Pin-Yu Chen. Bypassing feature squeezing by increasing adversary strength. *arXiv preprint arXiv:1803.09868*, 2018.

[120] Yongming Shen, Michael Ferdman, and Peter Milder. Maximizing cnn accelerator efficiency through resource partitioning. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 535–547, New York, NY, USA, 2017. ACM.

[121] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[122] Clinton W. Smullen, Anurag Nigam, Sudhanva Gurumurthi, and Mircea R. Stan. The stetsims stt-ram simulation and modeling system. In *iccad*, pages 318–325. IEEE Press, 2011.

[123] Z. Sun, X. Bi, W. Wu, S. Yoo, and H. (. Li. Array organization and data management exploration in racetrack memory. *IEEE Transactions on Computers*, 65(4):1041–1054, April 2016.

[124] Zhanrui Sun, Yongxin Zhu, Yu Zheng, Hao Wu, Zihao Cao, Peng Xiong, Junjie Hou, Tian Huang, and Zhiqiang Que. Fpga acceleration of lstm based on data for test flight. In *2018 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 1–6. IEEE, 2018.

[125] Zhenyu Sun, Wenqing Wu, and Hai (Helen) Li. Cross-layer racetrack memory design for ultra high density and low power consumption. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 53:1–53:6, New York, NY, USA, 2013. ACM.

[126] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations 2014*, 2014.

[127] MT Tommiska. Efficient digital implementation of the sigmoid function for reprogrammable logic. *IEE Proceedings-Computers and Digital Techniques*, 150(6):403–411, 2003.

[128] Antonio Toral and Andy Way. What level of quality can neural machine translation attain on literary text? In *Translation Quality Assessment*, pages 263–287. Springer, 2018.

[129] Florian Tramer, Nicholas Carlini, Wieland Brendel, and Aleksander Madry. On adaptive attacks to adversarial example defenses. *arXiv preprint arXiv:2002.08347*, 2020.

[130] F. Vaverka, V. Mrazek, Z. Vasicek, L. Sekanina, M. A. Hanif, and M. Shafique. Tfapprox: Towards a fast emulation of dnn approximate hardware accelerators on gpu. In *2020 Design, Automation and Test in Europe Conference (DATE)*, page 4, 2020.

[131] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, and Anand Raghunathan. Scaledeep: A scalable compute architecture for learning and evaluating deep networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 13–26, New York, NY, USA, 2017. ACM.

[132] R. Venkatesan, M. Sharad, K. Roy, and A. Raghunathan. Dwm-tapestri - an energy efficient all-spin cache using domain wall shift based writes. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1825–1830, March 2013.

[133] Rangharajan Venkatesan, Vivek J. Kozhikkottu, Mrigank Sharad, Charles Augustine, Arijit Raychowdhury, Kaushik Roy, and Anand Raghunathan. Cache design with domain wall memory. *IEEE Trans. Comput.*, 65(4):1010–1024, April 2016.

[134] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: Lessons learned from the 2015 MSCOCO image captioning challenge. *CoRR*, abs/1609.06647, 2016.

[135] M. Wang, Z. Wang, J. Lu, J. Lin, and Z. Wang. E-lstm: An efficient hardware architecture for long short-term memory. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, pages 1–1, 2019.

[136] Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, Yanzhi Wang, and Yun Liang. C-lstm: Enabling efficient lstm using structured compression techniques on fpgas. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, pages 11–20, New York, NY, USA, 2018. ACM.

[137] X. Wang, J. Yu, C. Augustine, R. Iyer, and R. Das. Bit prudent in-cache acceleration of deep convolutional neural networks. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 81–93, Feb 2019.

[138] Xingbin Wang, Rui Hou, Boyan Zhao, Fengkai Yuan, Jun Zhang, Dan Meng, and Xuehai Qian. Dnnguard: An elastic heterogeneous dnn accelerator architecture against adversarial attacks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages*

and Operating Systems, ASPLOS '20, page 19–34, New York, NY, USA, 2020. Association for Computing Machinery.

[139] Yuhao Wang, Hao Yu, Leibin Ni, Guang-Bin Huang, Mei Yan, Chuliang Weng, Wei Yang, and Junfeng Zhao. An energy-efficient nonvolatile in-memory computing architecture for extreme learning machine by domain-wall nanowire devices. *IEEE Transactions on Nanotechnology*, 14(6):998–1012, 2015.

[140] Yuhao Wang, Hao Yu, Dennis Sylvester, and Pingfan Kong. Energy efficient in-memory AES encryption based on nonvolatile domain-wall nanowire. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–4, 2014.

[141] Yulong Wang, Hang Su, Bo Zhang, and Xiaolin Hu. Interpret neural networks by identifying critical data routing paths. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8906–8914, 2018.

[142] Zhisheng Wang, Jun Lin, and Zhongfeng Wang. Accelerating recurrent neural networks: A memory-efficient approach. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2763–2775, 2017.

[143] Cihang Xie, Jianyu Wang, Zhishuai Zhang, Zhou Ren, and Alan Yuille. Mitigating adversarial effects through randomization. In *International Conference on Learning Representations (ICLR)*, 2018.

[144] Xilinx. CHaiDNN. https://github.com/Xilinx/CHaiDNN.

[145] Cong Xu, Dimin Niu, Xiaochun Zhu, Seung H. Kang, Matt Nowak, and Yuan Xie. Device-architecture co-optimization of STT-RAM based memory for low power embedded systems. In *iccad*, pages 463–470. IEEE Press, 2011.

[146] Weilin Xu, David Evans, and Yanjun Qi. Feature squeezing: Detecting adversarial examples in deep neural networks. In *Proceedings 2018 Network and Distributed System Security Symposium*, 2018.

[147] Hao Yu, Yuhao Wang, Shuai Chen, Wei Fei, Chuliang Weng, Junfeng Zhao, and Zhulin Wei. Energy efficient in-memory machine learning for data intensive image-processing by non-volatile domain-wall memory. In *19th Asia and South Pacific Design Automation Conference, ASP-DAC 2014, Singapore, January 20-23, 2014*, pages 191–196, 2014.

[148] Chao Zhang, Guangyu Sun, Weiqi Zhang, Fan Mi, Hai Li, and W. Zhao. Quantitative modeling of racetrack memory, a tradeoff among area, performance, and power. In *The 20th Asia and South Pacific Design Automation Conference*, pages 100–105, Jan 2015.

[149] Chao Zhang, Guangyu Sun, Xian Zhang, Weiqi Zhang, Weisheng Zhao, Tao Wang, Yun Liang, Yongpan Liu, Yu Wang, and Jiwu Shu. Hi-fi playback: Tolerating position errors in shift operations of racetrack memory. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 694–706, New York, NY, USA, 2015. ACM.

[150] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[151] Y. Zhang, C. Zhang, J. Nan, Z. Zhang, X. Zhang, J. O. Klein, D. Ravelosona, G. Sun, and W. Zhao. Perspectives of racetrack memory for large-capacity on-chip memory: From device to system. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 63(5):629–638, May 2016.

[152] Yiwei Zhang, Chao Wang, Lei Gong, Yuntao Lu, Fan Sun, Chongchong Xu, Xi Li, and Xuehai Zhou. A power-efficient accelerator based on fpgas for lstm network. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 629–630. IEEE, 2017.

[153] Weisheng Zhao, Nesrine Ben Romdhane, Yue Zhang, Jacques-Olivier Klein, and Define Ravelosona. Racetrack memory based reconfigurable computing. In *Faible Tension Faible Consommation (FTFC), 2013 IEEE*, pages 1–4. IEEE, 2013.