UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Holistic Algorithm and System Co-Optimization for Trustworthy and Platform-Aware Deep Learning

Permalink
https://escholarship.org/uc/item/4vb1b6p2

Author
Javaheripi, Mojan

Publication Date
2023

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA SAN DIEGO

Holistic Algorithm and System Co-Optimization for Trustworthy and Platform-Aware Deep Learning

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

in

Electrical Engineering (Computer Engineering)

by

Mojan Javaheripi

Committee in charge:

Professor Farinaz Koushanfar, Chair
Professor Tara Javidi
Professor Truong Nguyen
Professor Rose Yu

2023

Copyright

Mojan Javaheripi, 2023

All rights reserved.

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

University of California San Diego

2023

DEDICATION

This dissertation is dedicated to my parents and my beloved husband for their constant love and support during my academic journey.

EPIGRAPH

"*Research is to see what everybody else has seen,*
and to think what nobody else has thought."
- Albert Szent-Györgyi

TABLE OF CONTENTS

Dissertation Approval Page iii

Dedication iv

Epigraph v

Table of Contents vi

List of Figures x

List of Tables xix

Acknowledgements xxii

Vita xxv

Abstract of the Dissertation xxviii

Chapter 1 Introduction 1
1.1 Trustworthy and Robust Deep Learning 2
1.1.1 Robustness to Runtime Attacks 3
1.1.2 Robustness to Training Time Attacks 4
1.2 Platform-aware Deep Learning on Massive Data 5
1.2.1 Efficient Training 5
1.2.2 Resource-Customized Inference 6
1.3 Acknowledgements 8

Chapter 2 Ensuring DL Robustness to Adversarial Attacks 10
2.1 Background and Preliminaries 13
2.2 Related Work 15
2.3 Statistical Analysis of Adversarial Samples 16
2.4 CuRTAIL Methodology 17
2.4.1 Latent Defenders 18
2.4.2 Input Defender 23
2.4.3 Model Fusion 25
2.4.4 Sensitivity Analysis 27
2.5 CuRTAIL Hardware Implementation 28
2.5.1 CuRTAIL Hardware Acceleration 29
2.5.2 Automated Design Customization 34
2.5.3 Computational Analysis and Scalability 36
2.6 Evaluations 37
2.6.1 Details of MRR Training 38
2.6.2 Attack Analysis and Resiliency 39

2.6.3 Black-Box Attacks 40
2.6.4 Adaptive White-Box Attack 41
2.6.5 Performance Analysis 43
2.6.6 Discussion on Transferability of Adversarial Samples 45
2.7 Conclusion 46
2.8 Acknowledgements 46

Chapter 3 Ensuring DL Robustness to Fault Injection Attacks 48
3.1 Background and Prior Work 51
3.1.1 Bit-Flip Attack 51
3.1.2 Existing Defenses 52
3.2 AccHashtag Methodology 53
3.2.1 Threat Model 55
3.3 AccHashtag Components 56
3.3.1 Hash-based Signature Extraction 56
3.3.2 Bounds on Detection Performance 57
3.3.3 Per-layer Sensitivity Analysis 60
3.3.4 Accelerating Hash Generation 61
3.4 Experiments 63
3.4.1 Experimental Setup 63
3.4.2 Analysis of Design Choices 65
3.4.3 AccHashtag Performance 68
3.5 Conclusion 75
3.6 Acknowledgements 76

Chapter 4 Ensuring DL Robustness to Backdoor Attacks 77
4.1 Background on Trojan Attacks and Defenses 79
4.2 CleaNN Methodology 81
4.2.1 Defense Construction and Execution 83
4.2.2 Threat Model 83
4.3 CleaNN Components 84
4.3.1 DCT extraction 84
4.3.2 Sparse Recovery 85
4.3.3 Detection 88
4.4 CleaNN Hardware 90
4.5 Experiments 92
4.5.1 Attack Configuration 92
4.5.2 Detection Performance 93
4.5.3 Hardware performance 95
4.6 Conclusion 98
4.7 Acknowledgment 99

Chapter 5 Improving Training Convergence via Architectural Modifications 100
5.1 Background on Small-World Networks 102

5.2 Related work . . . 103
5.3 SWANN: Small-World DNNs . . . 106
5.3.1 Metric for Small-Worldness . . . 106
5.3.2 Acquiring the Small-world Architecture . . . 107
5.3.3 SWANN Methodology . . . 110
5.4 Experiments . . . 113
5.4.1 Datasets . . . 113
5.4.2 Benchmarked Architectures . . . 114
5.4.3 Results on MNSIT . . . 114
5.4.4 Results on CIFAR . . . 115
5.4.5 Results on ImageNet . . . 118
5.4.6 Federated Learning . . . 120
5.5 Discussion on Long-range Connections . . . 122
5.6 Conclusion . . . 123
5.7 Acknowledgements . . . 124

Chapter 6 Improving Inference Performance via Neural Architecture Search . . . 125
6.1 Related Work . . . 128
6.2 Lightweight Transformer Search . . . 129
6.2.1 Training-free Architecture Ranking . . . 132
6.3 Experiments . . . 133
6.3.1 Experimental Setup . . . 134
6.3.2 How do training-free proxies perform compared to training-based methods? . . . 136
6.3.3 How does variation in model topology affect decoder parameter count as a proxy? . . . 139
6.3.4 How Good is the Decoder Parameters Proxy for Pareto-frontier Search? 141
6.3.5 Pareto-frontier models for various hardware platforms . . . 143
6.3.6 Zero and one-shot performance comparison with OPT . . . 147
6.4 Conclusion . . . 149
6.5 Acknowledgements . . . 149

Chapter 7 Automating Model Compression via Adaptive Non-uniform Sampling . . . 150
7.1 Background and Related Work . . . 153
7.2 Problem Formulation . . . 155
7.3 *AdaNS* Overview . . . 156
7.3.1 Search-Space Definition . . . 157
7.3.2 Scoring Mechanism . . . 157
7.3.3 Boundary Characterization for Directed Search . . . 159
7.3.4 Optimization through Adaptive Sampling . . . 161
7.4 *AdaNS* Adaptive Sampling Routines . . . 163
7.4.1 *AdaNS-Zoom* Sampling Subroutine . . . 164
7.4.2 *AdaNS-Genetic* Sampling Subroutine . . . 166
7.4.3 *AdaNS-Gaussian* Sampling Subroutine . . . 169

7.5 Reconstruction 171
7.6 Experiments 173
7.6.1 Effect of Sampling Strategy on Convergence 174
7.6.2 Quantitative Results on CIFAR-10 176
7.6.3 Quantitative Results on ImageNet 178
7.6.4 Compressing Compact Networks 178
7.6.5 Search Overhead and Scalability 179
7.6.6 Analysis and Discussion 181
7.6.7 Ablation Study 182
7.7 Conclusion 184
7.8 Acknowledgements 185

Bibliography 186

LIST OF FIGURES

Figure 1.1. Critical design objectives for autonomous DL-enabled systems. My research provides holistic solutions that co-optimize DL algorithm and hardware, thus jointly satisfying multiple design objectives, i.e., task evaluation metrics, robustness, and hardware performance. 2

Figure 2.1. (a) Data points (green and blue squares) can be easily separated in one-dimensional space. Extra dimensions add ambiguity in choosing the decision boundaries: all shown boundaries (dashed lines) result in the same classification accuracy but are not equally robust to noise. 11

Figure 2.2. An example of (a) input data, and (b) its corresponding adversarial sample. The added noise is imperceptible but can cause the victim model to misclassify. 13

Figure 2.3. Histogram of the estimated PDF for adversarial (red) and legitimate (blue) samples. Adversarial samples are generated using Deepfool for *Lenet* architecture. The PDF is learned in the second-to-last layer of the network. 17

Figure 2.4. High-level block diagram of MRR methodology. Multiple defenders checkpoint the input and intermediate features in parallel. The output of the victim DNN (green neurons) is augmented with a confidence measure (red neuron) determining the prediction legitimacy. 18

Figure 2.5. Block diagram of the training procedure for devising parallel redundancy modules. Each latent defender is built by minimizing the entanglement of intermediate data features in a Euclidean space at a particular checkpoint location. 19

Figure 2.6. Defender module optimization objective. 20

Figure 2.7. (a) Distance of legitimate (blue) and adversarial (red) samples from the corresponding centers C^i before, and (b) after realignment of data samples. In this example, we consider the *LeNet* model [106] trained on MNIST. . . 20

Figure 2.8. Illustration of the effect of security parameter on the detection policy. A high SP leads to a tight boundary which treats most samples as adversarial examples. 21

Figure 2.9. Enforcing negative correlation between MRRs. 22

Figure 2.10. Training multiple negatively correlated defenders at each checkpoint layer. 22

Figure 2.11. An input defender module is devised based on robust dictionary learning techniques to automatically filter out test samples that highly deviate from the typical PSNR of data points within the corresponding predicted class. . 23

Figure 2.12. Adversarial detection rate of input and latent defenders as a function of the perturbation level for various SP. Here, FGS is used to generate adversarial samples and the perturbation is adjusted by changing attack parameter ϵ. 25

Figure 2.13. CuRTAIL uses a score-based statistical method to aggregate MRR decisions. 26

Figure 2.14. Per-layer sensitivity analysis for ResNet56 28

Figure 2.15. Complexity and reliability trade-off for the LeNet model on MNIST dataset performed on an NVIDIA Geforce 980 GPU hosted by an Intel Core-i7 CPU. 29

Figure 2.16. Overview of CuRTAIL hardware acceleration stack. Based on the user-provided constraints, we output the best defense layout that ensures maximum robustness and throughput. 30

Figure 2.17. Latent defender structure: the pertinent activations are acquired by propagating the input sample through the defender. PCA is then applied to reduce the dimensionality of the obtained activation. The L_2 distance with the corresponding GMM center determines the legitimacy of the input. . . . 31

Figure 2.18. Design space exploration for MNIST and SVHN benchmarks on *Xilinx Zynq-ZC702* FPGA. CuRTAIL finds the optimal configuration of PEs and PUs to best fit the DL architecture and the available hardware resources. . . 32

Figure 2.19. Input defender: the OMP core iteratively reconstructs input vectors using a learned dictionary. Here, the *support set* contains columns of the dictionary that have been chosen so far in the routine. The final reconstruction error is used to determine input legitimacy. 33

Figure 2.20. Tree-based vector reduction algorithm. 33

Figure 2.21. CuRTAIL provides customized defense by balancing the design-space trade-offs. The goal of CuRTAIL is to maximize model robustness while adhering to the underlying memory and runtime constraints. 34

Figure 2.22. Example legitimate samples in ImageNet benchmark. Samples are randomly selected from the target classes. 38

Figure 2.23. CuRTAIL security parameter controls the TP and FP rates. The number of latent defenders in this experiment is 16. . . . 39

Figure 2.24. Using more MRRs improves the detection performance for all datasets. . . 39

Figure 2.25. CuRTAIL throughput with samples from the MNIST dataset versus the number of instantiated defenders (implemented on *Xilinx Zync-ZC702* FPGA). . . . 44

Figure 2.26. Performance-per-Watt comparison with embedded CPU (left) and CPU-GPU (right) platforms. Reports are normalized by the performance-per-Watt of *TK1*. . . . 45

Figure 2.27. Example adversarial confusion matrix (a) without MRR defense mechanism, and (b) with MRR defense and $SP = 1\%$. (c) Example adversarial samples for which accurate detection is hard due to the closeness of decision boundaries 45

Figure 3.1. Global flow of ACCHASHTAG detection. During the pre-processing phase, we first identify sensitive DNN layers that are most prone to fault-injection attacks. We then generate a customized signature from the identified sensitive layers. . . . 55

Figure 3.2. Reordering parameters in an example convolution layer for generating the hash input stream. The layer parameters are the convolution weight kernels $\in \mathbb{R}^{k\times k\times C_i\times C_o}$ where k, C_i, C_o denote the kernel size, input channels, and output channels, respectively. . . . 57

Figure 3.3. Collision rate versus number of trial runs for hashing an input stream of length 1000. Each trial randomly changes a subset $k \in [2, 3, 6, 8, 12, 16]$ of message elements. . . . 59

Figure 3.4. Overview of ACCHASHTAG accelerated hash generation and verification using a specialized FPGA compute kernel. . . . 62

Figure 3.5. Percentage of sign changes occurring during multiple runs of the bit-flip attack. The progressive bit-flip attack [154] changes the sign of the target parameter with high probability. . . . 65

Figure 3.6. Ranking DNN layers based on their vulnerability to bit-flips, defined as the average sensitivity score assigned to their γ most vulnerable weights. The most vulnerable layers (marked with green boxes), remain largely the same, when $\gamma \leq 10$. . . . 67

Figure 3.7. Maximum per-layer attack concentration, averaged across multiple runs for ResNet20 and ResNet18 trained on CIFAR10 and ImageNet, respectively. The progressive bit-flip attack [154] on average targets the weights in the same layer no more than ~ 5 times. 68

Figure 3.8. Ranking DNN layers based on their sensitivity, computed using a validation dataset with n samples per class. 68

Figure 3.9. ACCHASHTAG detection rate versus number of checkpoint layers, shown for various number of per-class samples (n) used in sensitivity calculation. We omit the plot for ViT for brevity as it shows a similar trend as the ResNet18 benchmark. 69

Figure 3.10. Per-layer sensitivity scores assigned by ACCHASHTAG versus the number of per-layer bit-flips. All values are normalized and sum to 1. Results are gathered across 50 runs of the bit-flip attack on the ResNet20 DNN trained with CIFAR10 dataset. 69

Figure 3.11. ACCHASHTAG detection rate versus the number of checkpoint layers used for signature extraction, evaluated on different victim CNNs. 70

Figure 3.12. ACCHASHTAG detection performance versus number of checkpoint layers. Evaluated DNNs are derived from the Transformer backend with self-attention layers. 71

Figure 3.13. Effect of victim DNN's bitwidth on ACCHASHTAG detection rate. The legend presents the utilized datasets along with the underlying bitwidths. . 71

Figure 3.14. System throughput as a function of the design parameter $TILE$, i.e., the burst length for AXI reads. Higher $TILE$ length facilitates larger overlap between CPU-FPGA communications and hash computation, thus increasing throughput. 75

Figure 4.1. Example Trojans: (a) BadNets [58] with a sticky note and TrojanNN [121] with (b) square and (c) watermark triggers. 78

Figure 4.2. High-level overview of CLEANN Trojan detection methodology. CLEANN detects both digital and physical attacks using a pair of input and latent feature analyzers. 81

Figure 4.3. Average magnitude of DCT components for Trojan samples, normalized by benign data, shown in the three RGB channels. Trojans contain abnormally larger amounts of high-frequency components (highlighted regions). 84

Figure 4.4. Illustration of sparse reconstruction for regular data (green circle) and out-of-distribution samples (red circle). 87

Figure 4.5. (a) Example Trojan data with watermark and square triggers [121], (b) reconstruction error heatmap, and (c) output mask from the outlier detection module. 89

Figure 4.6. Schematic of CLEANN MVM core with its internal parallelization levels. . . 91

Figure 4.7. Analysis of CLEANN sensitivity to Trojan trigger size. 95

Figure 4.8. Latency breakdown of CLEANN components running on embedded and high-end CPUs (left) and GPUs (right). 97

Figure 4.9. Cycle-count breakdown for running CLEANN components on FPGA. 97

Figure 4.10. (a) Performance-per-Watt and (b) throughput across hardware platforms. Reported values for performance per-watt are normalized by *TITAN Xp* and throughput values are normalized by *ARM Cortex-A57*. 98

Figure 5.1. Schematic representation of the connections within a small-world DNN. An arbitrary neuron's output is connected to selected neurons in the proceeding layers via sparse connections (convolutions) denoted by *S-CONV*. 101

Figure 5.2. Transition of a regular graph to a completely random network. Intermediate values of the random rewiring probability, p, generate SWNs, i.e., clustered structures where any arbitrary node pair is connected by a few edges. 103

Figure 5.3. Information flow within a ResNet (top), DenseNet (middle), and SWANN (bottom). Here, *CONV*, *BN*, *ReLU* denote a convolution kernel, batch normalization, and non-linear activation, respectively, and our customized sparse convolutions are shown as *S-CONV*. 105

Figure 5.4. Our proposed rewiring algorithm replaces edges to the subsequent layer (red) with long-range edges (blue). 109

Figure 5.5. Clustering coefficient (C), small-world property (S_G), and path length (L) versus rewiring probability. The region where the graph transforms into a small-world network is shown with the double-headed arrow. 109

Figure 5.6. Convergence to 99.0% test accuracy for a 5-layer DNN and its randomly rewired counterparts trained on the MNIST dataset. Here, the relative convergence rate is computed as $\frac{e_b}{e_r}$ 110

Figure 5.7. Conversion of a *CONV* layer to its graph representation. Each $k \times k$ convolution kernel is replaced by an edge in the corresponding graph where the input and output filter channels are shown as two consecutive rows of vertices with ch_1 and ch_2 nodes, respectively. 111

Figure 5.8. Coarse-grained sparse convolution between a layer with $ch_1 = 5$ output channels and a layer with $ch_2 = 3$ output channels. Left: Sparse convolution weights. For each removed connection from the graph, we show the corresponding filter in the sparse convolution weight by zero. 112

Figure 5.9. Comparison of a plain DNN's training convergence with its small-world equivalent. Here, the red and blue colors show SWANN and baseline, respectively. The $\star$ markers denote the point of convergence to final test accuracy for the models with the corresponding colors. 115

Figure 5.10. Test error and training loss versus iterations for a ConvNet-C model and the rewired SWANN trained on (a) CIFAR10 and (b) CIFAR100 datasets. Here, the red and blue colors show SWANN and baseline, respectively. . . . 116

Figure 5.11. Training loss and testing accuracy of the 40-layer (k=12) DenseNet [77] with 1M parameters and our corresponding SWANN with less than 100K parameters. 118

Figure 5.12. Training loss and testing accuracy of the baseline DNN and SWANN in the federated learning scenario with a) IID and b) non-IID data distributions. Here, the red and blue colors show SWANN and baseline, respectively. . . . 121

Figure 5.13. Visualization of average absolute value of trained weights within $CONV$ layers of SWANNs. Colors encode the connectivity strength between layers with red being the strongest and white denoting no connection. The marked rows with black box borders correspond to the input layer of the networks. 123

Figure 6.1. High-level overview of LTS. We propose a training-free zero-cost proxy for evaluating the validation perplexity of candidate architectures. Pareto-frontier search is powered by evolutionary algorithms which use the proposed proxy along with real latency and memory measurements 127

Figure 6.2. Elastic parameters in LTS search space. 131

Figure 6.3. Our training-free zero-cost proxy based on decoder parameter count is highly correlated with the (ground truth) validation perplexity after full training. Each plot contains 200 architectures sampled randomly from the search space of Transformer-XL or GPT-2 backbone. 133

Figure 6.4. Comparison between partial training and our zero-cost proxy, i.e., decoder parameter count, in terms of ranking performance and timing overhead. Each subplot corresponds to a top$k\%$ of the randomly sampled models, based on their validation perplexity after full training. 136

Figure 6.5. SRC between low-cost proxies and the ground truth ranking after full training of randomly sampled Transformers with (a) heterogeneous and (b) homogeneous decoder blocks. The decoder parameter count obtains the best SRC with zero cost. 137

Figure 6.6. (a) Validation perplexity after full training versus total parameters for 200 randomly sampled architectures trained on WikiText-103. The downward trend suggests a strong correlation between parameter count and perplexity. 138

Figure 6.7. Performance of parameter count proxies on 100 randomly sampled Transformers with homogeneous decoder blocks, trained on WikiText-103. The decoder parameter count provides a very accurate ranking proxy with an SRC of 0.95 over all models. 139

Figure 6.8. Validation perplexity after full training versus the (a) width-to-depth aspect ratio, (b) latency, and (c) peak memory utilization. Models are randomly generated from the GPT-2 backbone and trained on WikiText-103. 140

Figure 6.9. Validation perplexity after full training versus (a) the width-to-depth aspect ratio, (b) latency, and (c) peak memory utilization. Models are randomly generated from the Transformer-XL backbone and trained on WikiText-103. 141

Figure 6.10. Perplexity versus latency Pareto obtained from full training of 1200 architectures sampled during NAS on Transformer-XL backbone. Orange points are the Pareto-frontier extracted using decoder parameter count proxy, which lies close to the actual Pareto-frontier. 142

Figure 6.11. SRC between the decoder parameter count proxy and validation perplexity. Results are gathered on 1200 models grouped into four bins based on their decoder parameter count. Our proxy performs well even when exploring within a small range of model sizes. 143

Figure 6.12. 2D visualization of perplexity versus latency and memory Pareto-frontier found by LTS, versus the scaled backbone models with varying number of layers, trained on LM1B. 144

Figure 6.13. 2D visualization of perplexity versus latency and memory Pareto-frontier found by LTS versus layer-scaled backbone models. All models are trained on WikiText-103. 145

Figure 6.14. Average zero and one-shot accuracy of LTS models (dots) and the baseline OPT-350M (triangle) across 14 NLP tasks. Latency is measured on an A6000 NVIDIA GPU. 148

Figure 7.1. Pareto curves of accuracy versus number of floating point operations for pruning a pre-trained VGG network on CIFAR-10 benchmark. 151

Figure 7.2. Overview of *AdaNS* adaptive sampling for hyperparameter customization. 156

Figure 7.3. Vectorized representation of an example 4-layer DNN for *Pruning* and *Decomposition*. Here, CONV and FC denote convolutional and fully-connected layers, respectively. 157

Figure 7.4. *AdaNS* accuracy penalty function with $A_{thr} = 80\%$ and $A(M) = 93.5\%$. . 159

Figure 7.5. Search-space for pruning a 2-layer DNN. 160

Figure 7.6. Boundary characterization for pruning a VGG model on CIFAR-10 with $A_{thr} = 80\%$. 160

Figure 7.7. *AdaNS-Zoom* algorithm. Here, the good and bad samples are shown with blue circles and red crosses, respectively. 165

Figure 7.8. Overview of *AdaNS-Genetic* sampling subroutine. 166

Figure 7.9. $2D$ illustration of *AdaNS-Gaussian* sampling strategies. 169

Figure 7.10. Per-iteration analysis of *AdaNS* sampling. 173

Figure 7.11. Comparison between *AdaNS* sampling subroutines with the same sample count, for pruning a 2-layer DNN. *AdaNS-Gaussian* achieves better exploration/exploitation tradeoff as it identifies the global maximum and concentrates the sampling around it. 175

Figure 7.12. Convergence analysis of various sampling strategies across algorithm iterations. (left): mean score achieved by good samples. (right): Proximity parameter value. 176

Figure 7.13. (a) Set of randomly initialized samples at the first iteration. (b) Set of good samples $\mathbb{S}_g$ upon convergence. 182

Figure 7.14. Original per-layer FLOPs versus *AdaNS* pruning pattern. 182

Figure 7.15. Ablation studies for VGG on CIFAR-10:(a) Effect of initialization method. (b) Effect of sample count. We show the trend lines as well as a fraction of samples (black dots) across *AdaNS* iterations. (c) Effect of mutation and cross-over probabilities for *AdaNS-Genetic*. 183

Figure 7.16. Convergence curves for various pair selection methods for *Cross* samples (Section 7.4.3). Graphs are generated over 10 runs. 184

LIST OF TABLES

Table 2.1. Runtime and computational complexity of each custom layer in CuRTAIL framework. 36

Table 2.2. Benchmarked DL models for evaluating CuRTAIL effectiveness. $Conv$ layers are represented as $\langle input{-}channels \rangle \xrightarrow[stride]{\langle kernel\ size \rangle} \langle output{-}channels \rangle$ and FC layers are denoted by $\langle output - elements \rangle$FC. 38

Table 2.3. Attack parameters. For CarliniL2 attack [22], "C" denotes the confidence, "LR" is the learning rate, "steps" is the number of binary search steps, and "iterations" stands for the maximum number of iterations. 40

Table 2.4. AUC obtained by 16 latent defenders checkpointing the second-to-last layer of the victim model. For ImageNet, we only used 1 defender due to the high computational complexity of the pertinent neural network and attacks. 41

Table 2.5. Evaluation of MRR methodology against adaptive white-box attack. We compare our results with prior work including Magnet [135], Efficient Defenses Against Adversarial Attacks [229], and APE-GAN [174]. 42

Table 3.1. High-level comparison of ACCHASHTAG with prior work. 53

Table 3.2. Resource utilization of ACCHASHTAG components, synthesized on a Xilinx FPGA. 62

Table 3.3. Overview of the evaluated benchmarks. Here, CONV, FC, and ATTN represent convolution, fully-connected, and self-attention layers, respectively. Note that each self-attention layer consists of four fully-connected layers. 64

Table 3.4. Comparison with best prior works WED [119] and RADAR [111]. Runtime is measured on an ARM CPU and normalized by the inference time of the victim DNN. 73

Table 3.5. ACCHASHTAG overhead analysis. Here, # is the number of checkpoint layers. 74

Table 4.1. Evaluated datasets and attack algorithms. 92

Table 4.2. Parameters of CLEANN modules for various datasets. P: DCT windows size, l: feature size for sparse recovery, m : number of dictionary columns for sparse recovery, λ: sparsity parameter in sparse recovery, ϵ^2: distance threshold for outlier detection. 93

Table 4.3. Evaluation of CLEANN on various physical and digital attacks. Comparisons with state-of-the-art prior works, i.e., Neural Cleanse(NC) [200], Deep Inspect (DI) [27], Februus [36], and SentiNet [29] are provided where applicable. 94

Table 5.1. Benchmarked DNNs for evaluating SWANN effectiveness. $CONV$ layers are represented as $\langle kernel\ size \rangle$Conv and FC layers are denoted by $\langle output\ elements \rangle$FC. BN and $ReLU$ are not shown for brevity. 114

Table 5.2. Graph characteristics of SWANN models. 115

Table 5.3. Point-wise comparison of convergence speed-up for a SWANN and its equivalent baseline network (ConvNet-C) on CIFAR benchmarks. 117

Table 5.4. Comparison of the computational complexity and model parameter space between a 40-layer DenseNet with k=12 and the corresponding SWANN. . 118

Table 5.5. Performance of baseline AlexNet and its SWANN on ImageNet dataset. . . 119

Table 5.6. Point-wise convergence comparison of ResNet-18 and its SWANN equivalent on ImageNet dataset. 120

Table 5.7. Point-wise convergence comparison of the 5-layer baseline DNN and its corresponding SWANN in the federated learning scenario. 122

Table 6.1. LTS training hyperparameters for different backbones. Here, DO represents dropout. 135

Table 6.2. Ranking abilities of full and partial training versus our proxy for 1200 models sampled during LTS search. Training time is reported for WikiText-103 and NVIDIA V100 GPU. Decoder parameter count proxy obtains an SRC of 0.98 using zero compute. 147

Table 7.1. Accuracy and FLOPs of the final compressed model using various sampling methods. Sample size is $b = 50$ and the sampling algorithm runs for 100 iterations on a VGG model trained on CIFAR-10. A_{thr} is set to 70% and the reported accuracy is before fine-tuning. 176

Table 7.2. Comparison with contemporary compression methods. 177

Table 7.3. Pruning of MobileNetV1&V2 on ImageNet. 180

Table 7.4. Speedup of *AdaNS* compressed MobileNets on embedded CPU and GPU after applying structured pruning on ImageNet benchmark. 180

Table 7.5. Search runtime of *AdaNS-Gaussian* for pruning on various benchmarks. Here, b denotes the number of samples per iteration and N_{iters} is the number of search iterations. 181

ACKNOWLEDGEMENTS

First and foremost, I would like to express my heartfelt gratitude to my Ph.D. advisor professor Farinaz Koushanfar, for her unwavering support, guidance, and encouragement throughout my PhD journey. Her wealth of knowledge and expertise has been invaluable in shaping my research and helping me grow as a scholar. This dissertation would not have been possible without her dedicated mentorship and I am forever grateful for their contributions to my academic and personal growth.

I would also like to extend my deepest gratitude to professor Tara Javidi, whose academic insights have been an invaluable asset in navigating the complexities of my research. Additionally, her kindness and help have been a source of comfort and support during my academic pursuits.

I am deeply grateful to Professor Truong Nguyen and Professor Rose Yu for serving on my Ph.D. committee and for their constructive feedback on my dissertation. I would also like to sincerely thank my mentors Dr. Debadeepta Dey, Dr. Subhabrata Mukherjee, and Dr. Sebastien Bubeck at Microsoft Research and Nilesh Karia, Dr. Hadi Pouransari, and Dr. Oncel Tuzel at Apple. Working with them was an excellent opportunity for me to grow as a researcher and expand the impact of my work to ground-breaking industry applications.

I was fortunate to meet and work with outstanding researchers during the course of my Ph.D. I want to thank Dr. Mohammad Samragh, Dr. Siam Hussain, Dr. Zahra Ghodsi, Dr. Bita Rouhani, Dr. Sadegh Riazi, Dr. Huili Chen, Dr. Mohsen Imani, Shehzeen Hussain, Jung-woo Chang, Ruisi Zhang, Xinqiao Zhang, and Nojan Sheybani.

I was extremely blessed to go through the bumpy roads of Ph.D. side by side with my husband, Soroush. He was there to support me every step of the way, from proofreading my papers to staying up with me on deadline nights and giving me motivational speeches whenever I felt disappointed. Your faith in me gave me the strength and motivation to overcome all challenges. I am so proud of how much we have learned, grown, and achieved together.

I am deeply indebted to my beloved parents, Mehrdad and Fariba, who always encouraged me to strive for the best. Their continuous encouragement and sacrifices have been immeasurable,

and I will forever be grateful for everything they have done for me.

I also want to thank my furry family member, Blue, whose presence and companionship provided a much-needed break from the rigors of academic work.

The material in this dissertation is based on several published papers as outlined below.

Chapter 2 is a partial reprint of the material as it appears in: M. Javaheripi, M. Samragh, B. Rouhani, T. Javidi, and F. Koushanfar, "Curtail: Characterizing and Thwarting Adversarial Deep Learning", in *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2020. The dissertation author was the primary investigator and author of this paper.

Chapter 3 is a reprint of the material as it appears in: M. Javaheripi, J. Chang, and F. Koushanfar, "AccHashtag: Accelerated Hashing for Detecting Fault-Injection Attacks on Embedded Neural Networks", in *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 2022. The dissertation author was the primary investigator and author of this paper.

Chapter 4 is a partial reprint of the material as it appears in: M. Javaheripi, M. Samragh, G. Fields, T. Javidi, and F. Koushanfar, "CleaNN: Accelerated Trojan Shield for Embedded Neural Networks", in *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020. The dissertation author and Mohammad Samragh made equal contributions to this work.

Chapter 5 is a reprint of the material as it appears in: M. Javaheripi, B. Rouhani, and F. Koushanfar, "SWANN: Small-World Architecture for Fast Convergence of Neural Networks", in *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2021. The dissertation author was the primary investigator and author of this paper.

Chapter 6, in part, was published as: M. Javaheripi, G. de Rosa, S. Mukherjee, S. Shah, T. Religa, C. Mendes, S. Bubeck, F. Koushanfar, and D. Dey, "LiteTransformerSearch: Training-free Neural Architecture Search for Efficient Language Models", in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. The dissertation author was the primary investigator and author of this paper.

Chapter 7 is, in part, a reprint of the material as it appears in two publications: (1) M. Javaheripi, M. Samragh, T. Javidi, and F. Koushanfar, "AdaNS: Adaptive Non-uniform Sam-

pling for Automated Design of Compact DNNs", in *IEEE Journal of Selected Topics in Signal Processing (JSTSP)*, 2020, and (2) M. Javaheripi, M. Samragh, T. Javidi, and F. Koushanfar, "GeneCAI: Genetic Evolution for Acquiring Compact AI", in *Genetic and Evolutionary Computation Conference*, 2020. The dissertation author was the primary investigator and author of both published papers.

This dissertation was supported, in parts, by the Qualcomm Innovation Fellowship (QIF2019-US), NSF Grants CCF-1719133 and CCF-1513883, NSF-CNS award number 2016737, NSF TILOS AI institute award number 2112665, ARO (W911NF1910317), SRC-Auto (2019-AU-2899), the Intel PrivateAI Collaborative Research Institute, and industrial partners of UCSD Center for Machine Integrated Computing and Security (MICS).

VITA

2017 Bachelor of Science in Electrical Engineering, Sharif University of Technology

2020 Master of Science in Electrical Engineering (Computer Engineering), University of California San Diego

2023 Doctor of Philosophy in Electrical Engineering (Computer Engineering), University of California San Diego

PUBLICATIONS

R Zhang, **M. Javaheripi**, Z. Ghodsi, A. Bleiweiss, and F. Koushanfar. “AdaGL: Adaptive Learning for Agile Distributed Training of Gigantic GNNs.” In Design Automation Conference (DAC), 2023.

J. Chang, **M. Javaheripi**, and F. Koushanfar. “VideoFlip: Adversarial Bit-Flips for Reducing Video Service Quality.” In Design Automation Conference (DAC), 2023.

J. Chang, **M. Javaheripi**, S. Hidano, and F. Koushanfar. “RoVISQ: Reduction of Video Service Quality via Adversarial Attacks on Deep Learning-based Video Compression.” In Network and Distributed System Security (NDSS) Symposium, 2023.

M. Javaheripi, G. de Rosa, S. Mukherjee, S. Shah, T. Religa, C. Mendes, S. Bubeck, F. Koushanfar, and D. Dey. “LiteTransformerSearch: Training-free Neural Architecture Search for Efficient Language Models.” In Advances in Neural Information Processing Systems (NeurIPS), 2022.

Z. Ghodsi*, **M. Javaheripi***, N. Sheybani*, X. Zhang*, K. Huang, and F. Koushanfar. “zPROBE: Zero Peek Robustness Checks for Federated Learning.” In NeurIPS Workshop on Trustworthy and Socially Responsible Machine Learning, 2022. (*equal contribution)

M. Javaheripi, J. Chang, and F. Koushanfar. “AccHashtag: Accelerated Hashing for Detecting Fault-Injection Attacks on Embedded Neural Networks.” In ACM Journal on Emerging Technologies in Computing Systems (JETC), 2022.

M. Javaheripi, B. Rouhani, and F. Koushanfar. “SWANN: Small-World Architecture for Fast Convergence of Neural Networks.” In IEEE Journal on Emerging and Selected Topics in Circuits and Systems, 2021.

M. Javaheripi, M. Samragh, and F. Koushanfar. “AutoRank: Automated Rank Selection for Effective Neural Network Customization.” In IEEE Journal on Emerging and Selected Topics in Circuits and Systems, 2021.

S. Hussain*, **M. Javaheripi***, M. Samragh*, and F. Koushanfar. "COINN: Crypto/ML Codesign for Oblivious Inference via Neural Networks." In ACM SIGSAC Conference on Computer and Communications Security (CCS), 2021. (*equal contribution)

M. Javaheripi, and F. Koushanfar. "HASHTAG: Hash Signatures for Online Detection of Fault-Injection Attacks on Deep Neural Networks." In IEEE/ACM International Conference On Computer Aided Design (ICCAD), 2021.

M. Javaheripi, M. Samragh, B. Rouhani, T. Javidi, and F. Koushanfar. "Hardware/Algorithm Codesign for Adversarially Robust Deep Learning." In IEEE Design and Test, 2021.

G. Fields, M. Samragh, **M. Javaheripi**, F. Koushanfar, and T. Javidi. "Trojan Signatures in DNN Weights." In IEEE/CVF International Conference on Computer Vision (ICCV) Workshops, 2021.

H. Pouransari, **M. Javaheripi**, V. Sharma, and O. Tuzel. "Extracurricular Learning: Knowledge Transfer Beyond Empirical Distribution." In IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops, 2021.

M. Javaheripi, M. Samragh, G. Fields, T. Javidi, and F. Koushanfar. "CleaNN: Accelerated Trojan Shield for Embedded Neural Networks." In IEEE/ACM International Conference On Computer Aided Design (ICCAD), 2020.

M. Samragh, **M. Javaheripi**, and F. Koushanfar. "Encodeep: Realizing Bit-flexible Encoding for Deep Neural Networks." In ACM Transactions on Embedded Computing Systems (TECS), 2020.

M. Javaheripi, M. Samragh, B. Rouhani, T. Javidi, and F. Koushanfar. "Curtail: Characterizing and Thwarting Adversarial Deep Learning." In IEEE Transactions on Dependable and Secure Computing (TDSC), 2020.

M. Javaheripi, H. Chen, and F. Koushanfar. "Unified Architectural Support for Secure and Robust Deep Learning." In ACM/IEEE Design Automation Conference (DAC), 2020.

M. Javaheripi, M. Samragh, T. Javidi, and F. Koushanfar. "AdaNS: Adaptive Non-uniform Sampling for Automated Design of Compact DNNs." In IEEE Journal of Selected Topics in Signal Processing (JSTSP), 2020.

M. Javaheripi, M. Samragh, T. Javidi, and F. Koushanfar. "GeneCAI: Genetic Evolution for Acquiring Compact AI." In Genetic and Evolutionary Computation Conference, 2020.

M. Javaheripi, M. Samragh, and F. Koushanfar. "Peeking into the black box: A tutorial on Automated Design Optimization and Parameter Search." In IEEE Solid-State Circuits Magazine, 2019.

S. Hussain, **M. Javaheripi**, P. Neekhara, R. Kastner, and F. Koushanfar. "Fastwave: Acceler-

ating Autoregressive Convolutional Neural Networks on FPGA." In IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2019.

M. Riazi, **M. Javaheripi**, S. Hussain, and F. Koushanfar. "MPCircuits: Optimized Circuit Generation for Secure Multi-party Computation." In IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2019.

M. Imani, S. Bosch, **M. Javaheripi**, B. Rouhani, X. Wu, F. Koushanfar, and T. Rosing. "Semihd: Semi-supervised Learning Using Hyperdimensional Computing." In IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2019.

B. Rouhani, M. Samragh, **M. Javaheripi**, T. Javidi, and F. Koushanfar. "Assured Deep Learning: Practical Defense Against Adversarial Attacks." In IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2018.

B. Rouhani, M. Samragh, **M. Javaheripi**,T. Javidi, and F. Koushanfar. "Deepfense: Online Accelerated Defense Against Adversarial Deep Learning." In IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2018.

ABSTRACT OF THE DISSERTATION

Holistic Algorithm and System Co-Optimization for Trustworthy and Platform-Aware Deep Learning

by

Mojan Javaheripi

Doctor of Philosophy in Electrical Engineering (Computer Engineering)

University of California San Diego, 2023

Professor Farinaz Koushanfar, Chair

Simultaneous growth in the volume of available data along with rapid advancements in computing and hardware technology have paved the way for unprecedented breakthroughs in the field of Artificial Intelligence (AI). In particular, a modern class of AI algorithms, dubbed Deep Learning (DL), has shown great promise by achieving or even surpassing human-level capabilities in many tasks. The rise of DL has brought forth a new industrial revolution by taking over the modern landscape of smart applications, e.g., self-driving cars, virtual assistants, drug discovery, and manufacturing. Nevertheless, to date, there exist quite a few challenges for wide-scale adoption of DL in real-life scenarios.

Firstly, confidence characterization and ensuring robustness of DL-enabled services is imperative, particularly in safety-critical autonomous systems. Secondly, concerns over the scalability and efficiency of DL hinder its training and deployment on diverse hardware platforms. This dissertation addresses the above-mentioned challenges via a holistic customization of DL algorithm and system from the standpoint of task-based metrics (e.g., accuracy), physical constraints (e.g., memory and power budget), as well as new design metrics that facilitate DL integration in safety-sensitive tasks.

The presented research in this dissertation interlinks theoretical fundamentals, domain-specific architecture design, and automated tools that enable co-optimization of the DL algorithm with the underlying platform while satisfying various constraints. The key contributions of this dissertation are as follows:

- Devising CuRTAIL, the first end-to-end and automated framework that simultaneously enables efficient and safe execution of DL models in face of adversarial attacks. CuRTAIL formalizes the goal of thwarting adversarial attacks as an optimization problem and trains parallel defense modules to minimize vulnerability. The proposed framework leverages hardware/algorithm co-design and customized acceleration to enable just-in-time execution in resource-constrained settings.

- Designing a novel framework, dubbed ACCHASHTAG, which identifies any faults occurring during DL inference in real-time. I propose to summarize the ground-truth DL model as a unique hash signature, which is used to verify the model's integrity on the fly. Notably, ACCHASHTAG, for the first time, provides guaranteed lower bounds on the detection rate using a formal statistical analysis of hash collision.

- Proposing CLEANN, the first end-to-end framework that enables online mitigation of backdoor, a.k.a. Trojan, attacks on DL. CLEANN uses sparse recovery and statistical analysis to identify incoming Trojan samples and remove their effect on the victim model's

prediction. I design the algorithmic solutions as well as customized hardware-accelerated engines to enable real-time DL model decision verification via CLEANN.

- Innovating an approach for restructuring inter-layer connections in DL models, leading to faster convergence to a desired accuracy during training. This is achieved by transforming the DL model into a small-world network using principles from graph theory. The obtained DL model, dubbed SWANN, is a highly-connected, small-world topology with enhanced signal propagation characteristics and faster learning speed.

- Developing LTS, the first training-free, hardware-aware neural architecture search for autoregressive Transformers. The proposed method delivers high-performance specialized architectures for inference on a target hardware. The core of LTS is an ultra-low-cost proxy that can estimate the performance of candidate architectures without any need for training. Using this novel proxy, the search can be performed entirely on the target hardware, allowing us to incorporate hardware measurements, e.g., peak memory utilization and latency, within the architecture search loop.

- Automating DL model customization for various target hardware by formulating it as a constrained optimization. The optimization goal is to compress a large model to satisfy given accuracy and hardware performance constraints. I propose a highly-scalable black-box optimizer, dubbed *AdaNS*, to solve the aforesaid optimization problem. *AdaNS* leverages adaptive non-uniform sampling with carefully crafted probabilistic distributions to locate and reconstruct the optimization objective function around its maximizers.

Chapter 1

Introduction

Recent advances in Deep Learning (DL) have revolutionized modern applications by automating decision-making and outsourcing tasks to computerized agents. This has opened up new possibilities and opportunities in areas such as speech recognition, computer vision, natural language processing (NLP), and autonomous systems. Figure 1.1 demonstrates a Venn diagram of the critical design objectives for DL-enabled autonomous systems. Among these objectives, high evaluation performance on task-specific metrics such as accuracy has gained tremendous attention from the research community. Several research efforts aim to advance state-of-the-art evaluation performance, thus increasing the opportunities for incorporating automated DL-powered agents in various domains. Nevertheless, task-specific evaluation performance is insufficient to fuel the wide-scale adoption of DL in real-life applications.

Concerns are rising over the robustness of DL-enabled services, e.g., self-driving vehicles, particularly in applications that are tightly coupled with clients' safety. Thus, scalable and effective model-assurance techniques are required to ensure the integrity of autonomous systems. Another critical design objective for DL is the scalability and physical performance on various platforms. Addressing this objective is specifically challenging due to the increasing complexity of DL algorithms and the excessive variations in the tasks where DL is deployed. As such, devising automated DL customization techniques that can adapt to the existing diversity in the underlying applications, hardware platforms, and pertinent data is of utmost importance.

My research provides end-to-end frameworks that customize DL models to simultaneously optimize robustness, hardware performance, and task evaluation metrics, e.g., accuracy. Towards this goal, I develop automated tools that enable a holistic co-optimization of the DL algorithm and underlying system in various domains. Such a holistic approach is particularly beneficial as it integrates various necessary criteria for a ready-to-use intelligent system into the design process. This chapter summarizes the research contributions made by the author.

Figure 1.1. Critical design objectives for autonomous DL-enabled systems. My research provides holistic solutions that co-optimize DL algorithm and hardware, thus jointly satisfying multiple design objectives, i.e., task evaluation metrics, robustness, and hardware performance.

1.1 Trustworthy and Robust Deep Learning

While DL models have demonstrated superior capabilities for automated representation learning, a line of research has revealed their vulnerability to various attacks. These attacks target various life cycles of DL, i.e., training and inference, and pose a critical safety threat to DL; they can divert the behavior of the model, leading to incorrect decisions in sensitive tasks. My research proposes novel defenses that protect DL models against various security threats. Particularly, I propose end-to-end defense schemes that are developed using algorithm/hardware co-design, as outlined below. The proposed defenses enable efficient real-time DL decision verification for real-world applications.

1.1.1 Robustness to Runtime Attacks

The research in this dissertation provides defenses against two classes of runtime attacks on DL models, namely, adversarial attacks and hardware-induced faults. Adversarial samples are generated by adding carefully crafted perturbations to the input data at inference time. These perturbations, often human-imperceptible, misguide the DL model to output the wrong prediction at inference. Hardware-based fault-injection attacks take advantage of a vulnerability in the underlying platform to alter the execution flow of DL models, e.g., by altering the weights. This can cause the model to behave in unexpected ways and generate incorrect predictions.

Mitigating Adversarial Attacks. I propose CuRTAIL, the pioneering work in codesign of algorithm and hardware for defense against adversarial attacks. The proposed defense relies on the critical observation that benign and adversarial samples behave differently in the feature space of Deep Neural Networks (DNNs). Building upon this insight, CuRTAIL trains a set of Modular Robust Redundancies (MRRs) to learn the probability density functions (PDF) of the legitimate samples in each hidden layer. The MRRs are leveraged to checkpoint the DNN during inference and raise alarm flags for out-of-distribution adversarial data. CuRTAIL's unsupervised statistical modeling allows it to withstand new unseen adversarial attacks.

A key distinguishing feature of CuRTAIL compared to prior defenses is the specialized hardware modules, devised to accelerate the MRRs and enable detection in real-time. CuRTAIL is further accompanied by an automated tool that customizes the defense configuration to maximize robustness at a user-defined latency constraint while adhering to the underlying platform constraints on memory and compute resources. Specifically, CuRTAIL optimizes the trade-off between robustness and performance by determining the best number of MRRs along with the underlying design hyperparameters for their FPGA-based compute kernels. Extensive evaluations on various hardware platforms corroborate the robustness and efficiency of CuRTAIL against all attacks known to date including the most challenging attack scenario in the real world, i.e., the white-box attack.

Mitigating Fault-injection Attacks. I propose ACCHASHTAG, a holistic framework for high-accuracy detection of fault-injection attacks on DNNs with provable bounds on detection performance. Recent literature on fault-injection attacks shows the severe DNN accuracy degradation caused by bit flips. In this scenario, the attacker changes a few DNN weight bits during execution by injecting faults into the dynamic random-access memory (DRAM). To detect bit flips, ACCHASHTAG extracts a unique signature from the benign DNN prior to deployment. At runtime, new hashes are extracted from the DNN and compared against the ground-truth signatures to validate the model's integrity and verify the inference output on the fly. Notably, ACCHASHTAG, for the first time, provides guaranteed lower bounds on the detection rate using formal statistical analysis of hash collision in addition to delivering 0% false positive rate.

ACCHASHTAG balances the inherent trade-off between detection accuracy and defense overhead by minimizing the number of DNN layers used for hash extraction. Specifically, due to the high cost of fault-injection attacks, attackers are incentivized to target vulnerable layers to maximize the accuracy reduction with the minimum number of bit alterations. I propose a novel sensitivity analysis that quantifies the vulnerability of DNN layers to bit-level faults. By identifying and curating the hash generation to the most vulnerable layers, ACCHASHTAG enjoys a high detection rate while minimizing defense overhead. The proposed hash-based signature generation is easily adaptable to any given DNN topology and requires only lightweight bit-level operations that can be accommodated on various platforms with negligible overhead. I further devise a specialized compute core for ACCHASHTAG on field-programmable gate arrays (FPGAs) to facilitate online hash generation in parallel to DNN execution. Extensive evaluations using the state-of-the-art bit-flip attack on various DNNs demonstrate the competitive advantage of ACCHASHTAG defense in terms of both attack detection and execution overhead.

1.1.2 Robustness to Training Time Attacks

Model poisoning via backdoors, a.k.a Trojans, is a training time attack that compromises the integrity and reliability of DL models. In this scenario, the attacker appends a Trojan

trigger to a subset of the training dataset and re-labels them into their desired target class(es). By training on the poisoned dataset, a backdoor is inserted into the model. This backdoor can be maliciously activated during inference by adding the Trojan trigger to the input data, thus making the model predict the attacker's target class. To mitigate this threat, I propose CLEANN, the first end-to-end accelerated framework for online detection of Neural Trojans in resource-constrained embedded applications. The proposed lightweight Trojan defense is devised based on algorithm/hardware co-design; the algorithmic insights offer a highly accurate and low-overhead Trojan detection method; the accompanying specialized hardware accelerator enables low-latency and energy-efficient defense execution on embedded hardware.

What differentiates CLEANN from the prior work is its lightweight methodology which recovers the ground-truth class of Trojan samples without the need for labeled data, model retraining, or prior assumptions on the trigger or the attack. The proposed unsupervised defense leverages key concepts from sparse approximation [37] and dictionary learning to characterize the statistical behavior of benign data and identify outliers, i.e., Trojan triggers. Proof of concept evaluations against the state-of-the-art Neural Trojan attacks on visual benchmarks demonstrate CLEANN's competitive advantage in terms of attack resiliency and execution overhead.

1.2 Platform-aware Deep Learning on Massive Data

Modern AI algorithms and in particular, DNNs, comprise millions of parameters that render both the training and execution phases quite costly in terms of time, energy, and required hardware resources. Without addressing the aforementioned cost issue, DNNs remain nonviable in many real-world applications. In the following, I elaborate on aspects of my research that target efficient training and execution for DL models.

1.2.1 Efficient Training

A standing challenge for the ubiquitous adoption of machine intelligence in various tasks is the excessively high cost of training accurate DL models. These models require a large number

of computationally intensive training iterations to reach a high convergence accuracy. While the design of high-accuracy DNNs has received wide attention from the DL research community, a less-studied question yet remains: can the pertinent model architecture be modified to enhance training speed and convergence, while simultaneously achieving state-of-the-art accuracy? To answer this question, I propose a novel transformation that changes the topology of the DNN to reach an optimal cross-layer connectivity. This, in turn, significantly reduces the number of training iterations required for reaching a target accuracy.

The proposed transformation leverages the important observation that for a set level of accuracy, convergence is fastest when network topology reaches the boundary of a Small-World Network. Small-world graphs are known to possess a specific connectivity structure that enables enhanced signal propagation among nodes. The derived small-world DNNs, called SWANNs, provide several intriguing benefits: they facilitate data (gradient) flow within the network, enable feature-map reuse by adding long-range connections and accommodate various network architectures and datasets. The devised method for transforming a DNN to SWANN is fully automated, incurs negligible overhead, and does not affect the pertinent model's final accuracy. Compared to densely connected networks (e.g., DenseNets [77]), SWANNs require a substantially fewer number of training parameters while maintaining a similar level of classification accuracy. Proof-of-concept experiments on various architectures and image classification benchmarks demonstrate an average of $\approx 2.1\times$ improvement in the convergence speed of SWANNs to a desired accuracy, compared to the original DNNs.

1.2.2 Resource-Customized Inference

With the emerging swarm of intelligent applications, the number of possible variations of data, application, and platform has grown exponentially. As a result of this continuous growth, a one-size-fits-all solution for DL architectures fails to address the performance needs that are specific to each usage scenario. In particular, the capability to locally learn and infer data is undoubtedly critical. To make this possible, my research enables the optimized execution of

DNNs on various platforms. I have approached the problem from two different angles, namely, Neural Architecture Search and post-training model customization as detailed below. The former approach designs DL architectures, from scratch, that are particularly optimized for inference on target hardware. The latter approach customizes existing, pre-trained, models to enhance their performance on the underlying platform.

Neural Architecture Search. In the contemporary big data realm, DNNs are evolving towards more complex architectures to achieve higher inference accuracy, particularly in the NLP domain. The Transformer architecture is ubiquitously used as the building block of large-scale autoregressive language models. However, finding architectures with the optimal trade-off between task performance (perplexity) and hardware constraints like peak memory utilization and latency is non-trivial. This is exacerbated by the proliferation of various hardware. I leverage the somewhat surprising empirical observation that the number of decoder parameters in autoregressive Transformers has a high rank correlation with task performance, irrespective of the architecture topology. This observation organically induces a simple Neural Architecture Search (NAS) algorithm that uses decoder parameters as a proxy for perplexity without need for any model training.

The search phase of my training-free algorithm, dubbed Lightweight Transformer Search (LTS), can be run directly on target devices since it does not require GPUs. Using on-target-device measurements, LTS extracts the Pareto-frontier of perplexity versus any hardware performance cost. LTS is evaluated on diverse devices from ARM CPUs to NVIDIA GPUs and two popular autoregressive Transformer backbones: GPT-2 and Transformer-XL. Results show that the perplexity of 16-layer GPT-2 and Transformer-XL can be achieved with up to $1.5\times, 2.5\times$ faster runtime and $1.2\times, 2.0\times$ lower peak memory utilization. When evaluated in zero and one-shot settings, LTS Pareto-frontier models achieve higher average accuracy compared to the 350M parameter OPT across 14 tasks, with up to $1.6\times$ lower latency. Notably, LTS extracts the Pareto-frontier in under 3 hours while running on a commodity laptop. My proposed method effectively removes the carbon footprint of hundreds of GPU hours of training during search, offering a

strong simple baseline for future NAS methods in autoregressive language modeling.

Post-training Model Customization. Various compression techniques can be leveraged to efficiently deploy pre-trained, compute-intensive DNNs on resource-limited devices. Such methods comprise various hyperparameters that require per-layer customization to ensure high accuracy. Choosing the hyperparameters is cumbersome as the pertinent search space grows exponentially with model layers. I propose *AdaNS*, a novel optimization algorithm that automatically learns how to tune per-layer compression hyperparameters. The optimization objective is to locate an *optimal* hyperparameter configuration that leads to the lowest model complexity while satisfying a desired constraint on inference accuracy. I design a score function that evaluates the aforementioned *optimality*. The optimization problem is then formulated as searching for the maximizers of this score function.

I devise a non-uniform adaptive sampler that aims at reconstructing the band-limited score function, particularly around its maximizers. The total number of required objective function evaluations is reduced by realizing three targeted, adaptive sampling methodologies, i.e., *AdaNS-Zoom*, *AdaNS-Genetic*, and *AdaNS-Gaussian*, where new batches of samples are chosen based on the history of previous evaluations. *AdaNS* starts sampling from a uniform distribution over the entire search-space and iteratively adapts the sampling distribution to achieve the highest density around the function maxima. This, in turn, allows for a low-error reconstruction of the objective function around its maximizers. Extensive evaluations corroborate *AdaNS* effectiveness by outperforming existing rule-based and Reinforcement Learning methods in terms of DNN compression rate and/or inference accuracy.

1.3 Acknowledgements

This chapter is, in part, a reprint of the published material in 1) M. Javaheripi, M. Samragh, B. Rouhani, T. Javidi, and F. Koushanfar, “Curtail: Characterizing and Thwarting Adversarial Deep Learning”, in *IEEE Transactions on Dependable and Secure Computing*

(TDSC), 2020, 2) M. Javaheripi, J. Chang, and F. Koushanfar, “AccHashtag: Accelerated Hashing for Detecting Fault-Injection Attacks on Embedded Neural Networks”, in *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 2022, 3) M. Javaheripi, M. Samragh, G. Fields, T. Javidi, and F. Koushanfar, “CleaNN: Accelerated Trojan Shield for Embedded Neural Networks”, in *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, 4) M. Javaheripi, B. Rouhani, and F. Koushanfar, “SWANN: Small-World Architecture for Fast Convergence of Neural Networks”, in *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2021, 5) M. Javaheripi, G. de Rosa, S. Mukherjee, S. Shah, T. Religa, C. Mendes, S. Bubeck, F. Koushanfar, and D. Dey, “LiteTransformerSearch: Training-free Neural Architecture Search for Efficient Language Models”, in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022, and 6) AdaNS: Adaptive Non-uniform Sampling for Automated Design of Compact DNNs”, in *IEEE Journal of Selected Topics in Signal Processing (JSTSP)*, 2020, and 7) M. Javaheripi, M. Samragh, T. Javidi, and F. Koushanfar, “GeneCAI: Genetic Evolution for Acquiring Compact AI”, in *Genetic and Evolutionary Computation Conference*, 2020. The dissertation author was the (co)primary investigator and author of these papers.

Chapter 2

Ensuring DL Robustness to Adversarial Attacks

Security and safety consideration is a major obstacle to the wide-scale adoption of emerging learning algorithms in sensitive scenarios, such as intelligent transportation, healthcare, and video surveillance applications [31, 131]. While advanced learning techniques are essential for enabling coordination between autonomous agents and the environment, a careful analysis of their vulnerabilities and their reliability in face of adversarial attacks is still in its infancy.

Adversarial samples [22, 142, 148] are carefully crafted input instances which lead machine learning algorithms into misclassifying while the input changes are imperceptible to the human eye. For instance, in the case of traffic sign classifiers employed in self-driving cars, an adversary can add a specific imperceptible perturbation to a legitimate "stop" sign sample and fool the DL model to classify it as a "yield" sign, thus, jeopardizing the safety of the vehicle as shown in [131]. Thereby, it is highly important to identify and reject risky samples to ensure the integrity of DL models used in autonomous systems such as unmanned vehicles/drones.

This work provides an end-to-end solution (called CuRTAIL) to characterize and thwart adversarial attacks for DL models in an online manner. Our proposed solution addresses three main challenges regarding the adversarial attacks in the context of deep learning.

(i) Understanding the root cause of DL vulnerabilities to adversarial samples. Our hypothesis is that the vulnerability of Deep Neural Networks (DNNs) to adversarial samples originates

from the existence of rarely explored sub-spaces in each feature map. This phenomenon is particularly caused by the limited access to labeled data and/or inefficiency of regularization algorithms [34, 199]. Figure 2.1 provides a simple illustration of the partially explored space in a two-dimensional setup. We provide statistical analysis and empirically back up our hypothesis by extensive evaluations on various DL benchmarks and attacks.

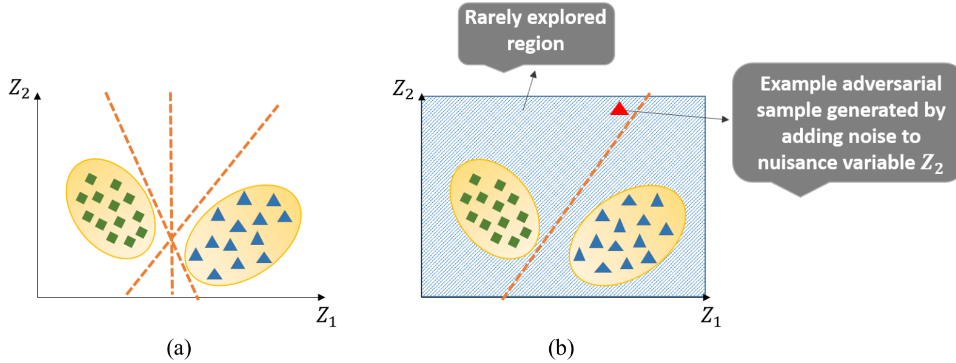

Figure 2.1. (a) Data points (green and blue squares) can be easily separated in one-dimensional space. Extra dimensions add ambiguity in choosing the decision boundaries: all shown boundaries (dashed lines) result in the same classification accuracy but are not equally robust to noise. (b) The rarely explored space leaves room for adversaries to manipulate the non-critical dimensions (Z_2 in this figure) and mislead the model by crossing the decision boundaries.

(ii) Characterizing and thwarting the adversarial subspace for model assurance. A line of research has shown that there is a trade-off between the robustness of a model and its accuracy [129, 149]. Taking this into account, instead of making a single model that is both robust and accurate, we introduce a new defense mechanism called Modular Robust Redundancy (MRR). In MRR methodology, the victim model is kept *as is* while separate defender modules are trained to checkpoint the hidden features and assess the reliability of the victim's prediction. Each defender module characterizes the explored sub-space in the pertinent layer by learning the probability density function (PDF) of legitimate data points and marking the complement sub-spaces as rarely observed regions. Once such characterization is obtained, MRRs evaluate the input sample in parallel with the victim model and raise alarm flags for data points that lie within the rarely explored regions.

(iii) Just-in-time online defense against adversarial attacks. We propose CuRTAIL, the first

end-to-end hardware-accelerated framework that enables robust and just-in-time defense against adversarial attacks on DNNs. CuRTAIL is devised based on algorithm/hardware co-design to enable safe DL execution while customizing system performance in terms of latency, energy consumption, and/or memory footprint based on the available resource. CuRTAIL leverages field-programmable gate arrays (FPGAs) to provide fine-grained parallelism and just-in-time response by our defender modules. The customized data path for memory access on FPGA improves the system energy efficiency.

We study various state-of-the-art attack algorithms and threat models and validate the robustness of our proposed approach for different DL benchmarks including MNIST, SVHN, CIFAR, and a subset of ImageNet data. The explicit contributions of this work are as follows:

- Proposing CuRTAIL, the first algorithm/hardware co-design enabling online defense against adversarial samples for DL models. Our methodology is unsupervised and robust against the most challenging attack scenario to date.

- Introducing Modular Robust Redundancy as a viable countermeasure for adversarial attacks on DL. CuRTAIL uses dictionary learning and probability density functions to statistically detect abnormalities in the input data.

- Providing quantitative metrics to characterize the sensitivity of DL layers from a statistical point of view. Based on the result of our analysis, we provide new insights on the reason behind the existence of adversarial transferability.

- Implementing the first streaming-based DL defense using FPGAs. We design an automated customization tool to adaptively maximize model robustness against adversaries while complying with the underlying hardware resource constraints, i.e., runtime, energy, and memory.

- Performing extensive evaluations in both black-box and adaptive white-box settings against various attack methodologies including Fast-Gradient-Sign [56], Jacobian Saliency Map attack [148], Deepfool [142], Basic Iterative Method [104], and Carlini&WagnerL2 [21, 22].

Thorough performance comparison on various hardware platforms including CPUs, GPUs, and FPGAs corroborates CuRTAIL's algorithmic practicality and system efficiency.

2.1 Background and Preliminaries

Adversarial machine learning can be cast as a zero-sum Stackelberg game between the machine learning oracle (victim) and the attacker. Let us denote a DL model by $f(x, \theta)$, where x and θ represent the input and model parameters, respectively. In an adversarial setting, the attacker aims to find a perturbed adversarial sample (x^a) such that it incurs minimal distance from the source sample (x^s) while its corresponding output is sufficiently different to mislead the victim. Figure 2.2 illustrates an example, where the image on the left is initially classified correctly as a dog by the victim model while adding a small amount of perturbation to the original image has misled the victim to infer it as a black swan (right image). Clearly, if the source instance is already misclassified by the victim model ($f(x^s, \theta) \neq y^*$), the adversarial problem becomes trivial. Therefore, we particularly focus on instances x^s that could have been classified correctly by the oracle before adding structured adversarial noises ($f(x^s, \theta) = y^*$).

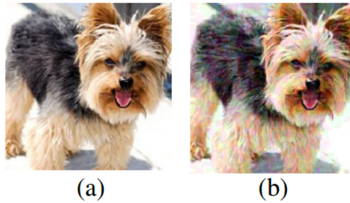

Figure 2.2. An example of (a) input data, and (b) its corresponding adversarial sample. The added noise is imperceptible but can cause the victim model to misclassify.

Several attack mechanisms have been proposed in the literature to craft adversarial samples. We evaluate CuRTAIL performance against a variety of attacks to empirically confirm the generalizability of out unsupervised MRR methodology across a wide range of attacks. In particular, we have evaluated CuRTAIL against (i) Fast Gradient Sign (FGS) [56], (ii) Jacobian Saliency Map Attack (JSMA) [148], (iii) Deepfool [142], (iv) Basic Iterative Method (BIM) [104],

and (v) Carlini&WagnerL2 adaptive attacks [21, 22] . In the following, we provide a brief explanation of the evaluated attack algorithms.

Fast-Gradient-Sign. FGS attack [56] crafts an adversarial sample as $x' = x + \epsilon \cdot Sign(\frac{\partial C}{\partial x})$, where C is the cost function of the neural network, and $Sign(\cdot)$ outputs the sign of its operand. The attack is parameterized by ϵ, which determines the amount of additive perturbation.

Basic Iterative Method. BIM [104] is an iterative version of FGS characterized by the number of iterative updates, n_{iters}, and the per-iteration perturbation coefficient, ϵ.

Deepfool. This algorithm iteratively modifies the input image based on a specific update rule to obtain an adversarial sample [142]. In each iteration, the perturbation vector $\frac{\partial C}{\partial x}$ is normalized and added to the sample. Deepfool is parameterized by the number of iterative updates n_{iters}.

Carlini&WagnerL2[1]. This attack is formalized as a minimization problem where the objective is the L_2 norm of the perturbation vector. The adversary performs an iterative method for solving this minimization objective. The detailed set of parameters for the attack is provided in [22].

Depending on the attacker's knowledge, three different attack models are feasible:

- **White-box attacks.** The attacker knows everything about the victim model including the learning algorithm, model topology and parameters, as well as the defense mechanism, the corresponding defender parameters.
- **Gray-box attacks.** The attacker knows the learning algorithm, model topology, and defense mechanism and has no access to the model/defender parameters.
- **Black-box attacks.** The attacker does not know anything about the pertinent machine learning algorithm, ML model, or defense mechanism. This attacker can merely obtain the outputs of the victim ML model by providing input samples. In this setting, the adversary can perform a differential attack by observing the output changes with respect to input variations.

For a complete taxonomy of adversarial capabilities and goals refer to [11, 80, 148].

[1] For brevity we denote this attack as CarliniL2 in the remainder of the text.

2.2 Related Work

The threat of adversarial samples to the integrity of autonomous systems have been shown in the literature for both shallow and deep models [7,13,14,47,56,80,148,184]. Related work ties the existence of adversarial samples to several factors, including high feature dimensionality [56], bias to texture [53], and inherent brittle features [85]. DL models trained on ImageNet are shown to have bias towards texture rather than object shapes, resulting in low robustness against distortions [53]. This can be tied directly with CuRTAIL hypothesis in the existence of rarely explored regions in trained models. To address this, authors of [53] propose augmenting the training data with strong textual cues, thus forcing the model to focus on shapes rather than textures. Such augmentations effectively reduce the rarely explored regions and increase robustness. Authors of [85] show that seemingly unrelated features, generally imperceptible to humans, can inherently exist in visual datasets. Surprisingly, such brittle features contribute significantly to the underlying model's generalization capability during training. Authors also connect adversarial samples to these brittle features. Building upon this connection, CuRTAIL input defenders aim at identifying such redundant feature at test time.

In response to the various adversarial attacks proposed in the literature [22, 56, 142, 148], several research attempts have been made to design DL strategies that are more robust in the face of adversarial examples. Existing countermeasures can be classified into two categories:
(i) Supervised strategies which incorporate noisy inputs as training samples [57, 95] and/or inject adversarial examples into the training phase [56, 81, 171, 184]. Such defenses are tailored for specific perturbation patterns and can only partially evade adversarial samples generated by other attack scenarios [57].
(ii) Unsupervised approaches which aim to smoothen the underlying gradient space (decision boundaries) by incorporating a regularization term in the loss function [22, 139] or compressing the neural network [149]. These works are mainly oblivious to the pertinent data density as they implicitly assume that adversarial samples originate from the piece-wise linear behavior

of decision boundaries. As such, their integrity can be jeopardized by crafting data points with specific perturbations that pass the smoothened decision boundaries [19]. [135] proposes manifold projection via auto-encoders to reform adversarial samples, which can be evaded by adaptive gray-box attacks as shown in [21].

The above works generally aim at correcting the decision of the victim network in face of adversaries. Alternatively, a line of research (including CuRTAIL) focuses on detection of adversarial samples without decision correction. They speculate that adversarial samples are not drawn from the same distribution as legitimate data. [128] proposes using Local Intrinsic Dimensionality (LID) to characterize properties of adversarial examples. However, LID is not able to detect high confidence adversarial examples [20]. [108] measures the uncertainty in the victim model's predictions using Mahalanobis distance-based scores. [160] attempts to detect adversaries by measuring the changes in the victim model's logits as the input is randomly perturbed. Nevertheless, all aforementioned methods study the statistics of the victim model's features which cannot optimally identify the rarely explored regions. This is because the victim model's main task and objective is accurate classification of *explored* regions. CuRTAIL alternatively focuses on altering the training objective of the MRRs to enable high-margin robust classification, which is later used for PDF estimation.

CuRTAIL unsupervised defense does not assume any particular attack strategy and/or perturbation pattern and is capable of withstanding all the existing attacks to date. Note that we target detection of adversarial samples without decision correction. As such, a stand-alone application of CuRTAIL remains vulnerable to denial of service attacks.

2.3 Statistical Analysis of Adversarial Samples

Let us denote the output of the ith layer of a DL model given an input sample x by $f_i(x)$. One can construct a probabilistic density function $P_X(f_i(x))$ for each layer where X is a random variable drawn from the model input space. Our conjecture is that adversarial samples cannot lie in high-probability regions of the PDF function $P_X(.)$, which is learned using the samples drawn

from legitimate training data. More formally, the expected value of the probability corresponding to legitimate samples is higher than that of adversarial samples:

$$E(P_X(f_i(x^s))) >> E(P_X(f_i(x^a))) \tag{2.1}$$

where $E(\cdot)$ is the expectation and x^s and x^a are safe and adversarial input samples, respectively.

This difference between the expected values of the probability distribution can be leveraged to characterize and thwart adversarial attacks. To do so, CuRTAIL approximates the PDF of each layer by training a set of defender modules, as will be explained in Section 2.4. Figure 2.3 empirically validates the criterion outlined in Equation (2.1). In this example, the PDF of benign samples in the second-to-last layer ($P_X(f_{N-1}(x))$, where N is the number of model layers) of the *LeNet* model is obtained by passing through legitimate MNIST data points and acquiring the corresponding activations. Once the PDF is learned, we generate adversarial samples x^a and compute $P_X(f_{N-1}(x^a))$. Figure 2.3 depicts the probabilistic histogram of legitimate and adversarial samples based on the learned PDF. As shown, the expected value of legitimate samples is orders of magnitude greater than that of adversarial samples.

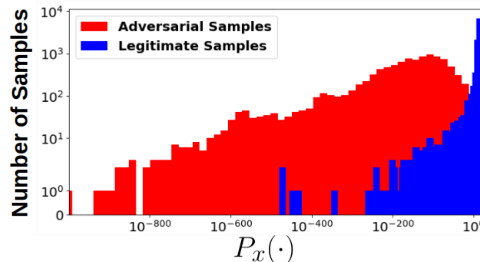

Figure 2.3. Histogram of the estimated PDF for adversarial (red) and legitimate (blue) samples. Adversarial samples are generated using Deepfool for *Lenet* architecture. The PDF is learned in the second-to-last layer of the network.

2.4 CuRTAIL Methodology

Figure 2.4 demonstrates the high-level block diagram of MRR methodology. In our proposed countermeasure, a number of modular redundancies (checkpoints) are trained to

characterize the data density distribution in the space spanned by the victim model. The defender modules are then used in parallel to checkpoint the reliability of the ultimate prediction and raise an alarm flag for risky samples. We refer to MRR modules that checkpoint the intermediate DL layers as "**latent defenders**". Whereas, the redundancy modules operating on the input space are referred to as the "**input defenders**". We use the term *checkpoints* and *modular redundancies* interchangeably throughout the text.

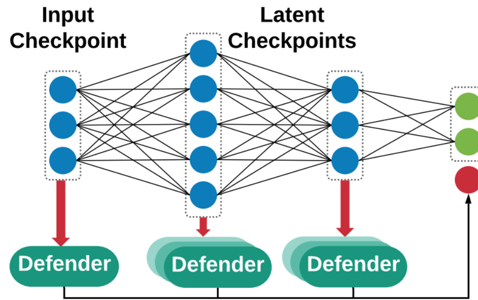

Figure 2.4. High-level block diagram of MRR methodology. Multiple defenders checkpoint the input and intermediate features in parallel. The output of the victim DNN (green neurons) is augmented with a confidence measure (red neuron) determining the prediction legitimacy.

2.4.1 Latent Defenders

The goal of each intermediate defender (checkpointing) module is to learn the PDF of the explored sub-spaces in a particular DL feature map. The learned density function is then used to identify the rarely observed regions. The latent defenders in CuRTAIL are DNNs with an identical topology to the victim model. The MRRs, however, are finetuned with a specific loss that condenses the latent features within each class, while enforcing separability between features of different classes. By enforcing such separability, the feature space can be well-modeled as a Gaussian mixture model (GMM)[2] representing benign data in different classes. Benign samples can then be effectively identified by measuring the distance to their corresponding class center.

Figure 2.5 shows the steps required to train the latent defenders in CuRTAIL. To train a single latent defender that checkpoints the nth layer of the victim DNN, we first copy the victim

[2]If needed, the GMM distribution can be replaced with other probabilistic priors to better accommodate the data geometry in new applications.

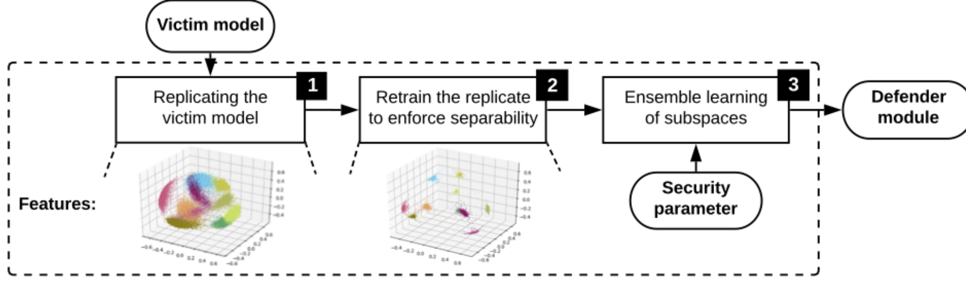

Figure 2.5. Block diagram of the training procedure for devising parallel redundancy modules. Each latent defender is built by minimizing the entanglement of intermediate data features in a Euclidean space at a particular checkpoint location. This goal is achieved through several rounds of iterative realignment of data abstractions. The latent data space is then characterized as an ensemble of lower dimensional sub-spaces to effectively learn the PDF of explored regions and detect atypical samples based on a user-defined security parameter.

model, including all parameters. We then fine-tune the replicated neural network to disentangle data features at the checkpoint location by adding the following loss, weighted by parameter γ, to the conventional cross entropy loss:

$$\gamma \; [\; \underbrace{\|C^{y^*} - f(x)\|^2}_{loss_1} \; - \; \underbrace{\Sigma_{i \neq y^*} \|C^i - f(x)\|^2}_{loss_2} \; + \; \underbrace{\Sigma_i (\|C^i\| - 1)^2}_{loss_3} \;] \tag{2.2}$$

where $f(x)$ is the L_2 normalized feature vector of input sample x at the checkpoint location, i.e., $\|f(x)\|_2 = 1$. Here, C^i is the center value for all benign normalized feature vectors $f(x)$ from class i and y^* is the ground-truth label for sample x. Both the center values C^i and intermediate feature vectors $f(x)$ are learned by fine-tuning the defender neural network.

Figure 2.6 illustrates the optimization goal of each defender module per Equation (2.2). The first term ($loss_1$) in Equation (2.2) condenses feature vectors that belong to the same class, to achieve a high concentration Gaussian distribution per class. The second term ($loss_2$) increases the intra-class distance between different centers, thus reducing the overlap between neighboring class distributions. Minimizing only the first two terms in Equation (2.2) can lead to all centers C^i being pushed to $\pm\infty$. Therefore, we add the third loss term $loss_3$ to ensure that the centers lie on a unit d-dimensional hyper-sphere and avoid divergence when training the latent defenders.

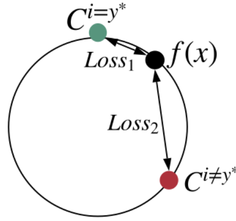

Figure 2.6. Defender module optimization objective.

Figures 2.7a and 2.7b demonstrate the distance of legitimate (blue) and adversarial (red) samples from the corresponding centers C^i in a checkpoint module before and after retraining. The centers C^i before fine-tuning the checkpoint (defender) module are equivalent to the mean of the data points in each class. As shown, fine-tuning the defender module with proposed objective function can effectively separate the distribution of legitimate samples from malicious data points. Note that training the latent defender modules is carried out in an unsupervised setting, meaning that no adversarial sample is included in the training phase.

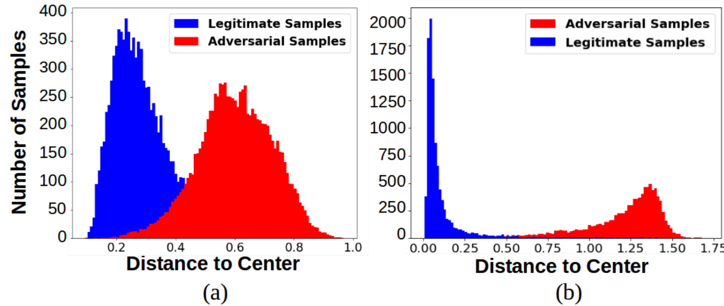

Figure 2.7. (a) Distance of legitimate (blue) and adversarial (red) samples from the corresponding centers C^i before, and (b) after realignment of data samples. In this example, we consider the *LeNet* model [106] trained on MNIST. The checkpoint is inserted in the second-to-last layer and adversarial samples are generated by FGS attack.

High dimensional real-world datasets can be represented as an ensemble of lower dimensional sub-spaces [17, 161]. As discussed in [17], under a GMM distribution assumption, data points belonging to each class can be characterized as a spherical density in two sub-spaces: (i) The sub-space where the data actually lives and (ii) its orthogonal complementary space. We use High Dimensional Discriminant Analysis (HDDA) [17] to learn the mean and conditional covariance of each class as a composition of lower dimensional sub-spaces.

The learned PDF variables (i.e., mean and conditional covariance) are used to compute the probability of a feature point $f(x)$ coming from a specific class. In particular, for each incoming test sample x, the probability $p(f(x)|y^i)$ is evaluated where y^i is the predicted class (output of the victim neural network) and $f(x)$ is the corresponding data abstraction at the checkpoint location. The acquired likelihood is then compared against a user-defined *cut-off threshold* which we refer to as the *security parameter*. The Security Parameter (SP) is a constant number in the range of $[0\% - 100\%]$ that determines the hardness of the defender decision boundaries as shown in Figure 2.8. In this example, we have depicted the latent features of one category that are projected into the first two Principal Component Analysis (PCA) components in the Euclidean space (each point corresponds to a single input image). The blue and black contours correspond to security parameters of 10% and 20%, respectively. For example, 10% of the legitimate training samples lie outside the contour specified with $SP = 10\%$.

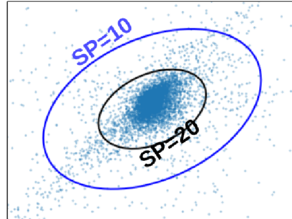

Figure 2.8. Illustration of the effect of security parameter on the detection policy. A high SP leads to a tight boundary which treats most samples as adversarial examples.

Training Multiple Latent Defenders. In what follows, we explain our methodology for creating multiple defender modules that are negatively correlated. Specifically, two MRRs are negatively correlated if deceiving one of them raises a high suspicion in the other one and vice versa. Consider the ith MRR that maps a legitimate input x to feature vector $f_i(x)$, where $f_i(x)$ is close (in terms of Euclidean distance) to the ground-truth cluster center $C_i^{y^*}$. An adversary trying to mislead this defender would generate a perturbed input $x + \eta$ such that $f_i(x + \eta)$ is far from $C_i^{y^*}$. Negative correlation means that for the subsequent MRR with feature vector $f_{i+1}(\cdot)$, adding perturbation η will bring $f_{i+1}(x + \eta)$ closer to its ground-truth cluster center $C_{i+1}^{y^*}$. Figure 2.9

shows this effect where the colored cloud represents data points in each MRR's latent feature map and the decision boundary specified by the security parameter (SP) is shown with the oval.

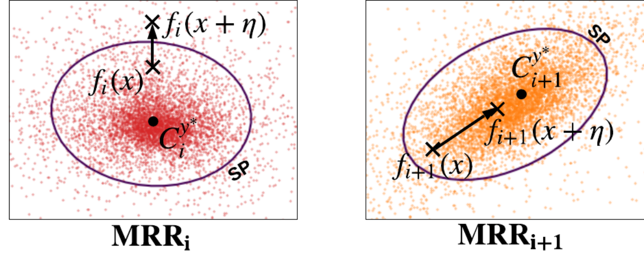

Figure 2.9. Enforcing negative correlation between MRRs.

To mitigate such adaptive attacks, we propose to train a Markov chain of detector modules as illustrated in Figure 2.10. To build this chain of MRRs, we first train a single defender module as explained earlier in this section. We then generate a new set of training data from adversarial samples $x + \eta$ of the previous defender module. The perturbation η is chosen as $\eta = \frac{\partial L1}{\partial x}$, where $L1$ is the $loss_1$ term in Equation (2.2) corresponding to the nth defender. Given these new perturbed samples, data points that deviate from the centers in the nth defender will be close to the corresponding center in the $(n+1)^{th}$ defender. As such, deceiving all the defenders requires a larger perturbation.

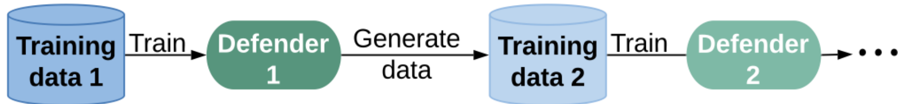

Figure 2.10. Training multiple negatively correlated defenders at each checkpoint layer.

An active adversary can find a structured noise that moves the data point from one cluster to the center of the other clusters; thus fooling the defender modules (Figure 2.11a). The risk of such an attack approach is significantly reduced in our proposed MRR countermeasure due to three main reasons: (i) Increasing intra-class distances in each checkpointing module; The latent defender modules are trained such that not only the inner-class diversity is decreased, but also the distance between each pair of different classes is increased (see Equation (2.2)). (ii) Use of parallel checkpointing modules as explained above; the attacker has to simultaneously deceive

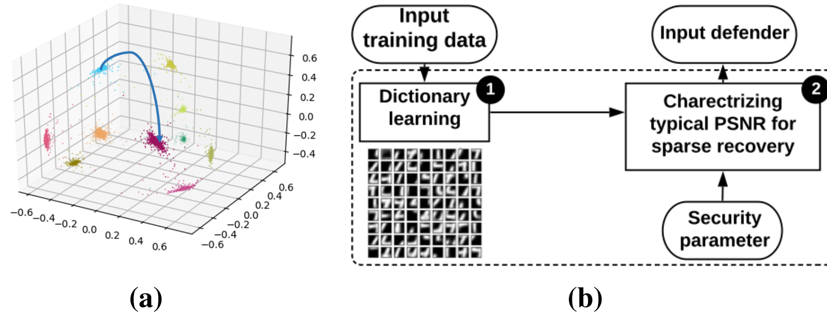

Figure 2.11. An input defender module is devised based on robust dictionary learning techniques to automatically filter out test samples that highly deviate from the typical PSNR of data points within the corresponding predicted class.

all the defender models in order to succeed. (iii) Learning a separate defender module in the input space to validate the Peak Signal-to-Noise Ratio (PSNR) level of the incoming samples as will be discussed in the next section.

2.4.2 Input Defender

We leverage dictionary learning and sparse signal recovery techniques to measure the PSNR of each incoming sample and automatically filter out atypical samples in the input space. Figure 2.11b illustrates the block diagram of an input defender module. An input checkpoint is configured in two main steps: (i) dictionary learning, and (ii) characterizing the typical PSNR per class after sparse recovery.

During the dictionary learning step, we solve the following optimization objective to learn a dictionary matrix D^i and the corresponding sparse matrix V_i such that D^iV^i best reconstructs the benign data in class i, i.e., with minimum L_2 error:

$$\underset{D^i,V^i}{argmin}\ \frac{1}{2}\|Z^i - D^iV^i\|_2 + \beta\|V^i\|_1 \qquad s.t.\ \|D^i[:,k]\| = 1, \quad 0 \leq k \leq K_{max} \tag{2.3}$$

Here K_{max} is the number of dictionary columns and Z_i is a matrix whose columns are constructed by flattening patches of the benign input images. For example, considering 8×8 patches of pixels, Z^i would have columns of length 64 elements. The Lasso problem defined in Equation (2.3) is

solved using Least Angle Regression (LAR) method [39].

Once the per-class dictionaries are learned, they can be used to reconstruct incoming input samples. Specifically, for a new input sample x with predicted class i, we leverage Orthogonal Matching Pursuit (OMP) [193] to obtain the sparse matrix V using the dictionary D^i and x as outlined in Algorithm 1. As shown, Performing OMP requires iterative execution of three main steps: (i) finding the best matching sample in the dictionary matrix D (Line 4 of Algorithm 1), (ii) least-square (LS) optimization (Line 6 of Algorithm 1), and (iii) residual update (Line 7 of Algorithm 1). In the provided pseudo code D_{col} represents the col^{th} column of the dictionary matrix D, and Λ_i is the subset of dictionary columns that have been chosen so far in the routine. OMP algorithm terminates when the number of non-zero elements in the output coefficient vector (V) is more than the user-specified sparsity level k.

A benign sample belonging to class i should be well-reconstructed as D^iV with a high PSNR value, where V is the optimal solution obtained by OMP and PSNR is defined as:

$$PSNR = 20 \times log_{10}(MAX_I) - 10 \times log_{10}(MSE) \tag{2.4}$$

where *MSE* is the L_2 difference between the input and reconstructed image using the dictionary, i.e., $\|r_k\|$ in Algorithm 1. Here, MAX_I is the maximum possible pixel value of the image (e.g., 255). We obtain a suitable threshold on the PSNR such that it satisfies the user-defined security parameter for benign data. Incoming samples with a lower PSNR than the derived threshold are marked as adversarial.

Figure 2.12 shows the effect of adversarial perturbation level ϵ on CuRTAIL detection rate for different values of the security parameter. In this experiment, we have considered the FGS attack on *LeNet* model trained on MNIST dataset. Table 2.2 summarizes the DL model topology used in each benchmark. The latent defender module (checkpoint) is inserted at the second-to-last layers. As shown, the dictionaries learned by the input defender allow us to detect high-perturbation adversarial samples (e.g., $\epsilon > 0.25$) while the latent defender effectively

Algorithm 1. OMP algorithm

Inputs: Dictionary D, input sample x, maximum sparsity level k.
Output: Coefficient vector V.

1: $r_0 \leftarrow x$
2: $\Lambda_0 \leftarrow \emptyset$ ▷ empty dictionary subset
3: **for** $i = 1, \ldots, k$ **do**
4: $\quad j = \text{argmax}_{col} | < r_{i-1}, D_{col} > |$
5: $\quad \Lambda_i \leftarrow \Lambda_{i-1} \cup D_{[:,j]}$ ▷ update dictionary subset
6: $\quad V \leftarrow \text{argmin } \|r_{i-1} - \Lambda_i \cdot V\|_2$ ▷ obtain sparse representation
7: $\quad r_i \leftarrow r_{i-1} - \Lambda_i \cdot V$ ▷ update residual error
8: **return** V

distinguishes malicious samples with very small perturbations. We extensively evaluate the impact of the security parameter on system performance for various benchmarks in Section 2.6.

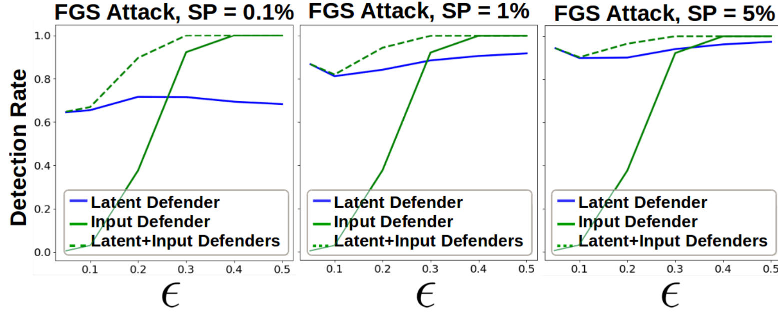

Figure 2.12. Adversarial detection rate of input and latent defenders as a function of the perturbation level for various SP. Here, FGS is used to generate adversarial samples and the perturbation is adjusted by changing attack parameter ϵ.

2.4.3 Model Fusion

Figure 2.13 depicts the configuration of the defender modules in the execution phase. The kth defender outputs a binary decision $d_k \in \{0, 1\}$ per input sample where $d_k = 1$ denotes an adversarial sample. The probability of the sample being categorized as adversarial in the final

aggregated decision is thus:

$$\begin{aligned}
&P(a=1|\{d_1,d_2,\ldots,d_n\}) = 1-\prod_{n=1}^{N}(1-P_n)^{d_n},\\
&P_k = P(a=1|d_k=1)
\end{aligned} \tag{2.5}$$

In this formulation, each MRR is parameterized by P_k which shows the probability that it can correctly categorize a sample as adversarial. We estimate P_k by evaluating the performance of each defender module on synthetic adversarial samples generated from a subset of the benign training data. In particular, for each legitimate sample x, we generate $x^a = x+\epsilon\cdot\nabla_x(\mathcal{L})$ where $\mathcal{L}$ is the victim model's cross-entropy loss. By using this generic form of adversary, we ensure that the calculated P_k is attack-agnostic and works well with different adversaries. The probability P_k is estimated as:

$$P_k = \frac{S_{True}}{S_{False}+S_{True}} \tag{2.6}$$

where S_{True} is the number of adversarial samples that are correctly detected by defender k and S_{False} denotes the number of legitimate samples that were mistaken for adversaries. In our experiments, we raise alarm flags for samples with $P(a=1|\{d_1,d_2,\ldots,d_n\}) \geq 0.5$.

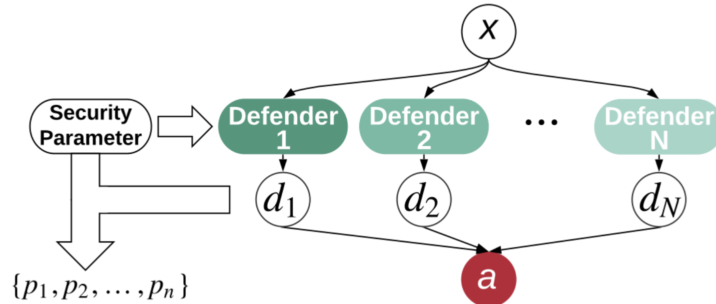

Figure 2.13. CuRTAIL uses a score-based statistical method to aggregate MRR decisions.

2.4.4 Sensitivity Analysis

The effectiveness of adversarial perturbations on DL classification can be quantified by their induced variations on intermediate activations. To study this effect, we extract the cluster centers corresponding to each hidden layer using a subset of (benign) training data. Let X^{y^*} denote samples with label y^*. The corresponding cluster center $C_l^{y^*}$ at layer l is calculated as:

$$C_l^{y^*} = \mathbb{E}_{x \sim X^{y^*}}[P_l \cdot f_l(x)] \tag{2.7}$$

where $f_l(x)$ is the layer activations. To reduce dimensionality, we perform PCA (P_l operator in Equation (2.7)) on the activation vectors such that more than 99% of the energy is preserved.

The perturbation signal in adversarial samples can be modeled as an additive noise to the input data. At each intermediate layer l, the added perturbation increases the distance between the activation vectors and their ground-truth cluster centers. Given the pre-computed center $C_l^{y^*}$ and PCA matrix P_l, the distance is measured as:

$$g_l(x) = \|P_l \cdot f_l(x) - C^{y^*}\|^2 \tag{2.8}$$

For each layer l we define the instability as:

$$\underset{r \neq 0}{Sup} \frac{g_l(x+r) - g_l(x)}{\|r\|} \tag{2.9}$$

where Sup denotes supremum and r is the input noise. For small perturbations, Taylor series can be leveraged to closely approximate the supremum by:

$$\underset{r \neq 0}{Sup} \frac{< r, \nabla_x(g_l) >}{\|r\|} \tag{2.10}$$

The upper bound in Equation (2.10) is achieved if and only if the perturbation vector r is

aligned with the gradient:

$$r = \|r\| \frac{\nabla_x(g_l)}{\|\nabla_x(g_l)\|} \tag{2.11}$$

Substituting this value in Equation (2.10), suggests that the instability of model layers is bounded by the magnitude of the gradient $\|\nabla_x(g_l)\|$. This measure allows for identification of most sensitive intermediate layers; layers with larger $\|\nabla_x(g_l)\|$ are better suited for MRR placement. We observed that the last layer shows highest sensitivity towards input perturbations. Figure 2.14 shows an example analysis for ResNet56 trained on CIFAR-100. We thus place all latent defenders at the output of the second-to-last layer in our experiments.

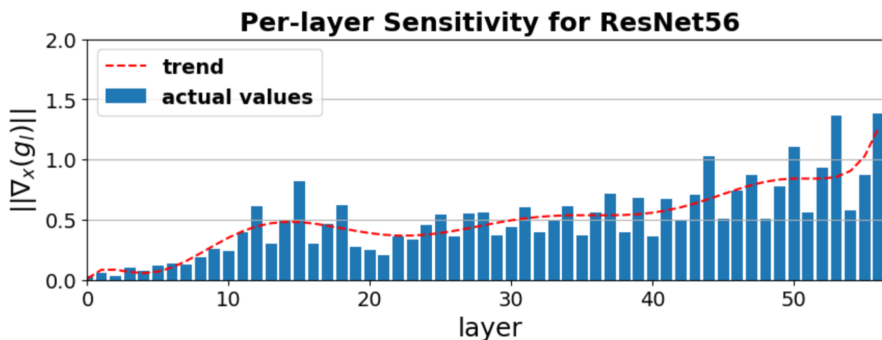

Figure 2.14. Per-layer sensitivity analysis for ResNet56

2.5 CuRTAIL Hardware Implementation

Motivation. There is an inherent trade-off between the computational complexity (e.g., runtime overhead) of the modular redundancies and the reliability of the system. On the one hand, a high number of validation checkpoints increases system reliability, but it also increases the computational load. On the other hand, a small number of checkpoints degrades the defense performance by treating adversarial samples as legitimate ones. Let us consider a naïve implementation of MRRs on commodity hardware where the checkpoints are executed *sequentially*.

Figure 2.15 demonstrates the pertinent utility and reliability trade-off under such settings

for LeNet model on MNIST dataset. Here, runtime is normalized to the cost of one forward propagation in the target neural network. As seen, the runtime in this setting increases linearly with the number of checkpoints, which is not desirable. To address this, we design an FPGA-based accelerator for optimized parallel execution of CuRTAIL MRRs. In the following, we elaborate on various components of the CuRTAIL accelerator.

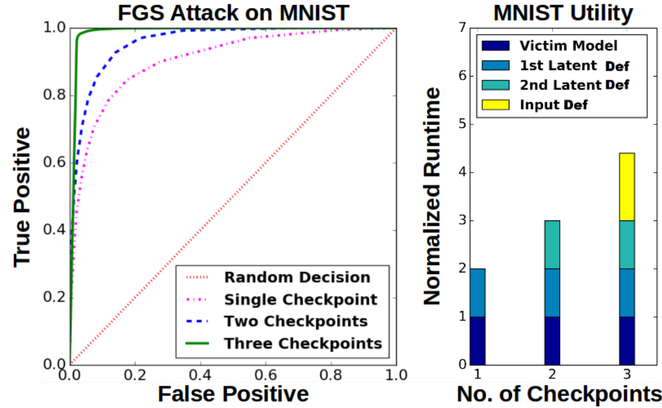

Figure 2.15. Complexity and reliability trade-off for the LeNet model on MNIST dataset performed on an NVIDIA Geforce 980 GPU hosted by an Intel Core-i7 CPU.

2.5.1 CuRTAIL Hardware Acceleration

CuRTAIL hardware acceleration stack enables just-in-time online detection of adversarial samples. Once the MRRs are trained, CuRTAIL automatically generates the hardware implementation for the modules by performing two main phases as illustrated in Figure 2.16: (i) offline pre-processing phase to obtain the MRR configurations, and (ii) online execution phase in which the legitimacy of each incoming input data is validated on the fly.

Pre-processing phase. This phase consists of two main tasks, i.e., resource profiling and design customization. During resource profiling, we estimate the FPGA resource utilization for implementing the victim DNN and the MRRs. Using the outcome of resource profiling, the design customization unit determines the best number of checkpoints according to the available resources and the user-defined performance constraints, e.g., runtime, as will be discussed in Section 2.5.2. This stage is performed only once and incurs negligible overhead.

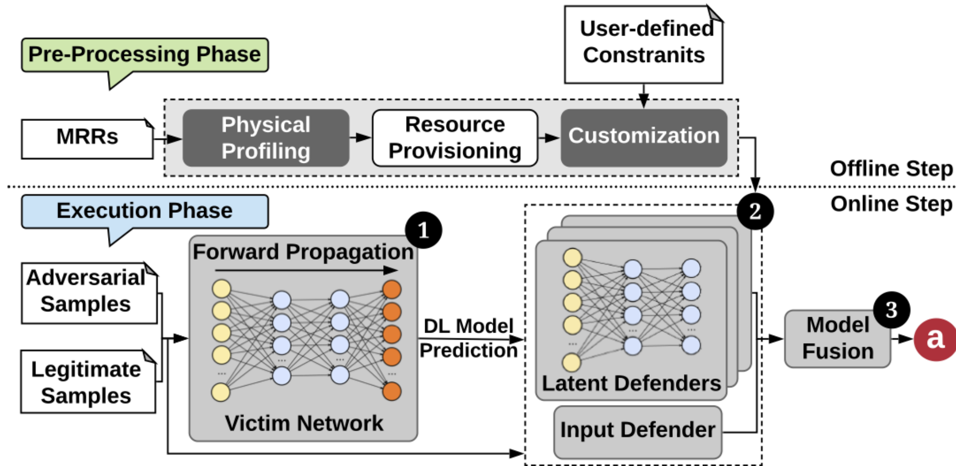

Figure 2.16. Overview of CuRTAIL hardware acceleration stack. Based on the user-provided constraints, we output the best defense layout that ensures maximum robustness and throughput.

Execution phase. Once the redundancy modules are customized per hardware and user-defined constraints, the victim DNN and accompanying MRRs are deployed for online execution. CuRTAIL performs three tasks in the execution phase.

❶ **Forward Propagation.** The victim DNN predicts a label for each incoming sample, which is then sent to the MRRs for validation.

❷ **Validation.** CuRTAIL executes the learned MRRs (Section 2.4) on FPGA to validate the legitimacy of the input data and its associated label. In particular, samples that do not lie in the *user-defined* probability interval, i.e., SP, are discarded.

❸ **Model Fusion.** The final decision regarding the legitimacy of the input data and its associated inference label is made by aggregating the output of all MRRs as explained in Section 2.4.3.

In the following, we first discuss the hardware architecture of latent and input defenders that enables high throughput and low energy realization of the recurring execution phase. We then discuss resource profiling, automated design customization, and the scalability of CuRTAIL.

Latent Defenders

Figure 2.17 illustrates the high-level schematic of a latent defender accelerator module. Recall from Section 2.4.1 that the latent defenders are DNNs with an identical architecture as the

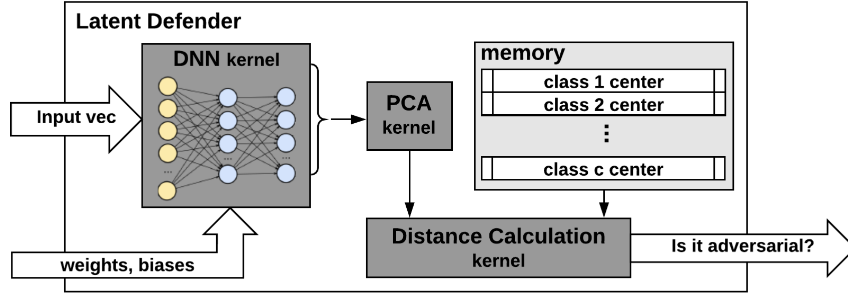

Figure 2.17. Latent defender structure: the pertinent activations are acquired by propagating the input sample through the defender. PCA is then applied to reduce the dimensionality of the obtained activation. The L_2 distance with the corresponding GMM center determines the legitimacy of the input.

victim model. Accelerating latent defenders thus requires running the DNN kernel on FPGA, which involves a high number of matrix multiplications. To efficiently execute these operations, we map them to the FPGA DSP slices that are specialized for multiplication and accumulation (MAC). Specifically, we leverage the methodology proposed in [172] to map the DNN kernel computations into multiple parallel operations. In this setting, per-layer computations are performed within several parallel-working processing units (PUs), each of which comprises a number of parallel processing elements (PEs). The parallelism can be controlled by parameters N_{PE} and N_{PU} which are static across all layers of the DNN. In order to achieve maximum throughput, it is essential to fine-tune the parallelism parameters.

There are two possible mappings between DNN layer computations and the underlying PUs. In the first scenario, multiple PUs work in parallel to compute the output of each DNN layer, where each PU computes a subset of the layer outputs. In the second scenario, each PU is responsible for computing the entire layer output, thus batches of inputs can be processed in parallel where the batch size equals the number of available PUs. CuRTAIL performs a design-space exploration to select the best execution scenario depending on the model architecture, layer dimensionality, and/or available FPGA resources. Figure 2.18 shows an example of the design space exploration for the two execution scenarios evaluated on the MNIST and SVHN benchmarks. Note that the horizontal axis in the figure, i.e., the number of PEs per PU, uniquely

determines the N_{PU} based on the available FPGA resources. As seen, determining a good balance between N_{PE} and N_{PU} is essential for minimizing the execution runtime/cycles. Specifically, increasing the number of parallel computation units will not always improve the throughput due to the underlying data dimensionality and divisibility into parallel batches.

Once the latent feature maps are computed for incoming samples, their legitimacy is determined by measuring the L_2 distance with the corresponding inference label's GMM center. To ensure our implementation readily scales to various data cardinalities, we perform Principal Component Analysis (PCA) prior to distance calculation. This operation effectively reduces the data dimensionality, thus preventing memory shortage and reducing distance computation complexity and latency. Executing PCA is equivalent to a vector-matrix multiplication $P \cdot f(x)$ where P is a matrix whose rows are eigenvectors learned from the legitimate data features $f(x)$.

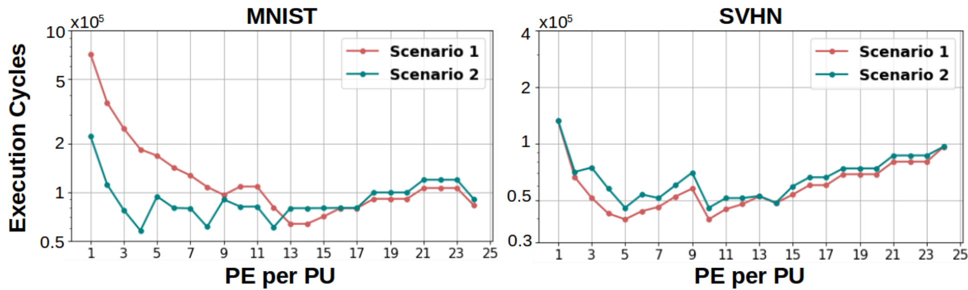

Figure 2.18. Design space exploration for MNIST and SVHN benchmarks on *Xilinx Zynq-ZC702* FPGA. CuRTAIL finds the optimal configuration of PEs and PUs to best fit the DL architecture and the available hardware resources.

Input Defender

Figure 2.19 shows a schematic of CuRTAIL's hardware kernels for the input defender. The main component in our design is an accelerator for the OMP algorithm which reconstructs the input at a given target sparsity using the pre-learned dictionaries. We boost the performance of our OMP core by modifying the algorithm to maximally utilize the available on-chip resources.

Dot-product is a frequently observed operation in OMP execution that hinders efficiency due to its sequential nature. We, therefore, implement dot-product using a *tree-based* reduction

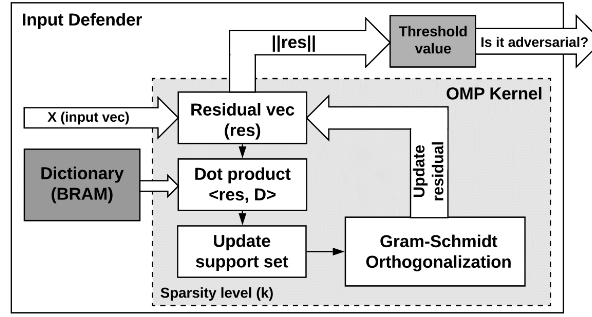

Figure 2.19. Input defender: the OMP core iteratively reconstructs input vectors using a learned dictionary. Here, the *support set* contains columns of the dictionary that have been chosen so far in the routine. The final reconstruction error is used to determine input legitimacy.

scheme which gradually aggregates the output from multiple parallel processing units as shown in Figure 2.20. Our tree-based reduction oscillates between two modes of operation to maximally re-use the available memory blocks (a and $temp$ in Figure 2.20).

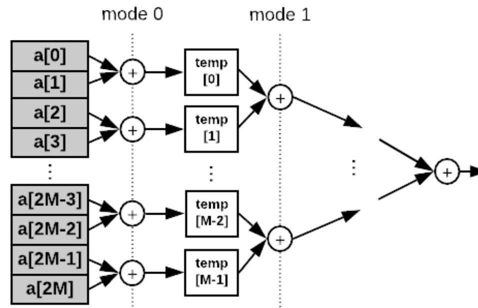

Figure 2.20. Tree-based vector reduction algorithm.

Another computationally expensive operation observed in OMP execution is LS optimization (Line 6 of Algorithm 1). We propose to use QR decomposition with the Gram-Schmidt orthogonalization technique [55] to efficiently implement LS optimization in our hardware accelerator. This technique decomposes the dictionary matrix D by iteratively forming the orthogonal matrix Q and a corresponding upper-triangular matrix R as shown in Algorithm 2. Using the acquired decomposition of the dictionary, the computation of the residual update in OMP (Line 7 of Algorithm 1) can be greatly simplified as:

$$r_i \leftarrow r_{i-1} - Q^i (Q^i)^T r_{i-1} \tag{2.12}$$

Algorithm 2. Incremental QR decomposition with modified Gram-Schmidt

Inputs: New column D_n, Q^{n-1}, R^{n-1}.
Output: Q^n, R^n.

1: $R^n \leftarrow \begin{bmatrix} R^{n-1} & 0 \\ 0 & 0 \end{bmatrix}$, $\epsilon^n \leftarrow D_n$
2: **for** j = 1,...,(n-1) **do**
3: $\quad R^n[j,n] \leftarrow (Q^{n-1}[:,n])^T \epsilon^n$
4: $\quad \epsilon^n \leftarrow \epsilon^n - R^n[j,n] Q^{n-1}[:,j]$
5: $R^n[n,n] = \|\epsilon^n\|$
6: $Q^n = Q^{n-1}\epsilon^n / R^n[n,n]$

2.5.2 Automated Design Customization

CuRTAIL provides an automated customization unit that maximizes DL model robustness while adhering to the limitations dictated by the underlying hardware platform and application, e.g., the available memory, computing resources, and system throughput. The input to the customization unit is the DNN architecture along with the application-specific runtime constraint and the available hardware resources. The output is the best combination of defender modules that balances the trade-offs depicted in Figure 2.21.

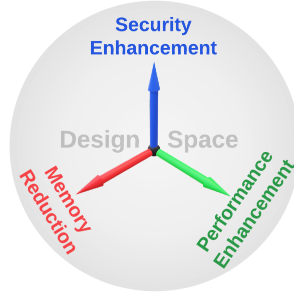

Figure 2.21. CuRTAIL provides customized defense by balancing the design-space trade-offs. The goal of CuRTAIL is to maximize model robustness while adhering to the underlying memory and runtime constraints.

To characterize the design trade-offs, we thoroughly examine the performance and resource utilization for different building blocks of a DL model. For FPGA platforms, the main resource bottlenecks for DL model implementation are the Block RAM (BRAM) capacity and

the number of DSP units. The dictionary matrices used in the input defender as well as the latent defender weights and biases are stored in the DRAM memory to be accessed during the execution phase. Upon computation, data is moved from the DRAM to BRAM which enables faster computation.

CuRTAIL sets the configuration of the defenders with regard to these two constraints, i.e., number of DSP units and the available BRAM. In particular, CuRTAIL solves the following optimization to find the best configuration for the number of defenders N_{def} and the number of processing units N_{PU} per defender.

$$\begin{aligned} & \underset{N_{PU},N_{def}}{Maximize}\,(DL_{robustness}) \qquad s.t.: \\ & T_{def}^{max} \leq T_u, \\ & N_{def} \times N_{PU} \times DSP_{PU} \leq R_u, \\ & N_{PU} \times [max_i(|W^i|) + max_i(|X^i| + |X^{i+1}|)] \leq M_u, \end{aligned} \tag{2.13}$$

where T_u, M_u, and R_u are the constraints on system latency, BRAM budget, and available DSP resources, respectively. Here, $|W^i|$ and $|X^i|$ denote the total number of parameters and the input dimension for layer i. The DSP_{PU} variable indicates the number of DSP slices used in one processing unit and T_{def}^{max} is the execution latency of the defender modules. CuRTAIL considers both sequential and parallel execution of defenders based on the available resources and size of the victim model. CuRTAIL performs an exhaustive search over the parameter N_{PU} and solves Equation (2.13) using the Karush-Kuhn-Tucker (KKT) method to calculate N_{def}. Once the optimization is solved for N_{PU}, N_{PE} is uniquely determined based on available resources. We note that this constraint-driven optimization is non-recurring and incurs negligible overhead ($10 - 100\ msec$ depending on the hardware platform.).

The OMP unit in CuRTAIL incurs a fixed memory footprint and latency for a given application. Therefore, the optimization of Equation (2.13) does not include this constant overhead. Instead, we deduct this overhead from the user-defined constraints and use the updated

upper bounds. The memory requirement for an OMP kernel is equivalent to $(patch_{len} \times (D_{size} + 1) + D_{size}^2) \times 4$ bytes, where $patch_{len}$ is the total number of pixels within a patch of an input sample (usually set to 64), and D_{size} is the number of columns in the dictionary matrix. The term $patch_{len} \times (D_{size} + 1)$ corresponds to the storage space required for the dictionary matrix as well as the input vector of a data patch. D_{size}^2 stands for the memory space required to store the R matrix while performing Algorithm 2. Note that the Q matrix re-uses the space originally dedicated to the dictionary to eliminate unnecessary use of memory resources. The required OMP computational time per input patch is superposed by the computational latency of the latent defenders that are executed in parallel with the input defender.

2.5.3 Computational Analysis and Scalability

Table 2.1 summarizes the computational complexity as well as the corresponding number of clock-cycles required for execution of each custom layer in CuRTAIL. In all entries from the third column of the table, β denotes a system-dependant constant that characterizes the runtime requirement per unit of floating point operation. For an OMP kernel (employed in the input defender), $patch_{len}$ indicates the number of elements in an input data patch, D_{size} is the dictionary size, and k represents the sparsity level (usually set to 5). Runtime of the ***Convolution*** layer is dependent on the input dimensionality ($W_{in} \times H_{in}$), convolution kernel size ($kernel_{size}$), number of input and output filter channels (f_{in}, f_{out}), and the values of N_{PE} and N_{PU} acquired from solving Equation (2.13). Same pattern holds for ***Dense*** layers where the input and output dimensions are denoted by N_{in} and N_{out}. PCA can be cast as a ***Dense*** layer as discussed in Section 2.5.1 with P and L representing the input and output dimensionalities, respectively.

Table 2.1. Runtime and computational complexity of each custom layer in CuRTAIL framework.

		Runtime	**Computational Complexity**
Input Defender	OMP Kernel	$\beta \times patch_{len}(kD_{size} + k^2)$	$\mathcal{O}(patch_{len}D_{size}^2)$
Latent Defender	Conv Layer	$\beta\lceil\frac{W_{in}}{N_{PE}}\rceil \times H_{in} \times f_{in} \times \lceil\frac{f_{out}}{N_{PU}}\rceil \times kernel_{size}^2$	$\mathcal{O}(W_{out}H_{out}f_{in}f_{out}kernel_{size}^2)$
	Dense Layer	$\beta\lceil\frac{N_{in}\times N_{out}}{N_{PE}\times N_{PU}}\rceil$	$\mathcal{O}(N_{in}N_{out})$
	PCA Layer	$\beta\lceil\frac{P\times L}{N_{PE}\times N_{PU}}\rceil$	$\mathcal{O}(PL)$

2.6 Evaluations

We evaluate CuRTAIL on five machine learning datasets: MNIST, SVHN, CIFAR-10, CIFAR-100, and ImageNet.

MNIST Benchmark. The MNIST data is a 28×28 gray-scale images of handwritten digits with $60,000$ train images and $10,000$ test samples. The images are normalized such that each pixel takes a real value in the range of $[0, 1]$. For this dataset, we train and use the DL topology proposed in [135] which is also available in Table 2.2.

SVHN Benchmark. This dataset consists of 32×32 real-world color images of house numbers in Google Street View images. We split the data into $\sim 60,500$ train images and $26,000$ test samples. The image pixels are normalized to the $[0, 1]$ range. Table 2.2 encloses the DL architecture used for this benchmark in our experiments.

CIFAR Benchmarks. We carry out our experiments on the two available CIFAR [100] datasets. CIFAR-10 and CIFAR-100 benchmarks consist of colored (RGB) images with dimensionality 32×32 that are categorized in 10 and 100 classes, respectively. We split the data samples into a set of $50,000$ training data and a set of $10,000$ test data. The images are normalized using per-channel mean and standard deviation such that each pixel takes a value in the $[0 - 1]$ range. In our experiments, we train and use the state-of-the-art DL topology proposed in [135] for CIFAR-10 and ResNet56-v2 [67] for CIFAR-100, as enclosed in Table 2.2.

ImageNet Benchmark. ImageNet is a large database consisting of over 15 million data samples. Typically, a subset of images belonging to 1000 different categories is used by the research community for learning evaluation of ImageNet data [102]. In our experiments, we train and use a DL architecture inspired by the well-known AlexNet [102] topology for ImageNet classification. Details about the trained model are available in Table 2.2. We down-sample ImageNet classes by a factor of 100 for execution efficiency purposes. Figure 2.22 illustrates several samples from each of the selected classes.

Figure 2.22. Example legitimate samples in ImageNet benchmark. Samples are randomly selected from the target classes.

Table 2.2. Benchmarked DL models for evaluating CuRTAIL effectiveness. $Conv$ layers are represented as $\langle input-channels \rangle \xrightarrow[stride]{\langle kernel\ size \rangle} \langle output-channels \rangle$ and FC layers are denoted by $\langle output-elements \rangle$FC.

	Conv+BN+ReLU	MaxPool	Conv+BN+ReLU	MaxPool	Conv+BN+ReLU	Conv+BN+ReLU	Conv+BN+ReLU	MaxPool	Classifier
MNIST	$3 \xrightarrow[\text{stride 1}]{5\times5} 20$	2×2 stride 2	$20 \xrightarrow[\text{stride 1}]{5\times5} 50$	2×2 stride 2	-	-	-	-	500FC 10FC, softmax
SVHN	$3 \xrightarrow[\text{stride 1}]{5\times5} 20$	2×2 stride 2	$20 \xrightarrow[\text{stride 1}]{5\times5} 50$	2×2 stride 2	-	-	-	-	1000FC 500FC 10FC, softmax
CIFAR-10	$[3 \xrightarrow[\text{stride 1}]{3\times3} 96]\times3$	2×2 stride 2	$[96 \xrightarrow[\text{stride 1}]{3\times3} 192]\times3$	2×2 stride 2	$192 \xrightarrow[\text{stride 1}]{3\times3} 192$	$192 \xrightarrow[\text{stride 1}]{1\times1} 192$	$192 \xrightarrow[\text{stride 1}]{1\times1} 10$	8×8 average pool	10FC, softmax
CIFAR-100 (ResNet56-v2 [67])	$3 \xrightarrow[\text{stride 1}]{3\times3} 16$	-	$\begin{bmatrix} 16 \xrightarrow{1\times1} 16 \\ 16 \xrightarrow{3\times3} 16 \\ 16 \xrightarrow{1\times1} 64 \end{bmatrix}\times6$	-	$\begin{bmatrix} 64 \xrightarrow{1\times1} 64 \\ 64 \xrightarrow{3\times3} 64 \\ 64 \xrightarrow{1\times1} 128 \end{bmatrix}\times6$	$\begin{bmatrix} 128 \xrightarrow{1\times1} 128 \\ 128 \xrightarrow{3\times3} 128 \\ 128 \xrightarrow{1\times1} 256 \end{bmatrix}\times6$	-	8×8 average pool	100FC, softmax
ImageNet	$\begin{bmatrix} 3 \xrightarrow[\text{stride 4}]{11\times11} 96 \\ 96 \xrightarrow[\text{stride 1}]{5\times5} 256 \end{bmatrix}$	3×3 stride 2	$256 \xrightarrow[\text{stride 1}]{3\times3} 128$	3×3 stride 2	$128 \xrightarrow[\text{stride 1}]{3\times3} 128$	$128 \xrightarrow[\text{stride 1}]{3\times3} 128$	-	3×3 stride 2	1024FC 1024FC 10FC, softmax

2.6.1 Details of MRR Training

In the following, we enclose the details for training CuRTAIL defenders that are evaluated in the experiments. Note that the MRR training phase is a one-time process and its cost will be amortized among all future executions of CuRTAIL.

Training input dictionaries. We learn separate input dictionaries for each class in the benchmarked dataset. For each image in the dataset, we randomly sub-sample 30 small patches and create a new training set. Patch are set to 7×7 for MNIST and SVHN, 8×8 for CIFAR-10 and CIFAR-100, and 16×16 for ImageNet. We set the number of columns in each dictionary to 225. Dictionaries are learned following the description in Section 2.4.2. Once the dictionaries are learned, we execute OMP (Algorithm 1) to denoise input samples. The PSNRs are then computed as in Equation (2.4), and compared against a cut-off threshold to raise alarms for high distortion values. We set the cut-off threshold value of input defender such that all the training data are considered legitimate samples.

Training Latent Defenders. For each application, we train a maximum of 16 latent defenders all of which checkpoint the second-to-last layer of the victim model. For ImageNet benchmark we only used 1 defender due to the high computational complexity of the pertinent neural network and attacks. We initialize the weights of latent defenders using those of the victim model, then retrain them by adding the extra term to the loss function with parameter γ set to 0.01 for all applications (see Equation (2.2)). Each defender module is trained for the same number of epochs as the original training of the victim model with the same optimizer. The learning rate is set to $\frac{1}{10}$ of that of the victim model as the model is already in a relatively good local minimum.

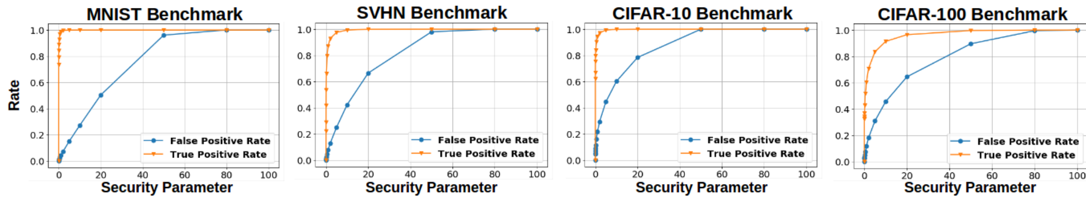

Figure 2.23. CuRTAIL security parameter controls the TP and FP rates. The number of latent defenders in this experiment is 16.

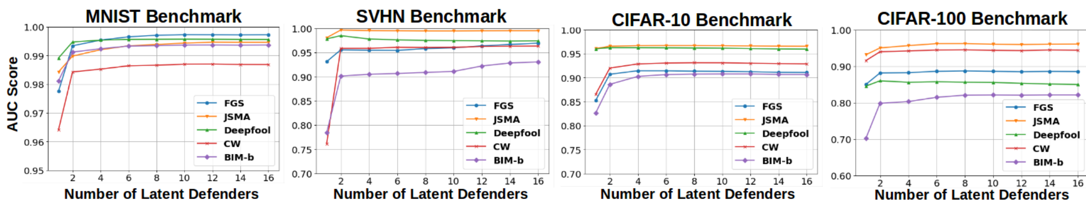

Figure 2.24. Using more MRRs improves the detection performance for all datasets.

2.6.2 Attack Analysis and Resiliency

We leverage a wide range of attack methodologies (namely, FGS [56], BIM [104], CarliniL2 [22], and Deepfool [142]) with varying parameters to ensure CuRTAIL's generalizability. The perturbation levels are selected such that the adversarial noise is undetectable by a human observer (Table 2.3 summarizes the pertinent attack parameters). We evaluate our defense mechanism in two attack scenarios:

- The attacker has complete access to the parameters and the architecture of the victim model but is not aware of the defense mechanism (a.k.a., **black-box attack**).
- The attacker has complete access to the parameters and the architecture of the victim model as well as the defender modules (a.k.a., **adaptive white-box attack**).

Table 2.3. Attack parameters. For CarliniL2 attack [22], "C" denotes the confidence, "LR" is the learning rate, "steps" is the number of binary search steps, and "iterations" stands for the maximum number of iterations. Superscripts ($m \rightarrow$ MNIST, $s \rightarrow$ SVHN, $c \rightarrow$ CIFAR, $a \rightarrow$ all) are used to indicate the benchmarks for which the parameters are used.

Attack	Attack Parameters
FGS	$\epsilon \in \{0.01^a, 0.05^a, 0.1^{m,c}, 0.2^m\}$
Deepfool	$n_{iters} \in \{2^a, 5^a, 10^a, 20^a, 50^a, 100^a\}$
BIM	$\epsilon \in \{0.001^a, 0.002^a\}, n_{iters} \in \{5^a, 10^a, 20^a, 50^m, 100^m\}$
CarliniL2	$C \in \{0^a, 10^a, 20^{s,c}, 30^{s,c}, 40^{s,c}, 50^c, 60^c, 70^c\}$ LR = 0.1^a, steps = 10^a, iterations = 500^a

To characterize the performance of the proposed defense methodology against adversarial attacks, we evaluate CuRTAIL in terms of both the True Positive (TP) and False Positive (FP) detection rates. In this context, TP refers to the ratio of adversarial samples correctly detected by the system while FP denotes the ratio of legitimate samples that are mistakenly categorized as being malicious. There is an inherent trade-off between the TP and FP detection rates that can be controlled using the security parameter discussed in Section 2.4. The Area Under Curve (AUC) for a TP versus FP plot fully encapsulates this trade-off and can be used as a measure to quantify the quality of adversarial detection. A random decision has an AUC of 0.5 while an ideal detector will have an AUC of 1.

2.6.3 Black-Box Attacks

In our first analysis, we study the relationship between detection success and the security parameter. We present the transition of the FP and TP rates with SP in Figure 2.23. In this experiment, we use the test data to generate a dataset of adversarial samples using all attack algorithms/parameters of Table 2.3; we then evaluate the performance of our MRRs against these

diverse adversarial samples to obtain the TP curve (shown in orange). The FP curve (shown in blue) is obtained by evaluating the defenders on the clean test data. An ideal defense would have FP≈ 0 and TP≈ 1.

In our next analysis, we evaluate the effect of the number of defenders on CuRTAIL detection. Figure 2.24 shows the AUC obtained by CuRTAIL for different attack configurations where the adversary knows everything about the model but is not aware of the defenders. For a given number of defenders, the AUC for MNIST is relatively higher compared to more complex benchmarks (e.g., CIFAR-10). This is consistent with our hypothesis since the unexplored subspace is larger in higher-dimensional benchmarks. Note that using more defenders eventually increases the AUC. We further summarize the performance of the CuRTAIL methodology against each of the FGS, JSMA, Deepfool, BIM, and Carlini&WagnerL2 attacks in Table 2.4. We used the open source library[3] provided by [147], for implementation of the attack algorithms. The JSMA attack was too slow on ImageNet, thus, we did not include the results in this study.

Table 2.4. AUC obtained by 16 latent defenders checkpointing the second-to-last layer of the victim model. For ImageNet, we only used 1 defender due to the high computational complexity of the pertinent neural network and attacks.

	MNIST	SVHN	CIFAR-10	CIFAR-100	ImageNet
FGS	0.997	0.969	0.911	0.885	0.881
JSMA	0.995	0.995	0.966	0.961	-
Deepfool	0.996	0.974	0.960	0.850	0.908
CarliniL2	0.987	0.963	0.929	0.944	0.907
BIM	0.994	0.931	0.907	0.821	0.820

2.6.4 Adaptive White-Box Attack

To further corroborate the robustness of MRR methodology, we applied the state-of-the-art Carlini&WagnerL2 attack in a white-box setting. A similar strategy was used in [21] to break the state-of-the-art countermeasures including MagNet [135], APE-GAN [174], and other recently proposed efficient defences methods (e.g., [229]). The attacks in [21] are gray-box attacks, meaning that the attacker is aware of the defense mechanism but does not have access

[3] https://github.com/tensorflow/cleverhans

Table 2.5. Evaluation of MRR methodology against adaptive white-box attack. We compare our results with prior work including Magnet [135], Efficient Defenses Against Adversarial Attacks [229], and APE-GAN [174]. For each evaluation, the L_2 distortion is normalized to that of the attack without the presence of any defense mechanism. Note that highly disturbed images (with large L_2 distortions) can be easily detected using the input defenders; however, for fair comparison to prior work, we did not include our non-differentiable input defenders in this experiment.

	MRR Methodology (White-box Attack)												**Prior-Art Defenses (Gray-box Attack)**		
Security Parameter	SP=1%						SP=5%						Magnet	Efficient Defenses	APE-GAN
Number of Defenders	N=0	N=1	N=2	N=4	N=8	N=16	N=0	N=1	N=2	N=4	N=8	N=16	N=16	-	-
Defense Success (TP Rate)	-	43%	53%	64%	65%	66%	-	46%	63%	69%	81%	84%	1%	0%	0%
Normalized Distortion (L_2)	1.00	1.04	1.11	1.12	1.31	1.38	1.00	1.09	1.28	1.28	1.63	1.57	1.37	1.30	1.06
FP Rate	-	2.9%	4.4%	6.1%	7.8%	8.4%	-	6.9%	11.2%	16.2%	21.9%	27.6%	-	-	-

to its parameters. We perform a more powerful attack against our defense where the attacker also knows the parameter set of the defenders. Following the guidelines in [21], we modify the objective function of the Carlini&WagnerL2 attack as follows:

$$
\begin{aligned}
&minimize \ \|x - x^a\|^2 + c \cdot l_c(x^a) + d \cdot l_d(x^a) \\
&l_d = \sum_{n=1}^{N} max(D_n(x^a) - \tau_n^i, 0),
\end{aligned}
\tag{2.14}
$$

where x is the original input, x^a is the adversarial sample, $l_c(\cdot)$ is a cost designed to mislead the victim classifier, and $l_d(\cdot)$ is a cost designed to deceive the defender models. Parameters c and d are constants which are tuned using binary search. We followed the instructions of [21] to set the loss function l_d in order to incorporate the MRR specific defense parameters. In particular, the value $D_n(x^a)$ in Equation (2.14) is the L_2 distance $\|f(x^a) - C^i\|^2$ in the nth defender, i is the target class for misclassification attack, and τ_n^i is the cut-off threshold for class i in the nth defender. We set the learning rate of the attack to 0.01, the confidence rate to 0, the maximum number of iterations to 10000, and the number of binary searches to 20 as suggested in [21]. For attack implementation, we used the library provided by the authors of the Carlini&WagnerL2 adaptive attack [4] and incorporated our defender specific loss into their algorithm.

Table 2.5 presents the success rate of the attack algorithm for different number of defender

[4]https://github.com/carlini/MagNet

modules and security parameters for the MNIST benchmark. As shown in Table 2.5, with a single defender, the defense success rate (TP) is 43% and 46% for security parameters of SP=1% and SP=5%, respectively. If we employ 16 defender modules in parallel, the TP rate increases to 66% for SP=1% and 84% for SP=5%, at the cost of a small increase in FP rate. Higher number of MRR modules in our defense mechanism also results in a larger perturbation in the generated adversarial samples. Note that such high perturbations can be detected via CuRTAIL input defenders; however, we did not include input defenders in this evaluation for comparison fairness. Compared to the state-of-the-art defenses of Table 2.5, CuRTAIL achieves a better TP rate. In addition, the attacker is required to inject a higher amount of perturbation (in terms of L_2 norm) to mislead CuRTAIL defenders in a white-box setting.

2.6.5 Performance Analysis

We implement the customized defender modules on *Xilinx Zynq-ZC702* and *Xilinx UltraScale-VCU108* FPGA platforms. All modules are synthesized using *Xilinx Vivado v2017.2*. We integrate the synthesized modules into a system-level block diagram with required peripherals, such as the DRAM, using Vivado IP Integrator. The frequency is set to $150\ MHz$ and power consumption is estimated using the synthesis tool. For comparison, we evaluate CuRTAIL performance against a highly-optimized TensorFlow-based implementation on two low-power embedded boards: (i) The *Jetson TK1* development kit which contains an *NVIDIA Kepler* GPU with 192 CUDA Cores as well as an *ARM Cortex-A15* 4-core CPU. (ii) A more powerful *Jetson TX2* board with an *NVIDIA Pascal* GPU with 256 cores and a 6-core *ARM v8* CPU.

Robustness and throughput trade-off. Increasing the number of checkpoints improves the reliability of model prediction in the presence of adversarial attacks (Section 2.6.2) at the cost of reducing the effective throughput of the system. In applications with severe resource constraints, it is crucial to optimize system performance to ensure maximum immunity while adhering to the user-defined timing constraints. In scenarios with more flexible timing budget, the customization tool automatically allocates more instances of the defender modules while under strict timing

constraints, the robustness is decreased in favor of the throughput.

Figure 2.25 demonstrates the throughput versus the number of defender modules for MNIST benchmark on Zynq FPGA. CuRTAIL accelerator can reach a throughput of 1400 samples per second with 16 MRRs. For the SVHN benchmark, which has a similar DL architecture to MNIST, the ARM v8 CPU can achieve a throughput of 1400 samples per second when only one defender is executed. CuRTAIL implementation on UltraScale FPGA can run 8 defenders in parallel the same throughput. This translates to an improvement in the AUC from 0.76 to 0.96.

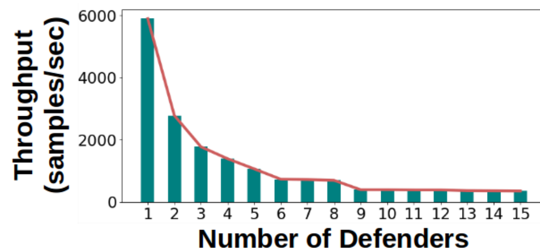

Figure 2.25. CuRTAIL throughput with samples from the MNIST dataset versus the number of instantiated defenders (implemented on *Xilinx Zync-ZC702* FPGA).

Throughput and energy analysis. To corroborate the efficiency of CuRTAIL, we also evaluate MRR performance on *TK1* and *TX2* boards operating in CPU-GPU and CPU-only modes. We define the performance-per-Watt measure as the throughput over the total power consumed by the system. This metric is an effective representation of the system performance since it integrates two influential factors for real-world embedded applications. All evaluations in this section are performed with only one input and latent defender. Figure 2.26 (left) illustrates the performance-per-Watt for different hardware platforms. Numbers are normalized by the performance-per-Watt of the *TK1* platform in CPU mode. As shown, CuRTAIL implementation on Zynq shows an average of $38\times$ improvement over *TK1* and $6.2\times$ improvement over *TX2* in CPU mode. The more expensive UltraScale FPGA performs relatively better with an average improvement of $193\times$ and $31.7\times$ over *TK1* and *TX2*, respectively.

Figure 2.26 (right) shows the comparisons with GPU platforms. All values are normalized by the *TK1* performance-per-Watt in the CPU-GPU mode. Our evaluations show an average

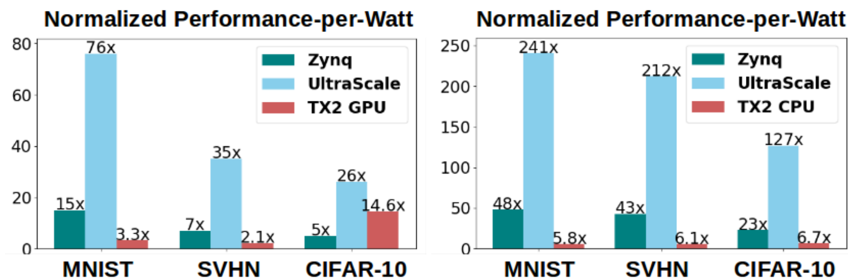

Figure 2.26. Performance-per-Watt comparison with embedded CPU (left) and CPU-GPU (right) platforms. Reports are normalized by the performance-per-Watt of *TK1*.

of $9\times$ and $45.7\times$ improvement over *TK1* by the Zynq and UltraScale FPGAs, respectively. Comparisons with *TX2* demonstrate $2.74\times$ and $41.5\times$ improvement for the Zynq and UltraScale implementations. Note that UltraScale performs noticeably better than Zynq which emphasizes the effect of resource constraints on parallelism and throughput.

2.6.6 Discussion on Transferability of Adversarial Samples

Figure 2.27 demonstrates an example of the adversarial confusion matrices for victim neural networks with and without using parallel checkpointing learners. In this example, we set the security parameter to only 1%. As shown, the adversarial sample generated for the victim model **are not transferred** to the checkpointing modules. In fact, the proposed approach can effectively remove/detect adversarial samples by characterizing the rarely explored sub-spaces and looking into the statistical density of data points in the pertinent space.

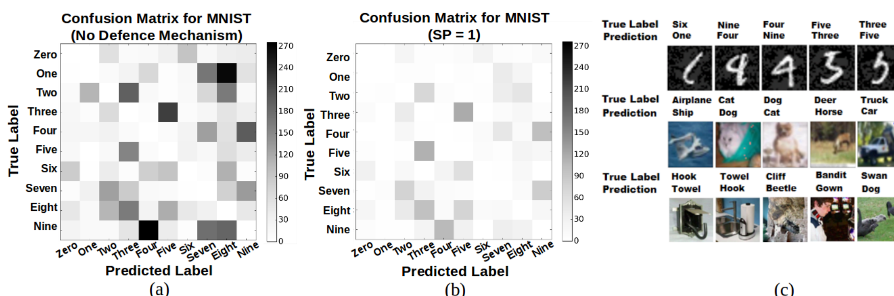

Figure 2.27. Example adversarial confusion matrix (a) without MRR defense mechanism, and (b) with MRR defense and $SP = 1\%$. (c) Example adversarial samples for which accurate detection is hard due to the closeness of decision boundaries

Note that the remaining adversarial samples that are not detected in this experiment are crafted from legitimate samples that are inherently hard to classify even by a human observer due to the closeness of decision boundaries corresponding to such classes. For instance, in the MNIST application, such adversarial samples mostly belong to class 5 that is misclassified to class 3 or class 4 misclassified as 9. Such misclassifications are indeed the model approximation error which is well-understood to the statistical nature of the models. As such, a more precise definition of adversarial samples is extremely required to distinguish malicious samples form those that simply lie near the decision boundaries.

2.7 Conclusion

This work proposes CuRTAIL, a novel end-to-end framework for online accelerated defense against adversarial samples in the context of deep learning. We introduce modular robust redundancy as a viable countermeasure to significantly reduce the risk of integrity attacks. The proposed MRR methodology explicitly characterizes statistical properties of the features within different layers of a neural network by learning the corresponding probability density functions. Using a hardware/algorithm co-design approach, CuRTAIL automated customization tool optimizes the defense layout to maximize model reliability (safety) while complying with the hardware and/or user constraints. This, in turn, ensures applicability to various DL tasks and hardware platforms. CuRTAIL robustness is evaluated against a wide range of attack models. Proof-of-concept experiments on various data collections corroborate successful detection of adversarial samples with relatively small probability of false alarm. Our evaluations on various hardware platforms indicates the effectiveness and practicality of CuRTAIL.

2.8 Acknowledgements

Chapter 2 is a partial reprint of the material as it appears in: M. Javaheripi, M. Samragh, B. Rouhani, T. Javidi, and F. Koushanfar, “Curtail: Characterizing and Thwarting Adversarial

Deep Learning", in *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2020. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Ensuring DL Robustness to Fault Injection Attacks

DNNs have enabled a transformative shift in various applications ranging from natural language processing and computer vision to healthcare and autonomous driving. With the deep integration of autonomous systems in safety-critical tasks, model assurance and decision robustness have gained imminent importance [46, 88, 92, 162]. Although DNNs demonstrate superb accuracy in controlled settings, it has been shown that they are particularly vulnerable to fault-injection attacks. Recent work [72, 154] demonstrates how changing a few bits of the victim DNN's weights can reduce the classification accuracy to below random guess. These malicious bit flips have been realized in DNN accelerators via rowhammer attacks on the DRAM containing the model weights [219].

In response to bit-flip attacks, prior work suggests adding specific constraints on DNN weights during training such as binarization [157], clustering [71], or block reconstruction [112]. Adding such constraints increases the number of bit-flips required to deplete the inference accuracy, however, they do not entirely mitigate the threat. Additionally, the proposed constraints often severely affect the underlying DNN's test accuracy. Other work [113, 119] propose to use machine learning (ML) based techniques where a simpler model is trained to detect faults in the victim DNN. However, their detection rate and false positive rate are bound by the accuracy of the ML-based detector. To ensure DNN robustness, it is crucial to augment autonomous systems

with an online fault detection strategy that delivers strict performance guarantees. To the best of our knowledge, none of the earlier works provide the needed detection strategy.

We propose AccHashtag, a highly accurate real-time fault detection methodology for DNNs deployed in embedded applications. AccHashtag is the first method to provide strict statistical bounds on fault detection performance and deliver 0% false positive rate. AccHashtag extracts a unique signature from the benign DNN prior to deployment. At runtime, the signature is used to validate the integrity of the DNN and verify the inference output on the fly. We propose to leverage a low-collision hashing scheme, called the Pearson hash, to extract an 8-bit signature from the pertinent weights in each DNN layer. Our hash-based signature extraction delivers several benefits: (1) hash-based integrity check enables accurate fault detection that is robust to false alarms. (2) The hash algorithm is devised particularly for low-overhead execution on commodity processors. We design a customized core for hash generation and verification on field-programmable gate arrays (FPGAs) that works alongside the co-processor hosting the target DNN. Concurrent with DNN execution, the weights are streamed to the FPGA core which then generates the hash signature. We further optimize the streaming size to maximally overlap the latency of hash generation core with the latency of communication through the underlying advanced extensible interface (AXI) with the host CPU.

There exist an inherent trade-off between fault detection performance and the storage/runtime overhead that is determined by the number of DNN layers used for signature extraction. To balance this trade-off, we propose a novel sensitivity analysis scheme that identifies the most vulnerable layers within the DNN to be used for signature extraction. This, in turn, leads to an extremely lightweight detection methodology that incurs negligible storage and runtime, making it amenable for use in resource-constrained embedded environments. Notably, our sensitivity analysis enables AccHashtag to achieve a 100% detection rate using as few as one layer for hash extraction.

Our detection strategy is compatible with the challenging threat model where the attacker has full control over the DRAM to freely select the location and number of bit flips. In addition,

the attacker has full knowledge of the underlying detection algorithm, i.e., the hash function. To calibrate ACCHASHTAG detection, the user does not require access to any labeled data, fine-tuning, or model training. The user only chooses a secret reordering rule to generate the input for the hash function from the DNN layer weights. Using the reordering rule, the hash signatures can be robustly extracted from the DNN at runtime without the attacker's interference.

We validate the effectiveness of ACCHASHTAG by performing extensive experiments on various DNN architectures and visual datasets. The evaluated DNNs are injected with the state-of-the-art progressive bit-flip attack [154]. We show that ACCHASHTAG achieves a 100% detection rate with 0 false alarms while incurring $< 1.3KB$ storage and $< 1\%$ runtime compared to DNN inference on an embedded GPU. When using our customized hash computation core on FPGA, the runtime can be further decreased by an average of $2.1\times$, thereby enabling online signature generation and verification alongside DNN inference. Our proposed methodology outperforms prior art across all benchmarks both in terms of attack detection and algorithm execution overhead. Compared to best prior work, ACCHASHTAG shows orders of magnitude faster execution and lower storage. In summary, ACCHASHTAG contributions are as follows:

- Introducing ACCHASHTAG, the first framework for online detection of DNN fault-injection attacks with provable guarantees on performance.
- Constructing a novel signature generation scheme based on Pearson hash which enables low-overhead and highly accurate fault detection.
- Providing lower bounds on attack detection rate using a statistical analysis of hash collision.
- Devising a sensitivity analysis to identify vulnerable layers within any given DNN architecture. ACCHASHTAG automatically finds DNN layers with a high probability for attack and tailors the fault detection to those layers.
- Designing an FPGA core for hash generation which enables high-throughput DNN integrity validation.

3.1 Background and Prior Work

3.1.1 Bit-Flip Attack

Recent work has developed various fault-injection techniques [97, 188, 196] that can be utilized to alter bits stored in the DRAM memory. These techniques give rise to the plethora of attacks that take advantage of the bit-flipping tools to induce adversarial behavior in deployed DNNs. Researchers have demonstrated the vulnerability of DNNs to fault-injection attacks that target model parameters. Perhaps the pioneer in this domain is [122] which alters a single parameter throughout the DNN to change the classification result. Follow-up work [72] analyzes the effect of targeted bit flips induced by the Row hammer attack on DNN accuracy. The authors perform the bit flips in the floating-point representation and show that their injected bitwise errors can lead to $> 90\%$ accuracy degradation when applied on certain DNN parameters.

Current state-of-the-art bit-flip attack [154] leverages a gradient-based progressive bit search to strategically identify the vulnerable bits in the DNN. Their attack is applied on quantized DNN parameters with the fixed-point representation. Other variants of the bit-flip attack exist which leverage a similar adaptive method to find the vulnerable bits but differ in the attack objective: rather than degrading the accuracy on all samples, authors of [155, 156] perform bit flips to misclassify certain input examples as a target class. In this work, we direct our focus to the generic untargeted bit-flip attack [154, 219] as it provides the most general attack objective. We emphasize that ACCHASHTAG is applicable to other attack variants as our methodology relies on signature extraction and verification. This, in turn, allows us to detect (adversarial) changes in DNN parameters regardless of the underlying attack objective.

Attack Formulation. Let us denote by $\{B_l\}_{l=1}^{L}$ the total bits from the Two's complement representation of per-layer DNN weights where l is the layer index. To maximally reduce the DNN accuracy, the attacker iteratively identifies the bit with the highest gradient $\max_{B_l} |\nabla_{B_l}\mathcal{L}|$ in each layer of the DNN. Here, $\mathcal{L}$ denotes the DNN inference loss. Once the per-layer most vulnerable bits are detected, the new loss will be measured for each candidate bit-flip. Finally,

the bit that results in the maximum loss is selected and flipped. The iterative process continues until the DNN accuracy falls below the attacker's desired value.

3.1.2 Existing Defenses

Prior art propose various techniques to increase robustness to fault-injection attacks that occur during DNN training and execution. To thwart training-time attacks, authors of [54], propose a trust-based framework as the fault detection mechanism. The performance of this method is strongly dependent on the accuracy of the trust evaluation mechanism [190, 191]. We direct our focus to fault injection attacks applied on the DNN's internal parameters at inference time. A high-level comparison of ACCHASHTAG with prior works is enclosed in Table 3.1.

Several prior defenses against inference-time fault injection attacks suggest adding specific constraints to the model during training. Authors of [71] show that adding a piece-wise clustering constraint to the training objective or performing binarized training can improve resiliency. Follow-up work [112] proposes to locally reconstruct DNN weights during inference to minimize or defuse the effect of the bitwise error caused by the bit flips. Such methods increase the number of bit flips required to reduce the victim DNN's classification accuracy. However, they do not detect or prevent fault-injection attacks. Additionally, due to the added constraints on the pertinent DNN, these methods reduce the inference accuracy of the victim model. Compared to these methods, ACCHASHTAG does not affect the inference accuracy in any way and is able to detect the occurrence of bit flips with 100% accuracy.

Other works suggest adding an ML-based attack detection mechanism. Authors of [113] train a smaller, checker network to verify the classification results produced by the original DNN. In case of a mismatch, the task is repeated and the output of the victim DNN is accepted, which results in a low detection rate. Compared to ACCHASHTAG lightweight detection method, the checker DNN incurs a higher computational/storage overhead and can itself be subject to fault-injection attacks. Another work [119] uses the magnitude of the gradient to find sensitive weights. The authors then train a binary classifier on the sensitive weights to find

bit flips. The ML-based detection techniques are bound by the classification accuracy of the underlying detector model and thereby have lower true positive rate and higher false positive rate compared to ACCHASHTAG. We provide a probabilistic lower bound on ACCHASHTAG detection performance that outperforms prior work.

Most recently, authors of [111] employ checksums to detect bitwise errors in weight groups. The detection performance of the proposed methodology relies on the choice of the group size, i.e., the number of weights used to compute each checksum value. To obtain a good trade-off between detection performance and the storage/runtime overhead, the authors suggested using higher group sizes. From a probabilistic point-of-view, checksum on large groups has higher false negative rate compared to our hash-based mechanism. This is because checksum inherently overlooks specific even-numbered bit flips. As shown in our experiments, the best reported results from [111] achieve lower detection accuracy compared to ACCHASHTAG while requiring higher storage and runtime.

Table 3.1. High-level comparison of ACCHASHTAG with prior work.

	100% TPR	0% FPR	Degrade DNN Accuracy	Require DNN Training	Low Overhead	Customized Hardware
Piece-wise Clustering [71]			✓	✓		
Weight Reconstruction [112]			✓	✓		
DeepDyve [113]				✓		
Weight Encoding [119]				✓		
RADAR [111]		✓	✓			
HASHTAG [86]	✓	✓			✓	
ACCHASHTAG	✓	✓			✓	✓

3.2 ACCHASHTAG Methodology

Figure 3.1 demonstrates the overview of ACCHASHTAG methodology for detecting fault-injection attacks in DNN parameters, i.e., bit flips. The core idea in ACCHASHTAG is to generate a compact (ground-truth) signature from the benign DNN. This is done by generating per-layer hashes of DNN parameters prior to model deployment. The signature is then used to verify the integrity of DNN parameters during execution to validate the inference result and mitigate

malicious behavior. Our methodology incurs minimal computation/storage overhead and is devised based on lightweight solutions to enable efficient and real-time execution in embedded systems. AccHashtag comprises two phases to detect anomalies in DNN parameters:

Pre-processing Phase. AccHashtag preprocessing is a one-time process in which the detection mechanism is calibrated for the underlying victim DNN. There exist an inherent tradeoff between attack detection performance and the computation/storage requirement for extracting layer signatures; On the one hand, hashing all layers ensures that the detection mechanism can universally adapt to attacks in any subset of layers. On the other hand, hash computation and storage are linear in the number of layers used for detection. We observe that various DNN layers are not equally targeted by fault-injection attacks. Motivated by this, we devise a novel sensitivity analysis scheme that models the vulnerability of DNN layers to bit-flip attacks. The top-k most vulnerable layers, called *checkpoint layers*, are then used to extract the hashes. This, in turn, allows AccHashtag to maximize detection performance under any given computation/storage budget. k is a tunable hyperparameter in AccHashtag which can range from 1 to L where L is the total number of linear layers in the victim DNN. If the user selects $k = L$, then hashes will be generated for all layers. However, as we show in our experiments (see Section 3.4.3 Figure 3.11), for the wide variety of evaluated models our detection rate reaches 100% when generating hashes for at most $k = 5$ layers. This is due to the ability of our sensitivity analysis to accurately locate layers with the highest probability of fault injection.

Online Execution. This recurring phase is activated when the underlying DNN is queried. During online execution, new hashes are extracted from checkpoint layers in parallel to the DNN inference. The new hashes are then validated against the ground-truth hash values from the pre-processing phase to verify the legitimacy of model parameters. Upon hash mismatch, an alarm flag is raised to notify the user that the system is compromised. The user shall then evict the deployed model and reload the ground-truth weights from the source. We accelerate the operations performed in AccHashtag's online execution phase using a customized FPGA core that interacts with the DNN's host processor.

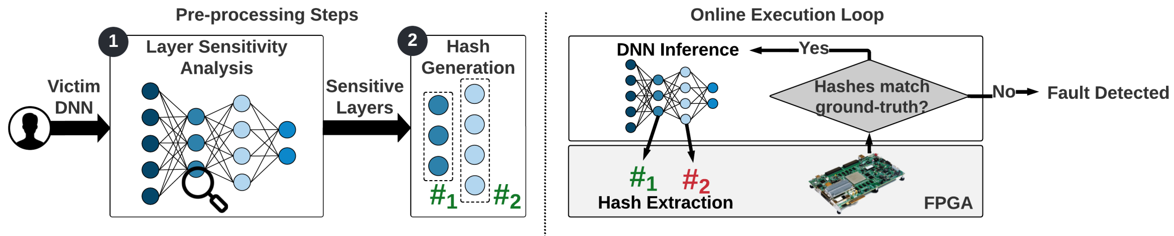

Figure 3.1. Global flow of AccHashtag detection. During the pre-processing phase, we first identify sensitive DNN layers that are most prone to fault-injection attacks. We then generate a customized signature from the identified sensitive layers. During online execution, the signature is used to validate the model's integrity in real-time, parallel to conducting inference.

3.2.1 Threat Model

In this work, we direct our focus to fault-injection attacks that target DNN parameters, i.e., the bit-flip attack. In this scenario, the attacker has full knowledge of the victim DNN architecture and its parameters. They further know the physical address of the model parameters and have access to a subset of the data used for training the DNN. The attacker uses the data to progressively identify vulnerable weights and flip their value. This is done by performing a Row Hammer Attack (RHA) [97] on DRAM locations where the model parameter are stored [72, 219]. To keep the attack stealthy and reduce the high cost of RHA, we assume the attacker is motivated to minimize the number of flipped bits as is observed in the state-of-the-art attacks [154, 155]. As such, we do not consider random bit flips since they are shown to be ineffective in reducing DNN accuracy even with a high number of flipped weights [154, 219].

We evaluate our detection in the challenging white-box scenario where the attacker knows which layers are used for detection. He is also fully aware of the hash algorithm used for generating the per-layer signatures. However, he does not know the secret hash values and the parameter ordering used for generating the hashes. Following prior work [111], we assume the secret hashes are stored in the secure on-chip static random access memory (SRAM) which is not accessible by the attacker. Note that even when SRAM storage is not available, our detection secrets are still immune to RHA. This is due to their low memory footprint (less than 5 KB) that

makes them hard to target by RHA as shown in [72].

3.3 ACCHASHTAG Components

3.3.1 Hash-based Signature Extraction

Hash functions generate a constant-length code value which is independent of the size of the corresponding hashed data. This property motivated us to leverage hashing as the underlying mechanism for extracting DNN layer signatures. Among the available hash functions, ACCHASHTAG incorporates the Pearson hash [151] which operates on input streams at Byte granularity. Below we present the Pearson scheme for generating an 8-bit hash value.

Pearson Hash Formulation. The user generates a *hash table* T which contains a random permutation of integer values in the range [0, 255], i.e., $\mathbb{Z}_{256}$. For an incoming vector of length N containing Byte values $\{x_i\}_{i=1}^{N}$, the Pearson hash is defined recursively as follows:

$$h(x_1, x_2, \ldots, x_N) = T(h(x_1, x_2, \ldots, x_{N-1}) \oplus x_N) \tag{3.1}$$

where $\oplus$ represents the `XOR` operation. Since T is an arbitrary permutation of values in $\mathbb{Z}_{256}$, there exists a total of $(256)!$ hash variations for a fixed input stream. The Pearson hash can be extended to generate hashes longer than 8 bits by repeating the above process several times and concatenating the results. However, as shown in our experiments, the 8-bit Pearson hash accurately detects the state-of-the-art bit-flip attack [154].

Our hashing scheme provides several desirable characteristics that makes it particularly amenable for low-overhead detection of fault injection attacks: (1) The hash computation is well-defined for execution in 8-bit processors and embedded CPUs [151]. (2) The hashing scheme is applicable to input streams of varying lengths, thereby providing high customizability for various DNN layer configurations. (3) Pearson hash accommodates input streams with fixed-point representation which have been target to contemporary bit-flip attacks [154, 155]. Fixed-point parameter values are observed in quantized DNNs that are widely deployed in

embedded systems.

Signature Generation. To extract the ground-truth signature from a benign DNN layer, we first generate a random hash table T. The pertinent layer parameters are then fed to Equation (3.1) as the input stream $x_1, x_2, \ldots, x_N$ to generate the secret hash of the layer. The hash input stream is generated using a user-defined secret *ordering*. An example of such ordering is shown in Figure 3.2. Here, the hash input stream is constructed by first traversing the layer's weight kernel in the output channel dimension. Ordering adds a zero-cost layer of complexity to ACCHASHTAG signature generation which prevents the attacker from reproducing the per-layer secret hashes. Note that the hash input ordering does not affect ACCHASHTAG detection performance. The user can easily choose different secret orderings for various layers or change the ordering at any time to reinforce system integrity.

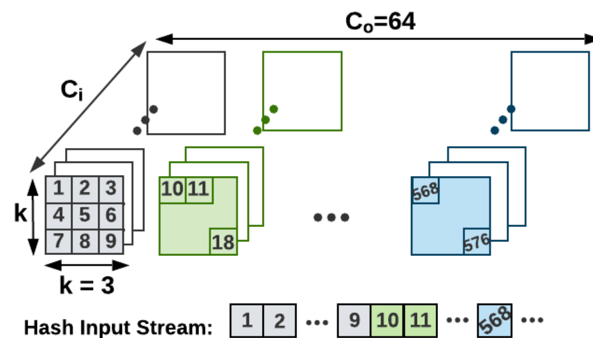

Figure 3.2. Reordering parameters in an example convolution layer for generating the hash input stream. The layer parameters are the convolution weight kernels $\in \mathbb{R}^{k \times k \times C_i \times C_o}$ where k, C_i, C_o denote the kernel size, input channels, and output channels, respectively.

3.3.2 Bounds on Detection Performance

In this section, we provide the worst-case performance bounds on our hash-based detection mechanism. Recall from the threat model (Section 3.2.1) that the attacker is not aware of the secret ordering used to generate the hashes from layer parameters. As such, even if the attacker gains full access to the Pearson hash tables, they will not be able to reproduce the ground-truth hash values. The attacker, therefore, performs the bit-flip attack without taking extra measures to

preserve the ground-truth hashes. In this context, the lower bound on ACCHASHTAG detection can be obtained by quantifying the probability of collision in our hashes. Collision occurs when multiple input streams are mapped to the same output hash. We analyze hash collision in two separate scenarios where the attacker alters 1) one or 2) more than one element of the parameter tensor in the target layer.

Single-element Alteration

When the attacker alters only one element in the weight block where the hash is computed, the user can detect the hash mismatch with 100% accuracy. This is due to an intrinsic collision property for the Pearson hash: for two input streams with exactly one value difference, the probability of collision is zero when the streams are Pearson hashed.

Let us denote the altered byte value inside DNN weights by $\tilde{x}_m$. The Pearson hash operation for the first m bytes can be written as:

$$h_m = T(h_{m-1} \oplus \tilde{x}_m) \tag{3.2}$$

where h_i is the short notation for $h(x_1, x_2, \ldots, x_i)$. Since the first $m-1$ bytes are unaltered, the value of h_{m-1} remains constant. By changing x_m, the hash value h_m changes due to the bijective property of the hash table T. Since the remaining elements $x_i|_{i=m+1}^{N}$ are unaltered, the new hash h_m propagates through the rest of the input chain, resulting in a different final hash h_N compared to the original weight block.

Multi-element Alteration

In cases where the attacker changes more than one weight value in the hash block, a possibility arises that the hash mismatch caused by the earlier perturbed elements is later corrected by a subsequent perturbed weight element such that the overall hash value h_N remains unchanged. Without loss of generality let us assume only two elements are altered: $\tilde{x}_m$ and $\tilde{x}_n$ $(m < n)$. As shown previously, changing the mth element, results in a new hash value

that propagates through the input chain until the next changed element. Let us denote the hash value of the first $n-1$ elements in the original and altered weight blocks by h_{n-1} and $\tilde{h}_{n-1}$, respectively. To ensure the final hash value of the block remains the same, the new value of the nth element $\tilde{x}_n$ needs to satisfy the following equation:

$$h_{n-1} \oplus x_n = \tilde{h}_{n-1} \oplus \tilde{x}_n \tag{3.3}$$

The above equation limits the number of allowed values for $\tilde{x}_n$ to only one. As such, the overall probability of obtaining the same hash after altering the bits in two elements is $\frac{1}{256} \sim 0.004$. This probability quantifies the chance of collision occurring in our hashing scheme and remains the same for any arbitrary number of elements altered bigger than one. As such, our (worst-case) lower bound on hash mismatch detection for the DNN is $\left(\frac{1}{256}\right)^{l_a}$. Here, l_a denotes the number of attacked layers where more than one weight element is flipped by the attacker.

We empirically evaluate our developed bound by performing multiple runs of hash extraction on an arbitrary input stream of length 1000. We randomly change a subset of k values within the input and measure the collision rate. As seen in Figure 3.3, by increasing the number of experiments, the collision probability asymptotically reaches 0.004 in all settings, which is compatible with the bound from our statistical analysis.

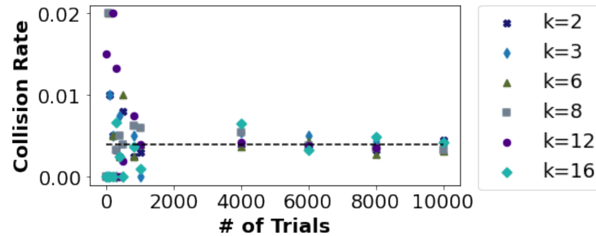

Figure 3.3. Collision rate versus number of trial runs for hashing an input stream of length 1000. Each trial randomly changes a subset $k \in [2, 3, 6, 8, 12, 16]$ of message elements.

3.3.3 Per-layer Sensitivity Analysis

State-of-the-art fault injection attacks leverage various techniques to identify weight values that most affect the accuracy if altered. By targeting the attack towards such vulnerable weights, the attacker requires very few bit flips to degrade the accuracy of the victim DNN below random guess. Motivated by this, we devise a sensitivity analysis that accurately finds the subset of layers inside the victim DNN that are most prone to fault injection. Our sensitivity formulation is inspired by prior work in DNN pruning [140]. Specifically, we utilize Taylor expansion to model the effect of per-layer weight change on DNN accuracy as an effective measure of sensitivity.

Linear layers in DNNs comprise two key parameters, namely the weight and bias: (W, b). Let us represent the entire parameter set for a given DNN with L layers by $P = \{(W, b)_1, (W, b)_2, \ldots (W, b)_L\}$ where the subscript denotes the layer index. Training the DNN is equivalent to minimizing a loss function $\mathcal{L}(D, P)$ over a corpus of data $D = (x_1, y_1), \ldots, (x_d, y_d)$ where x and y correspond to input examples and their labels, respectively. To degrade a pretrained DNN's accuracy, the attacker's goal is to maximize the loss over the given dataset. Let us denote by P and $\tilde{P}$, the DNN parameters before and after the attack. We model the attack objective as:

$$\max_{\tilde{P}} (\mathcal{L}(D, P) - \mathcal{L}(D, \tilde{P}))^2 \tag{3.4}$$

We, therefore, quantify the sensitivity of each DNN parameter by the increase in loss value caused by changing it. Bit-flip attacks often alter the sign as it causes the most dramatic change in the value of the underlying parameter, thereby greatly influencing the accuracy [154]. As such, we model parameter sensitivity by altering the sign $\tilde{p} = -p$ and measuring the effect on loss. Here the lower case p represents individual weight/bias elements in the DNN. The sensitivity

$S(\cdot)$ for the nth parameter p_n can thus be measured as:

$$S(p_n) = (\mathcal{L}(D, P) - \mathcal{L}(D, \tilde{P}|_{\tilde{p}_n = -p_n}))^2 \tag{3.5}$$

Since individual computation of (3.5) for each weight element inside the DNN is computationally prohibitive, we leverage Taylor expansion to estimate $S(\cdot)$. For a given function $f(x)$, the first-order approximation using Taylor polynomials at point $x = a$ is given by:

$$f(x) \approx f(a) + (x - a) \times \left.\frac{\partial f}{\partial x}\right|_{x=a} \tag{3.6}$$

By replacing f in the Taylor expansion formula with the loss function $\mathcal{L}$, we rewrite (3.5) as:

$$\mathcal{L}(D, P) - \mathcal{L}(D, \tilde{P}|_{\tilde{p}_n = -p_n}) \approx 2p_n \times \frac{\partial \mathcal{L}}{\partial p_n} \tag{3.7}$$

We thus measure the sensitivity of parameter p_n as:

$$S(p_n) \propto (p_n \times \frac{\partial \mathcal{L}}{\partial p_n})^2 \tag{3.8}$$

Note that the formula shown in (3.8) can be easily computed using a simple backward pass through the network to compute the first-order gradients. Once the sensitivity is obtained for each weight element, we define the sensitivity of each layer as the average over top-5 sensitivity values of its enclosing elements. We empirically explain our reason for choosing the top-5 weights by providing an analysis of the bit-flip attack in Section 3.4.2

3.3.4 Accelerating Hash Generation

To enable detection of faults in real time, we accelerate the hash computation on FPGA. This, in turn, allows for a parallel verification of weights and DNN execution. The top-level architecture of ACCHASHTAG FPGA accelerator is shown in Figure 3.4. Our FPGA design

communicates with the host CPU through AXI4 and AXI4-Lite interfaces. We leverage the AXI4 interface to receive the target DNN's weights in bursts. The burst reads are then fed to the input first in, first out (FIFO) buffer through the AXI master interface. The Pearson hash core interacts with the input FIFO buffer to receive the hash input stream sequentially and generate the corresponding signatures. This systolic features have low global data transfer and high clock frequency, which is suitable for large-scale parallel design, especially on FPGAs. We utilize the simpler AXI4-Lite interface to interact with the control unit and send the appropriate instructions for controlling the hash module and relevant memory transfers. ACCHASHTAG control unit operates in three modes, namely, populating the hash table, querying the Pearson hash module for signature generation, and sending the final hash value to the host CPU. Once the hash computation concludes, the final hash value enters the output FIFO buffer and is sent back to the host CPU through the AXI master interface. We provide a break down of the utilized resources for all components in ACCHASHTAG accelerated hash generation in Table 3.2. Here, the design is synthesized for a Xilinx VCU108 FPGA.

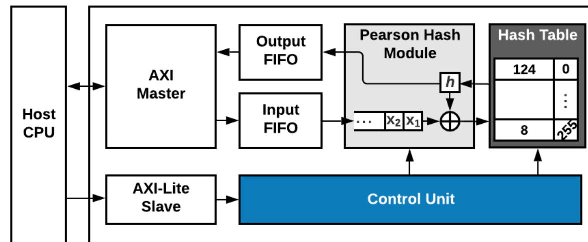

Figure 3.4. Overview of ACCHASHTAG accelerated hash generation and verification using a specialized FPGA compute kernel.

Table 3.2. Resource utilization of ACCHASHTAG components, synthesized on a Xilinx FPGA.

Module	BRAM	FF	LUT
Pearson Hash	-	244	962
Hash Table	2	2306	238
FIFO	-	512	580
Control Unit	-	-	1314
AXI Master	-	106	168
AXI-Lite Slave	-	144	232

We take several measures to increase the hash generation throughput. First, we design the Pearson hash module as a specialized form of parallel computing with a deeply pipelined processes inside the hash generation loop. Using pipeline forwarding, our design fetches new data from the input FIFO buffer and calculates the hash signatures within the same pipeline stage, thereby increasing end-to-end throughput. Secondly, we take advantage of the small footprint of the hash tables to implement them entirely using 8-bit Flip-flop registers on the FPGA. This, in turn, enables very low latency accesses to the table during hash computation. Finally, we optimize the burst length for AXI reads to maximally overlap the latency of hash computation with the input stream read latency from the AXI4 Master. An optimal burst length will result in a nearly diminished cost for AXI reads, thereby increasing throughput to that of the hash Pearson hash module, with negligible increase in FPGA resource utilization.

3.4 Experiments

In the following, we provide a comprehensive evaluation of ACCHASHTAG performance along with various analyses and discussions. Section 3.4.1 encloses details of our benchmarked models and datasets, attack setup and implementation, as well as definitions for the utilized evaluation metrics. Section 3.4.2 provides an analysis of the attack profile to clarify various design choices. Finally, in Section 3.4.3 we report the detection performance of ACCHASHTAG, compare it with best prior art, and analyze the storage and computation requirements.

3.4.1 Experimental Setup

Benchmarks. We evaluate ACCHASHTAG on two image datasets, namely, CIFAR10 [101] and ImageNet [164]. The datasets contain 10 and 1000 classes of RGB (red, green, and blue) images of dimensionality 32×32 and 224×224, respectively. We separate 20 examples from each class in the training data and create a small held-out *validation* dataset. This validation set is used to perform sensitivity analysis in the pre-processing phase.

Table 3.3 encloses an overview of the DNN architectures evaluated on each dataset and

their baseline test accuracies with 8-bit quantization. We evaluate CIFAR10 on two DNNs, namely, ResNet20 [66] and VGG11 [176]. For ImageNet, we perform experiments on four DNNs, namely ResNet18 [66], ResNet34 [66], AlexNet [102], and MobileNetV2 [170]. We further present the first systematic study of bit-flip attacks on Transformers by benchmarking two contemporary models used in vision tasks, namely ViT [38] and DeiT [93]. We leverage the open-source code in [226] to quantize pre-trained Transformer models. As shown in Table 3.3, Transformers show, on average, higher robustness towards fault-injection which results in a higher number of required bit flips for degrading their accuracy.

Table 3.3. Overview of the evaluated benchmarks. Here, CONV, FC, and ATTN represent convolution, fully-connected, and self-attention layers, respectively. Note that each self-attention layer consists of four fully-connected layers. The baseline top-1 accuracy and the average number of bit flips are reported for 8-bit quantized DNNs.

Dataset	Model	Layers	Top-1 Acc (%)	Bit Flips
CIFAR10	VGG11	8 CONV, 3 FC	89.3	89
	ResNet20	19 CONV, 1 FC	91.9	18
ImageNet	AlexNet	5 CONV, 3 FC	55.5	21
	ResNet18	20 CONV, 1 FC	68.8	8
	ResNet34	36 CONV, 1 FC	72.8	10
	MobileNet	52 CONV, 1 FC	70.3	3
	ViT	1 CONV, 12 ATTN, 1 FC	80.6	203
	DeiT	1 CONV, 12 ATTN, 1 FC	79.3	166

Attack Configuration. We leverage the open-source implementation[1] of the state-of-the-art bit-flip attack [154] to evaluate our detection. The attack batch size on convolutional neural networks is set to 128 and 64 for CIFAR10 and ImageNet benchmarks, respectively. Throughout the experiments, we repeat the attack 50 times with different initial random seeds for each of our DNN benchmarks and report the average obtained results. Each attack round consists of multiple iterations where one bit is flipped at each step. The iterations conclude once the DNN test accuracy falls below the random guess threshold, i.e., 10% and 0.1% for CIFAR10 and ImageNet, respectively. Due to the robustness of Transformer models to bit-flips, we consider the attack successful once the accuracy of the model falls below 0.2%. Table 3.3 encloses the

[1]Available at https://github.com/elliothe/BFA

average number of bit flips required for attacking the benchmarked 8-bit quantized DNNs.

Metrics. We use two evaluation metrics to quantify AccHashtag detection. Firstly, we define Detection Rate (DR) as the ratio of models under attack which are correctly detected by AccHashtag, as formulated in Equation (3.9).

$$DR = \frac{\text{\# of attacked models correctly detected}}{\text{Total \# of attack rounds}} \tag{3.9}$$

Secondly, we use the False Positive Rate (FPR) as the ratio of benign models mistaken for being malicious, i.e., containing a bit-flip that results in a hash mismatch.

3.4.2 Analysis of Design Choices

In this section, we perform an ablation study to analyze the characteristics of the bit-flip attack. We experiment with three victim DNNs with various types of (linear) layers, e.g., convolution, fully-connected, and self-attention. Specifically, we benchmark ResNet20 trained on CIFAR10 and ResNet18 and ViT trained on ImageNet. The weights in each victim DNN are quantized using a range of bitwidths. The minimum evaluated bitwidth is selected such that the classification accuracy is within 1%, 2%, and 3% of the floating-point accuracy for ResNet20, ResNet18, and ViT, respectively. For each configuration, we perform 50 runs of the bit-flip attack with different random seeds to ensure we capture the variances in the outcome. We summarize our findings below:

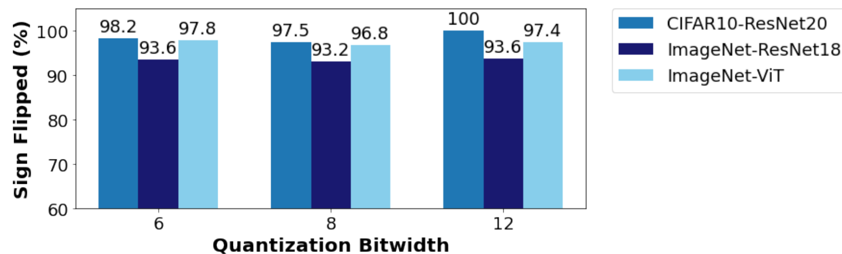

Figure 3.5. Percentage of sign changes occurring during multiple runs of the bit-flip attack. The progressive bit-flip attack [154] changes the sign of the target parameter with high probability.

Sign Change. Figure 3.5 demonstrates the percentage of bit flips resulting in a sign change

across various attack configurations. The consistent pattern among all experiments indicates that the attack significantly favors changing the sign of the target parameter. This is intuitive as flipping the sign of the underlying weight parameter can induce a dramatic change in the output of the layer. Commensurate with this finding, ACCHASHTAG sensitivity analysis models the effect of attack as a change in the underlying parameter's sign (See Equation (3.5)).

Sensitivity Computation. We quantify the per-layer vulnerability to bit-flips by averaging the sensitivities of γ most vulnerable weights enclosed in each layer. Figure 3.6 shows the effect of various γ values on ranking DNN layers in terms of their sensitivity. On the vertical axis, the layers in each model are ordered based on their sensitivity, where a higher rank corresponds to higher sensitivity. As highlighted with the green boxes, the ordering amongst most sensitive layers remain largely the same when $\gamma \leq 10$. This is intuitive as a higher γ includes weights in the sensitivity analysis that are not prone to bit-flips, while a lower γ ensures a more targeted sensitivity analysis only for the most vulnerable parameters. For the benchmarked Transformer models, due to their inherent robustness to faults, the number of bit-flips required to reduce the accuracy is extremely large (see Table 3.3). For convolution-based benchmarks, however, accuracy can be downgraded with very few bit-flips. For such models, we observe that while the attack could target different or same weights within a certain layer, on average, the same layer is not targeted more than ~ 5 times. To investigate the per-layer attack concentration, we count the number of times each layer is targeted during one execution of the attack. Figure 3.7 shows the maximum number of bit flips occurring per layer, averaged across different attack runs for two representative convolutional neural networks. Using the insights from attack concentration and the analysis in Figure 3.6, we quantify the sensitivity of each layer as the average over its $\gamma = 5$ most sensitive weights.

Dataset Size. We leverage a small held-out validation dataset to compute the sensitivity scores for model weights. In this section, we investigate the effect of validation dataset size, controlled by the number of held-out samples per class (n), on the sensitivity analysis. Figure 3.8 demonstrates the ranking of model layers in terms of sensitivity, shown for different n, where a higher rank

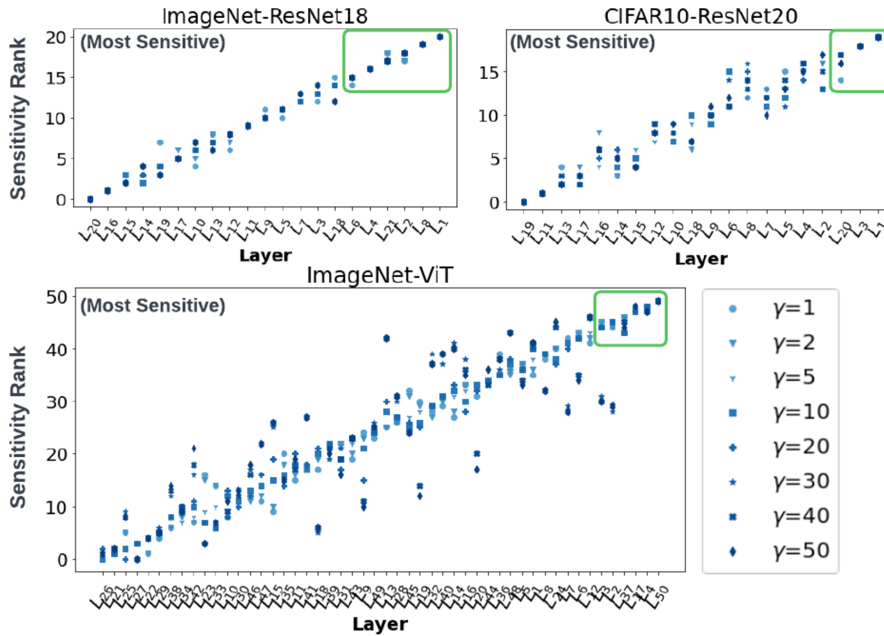

Figure 3.6. Ranking DNN layers based on their vulnerability to bit-flips, defined as the average sensitivity score assigned to their γ most vulnerable weights. The most vulnerable layers (marked with green boxes), remain largely the same, when $\gamma \leq 10$.

on the vertical axis corresponds to higher sensitivity. As shown, for ImageNet benchmarks, the ranking variance is very small and the sensitivity analysis delivers consistent results even in the extreme case of $n = 1$. For CIFAR10 dataset, the sensitivity analysis is more affected by n. This is due to the small number of classes in this dataset, which results in a very small validation dataset for small n. We show the detection rate of ACCHASHTAG versus various number of checkpoints in Figure 3.9. As seen for the CIFAR10 benchmark and the extreme case of $n = 1$, more checkpoints are needed to obtain 100% detection rate. However, for $n \geq 5$, hashing only the two most sensitive layers can achieve perfect detection. For the ImageNet benchmark, $n \geq 2$ can provide 100% detection with two checkpoints. Our analysis shows an intrinsic trade-off between validation dataset size and number of checkpoint layers. When data is scarce, the sensitivity analysis may be affected, thus more checkpoint layers are needed to ensure detection.

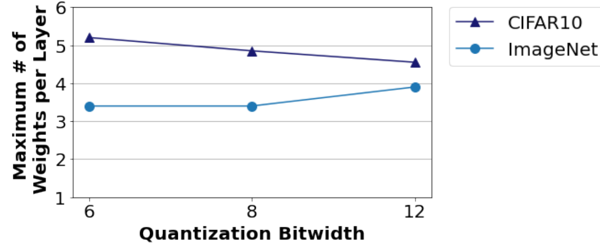

Figure 3.7. Maximum per-layer attack concentration, averaged across multiple runs for ResNet20 and ResNet18 trained on CIFAR10 and ImageNet, respectively. The progressive bit-flip attack [154] on average targets the weights in the same layer no more than ~ 5 times.

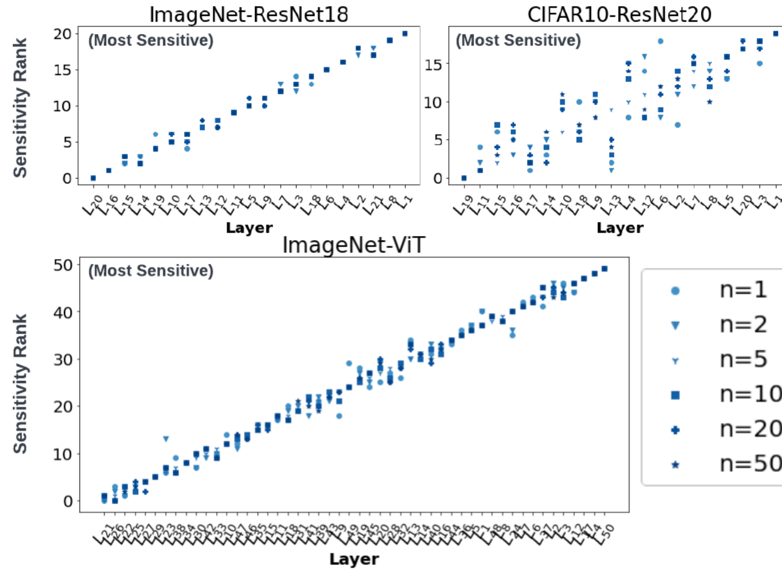

Figure 3.8. Ranking DNN layers based on their sensitivity, computed using a validation dataset with n samples per class.

3.4.3 AccHashtag Performance

Sensitivity Analysis

In this section, we showcase the stand-alone performance of AccHashtag sensitivity analysis. We benchmark the ResNet20 model on CIFAR10 to evaluate the effectiveness of our proposed method in finding the vulnerable layers within a DNN. Figure 3.10 demonstrates the sensitivity score assigned to each layer of the model versus the number of per-layer bit flips occurring across 50 runs of the attack. All values are normalized by the total summation.

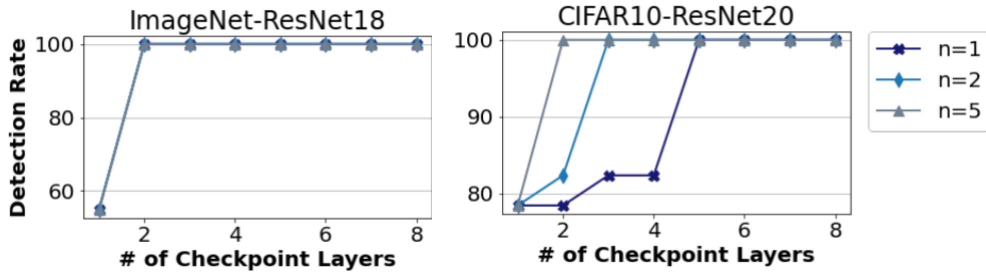

Figure 3.9. ACCHASHTAG detection rate versus number of checkpoint layers, shown for various number of per-class samples (n) used in sensitivity calculation. We omit the plot for ViT for brevity as it shows a similar trend as the ResNet18 benchmark.

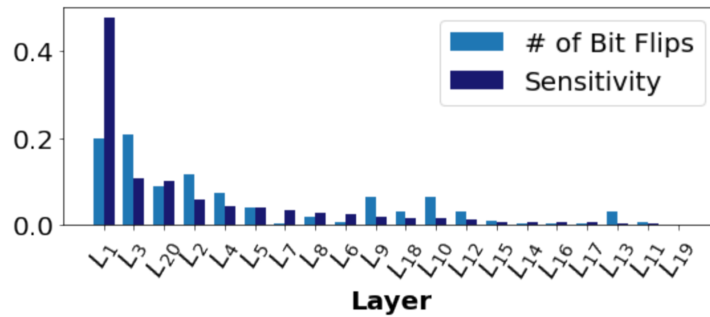

Figure 3.10. Per-layer sensitivity scores assigned by ACCHASHTAG versus the number of per-layer bit-flips. All values are normalized and sum to 1. Results are gathered across 50 runs of the bit-flip attack on the ResNet20 DNN trained with CIFAR10 dataset.

As seen, there exists a correlation between the sensitivity score and the number of times the pertinent layer has been subject to attack; most attacks occur in layers 1, 7 which are also the most sensitive layers found by ACCHASHTAG. Below, we provide a thorough evaluation of end-to-end ACCHASHTAG execution.

Detection Performance

We leverage our sensitivity analysis to rank DNN layers in the order of their attack vulnerability. The top-k most sensitive layers are then selected as checkpoints to extract hashes during the pre-processing and online phases. During online execution, if there exists *at least one* hash mismatch with the ground-truth signature among DNN layers, ACCHASHTAG marks the model as malicious. Figures 3.11 and 3.12 demonstrates the detection performance of

AccHashtag versus the number of checkpoint layers for various DNN benchmarks. For this experiment, all evaluated models are quantized with 8-bit parameters.

AccHashtag achieves a 100% attack detection rate with very few checkpoints. For the CIFAR10 benchmarks, AccHashtag detects faulty DNNs with only 1 and 2 checkpoints on the VGG11 and ResNet20 architectures, respectively. For ImageNet, AccHashtag achieves a perfect detection rate on AlexNet with only 1 checkpoint. On the more complex architectures ResNet18 and ResNet34, AccHashtag achieves 100% detection with only 2 and 3 checkpoints. For the most complex convolution neural network (CNN) benchmark, i.e., MobileNetV2 with 53 convolution and fully-connected layers, AccHashtag achieves 96.2% detection rate with 3 checkpoints and reaches perfect accuracy with 5. We further show the effectiveness of AccHashtag fault detection for large-scale Transformer-based models in Figure 3.12. As shown, AccHashtag successfully locates the sensitive layers that are most prone to bit-flips among more than 50 layers in the Transformer benchmarks. As such, our defense can detect the occurrence of faults with 100% using only 1 and 2 checkpoint layers for the DeiT and ViT models, respectively.

The results demonstrate AccHashtag's ability to correctly find the most vulnerable DNN layers and detect fault-injections using hash signatures. Note that AccHashtag has an FPR=0.0%, i.e., it never mistakes benign layers for attacked ones. This is due to the fact that the hash value is constant as long as the underlying layer parameters remain intact, i.e., in the

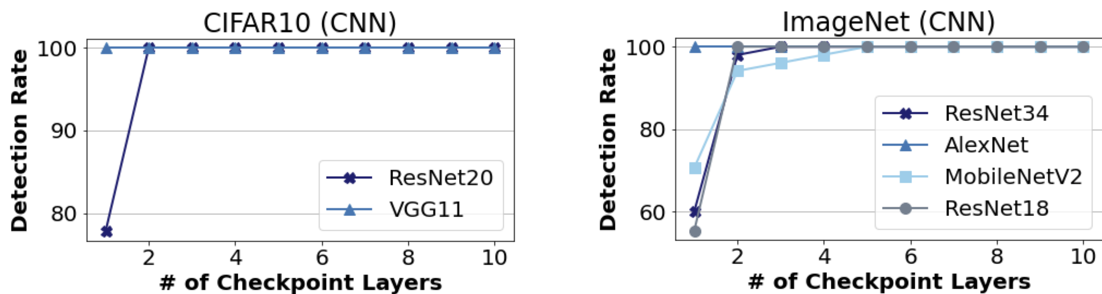

Figure 3.11. AccHashtag detection rate versus the number of checkpoint layers used for signature extraction, evaluated on different victim CNNs.

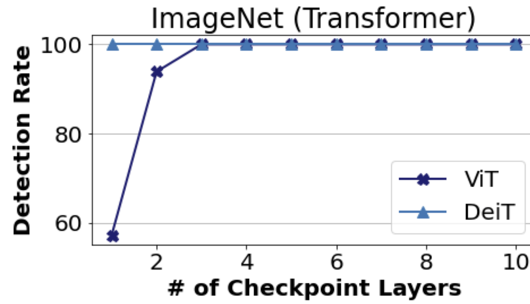

Figure 3.12. ACCHASHTAG detection performance versus number of checkpoint layers. Evaluated DNNs are derived from the Transformer backend with self-attention layers.

absence of bit flips.

Effect of Bitwidth. We benchmark ResNet20 and ResNet18 trained on CIFAR10 and ImageNet, respectively, and sweep the quantization bitwidth of the victim DNN. Figure 3.13 demonstrates the effect of DNN bitwidth on ACCHASHTAG detection rate. While the bitwidth can affect the detection rate with only one checkpoint, it can be observed that ACCHASHTAG becomes agnostic to the underlying bitwidth with more than 2 checkpoints. For > 2 checkpoints, ACCHASHTAG consistently achieves a detection rate of 100%. The same trend can be observed for the Transformer-based ViT benchmark trained on ImageNet, where ACCHASHTAG consistently achieves 100% detection rate when more than 2 (sensitive) layers are checked. the ability to maintain the detection rate in face of different quantization bitwidths allows ACCHASHTAG to be globally applicable to various DNN configurations employed in embedded applications.

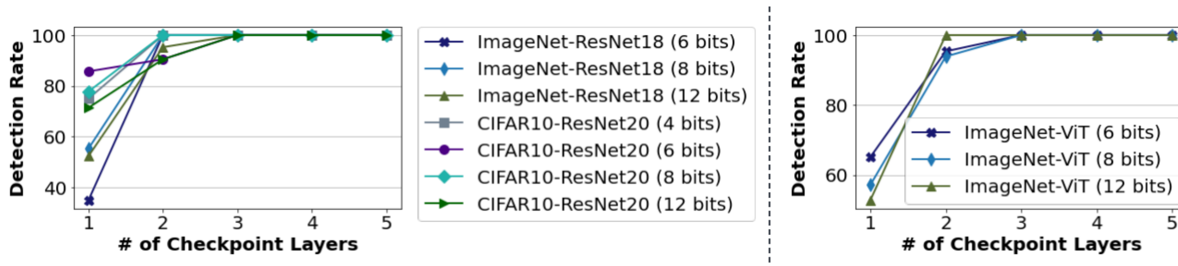

Figure 3.13. Effect of victim DNN's bitwidth on ACCHASHTAG detection rate. The legend presents the utilized datasets along with the underlying bitwidths.

Comparison with prior work

We compare AccHashtag with the best prior work, i.e., WED [119] and RADAR [111] in terms of detection performance and overhead. We baseline the best reported results in the original papers, i.e., the WED(2) configuration from [119], and $G = 8$ and $G = 512$ with interleaving for ResNet20 and ResNet18 from [111]. We devise two configurations for AccHashtag to enable on-par comparison with each of the prior work as follows.

Similar to AccHashtag, the proposed method in [119] checkpoints a subset of DNN layers to detect malicious models. Therefore, for best comparison with this work, we evaluate AccHashtag with the number of checkpoints set to the minimum value required to obtain 100% detection rate (see Figure 3.11). We call this configuration `Cfg-1`. The method in [111], however, checkpoints all layers within the DNN and reports the performance as the total number of detected bit flips. Therefore, to compare with this work, we devise `Cfg-2`, where the number of checkpoints is selected such that all bit flips are detected. For `Cfg-2`, we set the number of checkpoints to 7 and 8 for ResNet18 and ResNet20, respectively.

The comparison results are summarized in Table 3.4. As seen, AccHashtag provides state-of-the-art detection performance at a fraction of the storage/computation cost compared to best prior works. Compared to WED [119], AccHashtag significantly reduces the false-positive rate and achieves 100% detection rate with $FPR = 0.0\%$. Additionally, AccHashtag incurs $20 - 400\times$ lower storage footprint. Compared to RADAR [111], AccHashtag detects all bit flips within the model with 100% accuracy while incurring $3 - 4\times$ lower storage cost. We further compare AccHashtag runtime with RADAR [111]. We measure our runtime on an ARM Cortex-A57 embedded CPU. For a fair comparison, we report the normalized runtimes, i.e., relative to the inference time of the victim DNN on the target hardware. As seen, AccHashtag achieves $175 - 183\times$ faster runtime compared to [111].

We would like to emphasize that unlike [111], AccHashtag detection does not rely on the number of detected bit flips. Therefore, the setup in `Cfg-2` is purely for comparison

purposes. The most representative metric for evaluating AccHashtag is the detection rate corresponding to `Cfg-1`, as explained in Section 3.4.1, Equation (3.9).

Table 3.4. Comparison with best prior works WED [119] and RADAR [111]. Runtime is measured on an ARM CPU and normalized by the inference time of the victim DNN.

Benchmark	Work	Detection (%)	FPR (%)	Detection Overhead Storage (KB)	Detection Overhead Runtime (%)
ResNet20	WED	96	12	47	N/A
	RADAR	97.5	0	8.2	5.27
	`Cfg-1`	**100**	**0**	**0.5**	**0.01**
	`Cfg-2`	**100**	**0**	**2.1**	**0.03**
ResNet18	RADAR	96.2	0	5.6	1.83
	`Cfg-2`	**100**	**0**	**1.8**	**0.01**
ResNet34	WED	100	4	302	N/A
	`Cfg-1`	**100**	**0**	**0.8**	$<$ **0.01**
MobileNet	WED	100	6	26	N/A
	`Cfg-1`	**100**	**0**	**1.3**	$<$ **0.01**

Storage and Computation Overhead

Below we provide a more detailed analysis of the storage and runtime specifications of AccHashtag detection. AccHashtag storage and computation are linear in the number of checkpoint layers: we compute and store an 8-bit secret hash per checkpoint layer. In addition, the per-layer Pearson Pearson hash tables each incur a storage cost of $256B$. The Pearson hash tables can be reused among layers, however, here we report the maximum required storage, i.e., when utilizing a unique hash table per checkpoint layer. For l checkpoint layers, the storage overhead of AccHashtag is, therefore, $\mathcal{O}(257 \times l)B$.

To showcase the efficiency of AccHashtag detection, we measure the runtime on both an embedded Cortex-A57 CPU and an FPGA. We develop and optimize the 8-bit Pearson hash in C, which is then invoked during DNN execution to detect bit flips. We further synthesize our FPGA cores for AccHashtag on the Xilinx VCU108 development board. We utilize Vivado High-Level Synthesis to realize our FPGA design. The FPGA accelerator operates with a clock cycle of 2ns. As a baseline, we report the inference time of the victim DNN. For our CNN-based benchmarks, we report the runtimes on an embedded Jetson TX2 board which includes an ARM

Cortex-A57 CPU and an NVIDIA Pascal embedded GPU. For the large-scale Transformer-based benchmarks, we report their runtime on a server-grade Intel Xeon E5-2609 CPU and the Nvidia TITAN Xp GPU. The victim DNN is implemented and executed via PyTorch library.

Table 3.5 encloses the runtime and storage of ACCHASHTAG across different benchmarks. We report ACCHASHTAG's storage as the percentage of the memory (in Bytes) required by the victim DNN's weights. The number of checkpoints is set to the minimum value required for a 100% detection rate from Figure 3.11.

As evident from Table 3.5, ACCHASHTAG delivers perfect detection performance while incurring a negligible storage and computation cost, making it suitable for real-time embedded DNN applications. Additionally, by leveraging our accelerated hash core on FPGA, we relieve the host CPU of hash computation. ACCHASHTAG FPGA modules checkpoint the sensitive layers in parallel to DNN inference, enabling 1.5-$2.6\times$ faster hash generation compared to CPU.

Table 3.5. ACCHASHTAG overhead analysis. Here, $\#$ is the number of checkpoint layers.

Type	Benchmark	#	DNN Inference (ms)		CPU Detection		FPGA Detection
			CPU	GPU	Storage (%)	Time (ms)	Time (ms)
CNN	VGG11	1	1698.4	110.7	3e-3	0.009	0.003
	ResNet20	2	654.8	59.4	2e-2	0.012	0.005
	AlexNet	1	7957.9	240.7	4e-4	0.928	0.614
	ResNet18	2	20938.8	198.5	4e-3	0.066	0.035
	ResNet34	3	40870.6	229.7	3e-3	1.889	1.059
	MobileNet	5	2313.6	182.2	4e-2	0.020	0.007
Transformer	ViT	3	196.9	19.1	3e-3	4.692	3.127
	DeiT	1	181.3	19.5	1e-3	1.768	1.179

As discussed in Section 3.3.4, the hash computation is overlapped with the read latency of the hash input from AXI. To balance the latency bottleneck between these two stages, we define a design hyperparameter dubbed $TILE$, which corresponds to the length of the burst reads from AXI. Figure 3.14 demonstrates the relationship between design throughput (measured in number of hash computations per seconds) and $TILE$ length. Using a small $TILE$ will cause the memory reads to become the latency bottleneck in the design. By increasing the $TILE$ value, we increase the compute capacity to match the AXI read latency, thereby increasing the overall

system throughput. We empirically found $TILE = 1024$ to provide a suitable balance between AXI reads and hash computation as shown in Figure 3.14.

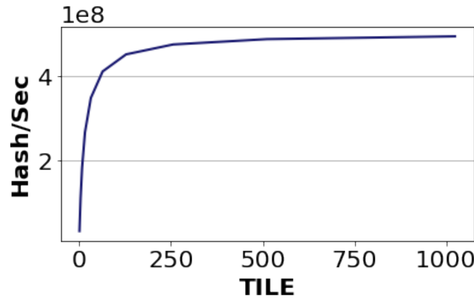

Figure 3.14. System throughput as a function of the design parameter $TILE$, i.e., the burst length for AXI reads. Higher $TILE$ length facilitates larger overlap between CPU-FPGA communications and hash computation, thus increasing throughput.

3.5 Conclusion

ACCHASHTAG is a highly accurate methodology for online detection of fault-injection attacks in DNN parameters. The core idea in ACCHASHTAG is to extract a ground-truth signature from the benign model which is then used for verification at inference time. We extract the signatures by encoding DNN layer weights using a low-collision hash function. To minimize detection overhead, we only extract the hashes from a subset of DNN layers where the probability of attack occurrence is high. Towards this goal, ACCHASHTAG is equipped with a novel sensitivity analysis that quantifies the vulnerability of DNN layers to bit-flip attacks. ACCHASHTAG detection strategy provides several benefits: (1) it delivers 100% detection rate with 0 false alarms across a variety of benchmarks. (2) The proposed detection is backed up by provable performance guarantees that provide a lower bound on the detection rate. (3) ACCHASHTAG incurs negligible storage and runtime overhead, enabling accurate fault detection on resource-constrained embedded devices. Our lightweight method and realistic threat model make ACCHASHTAG an attractive candidate for practical deployment. Our thorough evaluations show ACCHASHTAG's competitive advantage in terms of attack detection and execution overhead.

3.6 Acknowledgements

Chapter 3 is a reprint of the material as it appears in: M. Javaheripi, J. Chang, and F. Koushanfar, “AccHashtag: Accelerated Hashing for Detecting Fault-Injection Attacks on Embedded Neural Networks”, in *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 2022. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Ensuring DL Robustness to Backdoor Attacks

With the growing popularity of AI-powered autonomous systems, the demand for superior intelligence has led to increasingly more complex model development processes. Training contemporary deep learning models requires massive datasets and high-end hardware platforms [87, 130]. Amid this trend, clients rely on third party databases and/or major cloud providers to build their models. Unfortunately, outsourcing of content or computations opens up new challenges as it extends the potential attack surface to malicious third party entities [162]. This chapter focuses on Trojan attacks [58, 121], where the malicious third party provider inserts a hidden Trojan trigger, also dubbed a "backdoor", inside the model during training. During inference, the attacker can hijack the model prediction by inserting the Trojan trigger inside the input data. Figure 4.1 illustrates examples of Neural Trojans.

Identification and mitigation of Trojans is particularly challenging for the clients since the compromised model performs as expected on their benign data, i.e., when the Trojan is not activated. To tackle Trojan attacks, contemporary research proposes either reverse-engineering the trigger pattern from the model [27, 60, 120, 200] or identifying the presence of a trigger at the input [29, 36, 52]. The former class of methods require time-consuming reverse-engineering and retraining. The latter approaches induce a high overhead on DNN inference that hinders their applicability to embedded systems. To ensure model robustness in safety-sensitive autonomous

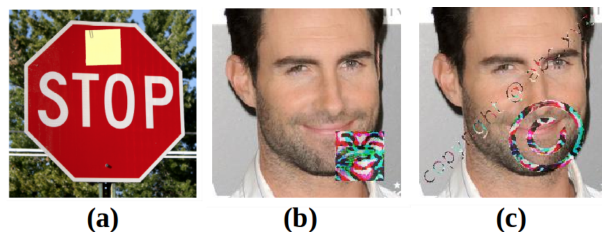

Figure 4.1. Example Trojans: (a) BadNets [58] with a sticky note and TrojanNN [121] with (b) square and (c) watermark triggers.

systems, it is crucial to augment the models with an online Trojan mitigation strategy. To the best of our knowledge, none of the earlier works provide the needed lightweight defense strategy.

We propose CLEANN, the first end-to-end accelerated framework that enables real-time Trojan shield for embedded DNN applications. CLEANN's lightweight method is devised based on algorithm/hardware co-design; our algorithmic insights offer a highly accurate and low-overhead method in terms of both the offline defense establishment and online execution; our hardware accelerator enables low-latency and energy-efficient defense execution on embedded platforms. CLEANN harvests the irregular patterns caused by Trojan triggers in the input space and/or the latent feature-maps of the victim DNN to detect adversaries. Our method leverages key concepts and theoretical bounds from sparse approximation [37] to learn dictionaries that absorb the distribution of the benign data. We then utilize the reconstruction error obtained from the sparse approximation to characterize the benign space and identify the Trojans.

To ensure applicability to various attacks and trigger patterns, CLEANN sparse recovery acts on both frequency and spatial domains. Our proposed defense is compatible with the challenging threat model in which the attacker has full control over the geometry, location, and content of the Trojan trigger. The contaminated model is shipped to the client, who is unaware of the existence of the Trojan and does not have access to any labeled data. CLEANN countermeasure is unsupervised, meaning that no labeled training data or contaminated Trojan sample is required to establish the defense. Notably, CLEANN is the first defense to recover the ground-truth labels of Trojan data without performing any model training and/or fine-tuning.

We validate the effectiveness of CLEANN by performing extensive experiments on various state-of-the-art Trojan attacks reported to-date. CLEANN outperforms prior art both in terms of Trojan resiliency and algorithm execution overhead. CLEANN brings down the attack success rate to 0% for a variety of physical [58] and complex digital [121] attacks with minimal drop in classification accuracy. Our customized accelerated defense shows orders of magnitude higher throughput and performance-per-watt compared to commodity hardware. In brief, the contributions of CLEANN are as follows:

- Introducing CLEANN, the first end-to-end accelerated framework for online detection of Neural Trojans in embedded applications.
- Constructing a novel unsupervised Trojan detection scheme based on sparse recovery and outlier detection. The proposed lightweight defense is, to our best knowledge, the first to enable recovering the original label of Trojan samples without model fine-tuning/training.
- Providing bounds on detection false positive rate using the theoretical ground of sparse approximation and outlier detection.
- Devising the first customized library of Trojan shields on field-programmable gate arrays (FPGAs) which enables high-throughput and low-energy Trojan mitigation.

4.1 Background on Trojan Attacks and Defenses

Attacks. Contemporary Trojan attacks on DNN classifiers fall under two general categories, i.e., physical versus digital attacks, depending on the format for the Trojan trigger. BadNets [58] is an example attack methodology that can be realized via physical Trojans. In this attack, a subset of the training data is poisoned with the Trojan trigger and relabeled as the attack target class. A trigger-activated backdoor is then embedded in the model after training on the poisoned dataset. In this attack, the Trojan trigger can be arbitrarily chosen by the attacker. As such, real-world physical objects can be leveraged as the Trojan trigger, e.g., a sticky note on a stop sign as

shown in Figure 4.1-a. Alternatively, if the attacker does not have access to the training data, the backdoor can be embedded by maliciously altering the victim model's weights. TrojanNN [121] attack is conducted by targeting a few neurons in the victim model and reverse-engineering a trigger such that the selected neurons are activated. The model is then trained to predict the attacker's desired class once the specific pattern of targeted neurons is activated. In TrojanNN, the attacker cannot arbitrarily choose the trigger pattern, rather the pattern is automatically derived in a digital format, e.g., the square and watermark noise patterns in Figure 4.1-c,d. Compared to the BadNets physical attack, TrojanNN's complex digital triggers are harder to mitigate as shown in prior work [27, 123].

Defenses. Prior work proposes robust learning methods that can detect malicious samples during training [23, 116, 192]. However, if a DNN has already been infected, the authors of [118] suggest pruning to remove the Trojans, but at the cost of reducing the model's accuracy. In contrast to these works, CLEANN assumes the user does not have access to the training data and can not perform extensive model retraining, which renders our defense more efficient. A number of other techniques detect backdoor attacks in DNNs by reverse-engineering the trigger. For example, Neural Cleanse [123] can uncover Trojan triggers without access to the training data. Subsequent work improves the efficiency [27] and quality [60] of trigger reverse-engineering. However, these methods can struggle when dealing with more complex triggers, such as those created by TrojanNN [121]. Instead of reverse-engineering the trigger, CLEANN analyzes the statistics of sparse representations derived from benign samples and detects any abnormal samples during inference. This allows us to identify even complex Trojan triggers without prior knowledge of the attack algorithm. Furthermore, our method is more efficient as it does not require costly reverse-engineering and can be run in real-time on embedded hardware.

Another set of prior defenses examine input data to detect the presence of Trojan triggers. For instance, authors of [25] cluster the latent features of the infected model to identify benign versus Trojan data. Similarly, NIC [127] compares incoming samples to benign and Trojan latent features to detect adversaries. However, these methods require access to the poisoned

training data and its labels, which may not be feasible in real-world scenarios. Sentinet [29] uses gradient information to extract critical regions, including the Trojan trigger, from the input data. Februus [36] uses a similar approach along with a generative adversarial network (GAN) to inpaint Trojan triggers. However, this approach requires a large number of data samples for GAN training. STRIP [52] runs the model multiple times on each image with intentionally injected noise to detect Trojans. Although these works have high detection accuracy, the computational cost of multiple forward and backward passes through the DNN makes them unsuitable for real-time execution. CLEANN, on the other hand, offers better detection accuracy with low computational complexity and requires only a small number of benign samples, making it suitable for real-time deployment in embedded systems.

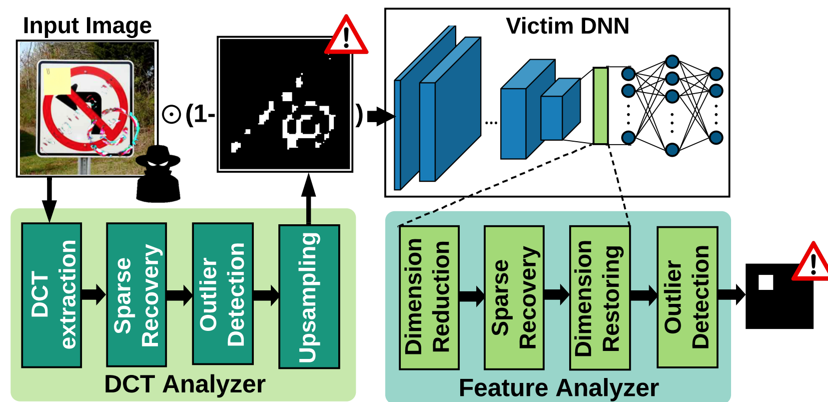

Figure 4.2. High-level overview of CLEANN Trojan detection methodology. CLEANN detects both digital and physical attacks using a pair of input and latent feature analyzers.

4.2 CLEANN Methodology

Figure 4.2 illustrates the high-level flow of CLEANN methodology for Trojan detection. CLEANN comprises two core modules, dubbed the DCT and feature analyzers, specializing in the characterization of the DNN input space and latent representations, respectively. By aggregating the decision of the two analyzers, CLEANN is able to thwart a wide range of physical and digital Trojan attacks.

❶ **DCT Analyzer.** The DCT analyzer acts as an image preprocessing step. This module investigates all incoming samples in the frequency domain in search for suspicious frequency components that are anomalous in clean data. Towards this goal, we design four components for this module as shown in Figure 4.2. First, the Discrete Cosine Transform (DCT) extraction module transforms the input image to the frequency domain. We then perform sparse recovery on the extracted frequency components and reconstruct the signal using a sparse approximation. The outlier detection module uses a concentration inequality to detect anomalous reconstruction errors and generate a binary mask with non-zero values denoting the potential Trojan-carrying regions. The anomalous regions in the input image are then suppressed by the binary mask before entering the victim DNN. To ensure compatible dimensions between the input image and the binary mask, a nearest neighbor upsampling component is also included inside the DCT analyzer. Frequency analysis is particularly useful for detecting digital Trojans. However, physical attacks, e.g., the sticky note in Figure 4.2, might evade frequency-domain detection.

❷ **Feature Analyzer.** This module investigates patterns in the latent features extracted by the victim DNN to find abnormal structures. The feature analyzer is placed at the penultimate layer inside the victim DNN. This choice of location allows us to leverage all the visual information extracted from the input image by the DNN for making the classification decision. The sparse recovery module in the feature analyzer serves two purposes: (i) denoising input features for use in the remaining layers of the victim DNN, (ii) anomaly detection on the reconstruction errors for distinguishing Trojans. Notably, the first property allows CLEANN to recover the ground-truth labels for Trojan samples by effective removal of Trojan triggers. To ensure scalability to various output dimensions, we include a dimension reduction module that adaptively adjusts the feature size while maximally preserving the informative content of the signals. To allow the reconstructed output to flow in the remaining layers of the DNN, a twin dimension restoring layer recovers the original tensor shape. The extracted distributions from latent layers successfully detect attacks in the physical domain.

4.2.1 Defense Construction and Execution

CLEANN consists of two main phases to mitigate Trojan attacks:

► **Offline Preprocessing.** During this phase, we learn the parameters for dictionary-based sparse recovery and outlier detection modules by leveraging a small set of unlabeled benign samples. Our methodology is entirely unsupervised, meaning no Trojan data is involved in defense construction. This, in turn, ensures applicability to a wide range of Trojan patterns and attacks. CLEANN pre-processing phase is low-complexity as it does not involve any training or fine-tuning of the victim DNN. We only perform this step once for each (model, dataset) pair. The learned analyzer modules can then be transferred to a variety of attacks without any fine-tuning overhead.

► **Online Execution.** CLEANN methodology is devised based on light-weight solutions to enable efficient adoption in embedded systems. We provide a hardware-accelerated pipeline for end-to-end execution of CLEANN where the analyzer modules are either integrated inside the victim DNN or running in parallel with it. The DCT extraction and upsampling components are implemented as an additional convolution layer at the input of the victim DNN. We devise a customized library for implementing the sparse recovery, outlier detection, and dimensionality reduction and restoring modules on FPGA. These FPGA-accelerated modules are executed synchronously with the victim DNN to raise alarm flags for Trojans.

4.2.2 Threat Model

In our threat model, we assume the client has purchased the trained DNN model infected with Trojans from a malicious party. Accordingly, we consider the following constraints on our defense strategy: (1) The client has access to model weights but not the training data. (2) The client has access to clean test data but they are unlabeled. (3) The client is not aware whether or not the model is infected with Trojans. (4) No prior knowledge is available about possible Trojan trigger shapes and/or patterns.

To construct the defense, we assume access to a small corpus of *unlabeled* data[1]. This is a realistic assumption as access to small amounts of data is possible via online resources. For instance, publicly available repositories enable data generation through generative networks for Faces[2]. We consider the most generic and challenging form of Trojan attacks in which the attacker can control the trigger size, shape, and content. CLEANN mitigation is made possible in such scenario by constructing the defense using benign unlabeled data.

4.3 CLEANN Components

4.3.1 DCT extraction

In natural images, most of the energy is contained in low frequencies. However, this property does not necessarily hold true for the Trojan triggers. Figure 4.3 shows the visualization of the frequency components for a Trojan sample, normalized by the magnitude of frequency components for benign data. Here, the magnitudes are averaged across $100,000$ image patches and the Trojan samples contain a watermark trigger generated by [121]. As seen, Trojans have much larger components in the high-frequency domain compared to benign samples.

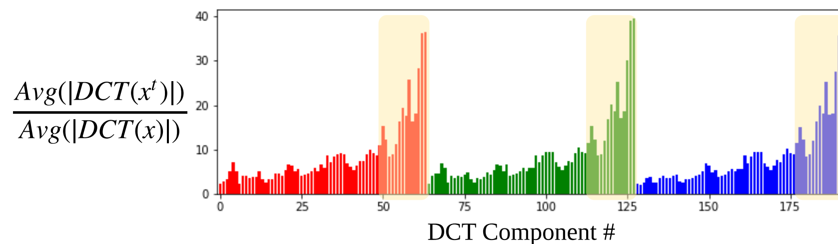

Figure 4.3. Average magnitude of DCT components for Trojan samples, normalized by benign data, shown in the three RGB channels. Trojans contain abnormally larger amounts of high-frequency components (highlighted regions).

To perform frequency analysis, we divide each input image into non-overlapping patches[3]

[1]less than 1% of the training set size across all of our evaluations

[2]http://www.whichfaceisreal.com/index.php

[3]We use $P = 4$ for small image benchmarks where input image dimensions are less than 32 pixels. For larger input image sizes we use $P = 8$.

of size $P \times P$. We transform each image patch to another patch of same size in the frequency domain using DCT. Equation (4.1) encloses the formula used to compute the DCT transformation $F_{u,v}$ of a $P \times P$ patch.

$$F_{u,v} = C_{u,v} \sum_{i=0}^{P-1} \sum_{j=0}^{P-1} x_{i,j} \cos\left[\frac{u\,\pi}{P}\left(i + \frac{1}{2}\right)\right] \cos\left[\frac{v\,\pi}{P}\left(j + \frac{1}{2}\right)\right] \tag{4.1}$$

Here, $x_{i,j}$ is the input pixel located at the (i, j) coordinate and $C_{u,v}$ is a scalar constant that depends on the frequency coordinates. The extracted $2D$ DCT components $F_{u,v}$ are then sorted in decreasing order in terms of the information they carry following a zigzag pattern [167].

We represent the DCT transform as a group convolution with kernel size P and c_{in} groups where $c_{in} = 3$ and 1 for RGB and gray-scale images, respectively. The kernel weights of the convolution layer are initialized with the DCT basis coefficients which are pre-computed based on Equation (4.1). The stride of the convolution is set to P to account for image patching. Such representation allows for an efficient implementation of the DCT Analyzer, which can be easily integrated into the architecture of the victim model as a pre-processing layer.

4.3.2 Sparse Recovery

Sparse coding is referred to learning methods where the goal is to efficiently represent the data using sets of over-complete bases. Given a matrix of (n) data observations $X \in \mathbb{R}^{l \times n}$, sparse coding extracts a dictionary of normalized basis vectors $D \in \mathbb{R}^{l \times m}$ and the sparse representation matrix $V \in \mathbb{R}^{m \times n}$. Formally, the sparse coding objective can be written as:

$$\underset{D,V}{min}\, f_D(X) = \underset{D,V}{min} \|X - D.V\|_2 + \gamma \|V\|_0 \tag{4.2}$$

where γ is a regularization coefficient that promotes sparsity in the coded representation V. Dictionary learning algorithms provide solutions to the above optimization problem by find-

ing a dictionary D that minimizes $\mathbb{E}_{x \sim \mathcal{X}} f_D(x)$, where $\mathcal{X}$ is the distribution over the inputs. CLEANN extracts D by performing dictionary learning over legitimate (benign) data. The out-of-distribution Trojan samples are thus expected to show a high reconstruction error, whereas benign samples will be accurately reconstructed with small error.

Figure 4.4 illustrates this behavior in an example $2D$ space. The light-blue dots represent the distribution of benign samples; the two solid arrows $\vec{d_1}$ and $\vec{d_2}$ are the dictionary atoms and only one of them is used for sparse reconstruction $\widetilde{\vec{x}}$. As seen, the magnitude of the reconstruction error on the outlier sample $\vec{x}_2$ is larger than that of regular data $\vec{x}_1$, i.e., $\|\vec{x}_2 - \widetilde{\vec{x}}_2\|_F >> \|\vec{x}_1 - \widetilde{\vec{x}}_1\|_F$ where $\| \cdot \|_F$ is the Frobenius norm.

While the above simple illustration shows the effectiveness of dictionary learning in 2 dimensions, a similar behavior is observed when generalizing sparse coding to higher dimensions. For a dictionary trained on n samples $x \sim \mathcal{X}$, there exist theoretical bounds on the generalization error for unseen samples drawn from the same distribution $\mathcal{X}$. Let us denote the average reconstruction error over the set of n observed samples by E_o. The generalization error of the dictionary $E_D(\cdot)$ on unseen samples $x_u \sim X$ is bounded by $E_D(x_u) \leq E_o + \delta$. Vainsencher et al. [195] prove that the generalization error δ for a λ-sparse representation is $\mathcal{O}(\sqrt{ml \ln(n\lambda)/n})$ under some orthogonality assumptions for the dictionary. CLEANN dictionaries are devised to minimize reconstruction error on benign samples. We therefore carefully tune the dictionary size m and sparsity level λ to ensure a low reconstruction error on the data at hand (E_o) as well as a low error bound δ.

► **Data.** We apply sparse recovery on two data subsets extracted from a small corpus of randomly selected benign samples.

1. At the input of the neural network (Section 4.3.1), each column of matrix X is the DCT of a single patch in the input image. For instance, for an 8×8, DCT window, the dimensionality would be $l = 3 \times 64$ (64 DCT coefficients per RGB channel).

2. At the latent space, each column of X represents a flattened feature-map with reduced

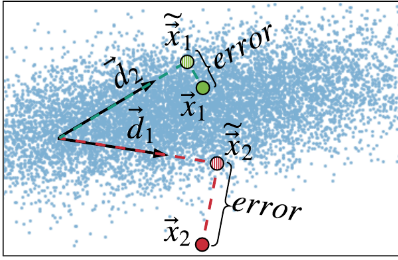

Figure 4.4. Illustration of sparse reconstruction for regular data (green circle) and out-of-distribution samples (red circle).

dimensionality.

► **Dictionary Learning.** We use an adaptive sampling distribution based on the reconstruction error of X, dubbed Column Selection-based Sparse Decomposition (CSSD) [138] for learning the dicrionaries. This algorithm initializes D by a small random subset of X and then iteratively adds columns to D; the probability of a data sample being appended at each step is proportional to its reconstruction error with the current column set. Formally, the probability of the i-th sample x_i being selected at the $(t+1)$-th iteration is given by:

$$p(i) \propto \frac{\|D_t D_t^{+} x_i - x_i\|_2}{\|x_i\|_2} \tag{4.3}$$

where D_t corresponds to the columns of the dictionary selected up to the t-th iteration and $D_t^{+} = (D_t^T D_t)^{-1} D_t^T$ is the pseudo inverse of D_t. The intuition behind Equation (4.3) is to give a higher chance of selection to those elements of X with higher reconstruction errors. This approach allows us to maximize the amount of embedded information from the data distribution inside D. While more sophisticated algorithms can be used [3,4,43], our empirical evaluations show that CSSD can sufficiently express the data distribution with minimal generalization error.

► **Reconstruction Algorithm.** We use Orthogonal Matching Pursuit (OMP) [33] for sparse recovery as summarized in Algorithm 1 in Chapter 2. OMP iteratively finds non-zero elements to construct the sparse representation V. The added non-zero element at each iteration is chosen

such that it minimizes the L_2 norm of the remaining residual error $\|r_{i-1} - \Lambda_i \cdot V\|_2$ which can be solved using Least-square (LS) optimization. The subset of dictionary columns (Λ_i) that contribute to the sparse recovery is also expanded over iterations. Finally, the reconstruction can be obtained as $\widetilde{\vec{x}} = \Lambda_k \cdot V$ where k is the sparsity level.

► **Distribution Learning with Few Samples.** An "over-complete" dictionary is necessary to ensure representation sparsity [138] and effective separation of outlier and benign samples. The term over-complete is used when the number of columns in the dictionary is higher than the data dimensionality ($m >> l$). In real-world DNN applications, however, the number of data samples (m) is often small while the feature-map dimensionality (l) is large. To tackle this, we apply Singular Value Decomposition on the high-dimensional feature-maps to reduce l. Inverse SVD can then be applied on the reconstructed output to recover the original dimensionality. We choose the SVD rank such that more than 90% of the original energy is preserved.

4.3.3 Detection

Figure 4.5-a, b shows example Trojan data together and the corresponding reconstruction error heat maps. As seen, areas of the image convered with the Trojan trigger have relatively higher reconstruction error compared to the rest of the benign regions. Inspired by this observation, We use the multivariate extension of Chebyshev's inequality [182] to capture the behavior of benign data and mark outliers as Trojans. Let us denote by $\{\vec{x}_i\}_{i=1}^{N}$ the reconstruction errors obtained from benign data using the sparse recovery in Section 4.3.2. Given the empirical mean $\vec{\mu}$ and the covariance Σ of the observed error values, the distance of new error values from the (benign) distribution can be formulated as:

$$dist(\vec{x}) = (\vec{x} - \vec{\mu})\Sigma^{-1}(\vec{x} - \vec{\mu})^T \tag{4.4}$$

The Chebyshev's inequality provides an upper bound on the probability of the above distance becoming greater than an arbitrary value ϵ as follows:

$$\mathcal{P}(dist \geq \epsilon^2) \leq min\left\{1, \frac{d(N^2 - 1 + N\epsilon^2)}{N^2\epsilon^2}\right\} \tag{4.5}$$

Using the above inequality, we can deduce that samples with a large enough ϵ are out-of-distribution, i.e., Trojan. Figure 4.5-c visualizes the binary output of the outlier detection where zeros and ones denote in-distribution and outlier values of the reconstruction error, respectively. As seen, parts of the input image that are covered with the Trojan trigger are correctly distinguished from benign regions.

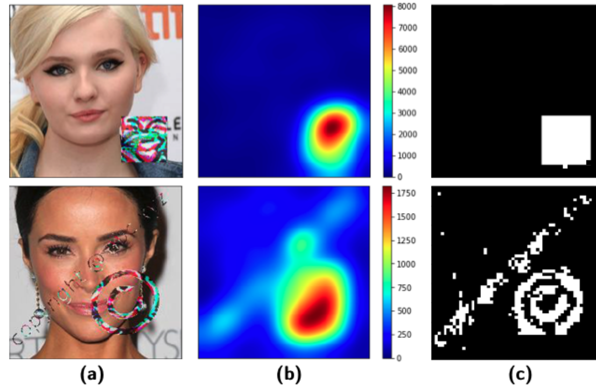

Figure 4.5. (a) Example Trojan data with watermark and square triggers [121], (b) reconstruction error heatmap, and (c) output mask from the outlier detection module.

An incoming sample $I \in \mathbb{R}^{d \times K \times K}$ is labeled as Trojan if at least one of its enclosing components $I_k \in \mathbb{R}^d$ is categorized as an outlier using Equation (4.5) and a given ϵ threshold. The probability of a sample being categorized as Trojan is therefore:

$$\mathcal{P}_I(Trojan) = 1 - \prod_{k=1}^{K \times K} \mathcal{P}_{I_k}(Benign) \tag{4.6}$$

Using the Chebyshev's inequality in Equation (4.5), the probability of a benign component being correctly categorized as benign for a given ϵ tends to $\mathcal{P}_{I_k}(Benign|I_k \in Benign) \geq 1 - \frac{d}{\epsilon^2}$ as $N \rightarrow \infty$. The probability of the corresponding full sample I being wrongfully labeled

as Trojan, i.e., the false positive rate (FPR), is thus upper-bounded by:

$$FPR = \mathcal{P}_I(Trojan|I \in Benign) \leq 1 - \Big(1 - \frac{d}{\epsilon^2}\Big)^{K\times K} \tag{4.7}$$

We can therefore determine the parameter ϵ based on any target FPR:

$$\sup_{\epsilon} FPR = 1 - \Big(1 - \frac{d}{\epsilon^2}\Big)^{K\times K} \leq FPR_{target} \tag{4.8}$$

$$\Rightarrow \frac{d}{\epsilon^2} \leq 1 - \sqrt[K\times K]{1 - FPR_{target}} \tag{4.9}$$

4.4 CLEANN Hardware

In the following, we delineate the hardware architecture of CLEANN components that enable a high throughput and low energy execution.

► **Matrix-Vector Multiplication Core.** Many of the fundamental operations performed in CLEANN include matrix-vector multiplication (MVM). In particular, the outlier detection module requires two MVMs to calculate the distance function shown in Equation (4.4). Additionally, the dimensionality reduction and restoring components in the feature analyzer are realized using MVMs with weight matrices $W \in \mathbb{R}^{l\times r}$ and $W \in \mathbb{R}^{r\times l}$, respectively, where l is the dimensionality of the input and r is the SVD rank. We devise an FPGA core for MVM and vector addition, realized using DSP blocks with Multiplication Accumulation (MAC) functionality [84, 169]. Figure 4.6 presents the high-level schematic of CLEANN vector-matrix multiplication.

We provide two levels of parallelism in our design controlled by parameters P and SIMD in figure (4.6). This approach allows our design to achieve maximum resource utilization and throughput on various FPGA platforms. The weight matrix is divided into subsets of length P and fed into parallel processing elements (PEs). These subsets are read from DRAM using a Ping-Pong weight buffer to overlap memory reads with PE computations. At each cycle, PEs perform partial dot-product on the fetched weight and input partitions of length $SIMD$; the

same input partition is shared across all PEs. We devise a tree-based reduction module and an accumulator to enable summation of partial dot-product outputs.

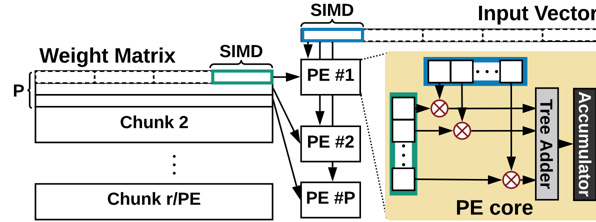

Figure 4.6. Schematic of CLEANN MVM core with its internal parallelization levels.

► **Sparse Recovery Core.** The sparse recovery module performs OMP to reconstruct input signals. We provide a reconfigurable and scalable OMP core on FPGA to accelerate sparse recovery. OMP relies on sequential execution of three steps: (1) The dictionary column with the maximum dot-product with the current residual vector is selected. (2) An LS optimization step generates the sparse representation of the current residual vector with the columns of the dictionary selected so far. (3) The residual is updated based on the new sparse representation and the selected dictionary columns.

We utilize CLEANN MVM core to implement the first step above. For the second step, we implement the LS optimization using a QR factorization of the dictionary matrix. We leverage the modified Gram Schmidt (MGS) method [55] to perform the factorization. Since the dictionary matrix expands by one column each iteration, it is not necessary to recompute the Q and R matrices very time. Instead, we iteratively form the Q and R matrices as outlined in Algorithm 2 of Chapter 2 and compute the residual update using Equation (2.12). Due to the low memory footprint of CLEANN components, we store all required data in the available on-chip Block RAMs. By eliminating the overhead of external memory access, CLEANN enjoys a low latency and high power efficiency.

4.5 Experiments

We evaluate CLEANN on three visual classification datasets of varying size and complexity, namely, MNIST [107] for handwritten digits, GTSRB [181] for road signs, and VGGFace [150] for face data. The number of classes for each dataset is 10, 43, and 2622, respectively. We corroborate CLEANN effectiveness against variations of two available state-of-the-art Neural Trojan attacks. In what follows, we provide detailed performance analysis and comparisons with prior art. We further demonstrate CLEANN accelerated execution on embedded hardware.

4.5.1 Attack Configuration

Throughout the experiments, we consider input-agnostic Trojans where adding the trigger to any image causes misclassification to the attack target class. Table 4.1 summarizes the evaluated benchmarks along with their corresponding Trojan attacks and triggers.

► **BadNets.** We implement the BadNets [58] attack with various triggers as an example of a realistic physical attacks. The injected Trojans include a white square and a Firefox logo placed at the bottom right corner of the input image. We embed the backdoor by injecting $\sim 10\%$ poisoned data samples during training.

► **TrojanNN.** We evaluate CLEANN against TrojanNN [121] as a digital attack with complex triggers. The attack is implemented using the open-source models shared by TrojanNN authors[4]. We perform experiments with two variants of TrojanNN triggers, namely, square and watermark, crafted for the VGGFace dataset.

Table 4.1. Evaluated datasets and attack algorithms.

Dataset	Input Size	Architecture	Attack	Trigger
MNIST	1x28x28	2CONV, 2MP, 2FC	BadNets	square
GTSRB	3x32x32	6CONV, 3MP, 2FC	BadNets	square Firefox
VGGFace	3x224x224	13CONV, 5MP, 3FC	TrojanNN	square watermark

[4] https://github.com/PurduePAML/TrojanNN

4.5.2 Detection Performance

We apply CLEANN Trojan mitigation at the input and latent space of infected DNNs. To create the defense, we separate 500, 430, and 2622 clean samples from MNIST, GTSRB, and VGGFace test sets, respectively. The aforementioned size for the benign dataset corresponds to 1% of the training data size for MNIST and GTSRB and 0.1% VGGFace training data. Such low data size requirements provide a competitive advantage for CLEANN defense in real-world scenarios. We summarize other defense parameters for our evaluated benchmarks in Table 4.2. These parameters are selected to maintain a high classification accuracy over the benign data.

Table 4.2. Parameters of CLEANN modules for various datasets. P: DCT windows size, l: feature size for sparse recovery, m : number of dictionary columns for sparse recovery, λ: sparsity parameter in sparse recovery, ϵ^2: distance threshold for outlier detection.

Dataset	Trigger	Input Analyzer					Feature Analyzer			
		P	l	m	λ	ϵ^2	l	m	λ	ϵ^2
MNIST	Square	4	48	1000	5	5×10^{-4}	279	500	80	2×10^{-3}
GTSRB	Square	4	48	1000	5	5×10^{-4}	85	420	80	3×10^{-3}
	FireFox								50	1×10^{-2}
VGGFace	Square	8	192	1000	5	5×10^{-4}	520	2622	80	1×10^{-4}
	Watermark					8×10^{-4}			80	1×10^{-4}

We evaluate CLEANN Trojan resiliency on physical and digital attacks in Table 4.3. Specifically, under "Defended Model", we evaluate the drop in clean data accuracy (ACC↓), the attack success rate (ASR), and Trojan ground-truth label recovery (TGR). In addition to our results, we include prior art performance in terms of the above-mentioned criteria. On MNIST, CLEANN achieves 0% ASR, with only 0.1% drop in clean data accuracy, outperforming the prior art. For GTSRB, CLEANN achieves an ASR of 0% and a lower drop of accuracy compared to all prior work, except for Deep Inspect, which suffers from a much higher ASR of 8.8%.

On digital attacks, CLEANN achieves 0.0% ASR with only 0.8% and 2.0% degradation of accuracy for square and watermark shapes. The watermark trigger covers a large area of the input image, obstructing the critical features. As such, while CLEANN detects the Trojan with high success, it shows a lower TGR compared to our other triggers. Note that Neural Cleanse and

Deep Inspect perform DNN training on synthetic datasets achieved with model inversion [48]. As a result, their post-defense accuracy is not directly comparable with CLEANN, which does not perform DNN retraining. We emphasize that while such retraining contributes to accuracy, it may not be feasible in real-world applications.

Table 4.3. Evaluation of CLEANN on various physical and digital attacks. Comparisons with state-of-the-art prior works, i.e., Neural Cleanse(NC) [200], Deep Inspect (DI) [27], Februus [36], and SentiNet [29] are provided where applicable.

Dataset	Trigger	Work	Retrain	Infected Model		Defended Model		
				ACC-C	ASR	ACC↓	ASR	TGR
MNIST (Physical Attack)	Square (4×4)	NC	yes	98.5	99.9	0.8	0.6	NA
		DI	yes	98.8	100.0	0.7	8.8	NA
		CLEANN	no	99.3	100.0	**0.1**	**0.0**	**98.7**
GTSRB (Physical Attack)	Square (4×4)	NC	yes	96.5	97.4	3.6	0.1	NA
		DI	yes	96.1	98.9	**-1.0**	8.8	NA
		Februus	yes*	96.8	100	1.2	**0.0**	**96.5**
		CLEANN	no	96.5	99.4	0.0	**0.0**	94.7
	Firefox (6×6)	CLEANN	no	92.6	99.8	0.4	1.7	83.5
VGGFACE (Digital Attack)	Square (59×59)	NC	yes	70.8	99.9	**-8.4**	3.7	NA
		DI	yes	70.8	99.9	0.7	9.7	NA
		SentiNet‡	no	NA	96.5	NA	0.8	NA
		CLEANN	no	74.9	93.52	0.8	**0.0**	**70.1**
	Watermark	NC	yes	71.4	97.60	**-7.4**	**0.0**	NA
		DI	yes	71.4	97.60	0.5	8.9	NA
		CLEANN	no	74.9	58.6	2.0	**0.0**	**41.38**

* Februus performs GAN training. † SentiNet only reports results on LFW [78] dataset.

► **Sensitivity to Trigger Size.** We perform experiments on the GTSRB dataset with a square Trojan trigger and change the trigger size such that it covers between $\sim 0.4\%$ to $\sim 14\%$ of the input image area. The size range is chosen to ensure that the corresponding triggers are viable in real settings and provide a high ASR. We summarize the obtained results in Figure 4.7. CLEANN significantly reduces the ASR while enabling recovery of ground-truth labels with a high accuracy across all trigger sizes. This is expected since CLEANN does not rely on the trigger size to construct the defense. For average sized Trojans, CLEANN successfully detects the existence of triggers and reduces the ASR to less than 1%. For larger trigger sizes, the TGR is relatively lower since the Trojan occludes the main objects in the image.

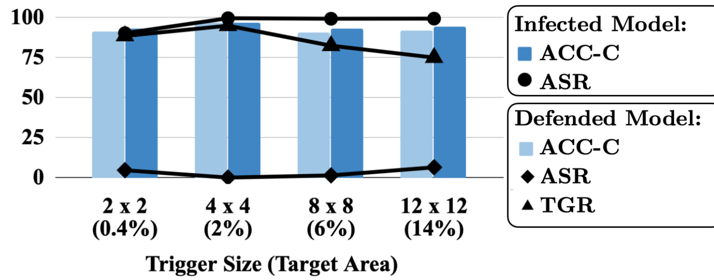

Figure 4.7. Analysis of CLEANN sensitivity to Trojan trigger size.

► **Offline Preprocessing Overhead.** The preparation of CLEANN defensive modules consists of the following steps:

- DCT extraction and dictionary leaning on benign inputs.
- Computing $\vec{\mu}$ and Σ for input outlier detection.
- Computing SVD and dictionary learning at latent feature maps.
- Computing $\vec{\mu}$ and Σ for latent outlier detection.

In practice, the above computation incurs negligible runtime compared to DNN training. We implement the above steps in PyTorch and measure the runtime on an NVIDIA TITAN Xp GPU. For our GTSRB benchmark, the above operations require 0.06, 0.19, 10.47, and 0.1 seconds, respectively. The defense construction time is therefore ~ 11 seconds which is $\sim 1.8\%$ of the time required to train the victim DNN on this benchmark. For the more complex VGGFace dataset, the above operations require 1.05, 0.54, 48.3, and 1.2 seconds, respectively, resulting in a total of ~ 51 seconds for defense preparation.

4.5.3 Hardware performance

We implement the proposed Trojan defense strategy on various hardware platforms and compare the performance of CLEANN components. The evaluated platforms include server-grade CPUs and GPUs, embedded CPUs and GPUs, and FPGA. We base our comparisons on

performance-per-Watt defined as the throughput over the total power consumed by the system. This measure effectively encapsulates two major performance metrics for embedded applications. Throughout this section, we will target our study on the GTSRB benchmark but similar trends are observed for other datasets.

► **Performance on General Purpose Hardware.** We provide an optimized software library for CLEANN defense components in Python. In order to benefit from highly optimized backend compilers for tensor operations on CPU and GPU, our codes are developed on top of the PyTorch deep learning library. Our provided software library can be readily instantiated within PyTorch API to enable simultaneous DNN execution and Trojan defense. We implement our defense pipeline on the *Jetson TX2* embedded development board running in CPU-GPU and CPU-only modes. We further run the defense on a server-grade *Intel Xeon E5* CPU and an *NVIDIA TITAN Xp* GPU. The overall achieved defense throughput with a batch size of 1 ranges from 11 fps on the embedded CPU up to 28 fps on the server GPU.

Figure 4.8 illustrates the runtime breakdown for various components of CLEANN running on each platform. Here, the sparse recovery and outlier detection modules are abbreviated as `SR` and `OLD` and the prefixes `D-` and `F-` correspond to the DCT and feature analyzers, respectively. The experiments are performed using a batch size of 1 to resemble real-world applications and runtimes are averaged across 100 runs. For each platform, we normalize the runtime of each component by the total defense execution time for one sample. As seen, the bulk of defense runtime belongs to the sparse recovery module. This is due to the inherently sequential nature of the OMP algorithm performed inside this module. CPU and GPU platforms are designed to excel in massively parallel operations while this does not hold for OMP. This motivates us to design specialized hardware to accelerate the execution of CLEANN components on FPGA.

► **Performance on Customized Accelerator.** We implement CLEANN components on FPGA using the developed sparse recovery and `MVM` cores as the basic blocks. The design is developed in Vivado High-Level Synthesis and synthesized in Vivado Design Suite for the *Xilinx UltraScale VCU108* board. Power consumption is estimated during synthesis with Vivado Design Suite.

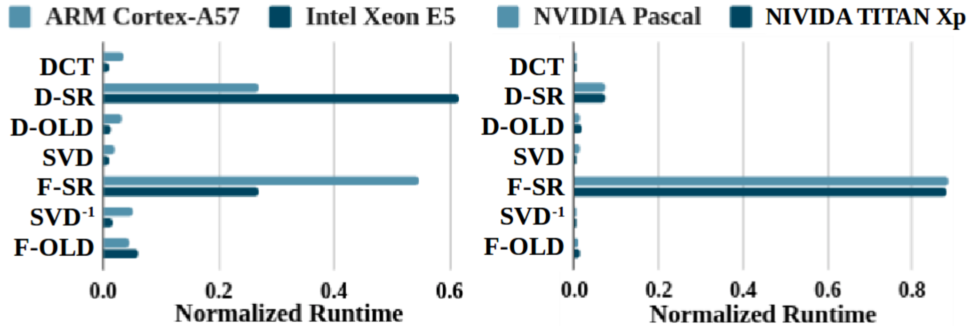

Figure 4.8. Latency breakdown of CLEANN components running on embedded and high-end CPUs (left) and GPUs (right).

Finally, a comprehensive timing and resource utilization analysis is performed. To maximize throughput, we tuned the parallelism factors in the MVM modules to the highest value such that the design fits within the available resources.

Figure 4.9 demonstrates the breakdown of execution cycles for CLEANN components. As seen, the sequential execution of the sparse recovery core accounts for the majority of computation cycles. Our FPGA-based sparse recovery core enjoys up to $10\times$ and $18\times$ faster execution, respectively, compared to their CPU and GPU counterparts. This is enabled by pipelined execution, fine-grained optimizations to data access patterns, and parallel computation.

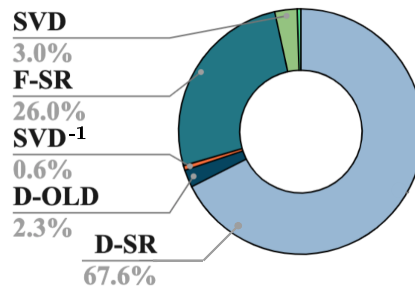

Figure 4.9. Cycle-count breakdown for running CLEANN components on FPGA.

We compare the performance-per-Watt and throughput of CLEANN on different hardware platforms in Figure 4.10. The performance-per-watt numbers are normalized by *TITAN Xp* and the throughput numbers are normalized by *ARM Cortex-A57*. As seen, the power-efficient implementation of CLEANN on FPGA not only enjoys a high throughput, but it also signifi-

cantly increases performance-per-watt compared to commodity hardware. Note that due to the lightweight nature of CLEANN defense strategy, the server-grade GPU performs poorly in terms of performance-per-watt compared to other platforms due to under-utilization and excessive power consumption.

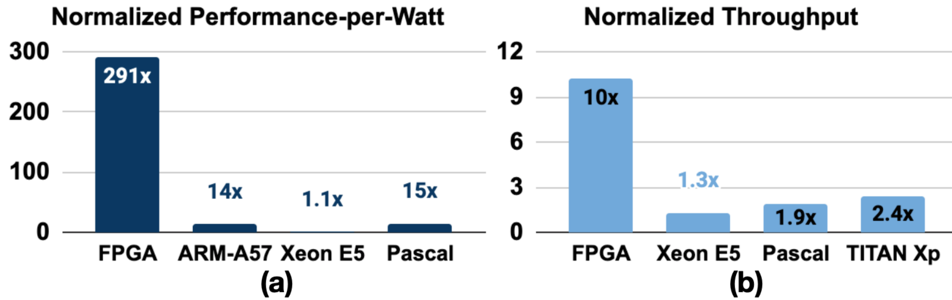

Figure 4.10. (a) Performance-per-Watt and (b) throughput across hardware platforms. Reported values for performance per-watt are normalized by *TITAN Xp* and throughput values are normalized by *ARM Cortex-A57*.

4.6 Conclusion

This chapter presents CLEANN, an end-to-end framework for online accelerated defense against Neural Trojans. The proposed defense strategy offers several intriguing properties: (1) The defense construction is entirely unsupervised and sample efficient, i.e., it does not require any labeled data and is established using a small clean dataset. (2) It is the first work to recover the original label of Trojan data without need for any fine-tuning or model training. (3) CLEANN provides theoretical bounds on the false positive rate. (4) The framework is devised based on algorithm/hardware co-design to enable accurate Trojan detection on resource-constrained embedded devices. We consider a challenging threat model where the attacker can use Trojan triggers with arbitrary shapes and patterns while no knowledge about the attack is available to the client. CLEANN light-weight defense and realistic threat model makes it an attractive candidate for practical deployment. Our extensive evaluations corroborate CLEANN's competitive advantage in terms of attack resiliency and execution overhead.

4.7 Acknowledgment

Chapter 4 is a partial reprint of the material as it appears in: M. Javaheripi, M. Samragh, G. Fields, T. Javidi, and F. Koushanfar, "CleaNN: Accelerated Trojan Shield for Embedded Neural Networks", in *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020. The dissertation author and Mohammad Samragh made equal contributions to this work.

Chapter 5

Improving Training Convergence via Architectural Modifications

DL models are increasingly popular for various automated learning tasks, particularly in visual computing applications. Recently, there has been a shift to incorporate DL training and execution on smart devices rather than offloading the computations to cloud-based servers. This transition is motivated by the compelling properties of on-device computation, e.g., preserving user data privacy and eliminating the need for continuous network connection. A standing challenge for on-device intelligence is the limited resources available on embedded devices that slow down DL execution compared to the cloud. The constraints of the embedded environment are specially critical for lengthy DL training. Contemporary DL models require high number of training iterations to converge, hindering their applicability to on-device *learning*. In this work, we focus on reducing the required training iterations for convergence, thereby paving the way for on-device learning applications such as federated learning and (local) personalization.

The literature in Neural Architecture Search (NAS) is primarily focused on generating compact and accurate DNNs for inference. To increase the search flexibility and reach a higher accuracy, a recent body of work in NAS explores the use of irregular wirings, aka bypass connections [77, 209, 213, 222]. These bypasses connect (non-consecutive) layers in the DL architecture that would otherwise be disconnected in a traditional DNN. While prior work in NAS can reduce the computational complexity of DNN inference, there has been little focus on

the training cost of the obtained DNNs for reaching their target accuracy. To enable on-device learning, we study irregular network wirings through the lens of DL training speed.

We propose a novel methodology that transforms the topology of conventional DNNs such that they reach an optimal cross-layer connectivity. This, in turn, significantly reduces the number of training iterations required for reaching a target accuracy. This transformation is based on our observation that the pertinent connectivity pattern highly impacts training speed and convergence. To ensure computational efficiency, our architectural modification takes place **prior** to training. Thus, the incorporated connectivity measure must be independent of network gradients/loss and training data. Towards this goal, we view DNNs as graphs and revisit Small-World Networks (SWNs) [206] from graph theory to transform DNNs into highly-connected small-world topologies. Watts-Strogatz SWNs [206] are widely used in the analysis of complex graphs; Due to SWNs' specific connection pattern, these structures provide theoretical guarantees for considerably decreased consensus times [146, 185, 228].

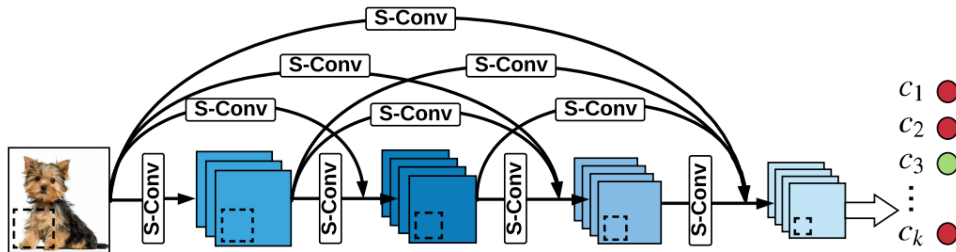

Figure 5.1. Schematic representation of the connections within a small-world DNN. An arbitrary neuron's output is connected to selected neurons in the proceeding layers via sparse connections (convolutions) denoted by *S-CONV*.

Our network modification algorithm takes as input a conventional DNN architecture and enforces the small-world property on its topology to generate a new network, called SWANN. We leverage a quantitative metric for *small-worldness* and devise a customized rewiring algorithm. Our algorithm restructures the inter-layer connections in the input DNN to find a topology that balances *regularity* and *randomness*, which is the key characteristic of SWNs [206]. Small-world property in DNNs translates to an architecture where all layers are interlinked via sparse

connections, an example of which is shown in Figure 5.1.

SWANNs have similar quality of prediction and number of trainable parameters as their baseline feed-forward architectures, but due to the added sparse links and the optimal SWN connectivity, they warrant better data flow. In summary, our architecture modification has three main properties: (i) It removes non-critical connections. (ii) It increases the degrees of freedom during training, allowing faster convergence. (iii) It provides customized data paths in the model for better cross-layer information propagation.

We conduct comprehensive experiments on various network architectures and showcase SWANNs' performance on popular image classification benchmarks including MNIST, CIFAR10, CIFAR100, and ImageNet. Our small-world DNNs achieve an average of 2.1-fold reduction in training iterations required to achieve comparable test accuracy as the baseline models. We further compare SWANN with the DenseNet model and show that with $10\times$ fewer parameters, SWANNs demonstrate identical performance during training. Finally, as a popular application of on-device learning, we benchmark SWANN in the federated learning scenario where multiple embedded devices collaboratively train a global model on their local datasets. In the federated scenario, SWANN reduces the number of (global) training iterations by $1.4\times$ on average, thereby reducing both the computation and communication in decentralized learning.

5.1 Background on Small-World Networks

Watts and Strogatz [206] observed that real-world complex networks, e.g., the anatomical connections in the brain and the neural network of animals, cannot be modeled using existing regular or random graph classes. As such, they introduced the new category of *small-world networks*. Members of the small-world class have two main characteristics: 1) They have a small average pairwise-distance between graph nodes. 2) Nodes within the graph exhibit a relatively high (local) clustered structure. The first property is mainly associated with random graphs while the second property is prominent in regular graphs. SWNs have shown significantly

enhanced signal propagation speed, consensus, synchronization, and computational capability [10, 103, 105, 183, 228].

Randomness is introduced into a regular graph structure by iterative removal and addition of edges with probability, p, in order to construct an SWN. Figure 5.2 demonstrates the transition between a regular graph and the corresponding random graph as the rewiring probability increases from 0 to 1. Intermediate values of p interpolate between complete regularity and randomness to generate an SWN.

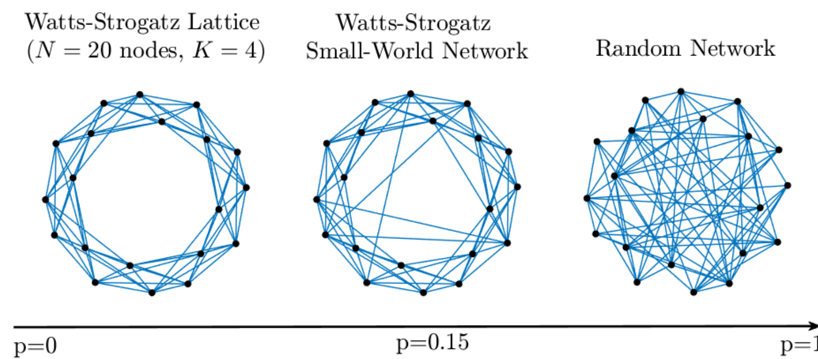

Figure 5.2. Transition of a regular graph to a completely random network. Intermediate values of the random rewiring probability, p, generate SWNs, i.e., clustered structures where any arbitrary node pair is connected by a few edges.

5.2 Related work

A line of research has focused on the addition of bypass connections to the DNN architecture to enhance inter-layer information flow and enable feature reuse. Perhaps the pioneer work is ResNet [66], which uses identity links (skip connections) to connect non-sequential layers. ResNet's skip connections follow a modular structure which results in redundancies since not all identity links are necessary as shown by [79]. DenseNets [77] are another example that use skip connections to connect each layer to all its preceding layers in a block. This is done by concatenating previous layers' feature-maps and using them as the input. Another work [76] argues that such dense connectivity incurs redundancies since earlier features might not be required in later layers. The authors of [76] prune the redundancies to generate a more

efficient architecture for the DNN inference phase.

Figure 5.3 illustrates the connection pattern in a ResNet, DenseNet, and SWANN architecture. In contrast to these two models, SWANN is not structured upon fixed building blocks and therefore can adapt to any given network architecture. Different from DenseNets which only accommodate fully dense connections, SWANNs leverage customized sparse convolutions. This sparsity enables selective connectivity between pairs of layers that enhance convergence speed while ensuring a low redundancy. Using sparse connections is explored in [99] where a trained DNN is pruned in a post-processing phase to reduce parameter count and improve inference performance. However, the pruning does not directly incorporate small-world characteristics and there is no analysis to show that the pruned networks are small-world. Additionally, the focus of [99] is on the inference phase and the pruning is performed after the DNN is trained. Rather, our approach is performed prior to DNN training with the goal of improving convergence. Finally, since the number of parameters is reduced in [99], the accuracy degrades compared to the baseline DNN. SWANN keeps the total number of parameters constant when converting the DNN to a small-world, thereby maintaining the accuracy.

Recent literature in NAS suggests exploration of irregularly wired DNN topologies [158, 186] and random connections [209, 213, 222] to obtain higher accuracies. Irregularly wired DNNs deviate from the regular DNN topology where the output of each layer is only fed to its immediately preceding layer. From the accuracy and computation perspective, the irregularly wired models have been shown to outperform regular DNNs for inference. The accuracy gains of random DNN connections has also been recently explored from a theoretical standpoint in [12]. Prior work also explores customized compilers and scheduling schemes for the irregularly wired networks to boost their execution performance on edge devices [5].

Perhaps the first investigation of SWNs in the context of machine learning was performed in [175], where small-scale Multilayer Perceptrons (MLPs) are transformed to a small-world graph. The paper shows that the small-world graph achieve lower error after the same total number of training iterations. Similarly, [44] transforms simple MLPs to small-world graphs and

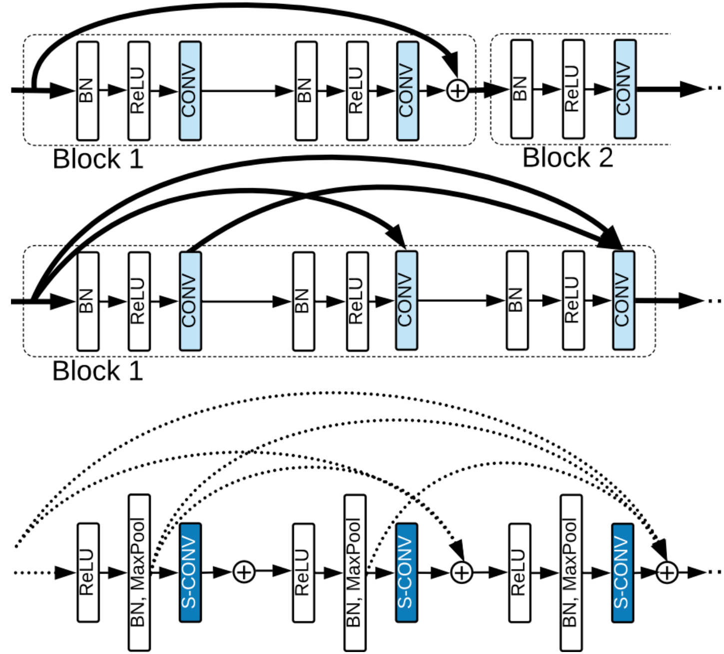

Figure 5.3. Information flow within a ResNet (top), DenseNet (middle), and SWANN (bottom). Here, *CONV*, *BN*, *ReLU* denote a convolution kernel, batch normalization, and non-linear activation, respectively, and our customized sparse convolutions are shown as *S-CONV*. Normal inter-layer connections are shown with bold lines and dotted lines are SWANNs' selective links.

study the accuracy benefits for diagnosis of diabetes. SWANN substantially differs from these works as our solution is applicable to contemporary convolutional neural networks containing various linear kernels and irregular long/short-range connections. Additionally, [44, 175] use a different mathematical model and metric for small-worldness. Authors of [211] randomly rewire MLPs to improve their function approximation capabilities. While the models are generated using random rewiring, there is no systematic choice of rewiring probability and no analysis of small-world characteristics for the developed models. Follow-up work studied the properties of randomly rewired DNNs [12, 213, 222] on classification accuracy, however, they did not specifically focus on small-world characteristics.

In summary, prior work mainly focuses on accuracy gains of long-range connections with little attention to the training process. To the best of our knowledge, SWANN is the first work to intertwine the small-world property with DNNs and examine the acquired networks in terms of both training convergence speed and accuracy.

5.3 SWANN: Small-World DNNs

We propose to restructure the inter-layer connections in a DL model such that its topology falls into the small-world category while the total number of parameters in the network is held constant. Throughout the text, we use the terms DL model and DNN interchangeably but emphasize that our approach is easily applicable to models without convolution layers, e.g., Multi-Layer Perceptrons (MLPs).

In the following, we first elaborate on the small-world criteria and introduce methods to distinguish SWNs from other topologies (Sec. 5.3.1). We then explain our conversion of an arbitrary DNN into its equivalent SWANN (Sec. 5.3.2). Lastly, we delineate our implementation and formalize the operations performed in an SWANN (Sec. 5.3.3).

5.3.1 Metric for Small-Worldness

To examine the small-world property for a given graph, we study two properties, namely, the *characteristic path length* (L) and the *global clustering coefficient* (C). For a given graph, L is calculated by taking the average of minimum path lengths over all node pairs. In this context, the minimum path length is equal to the smallest number of edges one must traverse to get from the first node to the second, or vice versa. The clustering coefficient C is a measure for the connection density between neighbors of any node in the network and is formulated as follows:

$$C = \sum_{i=1}^{V} C_i, \quad \text{where } C_i = \frac{E_i}{\frac{1}{2}N_i(N_i-1)} \tag{5.1}$$

Here, C_i denotes the local clustering coefficient of the ith node (v_i), E_i is the number of edges between neighbors of v_i, N_i is the number of neighbors of v_i, and V is the total number of nodes. As shown, the global clustering coefficient, C, is the mean of local coefficients. Regular lattices are highly clustered (large C) but have a very high L which conflicts with our desire to create bypass connections in the DNN. A completely random graph enjoys a small L but lacks

clustering. Small-world graphs strike a balance between randomness and regularity by having a large C and small L.

By definition, a graph is small-world if it has a similar L but higher C compared to an $Erdös - Re'nyi$ ($E - R$) random graph [233] constructed using the same number of nodes and edges. Let us denote the clustering coefficient and the characteristic path length of a given graph (G) by C_G and L_G, respectively. In a similar fashion, we represent the corresponding characteristics of the equivalent $E - R$ random graph by C_{rand}, L_{rand}. We use a quantitative measure of the small-world property form [83] which categorizes a network as an SWN if $S_G > 1$ where S_G is calculated using Equation (5.2).

$$S_G = \frac{\gamma_G}{\lambda_G}, \quad \gamma_G = \frac{C_G}{C_{rand}}, \quad \lambda_G = \frac{L_G}{L_{rand}} \tag{5.2}$$

5.3.2 Acquiring the Small-world Architecture

Graph Generation

In order to modify a given DNN architecture and generate the equivalent SWANN, we first model all connections within the network as a graph representation. In this context, a connection is defined as a linear operation performed between an input element and a trainable weight (network parameter) found in *Convolution* ($CONV$) and *Fully-Connected* (FC) layers. For $CONV$ layers, each feature-map channel is represented by one node and each edge represents a $k \times k$ kernel. For FC layers each neuron is assigned a separate node and the edges correspond to weight matrix elements.

Architecture Search

After generating the graph that corresponds to the input DNN architecture, we proceed to find the equivalent SWANN. To perform this task, the initial graph is randomly rewired with different probabilities, $p \in [0, 1]$, similar to Figure 5.2. For each rewired graph, we compute the characteristic path length L and clustering coefficient C and use the captured pattern for each

criterion to detect the small-world topology using the small-worldness measure in Equation (5.2).

► *Rewiring Policy.* Let us denote an edge with $e(v_i, v_j)$ where v_i and v_j are the start and end nodes. To perform random rewiring with probability p, we visit all edges in the graph once. Each edge is rewired with probability p or kept the same with probability $1-p$. If the edge needs to be rewired, a new second node $v_{j'}$ is randomly sampled from the set of nodes that are non-neighbor to the edge's start node, v_i. This second node is selected such that no self-loops or repeated links exist in the rewired graph. Once the destination node is chosen, the initial edge, $e(v_i, v_j)$ is removed and replaced by $e(v_i, v_{j'})$. Algorithm 3 summarizes the rewiring procedure performed on a baseline DNN to generate the rewired counterpart. Here, $\mathcal{N}$ is the total number of layers in the network, V_l denotes the nodes in the lth layer, and $E_{l,l+1}$ is the set of edges connecting neurons in layer l to its preceding layer $(l+1)$. Input layer is shown as $l=0$.

Algorithm 3. Random Rewiring Procedure

Inputs: Input DNN's graph G, rewiring probability p.
Output: Rewired network G_{rwd}.

1: $G_{rwd} \leftarrow G$
2: **for** $l = 0$ **to** $(\mathcal{N}-2)$ **do**
3:     **for** v_i **in** V_l **do**
4:         **for** v_j **in** V_{l+1} **do**
5:             **if** $|E_{l,l+1}| \geq 1$ **and** $e(v_i, v_j) \in G_{rwd}$ **then**
6:                 $r \sim \mathcal{U}[0,1]$
7:                 **if** $r \leq p$ **then**
8:                     $G_{rwd} \leftarrow \{G_{rwd} - e(v_i, v_j)\}$
9:                     **while** $e(v_i, v_{j'}) \in G_{rwd}$ **do**
10:                         $v_{j'} \sim \{V_{l+2} \cup \cdots \cup V_{\mathcal{N}}\}$
11:                     $G_{rwd} \leftarrow \{G_{rwd} + e(v_i, v_{j'})\}$

Figure 5.4 demonstrates the removal/addition of edges in the DNN architecture during our rewiring procedure. Note that our rewiring methodology does not alter the number of connections in the DNN. As a result, the total number of trainable parameters in the final obtained SWANN equals that of the original network.

► *Network Profiling.* Using the aforementioned rewiring policy, we generate various graphs by

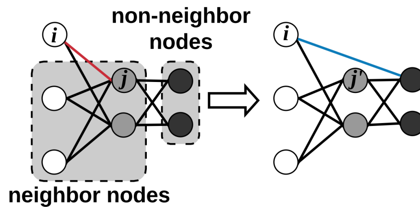

Figure 5.4. Our proposed rewiring algorithm replaces edges to the subsequent layer (red) with long-range edges (blue).

sweeping the rewiring probability in the [0,1] interval. Figure 5.5 demonstrates the correlation between C and L as the rewiring probability is changed for a 14-layer DNN. For conventional DNNs, the clustering coefficient is zero and the characteristic path length can be quite large specifically for very deep networks (leftmost points on Figure 5.5). As such, DNNs are far from networks with the small-world property. Random rewiring replaces short-range connections between subsequent layers with longer-range connections. Consequently, L is reduced while C increases as the network shifts towards its small-world equivalent. We select the topology with the maximum value of small-world property, S_G, as the SWANN. As a direct result of such architectural modification, the new network enjoys enhanced connectivity in its dataflow graph which results in better gradient propagation and training speedup.

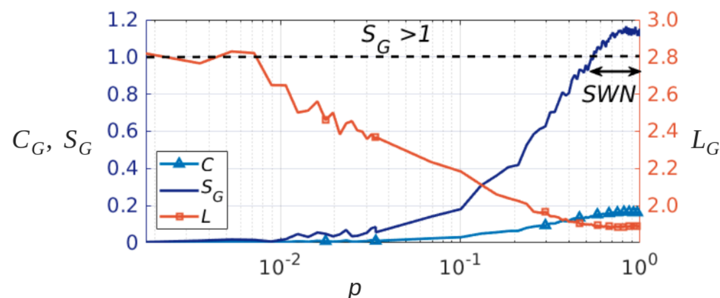

Figure 5.5. Clustering coefficient (C), small-world property (S_G), and path length (L) versus rewiring probability. The region where the graph transforms into a small-world network is shown with the double-headed arrow.

To compare the training convergence of SWANN with other configurations generated during the probability sweep, we train several rewired networks on the MNIST dataset [107],

each of which is constructed from a 5-layer DNN. Figure 5.6 demonstrates the convergence rate of these various architectures versus the rewiring probability p that is used to generate them from the baseline DNN. Due to the addition of long-range connections, all models show convergence improvements over the baseline. However, the perfect balance between node clustering and average path length is achieved by the SWN which leads to the fastest training convergence.

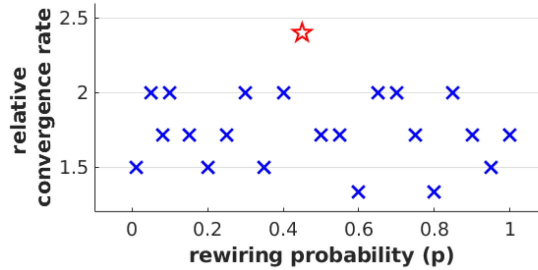

Figure 5.6. Convergence to 99.0% test accuracy for a 5-layer DNN and its randomly rewired counterparts trained on the MNIST dataset. Here, the relative convergence rate is computed as $\frac{e_b}{e_r}$, where e_b and e_r denote the number of training epochs required for the baseline and rewired models to reach the target accuracy, respectively. The SWN is shown with a red star.

5.3.3 SWANN Methodology

► *DNN Formulation.* Conventional DNNs are comprised of subsequent layers where each layer, l, in the network performs a combination of linear and nonlinear operations on its input, x_l, to generate the corresponding output, y_l. We denote core linear operations ($CONV$ and FC) in a DNN by $W_l(\cdot)$ with the subscript representing the layer index. Other operations can take the form of Batch Normalization (BN), $ReLU$ activation, and Pooling. For each linear layer, we bundle one or more of such operations together and show them as one composite function, $C_l(.)$. For an arbitrary layer l in a conventional DNN, the output is thus formalized as:

$$y_l = C_l(W_l(x_l)) \tag{5.3}$$

Note that the cascaded nature of DNNs implies that the generated output from one layer serves as the input to the immediately succeeding layer, i.e., $x_{l+1} = y_l$.

► *Sparse Connections in SWANNs.* One major difference between SWANNs and conventional DNNs is that SWANN layers can be interconnected regardless of their position in the network hierarchy. More specifically, the output of each layer of a SWANN is connected to all its succeeding layers via sparse weight tensors. These connections are implemented via convolution kernels with coarse-grained sparsity patterns.

By definition, *CONV* layers sweep a $k \times k$ convolution kernel across an input tensor of dimensionality $W_{in} \times H_{in} \times ch_1$ to generate an output feature map with dimensions $W_{out} \times H_{out} \times ch_2$ where ch_1 and ch_2 denote the number of input and output channels, respectively. In order to generate the graph-equivalent of such layer, we represent each $k \times k$ kernel by a single edge in the graph as shown in Figure 5.7. To remove each connection from the graph, we mask the corresponding $k \times k$ kernel to zero to generate a sparse weight tensor. Figure 5.8 shows the convolution filters of an example sparse connection from a layer with 5 output channels to a layer with 3 output channels and the corresponding small-world graph representation.

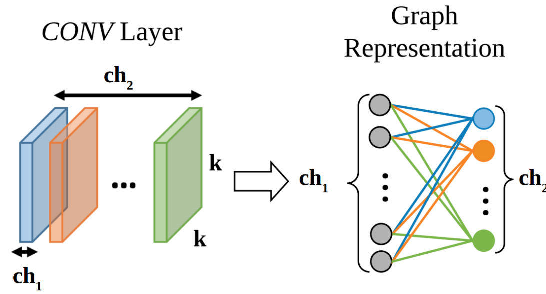

Figure 5.7. Conversion of a *CONV* layer to its graph representation. Each $k \times k$ convolution kernel is replaced by an edge in the corresponding graph where the input and output filter channels are shown as two consecutive rows of vertices with ch_1 and ch_2 nodes, respectively.

Let us denote sparse connections from layer l_1 to layer l_2 by $W^s_{l_1 l_2}(.)$. The output of the l-th layer in SWANN can then be calculated as:

$$y_l = C_l(W^s_l(x_l) + \sum_{l_1 < l-1} W^s_{l_1\ l}(y_{l_1})) \tag{5.4}$$

Comparing the above formulation with Equation (5.3), we highlight the extra summation term

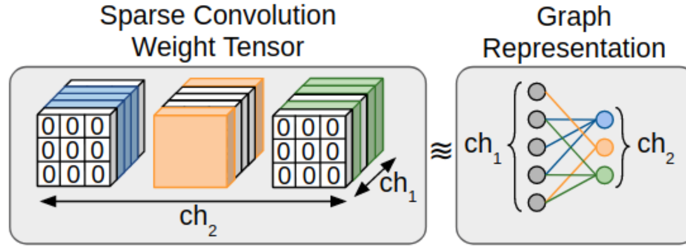

Figure 5.8. Coarse-grained sparse convolution between a layer with $ch_1 = 5$ output channels and a layer with $ch_2 = 3$ output channels. Left: Sparse convolution weights. For each removed connection from the graph, we show the corresponding filter in the sparse convolution weight by zero. The colored channels represent trainable DNN weights which can take on any arbitrary floating-point value. Right: Equivalent graph with nodes representing channels.

that accounts for the inter-layer connections. Note that in Equation (5.4), both W_l^s and $W_{l_1\ l}^s$ are sparse tensors. The inter-layer connectivity in SWANN enables enhanced data flow, both during inference and training stages, while the sparse connections mitigate unnecessary parameter utilization. Unlike the previously proposed feature concatenation methodology [77], we perform summation over the feature-maps. This, in turn, mitigates the appearance of extremely high dimensional weight kernels that result from channel-wise feature-map concatenation. Furthermore, the summation of feature-maps enables SWANN to be applicable to all network architectures with various layer configurations. We gradually increase the stride in the long-range sparse connections as a function of the distance between the inter-linked layers. This allows us to reduce the dimensionality of the produced feature-maps as well as tune the impact of added long-range connections. In addition to adjusting the convolution strides, we use zero-padding where necessary to match the dimensionality of inter-layer connected feature-maps.

► *Composite Non-linear Operation.* Unlike DenseNets [77] and ResNets [66] where several linear layers are concatenated before pooling is performed, SWANNs support pooling immediately after each $CONV$ layer as seen in conventional DNN architectures. We experiment with various configurations of the widely-used non-linear operations, i.e., BN, $ReLU$, and $Maxpool$ to investigate the effect of ordering on network convergence. Our experiments demonstrate that SWANN convergence is enhanced when the composite non-linear function, C_l is implemented

as a $ReLU$, followed by $Maxpool$ing and BN as shown in Figure 5.3.

5.4 Experiments

We conduct proof-of-concept experiments on different network architectures and image classification benchmarks to empirically demonstrate the enhanced training convergence of SWANNs compared to the baseline (conventional) DNNs. We leverage popular DL libraries Keras and PyTorch for our implementations. All experiments are performed on a machine with Nvidia Titan XP GPU and Intel Xeon CPU.

Our evaluations target both centralized and decentralized on-device learning scenarios. Sec. 5.4.3, 5.4.4, 5.4.5 enclose our evaluations in the centralized training setup. This setup directly simulates on-device learning applications where users train/finetune a model locally on their personal data samples, e.g., personalization. Sec. 5.4.6 encloses the evaluation in the decentralized (federated) learning setup where several users collaboratively train a global model by performing multiple local updates and a global aggregation.

5.4.1 Datasets

► *MNIST.* This dataset consists of 10 classes of 28×28 gray-scale images from handwritten digits with 60,000 train and 10,000 test images. We normalize the data using the per-channel mean and variance prior to training and testing.

► *CIFAR.* We carry out our experiments on the two available CIFAR datasets. CIFAR10 (C10) and CIFAR100 (C100) benchmarks consist of colored images with dimensionality 32×32 that are categorized in 10 and 100 classes, respectively. Each dataset contains 50,000 samples for training and 10,000 samples for testing. We use standard data augmentation routines popular in prior work [66, 79]: samples are normalized using per-channel mean and standard deviation. At training time, random horizontal mirroring, shifting, and slight rotation are also applied.

► *ImageNet.* The ISLVRC-2012 dataset, widely known as the ImageNet, consists of 1000

Table 5.1. Benchmarked DNNs for evaluating SWANN effectiveness. $CONV$ layers are represented as $\langle kernel\ size \rangle$Conv and FC layers are denoted by $\langle output\ elements \rangle$FC. BN and $ReLU$ are not shown for brevity.

	Convolution	**MaxPool**	**Convolution**	**MaxPool**	**Convolution**	**MaxPool**	**Convolution**	**MaxPool**	**Convolution**	**MaxPool**	**Classifier**
MNIST	5×5 Conv	2×2 stride 2	5×5 Conv	2×2 stride 2	5×5 Conv	2×2 stride 2	-	-	-	-	84FC 10FC, softmax
ConvNet-C [176]* **(C10, C100)**	[3×3 Conv] $\times2$	2×2 stride 2	[3×3 Conv] $\times2$	2×2 stride 2	[3×3 Conv] $\times3$	2×2 stride 2	[3×3 Conv] $\times3$	2×2 stride 2	[3×3 Conv] $\times3$	2×2 stride 2	512FC 10FC, softmax
AlexNet [102] **(ImageNet)**	11×11 Conv (stride 4)	2×2 stride 2	5×5 Conv	2×2 stride 2	3×3 Conv	-	3×3 Conv	-	3×3 Conv	2×2 stride 2	4096FC 4096FC 1000FC, softmax
ResNet-18 [66] **(ImageNet)**	7×7 Conv (stride 2)	3×3 stride 2	[3×3 Conv] $\times4$	-	[3×3 Conv] $\times4$	-	[3×3 Conv] $\times4$	-	[3×3 Conv] $\times4$	7×7 average pool	1000FC, softmax

* We modify the ConvNet-C fully-connected layers form [176] to comply with the CIFAR datasets.

	Convolution	**Dense Block (1)**	**Transition Block**	**Dense Block (2)**	**Transition Block**	**Dense Block (3)**	**Classifier**
DenseNet-40 [77] **(C10)**	3×3 Conv	[3×3 Conv]$\times12$	1×1 Conv 2×2 average pool	[3×3 Conv]$\times12$	1×1 Conv 2×2 average pool	[3×3 Conv]$\times12$	8×8 average pool 10FC, softmax

* Conv denotes a BN, followed by a $ReLU$ and a convolution layer.

different classes of colored images with 1.2 million samples for training and 50,000 samples for validation. We use the augmentation scheme proposed in [67, 176] to preprocess input samples: during training, we resize the images by randomly sampling the shorter edge from $[256, 480]$. A 224×224 crop is then randomly sampled from the image. We also perform per-channel normalization as well as horizontal mirroring [102].

5.4.2 Benchmarked Architectures

Table 5.1 encloses our baseline DNN architectures. SWANNs maintain the same feed-forward architecture as the baseline networks and are constructed by 1) replacing $CONV$ layers with sparse convolutions and 2) adding sparse convolutions between non-consecutive layers. Table 5.2 encloses the relative clustering coefficient γ_G, relative average path length λ_G, small-world property S_G, and rewiring probability p to achieve the corresponding SWANN for each baseline DNN. Here, γ_G, λ_G, and S_G follow the definitions in Equation (5.2). As seen, all SWANN models satisfy the small-world characteristic, i.e., $S_G > 1$.

5.4.3 Results on MNSIT

We train the 5-layer architecture shown in Table 5.1 as our baseline. The small-world equivalent of the baseline model is generated using a rewiring probability of 0.5. To prevent

Table 5.2. Graph characteristics of SWANN models.

model	γ_G	λ_G	S_G	p
MNIST	1.09	0.99	1.09	0.45
ConvNet-C	1.20	1.01	1.19	0.85
AlexNet	1.48	1.00	1.48	0.80
ResNet-18	1.06	1.00	1.06	0.83
DenseNet-40	1.20	1.01	1.19	0.92

overfitting, a dropout layer with the rate of 0.5 is added between the two FC layers in both the baseline DNN and SWANN. We train the models using Adadelta optimizer [230] with an initial learning rate of 1 and a decay of 0.95. Batch size is set to 256 for both models.

► *Convergence.* Figure 5.9 compares the test error and training loss of the baseline DNN and its small-world counterpart throughout training. Both models achieve a final test accuracy of 99.1% on the MNIST dataset. The plain baseline DNN converges to the aforesaid accuracy in 19 epochs (= 4864 iterations) while SWANN reaches the convergence accuracy in 9 epochs (= 2304 iterations) which shows $2.1\times$ improvement over the baseline.

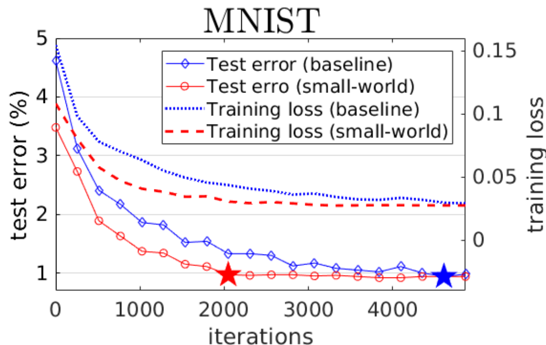

Figure 5.9. Comparison of a plain DNN's training convergence with its small-world equivalent. Here, the red and blue colors show SWANN and baseline, respectively. The $\star$ markers denote the point of convergence to final test accuracy for the models with the corresponding colors.

5.4.4 Results on CIFAR

ConvNet-C

We train the ConvNet-C [176] model on C10 and C100 benchmarks with a batch size of 128. To prevent overfitting, a dropout layer with a rate of 0.5 is added before the first FC

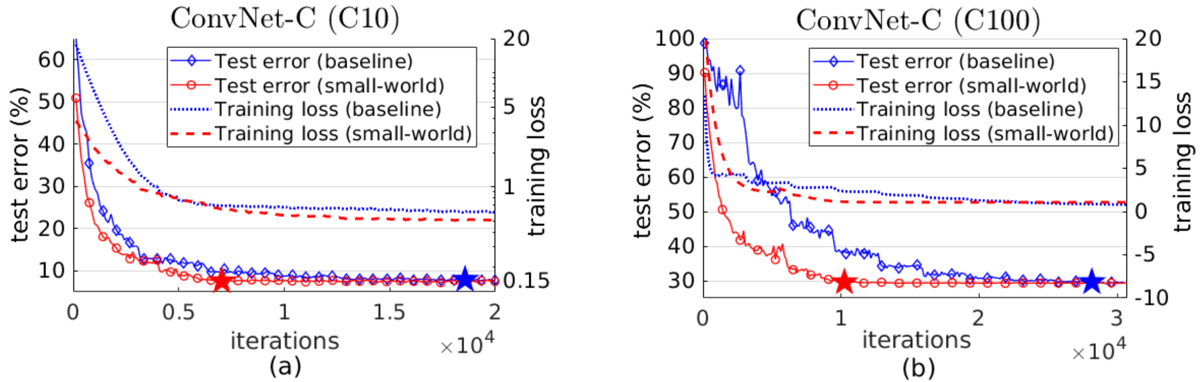

Figure 5.10. Test error and training loss versus iterations for a ConvNet-C model and the rewired SWANN trained on (a) CIFAR10 and (b) CIFAR100 datasets. Here, the red and blue colors show SWANN and baseline, respectively. The $\star$ markers denote the point of convergence to final test accuracy for the models with the corresponding colors.

layer. The small-world model is constructed using the same configuration of layers as the baseline, including the dropout layers. We use Stochastic-Gradient-Descent (SGD) optimizer with Nesterov, 0.9 momentum, and a $5e-4$ weight decay. Models are trained for $2e4$ and $3e4$ iterations on C10 and C100, respectively. The initial learning rate is set to 0.01 for both datasets and learning rate is decayed by 0.5 upon optimization plateau.

► *Convergence.* Figure 5.10-(a) illustrates the test error and training loss of the baseline and SWANNs as two representatives of the convergence speed. Similarly, for C100 benchmark, the corresponding convergence curve is presented in Figure 5.10-(b). While these figures qualitatively demonstrate the effectiveness of our methodology, we provide a quantitative measure for a solid comparison between SWANN and the baseline. We investigate several points corresponding to various test accuracies and compare the two models' convergence to these points.

Table 5.3 summarizes the per-accuracy speed-up of SWANN over the baseline model. As seen, the speed-up varies for different accuracies, however, for all test accuracies, SWANN requires a substantially fewer number of iterations for convergence. At the final saturation point (marked by $\star$ on Figure 5.10), both models achieve comparable accuracies while SWANN enjoys a $2.6\times$ and $2.8\times$ reduction in convergence time for C10 and C100 datasets, respectively.

Table 5.3. Point-wise comparison of convergence speed-up for a SWANN and its equivalent baseline network (ConvNet-C) on CIFAR benchmarks.

Dataset	Model	Metric					
CIFAR10	**Baseline**	Test Error (%)	24.21	17.80	9.22	8.51	**7.56**
		Iterations	1408	2560	8704	11008	18560
	SWANN	Test Error (%)	23.73	17.57	8.64	8.25	**7.44**
		Iterations	896	1536	4992	5888	7040
	Speed-up		1.57×	1.67×	1.74×	1.87×	**2.64**×
CIFAR100	**Baseline**	Test Error (%)	77.08	52.3	41.54	31.14	**29.52**
		Iterations	2944	6144	9472	16128	28928
	SWANN	Test Error (%)	76.67	50.57	40.18	31.15	**29.26**
		Iterations	384	1408	3072	7808	10240
	Speed-up		7.67×	4.36×	3.08×	2.1×	**2.82**×

DenseNet

DenseNets [77] achieve state-of-the-art accuracy by connecting all neurons from different layers of a dense block with trainable (dense) parameters. Such dense connectivity pattern results in high redundancy in the parameter space and extra training overhead. We show that a SWANN with only sparse connections and much fewer parameters achieves similar results as DenseNet. We train a DenseNet model with 40 layers and $k = 12$ (Table 5.1) on C10 dataset. The equivalent SWANN is constructed by removing all long-range dense connection from the architecture and rewiring the remaining (short) edges such that each dense block becomes small-world. The SWANN maintains the same number of layers while the inter-layer connections are implemented using sparse convolution kernels, thus incurring substantially fewer number of trainable parameters.

We use the publicly available PyTorch implementation for DenseNets[1] and replace the model with our small-world network. Same training scheme as explained in the original DenseNet paper [77] is used: models are trained for 19200 iterations with a batch size of 64. Initial learning rate is 0.1 and decays by 10 at $\frac{1}{2}$ and $\frac{3}{4}$ of the total training iterations.

► *Convergence.* Figure 5.11 demonstrates the test accuracy of the models versus the number of epochs. As can be seen, although SWANN has much fewer parameters, both models achieve

[1] https://github.com/andreasveit/densenet-pytorch

comparable validation accuracy while showing identical convergence speed. We report the computational complexity (FLOPs) of the models as the total number of multiplications performed during a forward propagation through the network. Table 5.4 compares the benchmarked DenseNet and SWANN in terms of FLOPs and number of trainable weights in $CONV$ and FC layers. We highlight that SWANN achieves comparable test accuracy while having $10\times$ reduction in parameter space size.

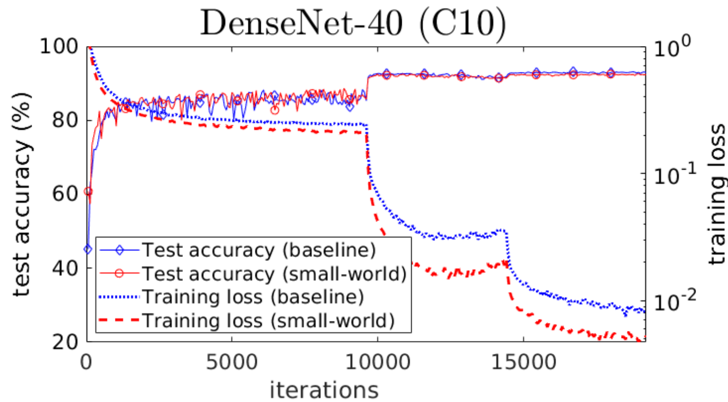

Figure 5.11. Training loss and testing accuracy of the 40-layer (k=12) DenseNet [77] with 1M parameters and our corresponding SWANN with less than 100K parameters.

Table 5.4. Comparison of the computational complexity and model parameter space between a 40-layer DenseNet with k=12 and the corresponding SWANN.

Model	Depth	Params	FLOPs	Test Error
DenseNet ($k = 12$)	40	910K	285.3M	0.071
SWANN	40	98K	85.5M	0.074

5.4.5 Results on ImageNet

AlexNet

We train the AlexNet [102] model on ImageNet dataset and follow the architecture provided in the Caffe model zoo [1] (See Table 5.1). In order to mitigate overfitting, we add dropout layers with probability 0.5 after the FC layers. Loss minimization is performed by means of SGD with Nesterov [143] and a 0.9 momentum. We set the batch size to 64 for both models and incorporate an exponential decay for the learning rate: initial learning rate is set to

$2.5e{-3}$ and the decay factor is 0.99999875 [177].

► *Convergence.* To fully examine the performance of our model, we report the speed-up of SWANN over the baseline for several values of test error. Table 5.5 encloses the point-wise comparison between the benchmarked models. As can be seen, for all values of test error, the convergence of our small-world architecture is faster. SWANN converges to the final test accuracy after 3776 iterations while the baseline model needs 5120 iterations, resulting in a $1.4\times$ overall speed-up.

Table 5.5. Performance of baseline AlexNet and its SWANN on ImageNet dataset.

AlexNet	**Baseline**	Test Error (%)	51.72	46.29	44.21	42.31	**42.01**
		Iterations	1088	2304	3264	4416	5120
	SWANN	Test Error (%)	51.97	46.49	44.25	42.31	**41.55**
		Iterations	768	1664	2368	3520	3776
	Speed-up		1.42×	1.38×	1.38×	1.25×	**1.36×**

ResNet

We adopt the training scheme in the original ResNet paper [66]. To build the SWANN, we first remove all shortcut and bottleneck connections from the model. We then rewire the connections in the acquired plain network such that it becomes small-world. No dropout is used for the baseline and SWANN. Batch size is set to 128 and we use SGD with 0.9 momentum and $1e-4$ weight decay. Initial learning rate is 0.1 and decays by 0.1 when the accuracy plateaus. We train the models for $9e5$ iterations and report single-crop accuracies.

► *Convergence.* We enclose point-wise comparisons between the baseline ResNet and SWANN for various iterations and test errors in Table 5.6. As seen, SWANN achieves faster convergence throughout training and reaches the final test accuracy with $1.3\times$ less iterations. This shows that the systematic restructuring of long edges in SWANN allows for a better convergence behavior compared to the replicated blocks in ResNet.

Table 5.6. Point-wise convergence comparison of ResNet-18 and its SWANN equivalent on ImageNet dataset.

ResNet-18	**BaseLine**	Test Error (%)	60.37	56.94	37.91	**31.72**
		Iterations	1792	3456	7424	9344
	SWANN	Test Error (%)	59.63	56.76	37.86	**31.68**
		Iterations	512	768	3584	7168
	Speed-up		3.50×	4.50×	2.07×	**1.31**×

5.4.6 Federated Learning

In this section, we corroborate SWANN enhanced convergence in the federated learning setting as a candidate application for on-device learning. In this setup, a server holds a global model and users each have a unique (local) dataset. The users compute the weight updates for the global model on their local datasets and communicate the updated weights to the server. The server then aggregates the weights from all users and updates the global model. We implement the popular `FedAvg` algorithm [132] for federated learning where each user performs several local updates on the global model before sending the updated weights to the global server.

Following the original paper [132] we consider a pool of 100 users and randomly select 10 users at each iteration to send their updates to the server. We perform 5 local updates on each user with a batch size of 10. This setting provides a good balance between the total number of communication rounds for convergence and the required amount of computation per iteration for each user [132]. Optimization is performed using SGD with learning rate of 0.01 for both the baseline DNN and the SWANN. Our evaluations are performed on the MNIST dataset with the baseline DNN architecture shown in Table 5.1.

We evaluate SWANN under two distributed data settings: 1) IID where the data is distributed uniformly across all users, i.e., each user has instances of all classes. 2) Non-IID where the entire data is sorted by their label, divided into 200 shards of size 300, and each user is assigned 2 shards [132]; users on average only have instances from two classes. Figure 5.12-a,b demonstrate the test accuracy and training loss versus number of local updates for the baseline

DNN and its corresponding SWANN in the IID and non-IID settings, respectively.

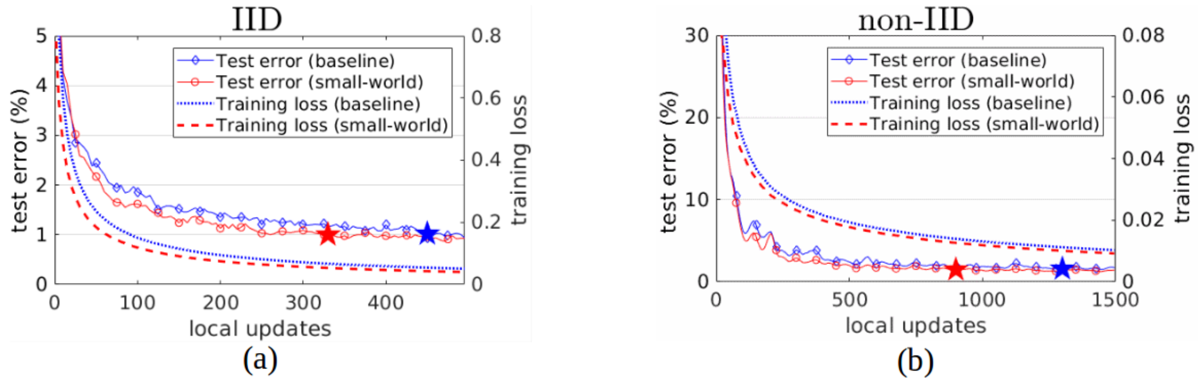

Figure 5.12. Training loss and testing accuracy of the baseline DNN and SWANN in the federated learning scenario with a) IID and b) non-IID data distributions. Here, the red and blue colors show SWANN and baseline, respectively. The $\star$ markers denote the point of convergence to final test accuracy for the models with the corresponding colors.

► *IID data.* To reach the final test accuracy of 99.1%, the baseline DNN requires 450 local updates while SWANN only requires 330, thereby showing a $1.4\times$ reduction in the computation performed on the user devices. In addition to the savings in computation, SWANN also reduces the total number of required global aggregations by $1.4\times$, which directly translates to a reduced communication cost between users and the server.

► *non-IID data.* In the non-IID setting, SWANN outperforms the baseline DNN both in terms of the convergence rate and final test accuracy. The baseline DNN requires 1300 local updates to converge to the final test accuracy of 98.7% while SWANN converges to 98.9% test accuracy with 1020 local updates. As a result, SWANN achieves $1.3\times$ reduction in the users' computation and communication during training. The higher accuracy of SWANN in the challenging non-IID data setting further demonstrates the better stability of SWANN during training compared to non small-world architectures.

For a more detailed comparison, we report convergence steps needed for various test accuracies for the baseline DNN and SWANN in Table 5.7. SWANN reaches all target accuracies considerably faster than the baseline. On average, SWANN requires $1.5\times$ and $1.6\times$ less

computation and communication for user devices in the IID and non-IID settings, respectively.

Table 5.7. Point-wise convergence comparison of the 5-layer baseline DNN and its corresponding SWANN in the federated learning scenario.

IID	**BaseLine**	Test Error (%)	98.16	98.65	98.90	**99.07**
		Local Updates	85	200	355	450
	SWANN	Test Error (%)	98.14	98.65	98.89	**99.07**
		Local Updates	60	130	220	330
	Speed-up		1.42×	1.54×	1.61×	**1.36**×
non-IID	**BaseLine**	Test Error (%)	96.12	96.67	98.10	**98.74**
		Local Updates	170	405	555	1300
	SWANN	Test Error (%)	96.14	97.63	98.16	**98.72***
		Local Updates	100	230	325	900
	Speed-up		1.70×	1.76×	1.71×	**1.44**×

*SWANN reaches a final test accuracy of 98.90% after 1020 local updates.

5.5 Discussion on Long-range Connections

The selected small-world structure for a given DNN has two main characteristics, namely high clustering of nodes and small average path length between neurons across layers. We postulate that such qualities render the SWN desirable during training due to the enhanced information flow paths existent in these efficiently-connected networks. To examine our hypothesis, we visualize the weights connecting different layers of the trained SWANN for C10, C100 (ConvNet-C), and ImageNet (AlexNet) benchmarks. Figure 5.13 presents a heat map of the average absolute values of weights connecting each pair of $CONV$ layers.

Each square at position (l_1, l_2) of the heatmap represents the strength of the connections between layers l_1 and l_2 where l_0 denotes network input. Color shades of orange, red and maroon indicate strong inter-layer dependency while the white color indicates that no connections are present between the corresponding layers in SWANN. We summarize our observations based on the heat map as the following:

1. Each layer has strong connections to its non-subsequent layers indicating that long-range edges established in SWANN are crucial to performance.

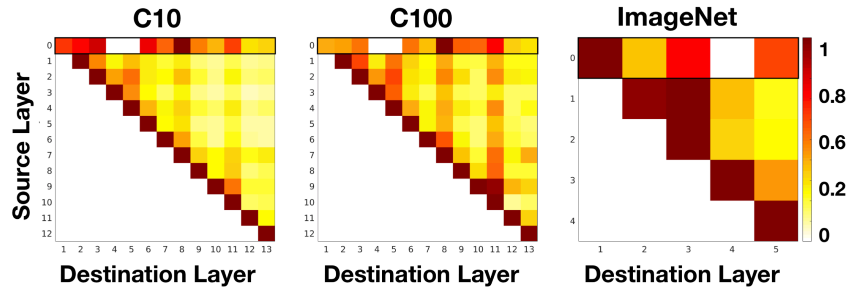

Figure 5.13. Visualization of average absolute value of trained weights within $CONV$ layers of SWANNs. Colors encode the connectivity strength between layers with red being the strongest and white denoting no connection. The marked rows with black box borders correspond to the input layer of the networks.

2. The input layer has spread weights across all layers of the network which demonstrates the importance of connections between earlier and deeper layers.

3. SWANN preserves the strong connections between one layer and the immediately proceeding layer, thus, maintaining the conventional DNN data flow.

5.6 Conclusion

We propose a novel methodology that adaptively modifies conventional feed-forward DL models to new architectures, called SWANN, that fall into the category of small-world networks—a class of complex graphs used to study real-world models such as human brain and the neural networks of animals. By leveraging the intriguing features of small-world networks, e.g., enhanced signal propagation speed and synchronizability, SWANNs enjoy improved data flow within the network, resulting in substantially faster convergence speed during training. Our small-world models are implemented via sparse connections from each DNN layer to all succeeding layers. Such sparse convolutions enable SWANNs to benefit from long-range connections while mitigating the redundancy in the parameter space existent in prior art.

As our experiments demonstrate, SWANNs are able to achieve state-of-the-art accuracy in $\approx 2.1\times$ lower number of training iterations, on average. Furthermore, compared to a densely-connected architecture, SWANNs achieve comparable accuracy while having $10\times$ reduction in

the number of parameters. In summary, due to their optimal graph connectivity and fast response to training, SWANNs can be advantageous for on-device learning in embedded applications.

5.7 Acknowledgements

Chapter 5 is a reprint of the material as it appears in: M. Javaheripi, B. Rouhani, and F. Koushanfar, “SWANN: Small-World Architecture for Fast Convergence of Neural Networks”, in *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2021. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Improving Inference Performance via Neural Architecture Search

The Transformer architecture [197] has been used as the de-facto building block of most pre-trained language models like GPT [163]. A common problem arises when one tries to create smaller versions of Transformer models for edge or real-time applications (e.g. text prediction) with strict memory and latency constraints: it is not clear what the architectural hyperparameters should be, e.g., number of attention heads, number of layers, embedding dimension, and the inner dimension of the feed forward network, etc. This problem is exacerbated if each Transformer layer is allowed the freedom to have different values for these settings. This results in a combinatorial explosion of architectural hyperparameter choices and a large heterogeneous search space. For instance, the search space considered in this work consists of over 10^{54} possible architectures.

NAS is an organic solution due to its ability to automatically search through candidate models with multiple conflicting objectives like latency vs. task performance. The central challenge in NAS is the prohibitively expensive function evaluation, i.e., evaluating each architecture requires training it on the dataset at hand. Thus it is often infeasible to evaluate more than a handful of architectures during the search phase. Supernets [152] have emerged as a dominant paradigm in NAS which combine all possible architectures into a single graph and jointly train them using weight-sharing. Nevertheless, supernet training imposes constraints on

the expressiveness of the search space [144] and is often memory-hungry [18, 215, 216] as it creates large networks during search. Additionally, training supernets is non-trivial as children architectures may interfere with each other and the ranking between sub-architectures based on task performance is not preserved [144][1].

We consider a different approach by proposing a training-free proxy that provides a highly accurate ranking of candidate architectures during NAS without need for costly function evaluation or supernets. Our scope is NAS for efficient autoregressive Transformers used in language modeling. We design a lightweight search method that is target hardware-aware and outputs a gallery of models on the Pareto-frontier of perplexity versus hardware metrics. We term this method Lightweight Transformer Search (LTS). LTS relies on our somewhat surprising observation: *the decoder parameter count has a high rank correlation with the perplexity of fully trained autoregressive Transformers.*

Given a set of autoregressive Transformers, one can accurately rank them using decoder parameter count as the proxy for perplexity. Our observations are also well-aligned with the power laws in [96], shown for homogeneous autoregressive Transformers, i.e., when all decoder layers have the same configuration. We provide extensive experiments that establish a high rank correlation between perplexity and decoder parameter count for *both* homogeneous and heterogeneous search spaces.

The above phenomenon coupled with the fact that a candidate architecture's hardware performance can be measured on the target device leads to a training-free search procedure: *pick one's favorite discrete search algorithm (e.g. evolutionary search), sample candidate architectures from the search space; count their decoder parameters as a proxy for task performance (i.e., perplexity); measure their hardware performance (e.g., latency and memory) directly on the target device; and progressively create a Pareto-frontier estimate.* While we have chosen a reasonable search algorithm in this work, one can plug and play any Pareto-frontier search method such as those in [59].

[1] See [144] for a comprehensive treatment of the difficulties of training supernets.

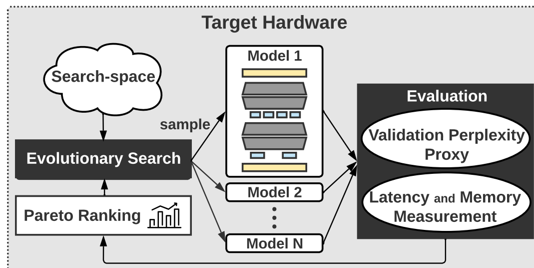

Figure 6.1. High-level overview of LTS. We propose a training-free zero-cost proxy for evaluating the validation perplexity of candidate architectures. Pareto-frontier search is powered by evolutionary algorithms which use the proposed proxy along with real latency and memory measurements on the target hardware to evaluate sampled architectures.

Building upon these insights, Figure 6.1 shows a high-level overview of LTS. We design the first training-free Transformer search that is performed entirely on the target (constrained) platform. As such, LTS easily performs a multi-objective NAS where several underlying hardware performance metrics, e.g., latency and peak memory utilization, are simultaneously optimized. Using our training-free proxy, we extract the 3-dimensional Pareto-frontier of perplexity versus latency and memory in a record-breaking time of < 3 hours on a commodity Intel Core i7 CPU. Notably, LTS eliminates the carbon footprint from hundreds of GPU hours of training associated with legacy NAS methods.

To corroborate the effectiveness of our proxy, we train over 2900 Transformers on three large language modeling benchmark datasets, namely, WikiText-103 [136], One Billion Word [24], and Pile [50]. We use LTS to search for Pareto-optimal architectural hyperparameters in two popularly used autoregressive Transformer backbones, namely, Transformer-XL [32] and GPT-2 [153]. We believe decoder parameter count should be regarded as a competitive baseline for evaluating Transformer NAS, both in terms of ranking capabilities and easy computation. We open-source our code along with tabular information of our trained models to foster future NAS research on Transformers.

6.1 Related Work

Here, we discuss literature on automated search for Transformers in the language domain. Please refer to extensive surveys on NAS [41, 208] for a broader overview of the field.

Decoder-only Architectures. Authors of [179] search over TensorFlow programs that implement an autoregressive language model via evolutionary search. Since most random sequences of programs either have errors or underperform, the search has to be seeded with the regular Transformer architecture, termed "Primer". As opposed to "Primer" which uses large computation to search a general space, we aim to efficiently search the "backbone" of traditional decoder-only Transformers. Additionally, the objective in "Primer" is to find models that train faster. Our objective for NAS, however, is to deliver Pareto-frontiers for inference, with respect to perplexity and hardware constraints.

Encoder-only Architectures. Relative to decoder-only autoregressive language models, encoder-only architectures like BERT [35] have received much more recent attention from the NAS community. NAS-BERT [214] trains a supernet to efficiently search for masked language models (MLMs) which are compressed versions of the standard BERT, Such models can then be used in downstream tasks as is standard practice. Similar to NAS-BERT, authors of [215] train a supernet to conduct architecture search with the aim of finding more efficient BERT variants. They find interesting empirical insights into supernet training issues like differing gradients at the same node from different child architectures and different tensors as input and output at every node in the supernet. The authors propose fixes that significantly improve supernet training. Several other works [49, 194, 220] also conduct variants of supernet training with the aim of finding more efficient BERT models.

Encoder-Decoder Related: Applying the well-known DARTS [117] approach to Transformer search spaces leads to memory-hungry supernets. To mitigate this issue, [234] proposes a multi-split reversible network and a memory-efficient backpropagation algorithm. One of the earliest papers that applied discrete NAS to Transformer search spaces was [178], which uses

a modified form of evolutionary search. Due to the expense of directly performing discrete search on the search space, this work incurs extremely large computation overhead. Follow-up work by [203] uses the Once-For-All [18] approach to train a supernet for encoder-decoder architectures used in machine translation. Search is performed on subsamples of the supernet that inherit weights to estimate task accuracy. For each target device, the authors train a small neural network regressor on thousands of architectures to estimate latency. As opposed to using a latency estimator, LTS evaluates the latency of each candidate architecture on the target hardware. Notably, by performing the search directly on the target platform, LTS can easily incorporate various hardware performance metrics, e.g., peak memory utilization, for which accurate estimators may not exist. To the best of our knowledge, such holistic integration of multiple hardware metrics in Transformer NAS has not been explored previously.

6.2 Lightweight Transformer Search

We perform an evolutionary search over candidate architectures to extract models that lie on the Pareto-frontier. In contrast to the vast majority of prior methods that deliver a single architecture from the search space, our search is performed over the entire Pareto, generating architectures with a wide range of latency, peak memory utilization, and perplexity with one round of search. This alleviates the need to repeat the NAS algorithm for each hardware performance constraint.

To evaluate candidate models during the search, LTS uses a training-free proxy for the validation perplexity. By incorporating training-free evaluation metrics, LTS, for the first time, performs the entire search directly on the target (constrained) hardware. Therefore, we can use real measurements of hardware performance during the search. Algorithm 4 outlines the iterative process performed in LTS [2] for finding candidate architectures in the search space ($\mathcal{D}$), that lie on the 3-dimensional Pareto-frontier ($\mathbb{F}$) of perplexity versus latency and memory. At each

[2]The Pareto-frontier search method in Algorithm 4 is inspired by [40] and [74]. Other possibilities include variations proposed in [59], evaluation of which is orthogonal to our contributions in this work.

iteration, a set of points ($\mathbb{F}'$) are subsampled from the current Pareto-frontier. A new batch of architectures ($\mathbb{S}_N$) are then sampled from $\mathbb{F}'$ using evolutionary algorithms ($EA(.)$). The new samples are evaluated in terms of latency ($\mathcal{L}$), peak memory utilization ($\mathcal{M}$), and validation perplexity ($\mathcal{P}$). Latency and memory are measured directly on the target hardware while the perplexity is indirectly estimated using our accurate and training-free proxy methods. Finally, the Pareto-frontier is recalibrated using the lower convex hull of all sampled architectures. In the context of multi-objective NAS, Pareto-frontier points are those where no single metric (e.g., perplexity, latency, and memory) can be improved without degrading at least one other metric [59]. To satisfy application-specific needs, optional upper bounds can be placed on the latency and/or memory of sampled architectures during search.

Algorithm 4. LTS's training-free NAS

Inputs: Search space $\mathcal{D}$, n_{iter}.
Output: Perplexity-latency-memory Pareto-frontier $\mathbb{F}$.

1: $\mathcal{L}, \mathcal{M}, \mathcal{P}, \mathbb{F} \leftarrow \emptyset, \emptyset, \emptyset, \emptyset$
2: **while** $N \leq n_{iter}$ **do**
3: $\quad \mathbb{F}' \leftarrow$ Subsample($\mathbb{F}$)
4: $\quad \mathbb{S}_N \leftarrow EA(\mathbb{F}', \mathcal{D})$
5: $\quad \mathcal{L} \leftarrow \mathcal{L} \bigcup$ Latency($\mathbb{S}_N$) ▷ hardware profiling
6: $\quad \mathcal{M} \leftarrow \mathcal{M} \bigcup$ Memory($\mathbb{S}_N$) ▷ hardware profiling
7: $\quad \mathcal{P} \leftarrow \mathcal{P} \bigcup$ Proxy($\mathbb{S}_N$) ▷ estimate perplexity
8: $\quad \mathbb{F} \leftarrow$ LowerConvexHull($\mathcal{P}, \mathcal{L}, \mathcal{M}$) ▷ update the Pareto-frontier

Search Space. Figure 6.2 shows all elastic parameters in LTS search space, namely, number of layers (n_{layer}), number of attention heads (n_{head}), decoder output dimension (d_{model}), inner dimension of the feed forward network (d_{inner}), embedding dimension (d_{embed}), and the division factor (k) of adaptive embedding [9]. These architectural parameters are compatible with popularly used autoregressive Transformer backbones, e.g., GPT. We adopt a heterogeneous search space where the backbone parameters are decided on a per-layer basis. This is in contrast to the homogeneous structure commonly used in Transformers [32, 163], which reuses the same configuration for all layers. Compared to homogeneous models, the flexibility of heterogeneous

architectures enables them to obtain much better hardware performance under the same perplexity budget (see Section 6.3.5).

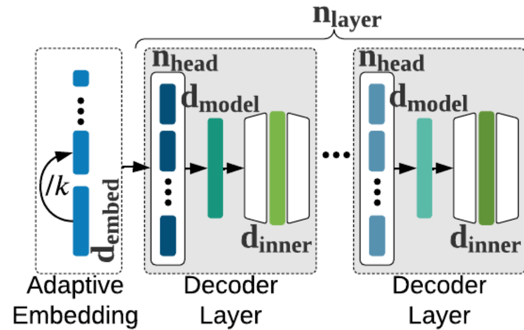

Figure 6.2. Elastic parameters in LTS search space.

Heterogeneous search space was previously explored in [203]. However, due to the underlying supernet structure, not all design parameters can change freely. As an example, the dimensionality of the *Q, K, V* vectors inside the encoder and decoder layers is fixed to a large value of 512 to accommodate inheritance from the supernet. Our search space, however, allows exploration of all internal dimensions without constraints. By not relying on the supernet structure, our search space easily encapsulates various Transformer backbones with different configurations of the input/output embedding layers and elastic internal dimensions.

We use the notation $\{v_{min}, \ldots, v_{max}|\text{step size}\}$ to show the valid range of values. LTS searches over the following values for the architectural parameters in our backbones: $n_{layer} \in \{2, \ldots, 16|1\}$, $d_{model} \in \{128, \ldots, 1024|64\}$, $d_{inner} \in \{256, \ldots, 4096|64\}$, and $n_{head} \in \{2, 4, 8\}$. Additionally we explore adaptive input embedding [9] with $d_{embed} \in \{128, 256, 512\}$ and factor $k \in \{1, 2, 4\}$. Once a d_{model} is sampled, we adjust the lower bound of the above range for d_{inner} to $2 \times d_{model}$. Encoding this heuristic inside the search ensures that the acquired models will not suffer from training collapse. Our heterogeneous search space encapsulates more than 10^{54} different architectures. Such high dimensionality further validates the critical need for training-free NAS.

6.2.1 Training-free Architecture Ranking

► **Low-cost Ranking Proxies.** Recently, authors of [2] utilize the summation of pruning scores over all model weights as the ranking proxy for Convolutional Neural Networks (CNNs), where a higher score corresponds to higher architecture rank in the search space. White et al. [207] analyze these and more recent proxies and find that no particular proxy performs consistently well over various tasks and baselines, while parameter and floating point operations (FLOPS) count proxies are quite competitive. However, they did not include Transformer-based search spaces in their analysis. To the best of our knowledge, low-cost (pruning-based) proxies have not been evaluated on Transformer search spaces in the language domain. Note that one cannot naively apply these proxies to language models. Specifically, since the embedding layer in Transformers is equivalent to a lookup operation, special care must be taken to omit this layer from the proxy computation. Using this insight, we perform the first systematic study of low-cost proxies for NAS on autoregressive Transformers for text prediction.

We leverage various pruning metrics, namely, `grad_norm`, `snip` [109], `grasp` [201], `fisher` [189], and `synflow` [187]. We also study `jacob_cov` [133] and `relu_log_det` [134] which are low-cost scoring mechanisms proposed for NAS on CNNs in vision tasks. While these low-cost techniques do not perform model training, they require forward and backward passes over the architecture to compute the proxy, which can be time-consuming on low-end hardware. Additionally, the aforesaid pruning techniques, by definition, incorporate the final softmax projection layer in their score assessment. Such an approach seems reasonable for CNNs dealing with a few classification labels, however, it can skew the evaluation for autoregressive Transformers dealing with a large output vocabulary space. To overcome these shortcomings, we introduce a zero-cost architecture ranking strategy in the next section that outperforms the proposed low-cost proxies in terms of ranking precision, is data free, and does not perform any forward/backward propagation.

► **Decoder Parameter Count as a Proxy.** We empirically establish a strong correlation between

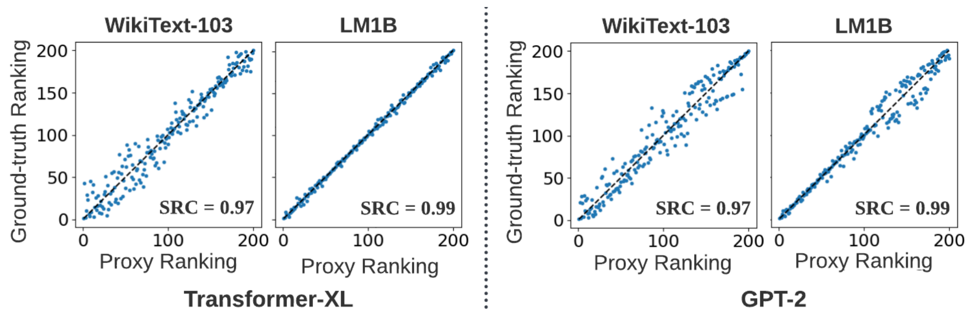

Figure 6.3. Our training-free zero-cost proxy based on decoder parameter count is highly correlated with the (ground truth) validation perplexity after full training. Each plot contains 200 architectures sampled randomly from the search space of Transformer-XL or GPT-2 backbone.

the parameter count of decoder layers and final model performance in terms of validation perplexity. We evaluate 200 architectures sampled uniformly at random from the search space of two autoregressive Transformer backbones, namely, Transformer-XL and GPT-2. These architectures are trained fully on WikiText-103 and One Billion Word (LM1B) datasets, which consumes over **25000** GPU-hours on NVIDIA A100 and V100 nodes. We compare the ranking obtained using decoder parameter count proxy and the ground truth ranking after full training in Figure 6.3. On WikiText-103, zero-cost ranking using the decoder parameter count obtains a Spearman's Rank Correlation (SRC) of 0.97 with full training. SRC further increases to 0.99 for the more complex LM1B benchmark on both backbones. This validates that the decoder parameter count is strongly correlated with final model performance, thereby providing a reliable training-free proxy for NAS.

6.3 Experiments

We conduct experiments to seek answers to the following critical questions:

❶ How well can training-free proxies perform compared to training-based methods for estimating the performance of Transformer models?

❷ How does model topology affect the performance of the proposed decoder parameter proxy?

❸ Can our training-free decoder parameter count proxy be integrated inside a search algorithm

to estimate the Pareto-frontier? How accurate is such an estimation of the Pareto?

❹ Which models are on the Pareto-frontier of perplexity, latency, and memory for different hardware?

❺ How well do LTS models perform in zero and one-shot settings compared to hand-designed variants when evaluated on downstream tasks?

We empirically answer questions ❶, ❷, ❸, ❹, and ❺ in Sections 6.3.2, 6.3.3, 6.3.4, 6.3.5, and 6.3.6 respectively.

6.3.1 Experimental Setup

Datasets. We conduct experiments on three datasets, namely, WikiText-103, LM1B, and the Pile. The datasets are tokenized using word-level and byte-pair encoding for models with Transformer-XL and GPT-2 backbones, respectively.

Training and Evaluation. We adopt the open-source code by [45] and [145] to implement the GPT-2 and Transformer-XL backbones, respectively. We further use the source code provided in [6] to implement the baseline OPT-350M and LTS models used in zero and one-shot evaluations. Table 6.1 encloses the hyperparameters used for training. In this work, each model is trained separately from scratch. In many scenarios, the user only needs to train one model from the Pareto-frontier, which is selected based on their needs for perplexity, latency, and memory. However, if the users are interested in multiple models, they can either train all models separately or fuse them and train them simultaneously using weight sharing as in [203, 227].

Throughout the text, validation perplexity is measured over a sequence length of 192 and 32 tokens for WikiText-103 and LM1B datasets, respectively. For our zero and one-shot evaluations, we adopt the open-source code by [51]. Inference latency and peak memory utilization are measured on the target hardware for a sequence length of 192, averaged over 10 measurements. The sequence length is increased to 2048 for latency comparison with the OPT baseline. We utilize PyTorch's native benchmarking interface for measuring the latency and memory utilization of candidate architectures.

Table 6.1. LTS training hyperparameters for different backbones. Here, DO represents dropout.

Backbone	Dataset	Tokenizer	# Vocab	Optim.	# Steps	Batch size	LR	Scheduler	Warmup	DO	Attn DO
Transformer-XL	WT103	Word	267735	LAMB [223]	4e4	256	1e-2	Cosine	1e3	0.1	0.0
	LM1B	Word	267735	Adam	1e5	224	2.5e-4	Cosine	2e4	0.0	0.0
GPT-2	WT103	BPE	50264	LAMB [223]	4e4	256	1e-2	Cosine	1e3	0.1	0.1
	LM1B	BPE	50264	LAMB [223]	1e5	224	2.5e-4	Cosine	2e4	0.1	0.1
	Pile	BPE	50272	Adam	5.48e4	256	3e-5	Linear	715	0.1	0.0

Search Setup. Evolutionary search is performed for 30 iterations with a population size of 100; the parent population hash 20 samples out of the total 100; 40 mutated samples are generated per iteration from a mutation probability of 0.3, and 40 samples are created using crossover.

Backbones. We apply our search on two widely used autoregressive Transformer backbones, namely, Transformer-XL [32] and GPT-2 [153] that are trained from scratch with varying architectural hyperparameters. The internal structure of these backbones are quite similar, containing decoder blocks with attention and feed-forward layers. The difference between the backbones lies mainly in their dataflow structure; the Transformer-XL backbone adopts a recurrence methodology over past states coupled with relative positional encoding which enables modeling longer-term dependencies.

Performance Criteria. To evaluate the ranking performance of various proxies, we first establish a ground truth ranking of candidate architectures by training them until full convergence. This ground truth ranking is then utilized to compute two performance criteria as follows:

► Common Ratio (CR): We define CR as the percentage overlap between the ground truth ranking of architectures versus the ranking obtained from the proxy. CR quantifies the ability of the proxy to identify the top$k\%$ architectures with lowest validation perplexity after full training.

► Spearman's Rank Correlation (SRC): We use this metric to measure the correlation between the proxy ranking and the ground truth. Ideally, the proxy ranking should have high correlation with the ground truth over the entire search space as well as high-performing candidate models.

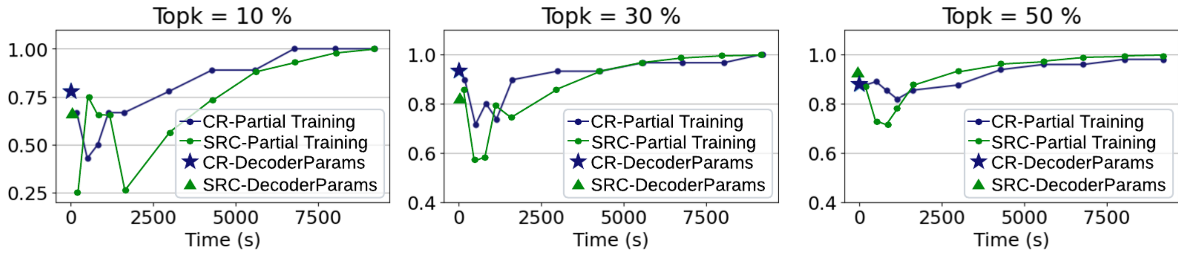

Figure 6.4. Comparison between partial training and our zero-cost proxy, i.e., decoder parameter count, in terms of ranking performance and timing overhead. Each subplot corresponds to a top$k\%$ of the randomly sampled models, based on their validation perplexity after full training.

6.3.2 How do training-free proxies perform compared to training-based methods?

In this section, we benchmark several proxy methods for estimating the rank of candidate architectures. Specifically, we investigate three different ranking techniques, namely, partial training, low-cost methods, and number of decoder parameters.

▶ **Partial Training.** We first analyze the relationship between validation perplexity after a shortened training period versus that of full training for ranking candidate models. We stop the training after $\tau \in [1.25\%, 87.5\%]$ of the total training iterations needed for model convergence. Figure 6.4 demonstrates the SRC and CR of partial training with various τs, evaluated on 100 randomly selected models from the Transformer-XL backbone, trained on WikiText-103. The horizontal axis denotes the average time required for τ iterations of training across all sampled models. Intuitively, a higher number of training iterations results in a more accurate estimate of final perplexity. Nevertheless, the increased wall-clock time prohibits training during search and also imposes the need for GPUs. Interestingly, very few training iterations, i.e., 1.25%, provide a good proxy for final performance with an SRC of > 0.9 on the entire population. Our training-free proxy, i.e., decoder parameter count, shows a similar SRC as partial training.

▶ **Low-cost Proxies.** We evaluate various low-cost methods introduced in Section 6.2.1 on 200 randomly sampled architectures from the Transformer-XL backbone, trained on WikiText-103. Figure 6.5a shows the SRC between low-cost proxies and ground truth ranking after

full training. We measure the cost of each proxy in terms of FLOPs. As seen, the evaluated low-cost proxies have a strong correlation with the ground truth ranking (even the lowest performing `relu_log_det` has > 0.8 SRC), validating the effectiveness of training-free NAS on autoregressive Transformers. The lower performance of `relu_log_det` can be attributed to the much higher frequency of ReLU activations in CNNs, for which the method was originally developed, compared to Transformer-based architectures. Our analysis on 100 randomly selected models with homogeneous structures also shows a strong correlation between the low-cost proxies and validation perplexity, with decoder parameter count outperforming other proxies as shown in Figure 6.5b. We omit the `relu_log_det` method from Figure 6.5b as it provides a low SRC of 0.42 due to heavy reliance on ReLU activations.

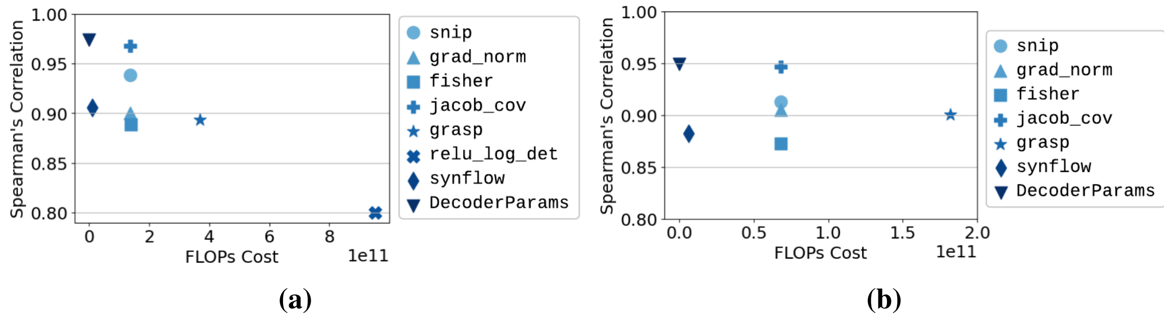

Figure 6.5. SRC between low-cost proxies and the ground truth ranking after full training of randomly sampled Transformers with (a) heterogeneous and (b) homogeneous decoder blocks. The decoder parameter count obtains the best SRC with zero cost.

► **Parameter Count.** Figure 6.6a demonstrates the final validation perplexity versus the total number of model parameters for 200 randomly sampled architectures from GPT-2 and Transformer-XL backbones. This figure contains two important observations: (1) the validation perplexity has a downward trend as the number of parameters increases, (2) The discontinuity is caused by the dominance of embedding parameters when moving to the small Transformer regime. We highlight several example points in Figure 6.6a where the architectures are nearly identical but the adaptive input embedding factor k is changed. Changing $k \in \{1, 2, 4\}$ (shown with different colors in Figure 6.6a) varies the total parameter count without much influence on

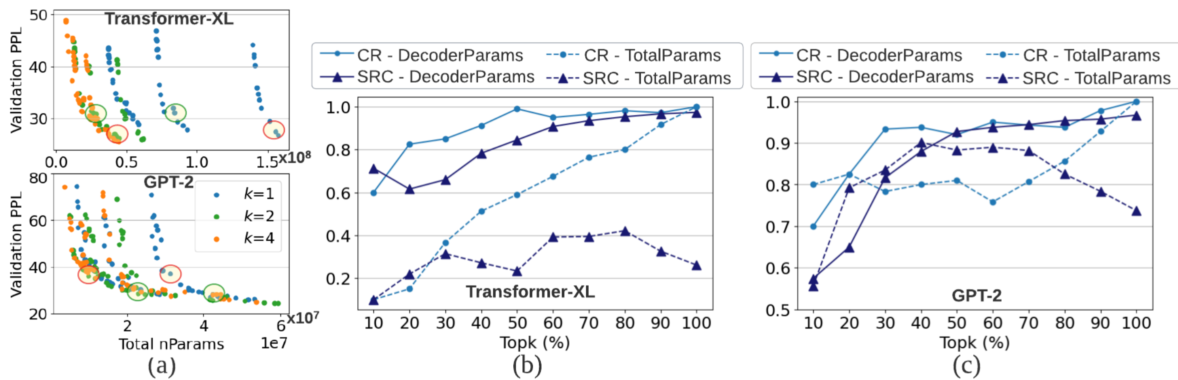

Figure 6.6. (a) Validation perplexity after full training versus total parameters for 200 randomly sampled architectures trained on WikiText-103. The downward trend suggests a strong correlation between parameter count and perplexity. (b), (c) Performance of parameter count proxies for ranking the randomly sampled architectures with Transformer-XL and GPT-2 backbones.

the validation perplexity.

The above observations motivate us to evaluate two proxies, i.e., total number of parameters and decoder parameter count. Figures 6.6b and 6.6c demonstrate the CR and SRC metrics evaluated on the 200 randomly sampled models divided into top$k\%$ bins based on their validation perplexity. As shown, the total number of parameters generally has a lower SRC with the validation perplexity, compared to decoder parameter count. This is due to the masking effect of embedding parameters, particularly in the Transformer-XL backbone. The total number of decoder parameters, however, provides a highly accurate, zero-cost proxy with an SRC of 0.97 with the perplexity over all models, after full training. We further show the high correlation between decoder parameter count and validation perplexity for 100 randomly sampled Transformer architectures with homogeneous decoder blocks in Figure 6.7. As seen, the total parameter count has a low SRC with the validation perplexity while the decoder parameter count provides an accurate proxy with an SRC of 0.95 over all architectures.

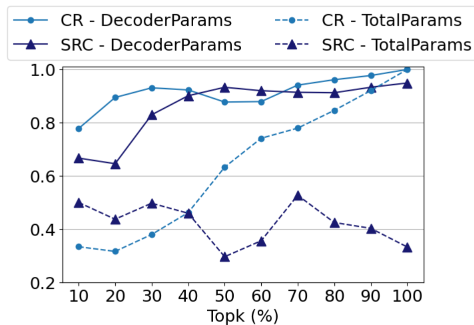

Figure 6.7. Performance of parameter count proxies on 100 randomly sampled Transformers with homogeneous decoder blocks, trained on WikiText-103. The decoder parameter count provides a very accurate ranking proxy with an SRC of 0.95 over all models.

6.3.3 How does variation in model topology affect decoder parameter count as a proxy?

The low-cost proxies introduced in Section 6.2.1, rely on forward and backward passes through the network. As such, they automatically capture the topology of the underlying architecture via the dataflow. The decoder parameter count proxy, however, is topology-agnostic. In this section, we investigate the effect of topology on the performance of decoder parameter count proxy. Specifically, we seek to answer whether for a given decoder parameter count budget, the aspect ratio of the architecture, i.e., trading off the width versus the depth, can affect the final validation perplexity.

We define the aspect ratio of the architecture as d_{model} (=width), divided by n_{layer} (=depth). This metric provides a sense of how skewed the topology is and has been used in prior works which study scaling laws for language models [96]. For a given decoder parameter count budget, we generate several random architectures from the GPT-2 backbone with a wide range of the width-to-depth aspect ratios[3]. The generated models span wide, shallow topologies (e.g., d_{model}=1256, n_{layer}=2) to narrow, deep topologies (e.g., d_{model}=112, n_{layer}=100). Figure 6.8a shows the validation perplexity of said architectures after full training on WikiText-103 versus their

[3]We control the aspect ratio by changing the width, i.e., d_{model} while keeping d_{inner}=$2\times d_{model}$ and n_{head}=8. The number of layers is then derived such that the total parameter count remains the same.

aspect ratio. The maximum deviation (from the median) of the validation perplexity is $< 12.8\%$ for a given decoder parameter count, across a wide range of aspect ratios $\in [1, 630]$. Our findings on the heterogeneous search space complement the empirical results by [96] where decoder parameter count largely determines perplexity for homogeneous Transformer architectures, irrespective of shape (see Figure 5 in [96]). The effect of topology on decoder parameter count proxy for the Transformer-XL backbone is shown in Figure 6.9. Our results demonstrate less than 7% deviation (from the median) in validation perplexity for different aspect ratios $\in [8, 323]$.

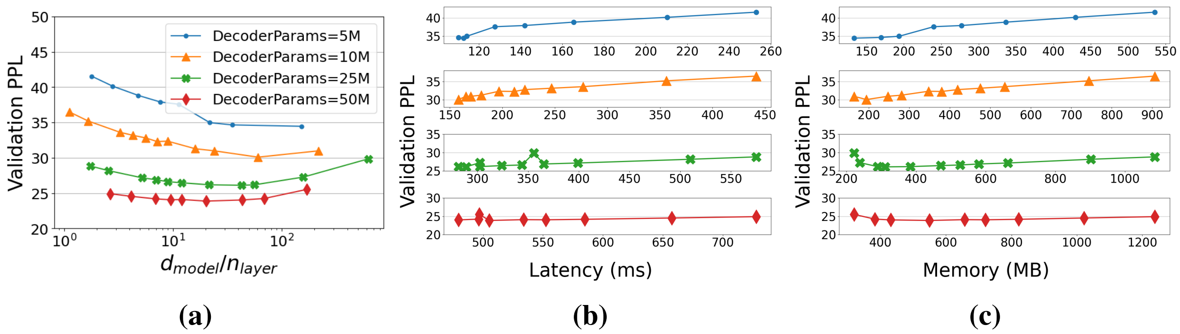

Figure 6.8. Validation perplexity after full training versus the (a) width-to-depth aspect ratio, (b) latency, and (c) peak memory utilization. Models are randomly generated from the GPT-2 backbone and trained on WikiText-103. For a given decoder parameter count, we observe low variation in perplexity across different models, regardless of their topology. The topology, however, significantly affects the latency (up to $2.8\times$) and peak memory utilization (up to $5.5\times$) for models with the same perplexity.

We observe stable training when scaling models from the GPT-2 backbone up to 100 layers, with the perplexity increasing only when the aspect ratio nears 1. For deeper architectures with the Transformer-XL backbone, i.e., more than 40 layers, we observe an increase in the validation perplexity, which results in a deviation from the pattern in Figure 6.9a. This observation is associated with the inherent difficulty in training deeper architectures, which can be mitigated with the proposed techniques in the literature [202]. Nevertheless, such deep models have a high latency, which makes them unsuitable for lightweight inference. For the purposes of hardware-aware and efficient Transformer NAS, our search space contains architectures with less than 16 layers. In this scenario, the decoder parameter count proxy holds a very high correlation

with validation perplexity, regardless of the architecture topology as shown in Figures 6.9a, 6.9a.

Note that while models with the same parameter count have very similar validation perplexities, the topology in fact affects their hardware performance; on the GPT-2 backbone, latency can vary by up to $2.8\times$ and peak memory utilization by up to $5.5\times$ as shown in Figures 6.8b and 6.8c. Similarly, for the Transformer-XL backbone, the latency and peak memory utilization can increase across architectures by $1.3\times$ and $2.0\times$ as shown in Figures 6.9b and 6.9c. This motivates the need for incorporating hardware metrics in NAS to find the best topology.

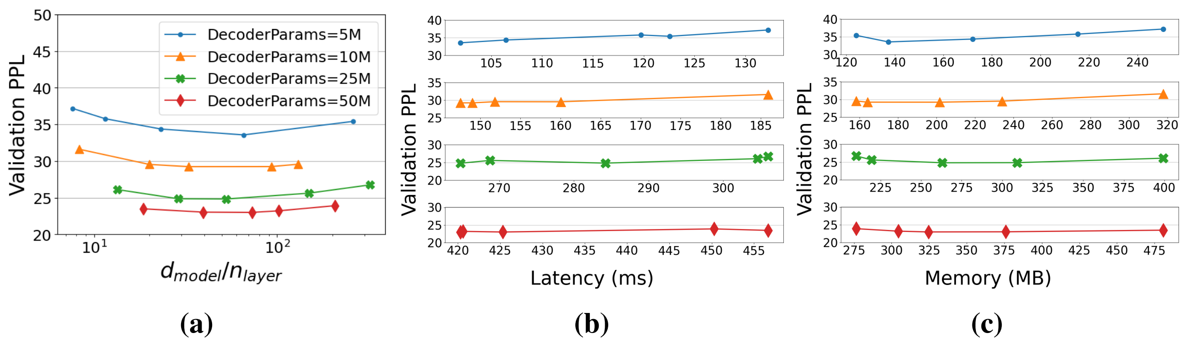

Figure 6.9. Validation perplexity after full training versus (a) the width-to-depth aspect ratio, (b) latency, and (c) peak memory utilization. Models are randomly generated from the Transformer-XL backbone and trained on WikiText-103. For a given decoder parameter count, we observe low variation in perplexity across models, regardless of their topology. The topology, however, significantly affects the latency and peak memory utilization for models with similar perplexity.

6.3.4 How Good is the Decoder Parameters Proxy for Pareto-frontier Search?

In this Section, we validate whether the decoder parameter count proxy actually helps find Pareto-frontier models which are close to the ground truth Pareto front. We first fully train all 1200 architectures sampled from the Transformer-XL backbone during the evolutionary search (Algorithm 4). Using the validation perplexity obtained after full training, we rank all sampled architectures and extract the ground truth Pareto-frontier of perplexity versus latency. We train the models on the WikiText-103 dataset and benchmark Intel Xeon E5-2690 CPU as our target hardware platform for latency measurement in this experiment.

Figure 6.10 represents a scatter plot of the validation perplexity (after full training) versus latency for all sampled architectures during the search. The ground truth Pareto-frontier, by definition, is the lower convex hull of the dark navy dots, corresponding to models with the lowest validation perplexity for any given latency constraint. We mark the Pareto-frontier points found by the training-free proxy with orange color. As shown, the architectures that were selected as the Pareto-frontier by the proxy method are either on or very close to the ground truth Pareto-frontier.

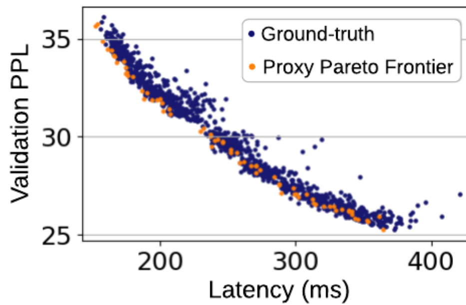

Figure 6.10. Perplexity versus latency Pareto obtained from full training of 1200 architectures sampled during NAS on Transformer-XL backbone. Orange points are the Pareto-frontier extracted using decoder parameter count proxy, which lies close to the actual Pareto-frontier. Decoder parameter count holds an SRC of 0.98 with the ground truth perplexity after full training.

We define the mean average perplexity difference as a metric to evaluate the distance (d_{avg}) between the proxy and ground truth Pareto-frontier:

$$d_{avg} = \frac{1}{N} \sum_{i=1}^{N} \frac{|p_i - p_{gt,i}|}{p_{gt,i}} \tag{6.1}$$

Here, p_i denotes the i-th point on the proxy Pareto front and $p_{gt,i}$ is the closest point, in terms of latency, to p_i on the ground truth Pareto front. The mean average perplexity difference for Figure 6.10 is $d_{avg} = 0.6\%$. This small difference validates the effectiveness of our zero-cost proxy in correctly ranking the sampled architectures and estimating the true Pareto-frontier. In addition to the small distance between the prxoy-estimated Pareto-frontier and the ground truth, our zero-cost proxy holds a high SRC of 0.98 over all 1200 sampled architectures.

We further study the decoder parameter proxy in scenarios where the range of model sizes provided for search is limited. We categorize the total 1200 sampled architectures into

different bins based on the decoder parameters. Figure 6.11 demonstrates the SRC between the decoder parameter count proxy and the validation perplexity after full training for different model sizes. The proposed proxy provides a highly accurate ranking of candidate architectures even when exploring a small range of model sizes.

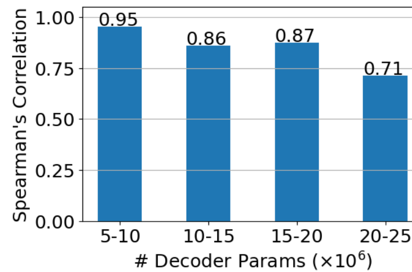

Figure 6.11. SRC between the decoder parameter count proxy and validation perplexity. Results are gathered on 1200 models grouped into four bins based on their decoder parameter count. Our proxy performs well even when exploring within a small range of model sizes.

6.3.5 Pareto-frontier models for various hardware platforms

We run LTS on different target hardware and obtain a range of Pareto-optimal architectures with various latency/memory/perplexity characteristics. During search, we fix the adaptive input embedding factor to $k = 4$ to search models that are lightweight while ensuring nearly on-par validation perplexity with non-adaptive input embedding. As the baseline Pareto, we benchmark the Transformer-XL (base) and GPT-2 (small) models with homogeneous layers $\in [1, 16]$. This is because the straightforward way to produce architectures of different latency/memory is varying the number of layers (layer-scaling) [32, 197]. We compare our NAS-generated architectures with layer-scaled backbones and achieve better validation perplexity and/or lower latency and peak memory utilization. This is because our heterogeneous search space allows us to find a better parameter distribution among decoder layers. All baseline[4] and NAS-generated models are trained using the same setup enclosed in Table 6.1.

LM1B Dataset. Figure 6.12 shows the Pareto-frontier architectures found by LTS versus the

[4]The best reported result in the literature for GPT-2 or Transformer-XL might be different based on the specific training hyperparameters, which is orthogonal to our investigation.

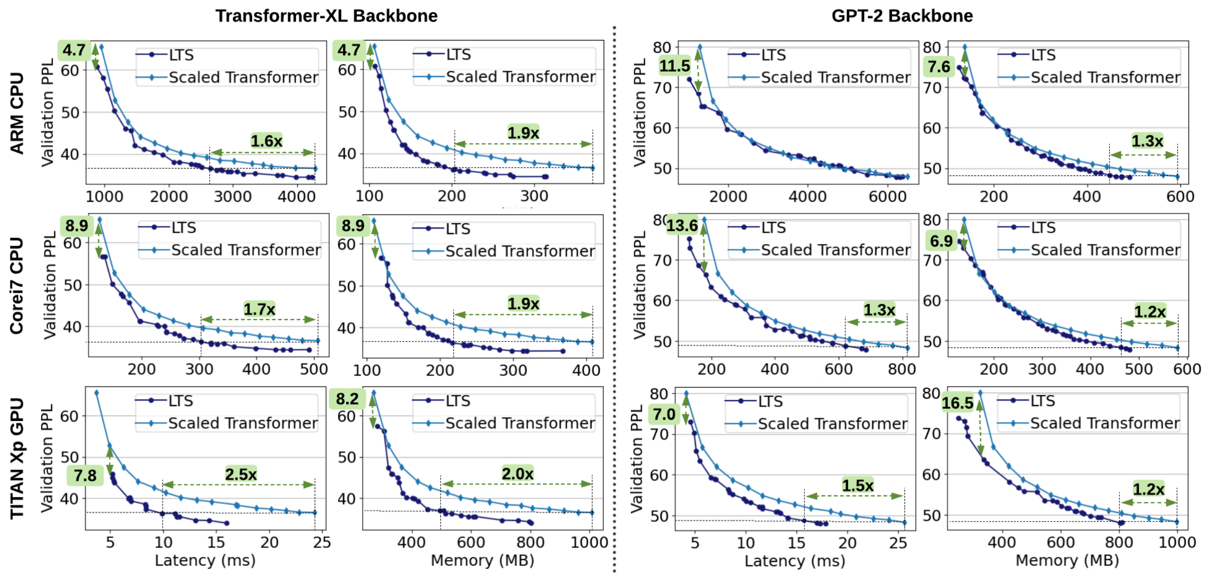

Figure 6.12. 2D visualization of perplexity versus latency and memory Pareto-frontier found by LTS, versus the scaled backbone models with varying number of layers, trained on LM1B.

layer-scaled baseline when trained on the LM1B dataset. Note that the Pareto-frontier search is performed in a 3-dimensional space but for better visualization, in Figure 6.12 we plot 2-dimensional slices of the Pareto-frontier with validation perplexity on the y-axis and one hardware performance metric (either latency or memory) on the x-axis. As seen, in the low-latency regime, LTS consistently finds models that have significantly lower perplexity compared to naive scaling of the baseline Transformer-XL or GPT-2.

On the Transformer-XL backbone, LTS finds architectures with an average of 19.8% and 28.8% lower latency and memory, while achieving similar perplexity compared to the baseline on ARM CPU. Specifically, the perplexity of the 16-layer Transformer-XL base can be replicated on the ARM device with a lightweight model that is $1.6\times$ faster and utilizes $1.9\times$ less memory during execution. On the Corei7 CPU, the Pareto-frontier models found by LTS are on average 25.8% faster and consume 30.0% less memory under the same validation perplexity constraint. In this setting, LTS finds a model that replicates the perplexity of the 16-layer Transformer-XL base while achieving $1.7\times$ faster runtime and $1.9\times$ less peak memory utilization. The savings are even higher on the GPU device, where the NAS-generated models achieve the same perplexity

as the baseline with average 30.5% lower latency and 27.0% less memory. Specifically, an LTS model with the same perplexity as the 16-layer Transformer-XL base has $2.5\times$ lower latency and consumes $2.0\times$ less peak memory on TITAN Xp.

On the GPT-2 backbone, NAS-generated models consume on average 11.8% less memory while achieving the same validation perplexity and latency on an ARM CPU. The benefits are larger on Corei7 and TitanXP where the latency savings are 13.8% and 11.9%, respectively. The peak memory utilization also decreases by 9.7% and 12.9%, on average, compared to the baseline GPT-2s on Corei7 and TITAN Xp. Notably, NAS finds new architectures with the same perplexity as the 16-layer GPT-2 with $1.3\times$, $1.5\times$ faster runtime and $1.2\times$ lower memory utilization on Corei7 and TITAN Xp.

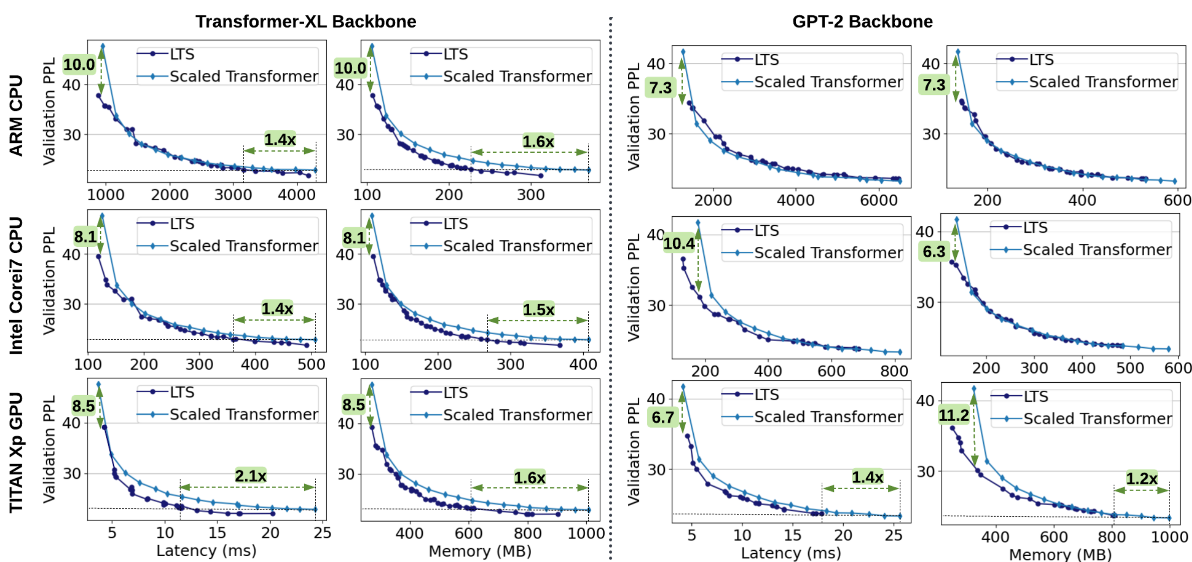

Figure 6.13. 2D visualization of perplexity versus latency and memory Pareto-frontier found by LTS versus layer-scaled backbone models. All models are trained on WikiText-103.

WikiText-103 Dataset. We compare the Pareto-frontier architectures found by LTS with the baseline after full training on the WikiText-103 dataset in Figure 6.13. Commensurate with the findings on the LM1B dataset, the NAS-generated models outperform the baselines in at least one of the three metrics, i.e., perplexity, latency, and peak memory utilization. We note that the gap between the baseline models and those obtained from NAS is larger when training on the

LM1B dataset. This is due to the challenging nature of LM1B, which exceeds the WikiText-103 dataset size by $\sim 10\times$. Thus, it is harder for hand-crafted baseline models to compete with the optimized LTS architectures on LM1B.

On the Transformer-XL backbone, the models on LTS Pareto-frontier for the ARM CPU have, on average, 3.8% faster runtime and 20.7% less memory under the same validation perplexity budget. On the Corei7, the runtime and memory savings increase to 13.2% and 19.6%, respectively, while matching the baseline perplexity. We achieve our highest benefits on TITAN Xp GPU where LTS Pareto-frontier models have, on average, 31.8% lower latency and 21.5% lower memory utilization. Notably, the validation perplexity of the baseline 16-layer Transformer-XL base can be achieved with a lightweight model with $2.1\times$ lower latency while consuming $1.6\times$ less memory at runtime.

On the GPT-2 backbone, LTS achieves $6.3 - 11.2$ lower perplexity in the low-latency-and-memory regime. As we transition to larger models and higher latency, our results show that the GPT-2 architecture is nearly optimal on WikiText-103 when performing inference on a CPU. The benefits are more significant when targeting a GPU; For any given perplexity achieved by the baseline, LTS Pareto-frontier on TITAN Xp delivers, on average, 9.0% lower latency and 4.5% lower memory. Thus, the perplexity and memory of the baseline 16-layer GPT-2 is achieved by a new model that runs $1.4\times$ faster and consumes $1.2\times$ less memory on TITAN Xp.

► **Search Efficiency.** The main component in LTS search time is the latency/peak memory utilization measurement for candidate architectures since evaluating the model perplexity is instant using the decoder parameter count. Therefore, our search finishes in a few hours on commodity hardware, e.g., taking only 0.9, 2.6, and 17.2 hours on a TITAN Xp GPU, Corei7 CPU, and an ARM core, respectively. To provide more context into the timing analysis, full training of even one 16-layer Transformer-XL base model on LM1B using a machine with $8\times$ NVIDIA V100 GPUs takes 15.8 hours. Once the Pareto-frontier models are found, the user can pick a model based on their desired hardware constraints and fully train it on the target dataset. LTS is an alternate paradigm to that of training large supernets; our search runs on the target

device and GPUs are only needed for training the final chosen Pareto-frontier model after search.

In Table 6.2 we study the ranking performance of partial training (500 steps) versus the decoder parameter count proxy over 1200 architectures with the Transformer-XL backbone during LTS search. Astonishingly the decoder parameter count proxy gets higher SRC compared to partial training, while effectively removing training from the inner loop of search for NAS.

Table 6.2. Ranking abilities of full and partial training versus our proxy for 1200 models sampled during LTS search. Training time is reported for WikiText-103 and NVIDIA V100 GPU. Decoder parameter count proxy obtains an SRC of 0.98 using zero compute.

	Train Iter	GPU Hours	CO_2e (lbs)	SRC
Full Training	40,000	19,024	5433	1.0
Partial Training	500	231	66	0.92
	5,000	2690	768	0.96
# Decoder Params	**0**	**0**	∼**0**	**0.98**

6.3.6 Zero and one-shot performance comparison with OPT

Zhang et al. [232] open-source a set of pre-trained decoder-only language models, called OPT, which can be used for zero or few-shot inference on various NLP tasks. Below, we compare the performance of LTS Pareto-frontier models with the hand-designed OPT architecture in zero and one-shot settings. We use LTS to search for language models with a GPT-2 backbone which have 300M to 500M total parameters to compare with the 350M parameter OPT. To cover models with a similar parameter count budget as the OPT-350M model, we search over the following values for the architectural parameters: $\text{n}_{\text{layer}} \in \{3, \dots, 29|1\}$, $\text{d}_{\text{model}} \in \{512, \dots, 1472|64\}$, $\text{d}_{\text{inner}} \in \{512, \dots, 6080|64\}$, and $\text{n}_{\text{head}} \in \{2, 4, 8, 16\}$. To directly compare with OPT, we use a generic, non-adaptive embedding layer for our models. Therefore, the search space does not include the k factor and d_{embed}=d_{model}. The search is conducted with latency as the target hardware metric and decoder parameter count as a proxy for perplexity.

Once the search concludes, We train 20 models from the Pareto-frontier along with OPT-350M on 28B tokens from the Pile [50]. The pretrained models are then evaluated on 14

downstream NLP tasks, namely, HellaSwag [231], PIQA [16], ARC (easy and challenge) [30], OpenBookQA [137], WinoGrande [165], and SuperGLUE [198] benchmarks BoolQ, CB, COPA, WIC, WSC, MultiRC, RTE, and ReCoRD. The training hyperparameters and the evaluation setup are outlined in Section 6.3.1. Figure 6.14 shows the overall average accuracy obtained across all 14 tasks versus the inference latency for LTS models and the baseline OPT. As shown, NAS-generated models achieve a higher average accuracy with lower latency compared to the hand-designed OPT-350M model.

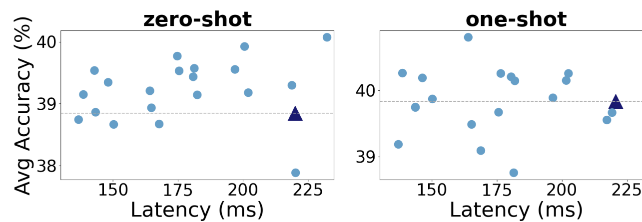

Figure 6.14. Average zero and one-shot accuracy of LTS models (dots) and the baseline OPT-350M (triangle) across 14 NLP tasks. Latency is measured on an A6000 NVIDIA GPU.

► **Zero-shot Performance.** Compared to the OPT-350M architecture, LTS finds models that achieve higher accuracy and lower latency in the zero-shot setting on all evaluated downstream tasks. Specifically, the maximum achievable accuracy of our NAS-generated models is $0.2-8.6\%$ higher than OPT-350M with an average speedup of $1.2\times$. If latency is prioritized, LTS delivers models that are, on average, $1.5\times$ faster and up to 4.6% more accurate than OPT-350M.

► **One-shot Performance.** Similar trends can be observed for one-shot evaluation as shown for different tasks. LTS Pareto-frontier models improve the per-task accuracy of OPT-350M on 12 out of 14 tasks by $0.1 - 8.0\%$, while achieving an average speedup of $1.2\times$. On the same tasks, LTS Pareto-frontier includes models that enjoy up to $1.6\times$ speedup over OPT-350M with an average 1.5% higher accuracy. On the RTE task, the best LTS model has 0.4% lower accuracy but $1.6\times$ faster runtime. On the WSC task, the best performing LTS model obtains a similar one-shot accuracy as OPT-350M, but with $1.5\times$ faster runtime.

6.4 Conclusion

We empirically establish a critical insight that there exists a strong correlation between the number of decoder parameters and final model performance for autoregressive Transformers. Building upon this finding, we develop an efficient on-device search algorithm (LTS) that outputs models on the Pareto-frontier of perplexity versus various hardware metrics, e.g., latency and peak memory utilization. LTS utilizes the decoder parameter count as a training-free and zero-cost proxy for relative ranking of architectures during search. Our search can be performed locally on the target (constrained) platform, where hardware performance metrics, e.g., latency, are directly measured. We provide large-scale proof-of-concept experiments on the effectiveness of decoder parameter count as a ranking proxy using $2900+$ autoregressive Transformers of varying size, backbones, and two large language datasets.

6.5 Acknowledgements

Chapter 6, in part, was published as: M. Javaheripi, G. de Rosa, S. Mukherjee, S. Shah, T. Religa, C. Mendes, S. Bubeck, F. Koushanfar, and D. Dey, "LiteTransformerSearch: Training-free Neural Architecture Search for Efficient Language Models", in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. The dissertation author was the primary investigator and author of this paper.

Chapter 7

Automating Model Compression via Adaptive Non-uniform Sampling

With the growing range of applications for Deep Neural Networks (DNNs), the demand for higher accuracy has led to a continuous increase in the complexity of state-of-the-art models. Such high execution cost hinders the deployment of DNNs in real-time applications on commodity hardware. Fortunately, modern neural networks have been shown to incur high redundancies that can be eliminated without compromising inference accuracy. Effective identification and removal of such redundancies has fueled a myriad of research in two interlinked domains: (i) Developing model compression techniques, e.g., pruning [68, 70, 94, 110, 114, 115, 126, 204, 210] and coding [168]. (ii) Devising automated policies that learn how to configure compression techniques to simultaneously achieve accuracy and compactness [42, 69, 90, 91, 168, 205]. In this work, we focus on the latter.

The effectiveness of contemporary compression techniques relies on careful tuning of several hyperparameters across DNN layers, e.g., pruning rates. These hyperparameters directly control the trade-off between accuracy and execution cost on a constrained device. The question to be answered is how to find an optimal hyperparameter configuration that results in a high compression rate while minimally affecting inference accuracy. Figure 7.1 shows how an intelligent hyperparameter selection policy can better estimate the geometry of the optimal Pareto front for the same compression technique (Pruning). Existing research in automated compression

suggests the use of heuristic methods [68, 70, 94] or Reinforcement Learning (RL) [42, 69, 218]. To tackle the high-dimensionality of the search space, heuristics and RL-based algorithms specify the hyperparameters one layer at a time. One downside of such approach is the need for many learning episodes to enable identification of the inherent inter-layer correlations. This, in turn, results in a rather slow convergence to the optimal hyperparameter solution.

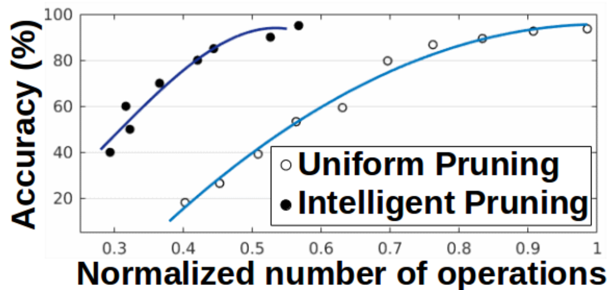

Figure 7.1. Pareto curves of accuracy versus number of floating point operations for pruning a pre-trained VGG network on CIFAR-10 benchmark.

We propose an alternative approach that *simultaneously* tunes the compression hyperparameters for all DNN layers by encapsulating them as *one* fixed-length vector $\vec{x} \in \mathbb{R}^d$. Each hyperparameter vector translates to a unique compressed DNN by leveraging the transformations suggested in existing DNN compression techniques. The search for optimal compression hyperparameters directly translates to optimizing an objective function $f(\vec{x}) : \mathbb{R}^d \rightarrow \mathbb{R}$ over $\vec{x}$ where $f(\cdot)$ is an arbitrary measure of quality. We suggest a unified formulation for $f(\cdot)$, called the *score*, which assesses the quality of $\vec{x}$ by combining accuracy and a desired execution cost.

One major challenge in maximizing $f(\cdot)$ is that the underlying algebraic model that relates compression hyperparameters to inference accuracy (and possibly the execution cost) is not known. In other words, one can evaluate the pertinent objective function $f(\vec{x})$ for any arbitrary input $\vec{x}$ but does not have access to other information such as the gradient $\frac{\partial f}{\partial x}$. Thus, numerical optimization methods such as stochastic gradient descent are not directly applicable. Since the vectorized search-space grows exponentially with number of DNN layers, brute-force search is infeasible, and more intelligent approaches are needed to find the optimal solution.

We resort to empirical evaluations of $f(\cdot)$ and propose *AdaNS*, a non-uniform adaptive sampling methodology that aims at reconstructing the opaque objective function $f(.)$ around its maxima. Different from classic sampling (which accurately reconstructs the function over the entire input space), our sampler's reconstruction objective focuses more on regions of interest, i.e., the maximizers of $f(.)$. To the best of our knowledge, this is the first time that designing compact DNNs, a topic often viewed in the field of computer engineering, has been framed in the context of adaptive sampling, an active area of research in signal processing.

To reduce the total number of required function evaluations while enabling a low-error reconstruction around the maxima, we choose our samples following two incentives: (i) we should sample from likely maximas to reach the optimization goal, and (ii) samples should be drawn from unexplored regions where we have a high reconstruction error (uncertainty). We satisfy the above two properties by realizing targeted sampling. Our proposed algorithm is an iterative process, where a batch of samples (hyperparameter vectors) are obtained and evaluated at each phase. The sampling distribution over the input space for each phase is then refined adaptively based on prior observations before sampling the next batch of hyperparameter vectors. *AdaNS* exploits parallelism to reduce optimization time by concurrent sample evaluations. We devise three adaptive sampling subroutines, namely, *AdaNS-Zoom*, *AdaNS-Genetic*, and *AdaNS-Gaussian*, each of which incorporates a different (posterior) sampling distribution.

We study the impact of the sampling strategy on the convergence rate and the maximal returned value. Our empirical evaluations show that *AdaNS-Gaussian* achieves higher values of $f(\cdot)$ with a lower number of function evaluations. To demonstrate *AdaNS* generalizability, we apply it to optimization tasks of various complexity. Our experiments show that *AdaNS* can learn near-optimal hyperparameters in very high-dimensional search-spaces; to the best of knowledge, *AdaNS* is the only black-box optimization method shown to tackle search-space sizes as high as 10^{132}. We show the superiority of *AdaNS* over prior work in RL [69], Bayesian optimization [26], expert-designed architectures [73, 170, 225], and heuristic methods [68, 70, 82, 94, 114, 115, 124, 126, 141, 166, 204, 210]. We unveil the full potential of *AdaNS* by learning to

effectively combine multiple methods: for VGG-16 on ImageNet, *AdaNS* pushes the state-of-the-art floating-point operation count (FLOPs) reduction from $5\times$ to $7.1\times$ with higher accuracy. For compact MobileNets, *AdaNS* obtains on average 1.2% higher top-1 accuracy than the MobileNet Pareto curve. We further show that *AdaNS* is highly scalable and enjoys a linear search speedup with number of distributed computing resources.

Main Contributions:

- We devise three adaptive sampling strategies, i.e., *AdaNS-Zoom*, *AdaNS-Genetic*, and *AdaNS-Gaussian*, that search for the optimal hyperparameters $\vec{x} \in \mathbb{R}^d$ for DNN compression. Each strategy iteratively refines the sampling posterior distribution towards generating better samples.
- We suggest a unified formulation for the optimization objective $f(\vec{x}) : \mathbb{R}^d \rightarrow \mathbb{R}$, i.e., the *score*. Our customized $f(\cdot)$ simultaneously incorporates inference accuracy and execution cost to quantitatively asses compressed DNNs.
- We propose a context-aware boundary characterization scheme that prevents sampling from regions that are unlikely to contain the optimal solution.
- We connect *AdaNS* to function reconstruction and sampling theory. Specifically, we devise a non-uniform reconstruction scheme and empirically show that *AdaNS* reduces the reconstruction error around the maxima.

7.1 Background and Related Work

Automated Policy Making. Designing automated methodologies for achieving compact and accurate neural networks has been the focus of recent work [69, 84, 87, 168, 205]. In order to achieve high-accuracy and low-complexity neural networks, genetic algorithms have been applied to Neural Architecture Search (NAS) [82] and DNN compression [75]. Reinforcement Learning (RL) [69, 205] is another promising tool for DNN compression. Although effective in finding near-optimal solutions, RL relies on gradient-based training, which can lead to a high

computational burden as well as a slow convergence. Furthermore, RL is not scalable in large continuous action-spaces [166, 217].

We develop a multi-objective optimization method based on an adaptive sampling strategy. *AdaNS* is dimensionality-agnostic and can scale well to large search-spaces. This, in turn, allows for simultaneous optimization of all layers' hyperparameters. Our solution offers several benefits: (1) it is inexpensive to implement since it does not involve gradient-based algorithms. (2) Unlike RL algorithms that are inherently sequential, *AdaNS* is highly parallelizable and can offer scalability in distributed settings. (3) *AdaNS* can support both continuous and sparse-valued objective functions. (4) *AdaNS* is the only known method to-date that optimizes *heterogeneous* parameters in search spaces as large as 10^{132}. Prior work either constrain the search-space size, e.g., [26, 69] or are designed specifically for one compression task, e.g., [225].

Sampling. Our approach to DNN compression is loosely related to three classical problems in the literature: Bayesian empirical optimization [26, 173, 180], spectral methods [65], and adaptive sampling theory [64]. Bayesian empirical optimization provides a method for hyperparameter tuning by assuming a prior distribution for the score function and then updating this prior based on new observations. However, this approach depends on the availability of reliable Bayesian priors which, in some regimes, might not be realistic. *AdaNS* does not make any probabilistic assumptions about the objective function and is compatible with arbitrary $f(\cdot)$. Furthermore, Bandit methods such as Bayesian optimization are inherently serial and difficult to parallelize while our batch implementation allows parallelism in distributed settings.

Spectral methods, when applied to empirical optimization, leverage the structure of the score function together with known results from Fourier analysis and compressed sensing. The main drawback of spectral approaches is their reliance on prior knowledge about the sparse structure of the score function, which cannot be immediately assumed in the context of DNN compression. Adaptive sampling methods are generally well-suited to scenarios where the structure of the score function is unknown but has local features. In such scenarios, sample spacing can be controlled by leveraging the observed values of the variable of interest. Although

adaptive sampling has been amenable to many fields of research, little is concretely known about its optimal strategies.

7.2 Problem Formulation

DNN compression, in high-level, is a transformation $\mathcal{T}(M, \vec{x})$ that converts a pretrained model M to a compressed model $\hat{M}_{\vec{x}}$ with lower computational complexity. In this process, the adjustable hyperparameter vector $\vec{x} \in \mathbb{R}^d$ controls the complexity and accuracy of the output model. A desirable compressed network satisfies two properties: (i) the generalization capability of $\hat{M}_{\vec{x}}$ should resemble the original network and (ii) the execution cost of $\hat{M}_{\vec{x}}$ on the target hardware platform should be as low as possible. We assume we have access to a scoring oracle, $f(\cdot)$, that assesses $\vec{x}$ based on its corresponding compressed model's accuracy $A(\hat{M}_{\vec{x}})$ and complexity $C(\hat{M}_{\vec{x}})$. Our objective is to empirically optimize this customized score:

$$\max_{\vec{x} \in \mathbb{R}^d} f(A(\hat{M}_{\vec{x}}), C(\hat{M}_{\vec{x}})), \tag{7.1}$$

For simplicity, we show $f(A(\hat{M}_{\vec{x}}), C(\hat{M}_{\vec{x}}))$ as $f(\vec{x})$ in the rest of the chapter. Since full knowledge about $f(\cdot)$ and/or its first derivatives cannot be assumed, often empirical evaluations and optimization is the only viable strategy. Brute-force empirical evaluation of $f(\cdot)$ over $\vec{x} \in \mathbb{R}^d$, in general, is infeasible as the search-space grows exponentially with d. Instead, we propose an empirical zeroth order optimization based on adaptive non-uniform sampling, dubbed *AdaNS*, to find the maximum value in Equation (7.1), i.e., f^*.

In the context of DNN compression, $f(\vec{x})$ can be viewed as a band-limited signal, i.e., the corresponding Fourier transform $F(\omega)$ is contained in a frequency interval $[-B, B]$. As such, one can approximately solve the maximization problem in Equation (7.1) by sampling. *AdaNS* iteratively samples the hyperparameter vector space to find an optimized compression configuration $\vec{x}^*$. We consider an adaptive sampler generating $\mathbb{S} = \mathbb{S}_1 \cup \mathbb{S}_2 \cup \cdots \cup \mathbb{S}_T$ where each $\mathbb{S}_t$ represents a fixed-sized set of b samples to be evaluated in iteration t. Let $\vec{x}^*_{\mathbb{S}}$ be the current

maximizer of $f(\cdot)$ on the entire set of observed (evaluated) samples $\mathbb{S}$. Our objective is to select $\mathbb{S}_1, \mathbb{S}_2, \ldots, \mathbb{S}_T$ such that $f(\vec{x}^*_{\mathbb{S}})$ is not too far from the actual function maximum f^*. To this end, *AdaNS* adopts a guided search such that samples generated at each iteration are more competent than the previously observed samples, i.e., they result in better DNNs with higher $f(\vec{x})$.

7.3 *AdaNS* Overview

We provide a generic solution to effectively compress a pre-trained DNN while maximally preserving model accuracy. *AdaNS* automation policy acts on a pool of hyperparameters and explores the corresponding search-space using adaptive non-uniform sampling. An overview of *AdaNS* optimization is shown in Figure 7.2 and summarized below:

I. First, a pre-processing step characterizes the search-space boundaries within which the optimal solution can reside. These boundaries are specified based on task-enforced constraints on inference accuracy. Using the boundaries, initial hyperparameter vectors $\mathbb{S}_1 = \{\vec{x}_1, \ldots, \vec{x}_b\}$ are sampled (Section 7.3.3).

II. Each iteration, newly generated samples $\vec{x} \in \mathbb{S}_t$ are translated to their compressed DNNs $\hat{M}_{\vec{x}}$ (Section 7.3.1). The scores $f(\vec{x})$ are then evaluated in parallel (Section 7.3.2).

III. Based on the knowledge acquired by new evaluations and previously observed samples, *AdaNS* adaptive sampling subroutine identifies a batch of more competent hyperparameter vectors $\mathbb{S}_{t+1}$. We propose three different subroutines that select the new set of samples, namely, *AdaNS-Zoom*, *AdaNS-Genetic*, and *AdaNS-Gaussian* (Section 7.4).

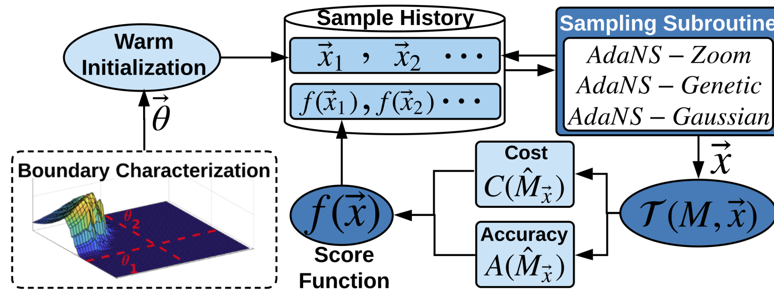

Figure 7.2. Overview of *AdaNS* adaptive sampling for hyperparameter customization.

7.3.1 Search-Space Definition

An initial step for the application of *AdaNS* is defining the pertinent search-space for black-box optimization. To this end, we propose a vectorized representation of the hyperparameters for various compression methods. The problem of compression optimization can then be solved by performing a search over this vector-space. Throughout this chapter, we focus on *four* compression tasks, namely, structured [70] and non-structured [62] Pruning, Singular Value Decomposition (SVD) [235], and Tucker-2 approximation [98].

Figure 7.3 shows our vectorized compression hyperparameters for a 4-layer neural network. For pruning, we allocate one continuous value $p \in [0, 1]$, per layer, inside $\vec{x}$ to represent the ratio of non-zero values. We apply SVD on the weights of fully-connected layers ($W \in \mathbb{R}^{c\times f}$) and point-wise convolutions ($W \in \mathbb{R}^{1\times1\times c\times f}$). To represent the integer-valued rank $\in \{1, \ldots, R\}$ for SVD, we allocate one continuous value $rank \in [\frac{1}{R}, 1]$ per decomposed layer to form $\vec{x}$, where $R = min(c, f)$. Tucker-2 is applied on four-way tensors $W \in \mathbb{R}^{k\times k\times c\times f}$ in convolution layers where the compression parameter is a tuple of ranks. To form $\vec{x}$, we assign two normalized, real-valued hyperparameters $rank_1 \in [\frac{1}{c}, 1]$ and $rank_2 \in [\frac{1}{f}, 1]$ per convolution.

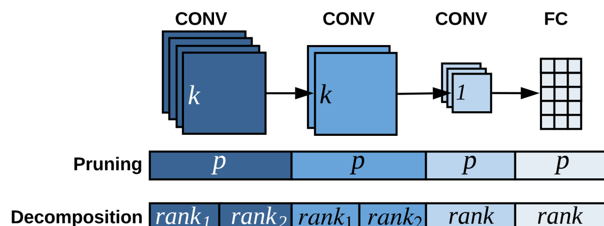

Figure 7.3. Vectorized representation of an example 4-layer DNN for *Pruning* and *Decomposition*. Here, CONV and FC denote convolutional and fully-connected layers, respectively.

7.3.2 Scoring Mechanism

We formalize a multi-objective score $f(\vec{x})$ which simultaneously reflects the compressed DNN's accuracy and computational complexity. This score can thus be leveraged to assess the compression quality of each hyperparameter vector $\vec{x}$. To define $f(\cdot)$, let us first represent DNN

compression as a constrained optimization as follows:

$$\max_{\vec{x}\in\mathbb{R}^d} \Delta C(M,\vec{x}) \quad s.t. \quad A(\hat{M}_{\vec{x}}) > A_{thr} \tag{7.2}$$

where $\Delta C(M,\vec{x})$ represents the normalized difference in hardware cost, e.g., FLOPs, between the uncompressed network, M, and the compressed model, $\hat{M}_{\vec{x}}$. Here, A_{thr} is a task-enforced threshold on the post-compression accuracy. Having an accuracy constraint is crucial since the optimization algorithm will converge to a model size of zero otherwise. To solve the constrained optimization problem in Equation (7.2), we propose to formulate it as the following primal unconstrained optimization using penalty methods [8]:

$$\max_{\vec{x}\in\mathbb{R}^d} \Delta C(M,\vec{x}) - log(PEN_A(\vec{x}))) \tag{7.3}$$

where the term $log(PEN_A(\vec{x}))$ is the exterior penalty function [28] that enforces a constraint on the compressed model's accuracy, i.e., $A(\hat{M}_{\vec{x}}) > A_{thr}$. The function $PEN_A(\vec{x})$ measures the accuracy degradation as follows:

$$PEN_A(\vec{x}) = \begin{cases} \Delta A(M,\vec{x}) & A(\hat{M}_{\vec{x}}) \geq A_{thr} \\ \Delta A(M,\vec{x}) + e^{[A_{thr} - A(\hat{M}_{\vec{x}})]} & A(\hat{M}_{\vec{x}}) < A_{thr} \end{cases} \tag{7.4}$$

where $\Delta A(M,\vec{x}) = A(M) - A(\hat{M}_{\vec{x}})$ and $A(M)$ is the baseline accuracy of the uncompressed model. Figure 7.4 visualizes the accuracy penalty. To prevent undesirable drop of accuracy, we greatly diminish the score of individuals that cause lower accuracies than the set constraint, A_{thr}. The log penalty term is estimated as follows:

$$log(PEN_A(\vec{x}))) = \begin{cases} log(\Delta A(M,\vec{x})) & A(\hat{M}_{\vec{x}}) \geq A_{thr} \\ A_{thr} - A(\hat{M}_{\vec{x}}) & A(\hat{M}_{\vec{x}}) < A_{thr} \end{cases} \tag{7.5}$$

For accuracy values satisfying the threshold, this term enforces the accuracy maximization objective. Applying the logarithm smoothens the accuracy variations by damping sudden changes.

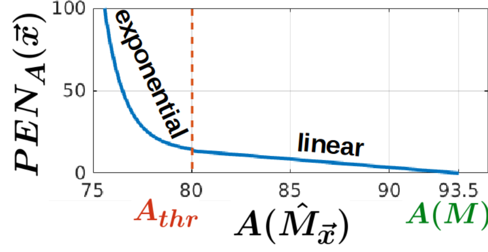

Figure 7.4. *AdaNS* accuracy penalty function with $A_{thr} = 80\%$ and $A(M) = 93.5\%$.

For accuracy values below A_{thr}, a linear penalty is applied to stop further accuracy loss. To avoid numerical instability, we define the exponential of the primal optimization in Equation (7.3) as our score function:

$$f(\vec{x}) = \frac{e^{\Delta C(M,\vec{x})}}{PEN_A(\vec{x})} \tag{7.6}$$

Maximizing the score function of Equation (7.6) is equivalent to maximizing its logarithm value in Equation (7.3). To ensure efficiency, inference accuracies are measured on a small held-out portion of the training data, dubbed *validation* set. *AdaNS* score function successfully models the goal of high compression with minimal accuracy loss; it is applicable to various compression tasks and can reflect different hardware costs, e.g., power, memory, and runtime.

7.3.3 Boundary Characterization for Directed Search

A naïve initialization of samples in the first iteration can result in slow and sub-optimal convergence. We utilize boundary characterization as a pre-processing step to enable a targeted initialization. This approach eliminates unnecessary exploration of outlier subspaces, i.e., regions that are unlikely to contain the optimization solution. Inference accuracy for a compressed DNN $\hat{M}_{\vec{x}}$ is coordinate-wise monotonic with respect to per-layer compression rates: as the compression rates increase, the accuracy drops. As such, we can characterize the boundaries of $\vec{x}[i]$ on a per-layer basis, based on the accuracy threshold without performing any fine-tuning. Figure 7.5 visualizes the hyperparameter search-space and the outlier regions for pruning a two-layer model.

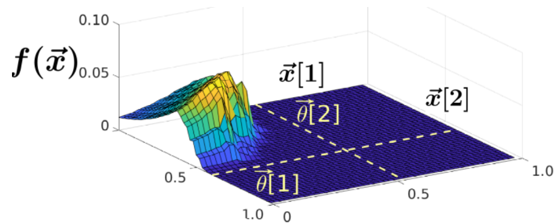

Figure 7.5. Search-space for pruning a 2-layer DNN.

The optimal solution to the search problem is a configuration with the highest score. The outlier regions in Figure 7.5 are therefore the flat sectors of the space. We find a threshold vector $\vec{\theta}$ where each element constrains a single hyperparameter corresponding to the compressed DNN. In Figure 7.5, $\vec{\theta}$ has two elements, each presented by a dashed line. Below we describe how the boundaries $\vec{\theta}[i]$ are obtained given an accuracy threshold A_{thr} for each compression task.

Pruning. The threshold vector elements $\vec{\theta}[i] \in [0, 1)$ specify the maximum pruning rate for layer i such that the compressed DNN's accuracy does not violate A_{thr}:

$$\vec{\theta}[i] = max\{p\} \;\; s.t. \;\; \vec{x}[j] = \begin{cases} p & j = i \\ 0 & j \neq i \end{cases}, \; A(\hat{M}_{\vec{x}}) > A_{thr}$$

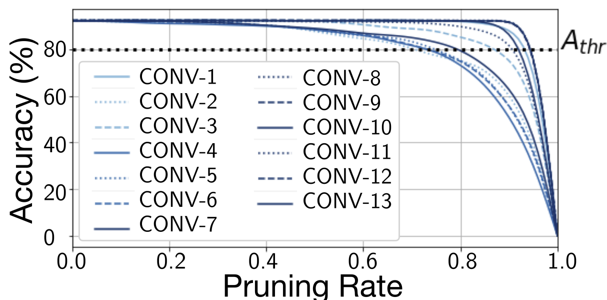

Figure 7.6. Boundary characterization for pruning a VGG model on CIFAR-10 with $A_{thr} = 80\%$.

Figure 7.6 demonstrates an example of boundary characterization for pruning a VGG network on CIFAR-10. Each curve is obtained by varying the pruning rate for one layer while no other layer is pruned. The collision between the dashed horizontal line, i.e., the accuracy threshold, and the ith curve provides the threshold boundary θ_i.

Decomposition. For SVD and Tucker-2 decomposition, the threshold vector $\vec{\theta}$ represents per-layer minimum ranks $\vec{\theta}[i] \in (0, 1]$ satisfying A_{thr} where a normalized rank of 1 corresponds to a non-decomposed layer:

$$\vec{\theta}[i] = min\{rank\} \;\; s.t. \;\; x_j = \begin{cases} rank & j = i \\ 1 & j \neq i \end{cases}, \; A(\hat{M}_{\vec{x}}) > A_{thr}$$

Note that real-world models have many more layers and the pertinent search-space is of much higher dimensionality than in Figure 7.5. For a d-dimensional space, the proposed boundary characterization scheme reduces the effective (continuous) search volume from 1 to $\prod_{i=1}^{d} \vec{\theta}[i]$ for pruning and $\prod_{i=1}^{d}(1 - \vec{\theta}[i])$ for decomposition, therefore, significantly improving search convergence and solution quality.

By filtering out the non-optimal regions, *AdaNS* sampling can find the near-optimal solution within the found margins. After boundary characterization, we randomly draw the initial samples from the space enclosed by the threshold vector $\vec{\theta}$. For pruning, the ith element $\vec{x}[i]$ in each sample vector is drawn from $\mathcal{N}(\vec{\theta}[i]/2, \vec{\theta}[i]/2)$. For decomposition, the ith element $\vec{x}[i]$ is randomly selected from $\mathcal{U}[\vec{\theta}[i], 1]$.

7.3.4 Optimization through Adaptive Sampling

In this section, we enclose the mathematical formulation of *AdaNS* adaptive sampling and its connection to optimization. Let us consider a pool of samples $\mathbb{S} = \{\vec{x}_i\}_{i=1}^{N}$ which are generated iteratively and then evaluated via the scoring oracle. Also, let $\vec{x}^*_{\mathbb{S}}$ be the current maximizer for the objective function $f(\cdot)$, found across the observed samples. To maximize the probability of finding a near-optimal solution, we seek the following property on $\mathbb{S}$:

$$\max_{\mathbb{S}} \mathcal{P}(f(\vec{x}^*_{\mathbb{S}}) > \alpha_{max} f^*), \quad 0 < \alpha_{max} < 1 \tag{7.7}$$

where $\mathcal{P}(\cdot)$ denotes probability and α_{max} is a constant value called the *proximity parameter*. As $\alpha_{max} \rightarrow 1$, the sampled maxima get closer to the real function maximum: $f(\vec{x}^*_{\mathbb{S}}) \rightarrow f^*$. To reach

the objective of Equation (7.7), we devise an adaptive sampler to generate $\mathbb{S}$ based on two ideas:

- We choose our samples ($\mathbb{S}$) in a sequential fashion: $\mathbb{S}_1,\ \mathbb{S}_2,\ \ldots,\ \mathbb{S}_T$[1] such that $f(\mathbb{S}_{[1:t]})$ is utilized[2] in choosing $\mathbb{S}_{t+1}$.
- We allow $\mathbb{S}_1,\ \ldots,\ \mathbb{S}_T$ to be random sets. That is, instead of choosing $\mathbb{S}_{t+1}$ deterministically, we devise a sampling distribution $\mathcal{G}_{t+1}(\cdot)$ such that $\mathbb{S}_{t+1} \sim \mathcal{G}_{t+1}(\cdot)$. More precisely, $\mathbb{S}_{t+1}$ is a random draw from the conditional distribution $\mathcal{G}_{t+1|t}(\cdot|\mathbb{S}_{[1:t]})$.

Algorithm 5 presents a high-level overview of one iteration in *AdaNS* sampling methodology. At each iteration t, the current estimate of the function maxima $\widetilde{f}^*$ is obtained based on the previous observations (Line 1). *AdaNS* uses a proximity parameter $\alpha_t \in [0, 1]$ to extract the set of good samples $\mathbb{S}_g$ from the history of previous observations $\mathbb{S}_{[1:t]}$ where $\mathbb{S}_g$ is the set of samples $\vec{x} \in \mathbb{S}_{[1:t]}$ with $f(\vec{x}) > \alpha_t \widetilde{f}^*$ (Lines 2 & 3). A large value for the proximity parameter (≈ 1) is ideal since it allows for better identification of the function maximum. However, setting the proximity parameter to a high initial value causes the underlying sampling to become biased towards the initial good samples when the local maxima are not yet found. To mitigate this issue, we adaptively tune α_t throughout iterations, such that $\mathbb{S}_g$ has a diverse set of members, therefore balancing exploration and exploitation in the sampling distributions. At each iteration, α_t is set to the maximum value within $[0, \alpha_{max}]$ that allows $\mathbb{S}_g$ to have at least b members (Line 2). In practice, α_t becomes close to zero in the initial search iterations and gradually increases towards $\alpha_{max} \approx 0.95$ as the *AdaNS* search algorithm proceeds.

Once the set of good samples $\mathbb{S}_g$ is extracted, a sampling subroutine is applied to generate a new set of samples, $\mathbb{S}_{t+1}$. The sampling subroutine utilizes $\mathbb{S}_g$ to update the sampling distribution $\mathcal{G}_{t+1}$ from which the next batch of samples $\mathbb{S}_{t+1}$ are drawn (Lines 5 & 6). The details of *AdaNS* sampling subroutines are enclosed in Section 7.4. The newly generated samples are

[1] $\mathbb{S}_t$ is the set of samples from the tth iteration of *AdaNS*.

[2] We use the notation $f(\mathbb{S})$ as follows: $f(\mathbb{S}) = \{f(\vec{x})|\vec{x} \in \mathbb{S}\}$

then evaluated and appended to the sample history. This process is repeated until convergence, or up to a maximum number of iterations (T).

Algorithm 5. Overview of *AdaNS* Sampling

Inputs: Previous samples $\mathbb{S}_{[1:t]}$, fitness oracle $f(\cdot)$, batch size b, proximity parameter α_{max}.
Outputs: $\mathbb{S}_{[1:t]}$, $f(\mathbb{S}_{[1:t]})$.

1: $\widetilde{f}^* = \max \; f(\mathbb{S}_{[1:t]})$
2: $\alpha_t = \max_{\alpha \in [0, \alpha_{max}]}(\alpha) \;\; s.t. \;\; |\{\vec{x} \in \mathbb{S}_{[1:t]} | f(\vec{x}) > \alpha \widetilde{f}^*\}| \geq b$
3: $\mathbb{S}_g = \{\vec{x} \in \mathbb{S} | f(\vec{x}) > \alpha_t \widetilde{f}^*\}$
4: **procedure** SAMPLING SUBROUTINE
5: Update sampling distribution $\mathcal{G}_{t+1}(\cdot)$ based on $\mathbb{S}_g$
6: Sample $\mathbb{S}_{t+1} = \{\vec{x}_i\}_{i=1}^{b} \sim \mathcal{G}_{t+1}(\cdot)$
7: **return** $\mathbb{S}_{t+1}$
8: $\mathbb{S}_{[1:t+1]} = \mathbb{S}_{t+1} \cup \mathbb{S}_{[1:t]}$
9: $f(\mathbb{S}_{[1:t+1]}) = f(\mathbb{S}_{t+1}) \cup f(\mathbb{S}_{[1:t]})$

7.4 *AdaNS* Adaptive Sampling Routines

Recall that *AdaNS* iteratively establishes a set of evaluated samples, $\mathbb{S}_{[1:T]} = \mathbb{S}_1 \cup \cdots \cup \mathbb{S}_T$, such that the samples generated at each iteration are more competent than the previously observed samples. In other words:

$$\max f(\mathbb{S}_{[1:t+1]}) \geq \max f(\mathbb{S}_{[1:t]}) \tag{7.8}$$

This, in turn, ensures that the evaluated samples gradually move closer to the objective function maximizer. The choice of new samples at each iteration in *AdaNS* is based on the adaptive sampling distribution $\mathcal{G}_{t+1}(\cdot)$ utilized in the sampling subroutine (see Algorithm 5). The choice of $\mathcal{G}_{t+1}(\cdot)$ directly affects optimization performance, i.e., the maximal returned value for the objective function as well as the convergence time. To investigate such effects, we design three different sampling subroutines for *AdaNS*, namely, *AdaNS-Zoom*, *AdaNS-Genetic*, and

AdaNS-Gaussian. While each subroutine incorporates a different prior for $\mathcal{G}_{t+1}(\cdot)$, all of the sampling distributions are developed based on the following key insights:

❶ In the absence of prior knowledge, uniform sampling provides the most effective exploration of the functional landscape of the optimization objective $f(\cdot)$.

❷ Assuming mild regularity conditions on $f(\cdot)$, it is intuitive that increasing sampling rate locally around "good samples" observed so far is more likely to result in additional (future) high-score observations.

❸ Assuming mild regularity conditions on $f(\cdot)$, it is intuitive that the line connecting two good samples is likely to be aligned with the optimal (but unknown) gradient ascend.

In the following, we explain how the above insights are leveraged to design the sampling policy for *AdaNS* subroutines.

7.4.1 *AdaNS-Zoom* Sampling Subroutine

Our first sampling subroutine, i.e., *AdaNS-Zoom*, establishes $\mathcal{G}_{t+1}(\cdot)$ using a set of uniform probability distributions as the prior. Uniform sampling requires a large number of samples that maximally cover the optimization space, in order to locate the objective function maximizers. This solution, however, is infeasible for the problem at hand as the number of required samples and evaluations increases exponentially with the number of DNN layers. To alleviate this high sample count, we propose an adaptive methodology that intelligently distributes random samples across the search-space. *AdaNS-Zoom* iteratively (1) divides the space into multiple sub-regions and (2) adaptively tunes the per-region sample density.

Division. This step determines the boundaries that divide up the search-space into the aforementioned sub-regions. Our division algorithm gradually generates a hierarchy of sub-regions based on the information acquired from previously observed samples. We start with the whole search-space considered as one sub-region as shown in Figure 7.7-ⓐ. Our goal is to use samples

to accurately characterize sub-regions that are more likely to contain near-optimal solutions. To increase sampling resolution in such *good* areas, *AdaNS* gradually zooms into high-quality sub-regions by dividing them in half as shown in Figure 7.7-ⓑ and 7.7-ⓓ. We quantify the quality of the ith sub-region by its share of good samples w_i:

$$w_i = \frac{|\mathbb{S}_g|_i}{|\mathbb{S}_{[1:t]}|_i} \tag{7.9}$$

where $|\cdot|_i$ counts the total number of samples contained in sub-region i throughout all iterations. We then choose the sub-region with maximum w_i in each iteration to be divided in half along its longest dimension.

Sampling. In the first iteration of the algorithm, we uniformly sample the entire search-space since no prior knowledge is available. Throughout next iterations, after each division takes place, new w_is are computed to reflect the portion of good samples lying in the new sub-regions. To generate a new sample, we randomly select a sub-region with w_i representing the (relative) chance of the ith region being selected. We then draw a sample from a uniform distribution over the selected sub-region. By repeating this process b times, we obtain the next batch of samples $\mathbb{S}_{t+1} = \{\vec{x}_i\}_{i=1}^{b}$ in Algorithm 5, Line 6.

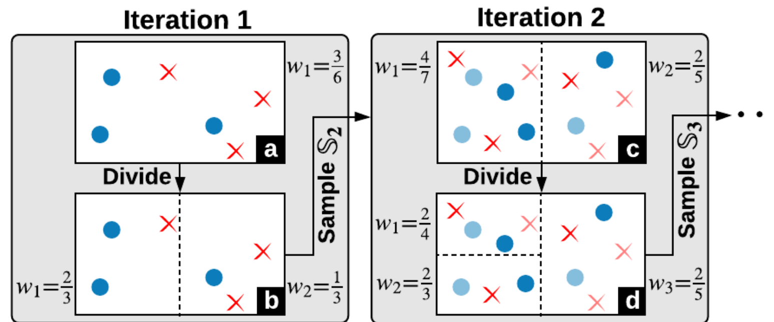

Figure 7.7. *AdaNS-Zoom* algorithm. Here, the good and bad samples are shown with blue circles and red crosses, respectively.

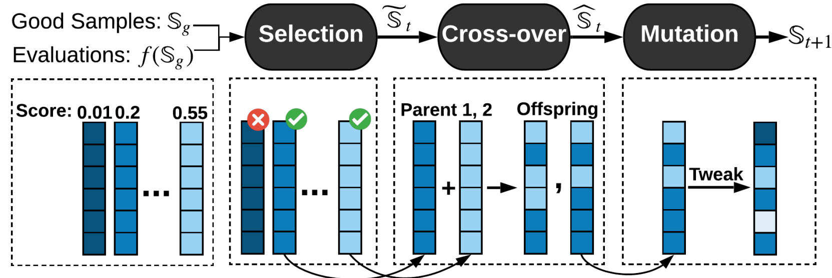

Figure 7.8. Overview of *AdaNS-Genetic* sampling subroutine.

7.4.2 *AdaNS-Genetic* Sampling Subroutine

To generate higher quality samples based on previous observations, our second subroutine utilizes a genetic algorithm. Genetic algorithms are metaheuristic approaches inspired by natural evolution and the notion of "survival of the fittest". These methods can be leveraged as powerful tools to explore large search-spaces while enjoying high scalability and significantly low training overhead [159, 166, 212].

At iteration t, the genetic algorithm evolves the previously evaluated good samples $\mathbb{S}_g$ into a new, more competent batch of samples. This is achieved by performing a set of bio-inspired operations, i.e., selection, crossover, and mutation, shown in Figure 7.8. Below we delineate the details of each aforementioned operation.

Selection. The selection step chooses high-quality samples $\widetilde{\mathbb{S}}_t = \{\widetilde{x}_1 \dots, \widetilde{x}_b\}$ from $\mathbb{S}_g$ that will be utilized in generating the next batch of samples. This is done by performing a non-uniform sampling (without replacement) from $\mathbb{S}_g$, where the probability of selecting each sample $\vec{x}_i^g$ is proportional to its score $f_i = f(\vec{x}_i^g)$. We normalize the scores as follows:

$$f_i \leftarrow \frac{f_i - f_{min}}{\sum_{i=1}^{K}(f_i - f_{min})}, \tag{7.10}$$

Here, f_{min} is the minimum score within $f(\mathbb{S}_g)$. Subtraction of the minimum score ensures that the probability of selecting the lowest-quality sample is zero and it is always eliminated. Such score-based selection is inherently random and allows for exploration. As the same time,

due to the score-proportionate selection, high-quality samples are more likely to appear in the selection. This approach enables *AdaNS-Genetic* sampling to achieve a balance between exploration/exploitation.

Crossover. We design a crossover operation that creates new samples by inheriting and combining hyperparameters from a pair of parent samples. The crossover is performed by randomly swapping corresponding elements of parent hyperparameter vectors. Given the selected set $\widetilde{\mathbb{S}}_t = \{\widetilde{x}_1 \dots \widetilde{x}_b\}$, we sort the individuals in descending order based on their score. To form the crossover pairs, we pick the best sample available as the first parent. We then choose the sample which has the highest *distance* with the first parent as the second parent. Such a distance-based selection of pairs is motivated by increasing diversity among the newly generated offspring and promoting exploration. For two samples $\widetilde{x}_1, \widetilde{x}_2 \in \mathbb{R}^d$, the distance is calculated as follows:

$$dist(\widetilde{x}_1, \widetilde{x}_2) = \frac{1}{d \times K} \sqrt{\sum_{i=1}^{d} (\widetilde{x}_1[i] - \widetilde{x}_2[i])^2} \tag{7.11}$$

where K is a constant equal to the maximum allowed value for the corresponding hyperparameter. We use two parameters to control the degree of crossover operation: p_{cross} determines the probability of applying crossover between two samples, and p_{swap} is the per-element swapping probability. The intuition behind crossover is to allow high-quality hyperparameter configurations to exchange learned patterns and enable knowledge transfer across samples.

Mutation. Mutation randomly tweaks the elements in each sample vector in the crossover-ed set $\widehat{\mathbb{S}}_t$. Each element of a sample $\widehat{x}_i \in \mathbb{R}^d$ is mutated by adding a random value drawn from a zero-mean Normal distribution $\mathcal{N}(0,\, 0.2)$. We then clip the values to ensure they remain in the valid range, i.e., $[0, 1]$. Similar to crossover, we define two control parameters: p_{mutate} is the probability that the sample gets mutated and p_{tweak} determines the per-element tweaking probability. Mutation allows for the exploration of neighborhoods around the selected points. To maintain the balance between exploration and exploitation and avoid premature convergence, we tune the mutation parameters based on the diversity of samples in $\widehat{\mathbb{S}}_t$. We use the dispersion of

samples and define diversity as follows:

$$div(\widehat{\mathbb{S}}_t) = \frac{1}{N}\sum_{n=1}^{N}\left[\sum_{i=1}^{d}(\hat{x}_n[i] - \vec{\mu}_{\hat{x}}[i])^2\right] \tag{7.12}$$

where $\vec{\mu}_{\hat{x}} \in \mathbb{R}^d$ is the mean of all samples in $\widehat{\mathbb{S}}_t$. As can be seen, the diversity function is closely tied to the variance. The diversity can, therefore, be adjusted by controlling the per-element variance of the samples in the crossovered set $Var[\widehat{X}_i]$. Let us denote an arbitrary individual after crossover by $\widehat{x} \in \widehat{\mathbb{S}}_t$, which transforms to $\vec{y} \in \mathbb{S}_{t+1}$ after mutation. The ith element of $\vec{y}$ is thus sampled from a random variable Y_i with the following probability distribution:

$$P(Y_i = \vec{y}[i]) = \begin{cases} p_M & \vec{y}[i] = \hat{x}[i] + \eta \\ 1 - p_M & \vec{y}[i] = \hat{x}[i] \end{cases} \tag{7.13}$$

where $P_M = P_{mutate} \times P_{tweak}$ and η is the random perturbation applied during mutation. Note that $\hat{x}$ is also a random variable. The per-element variance of $\vec{y}$ is thus:

$$Var[Y_i] = Var[\widehat{X}_i] + p_M\sigma_\eta^2 \tag{7.14}$$

Here, σ_η^2 is the variance of the added perturbation. Summing up the vector values in Equation (7.14) provides the new population diversity:

$$div(\mathbb{S}_{t+1}) = div(\widehat{\mathbb{S}}_t) + d \times P_M\sigma_\eta^2 \tag{7.15}$$

For a desired threshold on population diversity, we can, therefore, determine the mutation parameters using Equation (7.15). Since tweaking multiple elements of the individual vector can result in drastic changes in the corresponding compressed DNN's architecture and accuracy, we restrict P_{tweak} to a small value ($P_{tweak} = 0.05$) and merely adjust P_{mutate} for diversity control. In our experiments, we set the diversity threshold to be half the diversity for the randomly initialized population. Such adaptive tuning of mutation allows for a diversity-guided search and ensures a fast and stable convergence.

7.4.3 *AdaNS-Gaussian* Sampling Subroutine

Our last sampling subroutine utilizes a combination of Gaussian and uniform distributions to model the adaptive sampling distribution $\mathcal{G}(\cdot)$ based on the previously observed good samples $\mathbb{S}_g$. Specifically, we draw the new set of samples from a combination of three sampling distributions that collectively form $\mathcal{G}(\cdot)$ as illustrated in Figure 7.9; here, previously observed good samples are shown in dark blue, red crosses denote observed low-score (bad) samples, and the sampling density is marked with light blue. We design each of *AdaNS-Gaussian* sampling distributions to address one of the insights mentioned at the beginning of Section 7.4:

❶ We draw a portion of new samples uniformly from $\mathbb{X} \subset [0, 1]^d$ (Figure 7.9a), dubbed *Uniform* samples. This policy explores the whole space in the absence of prior knowledge.

❷ We draw another portion of samples, dubbed *Local*, from the vicinity of good samples $\mathbb{S}_g$ (Figure 7.9b). The underlying PDF is a mixture of Gaussians, with centers located at $\mathbb{S}_g$.

❸ We draw the last portion of samples, dubbed *Cross*, from a mixture of Gaussians centered in the mid-points of good samples $\mathbb{S}_g$ (Figure 7.9c). This policy has a high sampling density along the line connecting two good samples.

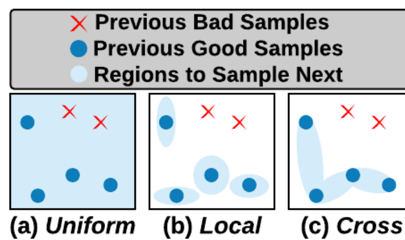

Figure 7.9. $2D$ illustration of *AdaNS-Gaussian* sampling strategies.

The intuitions behind *AdaNS-Gaussian* sampling strategy rest upon a line of work in sampling theory [61,63,221] that prove Gaussian Kernels with adaptive variances can reconstruct smooth non-linear functions. Inspired by that, we vary the Gaussian variance spatially according to local information about the approximand [63]; the parameters of the Gaussian Mixture Models

(GMMs) are chosen to maximize the likelihood of previously successful samples. Below we elaborate on the above sampling policies.

***Uniform* Samples.** To allow *AdaNS-Gaussian* to explore unseen regions, we select a portion of samples uniformly at random from the entire search-space. This prevents the search from becoming too localized upon observing several good samples in a small region.

***Local* Samples.** To effectively explore the vicinity of good samples, we use a GMM for sampling [236]. Formally, the sampling PDF is:

$$\mathcal{P}(\vec{x}) = \sum_{i=1}^{K} w_i \, \mathcal{N}(\vec{x}_i^g, \Sigma), \qquad \vec{x}_i^g \in \mathbb{S}_g \tag{7.16}$$

where $\mathcal{N}(\vec{x}, \Sigma)$ is a multi-variate Gaussian distribution [125] with mean vector $\vec{x} \in \mathbb{R}^d$ and diagonal covariance matrix represented with $\Sigma \in \mathbb{R}^d \times \mathbb{R}^d$. Here, w_i is a weight parameter that adjusts the probability of choosing the ith multi-variate Gaussian. We set the weights proportional to the value of the objective function, i.e., $w_i \propto f(\vec{x_i}^g)$. The standard deviation Σ is determined such that the Gaussians $\mathcal{N}(\cdot, \cdot)$ cover the so-far observed span of $\mathbb{S}_g$:

$$\begin{cases} \sigma_{jj}^2 = (\max\limits_{\vec{x} \in \mathbb{S}_g} \vec{x}[j] - \min\limits_{\vec{x} \in \mathbb{S}_g} \vec{x}[j])^2 & \forall j \in \{1 \dots d\} \\ \sigma_{ij}^2 = 0 & \forall j \neq i \end{cases} \tag{7.17}$$

The above choices for weights and covariance matrix allow for an adaptive sampling scheme. First, by incorporating the scores into the weights ($w_i \propto f(\vec{x_i}^g)$), regions around high-score samples are given a higher priority to be explored. Second, the standard deviation in Equation (7.17) adaptively configures the sampling range in all dimensions: if members of $\mathbb{S}_g$ agree on dimension j, the corresponding σ_{jj}^2 will be small and the generated samples will be very similar in the jth dimension; conversely, if members of $\mathbb{S}_g$ disagree on dimension j, the corresponding σ_{jj}^2 will be large and the generated samples will be scattered in the jth dimension. This allows *AdaNS-Gaussian* to automatically tune the per-dimension exploration

and exploitation.

Cross **Samples.** The line connecting two good samples may represent the direction of gradient ascent for the objective function $f(\cdot)$. To explore such regions, we draw a portion of samples from the mid points of current good samples $\mathbb{S}_g$. To generate one sample, we pick a pair of good samples $\{\vec{x}_1, \vec{x}_2\} \in \mathbb{S}_g$, and draw a sample from the multivariate Gaussian $\mathcal{N}(\vec{\mu}_{1,2}, \Sigma_{1,2})$. The mean value is $\vec{\mu}_{1,2} = \frac{\vec{x}_1+\vec{x}_2}{2}$ and the diagonal covariance matrix is set as:

$$\begin{cases} \sigma_{jj}^2 = (\vec{x}_1[j] - \vec{x}_2[j])^2/4 & \forall j \in \{1 \ldots d\} \\ \sigma_{ij}^2 = 0 & \forall j \neq i \end{cases} \tag{7.18}$$

This approach searches for samples in the confined space between pairs of good samples. Multiple *Cross* samples are generated by repeating the above process. We explore several strategies for selecting $\{\vec{x}_1, \vec{x}_2\}$:

1. Select both $\vec{x}_1$ and $\vec{x}_2$ randomly from $\mathbb{S}_g$.

2. Sort all possible pairs $\{\vec{x}_1, \vec{x}_2\} \in \mathbb{S}_g$ based on their sum of scores $f(\vec{x}_1) + f(\vec{x}_2)$ and select pairs from the sorted list.

3. Sort individual members $\vec{x} \in \mathbb{S}_g$ based on their scores $f(\vec{x})$, sequentially select $\vec{x}_1$ from the sorted list and $\vec{x}_2$ as the sample with maximum Euclidean distance to $\vec{x}_1$.

4. Choose $\vec{x}_1$ as in case 3 but select $\vec{x}_2$ randomly.

Our experiments show that the fourth strategy renders the best performance on average. Therefore throughout the rest of the chapter, we use the latter method for pair selection.

7.5 Reconstruction

In this section, we provide an empirical analysis to verify the effectiveness of the designed sampling distributions $\mathcal{G}(\cdot)$ in finding near-optimal solutions. Towards this goal, we devise a

methodology that reconstructs the opaque objective function, based on previously evaluated samples. Specifically, at each iteration t, we create an estimate $\widetilde{f}(\cdot)$ for the hidden function $f(\cdot)$ using $f(\mathbb{S}_{[1:t]})$. Upon obtaining the new batch of (unseen) samples $\mathbb{S}_{t+1}$ and new measurements $f(\mathbb{S}_{t+1})$, we compute the (normalized) estimation errors as:

$$e(\vec{x}) = \frac{f(\vec{x}) - \widetilde{f}(\vec{x})}{f(\vec{x})}, \qquad \vec{x} \in \mathbb{S}_{t+1} \tag{7.19}$$

We then compute the mean mean absolute error e_{avg} over all samples. e_{avg} measures how much the obtained value of the objective function for new samples deviates from our estimation based on reconstruction. A high error implies that the sampling subroutine is exploring the space rather than using knowledge from $\mathbb{S}_{[1:t]}$ to find better samples. Conversely, a small e_{avg} shows that the adaptive sampler is exploiting previous good samples to generate $\mathbb{S}_{t+1}$. Figure 7.10 presents the task of pruning a VGG network trained on CIFAR-10 with *AdaNS-Gaussian*. Below, we summarize our observations:

1. The growth in the average score among good samples $f(\mathbb{S}_g)$ shows that *AdaNS* adaptive sampler iteratively sample values closer to the function maximizer.

2. The growth in the proximity parameter α_t together with property 1 suggests that the probability of finding the near-optimal solution is increasing as desired in Equation (7.7).

3. The drop in average reconstruction error e_{avg} together with property 1 demonstrates the ability of *AdaNS* in reconstructing the objective function around its potential maximizers. Specifically, the history of prior samples across *AdaNS* iterations are sufficient to estimate the value of the objective function at the future (unseen) samples that are closer to the function maximizers.

Establishing $\widetilde{\mathbf{f}}$. We consider a GMM prior for $\widetilde{f}(\cdot)$:

$$\widetilde{f}(\vec{x}) = \sum_{\vec{x}_i \in \mathbb{S}_{[1:t]}} w_i \, \mathcal{N}(\vec{x} - \vec{x}_i, \Sigma_i) \tag{7.20}$$

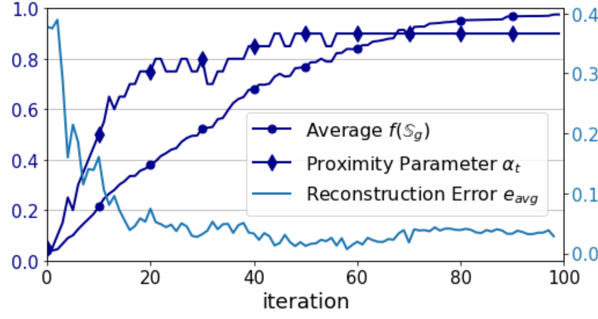

Figure 7.10. Per-iteration analysis of *AdaNS* sampling.

where w_i are scalar weights, $\vec{x}_i \in \mathbb{S}_{[1:t]}$ are the centers of the GMM model located at all previously evaluated samples, and $\Sigma_i \in \mathbb{R}^{d \times d}$ are diagonal covariance matrices. For the ith component of the GMM centered at $\vec{x}_i$, we find the closest sample $\vec{x}_j$ with minimum Euclidean distance to $\vec{x}_i$. We use the element-wise absolute difference, $\Delta\vec{x}_i = |\vec{x}_i - \vec{x}_j|$ to compute the covariance matrix:

$$\Sigma_i = \begin{cases} \sigma_{mm}^2 = \beta(\Delta\vec{x}_i[m])^2 & \forall m \in \{1 \dots d\} \\ \sigma_{mn}^2 = 0 & \forall m \neq n \end{cases} \tag{7.21}$$

The scalar $\beta = \frac{d}{8log(2)}$ normalizes the diagonal elements such that the multivariate Gaussian component is diminished by a factor of 2 in the mid-point of $\vec{x}_i, \vec{x}_j$. Given the Gaussian means $\vec{x}_i$ and covariance matrices Σ_i, the weights w_i can be determined by minimizing the error between real function evaluations $f(\mathbb{S}_{[1:t]})$ and the estimates $\widetilde{f}(\mathbb{S}_{[1:t]})$:

$$w_1 \dots w_K = \underset{w_1 \dots w_K}{\operatorname{argmin}} \sum_{x \in \mathbb{S}_{[1:t-1]}} |f(\vec{x}) - \widetilde{f}(\vec{x})|^2 \tag{7.22}$$

which is solved by Least-Square Optimization [15].

7.6 Experiments

We provide extensive evaluations on CIFAR-10 and ImageNet benchmarks and compare with prior works in RL, Bayesian Optimization, and several heuristics. The evaluated network

architectures include AlexNet, VGG, ResNet family, and MobileNets, implemented in PyTorch. The networks are trained from scratch following the parameter setup and training schedule adopted by the original papers [66, 73, 102, 170, 176]. For CIFAR-10, we use a VGG-variant as used in [94]. The models are compressed using *AdaNS* and fine-tuned for 60 and 20 epochs on CIFAR-10 and ImageNet, respectively. We randomly select $1,000$ images from the training data to use as validation set for score computation. We optimize hyperparameters for non-structured (P_n) and structured (P_s) pruning, SVD and Tucker decomposition (D), and combination of multiple methods ($D + P_s$). In our experiments, sample portions in *AdaNS-Gaussian* are assigned to be 45% *Local*, 45% *Cross*, and 10% *Uniform*. For *AdaNS-Genetic*, the crossover and mutation parameters are set to $p_{mutate} = 0.2$, $p_{tweak} = 0.05$, $p_{cross} = 0.8$, $p_{swap} = 0.2$ following [89, 212].

7.6.1 Effect of Sampling Strategy on Convergence

In this section, we investigate the impact of the sampling strategy on convergence and optimization end result. We first visualize the samples from our three subroutines for an example 2-layer network. Next, we move to pruning a real-world DNN benchmark and compare the convergence behavior of *AdaNS-Zoom*, *AdaNS-Genetic*, and *AdaNS-Gaussian*.

2-layer Example Network. The hyperparameter space for this example is $\vec{x} \in [0, 1]^2$ where each element of $\vec{x}$ represents the pruning rate for one layer. For this small example network, we executed a brute-force grid search to extract the heatmap of the pertinent objective function $f(\vec{x})$ as shown in Figure 7.11. Here, the blue and yellow colors denote minimum and maximum score function values, respectively, and red dots represent the samples. As seen, the *AdaNS-Zoom* subroutine (left) becomes concentrated on two of the local maxima regions (shown by black circles) but misses the global maximum located at $(0.4, 0.3)$. *AdaNS-Genetic* (middle) achieves more diversity than *AdaNS-Zoom*, and finds the neighborhood of the actual maximum; however, it does not achieve concentrated sampling. Finally, *AdaNS-Gaussian* (right) identifies the global maximum and concentrates the sampling around it.

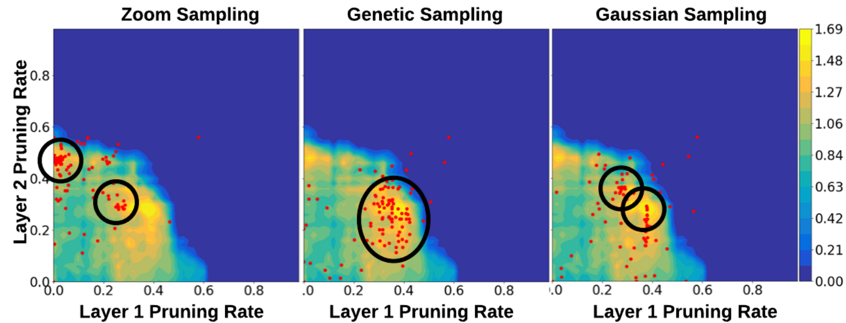

Figure 7.11. Comparison between *AdaNS* sampling subroutines with the same sample count, for pruning a 2-layer DNN. *AdaNS-Gaussian* achieves better exploration/exploitation tradeoff as it identifies the global maximum and concentrates the sampling around it.

VGG Benchmark. We use *AdaNS* for the task of structured pruning to compress a VGG network on CIFAR-10 dataset. We also provide the performance of a naïve method that uniformly samples the entire search-space. Figure 7.12-(left) presents the evolution of averageed good scores $f(\mathbb{S}_g)$ at each iteration, with $\mathbb{S}_g$ denoting the set of good samples. As seen, the naïve sampling fails to find the maxima in the high-dimensional optimization space at hand. This means unless carefully designed, a given naïve sampling strategy that arbitrarily changes the underlying hyperparameters fails to sample from correct regions of the space. However, our careful design of the sampling strategies can significantly change the performance by successfully arriving at the optima. *AdaNS-Gaussian* shows the largest growth in score suggesting that it is more successful in generating more competent samples based on prior observations. This is due to the Gaussian kernels in *AdaNS-Gaussian* which enable maximal exploration of sub-regions that potentially contain near-optimal solutions. *AdaNS-Zoom* and *AdaNS-Genetic* perform a more localized search and thus demonstrate slower convergence.

Figure 7.12-(right) shows how the proximity parameter α_t evolves over time. Initially, α_t is tuned to a small value to allow exploration of the entire search-space. As the search proceeds, the value of α_t is increased to direct the sampling towards the identified good samples. As seen, α_t rises quickly to $\alpha_{max} \approx 0.9$ for *AdaNS-Gaussian* and *AdaNS-Genetic* while it has a slower growth in *AdaNS-Zoom*. This suggests that *AdaNS-Gaussian* and *AdaNS-Genetic* quickly learn

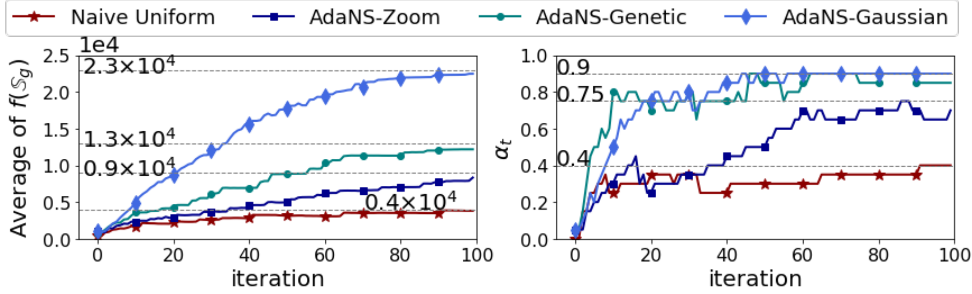

Figure 7.12. Convergence analysis of various sampling strategies across algorithm iterations. (left): mean score achieved by good samples. (right): Proximity parameter value.

to generate samples that are equally good or better than previously seen samples.

We further dissect the maximal score function in terms of accuracy and FLOPs in Table 7.1. The obtained accuracy and execution cost from each method confirms the importance of the sampling strategy on the final optimization result.

Table 7.1. Accuracy and FLOPs of the final compressed model using various sampling methods. Sample size is $b = 50$ and the sampling algorithm runs for 100 iterations on a VGG model trained on CIFAR-10. A_{thr} is set to 70% and the reported accuracy is before fine-tuning.

Sampling Strategy	Accuracy (%)	FLOPs (%)
Naïve Uniform	73.5	40.1
AdaNS-Zoom	70.4	37.4
AdaNS-Genetic	70.6	35.3
AdaNS-Gaussian	73.5	34.3

7.6.2 Quantitative Results on CIFAR-10

We apply *AdaNS-Gaussian* and *AdaNS-Genetic* to pre-trained CIFAR-10 architectures and compare our results with prior art in Table 7.2. We set the number of samples to 100 for ResNet-56 and ResNet-50, 200 for ResNet-110, and 50 for VGG. A_{thr} is set to 90% for ResNet-X and 65% for VGG. For all networks, we let *AdaNS* sampling run for 50 iterations.

Non-structured Pruning (P_n in Table 7.2). We perform non-structured pruning on ResNet-50 and report the ratio of non-zero model parameters. A_{thr}, is set to 93% and we do not perform any fine-tuning on the compressed model. As shown, *AdaNS-Gaussian* and *AdaNS-Genetic* achieve higher accuracy with $1.7\times$ and $1.3\times$ lower parameters compared to state-of-the-art

Table 7.2. Comparison with contemporary compression methods.

CIFAR-10

Model	Policy	Top1 (%)	Cost (%)
ResNet-50	Baseline	93.7	100
	AMC [69]	93.5	40.0
	Rethinking [124]	93.4	20.0
	Genetic (P_n)	**93.6**	**30.0**
	Gaussian (P_n)	**94.0**	**23.1**
ResNet-56	Baseline	93.6	100
	Rethinking [124]	93.1	72.4
	CP [70]	91.8	50.0
	AMC [69]	91.9	50.0
	SFP [68]	93.3	47.4
	PocketFlow [210]	92.8	40.0
	Genetic (P_s)	**93.2**	**44.0**
	Gaussian (P_s)	**93.1**	**40.6**
	Gaussian (D)	**93.5**	**59.1**
	Gaussian ($D + P_s$)	**93.2**	**36.9**
ResNet-110	Baseline	94.0	100
	Rethinking [124]	93.6	61.4
	Filter Pruning [110]	93.3	61.4
	AMC [69]	93.8	59.2
	Genetic (P_s)	**93.6**	**41.2**
	Gaussian (P_s)	**93.9**	**33.9**
	Gaussian (D)	**93.9**	**55.3**
	Gaussian ($D + P_s$)	**92.6**	**21.9**
VGG	Baseline	93.6	100
	Rethinking [124]	93.7	65.8
	ThiNet [126]	93.4	39.4
	NRE [94]	93.4	32.4
	Genetic (P_s)	**93.3**	**32.8**
	Gaussian (P_s)	**93.2**	**29.6**
	Gaussian (D)	**93.5**	**25.5**
	Gaussian ($D + P_s$)	**93.1**	**14.5**

ImageNet

Model	Policy	Top1 (%)	Top5 (%)	Cost (%)
AlexNet	Baseline	60.7	80.0	100
	CAC [26]	54.8	-	5.0
	Genetic (P_n)	**56.1**	**78.2**	**8.5**
	Gaussian (P_n)	**55.1**	**78.3**	**7.0**
ResNet-50	Baseline	75.1	93.0	100
	SFP [68]	62.1	84.6	58.2
	SSS [82]	71.8	90.8	57.0
	Rethinking [124]	75.0	-	50.0
	CP [70]	-	90.8	50.0
	GDP [115]	71.9	90.7	48.7
	ThiNet [126]	71.0	90.0	44.0
	Rethinking [124]	71.6	-	30.0
	Genetic (P_s)	**73.2**	**91.4**	**41.9**
	Gaussian (P_s)	**72.6**	**91.1**	**29.1**
	Gaussian (D)	**74.3**	**92.1**	**44.5**
	Gaussian ($D + P_s$)	**72.1**	**90.9**	**23.7**
VGG-16	Baseline	71.1	90.0	100
	GDP [115]	67.5	87.9	24.5
	RNP [114]	-	86.3	20.0
	SPP [204]	-	87.6	20.0
	AMC [69]	-	88.2	20.0
	Rethinking [124]	71.0	-	20.0
	Genetic (P_s)	**-**	**88.1**	**20.0**
	Gaussian (P_s)	**68.8**	**88.3**	**19.6**
	Gaussian (D)	**70.6**	**90.1**	**31.0**
	Gaussian ($D + P_s$)	**68.4**	**88.5**	**14.1**

Reinforcement Learning method, AMC [69]. Note that the lower FLOPs and comparable accuracy of [124] are due to training the model from scratch whereas *AdaNS* and [69] do not include any fine-tuning.

Structured Pruning (P_s in Table 7.2). Comparisons are based on the number of operations per inference, i.e., FLOPs, relative to the uncompressed baseline. With similar accuracy, *AdaNS-Gaussian* compression achieves $1.5\times$ lower FLOPs than prior art (on average).

Decomposition and Pruning ($D + P_s$ in Table 7.2). To unveil the full optimization potential of our adaptive sampling methodology, we allow *AdaNS* to learn and combine multiple compression

techniques, namely, structured pruning, SVD, and Tucker. The $D+P_s$ experiments are conducted by first decomposing the network and then applying pruning. As shown in Table 7.2, *AdaNS* pushes the limits of compression by $2.4\times$ on average with less than 1% drop in accuracy compared to state-of-the-art works. We also report FLOPs reduction by the combination of SVD and Tucker decomposition methods (shown by D in Table 7.2).

7.6.3 Quantitative Results on ImageNet

Table 7.2 summarizes *AdaNS* results on ImageNet. Number of samples is 20 for AlexNet, 100 for ResNet-50, and 50 for VGG-16. We run *AdaNS* for 50 iterations with A_{thr} of 10% for all models. The final accuracy is improved by fine-tuning.

Non-structured Pruning (P_n in Table 7.2). We perform non-structured pruning on AlexNet and report the ratio of non-zero model parameters. As seen, *AdaNS-Gaussian* achieves higher accuracy with 2% more parameters compared to a Bayesian Optimization approach, i.e., CAC [26].

Structured Pruning (P_s in Table 7.2). On ResNet-50, *AdaNS-Genetic* and *AdaNS-Gaussian* compress the models to $1.2\times$ and $1.6\times$ less FLOPs on average while achieving higher top-5 accuracy compared to prior works. On VGG-16, *AdaNS* outperforms all heuristic methods and gives competing results with [124] and [69]. Note that [124] does not propose a hyperparameter optimization algorithm and merely focuses on training already-compressed DNNs. Their approach is orthogonal to *AdaNS* and can be combined with it to further improve final accuracy.

Decomposition and Pruning ($D + P_s$ in Table 7.2). Using a combination of decomposition and structured pruning, *AdaNS* achieves $2.0\times$ lower FLOPs than related work (on average) with slightly higher accuracy on ResNet-50. On VGG-16, *AdaNS* pushes the state-of-the-art FLOPs reduction from $5.0\times$ to $7.1\times$ with higher accuracy.

7.6.4 Compressing Compact Networks

To further demonstrate the effectiveness of *AdaNS* optimization, we apply compression to MobileNet architectures trained on ImageNet dataset. These networks are specifically designed

for embedded applications with strict efficiency constraints. As such, MobileNets inherently have very low complexity/redundancy which renders their compression quite challenging. We apply pruning to MobileNetV1 and MobileNetV2 with a sample size of $b = 50$, and let the adaptive sampling run for 100 iterations.

We compare the compression rate and accuracy achieved by *AdaNS* with the FLOPs-accuracy Pareto curve of the original MobileNet architectures [73, 170]. We further compare *AdaNS* with the state-of-the-art AutoML approaches [69, 218] and a compression-aware training methodology, US-Nets [224, 225]. Table 7.3 encloses the results of applying structured pruning to MobileNetV1 and MobileNetV2. We benchmark several target FLOPs and compare them with prior work with similar computational complexities. On average, *AdaNS* achieves 1.2% better accuracy than the MobileNetV1 Pareto curve. Compared to US-Nets, *AdaNS* achieves an average of 1.0% higher accuracy. Under $\sim 50\%$ FLOPs, *AdaNS* achieves 1.3% higher accuracy than NetAdapt. Compared to AMC, *AdaNS* achieves lower FLOPs with comparable accuracy (-0.1%). On MobileNetV2, for a 30% FLOPs reduction, *AdaNS* achieves lower FLOPs and higher accuracy than US-Nets and higher accuracy with the same FLOPs compared to AMC and the MobileNetV2 Pareto curve.

Measured Speedup. We present measured speedups of *AdaNS* compressed MobileNets on an embedded CPU (ARM Cortex-A57) and GPU (NVIDIA Pascal) in Table 7.4. Measurements are averaged on 100 runs using a batch size of 32. *AdaNS* successfully models the hardware cost to achieve real speedups on par with theory.

7.6.5 Search Overhead and Scalability

The core computational load in *AdaNS* algorithm corresponds to the evaluation of a batch of b samples. For each sample $\vec{x}_i$ in the batch, the evaluation phase comprises transforming the sample to its corresponding compressed DNN $\hat{M}_{\vec{x}_i}$, measuring the inference accuracy on the validation data, and emulating the execution cost. Since samples in a batch are independent, the evaluation step can be well-parallelized on multiple GPU devices to achieve faster search

Table 7.3. Pruning of MobileNetV1&V2 on ImageNet.

	Policy	Top1 (%)	Top5 (%)	FLOPs
MobileNetV1	Baseline (1×)	70.6	89.5	569 M
	MobileNetV1 (0.75×) [73]	68.4	88.2	325 M
	US-Nets [224]	69.5	-	325 M
	AdaNS-Gaussian	**70.5**	89.3	**323 M**
	US-Nets [224]	68.8	-	287 M
	AMC [69]	70.5	89.1	285 M
	NetAdapt [218]	69.1	-	284 M
	AdaNS-Gaussian	**70.4**	89.1	**283 M**
	US-Nets [224]	66.8	-	217 M
	AdaNS-Gaussian	**67.9**	88.1	**210 M**
	MobileNetV1 (0.5×) [73]	63.7	-	149 M
	US-Nets [224]	63.5	-	136 M
	AdaNS-Gaussian	**64.1**	85.4	**136 M**
MobileNetV2	Baseline (1×)	71.6	90.3	313 M
	MobileNetV2 (0.75×) [170]	69.8	88.3	220 M
	US-Nets [224]	70.0	-	222 M
	AMC [69]	-	89.3	220 M
	AdaNS-Gaussian	**70.1**	**89.5**	**220 M**

Table 7.4. Speedup of *AdaNS* compressed MobileNets on embedded CPU and GPU after applying structured pruning on ImageNet benchmark.

Model	Theoretical Speedup	Real Speedup	
		Cortex-A57 (CPU)	Pascal (GPU)
MobileNetV1	1.7×	1.6×	1.3×
	2×	1.7×	1.4×
	2.7×	2.5×	1.7×
	4×	3.4×	1.9×
MobileNetV2	1.4×	1.4×	1.4×

convergence. Aside from evaluation, each *AdaNS* iteration includes updating the sampling strategy and generating a new batch of samples. These steps, however, are very lightweight and incur negligible runtime compared to the evaluation stage.

Table 7.5 summarizes the runtime of *AdaNS* algorithm for several benchmarks and datasets. Runtimes are measured on a machine with an Intel Xeon E5 CPU and four NVIDIA Titan Xp GPUs. The results show high scalability: runtime drops almost linearly with the number of GPUs. The state-of-the-art RL algorithm reports ~ 1 hour to compress CIFAR-10 architectures [69]. For their most complex benchmark, i.e., ResNet-56, *AdaNS* achieves a search

time of only ~ 12 minutes on a single GPU and ~ 3 minutes on four GPUs.

Table 7.5. Search runtime of *AdaNS-Gaussian* for pruning on various benchmarks. Here, b denotes the number of samples per iteration and N_{iters} is the number of search iterations.

Dataset	Arch.	b	N_{iters}	Search Time (minutes)			
				1 GPU	2 GPU	3 GPU	4 GPU
ImageNet	AlexNet	50	50	10	5	3	3
	VGG-16	50	50	112	57	38	28
	ResNet-50	100	50	145	73	49	36
	MobileNetV1	50	100	97	48	32	24
	MobileNetV2	50	100	116	57	38	25
CIFAR-10	VGG	50	50	3	2	1	1
	ResNet-50	100	50	35	19	13	11
	ResNet-56	100	50	12	7	5	3
	ResNet-110	200	50	55	30	22	16

7.6.6 Analysis and Discussion

In this section, we look into *AdaNS* generated samples and provide discussions. For brevity, we only focus on *AdaNS-Gaussian* subroutine that achieves superior results compared to *AdaNS-Zoom* and *AdaNS-Genetic*. We consider the VGG architecture trained on CIFAR-10 and compress it with structured pruning for $A_{thr} = 60\%$. The initial samples obtained from our directed initialization method are shown in Figure 7.13a, where each column corresponds to a hyperparameter vector and each row represents a certain DNN layer. After applying *AdaNS-Gaussian* for 50 iterations, the set of good samples in Figure 7.13b are learned; the columns (members of $\mathbb{S}_g$) strongly resemble one another and have similarly high scores upon convergence. *AdaNS* successfully learns expert-designed rules: first and last layers of the network (first and last rows in Figure 7.13b) are given high densities to maintain the inference accuracy.

AdaNS performs *whole-network* compression by encoding all DNN layers' hyperparameters in each sample. As such, our algorithm can learn which configuration of hyperparameters least affects model accuracy and most reduces the overall FLOPs. To show this capability, we present the per-layer FLOPs from one of the obtained good samples of Figure 7.13b in Figure 7.14. For each layer, the bars show the percentage of total FLOPs in the original model;

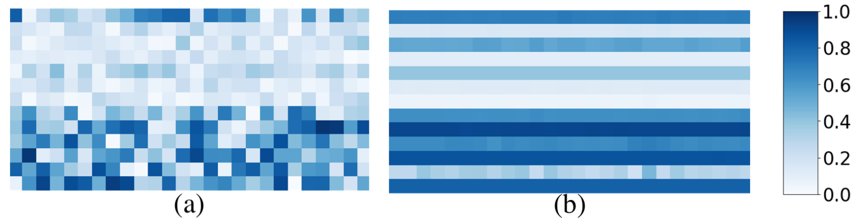

Figure 7.13. (a) Set of randomly initialized samples at the first iteration. (b) Set of good samples $\mathbb{S}_g$ upon convergence.

the curve shows the percentage of pruned FLOPs in the compressed network. As seen, the optimal pruning rates (red curve) show arbitrary patterns that are very hard to identify for human experts. *AdaNS* automatically extracts such patterns by an adaptive search.

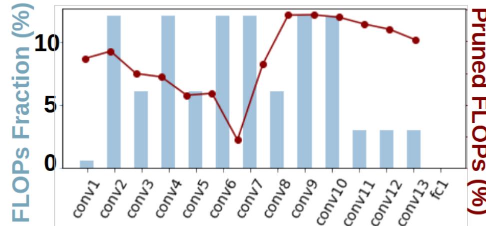

Figure 7.14. Original per-layer FLOPs versus *AdaNS* pruning pattern.

7.6.7 Ablation Study

We study the effect of various *AdaNS* parameters on algorithm convergence. For brevity, we only focus on structured pruning for VGG on CIFAR-10.

Effect of Initialization. Figure 7.15a shows the evolution of FLOPs ratio when running *AdaNS-Genetic* algorithm with two initialization policies: one with uniformly random samples and one with our proposed initialization scheme discussed in Section 7.3.3. As seen, naive initialization harms the convergence rate and final FLOPs.

Effect of Sample Count. Figure 7.15b presents the effect of number of evaluated samples per iteration of *AdaNS-Genetic* on convergence. A higher number of samples results in a smoother convergence and lower final FLOPs, due to the higher capacity for exploration/exploitation. This effect saturates for a large enough sample set. We further observed that the per-iteration sample

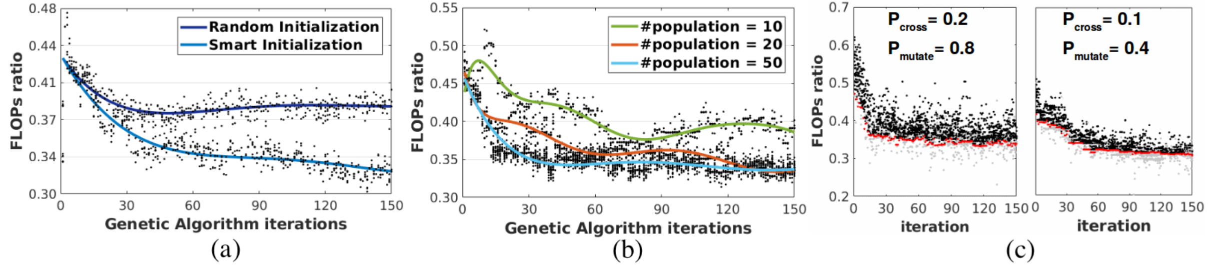

Figure 7.15. Ablation studies for VGG on CIFAR-10:(a) Effect of initialization method. (b) Effect of sample count. We show the trend lines as well as a fraction of samples (black dots) across *AdaNS* iterations. (c) Effect of mutation and cross-over probabilities for *AdaNS-Genetic*. Here, grey samples have lower accuracy than $A_{thr} = 60\%$ while black samples meet the accuracy threshold. Red dots correspond to the highest-quality sample per iteration.

count should be proportional to the length of each individual.

Effect of Accuracy Threshold. The accuracy threshold A_{thr} in Equation (7.4) determines model accuracy after compression. In our experiments, we observed a monotonic correlation between A_{thr} and final accuracy after fine-tuning. This property eliminates the need for fine-tuning each compressed DNN configuration in between algorithm iterations which significantly improves *AdaNS* search efficiency and timing overhead. Note that a smaller A_{thr} generally results in a lower hardware cost.

Mutation and Crossover Parameters. These parameters affect *AdaNS-Genetic* sampling convergence. We conduct two experiments, one with $(P_{mutate}, P_{cross}) = (0.8, 0.2)$ and the other with $(P_{mutate}, P_{cross}) = (0.4, 0.1)$ and compare the convergence in Figure 7.15c. Higher (P_{mutate}, P_{cross}) allows more exploring, leading to faster convergence while smaller probabilities result in a more stable evolution. As seen, both settings converge to similar final FLOPs.

Effect of Pair Selection for *Cross* Samples. Figure 7.16 compares the convergence behavior of the four proposed pair selection strategies for *Cross* samples (Section 7.4.3). As seen, strategy ❹ achieves the highest final score with lowest variations (shown with the error bars). Strategy ❷ has a fast but premature convergence due to lack of exploration. In addition, strategy ❷ has a high variation. This is due to the high dependency of strategy ❷ on the configuration of good samples, which differs across runs. Strategy ❶ and ❸ offer a smooth convergence with low variance but

their final score is lower than strategy ❹.

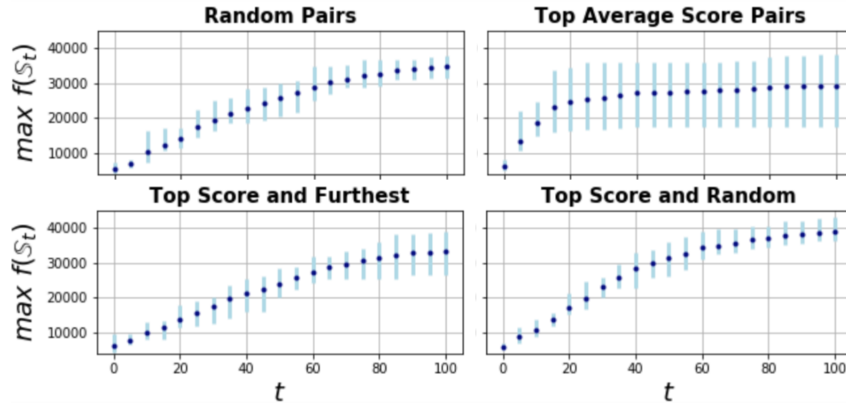

Figure 7.16. Convergence curves for various pair selection methods for *Cross* samples (Section 7.4.3). Graphs are generated over 10 runs.

7.7 Conclusion

This chapter proposes *AdaNS*, and adaptive sampling methodology that can automatically tune hyperparameters for DNN compression. We first formulate DNN compression as searching a vector space that spans possible hyperparameters. Next, we define a generic score function to quantify the "goodness" of each hyperparameter vector by combining inference accuracy and execution cost. This approach allows us to address DNN compression as optimizing the unknown score function by sampling from its input space. To find the maximizer of the score function, we develop an iterative sampling strategy, along with three adaptive sampling subroutines: *AdaNS-Zoom*, *AdaNS-Genetic*, and *AdaNS-Gaussian*. We evaluate these sampling strategies and show that *AdaNS-Gaussian* can achieve superior search convergence. We examine *AdaNS* on structured and non-structured pruning of deep neural networks and show that outperforms the majority of human-designed state-of-the-art network pruning algorithms.

7.8 Acknowledgements

Chapter 7 is, in parts, a reprint of the material as it appears in two publications: (1) M. Javaheripi, M. Samragh, T. Javidi, and F. Koushanfar, "AdaNS: Adaptive Non-uniform Sampling for Automated Design of Compact DNNs", in *IEEE Journal of Selected Topics in Signal Processing (JSTSP)*, 2020, and (2) M. Javaheripi, M. Samragh, T. Javidi, and F. Koushanfar, "GeneCAI: Genetic Evolution for Acquiring Compact AI", in *Genetic and Evolutionary Computation Conference*, 2020. The dissertation author was the primary investigator and author of both published papers.

Bibliography

[1] Bair/bvlc alexnet model. https://github.com/BVLC/caffe/tree/master/models/bvlc_alexnet.

[2] M. S. Abdelfattah, A. Mehrotra, Ł. Dudziak, and N. D. Lane. Zero-cost proxies for lightweight nas. In *International Conference on Learning Representations*, 2020.

[3] M. Aharon and M. Elad. Sparse and redundant modeling of image content using an image-signature-dictionary. *SIAM Journal on Imaging Sciences*, 1(3):228–247, 2008.

[4] M. Aharon, M. Elad, and A. Bruckstein. K-svd: An algorithm for designing overcomplete dictionaries for sparse representation. *IEEE Transactions on signal processing*, 54(11):4311–4322, 2006.

[5] B. H. Ahn, J. Lee, J. M. Lin, H.-P. Cheng, J. Hou, and H. Esmaeilzadeh. Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices. *Proceedings of Machine Learning and Systems*, 2:44–57, 2020.

[6] M. AI. Codebase for open pre-trained transformers. https://github.com/facebookresearch/metaseq.

[7] A. Anjos and S. Marcel. Counter-measures to photo attacks in face recognition: a public database and a baseline. In *2011 international joint conference on Biometrics (IJCB)*, pages 1–7. IEEE, 2011.

[8] T. Bäck, D. B. Fogel, and Z. Michalewicz. *Handbook of evolutionary computation*. CRC Press, 1997.

[9] A. Baevski and M. Auli. Adaptive input representations for neural language modeling. In *International Conference on Learning Representations*, 2018.

[10] M. Barahona and L. M. Pecora. Synchronization in small-world systems. *Physical review letters*, 89(5):054101, 2002.

[11] M. Barreno, B. Nelson, R. Sears, A. D. Joseph, and J. D. Tygar. Can machine learning be secure? In *ACM Symposium on Information, computer and communications security*, pages 16–25, 2006.

[12] K. Bhardwaj, G. Li, and R. Marculescu. How does topology influence gradient propagation and model performance of deep networks with densenet-type skip connections? In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13498–13507, 2021.

[13] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2013.

[14] B. Biggio, G. Fumera, and F. Roli. Pattern recognition systems under attack: Design issues and research challenges. *International Journal of Pattern Recognition and Artificial Intelligence*, 28(07):1460002, 2014.

[15] R. T. Birge. The calculation of errors by the method of least squares. *Physical Review*, 40(2):207, 1932.

[16] Y. Bisk, R. Zellers, J. Gao, and Y. Choi. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 7432–7439, 2020.

[17] C. Bouveyron, S. Girard, and C. Schmid. High-dimensional discriminant analysis. *Communications in Statistics—Theory and Methods*, 36, 2007.

[18] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han. Once-for-all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*, 2019.

[19] N. Carlini and D. Wagner. Defensive distillation is not robust to adversarial examples. *arXiv:1607.04311*, 2016.

[20] N. Carlini and D. Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *ACM Workshop on AISec*, pages 3–14, 2017.

[21] N. Carlini and D. Wagner. Magnet and "efficient defenses against adversarial attacks" are not robust to adversarial examples. *arXiv:1711.08478*, 2017.

[22] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 39–57. IEEE, 2017.

[23] M. Charikar, J. Steinhardt, and G. Valiant. Learning from untrusted data. In *49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 47–60, 2017.

[24] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, P. Koehn, and T. Robinson. One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005*, 2013.

[25] B. Chen, W. Carvalho, N. Baracaldo, H. Ludwig, B. Edwards, T. Lee, I. Molloy, and B. Srivastava. Detecting backdoor attacks on deep neural networks by activation clustering. *arXiv preprint arXiv:1811.03728*, 2018.

[26] C. Chen, F. Tung, N. Vedula, and G. Mori. Constraint-aware deep neural network compression. In *European Conference on Computer Vision (ECCV)*, pages 400–415, 2018.

[27] H. Chen, C. Fu, J. Zhao, and F. Koushanfar. Deepinspect: A black-box trojan detection and mitigation framework for deep neural networks. In *28th International Joint Conference on Artificial Intelligence.*, pages 4658–4664, 2019.

[28] E. K. Chong and S. H. Zak. *An introduction to optimization*, volume 76. John Wiley & Sons, 2013.

[29] E. Chou, F. Tramèr, G. Pellegrino, and D. Boneh. Sentinet: Detecting physical attacks against deep learning systems. *arXiv preprint arXiv:1812.00292*, 2018.

[30] P. Clark, I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, and O. Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

[31] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu. Large-scale malware classification using random projections and neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3422–3426. IEEE, 2013.

[32] Z. Dai, Z. Yang, Y. Yang, J. G. Carbonell, Q. Le, and R. Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988, 2019.

[33] G. M. Davis, S. G. Mallat, and Z. Zhang. Adaptive time-frequency decompositions. *Optical engineering*, 33(7):2183–2192, 1994.

[34] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. De Freitas. Predicting parameters in deep learning. *Advances in neural information processing systems*, 26, 2013.

[35] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT (1)*, 2019.

[36] B. G. Doan, E. Abbasnejad, and D. C. Ranasinghe. Februus: Input purification defense against trojan attacks on deep neural network systems. In *Annual Computer Security Applications Conference*, pages 897–912, 2020.

[37] D. L. Donoho and M. Elad. Optimally sparse representation in general (nonorthogonal) dictionaries via l1 minimization. *Proceedings of the National Academy of Sciences*, 100(5):2197–2202, 2003.

[38] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[39] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani. Least angle regression. *The Annals of statistics*, 32(2):407–499, 2004.

[40] T. Elsken, J. H. Metzen, and F. Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. In *International Conference on Learning Representations*, 2019.

[41] T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.

[42] A. Elthakeb, P. Pilligundla, F. Mireshghallah, A. Yazdanbakhsh, S. Gao, and H. Esmaeilzadeh. Releq: An automatic reinforcement learning approach for deep quantization of neural networks. In *NeurIPS ML for Systems workshop*, 2018.

[43] K. Engan, S. O. Aase, and J. H. Husoy. Method of optimal directions for frame design. In *1999 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings. ICASSP99 (Cat. No. 99CH36258)*, volume 5, pages 2443–2446. IEEE, 1999.

[44] O. Erkaymaz, M. Ozer, and M. Perc. Performance of small-world feedforward neural networks for the diagnosis of diabetes. *Applied Mathematics and Computation*, 311:22–28, 2017.

[45] H. Face. Openai gpt2 by hugging face. https://huggingface.co/docs/transformers/model_doc/gpt2.

[46] G. Fields, M. Samragh, M. Javaheripi, F. Koushanfar, and T. Javidi. Trojan signatures in dnn weights. In *IEEE/CVF International Conference on Computer Vision*, pages 12–20, 2021.

[47] P. Fogla and W. Lee. Evading network anomaly detection systems: formal reasoning and practical techniques. In *13th ACM conference on Computer and communications security*, 2006.

[48] M. Fredrikson, S. Jha, and T. Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1322–1333, 2015.

[49] J. Gao, H. Xu, H. Shi, X. Ren, L. Philip, X. Liang, X. Jiang, and Z. Li. Autobert-zero: Evolving bert backbone from scratch. In *AAAI Conference on Artificial Intelligence*, volume 36, pages 10663–10671, 2022.

[50] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, and C. Leahy. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

[51] L. Gao, J. Tow, S. Biderman, S. Black, A. DiPofi, C. Foster, L. Golding, J. Hsu, K. McDonell, N. Muennighoff, J. Phang, L. Reynolds, E. Tang, A. Thite, B. Wang, K. Wang, and A. Zou. A framework for few-shot language model evaluation. https://github.com/EleutherAI/lm-evaluation-harness, Sept. 2021.

[52] Y. Gao, C. Xu, D. Wang, S. Chen, D. C. Ranasinghe, and S. Nepal. Strip: A defence against trojan attacks on deep neural networks. In *35th Annual Computer Security Applications Conference*, pages 113–125, 2019.

[53] R. Geirhos, P. Rubisch, C. Michaelis, M. Bethge, F. A. Wichmann, and W. Brendel. Imagenet-trained cnns are biased towards texture; increasing shape bias improves accuracy and robustness. *arXiv preprint arXiv:1811.12231*, 2018.

[54] A. Gholami, N. Torkzaban, and J. S. Baras. On the importance of trust in next-generation networked cps systems: An ai perspective. *arXiv preprint arXiv:2104.07853*, 2021.

[55] G. H. Golub and C. F. Van Loan. *Matrix computations*, volume 3. John Hopkins University Press, 2012.

[56] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[57] S. Gu and L. Rigazio. Towards deep neural network architectures robust to adversarial examples. *arXiv:1412.5068*, 2014.

[58] T. Gu, B. Dolan-Gavitt, and S. Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733*, 2017.

[59] J. Guerrero-Viu, S. Hauns, S. Izquierdo, G. Miotto, S. Schrodi, A. Biedenkapp, T. Elsken, D. Deng, M. Lindauer, and F. Hutter. Bag of baselines for multi-objective joint neural architecture search and hyperparameter optimization. *arXiv preprint arXiv:2105.01015*, 2021.

[60] W. Guo, L. Wang, X. Xing, M. Du, and D. Song. Tabor: A highly accurate approach to inspecting and restoring trojan backdoors in ai systems. *arXiv preprint arXiv:1908.01763*, 2019.

[61] K. Hamm. Nonuniform sampling and recovery of bandlimited functions in higher dimensions. *Journal of Mathematical Analysis and Applications*, 450(2):1459–1478, 2017.

[62] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015.

[63] T. Hangelbroek and A. Ron. Nonlinear approximation using gaussian kernels. *Journal of Functional Analysis*, 259(1):203–219, 2010.

[64] J. Haupt, R. M. Castro, and R. Nowak. Distilled sensing: Adaptive sampling for sparse detection and estimation. *IEEE Transactions on Information Theory*, 57:6222–6235, 2011.

[65] E. Hazan, A. Klivans, and Y. Yuan. Hyperparameter optimization: A spectral approach. *arXiv preprint arXiv:1706.00764*, 2017.

[66] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[67] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016.

[68] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang. Soft filter pruning for accelerating deep convolutional neural networks. In *27th International Joint Conference on Artificial Intelligence*, pages 2234–2240, 2018.

[69] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han. Amc: Automl for model compression and acceleration on mobile devices. In *European conference on computer vision (ECCV)*, pages 784–800, 2018.

[70] Y. He, X. Zhang, and J. Sun. Channel pruning for accelerating very deep neural networks. In *IEEE international conference on computer vision*, pages 1389–1397, 2017.

[71] Z. He, A. S. Rakin, J. Li, C. Chakrabarti, and D. Fan. Defending and harnessing the bit-flip based adversarial weight attack. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14095–14103, 2020.

[72] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitraș. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 497–514, 2019.

[73] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[74] H. Hu, J. Langford, R. Caruana, S. Mukherjee, E. J. Horvitz, and D. Dey. Efficient forward architecture search. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[75] Y. Hu, S. Sun, J. Li, X. Wang, and Q. Gu. A novel channel pruning method for deep neural network compression. *arXiv preprint arXiv:1805.11394*, 2018.

[76] G. Huang, S. Liu, L. Van der Maaten, and K. Q. Weinberger. Condensenet: An efficient densenet using learned group convolutions. In *IEEE conference on computer vision and pattern recognition*, pages 2752–2761, 2018.

[77] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[78] G. Huang, M. Mattar, H. Lee, and E. Learned-Miller. Learning to align from scratch. *Advances in neural information processing systems*, 25, 2012.

[79] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger. Deep networks with stochastic depth. In *European Conference on Computer Vision (ECCV)*, pages 646–661. Springer, 2016.

[80] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. D. Tygar. Adversarial machine learning. In *4th ACM workshop on Security and artificial intelligence*, pages 43–58, 2011.

[81] R. Huang, B. Xu, D. Schuurmans, and C. Szepesvári. Learning with a strong adversary. *arXiv:1511.03034*, 2015.

[82] Z. Huang and N. Wang. Data-driven sparse structure selection for deep neural networks. In *European Conference on Computer Vision (ECCV)*, pages 304–320, 2018.

[83] M. D. Humphries and K. Gurney. Network 'small-world-ness': a quantitative method for determining canonical network equivalence. *PloS one*, 3(4):e0002051, 2008.

[84] S. Hussain, M. Javaheripi, P. Neekhara, R. Kastner, and F. Koushanfar. Fastwave: Accelerating autoregressive convolutional neural networks on fpga. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[85] A. Ilyas, S. Santurkar, D. Tsipras, L. Engstrom, B. Tran, and A. Madry. Adversarial examples are not bugs, they are features. *Advances in neural information processing systems*, 32, 2019.

[86] M. Javaheripi and F. Koushanfar. Hashtag: Hash signatures for online detection of fault-injection attacks on deep neural networks. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2021.

[87] M. Javaheripi, B. D. Rouhani, and F. Koushanfar. Swann: Small-world architecture for fast convergence of neural networks. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 11(4):575–585, 2021.

[88] M. Javaheripi, M. Samragh, G. Fields, T. Javidi, and F. Koushanfar. Cleann: Accelerated trojan shield for embedded neural networks. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020.

[89] M. Javaheripi, M. Samragh, T. Javidi, and F. Koushanfar. Genecai: gene tic evolution for acquiring c ompact ai. In *Genetic and Evolutionary Computation Conference*, pages 350–358, 2020.

[90] M. Javaheripi, M. Samragh, and F. Koushanfar. Peeking into the black box: A tutorial on automated design optimization and parameter search. *IEEE Solid-State Circuits Magazine*, 11(4):23–28, 2019.

[91] M. Javaheripi, M. Samragh, and F. Koushanfar. Autorank: Automated rank selection for effective neural network customization. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 11(4):611–619, 2021.

[92] M. Javaheripi, M. Samragh, B. D. Rouhani, T. Javidi, and F. Koushanfar. Curtail: Characterizing and thwarting adversarial deep learning. *IEEE Transactions on Dependable and Secure Computing*, 18(2):736–752, 2020.

[93] D. Jia, K. Han, Y. Wang, Y. Tang, J. Guo, C. Zhang, and D. Tao. Efficient vision transformers via fine-grained manifold distillation. *arXiv preprint arXiv:2107.01378*, 2021.

[94] C. Jiang, G. Li, C. Qian, and K. Tang. Efficient dnn neuron pruning by minimizing layer-wise nonlinear reconstruction error. In *IJCAI*, volume 2018, pages 2–2, 2018.

[95] J. Jin, A. Dundar, and E. Culurciello. Robust convolutional neural networks under adversarial noise. *arXiv:1511.06306*, 2015.

[96] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[97] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[98] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

[99] G. Krishnan, Y. Ma, and Y. Cao. Small-world-based structural pruning for efficient fpga inference of deep neural networks. In *2020 IEEE 15th International Conference on Solid-State & Integrated Circuit Technology (ICSICT)*, pages 1–5. IEEE, 2020.

[100] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. 2009.

[101] A. Krizhevsky, V. Nair, and G. Hinton. Cifar-10 (canadian institute for advanced research). 2009.

[102] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. pages 1097–1105, 2012.

[103] M. Kuperman and G. Abramson. Small world effect in an epidemiological model. *Physical review letters*, 86(13):2909, 2001.

[104] A. Kurakin, I. Goodfellow, and S. Bengio. Adversarial examples in the physical world. *arXiv:1607.02533*, 2016.

[105] V. Latora and M. Marchiori. Efficient behavior of small-world networks. *Physical review letters*, 87(19):198701, 2001.

[106] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *IEEE Proceedings*, 86(11):2278–2324, 1998.

[107] Y. LeCun, C. Cortes, and C. J. Burges. The MNIST database of handwritten digits, 1998.

[108] K. Lee, K. Lee, H. Lee, and J. Shin. A simple unified framework for detecting out-of-distribution samples and adversarial attacks. In *Advances in Neural Information Processing Systems*, pages 7167–7177, 2018.

[109] N. Lee, T. Ajanthan, and P. Torr. SNIP: SINGLE-SHOT NETWORK PRUNING BASED ON CONNECTION SENSITIVITY. In *International Conference on Learning Representations*, 2019.

[110] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[111] J. Li, A. S. Rakin, Z. He, D. Fan, and C. Chakrabarti. Radar: Run-time adversarial weight attack detection and accuracy recovery. *arXiv preprint arXiv:2101.08254*, 2021.

[112] J. Li, A. S. Rakin, Y. Xiong, L. Chang, Z. He, D. Fan, and C. Chakrabarti. Defending bit-flip attack through dnn weight reconstruction. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[113] Y. Li, M. Li, B. Luo, Y. Tian, and Q. Xu. Deepdyve: Dynamic verification for deep neural networks. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 101–112, 2020.

[114] J. Lin, Y. Rao, J. Lu, and J. Zhou. Runtime neural pruning. In *Advances in Neural Information Processing Systems*, pages 2181–2191, 2017.

[115] S. Lin, R. Ji, Y. Li, Y. Wu, F. Huang, and B. Zhang. Accelerating convolutional networks via global & dynamic filter pruning. In *IJCAI*, pages 2425–2432, 2018.

[116] C. Liu, B. Li, Y. Vorobeychik, and A. Oprea. Robust linear regression against training data poisoning. In *10th ACM Workshop on Artificial Intelligence and Security*, pages 91–102, 2017.

[117] H. Liu, K. Simonyan, and Y. Yang. DARTS: Differentiable architecture search. In *International Conference on Learning Representations*, 2019.

[118] K. Liu, B. Dolan-Gavitt, and S. Garg. Fine-pruning: Defending against backdooring attacks on deep neural networks. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 273–294. Springer, 2018.

[119] Q. Liu, W. Wen, and Y. Wang. Concurrent weight encoding-based detection for bit-flip attack on neural network accelerators. In *39th International Conference on Computer-Aided Design*, pages 1–8, 2020.

[120] Y. Liu, W.-C. Lee, G. Tao, S. Ma, Y. Aafer, and X. Zhang. Abs: Scanning neural networks for back-doors by artificial brain stimulation. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1265–1282, 2019.

[121] Y. Liu, S. Ma, Y. Aafer, W.-C. Lee, J. Zhai, W. Wang, and X. Zhang. Trojaning attack on neural networks. In *25th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2018.

[122] Y. Liu, L. Wei, B. Luo, and Q. Xu. Fault injection attack on deep neural network. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 131–138. IEEE, 2017.

[123] Y. Liu, Y. Xie, and A. Srivastava. Neural trojans. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 45–48. IEEE, 2017.

[124] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell. Rethinking the value of network pruning. In *International Conference on Learning Representations*, 2018.

[125] E. Lukacs. A characterization of the normal distribution. *The Annals of Mathematical Statistics*, 13(1):91–93, 1942.

[126] J.-H. Luo, J. Wu, and W. Lin. Thinet: A filter level pruning method for deep neural network compression. In *IEEE international conference on computer vision*, pages 5058–5066, 2017.

[127] S. Ma and Y. Liu. Nic: Detecting adversarial samples with neural network invariant checking. In *26th Network and Distributed System Security Symposium (NDSS 2019)*, 2019.

[128] X. Ma, B. Li, Y. Wang, S. M. Erfani, S. Wijewickrema, G. Schoenebeck, D. Song, M. E. Houle, and J. Bailey. Characterizing adversarial subspaces using local intrinsic dimensionality. *arXiv preprint arXiv:1801.02613*, 2018.

[129] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv:1706.06083*, 2017.

[130] P. Mattson, C. Cheng, C. Coleman, G. Diamos, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, D. Brooks, D. Chen, D. Dutta, U. Gupta, K. Hazelwood, A. Hock, X. Huang, A. Ike, B. Jia, D. Kang, D. Kanter, N. Kumar, J. Liao, G. Ma, D. Narayanan, T. Oguntebi, G. Pekhimenko, L. Pentecost, V. J. Reddi, T. Robie, T. S. John, T. Tabaru, C.-J. Wu, L. Xu, M. Yamazaki, C. Young, and M. Zaharia. Mlperf training benchmark. *arXiv preprint arXiv:1910.01500*, 2019.

[131] P. McDaniel, N. Papernot, and Z. B. Celik. Machine learning in adversarial settings. *IEEE Security & Privacy*, 14(3):68–72, 2016.

[132] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.

[133] J. Mellor, J. Turner, A. Storkey, and E. J. Crowley. Neural architecture search without training. https://openreview.net/forum?id=g4E6SAAvACo, 2021.

[134] J. Mellor, J. Turner, A. Storkey, and E. J. Crowley. Neural architecture search without training. In *International Conference on Machine Learning*, pages 7588–7598. PMLR, 2021.

[135] D. Meng and H. Chen. Magnet: a two-pronged defense against adversarial examples. *arXiv:1705.09064*, 2017.

[136] S. Merity, C. Xiong, J. Bradbury, and R. Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[137] T. Mihaylov, P. Clark, T. Khot, and A. Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering. *arXiv preprint arXiv:1809.02789*, 2018.

[138] A. Mirhoseini, E. L. Dyer, E. M. Songhori, R. Baraniuk, and F. Koushanfar. Rankmap: A framework for distributed learning from dense data sets. *IEEE transactions on neural networks and learning systems*, 29(7):2717–2730, 2017.

[139] T. Miyato, S.-i. Maeda, M. Koyama, K. Nakae, and S. Ishii. Distributional smoothing with virtual adversarial training. *arXiv:1507.00677*, 2015.

[140] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz. Importance estimation for neural network pruning. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11264–11272, 2019.

[141] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning convolutional neural networks for resource efficient transfer learning. *arXiv preprint:1611.06440*, 3, 2016.

[142] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2574–2582, 2016.

[143] Y. Nesterov. Gradient methods for minimizing composite functions. *Mathematical programming*, 140(1):125–161, 2013.

[144] X. Ning, C. Tang, W. Li, Z. Zhou, S. Liang, H. Yang, and Y. Wang. Evaluating efficient performance estimators of neural architectures. *Advances in Neural Information Processing Systems*, 34, 2021.

[145] NVIDIA. Transformer-xl for pytorch. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/LanguageModeling/Transformer-XL.

[146] R. Olfati-Saber. Ultrafast consensus in small-world networks. In *American Control Conference*, pages 2371–2378. IEEE, 2005.

[147] N. Papernot, N. Carlini, I. Goodfellow, R. Feinman, F. Faghri, A. Matyasko, K. Hambardzumyan, Y.-L. Juang, K. Alexey, R. Sheatsley, A. Garg, and L. Yen-Chen. CleverHans v2.0.0: an adversarial machine learning library. *arXiv:1610.00768*, 2017.

[148] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *IEEE European symposium on security and privacy (EuroS&P)*, pages 372–387. IEEE, 2016.

[149] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *2016 IEEE symposium on security and privacy (SP)*, pages 582–597. IEEE, 2016.

[150] O. M. Parkhi, A. Vedaldi, and A. Zisserman. Deep face recognition. In *British Machine Vision Conference (BMVC)*, pages 41.1–41.12. BMVA Press, 2015.

[151] P. K. Pearson. Fast hashing of variable-length text strings. *Communications of the ACM*, 33(6):677–680, 1990.

[152] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean. Efficient neural architecture search via parameters sharing. In *International Conference on Machine Learning*, pages 4095–4104. PMLR, 2018.

[153] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[154] A. S. Rakin, Z. He, and D. Fan. Bit-flip attack: Crushing neural network with progressive bit search. In *IEEE/CVF International Conference on Computer Vision*, pages 1211–1220, 2019.

[155] A. S. Rakin, Z. He, and D. Fan. Tbt: Targeted neural network attack with bit trojan. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13198–13207, 2020.

[156] A. S. Rakin, Z. He, J. Li, F. Yao, C. Chakrabarti, and D. Fan. T-bfa: Targeted bit-flip adversarial weight attack. *arXiv preprint arXiv:2007.12336*, 2020.

[157] A. S. Rakin, L. Yang, J. Li, F. Yao, C. Chakrabarti, Y. Cao, J.-s. Seo, and D. Fan. Rabnn: Constructing robust & accurate binary neural network to simultaneously defend adversarial bit-flip attack and improve accuracy. *arXiv preprint arXiv:2103.13813*, 2021.

[158] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search. In *AAAI conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[159] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin. Large-scale evolution of image classifiers. In *International Conference on Machine Learning*, pages 2902–2911. PMLR, 2017.

[160] K. Roth, Y. Kilcher, and T. Hofmann. The odds are odd: A statistical test for detecting adversarial examples. In *International Conference on Machine Learning*, pages 5498–5507, 2019.

[161] B. D. Rouhani, A. Mirhoseini, and F. Koushanfar. Deep3: Leveraging three levels of parallelism for efficient deep learning. In *Design Automation Conference*, 2017.

[162] B. D. Rouhani, M. Samragh, M. Javaheripi, T. Javidi, and F. Koushanfar. Deepfense: Online accelerated defense against adversarial deep learning. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[163] T. B. rown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[164] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vision*, 115(3):211–252, Dec. 2015.

[165] K. Sakaguchi, R. L. Bras, C. Bhagavatula, and Y. Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

[166] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[167] D. Salomon. *Data compression: the complete reference*. Springer Science & Business Media, 2004.

[168] M. Samragh, M. Javaheripi, and F. Koushanfar. Codex: Bit-flexible encoding for streaming-based fpga acceleration of dnns. *arXiv preprint arXiv:1901.05582*, 2019.

[169] M. Samragh, M. Javaheripi, and F. Koushanfar. Encodeep: Realizing bit-flexible encoding for deep neural networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 19(6):1–29, 2020.

[170] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[171] U. Shaham, Y. Yamada, and S. Negahban. Understanding adversarial training: Increasing local stability of neural nets through robust optimization. *arXiv:1511.05432*, 2015.

[172] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. From high-level deep neural models to fpgas. In *IEEE/ACM International Symposium on Microarchitecture*, page 17. IEEE Press, 2016.

[173] S. Shekhar and T. Javidi. Gaussian process bandits with adaptive discretization. *Electronic Journal of Statistics*, 12(2):3829–3874, 2018.

[174] S. Shen, G. Jin, K. Gao, and Y. Zhang. APE-GAN: Adversarial perturbation elimination with GAN. *ICLR Submission, available on OpenReview*, 2017.

[175] D. Simard, L. Nadeau, and H. Kröger. Fastest learning in small-world neural networks. *Physics Letters A*, 336(1):8–15, 2005.

[176] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[177] L. N. Smith. Cyclical learning rates for training neural networks. In *2017 IEEE winter conference on applications of computer vision (WACV)*, pages 464–472. IEEE, 2017.

[178] D. So, Q. Le, and C. Liang. The evolved transformer. In *International conference on machine learning*, pages 5877–5886. PMLR, 2019.

[179] D. So, W. Mańke, H. Liu, Z. Dai, N. Shazeer, and Q. V. Le. Searching for efficient transformers for language modeling. *Advances in Neural Information Processing Systems*, 34:6010–6022, 2021.

[180] N. Srinivas, A. Krause, S. M. Kakade, and M. W. Seeger. Information-theoretic regret bounds for gaussian process optimization in the bandit setting. *IEEE Transactions on Information Theory*, 58(5):3250–3265, 2012.

[181] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural networks*, 32:323–332, 2012.

[182] B. Stellato, B. P. Van Parys, and P. J. Goulart. Multivariate chebyshev inequality with estimated mean and variance. *The American Statistician*, 71(2):123–127, 2017.

[183] S. H. Strogatz. Exploring complex networks. *nature*, 410(6825):268–276, 2001.

[184] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv:1312.6199*, 2013.

[185] A. Tahbaz-Salehi and A. Jadbabaie. Small world phenomenon, rapidly mixing markov chains, and average consensus algorithms. In *2007 46th IEEE Conference on Decision and Control*, pages 276–281. IEEE, 2007.

[186] M. Tan and Q. Le. Efficientnetv2: Smaller models and faster training. In *International Conference on Machine Learning*, pages 10096–10106. PMLR, 2021.

[187] H. Tanaka, D. Kunin, D. L. Yamins, and S. Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow. *Advances in Neural Information Processing Systems*, 33, 2020.

[188] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 213–226, 2018.

[189] L. Theis, I. Korshunova, A. Tejani, and F. Huszár. Faster gaze prediction with dense networks and fisher pruning. *arXiv preprint arXiv:1801.05787*, 2018.

[190] N. Torkzaban and J. S. Baras. Trust-aware service function chain embedding: A path-based approach. In *2020 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 31–36, 2020.

[191] N. Torkzaban, C. Papagianni, and J. S. Baras. Trust-aware service chain embedding. In *2019 Sixth International Conference on Software Defined Systems (SDS)*, pages 242–247, 2019.

[192] B. Tran, J. Li, and A. Madry. Spectral signatures in backdoor attacks. In *Advances in Neural Information Processing Systems*, pages 8000–8010, 2018.

[193] J. A. Tropp and A. C. Gilbert. Signal recovery from random measurements via orthogonal matching pursuit. *IEEE Transactions on information theory*, 53:4655–4666, 2007.

[194] H. Tsai, J. Ooi, C.-S. Ferng, H. W. Chung, and J. Riesa. Finding fast transformers: One-shot neural architecture search by component composition. *arXiv preprint arXiv:2008.06808*, 2020.

[195] D. Vainsencher, S. Mannor, and A. M. Bruckstein. The sample complexity of dictionary learning. *Journal of Machine Learning Research*, 12(Nov):3259–3281, 2011.

[196] V. Van Der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *ACM SIGSAC conference on computer and communications security*, pages 1675–1689, 2016.

[197] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[198] A. Wang, Y. Pruksachatkun, N. Nangia, A. Singh, J. Michael, F. Hill, O. Levy, and S. Bowman. Superglue: A stickier benchmark for general-purpose language understanding systems. *Advances in neural information processing systems*, 32, 2019.

[199] B. Wang, J. Gao, and Y. Qi. A theoretical framework for robustness of (deep) classifiers under adversarial noise. *arXiv:1612.00334*, 2016.

[200] B. Wang, Y. Yao, S. Shan, H. Li, B. Viswanath, H. Zheng, and B. Y. Zhao. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 707–723. IEEE, 2019.

[201] C. Wang, G. Zhang, and R. Grosse. Picking winning tickets before training by preserving gradient flow. In *International Conference on Learning Representations*, 2020.

[202] H. Wang, S. Ma, L. Dong, S. Huang, D. Zhang, and F. Wei. Deepnet: Scaling transformers to 1,000 layers. *arXiv preprint arXiv:2203.00555*, 2022.

[203] H. Wang, Z. Wu, Z. Liu, H. Cai, L. Zhu, C. Gan, and S. Han. Hat: Hardware-aware transformers for efficient natural language processing. In *58th Annual Meeting of the Association for Computational Linguistics*, pages 7675–7688, 2020.

[204] H. Wang, Q. Zhang, Y. Wang, and H. Hu. Structured probabilistic pruning for convolutional neural network acceleration. *arXiv preprint arXiv:1709.06994*, 2017.

[205] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han. Haq: Hardware-aware automated quantization with mixed precision. In *IEEE conference on computer vision and pattern recognition*, pages 8612–8620, 2019.

[206] D. J. Watts and S. H. Strogatz. Collective dynamics of ‘small-world’networks. *nature*, 393(6684):440–442, 1998.

[207] C. White, M. Khodak, R. Tu, S. Shah, S. Bubeck, and D. Dey. A deeper look at zero-cost proxies for lightweight nas. In *ICLR Blog Track*, 2022. https://iclr-blog-track.github.io/2022/03/25/zero-cost-proxies/.

[208] M. Wistuba, A. Rawat, and T. Pedapati. A survey on neural architecture search. *arXiv preprint arXiv:1905.01392*, 2019.

[209] M. Wortsman, A. Farhadi, and M. Rastegari. Discovering neural wirings. *Advances in Neural Information Processing Systems*, 32, 2019.

[210] J. Wu, Y. Zhang, H. Bai, H. Zhong, J. Hou, W. Liu, W. Huang, and J. Huang. Pocketflow: An automated framework for compressing and accelerating deep neural networks. *Neurips workshop on CDNNRIA*, 2018.

[211] L. Xiaohu, L. Xiaoling, Z. Jinhua, Z. Yulin, and L. Maolin. A new multilayer feedforward small-world neural network with its performances on function approximation. In *2011 IEEE International Conference on Computer Science and Automation Engineering*, volume 3, pages 353–357. IEEE, 2011.

[212] L. Xie and A. Yuille. Genetic cnn. In *IEEE international conference on computer vision*, pages 1379–1388, 2017.

[213] S. Xie, A. Kirillov, R. Girshick, and K. He. Exploring randomly wired neural networks for image recognition. In *IEEE/CVF International Conference on Computer Vision*, pages 1284–1293, 2019.

[214] J. Xu, X. Tan, R. Luo, K. Song, J. Li, T. Qin, and T.-Y. Liu. Nas-bert. *27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, Aug 2021.

[215] J. Xu, X. Tan, K. Song, R. Luo, Y. Leng, T. Qin, T.-Y. Liu, and J. Li. Analyzing and mitigating interference in neural architecture search. In *International Conference on Machine Learning*, pages 24646–24662. PMLR, 2022.

[216] Y. Xu, L. Xie, X. Zhang, X. Chen, G.-J. Qi, Q. Tian, and H. Xiong. Pc-darts: Partial channel connections for memory-efficient architecture search. In *International Conference on Learning Representations*, 2019.

[217] L. F. Yang and M. Wang. Reinforcement leaning in feature space: Matrix bandit, kernels, and regret bound. *arXiv preprint arXiv:1905.10389*, 2019.

[218] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam. Netadapt: Platform-aware neural network adaptation for mobile applications. In *European Conference on Computer Vision (ECCV)*, pages 285–300, 2018.

[219] F. Yao, A. S. Rakin, and D. Fan. Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1463–1480, 2020.

[220] Y. Yin, C. Chen, L. Shang, X. Jiang, X. Chen, and Q. Liu. Autotinybert: Automatic hyper-parameter optimization for efficient pre-trained language models. *arXiv preprint arXiv:2107.13686*, 2021.

[221] Y. Ying and D.-X. Zhou. Learnability of gaussians with flexible variances. *Journal of Machine Learning Research*, 8(Feb):249–276, 2007.

[222] J. You, J. Leskovec, K. He, and S. Xie. Graph structure of neural networks. In *International Conference on Machine Learning*, pages 10881–10891. PMLR, 2020.

[223] Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, and C.-J. Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.

[224] J. Yu and T. S. Huang. Universally slimmable networks and improved training techniques. In *IEEE International Conference on Computer Vision*, pages 1803–1811, 2019.

[225] J. Yu, L. Yang, N. Xu, J. Yang, and T. Huang. Slimmable neural networks. In *7th International Conference on Learning Representations, ICLR 2019*, 2019.

[226] Z. Yuan, C. Xue, Y. Chen, Q. Wu, and G. Sun. Ptq4vit: Post-training quantization framework for vision transformers. *arXiv preprint arXiv:2111.12293*, 2021.

[227] O. Zafrir, A. Larey, G. Boudoukh, H. Shen, and M. Wasserblat. Prune once for all: Sparse pre-trained language models. *arXiv preprint arXiv:2111.05754*, 2021.

[228] D. H. Zanette. Dynamics of rumor propagation on small-world networks. *Physical review E*, 65(4):041908, 2002.

[229] V. Zantedeschi, M.-I. Nicolae, and A. Rawat. Efficient defenses against adversarial attacks. In *10th ACM Workshop on Artificial Intelligence and Security*, 2017.

[230] M. D. Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[231] R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.

[232] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[233] X. Zhang, C. Moore, and M. E. Newman. Random graph models for dynamic networks. *The European Physical Journal B*, 90(10):1–14, 2017.

[234] Y. Zhao, L. Dong, Y. Shen, Z. Zhang, F. Wei, and W. Chen. Memory-efficient differentiable transformer architecture search. *arXiv preprint arXiv:2105.14669*, 2021.

[235] H. Zhou, J. M. Alvarez, and F. Porikli. Less is more: Towards compact cnns. In *European Conference on Computer Vision (ECCV)*, pages 662–677. Springer, 2016.

[236] Z. Zivkovic. Improved adaptive gaussian mixture model for background subtraction. In *17th International Conference on Pattern Recognition*, volume 2, pages 28–31. IEEE, 2004.