UNIVERSITY OF CALIFORNIA SAN DIEGO

End-to-end Customization of Efficient, Private, and Robust Neural Networks

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

in

Electrical Engineering (Computer Engineering)

by

Mohammad Samragh Razlighi

Committee in charge:

2021

University of California San Diego

2021

DEDICATION

To my loving parents Monireh and Aziz

EPIGRAPH

"We know very little, and yet it is astonishing that we know so much, and still more astonishing that so little knowledge can give us so much power."

—Bertrand Russell

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my PhD advisor professor Farinaz Koushanfar for her invaluable support and guidance throughout my doctorate studies. Aside from academic advising, she taught me to have high expectations of myself, dream big, work hard, and make impact.

I was honored to work closely with professor Tara Javidi, whose academic curiosity and thirst to discover the unknown have been exemplar to me. I am also deeply grateful to the committee members Professor Truong Nguyen, Professor Ryan Kastner, and Professor Hao Su for their valuable suggestions during my studies.

I was fortunate to meet and work with outstanding researchers, interacting with whom has made my PhD journey full of excitement, joy, and competence. I want to thank Dr. Siam Hussain, Dr. Sadegh Riazi, Dr. Bita Rouhani, Mohammad Ghasemzadeh, Mojan Javaheripi, Dr. Ebrahim Songhori, Dr. Azalia Mirhosseini, Dr. Mohsen Imani, Dr. Ilya Razenshteyn, Dr. Hao Chen, Dr. Kristin Lauter, Xinqiao Zhang, Gregory Fields, Dr. Hossein Hosseini, and Dr. Joseph Soriaga.

I am in debt to my parents and siblings for they have immersed me in their unconditional love. As I am writing these sentences, I cannot find the proper words to describe my gratitude to them.

This dissertation is based upon several papers that are either published or pending publication. In particular, Chapter 3 has been published as: Samragh, Mohammad, Mojan Javaheripi, and Farinaz Koushanfar. "Encodeep: Realizing bit-flexible encoding for deep neural networks." ACM Transactions on Embedded Computing Systems (TECS) 19.6 (2020): 1-29. The thesis author and Mojan Javaheripi made equal contributions to this work.

Chapter 4 has been published as: Razlighi, Mohammad Samragh, Mohsen Imani, Farinaz Koushanfar, and Tajana Rosing. "Looknn: Neural network with no multiplication." In Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, pp. 1775-1780. IEEE, 2017. The thesis author was the primary contributor of this material.

Chapter 5 is based on a paper that is pending publication. The paper will appear as “Siam U

Hussain, Mojan Javaheripi, Mohammad Samragh, and Farinaz Koushanfar. COINN: Crypto/ML Codesign for Oblivious Inference via Neural Networks. ACM Conference on Computer and Communications Security (CCS), 2021.” The thesis author along with Dr. Siam Hussain and Mojan Javaheripi were equal contributors to this material.

The material in Chapter 6 is based on a paper published as: Samragh, Mohammad, Siam Hussain, Xinqiao Zhang, Ke Huang, and Farinaz Koushanfar. "On the Application of Binary Neural Networks in Oblivious Inference." In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 4630-4639. 2021. The thesis author was the main contributor to this material.

Finally, Chapter 7 was published as: Javaheripi, Mojan, Mohammad Samragh, Gregory Fields, Tara Javidi, and Farinaz Koushanfar. "Cleann: Accelerated trojan shield for embedded neural networks." In 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD), pp. 1-9. IEEE, 2020. The thesis author and Mojan Javaheripi made equal contributions to this work.

VITA

2015	B. S. in Electrical Engineering, Sharif University of Technology, Tehran, Iran
2018	M. S. in Electrical Engineering (Computer Engineering), University of California San Diego, USA
2021	Ph. D. in Electrical Engineering (Computer Engineering), University of California San Diego, USA

PUBLICATIONS

Samragh, Mohammad, Hossein Hosseini, Aleksei Triastcyn, Kambiz Azarian, Joseph Soriaga, and Farinaz Koushanfar. "Unsupervised Information Obfuscation for Split Inference of Neural Networks." arXiv preprint arXiv:2104.11413 (2021). ICML-21 Workshop on Information-Theoretic Methods for Rigorous, Responsible, and Reliable Machine Learning.

Javaheripi, Mojan, **Mohammad Samragh**, Bita Darvish Rouhani, Tara Javidi, and Farinaz Koushanfar. "Hardware/Algorithm Codesign for Adversarially Robust Deep Learning." IEEE Design & Test 38, no. 3 (2021): 31-38.

Samragh, Mohammad, Siam Hussain, Xinqiao Zhang, Ke Huang, and Farinaz Koushanfar. "On the Application of Binary Neural Networks in Oblivious Inference." In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 4630-4639. 2021.

Javaheripi, Mojan, **Mohammad Samragh**, Gregory Fields, Tara Javidi, and Farinaz Koushanfar. "Cleann: Accelerated trojan shield for embedded neural networks." In 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD), pp. 1-9. IEEE, 2020.

Samragh, Mohammad, Mojan Javaheripi, and Farinaz Koushanfar. "Encodeep: Realizing bit-flexible encoding for deep neural networks." ACM Transactions on Embedded Computing Systems (TECS) 19, no. 6 (2020): 1-29.

Javaheripi, Mojan, **Mohammad Samragh**, Bita Darvish Rouhani, Tara Javidi, and Farinaz Koushanfar. "CuRTAIL: ChaRacterizing and thwarting AdversarIal deep learning." IEEE Transactions on Dependable and Secure Computing 18, no. 2 (2020): 736-752.

Javaheripi, Mojan, **Mohammad Samragh**, Tara Javidi, and Farinaz Koushanfar. "GeneCAI: gene tic evolution for acquiring c ompact AI." In Proceedings of the 2020 Genetic and Evolutionary Computation Conference, pp. 350-358. 2020.

Javaheripi, Mojan, **Mohammad Samragh**, Tara Javidi, and Farinaz Koushanfar. "AdaNS: Adaptive non-uniform sampling for automated design of compact DNNs." IEEE Journal of Selected Topics in Signal Processing 14, no. 4 (2020): 750-764.

Imani, Mohsen, **Mohammad Samragh Razlighi**, Yeseong Kim, Saransh Gupta, Farinaz Koushanfar, and Tajana Rosing. "Deep learning acceleration with neuron-to-memory transformation." In 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 1-14. IEEE, 2020.

Javaheripi, Mojan, **Mohammad Samragh**, and Farinaz Koushanfar. "Peeking into the black box: A tutorial on automated design optimization and parameter search." IEEE Solid-State Circuits Magazine 11, no. 4 (2019): 23-28.

Imani, Mohsen, Sahand Salamat, Behnam Khaleghi, **Mohammad Samragh**, Farinaz Koushanfar, and Tajana Rosing. "Sparsehd: Algorithm-hardware co-optimization for efficient high-dimensional computing." In 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 190-198. IEEE, 2019.

Rouani, Bita Darvish, **Mohammad Samragh**, Tara Javidi, and Farinaz Koushanfar. "Safe machine learning and defeating adversarial attacks." IEEE Security & Privacy 17, no. 2 (2019): 31-38.

Samragh, Mohammad, Mojan Javaheripi, and Farinaz Koushanfar. "AutoRank: Automated rank selection for effective neural network customization." In Proceedings of the ML-for-Systems Workshop at the 46th International Symposium on Computer Architecture (ISCA'19). 2019.

Riazi, M. Sadegh, **Mohammad Samragh**, Hao Chen, Kim Laine, Kristin Lauter, and Farinaz Koushanfar. "XONN: Xnor-based oblivious deep neural network inference." In 28th USENIX Security Symposium (USENIX Security 19), pp. 1501-1518. 2019.

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

Ghasemzadeh, Mohammad, **Mohammad Samragh**, and Farinaz Koushanfar. "ReBNet: Residual binarized neural network." In 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 57-64. IEEE, 2018.

Riazi, M. Sadegh, **Mohammad Samragh**, and Farinaz Koushanfar. "Camsure: Secure content-addressable memory for approximate search." ACM Transactions on Embedded Computing Systems (TECS) 16, no. 5s (2017): 1-20.

Razlighi, Mohammad Samragh, Mohsen Imani, Farinaz Koushanfar, and Tajana Rosing. "Looknn: Neural network with no multiplication." In Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, pp. 1775-1780. IEEE, 2017.

Samragh, Mohammad, Mohammad Ghasemzadeh, and Farinaz Koushanfar. "Customizing neural networks for efficient FPGA implementation." In 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 85-92. IEEE, 2017.

ABSTRACT OF THE DISSERTATION

End-to-end Customization of Efficient, Private, and Robust Neural Networks

by

Mohammad Samragh Razlighi

Doctor of Philosophy in Electrical Engineering (Computer Engineering)

University of California San Diego, 2021

Professor Farinaz Koushanfar, Chair

Advancements in machine learning (ML) algorithms, data acquisition platforms, and high-end computer architectures have fueled an unprecedented industrial automation. An ML algorithm captures the dynamics of a task by learning an abstract model from domain-specific data. Once the model is trained by the ML algorithm, it can perform the underlying task with relatively high accuracy. This thesis is specifically focused on Deep Neural Networks (DNNs), a modern class of ML models that have shown promising performance in various applications. Thanks to DNNs, the breadth of automation has been expanded to tasks that were formerly too complex to be performed by computers; nowadays DNNs establish the foundation of applications such as voice recognition, medical image analysis, face authentication, to name a few.

Despite DNNs' benefits, their deployment in real-world applications may be circumscribed by several factors. First, DNNs are computationally complex and their efficient execution on resource-constrained edge devices is a critical challenge. Second, users of DNN-based applications are often required to expose their data to the service provider, which may violate their privacy. Third, DNN models may fail to function correctly in the presence of malicious attackers. Having the aforementioned challenges in mind, it is a paramount challenge to design DNN-based systems that are efficient to execute, ensure users' privacy, and are robust to malicious attacks.

This dissertation provides holistic customization techniques that pave the way for efficient, private, and robust DNN inference. The key contributions of the thesis are as follows:

- **(Efficiency):** Development of encoded DNNs, a new family of memory-efficient neural networks. The thesis author's contributions provide customization techniques that enable incorporation of nonlinear encoding to the computation flow of neural networks. An end-to-end framework is introduced to facilitate encoding, bitwidth customization, fine-tuning, and implementation of neural networks on FPGA platforms.

- **(Efficiency):** Introducing the concept of lookup-table based execution of encoded neural networks. The proposed method replaces floating-point multiplications with look-up table search. A memory-based hardware architecture is then proposed to execute the lookup-based multiplications and accelerate encoded DNN inference.

- **(Privacy):** Establishing customized solutions for oblivious inference, where a client holds a data sample and a server holds a DNN model. After running the oblivious inference protocol, the client receives the inference result without revealing her input to the server. This thesis proposes automated customization solutions to speed up the oblivious inference while maintaining a high inference accuracy.

- **(Robustness):** Development of solutions for online detection of neural Trojan triggers, a class of malicious attacks that cause a DNN to perform faulty inferences. The thesis proposes a novel methodology that enhances robustness to Trojan attacks by leveraging dictionary learning and sparse approximation.

Chapter 1

Introduction

A DNN training algorithm extracts information related to a specific attribute from a massive corpus of data. During training, the extracted information is abstracted into the model. The trained model can then be used to automate the underlying task, e.g., a DNN can be trained to perform voice recognition, pose estimation, medical diagnosis, financial data analysis, etc. In real-world applications, domain-specific customization of DNNs is necessary prior to deployment. Particularly, the customization should satisfy three important properties explained below.

- **(Efficiency):** Early DNNs were developed to run on server-grade Graphic Processing Units (GPUs), which have a relatively high compute capacity and power consumption. However, many embedded applications require low power execution on devices with limited compute capacity. Therefore, customization techniques that can reduce the computational complexity and power consumption of DNNs are necessary in embedded settings.

- **(Privacy):** DNNs were originally designed for plaintext inference wherein the input to the neural network would be revealed to the model owner. A new line of research allows performing inference on (unintelligible) encrypted data but it incurs significant computation and communication overheads. Customization techniques that reduce the overhead of secure computation are crucial for practical privacy-preserving inference.

- **(Robustness):** Recent Trojan attacks show that DNNs are subject to failure when the input data and the model are poisoned. Customization techniques are of paramount importance for enhancing robustness to malicious Trojan attacks.

This dissertation addresses the above challenges in part by developing domain-specific customization techniques. The remainder of this chapter summarizes the research contributions made by the author in each category.

1.1 Efficiency

The massive computational complexity and memory footprint of DNNs hinders their efficient execution on resource-limited devices. Previous work demonstrates a significant redundancy in neural network parameters [13], suggesting that the underlying operations can be simplified with minimal loss of inference accuracy. Inspired by this fact, this dissertation proposes end-to-end customization techniques along with prototype hardware accelerators for efficient DNN inference. The thesis author's work in this category is summarized as follows.

1.1.1 Weight and activation encoding in DNNs

One major contributor to the inefficiency of DNN inference is its immense memory footprint. In high level, performing DNN inference on a hardware platform requires transferring data from the storage unit to the compute unit, performing the computations, and writing the output back to the memory. In practice, the data transfer causes a significant delay and power consumption when performing neural network inference.

To mitigate the data transfer bottleneck, Chapter 3 proposes the encoded neural network as a memory efficient DNN realization. The result of the research is an end-to-end framework, dubbed EncoDeep, that facilitates encoding, bitwidth customization, fine-tuning, and implementation of neural networks on FPGA platforms. EncoDeep incorporates nonlinear encoding to the computation flow of neural networks to save memory. In comparison to the raw full-precision activation values, the encoded features demand a significantly smaller storage capacity; The small memory footprint of EncoDeep allows one to perform data transfers using on-chip streaming buffers inside an FPGA without frequent accesses to the off-chip DRAM. A fully-automated optimization algorithm is also developed to determine the flexible encoding bitwidths across network layers. EncoDeep full-stack framework comprises of a compiler which takes a high-level Python description of an arbitrary neural network. The compiler then instantiates the corresponding elements from EncoDeep Hardware library for FPGA implementation. Proof-of-concept evaluations demonstrate an average of $\mathbf{4.65}\times$ throughput improvement compared to stand-alone weight encoding for small-scale DNNs, and an average of $\mathbf{3.6}\times$ throughput improvement compared to contemporary FPGA accelerators for large-scale DNN inference.

1.1.2 Lookup-table based multiplication in DNNs

The computational complexity of neural network inference is mainly due to the matrix multiplications. Chapter 4 presents LookNN, a methodology to replace floating-point multiplications with look-up table search to reduce the runtime and power consumption of multiplications within the computing unit. To this end, the weight parameters of a pre-trained neural network are encoded, such that the model's accuracy is minimally affected. This step adapts the original model into a format compatible with lookup-based multiplication. Next, enhanced general purpose processors are proposed for searching look-up tables: each processing element in a GPU can be augmented with a small associative memory, enabling it to bypass redundant computations. Simulations on AMD Southern Island GPU architecture show that LookNN results in $\mathbf{2.2}\times$ energy saving and $\mathbf{2.5}\times$ speedup running four different neural network applications with zero loss of accuracy. For the same four applications, if we tolerate an accuracy drop of less than 0.2%, LookNN can achieve an average of $\mathbf{3}\times$ energy improvement and $\mathbf{2.6}\times$ speedup compared to the traditional GPU architecture.

1.2 Privacy

Machine-learning-as-a-service (MLaaS) is a rapidly growing business model wherein a service provider holds a trained DNN model, and clients query the DNN with their data to perform inference. In many MLaaS applications, the data owned by the users contains sensitive information that should be kept private. This thesis introduces holistic DNN customization techniques that enable privacy-preserving inference, a.k.a. oblivious inference, with unprecedented efficiency. The results of the author's research are summarized in the following sections.

1.2.1 Customizing quantization and clustering for secure DNN inference

Fixed-point quantization is one of the main avenues that researchers have followed for optimizing plaintext DNN inference. However, existing quantization schemes cannot be directly optimize privacy-preserving (ciphertext) execution of DNNs. Chapter 5 presents the author's research in the development of DNN customization techniques that utilize quantization and encoding for efficient privacy-preserving DNN inference. The objective of DNN customization here is to speedup the secure inference while maintaining a high accuracy. This objective can be achieved through domain-specific low-bit quantization and weight encoding schemes that enable efficient ciphertext computations. The result of the research is a framework, called COINN, that shows an unprecedented level of efficiency in oblivious inference. The framework achieves a runtime speedup of $3\times$ to $7\times$ over the state-of-the-art and is scalable to run complex DNNs with over 100 layers.

1.2.2 Customizing binary neural networks for secure DNN inference

Chapter 6 explores the application of Binary Neural Networks (BNN) in oblivious inference, making two main contributions to the field. First, light-weight cryptographic protocols are designed to exploit the unique characteristics of BNNs towards development of efficient oblivious inference protocols. Second, a single-shot training process is developed to dynamically explore the runtime-accuracy tradeoff of BNNs for oblivious inference. While previous works trained multiple BNNs with different computational complexities (which is cumbersome due to the slow convergence of BNNs), the proposed techniques train a single BNN that can perform inference under different computational budgets. Compared to the contemporary state-of-the-art in oblivious inference of non-binary DNNs, the proposed method reaches 2$\times$ faster inference at the same accuracy. Compared to XONN, the state-of-the-art in oblivious inference of binary networks, the method achieves between 2$\times$ and 11$\times$ faster inference while obtaining higher accuracy.

1.3 Customized Solutions to Assure DNN Robustness against Trojan Attacks

Chapter 7 proposes an end-to-end framework, named CleaNN, for online detection of Trojans in embedded DNN applications. In a Trojan attack, the attacker injects a backdoor in the DNN during training by poisoning some percentage of the training data with a Trojan trigger. In the absence of the trigger, the resulting DNN has a high inference accuracy on the benign data. When the trigger is applied to the data, the DNN makes unexpected, incorrect decisions that might endanger the safety of the underlying system.

The author's proposed lightweight mitigation methodology can detect Trojaned samples without the need of labeled data or model fine-tuning. It also does not make prior assumptions about the trigger or the attack algorithm. The method works by learning a dictionary over benign data to characterize its statistics. By applying sparse approximations via the learned dictionary, the Trojan triggers can be identified and cleared from infected samples.

Chapter 2

Background

This chapter summarizes the necessary technical background. Section 2.1 outlines neural networks and their components. Basic concepts in secure function evaluation are described in Section 2.2.

2.1 Neural Networks

A neural network is composed of a stack of layers, where the output of each layer serves as the input of one or several subsequent layers. In contemporary DNNs, the layers can be categorized as linear and nonlinear. The linear layers considered in this thesis include convolution (CONV), fully-connected (FC), batch normalization (BN), and average pooling (AP). The nonlinear layers studied in this thesis are rectified linear unit (ReLU) and max pooling (MP). Below we briefly explain each layer.

Convolution. A (CONV) layer receives a 3-way input tensor $\mathbf{X} \in \mathbb{R}^{c \times d_1 \times d_1}$ and outputs a 3-way output tensor $\mathbf{Y} \in \mathbb{R}^{m \times d_2 \times d_2}$. Each $d_2 \times d_2$ channel in the output is computed by convolving a $c \times k \times k$ kernel tensor through the input tensor and adding a bias term b. Repeating the convolution with m different kernels results in the complete set of m channels in the output.

The collection of all kernels is called the weight tensor $\mathbf{W} \in \mathbb{R}^{m\times c\times k\times k}$ and the collection of all additive terms is called the bias vector $\mathbf{b} \in \mathbb{R}^{m}$. The CONV layer can be equivalently represented as a matrix-multiplication followed by bias addition $Y = W \cdot X + \mathbf{b}$. Here, $W \in \mathbb{R}^{m\times n}$ is achieved by reshaping the original 4-way tensor into a $2D$ matrix. Each column of $X \in \mathbb{R}^{n\times l}$ represents the $c \times k \times k$ features at a certain window of the input tensor. Each element of the output Y is computed via a vector dot product (VDP) and the total number of VDPs required for the matrix-multiplication is $m \times l$.

Fully-Connected. A fully-connected (FC) layer converts a vector $\mathbf{x} \in \mathbb{R}^{n}$ to another vector $\mathbf{y} = W \times \mathbf{x} + \mathbf{b}$. Here, $W \in \mathbb{R}^{m\times n}$ and $\mathbf{b} \in \mathbb{R}^{m}$ are called the weight and bias, respectively.

Batch Normalization. The outputs of CONV and FC layers are normalized via batch normalization (BN), such that the features have controlled mean and standard deviations. At the training time, the BN layer converts a feature x to $y = \gamma\frac{x-E[x]}{\sqrt{var[x]+\varepsilon}} + b$, where γ and b are trainable parameters and ε is a small constant value to avoid division by zero. $E[x]$ and $var[x]$ are the mean and variance of features which are computed over batches of training data. At the test time, the mean and variance are replaced with constant values obtained from moving averages during training. Therefore, the test time computation can be simplified as $y = \alpha x + \beta$, where $\alpha = \frac{\gamma}{\sqrt{var[x]+\varepsilon}}$ and $\beta = b - \frac{\gamma E[x]}{\sqrt{var[x]+\varepsilon}}$.

Pooling. Pooling layers reduce the dimensionality of feature-maps by taking the average (AP) or maximum (MP) of $k \times k$ windows in the input. Assuming the $k \times k$ windows are non-overlapping, a pooling layer reduces the data dimensionality from $c \times d_1 \times d_1$ to $c \times \frac{d_1}{k} \times \frac{d_1}{k}$.

ReLU. A ReLU layer truncates negative values to zero.

2.2 Secure Function Evaluation

Secure function evaluation allows two parties to compute the output of a function that takes inputs from both of them, without revealing the secret input from either party to the other.

Following the convention in the literature, this thesis refers to the involved parties as Alice and Bob. Several key concepts of secure function evaluation are explained below.

Secure Function Evaluation Protocol. A Secure Function Evaluation (SFE) protocol is a set of rules specifying the messages communicated between Alice and Bob. By following these rules, they jointly evaluate a function $f(x_A, x_B)$ that takes the x_A from Alice and x_B from Bob without disclosing any information about Alice's data to Bob and vice versa. Depending on protocol agreements, the result of the computation can be exposed to both parties, only one of them, or neither of them.

Additive Secret Sharing (AS) is a method for distributing a secret x between Alice and Bob such that Alice holds $[\![x]\!]_A = x + r$ and Bob holds $[\![x]\!]_B = -r$, where r is a random value. Individually, both $[\![x]\!]_A$ and $[\![x]\!]_B$ are random values, hence, Alice or Bob cannot independently decipher the original message x. Only by combining $[\![x]\!]_A$ and $[\![x]\!]_B$ can one recover the actual secret as $x = [\![x]\!]_A + [\![x]\!]_B$. There exist standard SFE protocols to perform addition and multiplication on secret-shared data such that the result is also shared between the two parties. As we show in this thesis, these protocols can be used in oblivious inference to ensure that neither the input nor the output of a linear DNN layer is revealed to the involved parties. Curious readers are referred to [14] for more details about secret sharing.

Oblivious Transfer (OT) is a protocol between two parties – a sender (Bob) who has two messages (μ_0, μ_1), and a receiver (Alice) who has a selection bit $i \in \{0, 1\}$ [15]. Through OT, Alice obtains the intended message μ_i, without revealing the selection bit i to Bob. Alice does not learn the other message μ_{1-i}. OT requires public key cryptography, which is costly in general. In the following, we introduce more efficient methods for OT computation.

OT extension enables extending a constant number of 'base OTs' to a large number of OTs through cheaper symmetric key cryptography [16]. The first step in OT-extension is called Random OT (ROT) [17]. In ROT, Alice provides the selection bit i and Bob does not provide any input. After ROT execution, Bob receives two random 128-bit keys (k_0, k_1) and Alice receives k_i.

The final step of OT-extension is as follows: Bob computes $\{v_0, v_1\} = \{\mu_0 \oplus H(k_0), \mu_1 \oplus H(k_1)\}$, where the $\oplus$ operator denotes bit-wise XOR and $H(k)$ is a cryptographically-secure random number generator [18] with k as the seed. Bob transmits $\{v_0, v_1\}$ to Alice, who computes $\mu_i = v_i \oplus H(k_i)$.

Garbled Circuit (GC) is an SFE protocol that can be used for evaluation of an arbitrary function (linear or non-linear). In GC, Alice (the garbler) and Bob (the evaluator) securely compute the output of a function $f(x_A, x_B)$ with secret inputs from Alice and Bob, respectively. The function $f(\cdot,\cdot)$ is represented via a Boolean circuit known to both parties. By evaluating the Boolean circuit gate by gate, the two parties are able to securely compute f. Curious readers are invited to read [19–21] for more details about the protocol and its efficient variants. The representation of a b-bit private scalar can be converted from GC to AS and vice versa by executing a b-bit addition through GC [22]. GC is less efficient than AS for computing multiplication and addition, however it is the most efficient protocol to date that is able to securely evaluate generic non-linear functions. Additionally, operations such as bit shift and XOR can be performed inn GC with zero *amortized* cost (without communication between Alice and Bob).

Chapter 3

Weight and activation encoding in DNNs

Deep Neural Networks (DNNs) are being widely developed for various machine learning applications, many of which are required to run on embedded devices. In the realm of embedded DNNs, real-time execution under severe power limitations is hard to satisfy [23, 24]. Contemporary research has focused on the FPGA-based acceleration of DNNs [25–28]. However, FPGAs are inherently limited in terms of on-chip memory capacity. Thus, the high-storage requirement of DNN models hinders an efficient and low power execution on FPGAs.

To reduce the computational complexity and memory requirement of DNNs, several pre-processing algorithms have been proposed. The existing methods generally convert conventional DNNs into compact representations that are better suited for execution on embedded devices. Examples of such compacting methods include quantization [29, 30], binarization [26, 31], tensor decomposition [32], parameter pruning [33], and compression with nonlinear encoding [34, 35]. A higher compression rate might not always translate to better hardware performance as the platform constraints could interfere with the intended compaction methodology [36].

This chapter specifically focuses on nonlinear encoding and provides solutions to tackle the challenges associated with optimizing physical performance. Encoding network parameters is rather beneficial as it reduces the memory footprint, i.e., the main source of delay and power

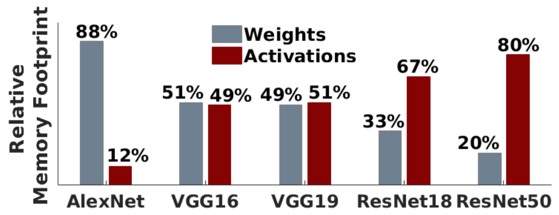


Figure 3.1: Relative memory footprint of weights and activations for various DNNs, evaluated on 10 samples of ImageNet.

consumption in FPGA accelerators. To devise a practical solution for implementing encoded DNNs, we simultaneously identify and address four critical issues.

DNN memory footprint is imposed by either weights or feature-maps. Figure 3.1 shows the relative memory requirements in several popular DNN models. As can be seen, the memory footprint of activations is notable; however, contemporary research mainly targets the (static) DNN weights for nonlinear quantization [34,35,37]. Developing online mechanisms for activation encoding can significantly reduce the memory footprint of DNN models. However, nonlinear quantization destabilizes DNN training by adding non-differentiable elements to the model. Therefore, novel computation routines must be developed to approximate gradients for DNN fine-tuning. Additionally, specifying the encoding bitwidth across all DNN layers by handcrafted try-and-error is exhaustive and generally sub-optimal. Hence, automated and intelligent solutions for bitwidth optimization are highly preferable. Finally, designing accelerators that are customized per application/hardware is cumbersome. Thus, easy-to-use tools are needed to ensure low, non-recurring engineering costs.

To tackle the aforementioned challenges, we introduce EncoDeep, a unified framework that facilitates encoding, training, bitwidth customization, and automated implementation of encoded DNNs on FPGA platforms. EncoDeep software stack allows users to automatically configure the encoding bitwidth across all DNN layers and retrain the encoded DNN. The hardware library of EncoDeep provides a set of configurable DNN layers that can be instantiated to compose a fully-functional DNN. In summary, the contributions of this chapter are listed as follows:

- Introducing a novel methodology for the (online) encoding of DNN **activations**. We establish the gradient computation routines required to fine-tune encoded DNNs, enabling restoration of DNN accuracy after encoding.
- Introducing an automated algorithm for customizing per-layer encoding bitwidths. Inspired by reinforcement learning, we establish an action-reward-state system to find a bitwidth configuration that minimally affects DNN accuracy while maximally reducing memory footprint.
- Establishing a hardware library for the bit-flexible implementation of customized encoded DNN layers. Activation encoding lowers memory footprint and facilitates the use of streaming buffers for inter-layer feature transmission.
- Providing an API for fast and easy hardware implementation of encoded DNNs. Developers describe the DNN as high-level Python code which is then automatically converted to Vivado_HLS.
- Performing extensive evaluations on various datasets and DNN architectures. On MNIST, SVHN, and CIFAR-10, EncoDeep demonstrates an average of $\mathbf{4.65}\times$ throughput improvement compared to stand-alone weight encoding. To uncover the benefits of encoding, we compare EncoDeep with six fixed-point FPGA accelerators on ImageNet, showing an average of $\mathbf{3.6}\times$ and $\mathbf{2.54}\times$ improvement in throughput and performance-per-watt, respectively.

3.1 Overview and insights

EncoDeep design flow is composed of an interlinked optimization scheme where algorithmic DNN compaction methods and hardware-level customization are performed in sync. In this section, we describe EncoDeep insights in high-level and look at the main components of our framework.

3.1.1 Streaming-based On-chip Execution

Traditional DNN accelerators store the weights and activations (features) of layers in the off-chip DRAM since commodity FPGAs are often limited in terms of on-chip memory capacity. Figure 3.2 (top) demonstrates the computation flow of DNNs in such settings. Alternatively, the weights and computed activations could be stored and accessed within the FPGA design using streaming buffers as depicted in the bottom of Figure 3.2. The benefits of the latter approach are three-fold: (i) it avoids the power-hungry and high-latency access to off-chip DRAM. (ii) The computation engines responsible for each DNN layer can be customized to comply with the pertinent layer. (iii) The streaming buffers allow pipelining for the computation engines to increase throughput. Although on-chip execution of DNNs is beneficial in many aspects, the memory requirement for weights and activations of DNN layers is often beyond the (limited) capacity of commodity FPGAs. To address this, EncoDeep employs nonlinear quantization to reduce memory footprint such that the weights/activations can be accommodated within FPGA block-RAMs.

3.1.2 Memory Compression

Quantization allows a reduction in memory footprint by approximating numerical values. To perform quantization, a finite set of best representatives (a.k.a. bins) are selected and each value is approximated with the closest bin. Perhaps the most popular quantization is fixed-point approximation. Figure 3.3-a depicts the bins for an unsigned fixed-point quantization. In this setting, quantization bins are fixed to certain points (e.g., $\{0.00, 0.25, 0.50, 0.75\}$), regardless of data distribution. Alternatively, in nonlinear quantization, the bins are carefully selected to

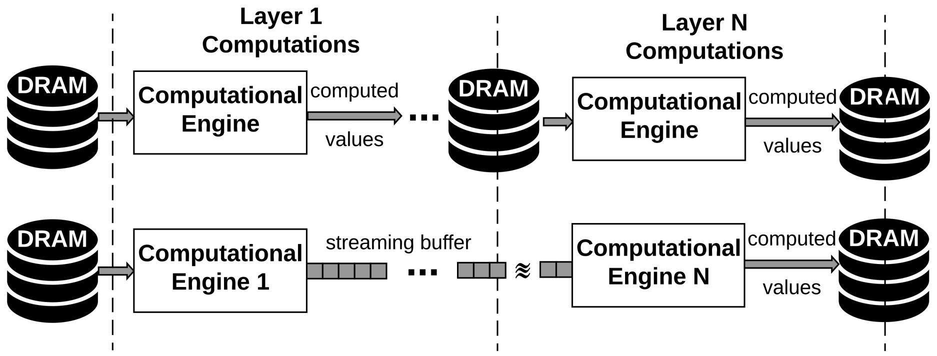
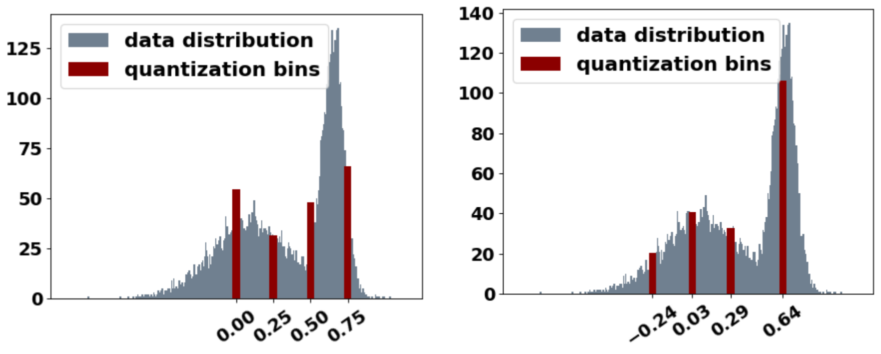


Figure 3.2: The workflow of traditional vs. streaming-based DNN inference. The top diagram shows the conventional approach where all resources are allocated to one computational engine and layer input/outputs are continuously read/written from/to off-chip memory. The bottom diagram presents a streaming-based approach where each layer is allocated a different computational engine and communications with off-chip memory are limited to the first/last layer.

best represent the data as shown in Figure 3.3-b. In this example, both fixed-point and nonlinear quantizations require the same number of bits to represent the (approximated) data: each real-valued signal can be represented with 2 bits when there are 4 quantization bins. However, the approximation error associated with the nonlinear scheme is drastically lower.

For a fixed nonlinear quantization bitwidth, the approximation error increases as the standard deviation σ of the data increases. Nevertheless, this error can be compensated by increasing the number of quantization bits; Figure 3.4 shows that for a large enough number of bits ($b > 6$ in this example), the error converges to zero. DNNs inherently have a low standard deviation due to specific measures taken during training to ensure convergence. In particular, to avoid exploding gradient values and promote a smooth convergence, contemporary DNNs comprise *batch normalization* which normalizes layer activation values. Moreover, to prevent over-fitting and drastic neuron transitions, weight regularization is used during training to suppress large weights. We empirically demonstrate this property in Figure 3.5 by plotting the σ range across DNN layers for all our benchmarks. Such small σ range allows low-error estimation of DNN parameters/activations with very few bits.



(a) Fixed-point (b) Nonlinear

Figure 3.3: Histogram of data samples and the quantization bins in fixed-point and nonlinear quantization. In this example, there are 4 quantization bins and scalars are encoded with $Log_2(4) = 2$ bits.

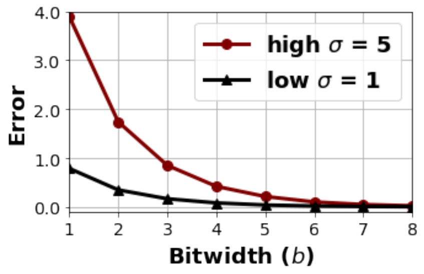


Figure 3.4: Nonlinear quantization error versus bitwidth for two Gaussian data distributions with $\sigma = 1$ and $\sigma = 5$.

3.2 Related Work

To enable ubiquitous deployment of DNNs, several recent research efforts have focused on DNN acceleration [26, 35, 38–47]. In a parallel track, designing efficient DNN graphs and architectural optimization has gained attraction from the community [4, 48–51]. EncoDeep bridges the gap between these two research tracks by incorporating algorithm-hardware co-design. As neural networks are memory-intensive, devising methods to decrease the memory footprint can significantly enhance accelerator performance in terms of throughput and power consumption. Perhaps the most popular method for neural network compression is network pruning, where network parameters with insignificant contributions to the model's accuracy are

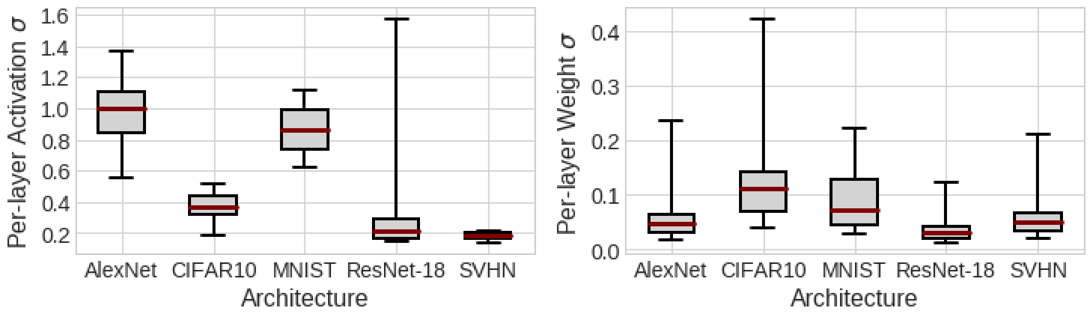


Figure 3.5: Standard Deviation (σ) range for activations (left) and weights (right) across DNN layers. Here, the black dot represents the mean σ for each benchmark.

removed. Many researchers have developed valuable work in DNN compression with pruning using either structured [52–59] or non-structured pruning [60]. Hardware accelerators have also been proposed to perform fast and efficient inference using pruned neural networks, e.g., [61,62]. The focus of our work, however, is another attractive solution for neural network compression: inference with few-bit encoded values per weight/neuron. In practice, one might employ both pruning and encoding to achieve better compression results [34]. However, to better discuss the contributions of this chapter, in our experiments we focus solely on the encoding technique.

Several methods for training DNNs with few bits have been proposed in [29,63–69]. QNN [29] was perhaps the first work to suggest extreme neural network quantization with binary ($\in \{\pm 1\}$) parameters and activations. Following their work, many researchers have proposed low-bit DNNs with improved accuracy. Authors of XNOR-Net [66] suggest computing the average absolute value of each input vector to the convolution operation in the forward pass. They show that multiplying this average value by the corresponding XNOR-Popcount result improves the inference accuracy. WRPN [64] shows that scaling layer widths uniformly can deliver more accurate low-bit DNNs. One immediate shortcoming of this approach is the quadratic increase in the memory and computational complexity as the scaling factor grows. ABC-Nets [67] proposes multi-bit binary approximation of weights and activations of DNNs. Their method shows great accuracy improvement at the expense of training every bitwidth configuration from scratch. HBNN [69] proposes to incorporate multi-level binarization while enabling heterogeneous level selection across layers. Although this approach achieves high memory efficiency, it still has limited accuracy due to binary approximation.

Instead of using strict binary values, as proposed by the above works, our proposal uses low-bit non-linear encodings which allows high-precision arithmetics with a low memory footprint. By leveraging non-linear encoding, we can scale down layer widths (as opposed to WRPN [64]) to match the memory of a wide binary neural network while still enjoying higher accuracy. Compared to ABC-Nets' multiple rounds of training from scratch, our experimental results show that the one-time post-training approach of EncoDeep can extract better (heterogeneous) bitwidth configurations with higher accuracy. Additionally, while the above works mainly focus on developing theoretical memory improvements, EncoDeep adopts hardware-algorithm co-design to show practical performance boost and memory reduction on hardware.

To create more hardware-friendly binary DNNs, ReBNet [26] proposes co-designing a DNN with residual binary approximation and an FPGA accelerator. LUTNet [68] takes a step forward and directly incorporates hardware characteristics such as the LUT structure of FPGAs into the designed activation function. These methodologies improve the hardware efficiency of BNNs, yet their accuracy is bounded. Similar to the above work, EncoDeep incorporates DNN-hardware co-design. However, unlike the above works, EncoDeep specifically configures the bitwidths across DNN layers by finding the optimal accuracy-memory Pareto front. This customization allows EncoDeep to achieve higher accuracy by using flexible (heterogeneous) bitwidths across DNN layers. Such heterogeneity is also specifically supported by EncoDeep modular hardware design and the accompanying compiler.

DoReFa-Net [63] incorporates specific training procedures to enable fixed-point quantization for both weights and activations. HWGQ [65] mentions that low-bit approximation of activations is more difficult than weights. The authors also note that fixed-point quantization with uniformly spaced quantization bins does not deliver the minimum approximation error. Thus, during the training phase, the authors propose to approximate the distribution of activation units via a half-way Gaussian prior and find non-uniform quantization bins accordingly. In our work, we show that it is possible to create a non-uniform quantizer *after* the DNN is trained, which has several direct benefits: first, EncoDeep does not impose extra overhead on the original DNN training and therefore training convergence speed is not altered. Second, EncoDeep does not need to assume a Gaussian prior on the activation distribution and can be applied to arbitrary distributions. Third, EncoDeep can efficiently tune the number of encoding bits across layers without training every configuration from scratch.

Nonlinear encoding allows for fixed-point arithmetics accompanied by a low storage requirement. Perhaps the closest method to our proposal is a stand-alone weight encoding, with no activation encoding, originally proposed in [34, 35, 37]. Weight encoding significantly reduces the memory footprint of model parameters but the activation units (especially in convolution layers) still require a large capacity of memory. To address this challenge, we extend the encoding to the activations of neural networks and introduce training routines for the corresponding encoded activations. In addition, prior work utilizes hand-crafted or rule-based heuristics to determine the encoding bitwidth. Such manual methods are generally sub-optimal and incur a drastic engineering cost. To address this issue, we propose an automated cross-layer bitwidth selection algorithm that aims to capture the accuracy/memory trade-off.

In a concurrent track, designing automated and easy-to-use tools for FPGA implementation of DNNs has been the focus of contemporary research [25, 70–74]. These works aim to maximize the throughput of fixed/floating-point DNN inference by distributing FPGA resources among parallel computing engines. Although accurate, fixed-point DNNs are generally memory intensive, where excessive access to off-chip memory becomes a design bottleneck. To alleviate this problem, authors of [26, 31] propose to perform inference solely using the on-chip memory and utilizing streaming buffers to realize inter-layer data transfers. These frameworks facilitate the design process of DNNs by providing configurable template functions in high-level synthesis language. However, [26, 31] are only compatible with binary DNNs and do not support fixed-point arithmetics. By incorporating activation encoding into DNN computational flow, EncoDeep hardware simultaneously enjoys the benefits of on-chip streaming buffers and high accuracy arithmetics. EncoDeep hardware stack supports flexible bitwidths, allowing the implementation of customized encoded DNNs.

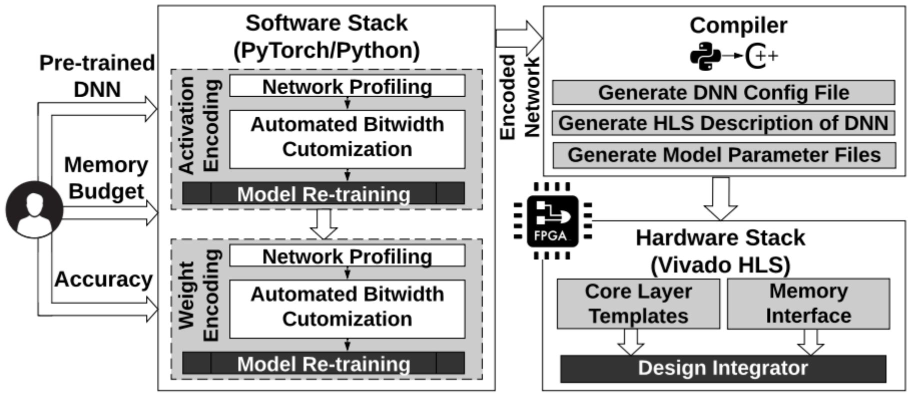


Figure 3.6: The global flow of EncoDeep framework. User provides a high-level Python description of a pre-trained DNN to the software stack, which is responsible for weight/activation encoding, layer-specific bitwidth configuration, and model fine-tuning. Our compiler converts the Python code into a hardware description. The hardware stack then uses a customized library for FPGA synthesis.

3.2.1 Global Flow

Figure 3.6 depicts the global flow of EncoDeep framework. EncoDeep is composed of three interlinked design units, namely the *Software Stack* (also referred to as the *Encoding Engine*), the *Compiler*, and the *Hardware Stack*. EncoDeep aims at alleviating the complications of DNN implementation on FPGAs by incorporating an automated design stack that separates users from the details of hardware design and optimization. We implement an end-to-end automated framework that eliminates all hand-optimizations and delivers a customized accelerator implementation for various DNN architectures and FPGA platforms.

EncoDeep leverages a novel and fully automated learning algorithm to output a maximally efficient DNN architecture in terms of memory footprint while adhering to the accuracy constraints provided by the user. The key insight of EncoDeep is capturing the trade-off between the classification accuracy and memory footprint of model parameters and feature maps (activations). We use the popular neural network development API, PyTorch, to describe the DNNs in the software stack. To implement the inference engines on FPGA, we choose Vivado High-Level Synthesis (HLS) which enables faster development as well as portability. To fill the bridge between the software and the hardware stacks, we develop a compiler unit in Python. Below, we elaborate more on the incorporated design units.

Software Stack. The software stack is responsible for weight/activation encoding, layer-specific bitwidth configuration, and model fine-tuning. This step analyses the input DNN and applies nonlinear quantization (encoding) to layer weights/activations. EncoDeep encoding scheme reduces memory footprint at the cost of a small reduction in inference accuracy. We devise an automated algorithm to determine the number of encoding bins in each layer for the weights and activations such that the memory footprint is maximally reduced and/or the accuracy is minimally affected. The following steps are performed sequentially in the software stack:

- **Activation Encoding.** This step takes as input a pre-trained DNN described in Pytorch format and generates a network with encoded activations.
- **Weight Encoding.** This step takes the DNN from the activation encoding step as input and generates a network with encoded weights and activations.

Both the activation and weight encoding steps consist of three tasks: (i) network profiling where the accuracy-memory trade-off is captured by calculating the correlation between accuracy loss and memory footprint reduction (Section 3.3.3). (ii) Bitwidth selection where the bitwidth of encoded activations/weights is customized based on the user-defined accuracy/memory budget primitives (Section 3.3.3). The encoded activations/weights are then inserted in the DNN graph to replace the full-precision values (Section 3.3.1). (iii) Model re-training where the encoded DNN is fine-tuned to improve classification accuracy (Section 3.3.2). The output of the EncoDeep software stack is an encoded architecture and the trained encoded network's parameters.

Compiler. To ensure ease-of-use and design automation, we design a customized compiler. This unit takes as input the high-level DNN graph description in PyTorch format and converts it to C++ code (as used in the Vivado HLS tool). EncoDeep compiler produces a configuration file that specifies the customized encoding bitwidths for the weights and feature maps of different DNN layers. The network description in C++ together with the configuration file enable instantiation of core layer template modules. The compiler further converts the trained encoded network's parameter into a format ready to be loaded to the on-chip memory of the FPGA upon execution.

Hardware Stack. The hardware description of the DNN is rendered using Vivado_HLS, which is a standard high-level-synthesis tool that enables faster development as well as portability. EncoDeep accelerator enjoys full-precision calculations while maintaining low memory footprint using the encoded values. We provide a library of template modules that can realize different DNN functionalities. An arbitrary architecture can be described by instantiating the corresponding core layer templates in a network description file. Each template module has customized configurable primitives such as the number of input/output neurons of the layer, the bitwidth of the weights/activations, and the parallelism factors for execution. The output of the hardware stack is a bitfile that can be used to efficiently execute the desired DNN on the FPGA. We will elaborate more on the hardware in Section 3.4.

3.3 EncoDeep Software Stack

In this section, we elaborate on the utilized concepts for non-linear encoding of DNN parameters/activations. Section 3.3.1 explains EncoDeep weight/activation encoding. Our gradient computation for encoded network training is formulated in Section 3.3.2. Finally, our automated bitwidth selection routine is explained in Section 3.3.3.

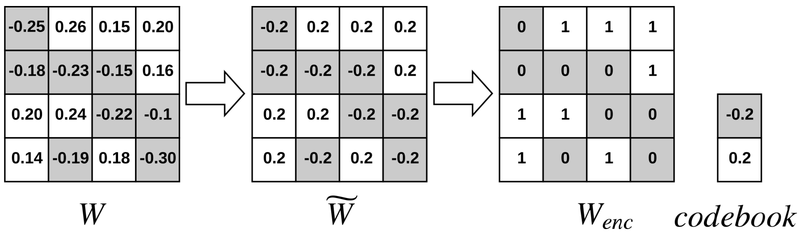


Figure 3.7: Illustration of EncoDeep weight encoding. left: original matrix W, middle: approximated matrix $\widetilde{W}$, right: encoded matrix W_{enc} along with the codebook.

3.3.1 Encoding Scheme

Our encoding scheme aims to estimate the parameters of a DNN layer with a subset of representatives, i.e., the *codebook*. In the rest of this section, we delineate EncoDeep encoding method for DNN weights/activations.

Weight Encoding.

Let us denote the weight parameters in a certain DNN layer as W. In order to encode W, we first find an approximation $\widetilde{W} \approx W$ such that the elements of $\widetilde{W}$ are restricted to a finite set of real-values, $\vec{c} = \{c[1], \ldots, c[K]\}$, i.e., the *codebook*. The encoded weight matrix is then constructed by replacing all elements with indices of the corresponding codebook values. We denote the encoded $\widetilde{W}$ as W_{enc}. Figure 3.7 illustrates this approximation for a 4×4 matrix W using a codebook of $K = 2$ elements.

To approximate $\widetilde{W}$, we use the well-known K-means clustering [34]. While K-means can effectively solve the aforementioned problem for a fixed codebook size, specifying the codebook sizes in different layers of a network is a challenge yet to be solved. Specifically, different layers require different codebook sizes to capture the statistical properties of their parameters. To tackle this, EncoDeep proposes an automated bitwidth selection algorithm explained in Section 3.3.3. Note that weight encoding is performed only once in an offline pre-processing step. The per-layer encoded weights and codebooks are then stored in binary files to be loaded in the FPGA memory.

Activation Encoding. EncoDeep activation encoding is performed in two phases: (i) offline phase performed in the software stack, where the layer codebooks are generated using the

K-means algorithm. (ii) Online phase performed during inference where each feature is encoded by its closest codebook value.

Offline Encoding. Algorithm 1 summarizes our methodology for computing DNN activation codebooks. First, a subsampled data set, $\{\vec{x}_n\}_{n=1}^{N}$, is used to generate the layer feature-maps, which we denote by $\vec{y}^{\,l}$. Next, $\vec{y}^{\,l}$ is flattened into an array, $\vec{a}^{\,l}$. For an arbitrary activation function, the K-means clustering is applied on all values of $\vec{a}^{\,l}$. For the especial case of *ReLU* activations, since the `ReLU` non-linearity produces many 0-valued outputs, we only perform K-means on non-zero elements of $\vec{a}^{\,l}$ to reduce the K-means clustering runtime. The $(K^l - 1)$ cluster centers along with the appended 0 value form the codebook for the l^{th} layer.

Using a subsampled dataset for finding the cluster centers enables for a fast and efficient search over the space of possible codebook sizes, i.e., encoding bitwidths (see Section 3.3.3). To ensure that the obtained cluster values are truly compatible with the distribution of layer feature-maps, we later fine-tune the cluster center values via customized gradient operations explained in Section 3.3.2.

Online Encoding. Online encoding is performed during FPGA execution. The value of a feature y is compared with the elements of the corresponding layer's codebook $\vec{c}_{act}^{\,l}$ to compute the encoding as $y_{enc} = argmin(|y - \vec{c}_{act}^{\,l}|)$. This is implemented by a linear search on a small memory block containing the codebook values (Section 3.4.1).

3.3.2 Training of Encoded Networks

Encoding weights/parameters often results in a drop in accuracy. To compensate for such accuracy loss, the codebook entries are fine-tuned after encoding using a customized back-propagation scheme. In this section, we explain the details for fine-tuning encoded neural networks via Stochastic Gradient Descent (SGD) [75]. For weights, the averaged gradient method [34, 37] is applied. For activations, we develop new gradient computation methods.

Algorithm 1 Offline Activation Encoding

```
Inputs:
    input samples: ${\vec{x}_n}_{n=1}^N$
    per-layer codebook sizes: ${K^l}_{l=1}^L$
Output:
    per-layer codebooks for activations: ${\vec{c}_{act}^{\,l}}_{l=1}^L$
1: for $l$ = 1, ..., $L$ do
2:     $\vec{y}^{\,l} \leftarrow DNN^l(\{\vec{x}_n\}_{n=1}^N)$
3:     $\vec{a}^{\,l} \leftarrow flatten(\vec{y}^{\,l})$
4:     $\vec{a}^{\,l} \leftarrow nonZeros(\vec{a}^{\,l})$
5:     $\vec{c}_{act}^{\,l} \leftarrow KMeans(\vec{a}^{\,l}, K^l - 1)$
6:     $\vec{c}_{act}^{\,l} \leftarrow \{0, \vec{c}_{act}^{\,l}\}$
7: end for
8: return $\{\vec{c}_{act}^{\,l}\}_{l=1}^L$
```

Feature encoding can be viewed as a non-linear transformation, $f(y) = y^*$, where y^* and y represent the approximated and original values, respectively. As depicted in Figure 3.8, the non-linear encoding function is made up of multiple *step* functions, rendering it non-differentiable. Given the gradient of the loss function with respect to the encoded values, $\nabla_{y^*} = \frac{\partial \mathscr{L}}{\partial y^*}$, we aim to compute the partial derivatives with respect to the non-encoded values ($\nabla_y = \frac{\partial \mathscr{L}}{\partial y}$) and the derivatives with respect to the codebook ($\vec{\nabla}_c = \frac{\partial \mathscr{L}}{\partial c}$).

Computing $\nabla_\mathbf{y}$. Given the partial derivative ∇_{y^*}, the gradient ∇_y can be obtained by applying the chain rule:

$$\nabla_y = \frac{\partial \mathscr{L}}{\partial y} = \frac{\partial \mathscr{L}}{\partial y^*} \times \frac{\partial y^*}{\partial y}. \tag{3.1}$$

This formulation, however, is not stable since the function $f(\cdot)$ is non-differentiable. To address this issue, we propose to approximate the derivative of $f(\cdot)$ as:

$$\frac{\partial f(y)}{\partial y} = \begin{cases} 1 & if\ c[1] < y < c[K] \\ 0 & otherwise \end{cases}, \tag{3.2}$$

where $c[1]$ and $c[K]$ are the smallest and largest codebook values, respectively. During forward propagation, y^* is computed as shown in Figure 3.8-left, whereas the backward propagation

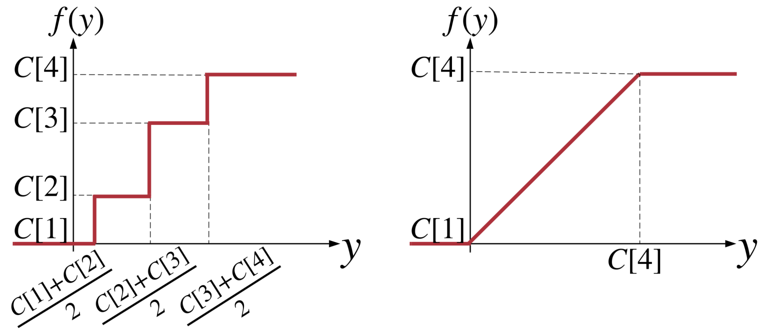


Figure 3.8: Example encoding non-linearity with a codebook of $K = 4$ elements. (Left) Non-linear function applied in the forward propagation. (Right) Smooth approximation of encoding used in backward propagation for gradient computation.

assumes the smooth function in Figure 3.8-right.

Computing $\nabla_\mathbf{c}$. Given a scalar gradient element ∇_{y^*}, the gradient with respect to $c[k]$ is computed as:

$$\vec{\nabla}_c[k] = I(c[k], y^*) \times \nabla_{y^*}, \tag{3.3}$$

with $I(a,b) = 1$ if $a = b$ and zero otherwise (identity operator). Given a vector of features $\vec{y}^*$ and the corresponding vector of gradients $\vec{\nabla}_{y^*}$, the derivative is:

$$\vec{\nabla}_c[k] = \sum_j I(c[k], \vec{y}^*[j]) \times \vec{\nabla}_{y^*}[j]. \tag{3.4}$$

Using the partial derivatives, standard back-propagation algorithms can fine-tune DNN parameters. We incorporate the customized gradient computation routines into EncoDeep software stack to support fine-tuning for encoded DNNs. As shown in the evaluations, the fine-tuning incurs negligible overhead compared to original training.

3.3.3 Automated Bitwidth Selection

Modern DNNs are composed of many layers with high dimensional input/output parameter space. In order to successfully reduce the memory footprint of such networks while minimally affecting the classification accuracy, one is required to customize the memory compression rate on a per-layer basis. EncoDeep automated bitwidth selection aims to adjust the encoding bitwidth (determined by the codebook size) for each layer such that the network's overall memory footprint is minimized while adhering to the user-provided accuracy constraint. To this end, an efficient algorithm is desired that can search the space of possible bitwidth configurations for the optimal solution.

Recent advances in Reinforcement Learning (RL) provide a powerful automated tool for effective search. In high-level, RL approaches traverse a series of states s by taking subsequent actions $a = \pi(s)$ based on a policy function $\pi(\cdot)$. Here, $\pi(\cdot)$ corresponds to a probability distribution over actions given states. Applying an action a at state s triggers a state transition $s \rightarrow s'$ and results in a reward $r(a,s)$ from the environment. In the training phase of RL, the goal is to find a series of actions that return the best discounted sum of future rewards. This is achieved by tuning the policy $\pi(\cdot)$ to incorporate the *long-term* return (reward) in the action-selection process. During RL training, the policy model is learned over a series of episodes $\{E_1, E_2, \dots\}$; each episode consists of all transitions form the initial state to the final state, given the policy $\pi(\cdot)$:

$$E_i : s_1 \xrightarrow{\pi_i(s_1)} s_2 \xrightarrow{\pi_i(s_2)} \dots \xrightarrow{\pi_i(s_{N-1})} s_N.$$

Training the RL policy can generally be time-consuming as it requires many training episodes and evaluations. To overcome this challenge, we propose an algorithm *inspired* by RL that does not learn probabilistic policies and relies solely on immediate rewards. Our method comprises only *one* episode where the path from the initial state to the end state is traversed *deterministically* by choosing greedy actions:

$$E_{greedy} : s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_N.$$

In this context, greedy actions are those with the maximum *immediate* reward when transitioning from state s_i to the next state s_{i+1}. By incorporating immediate rewards, EncoDeep can solve the multi-objective optimization problem at hand in a fraction of pure RL optimization time. Similar to RL, we define a state-action-reward system where the *state* is the encoding bitwidth in DNN layers at the current iteration of the algorithm. In the beginning, all layers are encoded with a maximum bitwidth (e.g, 4 bits for a 16-element codebook). The action-space for each state s, corresponds to all permissible actions that can be taken from that state. Each *action* chooses a layer l and reduces the corresponding bitwidth b_l to $b \in \{1, 2, \ldots, b_l - 1\}$. The action-space thus encloses all bitwidth configurations where only a single layer's bitwidth differs from that in state s. The *reward* for each action is formulated as follows:

$$r(a,s) = \frac{mem(s) - mem(s^{'})}{acc(s) - acc(s^{'})}, \tag{3.5}$$

where a is the action, s is the current state, $s^{'}$ is the state after taking action a, $mem(\cdot)$ and $acc(\cdot)$ denote the total memory footprint and accuracy at a given state, respectively. The accuracy is computed by evaluating the (encoded) network on a validation dataset and the total memory for encoded weights is formulated as:

$$mem(weights) = \sum_{l=1}^{L} size(W^l) \times Log_2(K^l) + K^l \times b_{fix}, \tag{3.6}$$

where the $size(\cdot)$ operator returns the number of elements, K^l is the weight codebook size corresponding to the l-th layer, and b_{fix} is the fixed-point bitwidth of each codebook element. With Y^l being the output feature map of the l-th layer, the total memory footprint for the activations of the neural network is computed as:

$$mem(acts) = \sum_{l=1}^{L} size(Y^l) \times Log_2(K^l) + K^l \times b_{fix}, \tag{3.7}$$

Taking an action in a given step will decrease both memory footprint and accuracy. Hence, the reward function is always positive. At each state, all actions in the action-space are evaluated and the one with maximum reward is chosen. Such a greedy approach is particularly beneficial for the problem statement at hand, i.e., bitwidth configuration, as the value of each state transition can be independently evaluated without relying on the end state and the long-term return. Compared to pure RL, which includes many episodes of policy training, our greedy approach renders drastically lower computation time. Moreover, EncoDeep is able to extract the memory-accuracy Pareto curve with only one state traversal (episode). Rather, in conventional RL settings, policy training must be repeated once for each target memory, further increasing runtime.

Algorithm 2 Automated Bitwidth Customization

```
Inputs:
    maximum bitwidth: $B$
    minimum accuracy threshold: $\theta$
    DNN model: $D$
Output:
    list of bitwidth configurations $\{cfg_1, cfg_2, \dots\}$ that render the optimal accuracy-memory
tradeoff.
 1: $cfg \leftarrow \{b_1 = B, \dots, b_L = B\}$
 2: $AllConfigs \leftarrow \{cfg\}$
 3: $A = Acc(D|b_1, \dots, b_L)$
 4: $M = Mem(D|b_1, \dots, b_L)$
 5: while $A > \theta$ do
 6:     for $l = 1, \dots, L$ do
 7:         for $b = 1, \dots, b_l - 1$ do
 8:             $A(l,b) = Acc(D|\{b_1, \dots, b_l = b, \dots, b_L\})$
 9:             $M(l,b) = Mem(D|\{b_1, \dots, b_l = b, \dots, b_L\})$
10:             $reward(l,b) = \frac{M - M(l,b)}{A - A(l,b)}$
11:         end for
12:     end for
13:     $\{l, b\} \leftarrow \arg\max_{l,b} reward(l,b)$
14:     $cfg \leftarrow \{b_1, \dots, b_l = b, \dots, b_L\}$
15:     $A = Acc(D|cfg\})$
16:     $M = Mem(D|cfg\})$
17:     $AllConfigs \leftarrow \{AllConfigs, cfg\}$
18: end while
19: return $AllConfigs$
```

We visualize EncoDeep search method in Figure 3.9. Here, we use an example 2-layer neural network that is initialized to 3-bit encodings for both layers. At state 0, there are 4 possible actions, each of which has a certain reward that can be computed using Equation 3.5. At this state, the second action renders the maximum reward and therefore the next state's encoding bitwidths are selected as 3 for the first layer and 1 for the second. This process continues until either the accuracy drops below the user-defined threshold or all layers are encoded with 1-bit values.

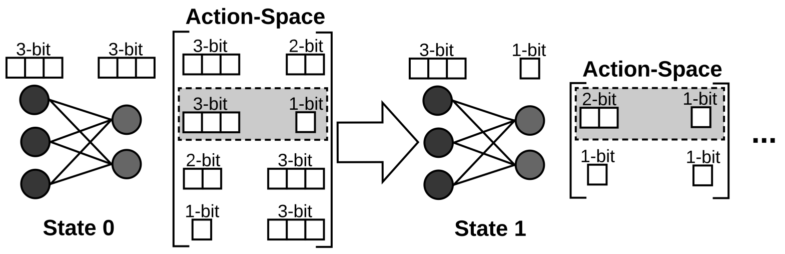


Figure 3.9: Automated bitwidth selection for a 2-layer perceptron.

Note that, after choosing the optimal action at each step of the algorithm, all actions resulting in a memory footprint higher than the new state are eliminated from the search space. This enables a diminishing search cost per iteration of the bitwidth selection method. We also emphasize that the iterative bitwidth selection algorithm does not perform any re-training of the DNN in between the steps. As a result, the (offline) computational overhead of bitwidth customization is drastically smaller than that of RL techniques.

The pseudo-code for the EncoDeep automated bitwidth customization is presented in Algorithm 2. The inputs are the starting encoding bitwidth B for all layers, a minimum threshold θ for the classification accuracy, and a pre-trained DNN model D. The algorithm then gradually decreases the bitwidths, one layer at a time. The algorithm outputs a set of configurations that specify per-layer bitwidths. These configurations capture the tradeoff between memory and accuracy: the first configuration has the highest memory and accuracy whereas the last configuration has the lowest.

3.4 EncoDeep Hardware Stack

EncoDeep inference kernel adopts a streaming-based architecture that facilitates pipelining and overlays the computational overhead of subsequent layers to increase overall throughput and minimize latency. Figure 3.10 presents such pipelined execution for a 3-layer DNN.

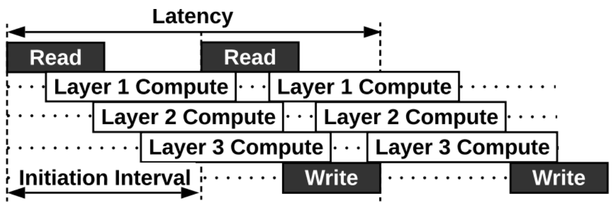


Figure 3.10: Pipelined execution of layer computations in a streaming-based architecture increases throughput.

Our accelerator is specifically designed to accommodate low-bitwidth encoded networks while supporting full-precision computations. Figure 3.11 compares the computational flow of EncoDeep with conventional fixed-point accelerators. In the conventional design (top), a convolution (`CONV`) or Fully-Connected (`FC`) layer receives the inputs and weight parameters in fixed-point format. Each layer starts the computation as soon as its preceding layer starts generating output. The streaming buffers and the weight memory should thus accommodate high bitwidth full-precision values (e.g., 32 bits in Figure 3.11). In practice, due to the low capacity of the on-chip memory in off-the-shelf FPGA platforms and the high number of parameters/features in state-of-the-art DNNs, it is not feasible to accommodate all weights and/or streaming buffers inside the FPGA. In response to this issue, we propose the encoded DNN data flow presented in the bottom schematic of Figure 3.11. Here, the weights are stored in the encoded format to save memory. The computed outputs of each layer are also encoded before being sent through the streaming buffer to enable use of low-capacity buffers. The `CONV` and `FC` layers of the DNN are therefore equipped with encoder and decoder modules.

EncoDeep is equipped with a hardware library described in high-level synthesis language that allows FPGA implementation of encoded DNNs. Our hardware library consists of the

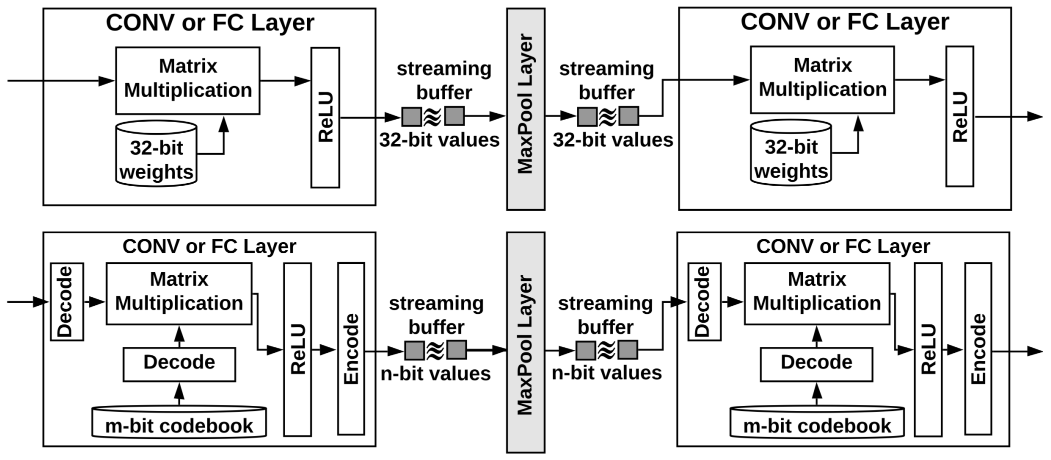


Figure 3.11: Computational flow of a conventional DNN (top) and our proposed DNN with encoded weights and activations (bottom).

essential building blocks to implement encoded DNN layers (e.g, convolution, max-pooling, etc.). Each layer-type is implemented as a template function with certain computing engines that are customized to the specifications of the pertinent layer such as the input/output dimensions. By means of this tailoring, EncoDeep exploits the benefits of FPGA reconfigurability and delivers a bit-flexible design.

To ensure portability and efficiency, we chose an open-source framework [31], i.e., FINN, from Xilinx as the base of our hardware accelerator. The FINN library was originally intended for the execution of binary neural networks and cannot be used for encoded DNNs as is. We thus extend the library to support customized streaming buffers that can accommodate flexible bitwidths rather than binary values across layers. We further design a data scheduling unit, dubbed the Sliding Window Unit, that reorders and populates layer input buffers in accordance with the underlying bitwidth. We implement new processing engines that support operations on encoded parameters/weights and fixed-point Multiply-Accumulate (MAC) operations, which replace the *XnorPopCount* operations required in BNNs [31]. Using our proposed encoding scheme, EncoDeep enjoys the benefits of on-chip buffering and high-precision MACs.

Figure 3.12 depicts the flow diagram of the EncoDeep accelerator for implementing an encoded DNN on FPGA. The Sliding Window Unit (SWU) reorders the convolution layer input feature-maps to generate appropriate streaming buffers for the Matrix-Vector-Activation Unit (`MVAU`). The `MVAU` is the core computational module of `CONV` and `FC` layers which performs the matrix-vector multiplication, activation, and batch normalization. The Max-Pooling Unit (MPU) performs max-pooling over the feature-maps. In the following, we discuss the core modules in EncoDeep Hardware.

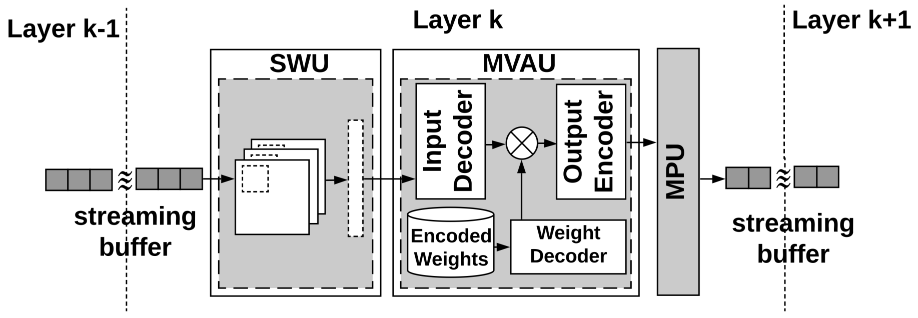


Figure 3.12: EncoDeep accelerator schematic for encoded DNN inference. SWU reorders the input buffer; `MVAU` and MPU perform core computations and max-pooling, respectively.

3.4.1 Matrix-Vector-Activation Unit (`MVAU`)

The `MVAU` in EncoDeep hardware library is instantiated in convolution (`CONV`) and fully-connected (`FC`) layers to generate output features using the corresponding layer's specifications. Figure 3.13 illustrates the `MVAU` computational flow. This module performs three core tasks required in state-of-the-art DNNs, namely matrix-vector multiplication, batch normalization, and applying non-linear activation. Internally, the `MVAU` is composed of an array of Processing Engines (`PE`s) which accept a shared lane of `SIMD` inputs in parallel. In addition, EncoDeep `MVAU` has customized encoding/decoding cores for processing the outputs, inputs, and weights.

Matrix-Vector Multiplication. The main operations performed in linear DNN layers can be represented as a series of matrix-vector multiplications. The matrix-vector multiplication core in EncoDeep `MVAU` offers two levels of parallelism to facilitate throughput control across

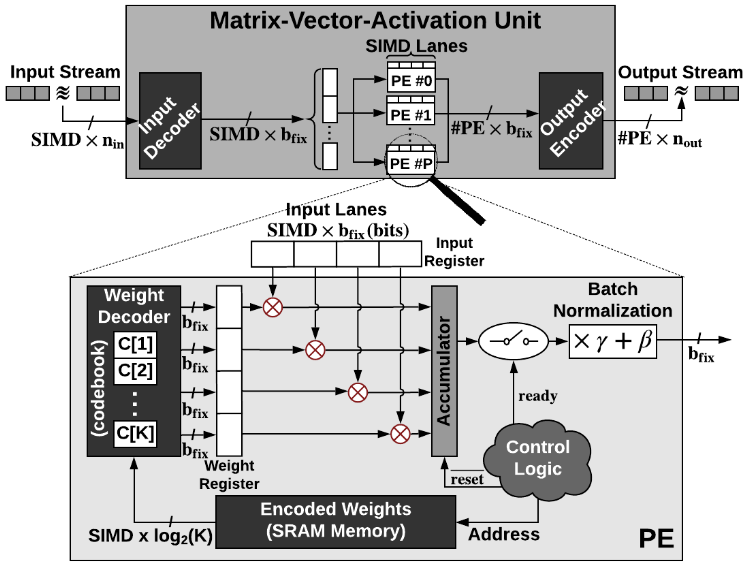


Figure 3.13: **(Top)** Computational flow of EncoDeep MVAU. This unit performs matrix-vector multiplication, batch normalization, and a non-linear activation. To increase throughput, the core computations in the MVAU are distributed across parallel PEs. The MVAU is further equipped with an input decoder, an output encoder, and several weight decoders (one per-PE) to comply with EncoDeep encoded DNNs. **(Bottom)** Internal configuration of a PE. Each PE performs parallel MAC operations over SIMD operands.

DNN layers.

- Layer output generation is distributed among several PEs working in parallel. In this setting, each PE is responsible for generating the output of multiple feature-map channels (neurons) in a CONV (FC) layer. For instance, in a CONV layer with 64 output channels and 16 PEs, each PE is responsible for computing 4 output channels.

- Each PE operates in single-instruction-multiple-data (SIMD) mode: MACs in a PE are parallelized across SIMD lanes.

The MAC operations in EncoDeep are performed in fixed-point on decoded input/weight values, implemented using DSP slices. The per-layer encoded weight matrix is stored in the on-chip memory of the FPGA and is partitioned among all PEs within the pertinent layer. This partitioning allows all PEs to simultaneously access their share of weights. Note that the computations performed inside each PE are independent of those performed in the neighbor PEs. Thus, there is no need for inter-PE communication.

Decoder Modules. The encoded features/weights of EncoDeep are converted into the equivalent fixed-point format before being used in matrix-vector multiplication. Each layer in the encoded DNN contains two decoding codebooks corresponding to the inputs (activations) and weights. This functionality is implemented by a memory containing all cluster centers (codebook values) stored in fixed-point format (e.g, 32 bits). For an encoded value $y_{enc} \in \{1,\ldots,K\}$ the corresponding fixed-point approximation y^* can be obtained by feeding y_{enc} as the address of a memory block storing the cluster centers $\{c_1,\ldots,c_K\}$. Note that the cluster centers incur a negligible memory footprint since K is small.

The decoder modules are implemented via register files, rather than SRAM blocks. This design choice allows for simultaneous decoding of SIMD inputs, in parallel. To achieve maximum efficiency, the input decoder is implemented inside the MVAU. This enables us to share the decoded inputs among all PEs within one MVAU. Unlike inputs which are shared among PEs, the weights for each PE are different. Therefore, to facilitate parallelism, each PE owns a copy of the corresponding weight decoder (codebook). Upon execution of multiply-accumulate operations, the replicated codebooks can be accessed in parallel to decode weights. The replication of weight codebook across PEs incurs a negligible memory overhead which is a reasonable cost for the throughput and performance gains obtained.

Equation 3.8 shows the overall weight memory saving for an arbitrary DNN layer, after wight encoding. Here, N denotes the number of weights, b_{fix} is the number of bits used to represent the fixed-point values, and K is the codebook size. As shown in the experiments (Section 3.5), K takes a value equal to or smaller than 64 while N is in the order of 10^6. As such, $(b_{fix} \times K \times PE) \ll (b_{fix} \times N)$ and the denominator in Equation 3.8 remains smaller than the numerator.

$$\frac{memory(W)}{memory(\vec{c}) + memory(W_{enc})} = \frac{b_{fix} \times N}{b_{fix} \times K \times PE + Log_2(K) \times N} \tag{3.8}$$

Encoder Module. The encoding module compares the distance of each computed feature y to all elements of the codebook (i.e., $|y - c[1]|, \ldots, |y - c[K]|$) and outputs the index of the closest element as the encoded value:

$$y_{enc} = \underset{i}{argmin}\ |y - c[i]|\ \ ,\ \ i \in \{1, \ldots, K\} \tag{3.9}$$

The encoded value is then sent through the output streaming buffer to be processed by the next layer. Most contemporary DNNs use `ReLU` activation. The encoder module inherently implements `ReLU` functionality when the first codebook value is set to 0. To implement other activation functions, we keep the encoding functionality of Equation 3.9 and merge the activation function into the next layer's input decoder. In other words, the input decoder of the next layer stores $Act(c[i])$ rather than $c[i]$ with $Act(\cdot)$ being the desired activation function. This modification is performed offline when the codebook values are loaded to FPGA memory.

Processing Element (`PE`). Figure 3.13-bottom shows the hardware architecture of a `PE` inside the `MVAU`. Each `PE` is responsible for performing `MAC` operations on `SIMD` parallel input lanes. To this end, each `PE` is equipped with `SIMD`×`MULT` units implemented using DSP slices. Each `MULT` performs one fixed-point multiplication on b_{fix}-bit values. The multiplication results are then accumulated in a register (Accumulator in Figure 3.13) in fixed-point format. The decoded inputs required for performing `MAC` are registered in the `MVAU` and provided to all `PE`s. Alternatively, the decoded weights are generated within each `PE`. The low-bit (encoded) weights are stored in an `SRAM` block, which is partitioned to allow `SIMD` parallel read operations via the weight decoder. To perform decoding, each `PE` comprises a weight decoder (codebook) that is implemented using a register file. Once decoded, the weights are written into `SIMD` registers for parallel MULT. For each PE, the size of the local memory is:

$$Memory_{PE} \approx \underbrace{(K + 2\texttt{SIMD}) \times b_{fix}}_{register} + \overbrace{N \times log_2(K)}^{\texttt{SRAM}\ block} \tag{3.10}$$

where N is the number of encoded weights in the SRAM block of each PE and K is the weight codebook size.

At each point of the computation, a control logic keeps track of the matrix-multiplication indices and generates address signals to the encoded weights memory block accordingly. Whenever computations of one neuron are finished, i.e., when one vector-dot-product is completed, the control logic resets the accumulator and activates batch normalization on the computed output. Applying batch normalization to the output of the accumulator y is equivalent to computing $y \leftarrow \gamma y + \beta$. Here, γ and β represent the scaling factor and bias, respectively, which are constants learned during DNN training. These values are extracted by the EncoDeep compiler from the trained encoded DNN and stored (in fixed-point format) on registers within each PE. Note that the memory requirement of these parameters is drastically lower than that of the weight matrices; thus, EncoDeep stores these parameters in the raw (non-quantized) format. After batch normalization, the output is ready to be encoded and sent to the next DNN layer through the streaming buffer.

3.4.2 Sliding Window Unit (SWU)

The convolutional layers of a DNN compute the dot product between a window of the layer input and the CONV weight kernel. The window slides over the input image to produce individual elements of the output feature-map. The SWU in EncoDeep hardware simulates the sliding window operation by reordering the values in the layer input image buffer. The input image values are then grouped in chunks of SIMD words to be sent to the MVAU sequentially for processing.

3.4.3 Max-pooling Unit (MPU)

EncoDeep software stack outputs a sorted list of codebook values for the output encoding: higher values are mapped to larger encodings. This sorting is particularly useful since comparison over encoded values becomes equivalent to comparison over the original fixed-point values; therefore, EncoDeep performs the max-pooling operation on low-bitwidth encoded values rather than the full-precision cluster centers. This approach provides two benefits: (i) the memory overhead of the buffers in the MPU is considerably reduced. (ii) The logic cost of comparison between low-bitwidth encoded values is significantly smaller than the full-precision counterpart.

3.5 Experiments

To evaluate EncoDeep effectiveness, we perform proof-of-concept experiments on four different classification benchmarks, namely, MNIST, CIFAR-10, SVHN, and ImageNet. Table 3.1 summarizes the DNN architectures used in our evaluations. EncoDeep software stack is implemented in Pytorch and the hardware stack is realized in Vivado_HLS design suite. All hardware resource utilizations are gathered after performing place-and-route via Vivado Design Suite 2017.2. Throughput values are reported from Vivado_HLS 2017.2.

3.5.1 EncoDeep Automated Bitwidth Selection

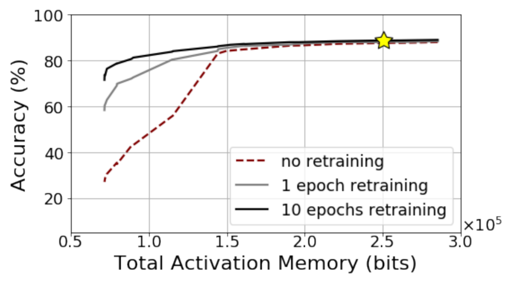
We showcase our bitwidth selection algorithm using the *VGG7* architecture trained on CIFAR-10 dataset. Our customization algorithm provides a set of configurations, each of which renders a certain memory footprint and accuracy. The first step of EncoDeep bitwidth customization is to encode the activations while the weights are kept at full-precision. Initially, the activations are encoded with 4 and 6 bits in `CONV` and `FC` layers, respectively. We then utilize our customization algorithm in Section 3.3.3 to extract the activation bitwidths. As the algorithm proceeds, both total memory footprint and DNN accuracy are decreased. The obtained accuracy/memory trade-off is shown in Figure 3.14-a.

Table 3.1: Benchmarked DNNs for evaluating EncoDeep effectiveness. CONV layers are represented as $\langle input-channels\rangle \xrightarrow[stride]{\langle kernel\ size\rangle} \langle output-channels\rangle$.

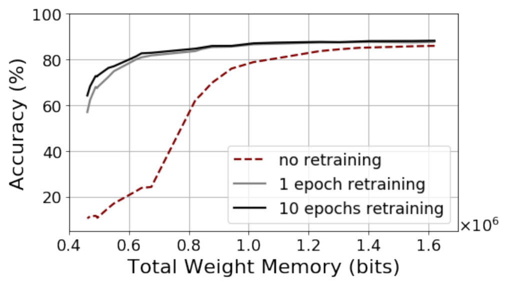
	LeNet [76] (MNIST)	**VGG7 [26] (CIFAR10 & SVHN)**	**AlexNet [77] (ImageNet)**	**ResNet-18 [78] (ImageNet)**
CONV +BN +ReLU	$1 \xrightarrow[\text{stride 1}]{5\times5} 16$	$[3 \xrightarrow[\text{stride 1}]{3\times3} 32]\ \times2$	$3 \xrightarrow[\text{stride 4}]{11\times11} 64$	$3 \xrightarrow[\text{stride 2}]{7\times7} 64$
Pooling	MP (2×2) stride 2	MP (2×2) stride 2	MP (2×2) stride 2	MP (3×3) stride 2
CONV +BN +ReLU	$16 \xrightarrow[\text{stride 1}]{3\times3} 32$	$[32 \xrightarrow[\text{stride 1}]{3\times3} 64]\ \times2$	$64 \xrightarrow[\text{stride 2}]{5\times5} 192$	$[64 \xrightarrow[\text{stride 1}]{3\times3} 64]\ \times4$
Pooling	MP (2×2) stride 2	MP (2×2) stride 2	MP (2×2) stride 2	-
CONV +BN +ReLU	-	$[64 \xrightarrow{3\times3} 128]\ \times2$	$192 \xrightarrow[\text{stride 1}]{3\times3} 384$	$[64 \xrightarrow{3\times3} 128]\ \times4$
CONV +BN +ReLU	-	-	$384 \xrightarrow[\text{stride 1}]{3\times3} 256$	$[128 \xrightarrow{3\times3} 256]\ \times4$
CONV +BN +ReLU	-	-	$256 \xrightarrow[\text{stride 1}]{3\times3} 256$	$[256 \xrightarrow{3\times3} 512]\ \times4$
Pooling	-	-	MP (2×2) stride 2	AP (7×7)
Classifier	FC (256) FC (10) softmax	FC (256) FC (256) FC (10) softmax	FC (2048) FC (4096) FC (1000) softmax	FC (1000) softmax

We use a small portion of the training data[1], dubbed the validation set, to compute the accuracy during the iterative bitwidth customization algorithm. We empirically observed that the accuracy measured on the validation set is correlated with the accuracy measured on the entire test set. Therefore, the validation accuracy can be leveraged as a suitable, low-cost, proxy for test accuracy during bitwidth configuration. The validation set is small enough to be cached into the GPU memory to ensure fast evaluations. Note that we do not retrain the model in between iterations to ensure fast customization. To illustrate the effect of retraining, we also plot the accuracy of each extracted bitwidth configuration after 1 and 10 epochs of fine-tuning in Figure 3.14-a. It can be seen that the accuracy is retrieved for most of the configurations even after 1 epoch, which is a fairly short post processing time compared to the original (floating-point) training which takes ~ 200 epochs.

[1] 1000 samples for all benchmarks.



(a)



(b)

Figure 3.14: Memory and accuracy trade-off for (a) activations and (b) weights of VGG-7 on CIFAR-10 dataset.

In the next step of our customization, one of the configurations for activation bitwidths (the $\star$ mark on Figure 3.14-a) is selected and fine-tuned to recover accuracy. We then proceed to the weight encoding step with initial 6-bit encoding for all layers. During this customization stage, activation bitwidths remain unchanged and only the weight bitwidths are configured. Similar to activation encoding, we obtain the accuracy/memory curves in Figure 3.14-b. In what follows, we evaluate EncoDeep automated bitwidth customization from two perspectives: (i) quality of the end result, i.e., the obtained bitwidth configuration. (ii) Search efficiency/performance of our heuristic algorithm.

Evaluation of EncoDeep Bitwidth Configurations. We apply bitwidth customization to weights and activations of various benchmarked DNNs and select several bitwidth configurations. Table 3.2 compares the total memory (activation+weights) and test accuracy between the original full-precision models and selected encoded DNNs. EncoDeep achieves 14.56× memory reduction with 0.026% accuracy loss for MNIST, 7.34× memory reduction with 0.37% accuracy loss for SVHN, and 6.85× memory reduction with 0.91% accuracy loss for CIFAR-10. On ImageNet, EncoDeep reduces the model size by 7.9× and 6.6× with 0.43% and 0.8% drop in top-1 accuracy for AlexNet and ResNet18, respectively.

To investigate whether the solution found by the heuristic method is the absolute best, one needs to perform brute-force evaluation of all bitwidth configurations. Nevertheless, brute-force

Table 3.2: Comparison of full-precision networks with EncoDeep models with flexible bitwidths across layers.

		Full-Precision (FP32)	**EncoDeep Configurations**		
MNIST	Memory ($\times 10^5$)	50.67	3.48	2.17	1.89
	Test Accuracy (%)	99.28	99.02	98.69	98.31
SVHN	Memory ($\times 10^6$)	12.34	1.67	1.21	0.65
	Test Accuracy (%)	97.67	97.30	97.15	95.07
CIFAR-10	Memory ($\times 10^6$)	12.34	1.80	1.20	1.02
	Test Accuracy (%)	89.05	88.14	87.01	85.06

			Full-Precision (FP32)	**Quantized (INT8)**	**EncoDeep Configurations**	
ImageNet	AlexNet	Memory ($\times 10^8$)	11.14	2.78	1.41	0.53
		Test Accuracy (%)	56.3	55.18	55.87	53.21
	ResNet18	Memory ($\times 10^7$)	44.76	11.19	6.78	4.45
		Test Accuracy (%)	69.70	68.97	68.90	65.40

evaluation is only viable for small networks as the bitwidth search-space grows exponentially in the number of network layers. In particular, for encoding an L-layer network with weight and activation bits in the range of $[1, B_w]$ and $[1, B_a]$, respectively, a total number of $B_w^L \times B_a^L$ evaluations is required. Here, we evaluate the effectiveness of our heuristic on a small-scale bitwidth optimization problem. We perform brute-force search on the activation bitwidths of the VGG7 network for CIFAR-10 dataset and summarize the results in Figure 3.15. For this experiment $L = 8$ and $B_a = 4$, resulting in a total of 2^{16} evaluations (shown with blue points) which takes $\sim$ 18 hours on an NVIDIA TITAN Xp GPU. As can be seen, the obtained activation bitwidths (shown with red points), lie on the memory-accuracy Pareto front, indicating that the heuristic successfully eliminates the non-optimal solutions and finds near-optimal configurations.

Due to the excessive runtime of brute-force search, especially for more complex benchmarks, we provide comparisons with prior art to evaluate the quality of our obtained bitwidths. Table 3.3 summarizes the comparison of EncoDeep configurations with prior methods for training low-bit DNNs in terms of memory, accuracy, and fine-tuning time. Each of our reported architectures and their corresponding bitwidths in Table 3.3 are chosen specifically to match the accuracy

Table 3.3: Comparison of EncoDeep with state-of-the-art low-bit DNNs. Our per-layer bitwidths are shown in Figure 3.16. We normalize the memory footprint of previous works to that of EncoDeep: lower memory and higher accuracy are desirable.

	Baselines	**Architecture**	**Test Accuracy (%)**	**Memory**	**Epochs**	**Bitwidth**	
						Weight	**Act**
MNIST	QNN [29]	MLP	99.04	193×	1000	1	1
	ReBNet3 [26]		98.25	1.76×	200	1	2
	EncoDeep	LeNet-I	99.02	1.84×	$10 \times 2^*$	flexible	
	EncoDeep	LeNet-II	98.31	1	10×2	flexible	
CIFAR-10	QNN [29]†	VGG7	89.85	5.4×	500	1	1
	ReBNet3 [26]‡		86.98	1.65×	200	1	3
	EncoDeep	VGG7-I	88.14	1.32×	10×2	flexible	
	EncoDeep	VGG7-II	87.01	1	10×2	flexible	
SVHN	ReBNet3 [26]‡	VGG7	97.00	1.62×	50	1	3
	QNN [29]‡		97.2	2.15×	200	1	1
	EncoDeep	VGG7-III	97.15	1	10×2	flexible	
ImageNet	ReLeQ [79]	AlexNet	56.82	2.01×	-	flexible	32
	EncoDeep**	AlexNet-I	55.87	1	0.25×2	flexible	
	HWGQ [65]	AlexNet	52.70	1.17×	68	1	2
	HBNN [69]		52.00	2.32×	-	flexible	
	PQTS [80]		51.60	2.32×	-	2	2
	QNN [29]		51.03	1.17×	-	1	2
	DoReFaNet [63]		49.80	1.17×	45	1	2
	WRPN [64]‡		48.30	4.61×	-	1	1
	XNORNet [66]		44.20	1.15×	18	1	1
	ReBNet3 [26]		41.43	1.17×	100	1	3
	EncoDeep**	AlexNet-II	53.21	1	0.25×2	flexible	
	ABC-Net [67]	ResNet18	65.00	1.57×	-	5	5
	ABC-Net [67]		62.50	1.05×	-	3	5
	EncoDeep	ResNet18-I	65.40	1	0.25×2	flexible	

* Fine-tuning for 10 epochs post-activation and 10 epochs post-weight encoding.

† This baseline has 4× more neurons per layer than ours.

‡ This baseline has 2× more neurons per layer than ours.

** Our architecture has 2× less neurons in the output of the first fully-connected layer (see Table 3.1)

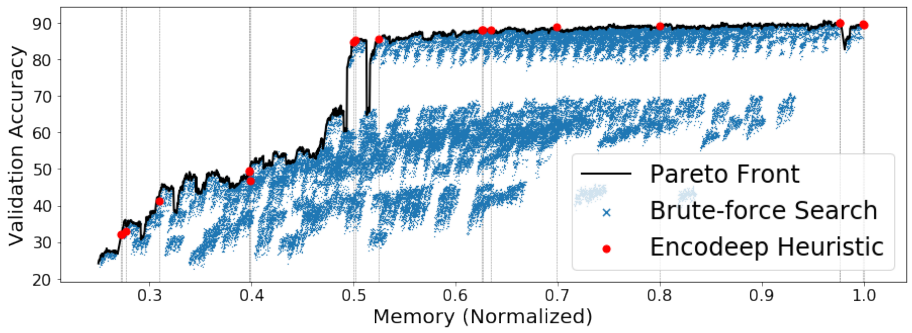


Figure 3.15: Memory-accuracy Pareto curve obtained by brute-force evaluation of bitwidth configurations for VGG7 on CIFAR-10 benchmark. EncoDeep generates near-optimal bitwidths by finding configurations that lie close to the non-dominated Pareto front.

and/or memory footprint of the prior art. We visualize the benchmarked per-layer bitwidths for each evaluated EncoDeep DNN in Figure 3.16. The memory efficiency of EncoDeep can be attributed to the following main reasons:

1. Looking at the bitwidth configurations of Figure 3.16, EncoDeep automatically chooses to have lower bitwidths for weights of fully-connected layers and activations of early convolutional layers. Doing so helps in minimizing the overall memory footprint since such layers have more contribution to the DNN memory.

2. To compensate for the drop in the inference accuracy, prior work in low-bit DNN inference increases the number of neurons/channels per DNN layer [64]. In contrast, we show that by adjusting the arithmetic encoding bitwidth, one can achieve comparable accuracy with even fewer neurons per layer. For our AlexNet benchmark, for instance, we use $2\times$ less neurons in the output of the first fully-connected layer compared to the original architecture (see Table 3.1). EncoDeep reduces the memory of AlexNet benchmark using this method, but preserves the accuracy by using non-linear encoding with flexible bitwidths.

It is worth mentioning that, unlike existing low-bit DNNs that train the whole network from scratch, EncoDeep extracts several near-optimal bitwidth configurations for a pre-trained DNN in one-shot execution. The benefits of this approach are three-fold: (i) EncoDeep eliminates the drastic cost of training from scratch per bitwidth configuration. Using our fine-tuning method explained in Section 3.3.2, model accuracy is retrieved after a few epochs, e.g., as low as 0.25 epochs for ImageNet. (ii) EncoDeep accuracy/memory to be tuned by picking different bitwidth configurations across layers. (iii) EncoDeep customization can be readily applied to publicly available pre-trained models.

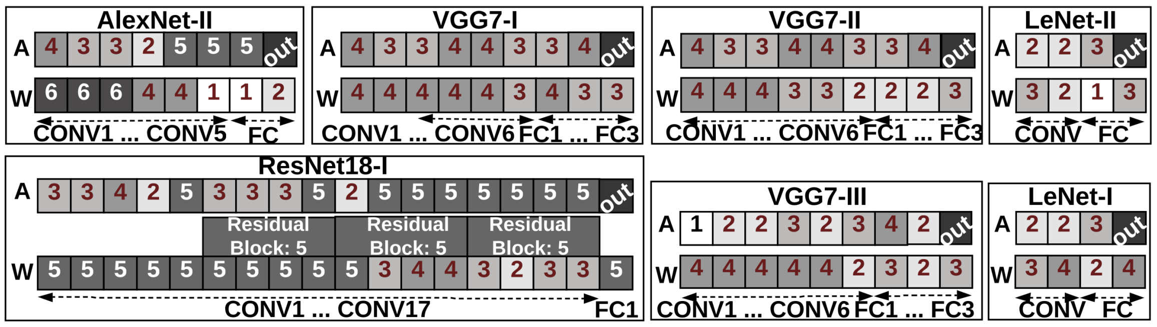


Figure 3.16: Per-layer encoding bits for evaluated DNNs.

Evaluation of EncoDeep Search Algorithm. While various hyperparameter optimization methods, e.g., RL or genetic algorithms, can potentially deliver similar end results, what distinguishes these approaches is the number of evaluations required to obtain the final result. This, in turn, directly affects the algorithm runtime. In this Section, we compare EncoDeep search with existing methods in discrete combinatorial optimization in terms of the quality of end results and the search efficiency (runtime).

Comparison with Q-learning. Recall from Section 3.3.3 that one of the main differences between pure RL-based methods and our search algorithm lies in pursuing reward-based immediate returns in EncoDeep rather than the traditional long-term returns. Our goal in this part of the evaluation is to show how the immediate-reward optimization of EncoDeep compares to pursuing long-term rewards in pure RL. Optimizing a long-term reward can potentially lead to better end results in RL tasks. Due to the nature of our problem, however, an immediate reward is beneficial as the value of each state transition can be independently evaluated without relying on the end state. More specifically, the immediate reward can be leveraged to assess the optimality

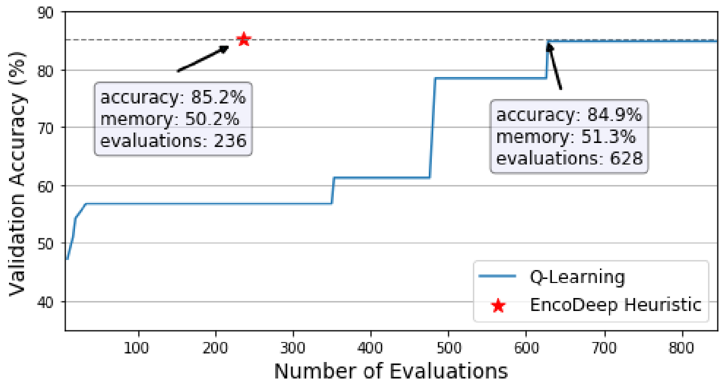


Figure 3.17: Comparison of the acquired accuracy and memory as well as the number of evaluations between a Q-learning approach and EncoDeep (★), upon convergence.

of each intermediate state without the need for traversing all states to the end. For a concrete comparison, we have implemented Q-learning [81] as an RL baseline with long-term rewards for comparison. We set the Q-learning reward for state s as follows:

$$r(s) = \begin{cases} \frac{mem(s)-mem(s_0)}{acc(s)-acc(s_0)} & if \;\; |mem(s) - \theta| < tolerance \\ 0 & otherwise \end{cases} \tag{3.11}$$

where θ is the target memory and s_0 is the initial state with all bitwidths set to B. An episode is finished when a state's memory drops below θ. We apply Q-learning on the activations of the VGG-7 model trained on CIFAR-10. In this experiment, we set θ to 0.5 to achieve $\sim 50\%$ memory reduction compared to the initial DNN. We set the tolerance to 0.05 so that any bitwidth configurations resulting in a memory $\in [45\% - 55\%]$ is given a reward during Q-learning. Figure 3.17 compares EncoDeep with Q-learning. The horizontal axis shows the number of model evaluations, i.e., number of DNN inference accuracy computations. The vertical axis shows the maximum accuracy seen so far for configurations with a memory in the range [0.45-0.55]. The ★ represents the configuration with comparable memory, found by EncoDeep.

Compared to Q-Learning, EncoDeep achieves slightly higher accuracy and lower memory while requiring fewer number of DNN evaluations. This is due to the fact that EncoDeep is policy-free and only consists of one greedy state-transition episode by relying on immediate rewards. Note that for each target memory and accuracy, policy training, as in Q-learning, must be repeated from scratch with a different threshold θ (see Equation 3.11) to extract the optimal bitwidth configuration. In contrast, EncoDeep extracts the entire memory-accuracy Pareto curve with only one state traversal (episode). This provides adaptability by allowing users to pick their desired bitwidth configuration based on various accuracy-memory constraints, without need for re-running the entire algorithm.

Comparison with Genetic Algorithms. Another important baseline for combinatorial optimization is genetic algorithms (GA). Carefully designed GA is shown to deliver similar end results to RL-based methods in various tasks [82]. For comparison, we use the optimization framework in [48], which is a generic tool for compressing DNNs with GA, as a new baseline. Figure 3.18 shows the reward[2] of the genetic population across GA iterations with an accuracy threshold of 85% and a genetic population size of 50. As can be seen, GA gradually and iteratively evolves the configurations to increase the average reward in the population, i.e., lower memory and higher accuracy. The red point on Figure 3.18 corresponds to the best solution found by GA, which takes 1234 evaluations to find a model with 50% of the original memory and 85.6% inference accuracy. For a similar target memory and accuracy, EncoDeep achieves 50.2% memory and 85.2% accuracy via only 236 evaluations. We attribute this improvement in number of evaluations to the single-episode greedy execution of EncoDeep, which is specifically designed to find the optimal configuration with very few iterations.

In summary, EncoDeep has the following benefits compared to genetic algorithms:

(1) The success of GA relies heavily on careful design of the underlying score function used in the evaluation step. Besides, multiple design choices and hyperparameters, e.g., mutation and crossover rates, affect the optimization performance. Hand-tuning such parameters remains a standing challenge that further hinders the GA design process. EncoDeep does not include extra

[2]Please refer to [48] for details about the utilized reward function for the GA.

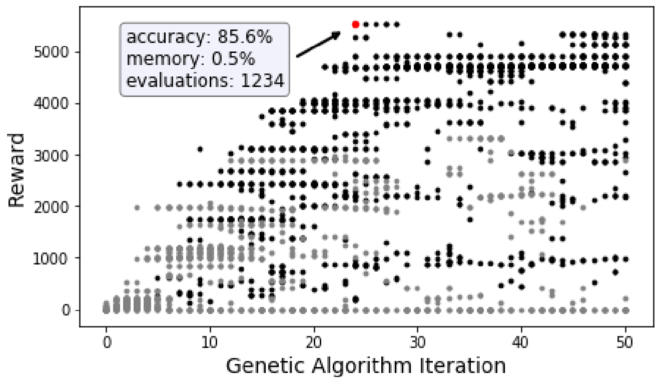


Figure 3.18: Genetic evolution for bitwidth customization of VGG-7 on CIFAR-10. The reward on the vertical axis measures bitwidth optimality by combining accuracy and memory (see [48] for details). Each point on the plot represents an individual in GA. points with black color have an accuracy higher than the accuracy threshold $\theta = 85\%$.

design hyperparameters and allows for easy automation.

(2) To obtain a trade-off between accuracy and memory, one needs to run GA multiple times with different target accuracies in the reward function. Therefore, extracting the memory-accuracy tradeoff using GA incurs a high timing overhead. In contrast, EncoDeep extracts all points lying close to the Pareto front in a single run.

Overhead of Customization and Re-training. We summarize the total runtime of EncoDeep bitwidth configuration and the break-down of different steps for all our benchmarks in Figure 3.19. Runtime values are gathered using a machine with a single NVIDIA Titan Xp GPU and an Intel Xeon-E5 CPU. As seen, the overhead of (offline) clustering at the pre-processing stage (`EN-A` and `EN-W`) is negligible compared to other steps. The bulk of runtime is due to running Algorithm 2 on activations (CU-A) and weights (CU-W), and retraining the customized models (RT-U and RT-W). Nevertheless, EncoDeep takes only $\sim$ 254 minutes to customize our most complex benchmark (ResNet 18), whereas training the original ResNet-18 model on the same machine takes over a day ($\sim$ 1800 minutes). EncoDeep customization incurs only 14% of the training time even for this many-layer network. Note that the evaluations in EncoDeep customization steps (`CU-A` and `CU-W`) can be distributed across multiple GPUs to further decrease runtime.

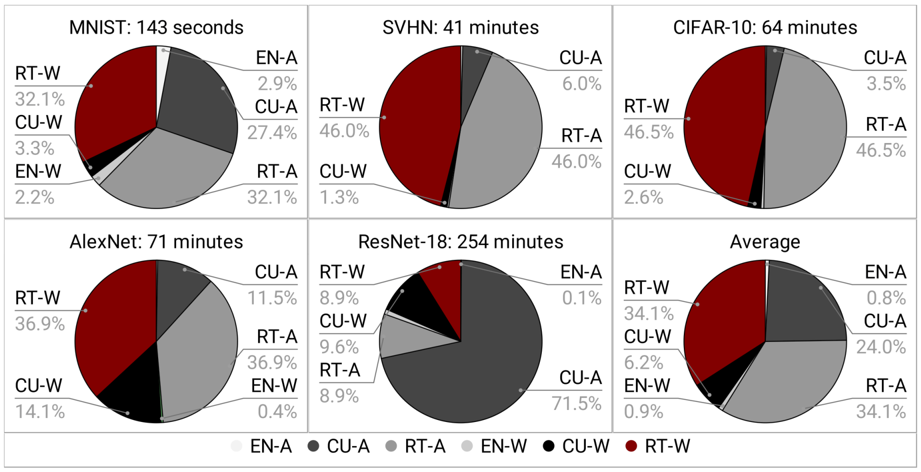


Figure 3.19: Total time and break-down of EncoDeep optimization runtime for the evaluated benchmarks. Here, EN-A and EN-W represent (offline) activation and weight encoding. CU-A and CU-W denote bitwidth customization for activations and weights. RT-A and RT-W correspond to the re-training time after activation and weight encoding.

It is worth mentioning that EncoDeep customization step is much faster than pure reinforcement learning-based approaches. ReLeQ [79] is an example RL-based method that trains an LSTM model using gradient computation while our method does not include any RL model training and is thus much faster in terms of runtime (e.g., ReLeQ has 600 episodes[3] for LeNet while our method requires only a single episode with 22 iterations to achieve similar results).

3.5.2 EncoDeep Hardware Implementation

In this section, we evaluate EncoDeep hardware accelerator. We implement one architecture per dataset from Figure 3.16, namely, LeNet-I for MNIST, VGG7-I for CIFAR10, VGG7-III for SVHN, and AlexNet for ImageNet. Table 3.4 summarizes the evaluation platforms for each DNN architecture.

Importance of Activation Encoding. We start the analysis by studying the advantages of activation encoding, from the hardware perspective, versus solely encoding the weights as

[3]unknown number of evaluations per episode

Table 3.4: Platform details in terms of block-RAM (BRAM), DSP, flip-flop (FF), and look-up table (LUT) resources.

Application	Platform	BRAM	DSP	FF	LUT
ImageNet	Virtex VCU108	3456	768	1075200	537600
CIFAR-10 & SVHN	Zynq ZC702	280	220	106400	53200
MNIST	Spartan XC7S50	120	150	65200	32600

proposed in [34]. Note that [34] also applies pruning and Huffman encoding which are the main contributors to the compression rate. Since these methods are orthogonal to our approach, we do not utilize them in EncoDeep to focus on the analysis of encoding itself. We compare two versions of encoded DNNs: one with encoded weights and fixed-point activations as proposed in [34], and another with both weights and activations encoded as suggested in EncoDeep. For each dataset, we separately optimize the per-layer parallelism factors `SIMD` and `PE` for both encoded and fixed-point DNNs to obtain maximum possible throughput. Table 3.5 summarizes resource utilization and throughput for each of the designs.

Overall, the realization of EncoDeep methodology achieves higher throughput while requiring a lower number of resources compared to a weight-only encoding approach [34]. The benefits become more prominent for architectures with higher complexity since the memory implication of activations is higher in complex networks. As seen for MNIST, CIFAR-10, and SVHN benchmarks, EncoDeep activation encoding improves the throughput by $1.1\times$, $6.2\times$, and $6.66\times$, respectively. Advantages of activation encoding are most significant for AlexNet: model memory with fixed-point activations is so large that it cannot fit in the FPGA block-RAM capacity, rendering the design infeasible within platform constraints. We compare EncoDeep with existing fixed-point accelerators that use off-chip memory in the following.

Comparison with Fixed-point Accelerators. We perform a comparison between EncoDeep and prior work in Table 3.6. Specifically, we consider AlexNet with customized encoding as in Figure 3.16, which corresponds to hardware results of Table 3.5. The reported results include performance, either in terms of throughput (frames per second) or latency. Since the existing

Table 3.5: Summary of hardware resource utilization and performance. *EncoDeep* presents our model with encoded weights and activations whereas *DNN$_{fix}$* denotes a network with encoded weights and (8-bit) Fixed-point activations.

		Resource Utilization				Latency
		BRAM	DSP*	FF	LUT	(ms)
MNIST	*EncoDeep*	33	53	15223	9992	0.39
	DNN$_{fix}$	93	53	25884	12048	0.43
	Ratio	2.82×	1×	1.7×	1.2×	**1.1**×
CIFAR-10	*EncoDeep*	197	111	53953	31632	3.58
	DNN$_{fix}$	181	35	68255	28433	22.21
	Ratio	0.92×	0.32×	1.26×	0.9×	**6.2**×
SVHN	*EncoDeep*	146	111	42748	28393	3.39
	DNN$_{fix}$	143	35	67944	27934	22.59
	Ratio	0.98×	0.32×	1.59×	0.98×	**6.66**×
ImageNet	*EncoDeep*	3336	308	159663	82791	25.05
	DNN$_{fix}$	Exceeds Platform Constraints				

*25×18 DSB array.

frameworks utilize various FPGA platforms, it is crucial to take into account the instantiated computational capacity[4] and power consumption. Therefore, we compare the frameworks by means of performance-per-resource and performance-per-Watt. EncoDeep achieves higher normalized performance compared to the prior art. This is a direct result of using on-chip memory instead of the off-chip DRAM for feature transfer among DNN layers. The streaming buffers of our design allow EncoDeep to better utilize the arithmetic units by overlapping the execution of DNN layers, achieving a higher performance-per-resource. EncoDeep power advantage over existing accelerators is also rooted in the elimination of power-hungry DRAM access.

Execution Overhead of Encoding/Decoding. We study the runtime implication of online activation encoding by measuring the number of clock cycles required for different stages of EncoDeep `MVAU` engine. Figure 3.20 demonstrates the runtime break-down for each of the evaluated architectures. For a conventional non-encoded network, the `MVAU` would only perform

[4]the computational capacity is defined as $CAP = DSP \times Arr$, where Arr is the array size per DSP, e.g., 18×25 for Xilinx Virtex platforms.

Table 3.6: Comparison of Alexnet implementation between EncoDeep and existing fixed-point (FXD) and floating-point (FLT) DNN accelerators. To account for platform variations, we compare the throughput (images-per-second) and $\frac{1}{\text{Latency}}$ metrics normalized by computation capacity (CAP). We also compare performance-per-Watt to reflect power efficiency.

	Criterion	[70]	[25]	[71]	[72]	[73]	[74]	EncoDeep
	Precision	FLT	FXD	FXD	FXD	FXD	FXD	Flexible
	Acc(%)	-	55.41	52.4	56.5	-	54.27	53.2
	FPGA	690T*	GSD8†	690T*	690T*	AX115‡	ZU9§	VCU108*
	Freq(MHz)	100	120	150	100	200	300	152
	DSP**	3177	1504	14400	2872	2688	442	308
Img/sec	/CAP	0.55×	1	0.88×	2.33×	1.42×	0.49×	**3.03**×
	/Watt	3.14×	1	1.82×	**5.00**×	1.36×	-	4.54×
$\frac{1}{\text{Latency}}$	/CAP	-	1	0.05×	0.15×	-	-	**3.03**×
	/Watt	-	1	0.10×	0.32×	-	-	**4.54**×

*Virtex †Stratix-V ‡Arria10 §Zynq
**DSP array size is 25 × 18 for Xilinx and 18 × 18 for Altera/Intel FPGAs.

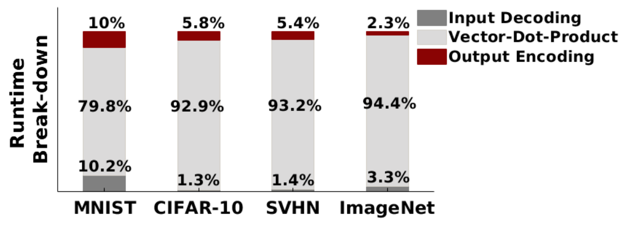


Figure 3.20: Runtime breakdown of EncoDeep accelerator.

Vector-Dot-Product (`VDP`) operations. As can be seen, for EncoDeep encoded models, the majority of clock cycles in MVAU execution still belong to `VDP` computation while the encoding/decoding overhead is small.

3.6 Conclusion

This chapter proposes a novel nonlinear quantization scheme to reduce the memory footprint of intermediate activations in convolutional neural networks' computation flow. The encoding compresses the activations and allows on-chip execution of the underlying FPGA accelerator without communicating the computed features with the off-chip DRAM. To ensure non-recurring engineering costs, an automated algorithm is proposed to configure the encoding bitwidth across layers of an arbitrary neural network. EncoDeep open-source API enables developers to convert high-level Pytorch description of a neural network into hardware modules without getting involved with the details of the design. We hope the provided API can advance research on reconfigurable DNN inference.

3.7 Acknowledgment

The material of this chapter has been published as: Samragh, Mohammad, Mojan Javaheripi, and Farinaz Koushanfar. "Encodeep: Realizing bit-flexible encoding for deep neural networks." ACM Transactions on Embedded Computing Systems (TECS) 19.6 (2020): 1-29. The thesis author and Mojan Javaheripi made equal contributions to this work. This work was supported in part by IARPA (2018-18022100004), ARO (W911NF1910317), and ONR (N00014-17-1-2500).

Chapter 4

Lookup-table based multiplication in DNNs

Arithmetic instructions on floating-point operands are executed in hardware via floating-point units (FPUs), which consume a lot of energy. In this chapter, we propose an alternative hardware architecture that computes multiplications using lookup tables. We then create LookNN as an end-to-end DNN that incorporates the aforsaid architecture. For each neuron, we store all possible input combinations and the corresponding outputs in a look-up table as illustrated in Figure 4.1. In order to make this implementation practical, we make sure that the operands (x, y) come from two finite sets in a pre-processing step. During execution, instead of utilizing the inefficient FPU, the neuron searches the look-up table to retrieve the pre-stored multiplication result.

To implement the lookup tables, we use associative memory architectures, which are shown promising in improving the performance and energy efficiency of parallel processors [83, 84]. An associative memory can implement a look-up table that returns the search result in a single cycle. In this chapter, we design associative memories to reduce the execution time and power consumption of an AMD Southern Island GPU. Each processing element of the enhanced

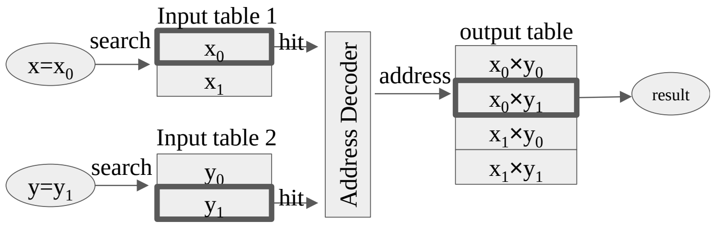


Figure 4.1: Implementation of multiplier with look-up table. Each neuron of LookNN has one such look-up table. The input operands are ensured to come from finite sets, $\{x_1, x_2\}$ and $\{y_1, y_2\}$, which are pre-stored in the two input tables. Pairwise multiplications, $x_i \times y_j$, are pre-stored in the output table.

GPU has access to a small associative memory, enabling it to realize efficient look-up table search. In summary, the main contributions of this chapter are as follows:

- We propose LookNN, a methodology to replace DNN multiplications with look-up table search.
- We provide theoretical analysis of LookNN's error, and summarize the design guidelines that are achieved based on the analysis.
- We design associative memory blocks to reduce the power and execution time of FPUs in AMD Southern Island GPU. The enhanced GPU can realize efficient look-up table search. Our evaluations on four DNN applications demonstrate an average of 2.5$\times$ speedup and 2.2$\times$ energy improvement without reducing the inference accuracy.

4.1 Related Work

Fixed-point quantization of DNNs is investigated in previous work [85], and the extreme quantization case is proposed in [86], which utilize trained binary parameters to avoid multiplication. Despite the effectiveness of quantization, many applications require floating-point precision since iterative training algorithms often update the parameters using gradients whose values are too small to compared to the error caused by quantization [87]. LookNN is similar to [86] in that it customizes the model during the training to reduce the additive error, and is different in utilizing floating-point values rather than fixed-point or binary values.

Model compression is another direction that has been investigated to optimize inference. Han et al. [34] train sparse models with shared weights to compress the model. LookNN trains shared weights with a different approach. The shared weights in our model are subjective to a single neuron, making it possible to achieve higher accuracy with fewer shared weights per neuron. The compressed parameters of [34] can be used to realize ASIC/FPGA accelerators [88]. However, compression does not help with execution on general purpose processors, in which case the compressed parameters should be decompressed into the original parameters.

Dimensionality reduction is investigated for efficient execution of DNNs [89]. Their method is orthogonal to LookNN; Our framework can receive pre-trained DNNs and further reduce their power consummation and execution time.

This chapter targets GPU implementation of lookNN using associative memory. Associative memory in a form of look-up table has been shown a great opportunity to improve the execution time and energy consumption of parallel processors [84, 90]. Arnau et al. [84] utilize associative memories beside GPU floating-point units to enable error-free execution. Ternary content addressable memory (TCAM) is part of the associative memory that can search its contents in a single cycle. Recently, efficient TCAMs have been designed using high density and low leakage power non-volatile memories (NVMs) such as Spin Transfer Torque RAMs (STT-RAMs), ReRAM and Ferro electric RAMs (FeRAMs) [91–93]. Approximate associative memories using voltage overscaling and long time precharging are proposed to reduce FPU computations [90, 94]. However, in all previous work, computations rely on the FPUs and the computation efficiency is bounded by the processor's pipeline stage. To the best of our knowledge, LookNN is the first floating-point implementation on DNNs that completely avoids using FPUs for multiplication.

4.2 LookNN Framework

The global flow of our methodology is presented in Figure 4.2. The user defines the error tolerance, hardware constraints, the training data, and the baseline DNN. Based on this information, the customization unit modifies the baseline DNN and maps it into LookNN. The customization unit encompasses three major operations: weight clustering, error estimation, and weight retraining. Throughout this chapter, we use the terms "clustered weights" and "shared weights" interchangeably. We use the term "customization" to denote error adjustment. Using

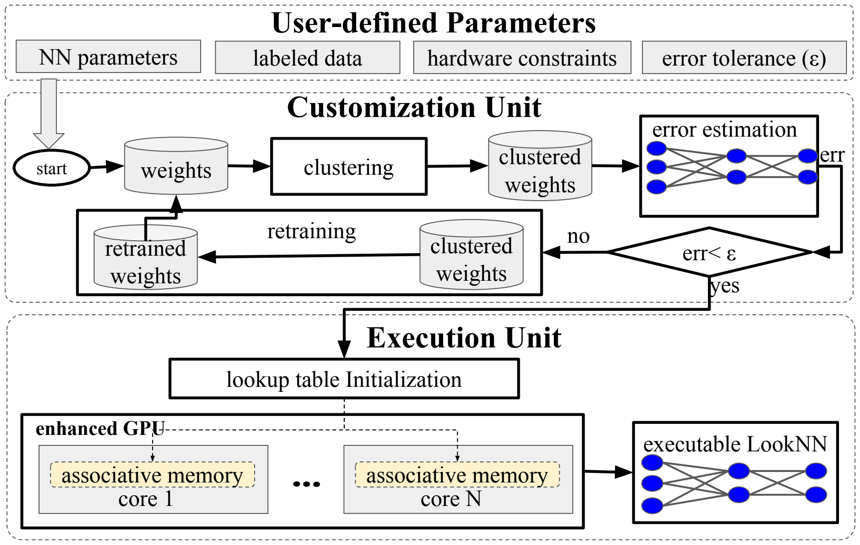


Figure 4.2: Global flow of LookNN. The user assigns hardware constraints such as the look-up table size in the underlying hardware. The customization unit runs a greedy algorithm to adapt the DNN to LookNN. The execution unit maps LookNN to the enhanced GPU and initiates the associative memories with proper values.

our proposed recursive greedy algorithm described in section 4.2.1, the customization unit trains a DNN based on the hardware constraints (i.e. look-up table size) provided by the user. The algorithm has the following minimization objective:

$$\min_{W}(\Delta e) \quad s.t. \quad U(W) = N_{clusters}, \qquad \Delta e = e_{LookNN} - e_{baseline} \tag{4.1}$$

Where Δe is the additive error, e_{LookNN} and $e_{baseline}$ are the ratio of misclassified validation examples using LookNN and the baseline DNN respectively, $U(W)$ is the number of distinct values in each row of the weight matrices, and $N_{clusters}$ is a pre-specified parameter provided by the user. $N_{clusters}$ directly translates to the power consumption and runtime in the execution phase.

After customization, the execution unit exploits the enhanced GPU to implement LookNN. Each GPU core has access to an associative memory whose energy consumption and runtime depend on the look-up tables' size. The associative memories are initialized with the multiplication operands and their pairwise multiplication results. DNN customization and associative memory initialization are done once and their overhead is amortized across all future executions.

4.2.1 LookNN Customization Unit

Consider the look-up table in Figure 4.1. In LookNN, the first input table stores N_q rows representing the values that the preceding layer's neurons could possibly take. The second input table stores the neuron's incoming weights, $W^l_{i:}$, denoting a row of the 2-D matrix, W^l.

Algorithm 3 presents a pseudo code for the customization phase. It iterates a loop consisting of three major operations: weight clustering, error estimation, and weight retraining. Below we describe each operation in detail.

weight clustering: Lines 2 through 6 of Algorithm 3 perform weight clustering. Each row of the matrix W^l is partitioned into $N_{clusters}$ clusters; The elements of the row are replaced by their closest centroids. The objective of clustering is to minimize the within cluster sum of squares (WCSS):

$$\min_{c_{i1},\dots,c_{iN_{clusters}}} (WCSS = \sum_{k=1}^{N_{clusters}} \sum_{W^l_{ij} \in c_{ik}} ||W^l_{ij} - c_{ik}||^2) \tag{4.2}$$

where $C = \{c_{i1}, c_{i2}, \dots, c_{iN_{clusters}}\}$ are the cluster centroids. We use K-means algorithm for clustering. Before clustering, the weights' values are scattered (Figure 4.3a) and the look-up table is

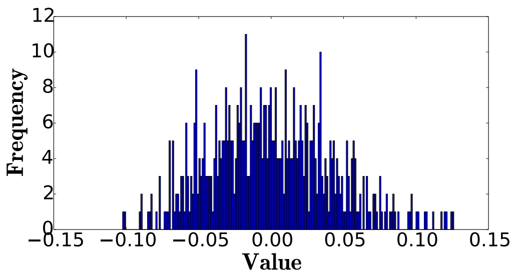
Algorithm 3 LookNN Customization Algorithm

```
inputs:
    pre-trained weights and biases: $\{(W^i, b^i)\}_{i=1}^{N_{layers}}$
    number of clusters: $N_{clusters}$
    labeled data: $(x_{train}, y_{train})$, $(x_{valid}, y_{valid})$
    inference error tolerance: $\varepsilon$
    maximum number of training epochs: $max\_iter$
outputs:
    LookNN clustered weights and biases: $\{(W^i, b^i)\}_{i=1}^{N_{layers}}$
    validation inference error: $e_{LookNN}$
 1: for $iteration = 1 \ldots max\_iter$ do
 2:     for $l = 1 \ldots N_{layers}$ do
 3:         for $i = 1 \ldots n_{l-1}$ do
 4:             $W^l_{i:} \leftarrow$ Kmeans($W^l_{i:}$, $N_{clusters}$)
 5:         end for
 6:     end for
 7:     $e_{LookNN} \leftarrow$ error($\{(W^i, b^i)\}_{i=1}^{N_{layers}}$, $x_{valid}$, $y_{valid}$)
 8:     if ($e_{LookNN} < \varepsilon$) then
 9:         return $W^{1 \ldots N_{layers}}$, $b^{1 \ldots N_{layers}}$, $e_{LookNN}$
10:     end if
11:     retrain($W^{1 \ldots N_{layers}}$, $b^{1 \ldots N_{layers}}$, $x_{train}$, $y_{train}$)
12: end for
13: return $W^{1 \ldots N_{layers}}$, $b^{1 \ldots N_{layers}}$, $e_{LookNN}$
```

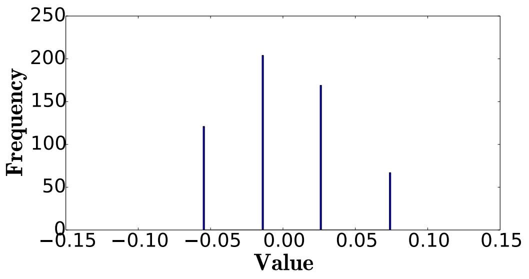
large. After clustering, the number of rows in the look-up table is significantly decreased (Figure 4.3b).

weight retraining: Weight clustering is often accompanied by some degree of additive error, $\Delta e = e_{LookNN} - e_{baseline}$, to compensate for which we retrain the DNN for a pre-specified number of epochs (line 11 of Algorithm 3). After retraining, the algorithm loops back to line 2 to cluster the retrained weights. Figure 4.4 depicts an example LookNN's additive error and the average WCSS of the weights. A retrained matrix is likely to exhibit less WCSS; Therefore, the error is reduced in subsequent iterations.

error estimation: The error estimation module computes the misclassification error, e_{LookNN}, over the validation data set (line 7 of Algorithm 3). This module simply computes the cross-validation error of LookNN with its clustered weights and quantized neurons. We use

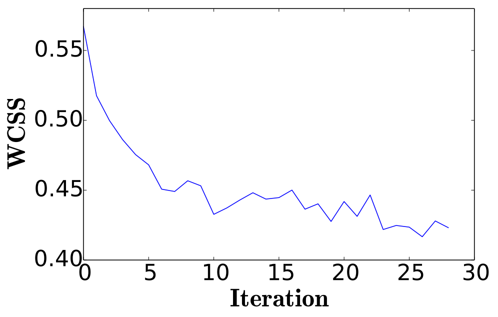


(a) before clustering

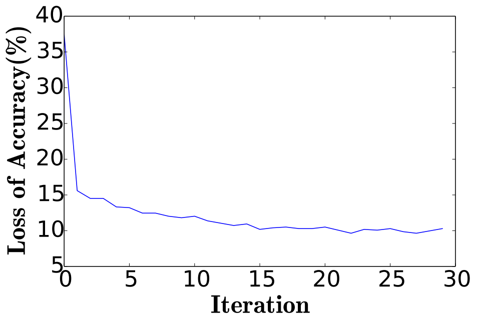


(b) after clustering

Figure 4.3: The distribution of a neuron's input weights. (a) The look-up table should maintain 561 rows. (b) The table should maintain $N_{clusters} = 4$ rows.



(a) WCSS



(b) loss of accuracy (Δe)

Figure 4.4: (a) The within cluster sum of squares (WCSS) is decreased in subsequent iterations. (b) The loss of accuracy is also decreased due to the reduction of the WCSS.

K-means to apply nonlinear quantization to the input layer, whereas hidden layers are quantized linearly.

4.2.2 LookNN Error Analysis

LookNN's additive error has two reasons: neuron quantization and weight clustering. We model the effect of both operations as additive Gaussian noise $N(\eta, \delta)$ with mean η and variance δ. Consider an $n \times m$ weight matrix W multiplied by an $m \times 1$ input vector a. We assume that clustering replaces each connection W_{ij} with $W_{ij} + N_{ij}(0, \delta_w)$ and quantization replaces each neuron a_j with $a_j + N(0, \delta_a)$. For simplicity, We also assume that the Gaussian noises are independent.

Consider a neuron computing a dot-product of the form $< W_{i:}, a >$. LookNN adds noise to the dot-product operands, resulting in the following noisy output:

$$z_{i-noisy} = \sum_{j=1}^{m} (W_{ij} + N(0, \delta_w)) \times (a_j + N(0, \delta_a)) = z_{i-original} + N_i \tag{4.3}$$

where $z_{i-noisy}$ is the noisy dot-product in LookNN, $z_{i-original}$ is the actual dot-product in the baseline DNN and N_i is additive noise. The additive noise can be approximated as:

$$N_i \approx \sum_{j=1}^{m} W_{ij} \times N(0, \delta_a) + \sum_{j=1}^{m} a_j \times N(0, \delta_w) \tag{4.4}$$

N_i is a linear sum of independent Gaussian random variables with zero means; Therefore, it is a Gaussian random variable $N(0, \delta_i)$. The variance δ_i is obtained using equation 4.5:

$$\delta_i = \sum_{j=1}^{m} ||W_{ij}||^2 \times \delta_a + \sum_{j=1}^{m} ||a_j||^2 \times \delta_w \tag{4.5}$$

The expected value of δ_i is:

$$E(\delta_i) = m \times (E(||W_{ij}||^2) \times \delta_a + E(||a_j||^2) \times \delta_w) \tag{4.6}$$

δ_i denotes the variance of the quantization noise, δ_a, for the next hidden layer's neurons; Therefore, the noise over the next layers can be characterized in a similar manner. In fact, the noise propagates from the input layer through the output layer. The variance of the noise in the output layer is correlated with LookNN's error. High variance is interpreted as high probability of changing the actual DNN's outputs. Equation 4.6 might not be perfectly exact for the general case, but it brings useful insights that we took into account for devising the customization process:

- The terms "δ_w" and "δ_a" suggest that minimizing the WCSS is effective for error reduction (see equation 4.2). The WCSS directly translates to "δ_w" of all layers and "δ_a" of the first layer.

- The expected value is proportional to "m", the number of input neurons in the preceding layer. Hence, matrices with higher number of elements per row require more shared values to reduce the WCSS.
- The term "$E(||W_{ij}||^2)$" demonstrates the importance of training regularized matrices. This can be achieved by applying Dropout [95] or adding a weight decay to the training minimization objective [96].

4.2.3 LookNN Execution Unit

Computing each neuron is assigned to one of the GPU streaming cores. Each core maintains an associative memory depicted in Figure 4.5. We use crossbar memristor, an access-free transistor memristive memory, to design both the TCAM and the crossbar memory. Prior to the execution, the first TCAM is initialized with N_q quantized values of the preceding layer. The second TCAM is initialized with $N_{clusters}$ shared weights. The crossbar memory holds $N_q \times N_{clusters}$ pairwise multiplication outputs. During execution, for a pair of input operands, the two TCAMs are searched in parallel, the address decoder generates the proper address, and the multiplication result is fetched from the crossbar memory. Below we discuss the memory blocks in detail.

TCAM: Figure 4.5b illustrates the TCAM architecture. We design the TCAM using non-volatile memories. The values are stored on cells based on the NVM resistance state. Low and high resistances denote Logics 1 and 0 respectively. Before each search operation, the match lines (MLs) in all rows are pre-charged to Vdd voltage. During the search operation, a buffer distributes the input data among all rows. All MLs will discharge except the TCAM row matching the input data. The sense amplifier samples MLs at each clock cycle to identify the hit rows.

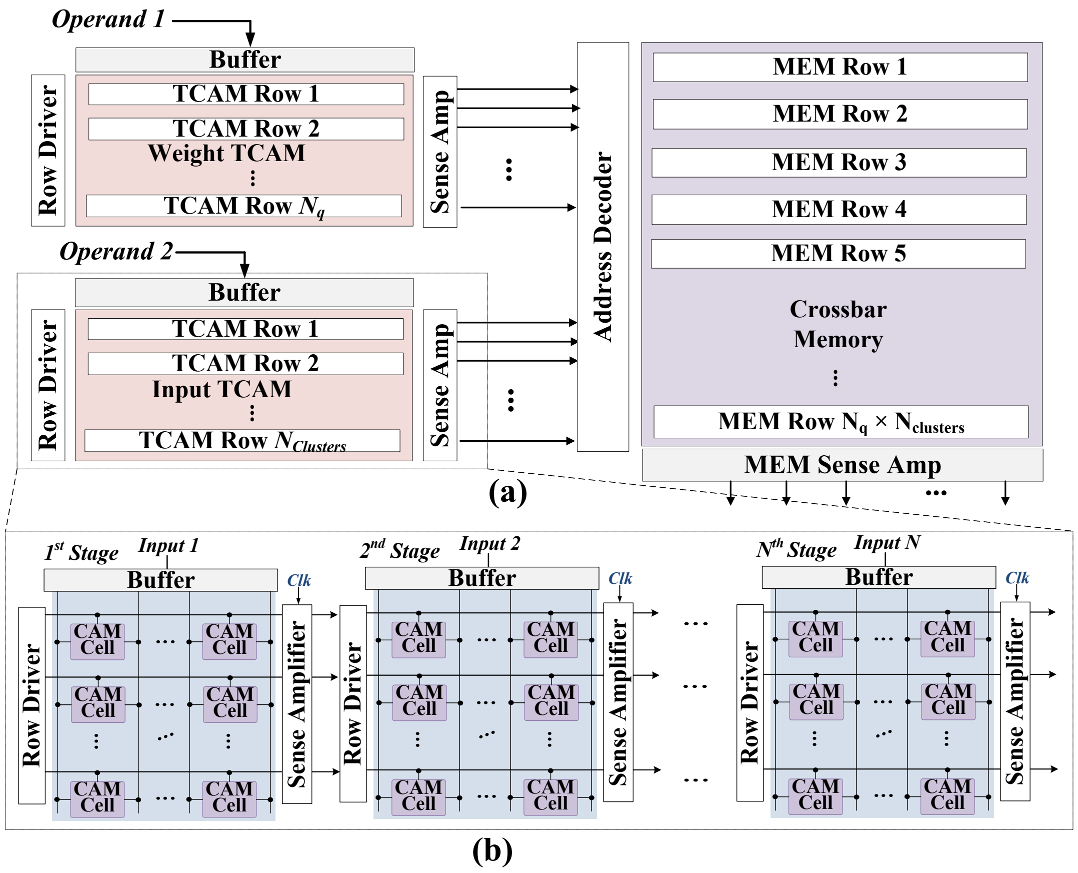


Figure 4.5: (a) LookNN hardware for memory-based computation. N_q and $N_{clusters}$ determine the number of active rows in the TCAMs. (b) The structure of multistage TCAM.

Mutli-Stage TCAM: In order to reduce TCAM's power consumption, we reduce its switching activity by splitting it into multiple stages. Figure 4.5b presents an N-stage TCAM. The first stage searches through 1/N of the data. The MLs of the subsequent stages are selectively pre-charged based on the hit rows of their preceding stages, resulting in significant energy saving. Dividing the TCAM into more stages reduces its energy consumption, but results in increased delay. Table 4.1 shows the trade-off for a table with $N_q = 16$ and $N_{clusters} = 16$. To choose the number of stages, we consider the energy-delay product (EDP) which is minimized using 4 stages.

Crossbar Memory: Compared to the two TCAMs, the crossbar memory consumes lower energy. It occupies negligible area since we implement it in three dimensional (3D)

Table 4.1: Block size impact on energy consumption and search delay of LookNN ($N_q = 16$, $N_{clusters} = 16$, $bit - width = 32$)

	16-stage	**8-stage**	**4-stage**	**8-stage**	**1-stage**
Energy(fJ)	208	266	447	834	1307
Delay(ns)	0.80	0.58	0.28	0.17	0.11
EDP(J.s)$\times 10^{-24}$	167	156	128	148	150

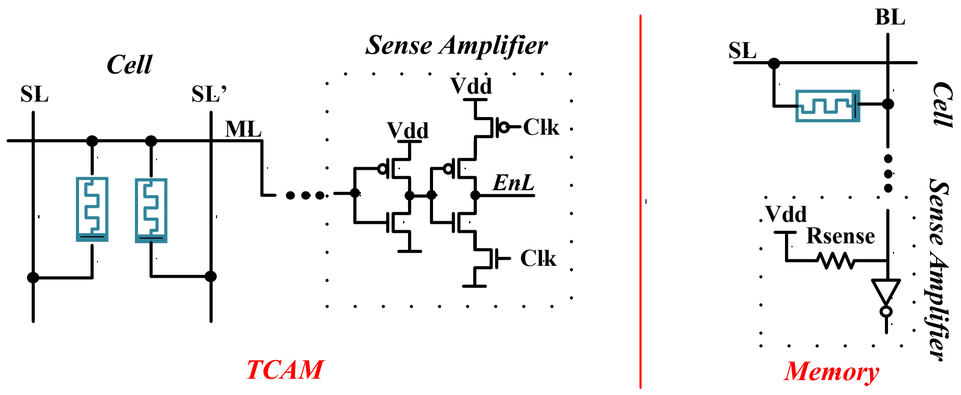


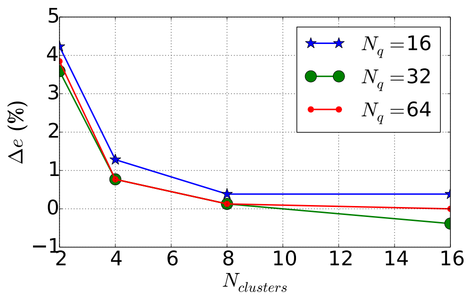
Figure 4.6: TCAM and crossbar memory cells are implemented over CMOS FPU.

architecture [97]. The crossbar memory is implemented over the CMOS FPU, resulting in zero area overhead. Figure 4.6 shows the structure of TCAM and crossbar memory cells.

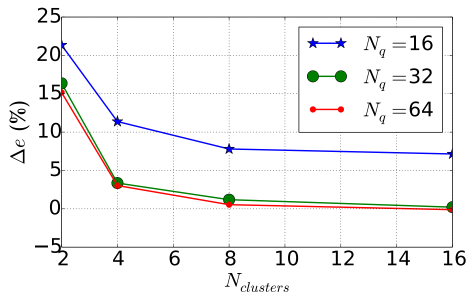
Associative Memory Configurations: LookNN can exchange accuracy for performance and energy efficiency. Increasing N_q and $N_{clusters}$ improves LookNN's accuracy at the cost of increase in energy consumption and execution time. The energy consumption of a look-up table search is characterized as:

$$E = E_T(N_q) + E_T(N_{clusters}) + E_C(N_q \times N_{clusters}) \tag{4.7}$$

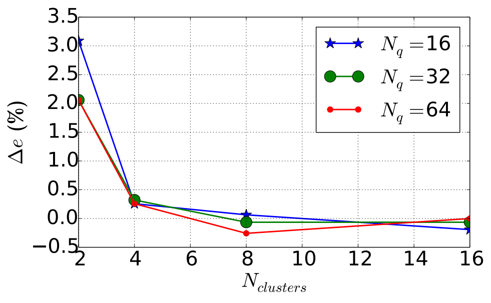
where E_T and E_C denote the energy consumption of TCAM and crossbar memory respectively. The execution time depends on the largest TCAM size, $max(N_q, N_{clusters})$, since the two TCAMs are searched in parallel. Table 4.2 shows LookNN search energy and delay in different associative memory configurations storing different numbers of patterns. Energy and delay are normalized to



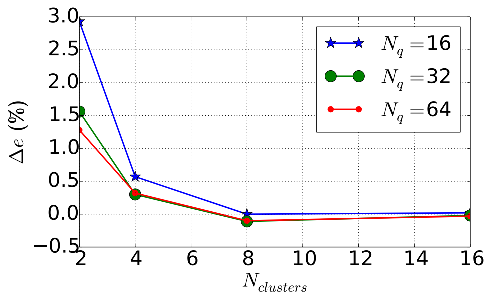
(a) Voice Recognition



(b) Hyperspectral Imaging



(c) Human Activity Recognition



(d) MNIST

Figure 4.7: Additive error of LookNN running four applications. The horizontal axis is the number of shared weights per neuron. The vertical axis represents $\Delta e = e_{LookNN} - e_{baseline}$. Each curve corresponds to a different neuron quantization.

those of CMOS-based FPU multipliers. As the results show, all these configurations outperform FPU computation in both energy and delay. Our customization unit can adapt LookNN to any of these configurations.

Table 4.2: LookNN normalized energy and delay in each configuration

	config1	***config2***	***config3***	***config4***	***config5***	***config6***
$(N_q, N_{clusters})$	(16,2)	(16,4)	(16,8)	(16,16)	(32,16)	(64,16)
Energy	0.03	0.04	0.05	0.07	0.12	0.26
Delay	0.12	0.13	0.15	0.18	0.23	0.34

Scalability: In large-scale DNN, each streaming core is responsible for execution of multiple neurons. In such scenarios, we share each streaming core among multiple neurons of the same layer. The first TCAM does not need to be changed. The second TCAM should be extended with new weights. The crossbar memory should also be extended with new output values. For instance, *config1* of Table 4.2 can be extended to $(N_q, N_{clusters}) = (16, 16)$, allowing the associative memory to be shared among 8 neurons, each of which searches 2 rows of the second TCAM. The increase in the search energy is negligible since the number of searched rows is the same as the associative memory before being extended. The delay is neither affected since it depends on $max(N_q, N_{clusters})$.

4.3 Experimental Results

4.3.1 Experimental Setup

We integrate LookNN on the AMD Southern Island GPU, Radeon HD 7970 device including 2048 streaming cores. We perform circuit level simulations on HSPICE simulator using 45-nm TSMC technology. We use multi2sim, a cycle accurate CPU-GPU simulator for architecture simulation [98] and change the GPU kernel code to enable memory pre-loading and runtime simulation. We use Synopsys Design Compiler [99] to calculate the energy consumption of the 6-stage balanced FPUs in GPU architecture in 45-nm ASIC flow. DNN applications are realized using OpenCL, an industry-standard programming model for heterogeneous computing. We use the Scikit-learn library [100] for Kmeans and Nearest Neighbour Search. Tensorflow [101] is used to realize the DNN in the customization unit. We evaluate LookNN on four applications described below.

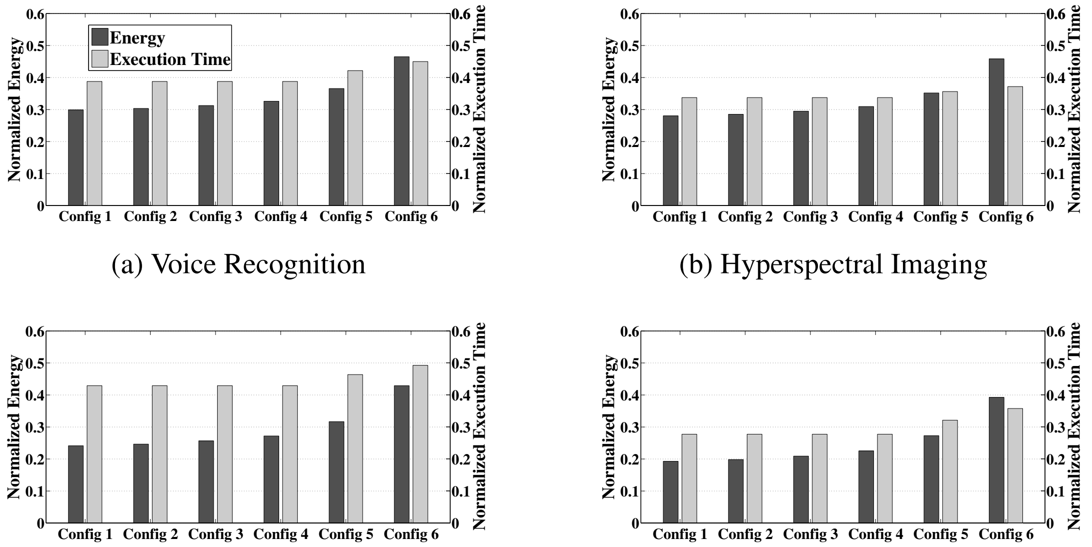
Voice Recognition: Many mobile applications require online processing of vocal data. We evaluate lookNN with the Isolet dataset [102] which consists of speech collected from 150 speakers. The goal of this task is to classify the vocal signal to one of the 26 English letters.

Hyperspectral Imaging: Hyperspectral imaging involves classification of different objects based on the reflectance spectra. The objective of this classification task is to recognize 9 different materials on the earth [103].

Human Activity Recognition: For this data set, the objective is to recognize human activity based on 3-axial linear acceleration and 3-axial angular velocity that have been captured at a constant rate of 50Hz [104].

MNIST: MNIST is a popular machine learning data set including images of handwritten digits [105]. The objective is to classify an input picture to one of the ten digits $\{0 \ldots 9\}$.

4.3.2 LookNN Evaluation



(a) Voice Recognition (b) Hyperspectral Imaging

(c) Human Activity Recognition (d) MNIST

Figure 4.8: Normalized energy consumption and execution time of LookNN in different configurations for four DNN applications. The baseline DNN is run on traditional GPU. LookNN is deployed on the enhanced GPU.

For each of the four data sets, we compare the baseline DNN and its corresponding LookNN. The baseline utilizes the FPUs of the GPU wherase LookNN exploits the associative memory. Specifically, we compare them in terms of accuracy, running time and energy consumption. Stochastic gradient descent with momentum [106] is used for training. The momentum is set to 0.1, the learning rate is set to 0.001, and a batch size of 10 is used. Dropout [95] with drop rate of 0.5 is applied to hidden layers to avoid over-fitting. All data sets are normalized prior to the training, such that the features have 0 mean and standard deviation of 1. Table 4.3 presents the baseline DNN Topologies and their error rates running four applications. The activation functions are set to “Rectified Linear Unit” clamped at 6. A “Softmax” function is applied to the output layer.

Table 4.3: Baseline DNN and their error running four applications

Application	**Network Topology** $(\mathbf{l^0,l^1,l^2,l^3})$	$\mathbf{e_{baseline}}(\%)$
Voice Recognition	617, 500, 500, 26	4.4
Hyper-spectral Imaging	200, 500, 500, 9	6.6
Human Activity Recognition	561, 500, 500, 12	3.4
MNIST	784, 500, 500, 10	2.4

The additive error, $\Delta e = e_{LookNN} - E_{baseline}$, for different $(N_q, N_{clusters})$ configurations is depicted in Figure 4.7. It is clear that increasing N_q and $N_{clusters}$ results in reduced error. Note that for some configurations the error is negative (e.g. Voice recognition with $(N_q, N_{clusters}) = (32, 16)$), meaning that LookNN can achieve a lower error rate than the baseline; This happens due to the approximate nature of the baseline DNN. For a fixed N_q, the error reduction exhibits diminishing return with respect to $N_{clusters}$. Therefore, both of the parameters $(N_q, N_{clusters})$ should be considered for error adjustment.

For each application, Table 4.4 summarizes the additive error using selected LookNN configurations, each of which results in a different execution time and energy consumption. We report zero additive error for a negative Δe. A LookNN configuration can result in different error rates for different applications. This is due to the fact that some applications require precise numerical computations (e.g. Hyperspectral Imaging) while others can tolerate more numerical inaccuracy (e.g. MNIST).

Figure 4.8 depicts the energy consumption and execution time of LookNN normalized to those of the baseline DNN. In both experiments, the overhead of data movement is accounted for. In order to get a zero Δe, our design requires to use associative memories at least in *config6* which results in an average of 2.2× energy improvement and 2.5× speedup compared to the conventional AMD GPU architecture. In addition, our enhanced GPU achieves 3× energy improvement and 2.6× speedup if we tolerate an additive error of less than 0.2%. The trade-off is much more sensible if we solely consider the cost of multiplication, which is the main focus of this chapter. For instance, compared to multiplication via FPU, LookNN achieves 33× energy improvement and 8.3× runtime improvement at *Config1*, while achieving 3.8× energy improvement and 3× runtime improvement at *Config6* (see Table4.2).

Table 4.4: Additive error in different LookNN configurations

	config1	***config2***	***config3***	***config4***	***config5***	***config6***
$(N_q, N_{clusters})$	(16,2)	(16,4)	(16,8)	(16,16)	(32,16)	(64,16)
Voice Recognition Δe	4.2%	1.3 %	0.4%	0.4%	0%	0%
Hyperspectral Imaging Δe	21%	11.4 %	7.8%	7.2%	0.2%	0%
Human Activity Recognition Δe	3.1%	0.25 %	0.2%	0%	0%	0%
MNIST Δe	3%	0.6 %	0%	0%	0%	0%

4.4 Conclusion

We propose LookNN, a simplified DNN that replaces multiplications with look-up table search, resulting in significant improvement in execution time and power consumption. Prior to converting DNNs to LookNN, our customization unit can adjust DNNs such that their accuracy is retained after converting to LookNN. The main advantage of LookNN over previous simplified models is that it enjoys floating-point parameters which is indeed necessary for many applications. LookNN can be deployed on either general purpose processors or FPGA/ASIC accelerators. Recently, associative memories have been used to enhance processors to bypass redundant computations. We employ one such enhanced GPU to evaluate LookNN. Our evaluations demonstrate an average of 2.2× energy improvement and 2.5× speedup with zero addtive error. LookNN can also be leveraged to exchange accuracy for efficiency; In our evaluations, LookNN achieves an average of 3× energy improvement and 2.6× speedup with an additive error rate of less than 0.2%.

4.5 Acknowledgment

The material in this chapter has been published as: Razlighi, Mohammad Samragh, Mohsen Imani, Farinaz Koushanfar, and Tajana Rosing. "Looknn: Neural network with no multiplication." In Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, pp. 1775-1780. IEEE, 2017. The thesis author was the primary contributor of this material. This work was supported in parts by NSF grant (1527034), ONR grant (N00014-11-1-0885), MURI grant (FA9550-14-1-0351), and Jacobs School of Engineering UCSD Powell Fellowship.

Chapter 5

Customizing quantization and clustering for secure DNN inference

There is an increasing surge in cloud-based inference services that employ deep learning models. In this setting, the server trains and holds the DNN model and clients query the model to perform inference on their data. One major shortcoming of such service is the leakage of clients' private data to the server, which can hinder commercialization in certain applications. For instance, in medical diagnosis [107], clients would need to expose their "plaintext" health information to the server, which violates patient privacy regulations such as HIPAA [108].

One attractive option for ensuring clients' content privacy is the use of modern cryptographic protocols as they provide provable security guarantees [1,5,6,109–116]. Let $f(\theta,x)$ be the inference result on client's input x using server's parameters θ. By executing cryptographically-secure operations, client and server can jointly compute $f(\theta,x)$ without revealing x to the server or θ to the client. We refer to this process as *oblivious inference* in the remainder of the thesis. Unlike plaintext inference, oblivious inference protects the privacy of both parties. The challenge, however, is the excessive computation and/or communication overhead associated with privacy-preserving computation. For example, the contemporary state-of-the-art for performing oblivious inference on a single CIFAR-10 image requires exchange of $\sim$ 3.4 GB of data and takes $\sim$ 10 seconds [4].

Early research on oblivious inference mostly focused on developing protocols for inference of a given DNN model, without making major modifications to the model itself [1, 5, 6, 109–116]. Recently, a body of work has explored modifying the DNN architecture such that the resulting model is more amenable to secure computation [2–4, 117]. The gained efficiency in the prior work comes at the cost of reduction in the inference accuracy. This chapter presents COINN, a provably secure framework for oblivious inference. The result of the thesis author's research provides novel customization techniques that enable the design of efficient, secure, and accurate oblivious inference. The contributions discussed in this chapter are outlined as follows:

Customizing DNN quantization for oblivious inference.. The execution cost of oblivious inference relies heavily on the numerical precision (bitwidth) of the underlying operations. Therefore, applying fixed-point quantization to DNN weights and activations can boost the performance of oblivious inference while maintaining the inference accuracy. The challenge is that off-the-shelf quantization techniques are not directly applicable for ciphertext (oblivious) execution as they are developed for plaintext DNN inference. More specifically, operations such as full-precision accumulation, rounding, and scaling incur a negligible cost in plaintext inference but the same operations are extremely costly in ciphertext inference. The thesis author's research in this chapter provides customization techniques to avoid such costly ciphertext operations while enjoying the efficiency benefits gained by quantization.

Customizing factored matrix multiplication for oblivious inference.. Recall from Chapter 3 that DNN weights can be encoded to improve the efficiency of plaintext inference. This chapter explores the benefit of weight encoding in oblivious inference. Specifically, matrix multiplication with encoded weights can be rendered via factored dot product operations, where the majority of multiplication operations are replaced with conditional additions. Through this replacement, COINN achieves significant performance boosts in oblivious inference of CONV

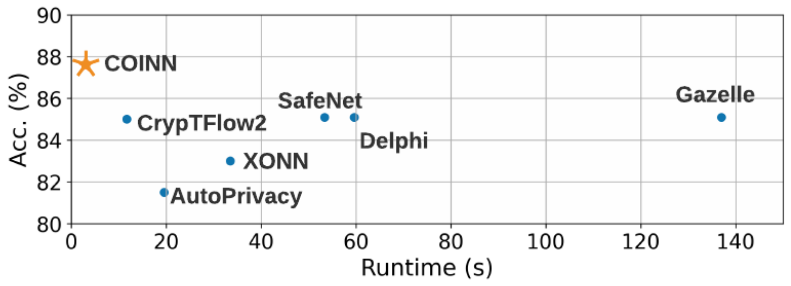


Figure 5.1: Accuracy and secure inference runtime of a 7-layer DNN on CIFAR-10 dataset using prior work: Gazelle [1], Delphi [2], SafeNet [3], XONN [4], Autoprivacy [5], and CrypTFlow2 [6]. The ★ symbol represents COINN.

and FC layers.

Automated parameter selection for oblivious inference.. To fully exploit the benefits of quantization and factored matrix multiplication, the aforementioned optimization techniques should be customized per layer, such that efficiency and accuracy are simultaneously satisfied. COINN provides an automated parameter selection module based on Genetic algorithms [48] to determine the heterogeneous quantization and encoding parameters across a given DNN's layers. As a result, as shown in Figure 5.1, COINN is able to achieve a higher accuracy with a lower execution time compared to all contemporary work in oblivious inference.

5.1 Notations

Throughout this chapter, we represent scalars with lowercase x, vectors with bold lowercase $\mathbf{x}$, 2-dimensional matrices with uppercase X, and higher order tensors with bold uppercase letters $\mathbf{X}$. Element selection is denoted by brackets $\mathbf{x}[i]$ and $x\langle i\rangle$ denotes the i-th bit of scalar x. $\mathbf{0}$ denotes a vector/matrix/tensor with all the entries set to 0. We denote the computational security parameter with κ and set it to 128 following common standard [1, 6, 116].

5.2 Related Work

In this section, we review the related work that employ similar settings as ours, i.e., cryptographically secure two-party protocols where the server owns the model and the client owns the input. There are two classes of techniques: Homomorphic Encryption (HE) [118], which is heavy on computation and Multi-Party Computation (MPC) techniques such as Garbled Circuits (GC) [20] and Arithmetic Sharing (AS) [14], which are heavy on communication.

CryptoNets [119] is perhaps the pioneer of 2-party oblivious inference. More efficient variants and compilers have since been proposed for optimized DNN inference [109–114]. HE-based methods such as [119] allow outsourcing the majority of the computations to the more capable party, i.e., the server. However, frameworks that are entirely based on HE replace the nonlinear activations with HE-friendly polynomial approximations, resulting in reduced inference accuracy. Oblivious inference based on GC has also been proposed [115] which provides better accuracy but suffers from long run times due to the large communication cost of multiplications in GC. To mitigate this, XONN [4] presents a GC-based framework for Binarized Neural Networks (BNN) where all multiplications are replaced with cost-free XNOR operations. Nevertheless, the binary weights and activations in a BNN have an adverse effect on the inference accuracy.

At present, most efficient secure inference engines employ a hybrid approach –using the most efficient cryptographic primitive for a particular layer. MiniONN [116] employs a combination of AS, GC, and HE. Follow-up works Gazelle [1] and Delphi [2], support efficient HE-based linear operations along with GC-based nonlinear functions, and perform secure protocol conversion when necessary [1, 116]. Subsequent works [4, 120] have pointed out security vulnerabilities in HE-based methods, safeguarding against which would result in increased runtime. CrypTFlow2 [6] proposes a hybrid protocol that supports both HE and AS-based linear layers and has custom protocols for secure comparison (used in ReLU and MP) which incur less communication at the cost of higher number of communication rounds compared to GC.

A parallel line of work in oblivious inference focuses on applying optimizations to reduce the secure execution cost of previously proposed security protocols. The contributions in this domain can be categorized in two separate directions: (1) adjusting the parameters for the secure protocol, and (2) changing the DNN architecture for improved secure execution. In the first category, recent work [5, 121] adjust the HE parameters for hybrid HE-GC protocols, i.e., Gazelle and Delphi, to reduce the secure execution cost. The methods in the second category [2, 3, 117] reduce the number of ReLU activations throughout the network to reduce the GC communication and runtime in hybrid HE/AS and GC protocols.

Perhaps the most related model-adjustment techniques to COINN are the quantization in [122, 123]. These works have two major differences with COINN quantization. Firstly, they simply use homogeneous bitwidths for all DNN weights/activations. We show that by solving the challenging problem of heterogeneous bitwidth selection, secure execution cost can be significantly lowered without hurting model accuracy. Secondly, the aforesaid works use the available quantization schemes optimized for the plaintext domain [124, 125], while COINN develops a new cipher domain optimized quantization scheme that replaces costly quantization operations with variants that incur a negligible GC cost.

COINN bridges the gap between protocol design and ML model adjustment to optimize the ciphertext execution of both linear and non-linear operations. Compared to works that only optimize the cryptographic protocols [1, 6, 116], the contributions of our work lie in designing security-aware low-bit quantization and introduction of factored multiplication and its accompanying custom secure execution protocol. Compared to works that optimize the ML model [2–5, 117], our model adjustment techniques are scalable to many-layer architectures trained for complex tasks such as ImageNet. Additionally, COINN quantization and factored multiplication together with our automated parameter configurator achieve a better accuracy-runtime tradeoff compared to prior model adjustment methods such as modifying ReLU layers [2, 3, 117].

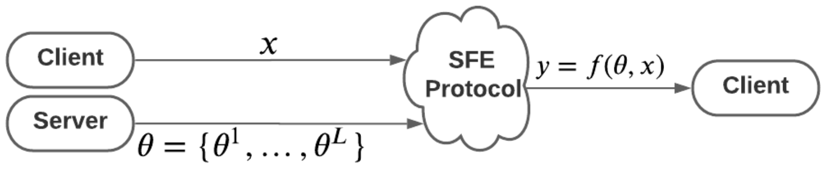


Figure 5.2: The server and client use a secure function evaluation (SFE) protocol to perform oblivious inference. At the end of the protocol, client learns $y = f(\theta,x)$ without learning server's parameters θ or revealing x to server.

5.3 Scenario and Threat Model

Figure 5.2 presents the scenario in oblivious inference. The neural network architecture f is known by both server and client. The server holds the set of trained parameters, i.e., $\theta = \{\theta^1, \ldots, \theta^L\}$, and the client holds the input query to the neural network, i.e., x. The two parties engage in a secure function evaluation protocol, where the client learns the inference result $y = f(\theta,x)$. Similar to prior work, we consider the honest-but-curious scenario [1–6,116,117]. In this threat model, the two parties follow the protocol that they agree upon to compute the output, yet they may try to learn about the other party's data as much as they can. As such, the protocol should guarantee the following requirements:

- x or $f(\theta,x)$ are not revealed to the server.
- θ is not revealed to the client.
- Client and server do not learn intermediate activations.

5.4 COINN Methodology

Figure 5.3 depicts the overall flow of the COINN framework. The model owner performs plaintext model customization, which requires quantization, clustering (encoding), and automated parameter configuration. Once the model is customized in plaintext, the optimized layers are

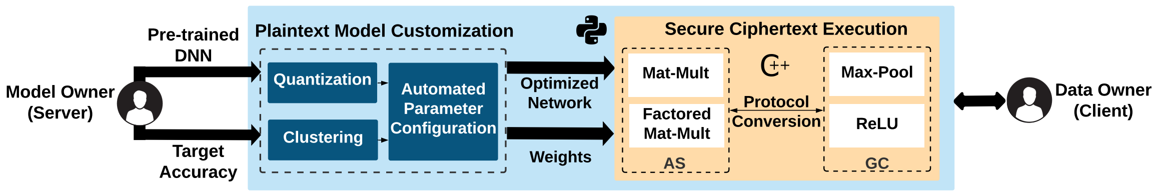


Figure 5.3: Overview of COINN. The plaintext model customization is only performed once per DNN and provides the optimized network for COINN secure inference.

converted into executable that run the oblivious inference. The linear and nonlinear layers of COINN are executed in AS and GC, respectively. The thesis author's contributions lie in the plaintext model customization, which is explained in details in this thesis. The ciphertext execution modules were not implemented by the thesis author, yet we will briefly cover them for completeness.

5.4.1 Ciphertext-aware Quantization

Most contemporary ML libraries utilize 32-bit floating-point format (`FP32`) for data representation. In practice, the extremely high computational cost and complex circuits make `FP32` unsuitable for secure computation. Quantization addresses the aforesaid shortcomings by representing data in the integer format with a lower number of bits. Figure 5.4 demonstrates how `FP32` values can be converted to low-bit integers through quantization. Let us denote the signed integer format with b bits by `INT-b`. The mapping of an `FP32` parameter x_f to its `INT-b` representation x_q is computed as:

$$x_q = round(s \cdot x_f), \quad s = \frac{2^b - 1}{2 \times \max(|x_f|)} \tag{5.1}$$

where s is called the scaling factor and $\max(|x_f|)$ denotes the maximum range that parameter x_f can take. In a linear layer with `FP32` inputs X_f, weight parameters W_f, and bias $\mathbf{b}_f$, the output can be approximated using quantized values as:

$$Y_f = W_f \cdot X_f + \mathbf{b}_f \approx \frac{1}{s_w s_x}(W_q \cdot X_q + \frac{s_w s_x}{s_b}\mathbf{b}_q) \tag{5.2}$$

where s_x, s_w, and s_b denote the quantization scales for the input, layer weights, and the bias, respectively. The quantized version of Y_f is calculated using the corresponding scale s_y as follows:

$$Y_q = round\left(\frac{s_y}{s_w s_x}(W_q \cdot X_q + \frac{s_w s_x}{s_b}\mathbf{b}_q)\right) \tag{5.3}$$

Y_q is the quantized output of the linear layer which serves as the input of the next layer in a quantized DNN. While evaluating Eq. 5.3 is straightforward in plaintext, multiplication by the quantization scales $s = \frac{s_y}{s_w s_x}$ and $round(\cdot)$ incur significant costs in ciphertext. In what follows, we first introduce our highly efficient counterparts for these operations designed to minimize the secure execution cost. We then explain how we manage overflow in the low-bit regime.

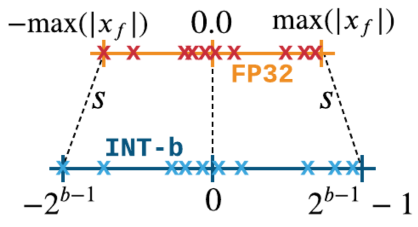


Figure 5.4: Quantizing FP32 values for INT-b representation.

Optimizing Scaling. In our framework, the matrix-multiplication $W_q.X_q$ as well as the addition with the bias vector are computed efficiently via AS. Scaling the result by $s = \frac{s_y}{s_w s_x}$ in AS would increases the overall multiplicative depth for computing $s(W_q \cdot X_q + \mathbf{b})$, increasing the AS computation bitwidth, thereby sacrificing the overall efficiency. To avoid bit-extending the matrix-multiplication operands, we separate scaling and evaluate it using GC ciphers. In this scenario, for scaling a b-bit number with a scale containing b' nonzero bits, the GC communication cost would be $2b(b'-1)\kappa$. Instead, we enforce the scale values to be powers of 2, which allows us to implement the previously costly scale operation with $\sim$*zero* cost logical shifts in GC. We do this by replacing the original formula in Eq. 5.1 with Eq. 5.4, and fine-tuning the DNN to adjust the quantization and preserve inference accuracy.

$$s = 2^{\left\lceil \log_2 \frac{2^b - 1}{2 \times \max(|x_f|)} \right\rceil} \tag{5.4}$$

Rounding Workaround. Let us consider an n-bit integer value, right shifted by $n-b$ bits through the scaling step to obtain a fixed-point value with b bits integer and $(n-b)$ bits fraction. Rounding operation in GC works by adding the MSB of the fraction with the b-bit integer. The GC cost is therefore equal to $2b \times \kappa$, which is quite significant considering it has to be repeated for all output elements across all DNN layers. To eliminate this cost, we replace $round(\cdot)$ with the floor operation $\lfloor \cdot \rfloor$ in our plaintext DNNs and fine-tune the model weights to adjust to this modification. Since flooring is equivalent to removing all fraction bits, it incurs no GC cost. To adapt the model weights to this modification, we use the original training data to fine-tune the model. This is done by applying flooring during the forward pass and straight-through gradients during the backward pass.

Overflow Management. Performing matrix-multiplication requires repeatedly updating an accumulator $y := y + \mathbf{w}[i]\mathbf{x}[i]$. An imminent challenge when moving to the low-bit quantized regime is the occurrence of overflow in the accumulator. To avoid overflows, existing ML libraries for quantization perform accumulations using high-precision data representations, e.g., y is `INT-32` while x and w are low-bit. In secure execution, high-precision accumulators are extremely costly. Therefore, we augment the underlying ML library with a new custom operation that simulates DNN execution with low-bit accumulators and directly models the occurrence of overflows. Eq. 5.5 presents our overflow simulation, which models the loss of MSB bits in case of overflow for an `INT-b` accumulator.

$$\begin{aligned}
\underset{x>0}{overflow(x)} &= \begin{cases} x\ mod\ 2^b, & (x\ mod\ 2^b) < 2^{b-1} \\ x\ mod\ 2^b - 2^b, & otherwise \end{cases} \\
\underset{x<0}{overflow(x)} &= \begin{cases} x\ mod\ 2^b, & (x\ mod\ 2^b) \geq -2^{b-1} \\ x\ mod\ 2^b + 2^b, & otherwise \end{cases}
\end{aligned} \tag{5.5}$$

Here, *mod* represents the modulo operation and $x\ mod\ 2^b$ checks for the occurrence of an overflow. In the forward pass (during DNN inference or training), the above operation is applied on all layer outputs to account for the occurrence of overflow according to the secure execution bitwidth. By leveraging the proposed overflow simulation, we accurately measure the secure execution accuracy in the presence of (occasional) overflows. Building upon our customized overflow simulation, we provide an automated strategy that finds the best allocation of bitwidths across DNN layers to minimize accuracy degradation, as will be discussed in Section 5.4.3.

Fine-tuning.. We further develop an overflow-aware training scheme which enables us to adjust the model parameters such that the adverse effect of overflow on inference accuracy is minimized. Since overflow simulation involves non-differentiable operations, we devise an approximate gradient for this function to allow fine-tuning of our quantized models. Let x be a scalar value, $\overline{x}$ denote its value after overflow, and $\nabla_{\overline{x}}$ be the gradient of the training loss function with respect to $\overline{x}$. We compute the gradient with respect to x as follows:

$$\nabla_x = \begin{cases} \nabla_{\overline{x}} & if \quad x = \overline{x} \\ 0 & otherwise \end{cases} \tag{5.6}$$

5.4.2 Factored Matrix-Multiplication

Matrix-multiplication accounts for the bulk of computations in DNN inference, which leads to a high communication cost in AS. Our goal in this section is to reduce this cost via factored matrix-multiplication, which replaces the majority of costly multiplications with cheaper conditional additions. Below, we introduce the building blocks of factored matrix-multiplication and explain our method in detail.

Consider a matrix-multiplication of the form $Y = W \cdot X$, where $Y \in \mathbb{R}^{M\times L}$, $W \in \mathbb{R}^{M\times N}$, and $X \in \mathbb{R}^{N\times L}$. This operation can be broken down into $M \times L$ VDPs, where each VDP operates on vectors of length N, $\mathbf{w} \in \mathbb{R}^N$ and $\mathbf{x} \in \mathbb{R}^N$, corresponding to a row of W and a column of X, respectively. Each VDP therefore requires N multiplications and N additions. We propose the factored VDP as the core operation in factored matrix-multiplication. We start with the definitions of the unique space and the encoded representation of vectors involved in VDP.

Definition 1. *The unique space of* $\mathbf{w} \in \mathbb{R}^N$ *is the set* $\mathbf{c} = \{c_1, \ldots, c_V\}$ *such that* $\mathbf{w}[i] \in \mathbf{c}$ $(\forall\, i \in [N])$. *We refer to* V *as the unique size of* $\mathbf{w}$.

Definition 2. *Given a vector* $\mathbf{w} \in \mathbb{R}^N$ *and its unique space* $\mathbf{c} = \{c_1, \ldots, c_V\}$*, the encoded representation of* $\mathbf{w}$ *is a vector of integer indices* $\widetilde{\mathbf{w}} \in [V]^N$ *such that* $\mathbf{w}[i] = \mathbf{c}[\widetilde{\mathbf{w}}[i]]$.

Knowing the unique space $\mathbf{c}$ and the encoded representation $\widetilde{\mathbf{w}}$, the factored VDP can be computed via V multiplications and $N+V$ additions: we first compute N conditional additions, each of which adds an input element to one of V accumulators based on its code:

$$\mathbf{s}[v] = \sum_{x\in\mathbb{S}_v} x, \qquad \mathbb{S}_v = \{\mathbf{x}[i] \mid \widetilde{\mathbf{w}}[i] = v\} \tag{5.7}$$

Next, a VDP is computed between the accumulated values and the unique space of $\mathbf{w}$, i.e., $\mathsf{VDP}(\mathbf{x},\mathbf{w}) = \mathsf{VDP}(\mathbf{s},\mathbf{c})$. The benefits of factored multiplication are most substantial when $V << N$[1]. In general, V can be as large as 2^b, where b is the quantization bitwidth of $\mathbf{w}$. Even

[1] N is in the order of 100-10000

after quantizing **w** with lower bitwidths, V can be quite large, e.g., $V = 64$ for 6-bit weights. To decrease V, we approximate apply weight encoding as described in Chapter 3 (3.3.1). The resulting encoded weights will have a unique space equal to the encoding codebook.

It is worth noting that the value of V directly affects the tradeoff between the DNN inference accuracy and the secure execution cost. Higher V values achieve higher accuracy but also incur higher secure execution cost. It is a great challenge to determine the per-layer V values and balance this trade-off such that the DNN is executed accurately and efficiently. To address this challenge, we provide an automated algorithm that specifies V for each linear layer in a desired DNN as will be discussed in Section 5.4.3. Once the codebook size is specified per layer, the model is finetuned to enhance the inference accuracy. The gradient of the loss function with respect to the weights and codebook values is obtained based on the same methodology explained in Chapter 3, (Section 3.3.2).

5.4.3 Automated Parameter Configuration

The quantization bitwidths and the unique spaces across different layers are not independent and they collectively determine the model accuracy as well as the secure execution cost. COINN is equipped with an automated parameter configurator that searches for the optimum number of quantization bits and weight clusters across DNN layers such that: (1) the secure execution cost is minimized and (2) a user-defined constraint on inference accuracy is met. COINN configurator initially reduces the optimization space, and then uses our customized optimizer and score function to find the optimal DNN. The configurator performs the above process separately for quantization and matrix factorization. Below we explain each component of COINN configurator in detail.

Optimization Space Reduction. For quantization, the bitwidths for the input (b_{inp}), weights (b_w), and the activation (b_{acc}) should be configured at each linear layer. Finding the optimal quantized DNN is therefore equivalent to searching over a parameter space containing

$B^{3\mathscr{L}}$ different network configurations where $\mathscr{L}$ and B denote the number of linear layers and the maximum bitwidth budget[2], respectively. Finding the best parameter configuration in such a large space is very time-consuming and the final obtained DNN configuration is often sub-optimal. We observe that many of the bitwidth configurations in this search-space violate the user-defined accuracy constraint. Therefore, prior to finding the optimal bitwidths, we first identify and eliminate the invalid bitwidth configurations from the search space.

This process is performed on a per-layer basis: for each linear layer in the network, we discard the subset of its corresponding quantization bitwidths that violate the accuracy constraint. In doing so, we keep the remaining layers in full-precision format. Note that such per-layer analysis allows us to shrink the original search space but does not determine the optimal bitwidth configuration across all layers. This is due to the fact that the per-layer analysis does not reflect the effect of inter-layer correlations on inference accuracy when all DNN layers are simultaneously quantized. We therefore devise an optimizer to search the reduced parameter space obtained from the per-layer analysis to find the optimal bitwidth configuration across all layers.

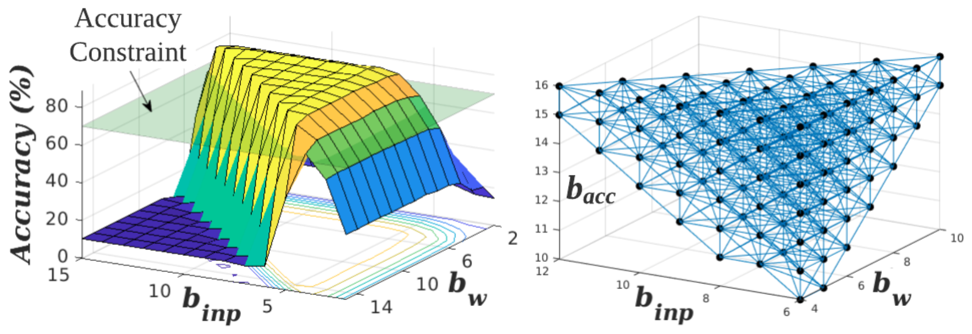


Figure 5.5: (left) Inference accuracy versus the input and weight bits of a CONV layer in an example DNN. (right) $3D$ visualization of the layer's valid bitwidth configurations.

Figure 5.5-(left) demonstrates the model accuracy versus the input and weight bitwidths for one layer of an example DNN when the activation bit is set to the maximum value ($b_{acc} = 16$). As seen, due to the occurrence of overflow, many of the configurations fall below the accuracy constraint plane. Using this intuition, we construct a $3D$ mesh of valid bitwidth configurations

[2]In our experiments we set $B = 16$.

that comply with the accuracy constraint for each layer as shown in Figure 5.5-(right). Each node corresponds to a tuple (b_{inp}, b_w, b_{acc}) and its neighbors are nodes with a maximum bit distance of 1. As seen in this example, the search space for one layer is reduced to $\sim \frac{1}{8}$, which provides a lot of saving for the overall DNN, i.e., $\sim (\frac{1}{8})^{\mathscr{L}}$. Our optimizer then traverses this mesh to find the optimal DNN configuration.

For clustering, the per-layer configuration comprises only one parameter, i.e., the unique size V, which undergoes a similar process for identifying the valid optimization space.

Optimizer. We develop a novel genetic algorithm [126] with customized graph operations to traverse our constructed mesh of valid configurations and find the optimal quantized/clustered DNN. Our genetic algorithm operates on a *population* of *individual*s where each individual corresponds to a candidate DNN configuration. Optimization is performed iteratively and the population is gradually evolved to obtain better DNN configurations that have higher accuracy and/or lower secure execution cost. At each iteration, all members of the current population are evaluated in terms of the secure execution cost and the inference accuracy. We utilize a customized score function to combine these two (conflicting) metrics and assign a measure of optimality to each individual. We then perform a random selection from the population where individuals with higher scores have higher chances of being selected. Each selected individual is then randomly tweaked by moving along the configuration mesh to adjacent neighbor nodes. This is equivalent to performing small-scale changes in the model architecture to explore new (unseen) configurations and find the optimal DNN.

Score Function. The objective of parameter optimization for secure inference is to minimize the secure execution cost while enforcing the inference accuracy to be higher than a user-defined threshold (constraint). The objective of this constrained optimization can be embedded into a single score function that absorbs both accuracy and secure execution cost. Let us denote the DNN configuration (quantization/clustering parameters) as $\mathbf{p} \in \mathbb{R}^d$ and the corresponding accuracy and secure execution cost as $\mathscr{A}(\mathbf{p})$ and $\mathscr{C}(\mathbf{p})$, respectively. For a given

DNN configuration, the secure execution cost $\mathscr{C}(\mathbf{p})$ is the cumulative per-layer costs calculated using Table 5.1 and the accuracy $\mathscr{A}(\mathbf{p})$ is measured on a held out validation dataset. We adapt the score function from ML-customization literature [48], which use fractions and an exponential penalty function [127] to enforce the inference accuracy constraint. Our score function is defined as:

$$\mathscr{S}(\mathbf{p}) = \frac{\mathscr{C}_{max} - \mathscr{C}(\mathbf{p})}{\xi(\mathscr{A}(\mathbf{p}))}, \tag{5.8}$$

where $\mathscr{C}_{max}$ is the execution cost of the reference DNN prior to optimization. The numerator of the score function encourages minimization of the ciphertext execution cost $\mathscr{C}(\mathbf{p})$ and the denominator $\xi(\cdot)$ enforces a strict lower bound (threshold) for the accuracy using exponential penalty methods [127, 128] as follows:

$$\xi(\mathscr{A}(\mathbf{p})) = \begin{cases} \mathscr{A}_{max} - \mathscr{A}(\mathbf{p}) & \mathscr{A}(\mathbf{p}) > \mathscr{A}_{min} \\ \mathscr{A}_{max} - \mathscr{A}(\mathbf{p}) + e^{\mathscr{A}(\mathbf{p}) - \mathscr{A}_{min}} & otherwise \end{cases} \tag{5.9}$$

where $\mathscr{A}_{max}$ is the accuracy of the reference point DNN. As seen, $\xi(\cdot)$ puts a linear penalty on points with a high accuracy but exponentially increases the penalty when the accuracy drops below the lower bound $\mathscr{A}_{min}$. As we show in our experiments, this score function ensures that our genetic algorithm finds a DNN configuration that has significantly lower ciphertext cost compared to the baseline (plaintext) model with comparable accuracy.

5.5 Oblivious Inference

In this section we provide a high-level view of COINN fast and efficient oblivious DNN inference. Figure 5.6 illustrates operations in plaintext DNN execution and their realization in ciphertext domain. In what follows, we describe how each DNN layer is securely computed in COINN in more detail.

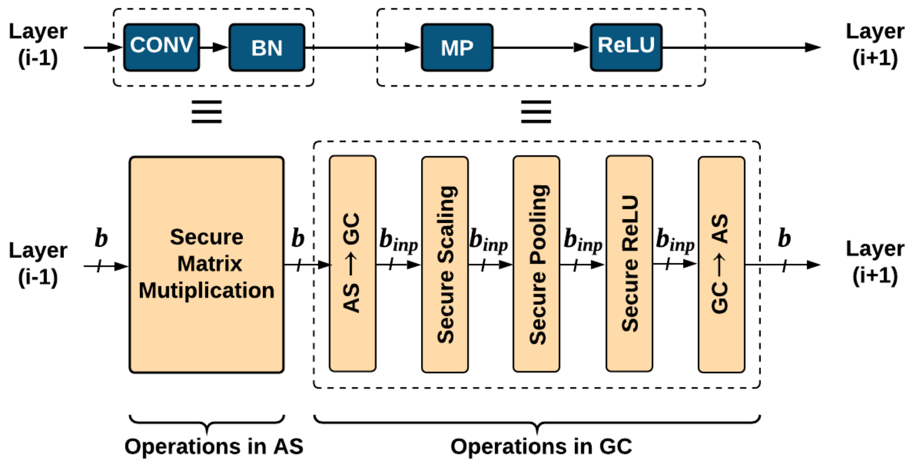


Figure 5.6: Plaintext DNN layers and their equivalent ciphertext realization.

Convolution/Fully-Connected. The CONV/FC operation is performed via our efficient AS-based matrix-multiplication method over secret shared data that is sign-extended to match the bitwidth of the accumulator b_{acc}. Based on the optimal unique size allocated to each layer's weights by the model configurator, our API automatically performs the pertinent secure matrix-multiplication via regular of factored operations, to maximize efficiency.

Batch Normalization. Recall that BN operates on the output of its preceding CONV layer, i.e., $Y \in \mathbb{R}^{M\times L}$, It multiplies each row by α_i and adds β_i to the result. A Naive implementation of the BN would treat this layer independently which incurs a non-negligible secure execution cost. Instead, we fuse the BN operation into the preceding CONV layer so that the combination of CONV +BN can be realized via a single matrix-multiplication. The i-th row of Y is originally computed in the preceding CONV layer as $Y_i = W_i \cdot X + \mathbf{b}_i$. Application of BN on this row vector renders:

$$\begin{aligned} BN(Y_i) &= \alpha_i Y_i + \beta_i \\ &= \alpha_i(W_i \cdot X + \mathbf{b}_i) + \beta_i \\ &= \alpha_i W_i \cdot X + \alpha_i \mathbf{b}_i + \beta_i \end{aligned} \tag{5.10}$$

We thus remove the BN layer and set the preceding CONV's weight matrix rows to $\alpha_i W_i$ and bias values to $\alpha_i \mathbf{b}_i + \beta_i$.

Average Pooling. Average pooling works by computing the sum over $k \times k$ windows of convolution outputs and dividing the summation result by k^2. Similar to fusion of BN with CONV, we can avoid division by k^2 by simply dividing the weight and bias of the preceding layer by k^2, and computing the sum instead of average values in the pooling layer. In our setting, we perform average pool layers in AS as summation is free in his protocol.

Scaling, ReLU, and Max-Pooling. Once the linear operations are computed in AS, we convert the data back to the GC domain to make it amenable for nonlinear operations such as ReLU and MP. After conversion to GC, the b_{acc}-bit streams are scaled down to b_{inp}^{i+1} bits, where b_{inp}^{i+1} is the number of input bits for the $i+1$-th linear layer. The two parties then run the required GC protocols to compute the pooling and ReLU operations. To prepare the resulting ciphertext values with b_{inp}^{i+1} bits for linear operations, COINN securely converts the GC data to AS representation, where the integers are sign-extended to match the bitwidth of the accumulator in the CONV/FC layers b_{acc}.

The (amortized) communication cost associated with each of the layers explained above is shown is Table 5.1. The automated design customization tool (Section 5.4.3) incorporates these costs inside the genetic algorithm score to search for the optimal configuration across the layers.

Table 5.1: COINN secure execution cost for core operations in a DNN. Here, κ is the security parameter that is set to 128.

Operation	**in$_{\text{dim}}$ $\rightarrow$ out$_{\text{dim}}$**	**Ciphertext Cost**	**Parameters**
Mal-Mult (regular)	$X_{x\times l} \xrightarrow{W_{m\times n}} Y_{m\times l}$	$\mathcal{O}\big(mnlb(b+1)\big)$	b: accumulator bitwidth
Mat-Mult (factored)	$X_{x\times l} \xrightarrow{W_{m\times n}} Y_{m\times l}$	$\mathcal{O}\big(mvlb(n+b+1)\big)$	b: accumulator bitwidth v: unique size of W
MaxPool	$X_{c\times d_1\times d_1} \xrightarrow{k\times k} Y_{c\times d_2\times d_2}$	$\mathcal{O}\big(4\kappa c d_2 d_2 (k^2-1)b\big)$	b: next layer input bitwidth k: pooling window size
ReLU	$X_{c\times d_1\times d_1} \xrightarrow{>0} Y_{c\times d_1\times d_1}$	$\mathcal{O}\big(2\kappa c d_1 d_1 b\big)$	b: next layer input bitwidth κ: security parameter (=128)
AS $\rightarrow$ GC	$X_{c\times d_1\times d_1} \rightarrow Y_{c\times d_1\times d_1}$	$\mathcal{O}\big(5\kappa c d_1 d_1 b\big)$	b: accumulator bitwidth κ: security parameter (=128)
GC $\rightarrow$ AS	$X_{c\times d_1\times d_1} \rightarrow Y_{c\times d_1\times d_1}$	$\mathcal{O}\big(3\kappa c d_1 d_1 b\big)$	b: next layer input bitwidth κ: security parameter (=128)

5.6 Experiments

In this section, we empirically evaluate the performance of COINN in various settings. We perform a detailed study of the efficiency gains achieved by each of COINN optimizations, namely, quantization, clustering, and end-to-end parameter configuration, in Section 5.6.1. Next, we provide a side-by-side comparison of COINN with recent works in Section 5.6.2, in terms of the ciphertext execution time, showing $4.7\times$–$36.8\times$ faster inference on contemporary DNNs in LAN setting. We further show that COINN achieves better performance compared to prior work in the high-latency setting.

Evaluation Setup. We use the PyTorch library for training the FP32 DNNs and develop our security-aware quantization, clustering, and automated parameter configuration with PyTorch backend for easy utilization by the community. Our ciphertext execution uses OT, and CS-PRNG implementations from EMP-toolkit [129] and GC implementation from TinyGarble2 [130]. For fast matrix-multiplication, we utilize the Intel intrinsic instructions and represent matrices with the Eigen library [131].

Table 5.2: COINN benchmarks.

Model	Layers	Acc	MACs	Params
MiniONN [116]	6 CONV, 1 FC, 2 MP, 6 ReLU	88.3	6.1e7	1.6e5
ResNet32	31 CONV, 1 FC, 1 AP, 31 ReLU	68.7	6.9e7	4.7e5
ResNet110	109 CONV, 1 FC, 1 AP, 109 ReLU	94.1	2.5e8	1.7e6
ResNet50	49 CONV, 1 FC, 1, MP, 1 AP, 49 ReLU	76.1	4.1e9	2.5e6

We run our ciphertext evaluations using 4 threads on machines with 2.2 GHz Intel Xeon CPU and 16 GB RAM. For runtime measurements, we consider two real-world network settings, namely LAN with a throughput of 1.25 GBps, round trip time of 0.25ms, and WAN with a throughput of 125 MBps, round trip time of 100ms. We simulate the network settings via Linux Traffic Control[3].

[3]`https://man7.org/linux/man-pages/man8/tc.8.html`

Benchmarks. We perform evaluations on the CIFAR-10, CIFAR-100, and ImageNet classification benchmarks. The number of classes in these datasets is 10, 100, and 1000, respectively. Table 5.2 presents details of our benchmarked DNNs along with their `FP32` accuracy. We evaluate the 7-layer network from MiniONN [132] and ResNet110 on CIFAR-10, ResNet32 on CIFAR-100, and ResNet50 on ImageNet dataset. Our DNN benchmarks cover a wide range of parameter sizes (0.5M to 23M) and number of MAC operations (60M to 4B) commonly observed in real-world models.

Accuracy Measurement. Throughout the evaluations, we report the secure model accuracy, which is measured efficiently (and correctly) by simulating ciphertext operations in PyTorch. The correctness is validated by matching all DNN layers' activations in secure inference with those from PyTorch on randomly selected inputs.

5.6.1 Evaluation of COINN Optimizations

In this section, we provide a breakdown of the savings in secure execution cost as a result of COINN's model adjustment methods.

Low-Bit Heterogeneous Quantization. We illustrate the benefits of our quantization scheme in reducing the secure communication cost, while maintaining accuracy, for a large scale real-world DNN – ResNet32. Figure 5.7 presents the communication cost and accuracy of secure execution as a function of the bitwidth. The numerical labels on the horizontal axis represent homogeneous quantization (equal bitwidths across all layers), where each label is $b_{inp} = b_w$ with b_{acc} set to $2b_{inp} + 1$. The label 16 represents the configuration implemented in prior works [2, 6] which we use as a baseline. Figure 5.7 shows that while reducing the bitwidth in the homogeneous setting results in a linear reduction of ciphertext communication, it also results in a significant drop in accuracy.

To mitigate the undesireable accuracy drop of homogeneous quantization, our automated parameter configurator finds a heterogeneous allocation of per-layer bitwidths that simultaneously

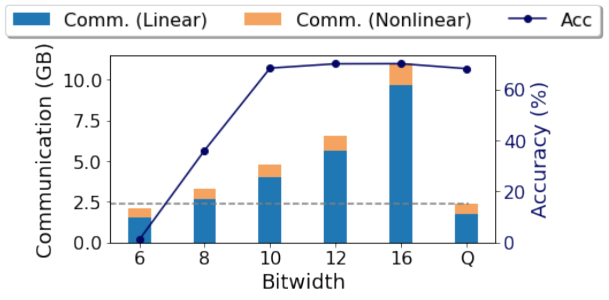


Figure 5.7: Effect of quantization bitwidth on communication cost (bars) and accuracy (curve). The numbers on the horizontal axis show the bitwidth for homogeneous quantization of weights/inputs across all layers. Label Q represents the heterogeneous bitwidths found by COINN.

ensures high accuracy and low communication cost. The rightmost label, Q in Figure 5.7, represents the COINN optimized model with heterogeneous quantization bitwidths across layers. This optimal set of bitwidths results in a communication cost equivalent to the 6-bit homogeneous model and achieves an accuracy comparable to the 16-bit baseline. Such optimization of per-layer bitwidths is made possible via our secure computation-aware quantization which accurately simulates the effect of low-bit quantization in ciphertext. This allows us to explore the trade-off between communication cost and model accuracy. We present the heterogeneous bitwidths found by COINN configurator for ResNet32 in Figure 5.8-a.

Factored Matrix-Multiplication. Figure 5.7 shows that the bulk of total communication cost in a quantized model corresponds to linear operations. We now showcase how COINN further reduces this cost via factored matrix-multiplication. Figure 5.10 presents the communication cost and accuracy as a function of the number of unique elements in each layer's weight matrices V. The label Q represents our model with heterogeneous quantization bitwidths from Figure 5.7. The numeric labels to its left represent models with a uniform selection of V across all layers. Such naïve selection results in accuracy degradation, particularly for small V. Our automated parameter configurator finds a heterogeneous allocation of V across DNN layers that balances the tradeoff between inference accuracy and ciphertext communication. The result is an optimal DNN represented with the label Q+C that reduces the secure communication cost of the quantized

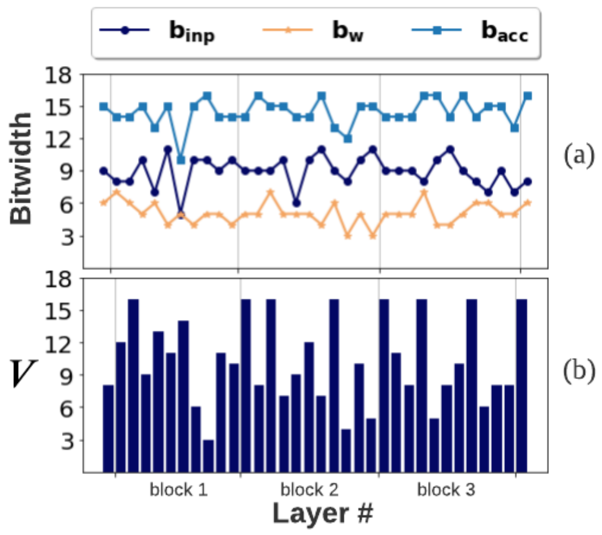


Figure 5.8: Heterogeneous parameters across ResNet-32 layers found by COINN configurator. (a) Quantization bitwidths. (b) Number of clusters V.

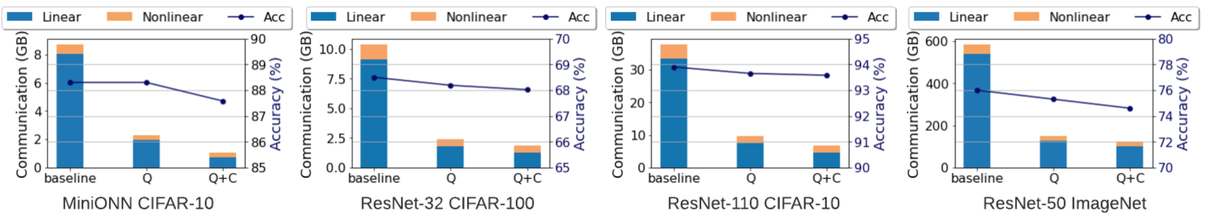


Figure 5.9: Communication for baseline and COINN optimized models, where Q represents quantized model and Q+C further applies clustering to enable factored multiplication.

model by 1.4× while maintaining the original model accuracy. We present the heterogeneous number of per-layer clusters found by our configurator for this benchmark in Figure 5.8-b.

Holistic Optimization. Figure 5.9 presents the reduction in communication cost achieved by applying COINN automated quantization and clustering on all benchmarks. As our baseline design, we adopt the bitwidths from prior work [2], i.e., 16-bit inputs/weights and 32-bit activations, and perform regular matrix-multiplication. For COINN results, we first find heterogeneous quantization configurations using our genetic algorithm and fine-tune the model to regain accuracy. We show the optimized quantized model via Q on Figure 5.9. Next, we use our automated parameter configurator to find the weight clusters for each layer and fine-tune the

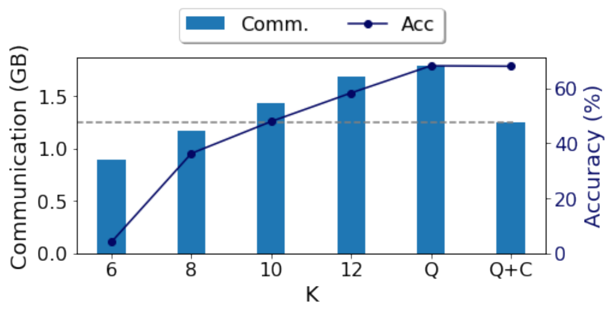


Figure 5.10: Effect of factored multiplication on inference accuracy and communication cost of linear operations. The label Q on the horizontal axis shows the baseline quantized DNN. The numbers to its left represent the homogeneous V used to cluster all layer weights. The label Q+C stands for the heterogeneous V configuration found by COINN.

resulting model once more to obtain the DNN labeled Q+C. The linear operations in the Q and Q+C settings are performed via regular and factored Matrix-Multiplication, respectively. As seen, by finding the best set of heterogeneous bitwidths across DNN layers, COINN successfully reduces the secure communication for linear and nonlinear layers by $3.9\times$–$4.3\times$ and $1.9\times$–$2.2\times$, respectively. By optimizing the weight clusters, we further push the efficiency gains on linear layers to $4.8\times$–$8.1\times$.

Table 5.3 provides the total runtime and communication cost of our baseline, Q, and Q+C configurations in both LAN and WAN settings. The evaluation verifies the effect of our optimization on the runtime: applying Q+C reduces the baseline runtime by $2.6\times$–$3.9\times$ and $2.3\times$–$4.2\times$ in LAN and WAN settings, respectively.

5.6.2 Comparison with Prior Work

In this section, we compare COINN amortized runtime with the prior art in oblivious inference.. In Table 5.4, we report the performance of COINN along with four contemporary works, namely, XONN [4] with extremely low-bit (binary) weights/activations, Delphi [2] with a hybrid HE-GC protocol, SafeNet [3] which perform ML optimization for Delphi's secure protocol, and CrypTFlow2 [6] which is the current state-of-the-art in oblivious inference. For a fair and

Table 5.3: Evaluation of COINN in LAN and WAN settings. Q and C denote quantization and clustering, respectively.

Model	Comm. (GB)			LAN Time (s)			WAN Time (s)		
	Base	Q	Q+C	Base	Q	Q+C	Base	Q	Q+C
MiniONN	8.7	2.3	1.0	4.85	1.9	1.45	74.6	26.5	18.5
Res32	10.4	2.4	1.9	9.8	3.8	3.68	143.9	67.1	62.9
Res110	37.6	9.7	6.8	36.0	14.2	14.0	518.1	242.8	226.0
Res50	583.1	148.0	122.0	571.46	165.3	145.7	4994	1420.4	1189.7

accurate comparison, we re-run the open-source codes provided by Delphi[4] and CrypTFlow2[5] to obtain runtime/communication measurements on our machines. For the remaining works [3,4], we directly report the numbers from the original papers since no public code was available.

Table 5.4 shows COINN achieves 4.7×–36.8× faster ciphertext execution in the LAN setting compared to prior work. Even though in the high latency setting the benefit margins are smaller, COINN still outperforms the best methods to date. This is achieved by optimizing both non-linear and linear computations/communications through quantization and factored multiplication. Furthermore, COINN achieves 0.6%– 4.7% higher accuracy with 23.1×–36.8× faster secure runtime compared to prior crypto/ML co-optimization work, namely [2–4].

Evaluation on Large-scale Benchmarks. To fully demonstrate the efficacy and scalability of COINN model adjustment techniques and custom secure protocols, we evaluate two exceptionally complex DNNs, namely, ResNet110 on CIFAR-10 and ResNet50 on ImageNet datasets. The first benchmark, i.e., ResNet110, is challenging due to the extremely high dimensionality of the parameter configuration space: there are 330 bitwidths and 110 clustering parameters that require per-layer adjustment. The second benchmark, i.e., ResNet50, is the largest DNN ever studied in the secure computation domain with over 4 Billion scalar multiplications and additions.

In Table 5.4, we present the runtime for the large scale networks and compare our

[4]`https://github.com/mc2-project/delphi`
[5]`https://github.com/mpc-msri/EzPC/tree/master/SCI`

Table 5.4: Performance comparison of COINN with best prior work. "Improv." shows the improvement in total runtime. CTF2 refers to CrypTFlow2 [6].

		LAN		**WAN**		**Acc.**
		Runtime (s)	Improv.	Runtime (s)	Improv.	**(%)**
MiniONN	XONN	33.5	23.1×	-	-	83.0
	Delphi	49.9	34.4×	59.8	3.2×	82.9
	SafeNet	53.4	36.8×	-	-	85.1
	CTF2 (HE)	20.8	14.4×	55.4	3.0×	86.0
	CTF2 (OT)	11.9	8.2×	108.2	5.8×	86.0
	COINN	1.45	1×	18.5	1×	87.6
Res32	Delphi	88.8	24.0×	145.9	2.3×	65.7
	SafeNet	128.0	34.6×	-	-	67.5
	CTF2 (HE)	32.6	8.8×	136.9	2.2×	68.0
	CTF2 (OT)	18.7	5.1×	176.7	2.8×	68.0
	COINN	3.7	1×	62.9	1×	68.1
Res110	CTF2 (HE)	110.3	7.8×	448.2	2.0×	94.1
	CTF2 (OT)	65.4	4.7×	579.3	2.6×	94.1
	COINN	14.0	1×	226.0	1×	93.4
Res50	CTF2 (HE)	893.2	6.1×	1463.3	1.2×	76.1
	CTF2 (OT)	1139.8	7.8×	4241.8	3.6×	76.1
	COINN	145.7	1×	1189.7	1×	73.9

results with the state-of-the-art CrypTFlow2. In the LAN setting, COINN achieves 4.7×–7.8× and 6.1×–7.8× runtime improvement compared to CrypTFlow2's OT-based and HE-based implementations, respectively. In the WAN setting, COINN achieves 2.6×–3.6× and 1.2×–2× runtime improvement compared to CrypTFlow2's OT-based and HE-based implementations, respectively. It is worth noting that the relatively lower improvement margin achieved by COINN in one specific setting (1.2× for ResNet50, WAN, HE) is due to the heavy imbalance of the cost towards linear layers in this particular benchmark.

5.7 Conclusion

We present COINN, an oblivious DNN inference framework that outperforms state of the art in both accuracy and efficiency. Through a unique combination of complimentary optimizations in ML and crypto domains, COINN brings us one step closer to real life deployment of contemporary DNNs in the privacy-preserving setting. The enhanced performance of COINN roots in three innovations, namely, ciphertext-aware quantization, enhanced data reuse, and automated parameter configuration. Our contributions in the plaintext are accompanied by efficient custom cryptographic protocols. We performed rigorous empirical analysis on every step of our optimization process to demonstrate their effect on reducing the secure communication and oblivious inference runtime. Our evaluations on practical DNN benchmarks showed an end-to-end runtime speedup of $1.2\times$– $14.4\times$ over the best prior work.

5.8 Acknowledgment

The material in this chapter is pending publication and will appear as “Siam U Hussain, Mojan Javaheripi, Mohammad Samragh, and Farinaz Koushanfar. COINN: Crypto/ML Codesign for Oblivious Inference via Neural Networks. ACM Conference on Computer and Communications Security (CCS), 2021.” The thesis author along with Dr. Siam Hussain and Mojan Javaheripi were equal contributors to this material.

Chapter 6

Customizing binary neural networks for secure DNN inference

In this chapter, we study BNN as a candidate for fast and scalable oblivious inference. We show that a BNN has several unique characteristics that allow translating its computations to simple and efficient cryptographic protocols.

The benefits of employing BNNs for oblivious inference were first noted by XONN [4]. Despite achieving significant runtime improvement compared to non-binary DNN inference, there are opportunities provided by BNNs that have not been leveraged by XONN. Part of the inefficiency of XONN is due to the usage of a single secure computation protocol as a blackbox for all neural network layers after the input layer. In this work, we introduce a new hybrid approach where the underlying secure computation protocol is customized to each layer, such that the total execution cost for oblivious inference on all layers is minimized. We design a composite custom secure execution protocol, specifically optimized for BNN operations, using standard security primitives. Our protocol significantly improves the efficiency of XONN as we show in our experiments.

One standing challenge in oblivious inference is finding architectures that are both accurate

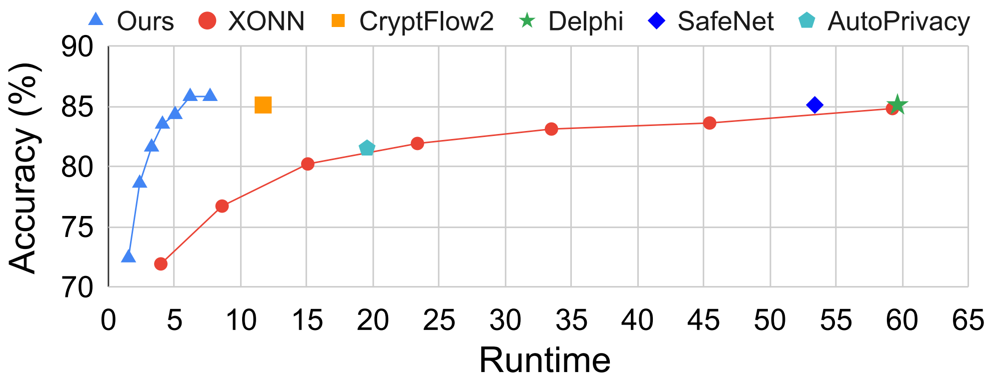


Figure 6.1: Accuracy and runtime of our oblivious BNN inference, compared with contemporary research that have the same server-client scenario setting as us (two-party, honest but curious). Among these, XONN [4] evaluates BNNs, whereas Cryptflow2 [6], Delphi [2], SafeNet [3], and AutoPrivacy [5] evaluate non-binary models.

and amenable to secure computation. Since BNNs suffer from long training time and poor convergence, searching for such architectures could be quite inefficient. We address the search inefficiency challenge by training a *single BNN* that can operate under different computational budgets. Our adaptive BNN offers a tradeoff between accuracy and inference time, without requiring to train separate models. Figure 6.1 presents the tradeoff achieved by our flexible BNN on the 7-layer VGG network trained on CIFAR-10. With the combined power of our custom oblivious inference protocols and adaptive BNN training schemes, our method outperforms prior art both in terms of accuracy and runtime. Our solution is $\sim 2\times$ faster than Cryptflow2 [6], the state-of-the-art non-binary DNN inference framework, and $2\times$ to $11\times$ faster than XONN, the previous oblivious BNN inference framework.

6.1 Cryptographically Secure BNN Inference

BNNs were originally introduced to minimize memory footprint and computation overhead of plaintext inference. In this section, we provide insights on why BNNs are also useful for very efficient and fast oblivious inference.

The first favorable property of BNNs is enforcing the weights to +1 or -1. With this

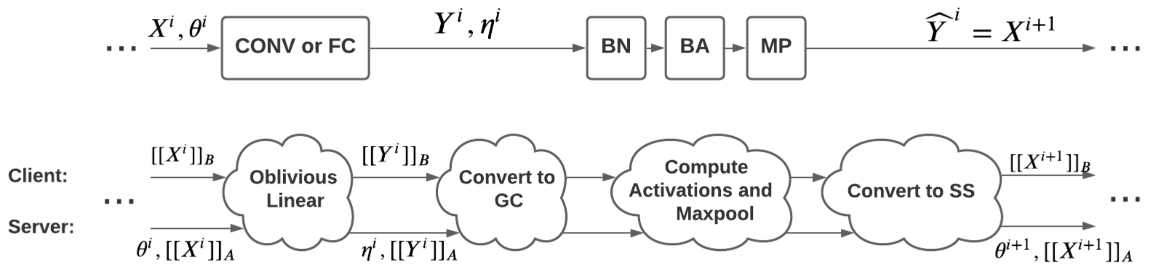


Figure 6.2: Illustration of plaintext inference (top) and our proposed equivalent oblivious inference (bottom). We denote linear layers by *CONV* and *FC*, Batch-Normalization by *BN*, Binary Activation by *BA*, and Max-Pooling by *MP*. Here, X^i, and Y^i, and θ^i are the input, output, and weight/bias parameters of the linear layer, respectively. η^i denotes BN parameters, and $\widehat{Y}$ is the output of binary activation.

restriction, multiplying a feature x by a weight w is equivalent to computing either $+x$ or $-x$. This simple property becomes useful when computing vector dot products of the form $\sum_{i=1}^{N} w_i x_i$, which can be computed via N conditional additions/subtractions. We show in Section 6.1.1 that conditional summations can be computed using OT and AS, both of which are known to be very efficient and light-weight cryptographic tools.

In oblivious inference, nonlinear operations are evaluated through heavy cryptographic primitives such as GC, resulting in large runtime and communication overheads. The large communication cost of GC is directly related to the bit-widths of GC inputs. The second advantage of BNNs is their 1-bit hidden layer feature representation, which significantly reduces the GC evaluation cost when compared to non-binary features. In Section 6.1.2, we expand on low-bit nonlinear operations and their efficient GC evaluation.

This section provides a high-level outline of the necessary terminologies in secure function evaluation between a client and a server. Following the convention in secure computation literature, we refer to server and client as Alice and Bob, respectively.

We present the overall flow for oblivious BNN inference in Figure 6.2. The inputs and outputs of all layers are in AS format, e.g., server and client have $[\![Y^i]\!]_A$ and $[\![Y^i]\!]_B$ rather than Y^i. To obliviously evaluate linear layers (CONV or FC), we propose a novel custom protocol for binary matrix multiplication that directly works on AS data. We merge batch normalization (BN), binary activation (BA), and max-pooling (MP) into a single nonlinear function $f(\cdot)$. To securely evaluate $f(\llbracket Y^i \rrbracket_A, \llbracket Y^i \rrbracket_B)$, three consecutive steps should be taken:

1. Securely translating the input from AS to GC. This step prepares the data to be processed by GC.
2. Computing the nonlinear layer through GC protocol.
3. Securely translating the result of the GC protocol to AS. This step prepares the data to be processed in the following linear layer.

Using this hybrid approach, we achieve a significantly faster oblivious inference compared to the state-of-the-art [4].

6.1.1 Linear Layers

Fully-connected and convolutional layers require computing $Y = WX$, with weight matrix W and input X. In secure matrix multiplication, the input is secret shared between the server and the client, i.e., $X = \llbracket X \rrbracket_A + \llbracket X \rrbracket_B$. Bob (the client) has $\llbracket X \rrbracket_B$ whereas Alice (the server) has the weight W and $\llbracket X \rrbracket_A$[1]. The matrix multiplication is computed as follows:

$$W(\llbracket X \rrbracket_A + \llbracket X \rrbracket_B) = W\llbracket X \rrbracket_A + W\llbracket X \rrbracket_B \tag{6.1}$$

Alice can compute $W\llbracket X \rrbracket_A$ locally and only $W\llbracket X \rrbracket_B$ needs secure evaluation. After evaluating $Y = WX$,

- Alice gets $\llbracket Y \rrbracket_A$ but does not learn $\llbracket X \rrbracket_B$ or $\llbracket Y \rrbracket_B$.
- Bob gets $\llbracket Y \rrbracket_B$ but does not learn W or $\llbracket Y \rrbracket_A$.

[1] At the first layer, only client has the input share, hence $\llbracket X \rrbracket_A = 0$

The above computation is performed in in two phases: (1) the setup phase, shown in Algorithm 5, where Alice and Bob perform ROTs. Note that the setup phase only depends on the weight matrix which remains unchanged over a large number of inferences. Therefore this phase is performed only once and the cost is amortized among all future oblivious inferences. (2) the inference phase, shown in Algorithm 5, which is performed separately for each inference. Initially, Alice sets her output share to $W[\![X]\!]_A$ (line 1) and Bob sets his share to zero (line 2). Next, they obliviously evaluate $W[\![X]\!]_B$ one row at a time in the outer loop of Algorithm 5 (lines 3-15). Specifically, the m-th iteration of the outer loop evaluates the m-th row of the output as:

$$y_n = [\![y_n]\!]_A + [\![y_n]\!]_B = \sum_{n=1}^{N} W[m,n]X[n,:]$$

The inner loop of Algorithm 5 (lines 6-12) computes the above summation by running OT for $n \in [N]$. After each OT invocation, Alice receives either $\mu_0 = r - [\![X[n,:]]\!]_B$ or $\mu_1 = r + [\![X[n,:]]\!]_B$ depending on the selection bit. It is easy to see that μ_i (known by Alice) and $-r$ (known by Bob) are the arithmetic shares of $W[m,n][\![X[n,:]]\!]_B$.

Algorithm 4 One-time setup for secure matrix-multiplication.

```
Input:
    from Alice $W \in \{-1,+1\}^{M\times N}$
Output:
    to Alice $K_A \in \mathbb{Z}^{M\times N}$
    to Bob $\{K_B^0 \in \mathbb{Z}^{M\times N}, K_B^1 \in \mathbb{Z}^{M\times N}\}$
Remark:
    $K_A[m,n] = \begin{cases} K_B^0[m,n] & if\ \ W[m,n] = -1 \\ K_B^1[m,n] & if\ \ W[m,n] = 1 \end{cases}$
1: for $m \in [M]$ do
2:     for $n \in [N]$ do Alice and Bob engage in ROT where:
3:         • Alice inputs $i = \frac{W[m,n]+1}{2}$
4:         • Alice receives $K_A[m,n]$
5:         • Bob receives $\{K_B^0[m,n], K_B^1[m,n]\}$
6:     end for
7: end for
```

Algorithm 5 Secure binary matrix multiplication.

```
Input:
        from Alice $W \in \{-1,+1\}^{M\times N}$
        from Alice $K_A \in \mathbb{Z}^{M\times N}$
        from Alice $[\![X]\!]_A \in \mathbb{Z}^{N\times L}$
        from Bob $K_B^0 \in \mathbb{Z}^{M\times N}$
        from Bob $K_B^1 \in \mathbb{Z}^{M\times N}$
        from Bob $[\![X]\!]_B \in \mathbb{Z}^{N\times L}$
Output:
        to Alice $[\![Y]\!]_A \in \mathbb{Z}^{M\times L}$
        to Bob $[\![Y]\!]_B \in \mathbb{Z}^{M\times L}$
Remark:
        $j$ is the number of inferences so far
        $[\![Y]\!]_A + [\![Y]\!]_B = W([\![X]\!]_A + [\![X]\!]_B)$
 1: Alice locally sets $[\![Y]\!]_A = W[\![X]\!]_A \in \mathbb{Z}^{M\times L}$
 2: Bob locally sets $[\![Y]\!]_B = \mathbf{0} \in \mathbb{Z}^{M\times L}$
 3: for $m \in [M]$ do
 4:     Alice locally sets $[\![y]\!]_A = [\![Y(m,:)]\!]_A$
 5:     Bob locally sets $[\![y]\!]_B = [\![Y(m,:)]\!]_B$
 6:     for $n \in [N]$ do
 7:         Bob generates random vector $r \in \mathbb{Z}^L$
 8:         Bob computes: $\begin{cases} v_0 = H(j, K_B^0[m,n]) \oplus (r - [\![X[n,:]]\!]_B) \\ v_1 = H(j, K_B^1[m,n]) \oplus (r + [\![X[n,:]]\!]_B) \end{cases}$
 9:         Bob sends $v_0, v_1$ to Alice
10:         Knowing $i = \frac{W[m,n]+1}{2}$, Alice computes: $\mu_i = H(j, K_A[m,n]) \oplus v_i$
11:         Alice locally updates $[\![y]\!]_A = [\![y]\!]_A + \mu_i$
12:         Bob locally updates $[\![y]\!]_B = [\![y]\!]_B - r$
13:     end for
14:     Alice locally updates $[\![Y(m,:)]\!]_A = [\![y]\!]_A$
15:     Bob locally updates $[\![Y(m,:)]\!]_B = [\![y]\!]_B$
16: end for
```

6.1.2 Nonlinear Layers

In this section, we outline and leverage characteristics of BNNs for oblivious inference of nonlinear layers. The cascade of batch normalization (BN) and binary activation (BA) takes input feature y and returns $\widehat{y} = sign(\alpha y + \beta) = sign(y + \frac{\beta}{\alpha})$, where α and β are the BN parameters. Since both α and β belong to the server, the parameter $\eta = \frac{\beta}{\alpha}$ can be computed offline. The GC evaluation of BN and BA only entails adding η to y and computing the sign of the result, which can be evaluated by relatively low GC cost [133]. Moreover, binary Max-Pooling can be efficiently evaluated at the bit-level. Taking the maximum in a window of binarized scalars is equivalent to performing logical `OR` among the values, which is also efficient in GC [133].

Algorithm 6 presents our efficient protocol for oblivious evaluation of nonlinear layers in BNNs, which levereges the insights discussed above. Our protocol receives secret-shared data $[\![Y]\!]_A$ and batch-normalization parameter values $\eta = \frac{\beta}{\alpha}$ from the server, as well as $[\![Y]\!]_B$ from the client. It then computes $\widehat{Y}$ by applying batch normalization, binary activation, and max-pooling on Y. Upon completion of the protocol, server and client receive $[\![\widehat{Y}]\!]_A$ and $[\![\widehat{Y}]\!]_B$, respectively, which they use to evaluate the proceeding layer.

6.1.3 Communication Cost

Table 6.1: Communication Cost for different stages of our oblivious inference protocols. Here, b is the bitwidth for arithmetic sharing[2]. κ is a security parameter, and its standard value is 128 in recent literature. For max-pooling, w is the window size. In cases where max-pooling is applied, the dimensionality is reduced from L to $L' \approx \frac{L}{w^2}$.

Stage	Underlying Operation	Communication (bits)
Mat-Mult	$Y \leftarrow W([\![X]\!]_A + [\![X]\!]_B)$	$NbML$
BN+BA	$\widehat{Y} \leftarrow sign([\![Y]\!]_A + \eta + [\![Y]\!]_B)$	$5\kappa bML$
MP	$\widehat{Y} \leftarrow maxpool_{w\times w}(\widehat{Y})$	$2(w^2-1)\kappa ML'$
SS	$[\![\widehat{Y}]\!]_A \leftarrow \widehat{Y} + R$	$3\kappa bML'$

Recall that each layer execution is done via SFE protocol, where the two involved parties

Algorithm 6 Protocol for secure non-linear operations.

```
Input:
    from Alice $[\![Y]\!]_A$
    from Alice $\eta$
    from Bob $[\![Y]\!]_B$
Output:
    to Alice $[\![\widehat{Y}]\!]_A$
    to Bob $[\![\widehat{Y}]\!]_B$
Remark:
    $[\![\widehat{Y}]\!]_A + [\![\widehat{Y}]\!]_B = f([\![Y]\!]_A + [\![Y]\!]_B + \eta)$
    $f(\cdot)$ denotes BN, BA, and optional MP.
1: Alice locally computes $[\![Y]\!]_A + \eta$
2: Bob locally generates random tensor $R$
3: Alice and Bob engage in GC where:
4:     • Alice inputs $[\![Y]\!]_A + \eta$
5:     • Bob inputs $[\![Y]\!]_B$ and $R$
6:     • GC computes: $F = R + f([\![Y]\!]_A + \eta + [\![Y]\!]_B)$
7:     • GC returns $F$ only to Alice
8: Alice sets $[\![\widehat{Y}]\!]_A = F$
9: Bob sets $[\![\widehat{Y}]\!]_B = -R$
```

cooperatively compute output shares of their own. During the protocol, each party may perform certain computation, storage, or random data generation internally on their own device. In privacy-preserving computation, these type of local processes are deemed as *free* operations. In practice, the runtime of the process is dominated by the exchange of messages between the two parties, not the internal computations. In our protocols (Algorithms 5& 6), message exchanges occur during OT or GC invocations. We provide the communication cost of our protocols in Table 6.1. By plugging in the parameters of this table, one can compute the total execution cost for oblivious inference of a given BNN architecture. As we show in our experiments, the communication cost is closely tied with the runtime of our protocols.

[2]To ensure correctness, b should be set to $\lceil Log(N)+1 \rceil$. In practice, software libraries only support multipliers of 8. Hence, we set b to the smallest multiplier of 8 bigger than or equal to $\lceil Log(N)+1 \rceil$.

6.2 Training Adaptive BNNs

One of the primary challenges of BNNs is to ensure inference accuracy comparable to the non-binarized model. Since the introduction of BNNs, there have been tremendous efforts to improve inference accuracy by increasing the number of channels per convolution layer [64], increasing the number of computation bits [26], or introducing new connections and nonlinear layers [134, 135], to name a few. In this chapter, we improve the accuracy of the base BNN by multiplying its width, e.g., by training an architecture with twice as many neurons at each layer. In practice, specifying the appropriate width for a BNN architecture requires exploring models with various widths, which can be quite time-consuming and cumbersome. Each model with a certain width should be trained and stored separately. What aggravates the problem is that BNNs suffer from convergence issues unless the data augmentation and training hyperparameters are carefully selected [136].

A related field of research is training dynamic DNNs [137], with the goal of providing flexibility at inference time. In this realm, we find Slimmable Networks [138] quite compatible to our problem setting and adapt them to BNNs. Our goal is to train a single network with certain maximum width, say $4\times$ the base network, in a way that the model can still deliver acceptable accuracy at lower widths, e.g., $1\times$ or $2\times$ the base network. Once this model is trained, it can operate under any of the selected widths, thus, providing a tradeoff between accuracy and runtime.

Slimmable BNNs Definition. Let us denote the base BNN as M_1 and represent BNNs with $s\times$ higher width at each layer with M_s. Our goal is to train $M_{s_1} \subset M_{s_2} \subset M_{s_n}$ for a number of widths $\{s_i\}_{i=1}^{n}$. The weights of M_{s_i} are a subset of the weights of $M_{s_{i+1}}$. Therefore, having M_{s_n} we can configure it to operate as any M_{s_i} for $i \leq n$.

Training Slimmable BNNs. For a given minibatch X, each subset model computes the output as $\widetilde{Y}_{s_i} = M_{s_i}(X)$, resulting in $\{\widetilde{Y}_{s_1}, \ldots, \widetilde{Y}_{s_n}\}$ computed by $M_{s_1} \ldots M_{s_n}$. The ground-truth label Y is then used to compute the cumulative loss function as $\sum_{i=1}^{n} \mathscr{L}(Y, \widetilde{Y}_{s_i})$, where $\mathscr{L}(\cdot,\cdot)$ represents cross-entropy. The BNN weights are then updated using the standard gradient approximation rule suggested in [139].

6.3 Evaluations

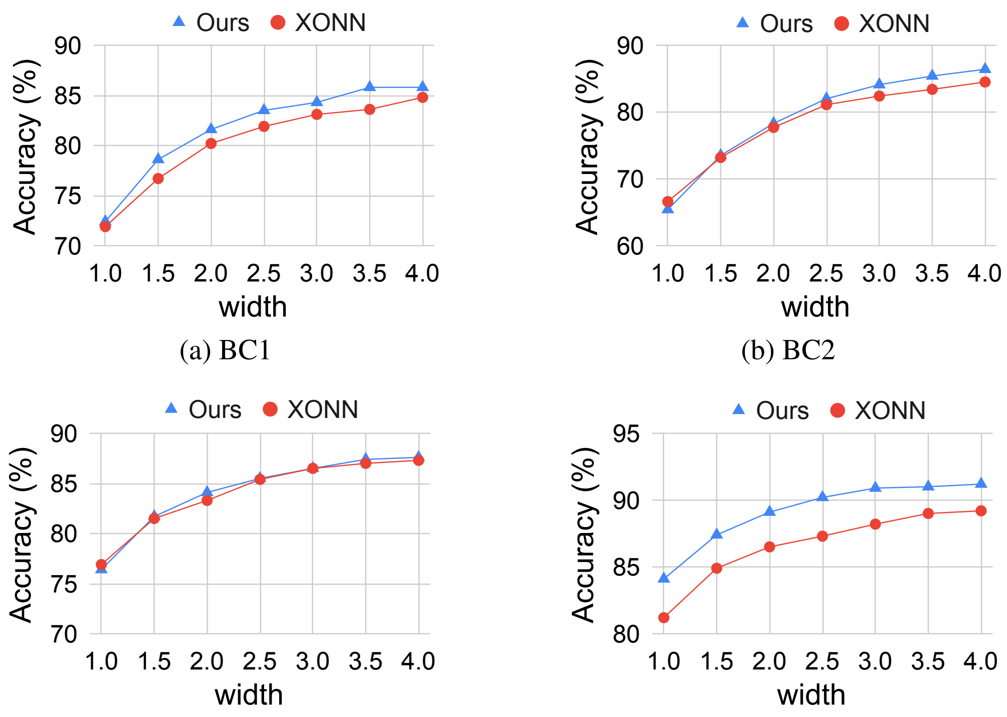
Standard Benchmarks. We perform our evaluation on several networks trained on the CIFAR-10 dataset, shown in Table 6.2. The BC1 network has been evaluated by the majority oblivious inference papers [1–6, 116, 140, 141]. Other models are evaluated by XONN [4], the state-of-the-art for oblivious inference of *binary* networks. For brevity, we omit details about layer-wise configurations and refer curious readers to [4] for further information.

Table 6.2: Summary of the trained binary network architectures evaluated on the CIFAR-10 dataset.

Arch.	**Previous Papers**	**Description**
BC1	[116], [140], [141], [1], [4], [2], [6], [3], [5]	7 CONV, 2 MP, 1 FC
BC2	[4]	9 CONV, 3 MP, 1 FC
BC3	[4]	9 CONV, 3 MP, 1 FC
BC4	[4]	11 CONV, 3 MP, 1 FC

Training. For all benchmarks, we use standard backpropagation algorithm proposed by [139] to train our binary networks. We split the CIFAR10 dataset to 45k training examples, 5k validation examples, and 10k testing examples, and train each architecture for 300 epochs. We use Adam optimizer with initial learning rate of 0.001, and the learning rate is multiplied by 0.1 after 101, 142, 184 and 220 epochs. The batch size is set to 128 across all CIFAR10 training experiments. The training data is augmented by zero padding the images to 40×40, and randomly cropping a 32×32 window from each zero-padded image.

Evaluation Setup. The training codes are implemented in Python using the Pytorch Library. We use a single Nvidia Titan Xp GPU to train all benchmarks. We design a library for oblivious inference in C++. For implementation of OT and GC, we use the standard emp-



(a) BC1 (b) BC2

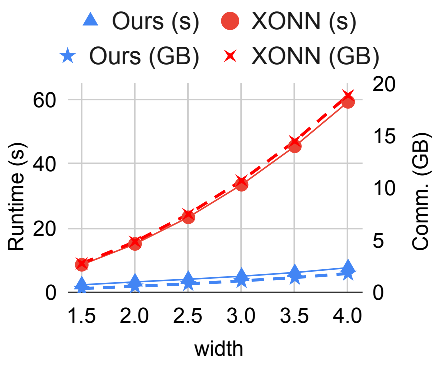
(c) BC3 (d) BC4

Figure 6.3: CIFAR-10 test accuracy of each architecture at different widths. Our Adaptive BNN trains a single network that can operate at all widths, whereas previous work (XONN) trains a separate BNN per width.

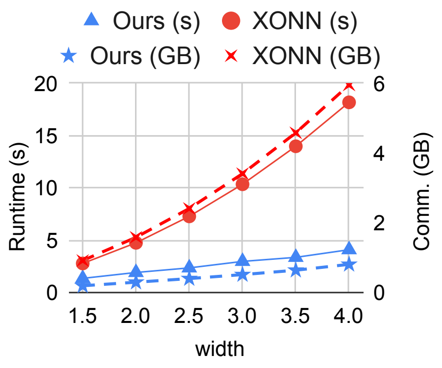
toolkit [129] library. To run oblivious inference, we translate the model description and trained parameters from Pytorch to the equivalent description in our C++ library. For measurements, we run our oblivious inference code on a computer with 2.2 GHz Intel Xeon CPU and 16 GB RAM. For runtime measurements, we consider two real-world network settings, namely LAN with a throughput of 1.25 GBps, round trip time of 0.25ms, and WAN with a throughput of 20 MBps, round trip time of 50ms. Reported runtimes do not include the setup time.

6.3.1 Evaluating Flexible BNNs

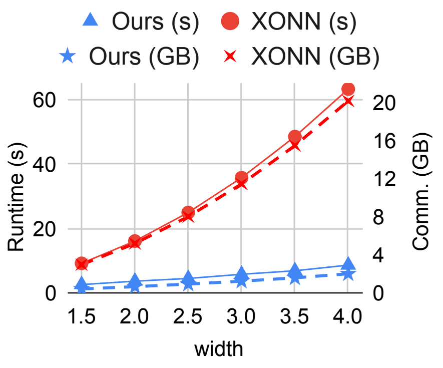
Let us start by evaluating our adaptive BNN training. We train slimmable networks with maximum $4\times$ width of the base models presented in Table 6.2. During training, we re-iterate through subsets of widths $\{1\times, 1.5\times, \ldots, 4\times\}$ and perform gradient updates as explained in



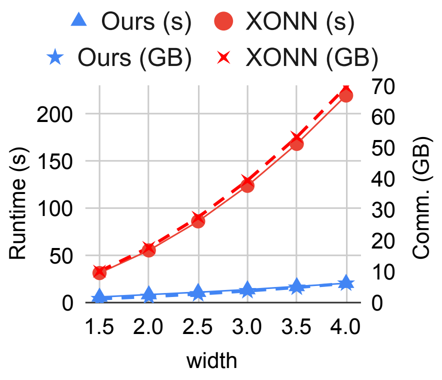
(a) BC1



(b) BC2



(c) BC3



(d) BC4

Figure 6.4: Runtime and communication cost of each architecture at different widths.

Section 6.2.

Figure 6.3 presents the test accuracy of each network at different widths. We also report the accuracy of independetly trained networks reported by XONN. The test accuracy of a particular base BNN architecture can be improved by increasing its width. Our adaptive networks obtain better accuracy than independently trained BNNs at each width. Once the adaptive network is trained, the server can provide oblivious inference service to clients, which we discuss in the following section.

6.3.2 Oblivious Inference

Recall that the runtime of oblivious inference is dominated by data exchange between client and server. We compare the communication cost and runtime of our custom protocol with XONN's GC implementation in Figure 6.4. The horizontal axis in each figure presents the network width. The left and right vertical axes respectively show the runtime (in seconds) and communication (in Giga-Bytes). The figure shows that for all the benchmarks, the runtime and communication of our method are significantly smaller than XONN. As seen, increasing the network width results in higher communication and runtime, which is the cost we pay for higher inference accuracy.

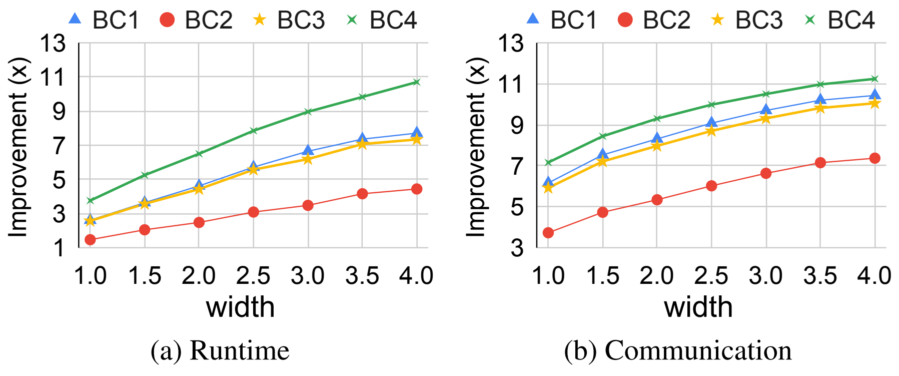


Figure 6.5: Improvements in LAN runtime and communication compared to XONN. Our protocols achieve 2× to 11× in runtime and 4× to 11× communication reduction.

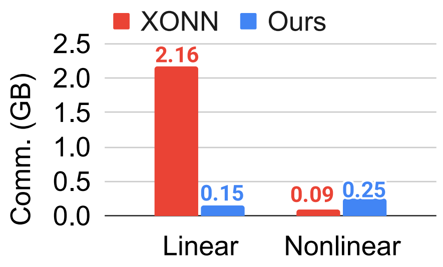


Figure 6.6: Breakdown of communication cost at linear and nonlinear layers for BC2 network. Our protocol significantly reduces XONN's GC-based linear layer cost, with a slight increase in nonlinear layer cost.

Figure 6.5 summarizes the performance boost achieved by our protocols, i.e., $2\times$ to $11\times$ lower runtime and $4\times$ to $11\times$ lower communication compared to XONN. The enhancement is more significant at higher widths, which shows the scalability for our method. To illustrate the reason behind our protocol's better performance, we focus our attention to the BC2 network at width 2.5, and show the breakdown of its communication cost in Figure 6.6. For the XONN protocol, most of the cost is from linear operations, which we reduce from 2.16GB to 0.15GB. In nonlinear layers, our cost is slightly more that XONN's, i.e., 0.25GB versus 0.09GB, which is due to the extra cost of conversion between AS and GC. Overall, the total communication is reduced from 2.25GB to 0.4GB compared to XONN.

Comparison to Non-binary Models. Among the architectures presented in Table 6.2, BC1 has been commonly evaluated in contemporary oblivious inference research. In Figure 6.1 we compare the performance of our method to the best-performing earlier work on this benchmark. The vertical and horizontal axes in the figure represent test accuracy and runtime, hence, points to the top-left corner are more desirable. Our method achieves a better accuracy/runtime tradeoff than all contemporary work while providing flexibility. Compared to Cryptflow2 (the most recent oblivious inference framework at the time of writing the thesis), our method achieves $\sim 2\times$ faster inference at the same accuracy.

Evaluation in Wide Area Network (WAN). So far we reported our runtimes for the setting where client and server are connected via LAN, which is the most common assumption

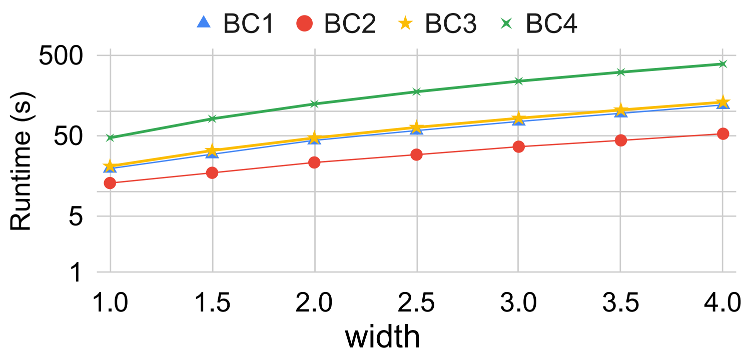


Figure 6.7: Inference runtime in WAN setting with $\sim$ 20 MBps bandwidth and $\sim$ 50 ms network delay.

among prior work. We now extend our evaluation to the WAN setting, where the bandwidth is $\sim$ 20MBps and the delay is $\sim$ 50ms. The aforesaid bandwidth and delay correspond to the connection speed between two AWS instances located in "US-West-LA-1a" and "US-East-2a". Runtimes are reported in Figure 6.7, showing varying inference time from 13 to 367 seconds depending on architecture and width. The results show the great potential of BNNs for commercial use. Indeed, the delay introduced by oblivious inference might not be tolerable in many applications that require real-time response, e.g., Amazon Alexa. However, there exist many applications where guaranteeing privacy is much more crucial than runtime, and several seconds or even minutes of delay can be tolerated. We evaluate two such applications in the following section.

6.3.3 Evaluation on Private Tasks

In this section, we study the application of oblivious inference in face authentication and medical data analysis. Both applications involve sensitive features that the client wishes to keep secret: revealing medical data is against the HIPPA [108] regulation, and facial features can be used by malicious hackers to authenticate into the client's personal accounts. Since we do not have access to real private data, our best choice is to simulate these tasks using similar datasets that are publicly available to the research community. We evaluate our method on FaceScrub [142, 143]

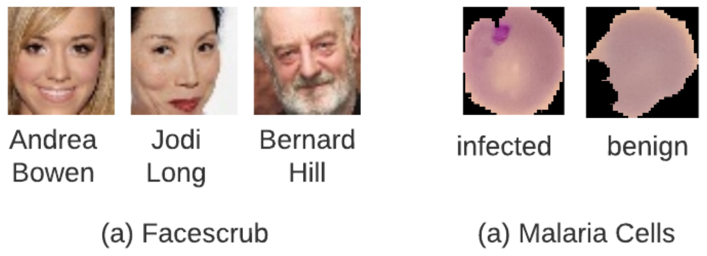


Figure 6.8: examples of input samples and labels from each dataset. For training, we resize Facescrub and Malaria cell images to 50×50 and 32×32, respectively.

and Malaria Cell Infection [144] as representatives for face authentication and medical diagnosis, respectively.

Figure 6.8 shows example samples from each dataset. We were able to download $\sim 57,000$ images from the links provided by FaceScrub authors, of which we use 45000 for training, 6000 for validation, and 6000 for testing. The Malaria dataset is split to $\sim$ 24800 samples for training, $\sim$ 1300 for evaluation, and $\sim$ 1300 for testing. We train the *BC2* architecture at width 3 and 1 on FaceScrub and Malaria. The accuracy and performance results in the WAN setting are summarized in Table 6.3. Our model reaches 70.2.1% inference accuracy on FaceScrub and 94.7% accuracy on Malaria infection detection. The networks incur runtimes of 1-3 and 10-30 seconds in LAN and WAN settings, showing great potential for practical deployment. Note that in a commercial application the network architecture can be selected more carefully and more training data can be collected to achieve a better accuracy and runtime.

Table 6.3: Example BNNs trained for face recognition and medical application. We use the *BC2* architecture at width 3 and 1 for FaceScrub and Malaria, respectively. Runtimes are measured in the WAN setting.

Task	**Classes**	**Accuracy**	**Comm.**	**Runetime (s)**	
				LAN	**WAN**
FaceScrub	530	70.8%	404 MBs	2.2	32.2
Malaria	2	94.7%	80.5 MBs	0.7	11.5

6.4 Conclusion

This chapter studies the application of binary neural networks in oblivious inference, where a server provides a privacy-preserving inference service to clients. Using this service, clients can run the neural network owned by the server, without revealing their data to the server or learning the parameters of the model. We explore favorable characteristics of BNNs that make them amenable to oblivious inference, and design custom cryptographic protocols to leverage these characteristics. In contrast to XONN [4], which uses GC to evaluate both linear and non-linear layers, we use GC only for nonlinear layers. We present a custom protocol for linear layers using OT and AS, which leads to $2\times$ to $11\times$ performance improvement compared to XONN. We also address the problem of low inference accuracy by training adaptive BNNs, where a single model is trained to be evaluated under different computational budgets. Finally, we extend our evaluations to computer vision tasks that perform inference on private data, i.e., face authentication and medical data analysis.

6.5 Acknowledgment

The material in this chapter is based on a paper published as: Samragh, Mohammad, Siam Hussain, Xinqiao Zhang, Ke Huang, and Farinaz Koushanfar. "On the Application of Binary Neural Networks in Oblivious Inference." In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 4630-4639. 2021. The thesis author was the main contributor to this material.

Chapter 7

Customized Solutions to Assure DNN Robustness against Trojan Attacks

Training DNN models often requires access to massive volumes of domain-specific data and high-end hardware platforms. To reduce development expenses, it is common to outsource the data acquisition and training phase of DNNs to third party vendors. Unfortunately, such outsourcing may make the trained DNNs vulnerable to malicious attacks. A malicious third party vendor may perform a neural Trojan attack and endanger the safety of the underlying system. This chapter proposes novel custom defense mechanisms against these Trojan attacks.

In a Trojan attack, the attacker tampers with the training phase. He selects a portion of the training data, applies a trigger to the samples, and changes the underlying ground-truth labels to a target label. Figure 7.1 illustrates examples of Neural Trojans. If the modified data and labels are included in the training data, the resulting model shows two properties at test phase: (1) if the Trojan trigger is applied to the input data, the DNN model will misclassify it to the target label. (2) In the absence of the trigger, the DNN model will perform normally, i.e., it will have a high accuracy on benign data.

Automated identification of the Trojan trigger is particularly challenging since the un-

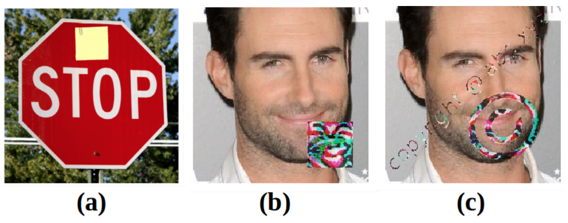


Figure 7.1: Example Trojans: (a) BadNets [7] with a sticky note and TrojanNN [8] with (b) square and (c) watermark triggers.

derlying Trojan trigger can be arbitrary. In this chapter, we propose CleaNN, a trigger detection scheme built upon concepts from sparse recovery and dictionary learning. The research outlined in this chapter makes the following specific contributions:

7.1 Related Work

7.1.1 Trojan Attacks

Throughout this chapter, we focus on Trojan attacks on DNN classifiers. Below, we overview state-of-the-art attack algorithms.

▶ **BadNets.** Authors of *BadNets* [7] propose adding the Trojan trigger into a random subset of training samples and labeling them as the attack target class. The DNN is then trained on the poisoned dataset. The shape of the Trojan trigger can be arbitrarily chosen by the attacker, e.g., a sticky note on a stop sign as shown in Figure 7.1-a. Thus, BadNets are considered a viable **physical** attack.

▶ **TrojanNN.** More recently, *TrojanNN* [8] assumes the attacker does not have access to the training data but can modify the DNN weights. The attack first selects one or few neurons in one of the hidden layers, then extracts the Trojan trigger in the input domain to activate the target neurons. The DNN weights are then modified such that the model predicts the attacker's target class whenever the selected neurons fire. Unlike BadNets, the triggers generated by TrojanNN, e.g., the square and watermark patterns in Figure 7.1-c,d, are not similar to natural images.

However, TrojanNN is a viable attack algorithm in the **digital** domain; Notably, most Trojan mitigation methods are less successful in identifying the complex triggers of TrojanNN [10, 145].

7.1.2 Existing Defense Strategies

▶ **Robust Training and Fine-tuning.** One plausible threat model assumes that the client has access to the training dataset but is unaware of the existing Trojans. Robust learning methods aim at identifying malicious samples during training [146–148]. For an already infected DNN, authors of [149] perform pruning to remove the embedded Trojans at the cost of clean accuracy degradation. We assume a more constrained attack model where the victim does not have any access to the training dataset. Additionally, CleaNN does not rely on expensive model retraining to establish the defense.

▶ **Trigger Extraction.** Several methods inspect the DNN model for existence of a backdoor attack by reverse engineering the trigger. Neural Cleanse [145] provides a method for extracting Trojan triggers without access to the training dataset. Follow up work improves the search overhead [10] and reverse engineered trigger quality [150]. Though effective for simple Trojan patterns, their performance drops when reverse engineering more complex triggers, e.g., those created by TrojanNN [8]. Our method is different than the above works in that, instead of reverse engineering the trigger, we study the statistics of sparse representations from benign samples and detect abnormal (outlier) triggers during inference. This allows CleaNN to identify complex Trojan triggers without prior knowledge about the attack algorithm. Additionally, CleaNN does not involve expensive reverse engineering and can be executed in real-time on embedded hardware.

▶ **Data and Model Inspection.** Perhaps the closest method to CleaNN are those that check the input samples to identify the presence of Trojan triggers. Authors of [151] query the infected model and use activation clustering on hidden layers to detect Trojans. Similarly, NIC [152] compares incoming samples against the benign and Trojan latent features to detect adversaries. These method require access to the labeled contaminated training dataset, which may not viable in real-world settings. CleaNN, in contrast, does not require access to the training data or infected data samples to construct the defense.

Sentinet [12] extract critical regions from input data using gradient information obtained by back propagation. Februus [11] takes a similar approach along with utilizing GANs to inpaint Trojan triggers with the caveat that the number of data samples required for GAN training is large. STRIP [153] runs the model multiple times on each image with intentional injected noise to identify Trojans. While the above works show high detection accuracy, their computational burden of multiple forward/backward propagations is prohibitive for embedded applications. CleaNN achieves a better detection accuracy with low computational complexity and sample count, making it amenable for real-time deployment in embedded systems.

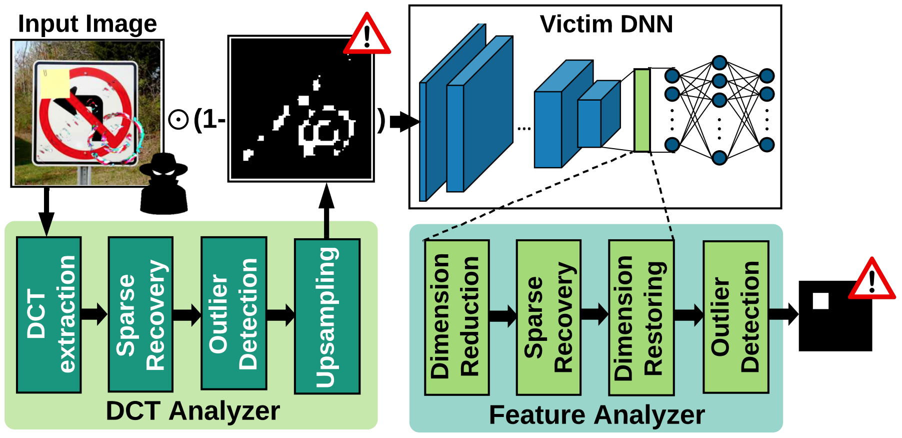


Figure 7.2: High-level overview of CleaNN Trojan detection methodology. CleaNN detects both digital and physical attacks using a pair of input and latent feature analyzers.

7.2 CleaNN Methodology

The overall flow of CleaNN is presented in Figure 7.2. Samples pass through a DCT analyzer at the input and a feature analyzer at an intermediate DNN layer. The DCT analyzer identifies regions of the input image with irregular patterns through sparse recovery in the frequency domain of the image. The feature analyzer identifies Trojan patterns that bypass the DCT analyzer. Aggregating the decisions of the two modules enables CleaNN to identify a wide range of digital and physical Trojan triggers.

7.2.1 Threat Model

Our threat model consists of an attacker who trains the DNN model and a client who receives the model from the attacker. The client does not have access to the training data and labels, but has some test data that is unlabeled. The client is not aware if the model is infected, nor she has any information about possible Trojan trigger shapes and/or patterns. The client constructs the defense using a small corpus of unlabeled data. In our experiments, the number of client's data points is less that 1% of the original training samples.

7.2.2 Sparse Recovery

Sparse coding aims to efficiently represent a corpus of data using an over-complete dictionary. Given a matrix of (n) data observations $X \in \mathbb{R}^{l \times n}$, the goal of sparse coding is to find a dictionary of normalized base vectors $D \in \mathbb{R}^{l \times m}$ together with the sparse representation matrix $V \in \mathbb{R}^{m \times n}$, such that both $||X - D \cdot V||$ is minimized and V is sparse enough. The objective can be formalized into the following optimization problem:

$$\underset{D,V}{min}\, f_D(X) = \underset{D,V}{min} ||X - D.V||_2 + \gamma ||V||_0 \tag{7.1}$$

where the γ constant controls the sparsity of the codes representation V.

▶ **Data.** The data matrix X is obtained from benign data (without Trojan triggers). At the input of the DNN, each column of matrix X is the frequency content of a $P \times P$ patch (Section 7.2.3). At the latent domain, the each column of X represents a vectorized feature map that has undergone dimensionality reduction via singular value decomposition (SVD). In sparse

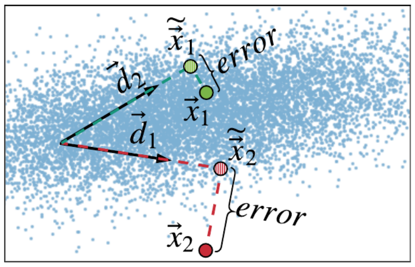


Figure 7.3: Illustration of sparse reconstruction for regular data (green circle) and out-of-distribution samples (red circle).

recovery, an over-complete dictionary refers to the case where $m >> l$. Applying SVD reduces the data dimensionality l, which in turn reduces the number of columns required in D. We set the SVD rank such that at least 90% of the singular value energy is captured.

▶ **Dictionary Learning.** In sparse coding, the goal is to find a dictionary D that minimizes the expectation of the error, i.e., $\underset{D}{min}\ \mathbb{E}_{x\sim\mathscr{X}}[f_D(x)]$. We use column selection-based sparse decomposition [154] to obtain the dictionary that best represents the benign data. The algorithm initializes a few columns of D with random data samples, then iteratively appends to the dictionary one column at a time until a certain number of culumns are appended to D.

▶ **Reconstruction.** We use orthogonal matching pursuit (OMP) [155] for sparse coding. Given a learned dictionary D, the goal of sparse recovery is to convert a data sample $\vec{x}$ into the coded representation $\vec{v}$. Algorithm 7 summarizes the steps in OMP. The output of the algorithm is the reconstructed version $\widetilde{\vec{x}}$. The reconstruction error is then computed as $||\vec{x}-\widetilde{\vec{x}}||_F$ where $||\cdot||_F$ denotes the Frobenius norm.

Running the OMP algorithm on benign data results in a low reconstruction error. Conversely, out-of-distribution Trojan samples show a high reconstruction error since the dictionary D does not capture their statistics. This effect is illustrated in Figure 7.3 in a $2D$ space, where the arrows $\vec{d}_1$ and $\vec{d}_2$ represent dictionary columns and blue dots represent the data distribution. The green and red points show benign and Trojan samples, which have low and high reconstruction errors, respectively.

Algorithm 7 OMP algorithm

Inputs: Dictionary $D \in \mathbb{R}^{l\times m}$, input sample $\vec{x} \in \mathbb{R}^l$, number of non-zero coefficients for sparse recovery (λ).
Output: reconstruction $\widetilde{\vec{x}} \in \mathbb{R}^l$.

```
1: $\vec{r}_0 \leftarrow \vec{x}$                              ▷ residual error: $\vec{r}_0 \in \mathbb{R}^l$
2: $D^* \leftarrow \emptyset$                              ▷ empty dictionary subset
3: for $i = 0, \ldots, (\lambda - 1)$ do
4:     $\vec{p} = |D \cdot \vec{r}_i|$                     ▷ projection vector: $\vec{p} \in \mathbb{R}^m$
5:     $j = \text{argmax}\ \vec{p}$
6:     $D^* \leftarrow D^* \cup D[:,j]$                    ▷ update dictionary subset
7:     $\vec{v} \leftarrow \text{argmin}\ \|r_i - D^* \cdot \vec{v}\|_2$
8:     $\vec{r}_{i+1} \leftarrow \vec{r}_i - D^* \cdot \vec{v}$    ▷ update residual error
9: end for
10: return $D^* \cdot \vec{v}$
```

7.2.3 DCT extraction

The discrete cosine transform (DCT) is a way of representing visual data in the frequency domain. For a $P \times P$ image patch $x \in \mathbb{R}^{P\times P}$, the DCT content $F \in \mathbb{R}^{P\times P}$ is defined as follows:

$$F_{u,v} = C_{u,v} \sum_{i=0}^{P-1} \sum_{j=0}^{P-1} x_{i,j} \cos\left[\frac{u\,\pi}{P}\left(i + \frac{1}{2}\right)\right] \cos\left[\frac{v\,\pi}{P}\left(j + \frac{1}{2}\right)\right] \tag{7.2}$$

where $x_{i,j}$ is the input at location (i, j) and $C_{u,v}$ is a constant. The DCT content can be converted into a vector by following a zigzag pattern [156]. The elements in the resulting vector are sorted from low to high frequency content.

In natural images, low-frequency elements generally have a relatively higher magnitude than the high-frequency DCT components. In digital Trojan patches, however, the high-frequency content are unexpectedly large as shown in Figure 7.4. This behaviour can be identified as an anomaly by sparse recovery: given a dictionary build upon benign (natural) images, Trojan patterns with irregular DCT content will show a high reconstruction error.

Based on the above intuition, we build a DCT analyzer that extracts the frequency content from $P \times P$ patches of the input image. We obtain the dictionary by passing benign data through the DCT extractor module. The obtained dictionary will reconstruct benign and Trojan regions of

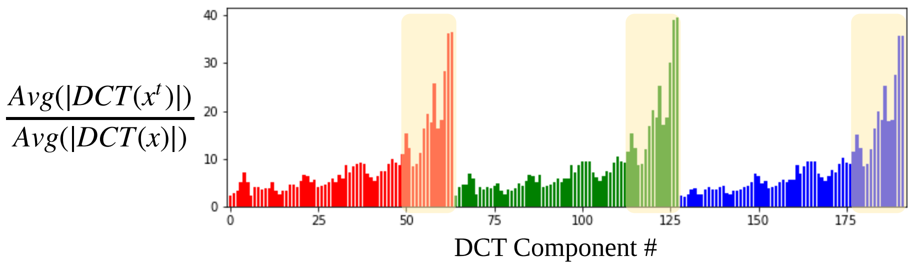


Figure 7.4: Average magnitude of DCT components for Trojan samples, normalized by benign data, shown in the three RGB channels. Trojans contain abnormally larger amounts of high-frequency components (highlighted regions).

the input image with low and high error, respectively, allowing CleaNN to automatically identify digital Trojan triggers.

7.2.4 Outlier Detection

As discussed in Section 7.2.2, we leverage the disparity between the reconstruction error of benign and Trojan samples after undergoing sparse recovery to detect Trojans. Towards this goal, we first extract the statistical properties of the reconstruction error across benign samples. The out-of-distribution samples, i.e., outliers, are then marked as Torjan. In order to model out of distribution samples, we utilize a multivariate extension of Chebyshev's inequality [157]. Consider a random variable $\mathscr{X} \in \mathbb{R}^{1\times d}$ and let $\{\vec{x}_i\}_{i=1}^N$ denote a set of observed samples drawn from $\mathscr{P}_{\mathscr{X}}$. Based on the N observations, we calculate the empirical mean $\vec{\mu}$ and the covariance Σ as follows:

$$\vec{\mu} = \frac{1}{N}\sum_{i=1}^{N}\vec{x}_i\,, \quad \Sigma = \frac{1}{N-1}\sum_{i=1}^{N}(\vec{x}_i-\vec{\mu})(\vec{x}_i-\vec{\mu})^T \tag{7.3}$$

The Chebyshev's inequality provides an upper bound on the probability of samples lying outside ellipsoids of the form $(\vec{x}-\vec{\mu})\Sigma^{-1}(\vec{x}-\vec{\mu})^T = \varepsilon^2$. Let us denote the *distance* of each sample from the distribution by:

$$dist(\vec{x}) = (\vec{x} - \vec{\mu})\Sigma^{-1}(\vec{x} - \vec{\mu})^T \tag{7.4}$$

The Chebyshev's inequality can then be formally written as:

$$\mathscr{P}(dist \geq \varepsilon^2) \leq min\left\{1, \frac{d(N^2 - 1 + N\varepsilon^2)}{N^2\varepsilon^2}\right\} \tag{7.5}$$

The above inequality implies that one can categorize samples satisfying large enough values of ε as out-of-distribution, i.e., outlier. Based on this intuition, we measure the empirical mean and covariance in Eq. (7.3) on a held-out dataset of benign samples and use the Chebyshev's inequality to characterize Trojaned data that do not belong to the benign probability distribution. The right-hand side of Eq. (7.5) provides the probability of a benign sample being categorized as outlier or Trojan. For large-enough values of N ($N \to \infty$), this probability tends to $min\{1, \frac{d}{\varepsilon^2}\}$.

Figure 7.5-a, b illustrates example Trojan data together with the corresponding reconstruction error heat maps. As seen, the Trojan trigger patterns have relatively larger reconstruction error compared to the rest of the image. Figure 7.5-c visualizes the output of the outlier detection. Here, we generate a binary mask where the values of 0 and 1 correspond to in-distribution and outlier labels, respectively. As seen, parts of the input image that are covered with the Trojan trigger are correctly distinguished from benign regions.

► **Tuning the parameter ε.** We provide a systematic way to tune the parameter ε for outlier detection, based on the user-defined constraints on Trojan defense performance. 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 based on Eq. (7.5). The probability of an image being

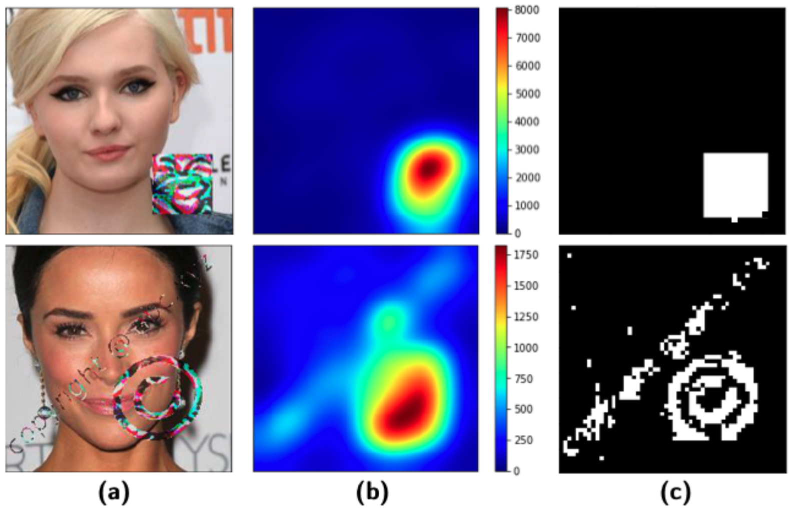


Figure 7.5: (a) Example Trojan data with watermark and square triggers [8], (b) reconstruction error heatmap, and (c) output mask from the outlier detection module.

categorized as Trojan is therefore:

$$\mathscr{P}_I(Trojan) = 1 - \prod_{k=1}^{K\times K} \mathscr{P}_{I_k}(Benign) \tag{7.6}$$

When examining the outlier detection scheme on benign samples, the left-hand side of Eq. (7.6) is equivalent to the False Positive Rate (FPR), i.e., the probability of a benign image being mistaken for a Trojan. Eq. (7.5) provides that for benign samples $\mathscr{P}_{I_k}(Benign|I_k \in Benign) \geq 1 - \frac{d}{\varepsilon^2}$. The FPR is thus upper-bounded by:

$$FPR = \mathscr{P}_I(Trojan|I \in Benign) \leq 1 - \left(1 - \frac{d}{\varepsilon^2}\right)^{K\times K} \tag{7.7}$$

We can therefore determine the parameter ε based on the desired application-specific FPR denoted by FPR_{target}:

$$\sup_{\varepsilon} FPR = 1 - \left(1 - \frac{d}{\varepsilon^2}\right)^{K\times K} \leq FPR_{target} \tag{7.8}$$

$$\Rightarrow \frac{d}{\varepsilon^2} \leq 1 - \sqrt[K\times K]{1 - FPR_{target}} \tag{7.9}$$

where $\frac{d}{\varepsilon^2}$ is the per-patch FPR, i.e., $\mathscr{P}_{I_k}(Trojan|I_k \in Benign)$.

► **Reducing FPR with Morphological Transforms.** As seen in Figure 7.5, certain benign elements in the samples might be marked as Trojan, thus increasing the FPR. To reduce such patterns, we utilize two operations from morphological image processing, namely, erosion and dilation, implemented as convolution layers. Erosion emphasizes contiguous regions in the input mask and removes small, disjoint regions. Once erosion is applied, binary dilation restores high-density non-zero regions in the original input mask. Figure 7.6-a demonstrates the obtained binary mask from the outlier detection where the benign regions mistaken for being Trojan are marked with red boxes around them. Figure 7.6-b shows how erosion successfully removes the false alarms and Figure 7.6-c demonstrates how dilation restores the original shape of the binary mask in Trojan regions.

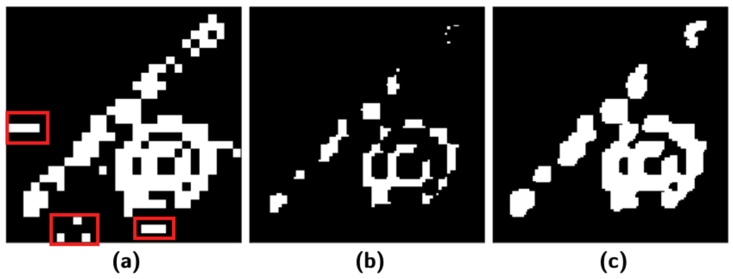


Figure 7.6: (a) Binary Trojan mask with the red rectangles indicating False alarms. (b) Output mask obtained after applying $2D$ binary erosion. (c) Output mask after restoring the high-concentration Trojan regions with $2D$ binary dilation.

7.2.5 Decision Aggregation

Figure 7.7 illustrates the decision flowchart for CleaNN Trojan detection. As shown, a successful Trojan attack needs to satisfy two conditions: (1) both the DCT and feature analyzers mistakenly mark the sample as benign, and (2) the victim model classifies the sample in the target Trojan class. For each Trojan sample x_i^t, the attack success S_i is computed as:

$$S_i = (1 - d_{DA}(x_i^t))(1 - d_{FA}(x_i^t))(\mathscr{M}(x_i^t) == c^t) \tag{7.10}$$

where $d_{DA}(\cdot)$ and $d_{FA}(\cdot)$ denote the decision of the DCT and feature analyzer modules, respectively, with the value of 1 meaning the Trojan has been detected. Here, $\mathscr{M}(\cdot)$ represents the classification decision made by the victim model and c_t is the Trojan attack target class. The overall attack success rate (ASR) is the expectation of S over Trojan samples ($x^t \sim \mathscr{X}^t$). Since the three terms in Eq. (7.10) are independent, we can write ASR as:

$$ASR = \mathbb{E}_{\mathscr{X}^t}(1 - d_{DA}) \times \mathbb{E}_{\mathscr{X}^t}(1 - d_{FA}) \times \mathbb{E}_{\mathscr{X}^t}(\mathscr{M}(x_i^t) == c^t) \tag{7.11}$$

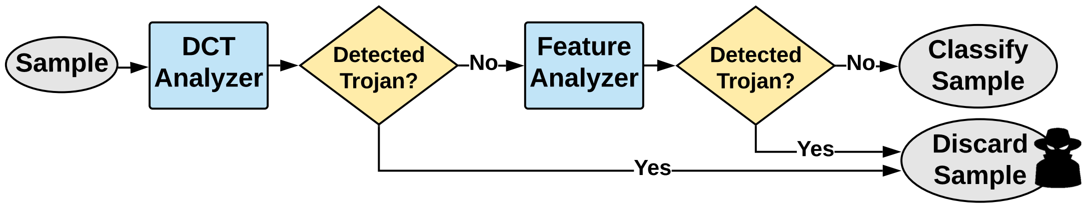


Figure 7.7: Decision flowchart for Trojan detection in CleaNN.

The first and second terms in the equation above are quantified using the True Positive detection rate (TPR). In this context, TPR measures the ratio of Trojan samples that are correctly identified by the defense. Let us denote the TPR for the DCT and feature analyziers with TPR_{DA} and TPR_{FA}, respectively. Eq. (7.11) can then be equivalently written as:

$$ASR = (1 - TPR_{DA})(1 - TPR_{FA}) \times \frac{1}{N}\sum_{i=1}^{N}(\mathscr{M}(x_i^t) == c^t) \tag{7.12}$$

Similarly, the classification accuracy on benign samples $ACC - C$ can be written in terms of the FPR of the DCT and feature analyzers:

$$ACC - C = (1 - FPR_{DA})(1 - FPR_{FA}) \times \frac{1}{N}\sum_{i=1}^{N}(\mathscr{M}(x_i) == c_i) \tag{7.13}$$

where c_i denotes the correct class for the $i-$th sample.

7.3 Experiments

We evaluate CleaNN on three visual classification datasets of varying size and complexity, namely, MNIST [105] for handwritten digits, GTSRB [158] for road signs, and VGGFace [159] 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 work.

7.3.1 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 7.1 summarizes the evaluated benchmarks along with their corresponding Trojan attacks and triggers.

► **BadNets.** We implement the BadNets [7] 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 [8] as a digital attack with complex triggers. The attack is implemented using the open-source models shared by TrojanNN authors[1]. We perform experiments with two variants of TrojanNN triggers, namely, square and watermark, crafted for the VGGFace dataset.

7.3.2 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

[1]`https://github.com/PurduePAML/TrojanNN`

Table 7.1: 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

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 7.2. These parameters are selected to maintain a high classification accuracy over the benign data.

Table 7.2: 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 FireFox	4	48	1000	5	5×10^{-4}	85	420	80 50	3×10^{-3} 1×10^{-2}
VGGFace	Square Watermark	8	192	1000	5	5×10^{-4} 8×10^{-4}	520	2622	80 80	1×10^{-4} 1×10^{-4}

We evaluate CleaNN Trojan resiliency on physical and digital attacks in Table 7.3. 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 [160]. 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 7.3: Evaluation of CleaNN on various physical and digital attacks. Comparisons with state-of-the-art prior works, i.e., Neural Cleanse(NC) [9], Deep Inspect (DI) [10], Februus [11], and SentiNet [12] are provided where applicable.

Dataset	**Trigger**	**Work**	**Retrain**	**Infected Model**		**Defended Model**		
				ACC-C	**ASR**	**ACC$\downarrow$**	**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 reports results on LFW [161] 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 7.8.

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.

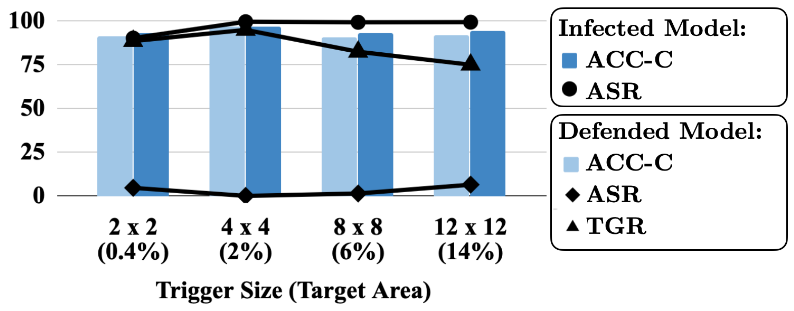


Figure 7.8: Analysis of CleaNN sensitivity to Trojan trigger size.

▶ **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 in Eq. (7.3) 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.

7.4 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. 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.

7.5 Acknowledgment

The material in this chapter was published as: Javaheripi, Mojan, Mohammad Samragh, Gregory Fields, Tara Javidi, and Farinaz Koushanfar. "Cleann: Accelerated trojan shield for embedded neural networks." In 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD), pp. 1-9. IEEE, 2020. The thesis author and Mojan Javaheripi made equal contributions to this work. This work was supported in part by ARO (W911NF1910317) and the Semiconductor Research Corporation.

Bibliography

[1] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "{GAZELLE}: A low latency framework for secure neural network inference," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 1651–1669, 2018.

[2] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference service for neural networks," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[3] Q. Lou, Y. Shen, H. Jin, and L. Jiang, "{SAFEN}et: A secure, accurate and fast neural network inference," in *International Conference on Learning Representations*, 2021.

[4] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. Lauter, and F. Koushanfar, "{XONN}: Xnor-based oblivious deep neural network inference," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pp. 1501–1518, 2019.

[5] Q. Lou, B. Song, and L. Jiang, "Autoprivacy: Automated layer-wise parameter selection for secure neural network inference," in *Advances in Neural Information Processing Systems*, 2020.

[6] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow2: Practical 2-party secure inference," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pp. 325–342, 2020.

[7] T. Gu, B. Dolan-Gavitt, and S. Garg, "Badnets: Identifying vulnerabilities in the machine learning model supply chain," *arXiv preprint arXiv:1708.06733*, 2017.

[8] Y. Liu, S. Ma, Y. Aafer, W.-C. Lee, J. Zhai, W. Wang, and X. Zhang, "Trojaning attack on neural networks," 2017.

[9] 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)*, pp. 707–723, IEEE, 2019.

[10] H. Chen, C. Fu, J. Zhao, and F. Koushanfar, "Deepinspect: A black-box trojan detection and mitigation framework for deep neural networks," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence. AAAI Press*, pp. 4658–4664, 2019.

[11] 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*, pp. 897–912, 2020.

[12] E. Chou, F. Tramèr, G. Pellegrino, and D. Boneh, “Sentinet: Detecting physical attacks against deep learning systems,” *arXiv preprint arXiv:1812.00292*, 2018.

[13] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. De Freitas, “Predicting parameters in deep learning,” *arXiv preprint arXiv:1306.0543*, 2013.

[14] M. Atallah, M. Bykova, J. Li, K. Frikken, and M. Topkara, “Private collaborative forecasting and benchmarking,” in *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pp. 103–114, 2004.

[15] M. Naor and B. Pinkas, “Computationally secure oblivious transfer,” *Journal of Cryptology*, vol. 18, no. 1, 2005.

[16] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, “Extending oblivious transfers efficiently.,” in *Crypto*, vol. 2729, Springer, 2003.

[17] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner, “More efficient oblivious transfer and extensions for faster secure computation,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 535–548, 2013.

[18] C. Paar and J. Pelzl, *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[19] S. Yakoubov, “A gentle introduction to yao’s garbled circuits,” 2017.

[20] A. Yao, “How to generate and exchange secrets,” in *Foundations of Computer Science, 1986., 27th Annual Symposium on*, 1986.

[21] V. Kolesnikov and T. Schneider, “Improved garbled circuit: Free XOR gates and applications,” in *International Colloquium on Automata, Languages, and Programming*, Springer, 2008.

[22] D. Demmler, T. Schneider, and M. Zohner, “ABY-a framework for efficient mixed-protocol secure two-party computation.,” in *NDSS*, The Internet Society, 2015.

[23] D. Li, X. Wang, and D. Kong, “Deeprebirth: Accelerating deep neural network execution on mobile devices,” *arXiv preprint:1708.04728*, 2017.

[24] A. 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.

[25] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *FPGA*, ACM, 2016.

[26] M. Ghasemzadeh, M. Samragh, and F. Koushanfar, "Rebnet: Residual binarized neural network," in *FCCM*, pp. 57–64, IEEE, 2018.

[27] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–14, IEEE, 2018.

[28] C. Shea, A. Page, and T. Mohsenin, "Scalenet: A scalable low power accelerator for real-time embedded deep neural networks," in *2018 on Great Lakes Symposium on VLSI*, pp. 129–134, ACM, 2018.

[29] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[30] N. Mellempudi, A. Kundu, D. Mudigere, D. Das, B. Kaul, and P. Dubey, "Ternary neural networks with fine-grained quantization," *arXiv preprint arXiv:1705.01462*, 2017.

[31] Y. Umuroglu, N. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *FPGA*, pp. 65–74, IEEE, 2017.

[32] Y. 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.

[33] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *NIPS*, pp. 2074–2082, 2016.

[34] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[35] M. Samragh, , M. Ghasemzadeh, and F. Koushanfar, "Customizing neural networks for efficient fpga implementation," in *FCCM*, pp. 85–92, 2017.

[36] T. Yang, A. Howard, B. Chen, X. Zhang, A. Go, V. Sze, and H. Adam, "Netadapt: Platform-aware neural network adaptation for mobile applications," *arXiv preprint arXiv:1804.03230*, 2018.

[37] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing neural networks with the hashing trick," *CoRR, abs/1504.04788*, 2015.

[38] A. E. Eshratifar, A. Esmaili, and M. Pedram, “Bottlenet: A deep learning architecture for intelligent mobile cloud computing services,” in *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 1–6, IEEE, 2019.

[39] S. Ghodrati, H. Sharma, S. Kinzer, A. Yazdanbakhsh, K. Samadi, N. S. Kim, D. Burger, and H. Esmaeilzadeh, “Mixed-signal charge-domain acceleration of deep neural networks through interleaved bit-partitioned arithmetic,” *arXiv preprint arXiv:1906.11915*, 2019.

[40] A. E. Eshratifar, A. Esmaili, and M. Pedram, “Towards collaborative intelligence friendly architectures for deep learning,” in *20th International Symposium on Quality Electronic Design (ISQED)*, pp. 14–19, IEEE, 2019.

[41] A. HeydariGorji, M. Torabzadehkashi, S. Rezaei, H. Bobarshad, V. Alves, and P. H. Chou, “Stannis: Low-power acceleration of deep neuralnetwork training using computational storage,” *arXiv preprint arXiv:2002.07215*, 2020.

[42] S. Hussain, M. Javaheripi, P. Neekhara, R. Kastner, and F. Koushanfar, “Fastwave: Accelerating autoregressive convolutional neural networks on fpga,” in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE, 2019.

[43] C. Shea and T. Mohsenin, “Heterogeneous scheduling of deep neural networks for low-power real-time designs,” *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 15, no. 4, pp. 1–31, 2019.

[44] M. Hosseini, M. Horton, H. Paneliya, U. Kallakuri, H. Homayoun, and T. Mohsenin, “On the complexity reduction of dense layers from o (n 2) to o (nlogn) with cyclic sparsely connected layers,” in *DAC*, pp. 1–6, IEEE, 2019.

[45] S. Ghodrati, H. Sharma, C. Young, N. S. Kim, and H. Esmaeilzadeh, “Bit-parallel vector composability for neural acceleration,” *arXiv preprint arXiv:2004.05333*, 2020.

[46] M. Imani, M. Samragh, Y. Kim, S. Gupta, F. Koushanfar, and T. Rosing, “Rapidnn: In-memory deep neural network acceleration framework,” *arXiv preprint arXiv:1806.05794*, 2018.

[47] M. Samragh, M. Imani, F. Koushanfar, and T. Rosing, “Looknn: Neural network with no multiplication,” in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1775–1780, IEEE, 2017.

[48] M. Javaheripi, M. Samragh, T. Javidi, and F. Koushanfar, “Genecai: Genetic evolution for acquiring compact ai,” *arXiv preprint arXiv:2004.04249*, 2020.

[49] M. Samragh, M. Javaheripi, and F. Koushanfar, “Autorank: Automated rank selection for effective neural network customization,” in *Proceedings of the ML-for-Systems Workshop at the 46th International Symposium on Computer Architecture (ISCA’19)*, 2019.

[50] M. Javaheripi, B. D. Rouhani, and F. Koushanfar, "Swnet: Small-world neural networks and rapid convergence," *arXiv preprint arXiv:1904.04862*, 2019.

[51] 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*, vol. 11, no. 4, pp. 23–28, 2019.

[52] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang, "Soft filter pruning for accelerating deep convolutional neural networks," *arXiv preprint arXiv:1808.06866*, 2018.

[53] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proceedings of the IEEE international conference on computer vision*, pp. 1389–1397, 2017.

[54] H. Wang, Q. Zhang, Y. Wang, and H. Hu, "Structured probabilistic pruning for convolutional neural network acceleration," *arXiv preprint arXiv:1709.06994*, 2017.

[55] C. Jiang, G. Li, C. Qian, and K. Tang, "Efficient dnn neuron pruning by minimizing layer-wise nonlinear reconstruction error," in *IJCAI*, pp. 2–2, 2018.

[56] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv:1608.08710*, 2016.

[57] J. Lin, Y. Rao, J. Lu, and J. Zhou, "Runtime neural pruning," in *Advances in Neural Information Processing Systems*, pp. 2181–2191, 2017.

[58] S. Lin, R. Ji, Y. Li, Y. Wu, F. Huang, and B. Zhang, "Accelerating convolutional networks via global & dynamic filter pruning.," in *IJCAI*, pp. 2425–2432, 2018.

[59] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," in *Proceedings of the IEEE international conference on computer vision*, pp. 5058–5066, 2017.

[60] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, pp. 1135–1143, 2015.

[61] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang, "An efficient hardware accelerator for sparse convolutional neural networks on fpgas," in *FCCM*, pp. 17–25, IEEE, 2019.

[62] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and effective sparse lstm on fpga with bank-balanced sparsity," in *FPGA*, pp. 63–72, ACM, 2019.

[63] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

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

[65] Z. Cai, X. He, J. Sun, and N. Vasconcelos, "Deep learning with low precision by half-wave gaussian quantization," *arXiv:1702.00953*, 2017.

[66] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *ECCV*, pp. 525–542, 2016.

[67] X. Lin, C. Zhao, and W. Pan, "Towards accurate binary convolutional neural network," in *NIPS*, pp. 345–353, 2017.

[68] E. Wang, J. J. Davis, P. Y. Cheung, and G. A. Constantinides, "Lutnet: Rethinking inference in fpga soft logic," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 26–34, IEEE, 2019.

[69] J. Fromm, S. Patel, and M. Philipose, "Heterogeneous bitwidth binarization in convolutional neural networks," in *Advances in Neural Information Processing Systems*, pp. 4006–4015, 2018.

[70] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *ISCA*, pp. 535–547, 2017.

[71] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, "Energy-efficient cnn implementation on a deeply pipelined fpga cluster," in *ISLPED*, ACM, 2016.

[72] Z. Liu, Y. Dou, J. Jiang, J. Xu, S. Li, Y. Zhou, and Y. Xu, "Throughput-optimized fpga accelerator for deep convolutional neural networks," *ACM TRETS*, vol. 10, no. 3, p. 17, 2017.

[73] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *IEEE/ACM International Symposium on Microarchitecture*, p. 17, 2016.

[74] "Chaidnn: Hls based deep neural network accelerator library for xilinx ultrascale+ mpsocs," 2018.

[75] J. Kiefer and J. Wolfowitz, "Stochastic estimation of the maximum of a regression function," *The Annals of Mathematical Statistics*, pp. 462–466, 1952.

[76] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[77] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[78] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, pp. 770–778, 2016.

[79] A. T. Elthakeb, P. Pilligundla, A. Yazdanbakhsh, S. Kinzer, and H. Esmaeilzadeh, "Releq: A reinforcement learning approach for deep quantization of neural networks," *arXiv preprint arXiv:1811.01704*, 2018.

[80] B. Zhuang, C. Shen, M. Tan, L. Liu, and I. Reid, "Towards effective low-bitwidth convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 7920–7928, 2018.

[81] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[82] 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.

[83] M. Imani, Y. Kim, A. Rahimi, and T. Rosing, "Acam: Approximate computing based on adaptive associative memory with online learning," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pp. 162–167, 2016.

[84] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 529–540, 2014.

[85] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International conference on machine learning*, pp. 2849–2858, PMLR, 2016.

[86] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, "Neural networks with few multiplications," *arXiv preprint arXiv:1510.03009*, 2015.

[87] D. D. Lin and S. S. Talathi, "Overcoming challenges in fixed point training of deep convolutional networks," *arXiv preprint arXiv:1607.02241*, 2016.

[88] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[89] B. D. Rouhani, A. Mirhoseini, and F. Koushanfar, "Delight: Adding energy dimension to deep neural networks," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pp. 112–117, 2016.

[90] M. Imani, A. Rahimi, and T. S. Rosing, "Resistive configurable associative memory for approximate computing," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1327–1332, IEEE, 2016.

[91] X. Yin, A. Aziz, J. Nahas, S. Datta, S. Gupta, M. Niemier, and X. S. Hu, "Exploiting ferroelectric fets for low-power non-volatile logic-in-memory circuits," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE, 2016.

[92] M. V. Beigi and G. Memik, "Tapas: Temperature-aware adaptive placement for 3d stacked hybrid caches," in *Proceedings of the Second International Symposium on Memory Systems*, pp. 415–426, 2016.

[93] Y. Kim, M. Imani, S. Patil, and T. S. Rosing, "Cause: Critical application usage-aware memory system using non-volatile memory for mobile devices," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 690–696, IEEE, 2015.

[94] M. Imani, S. Patil, and T. S. Rosing, "Masc: Ultra-low energy multiple-access single-charge tcam for approximate computing," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 373–378, IEEE, 2016.

[95] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[96] A. Krogh and J. A. Hertz, "A simple weight decay can improve generalization," in *Advances in neural information processing systems*, pp. 950–957, 1992.

[97] K.-H. Kim, S. Gaba, D. Wheeler, J. M. Cruz-Albrecht, T. Hussain, N. Srinivasa, and W. Lu, "A functional hybrid memristor crossbar-array/cmos system for data storage and neuromorphic applications," *Nano letters*, vol. 12, no. 1, pp. 389–395, 2012.

[98] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: A simulation framework for cpu-gpu computing," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 335–344, IEEE, 2012.

[99] D. Compiler, R. User, and M. Guide, "Synopsys," *See: http://www.synopsys.com*, 2000.

[100] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[101] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[102] "Uci machine learning repository." `http://archive.ics.uci.edu/ml/datasets/ISOLET`.

[103] "Hyperspectral remote sensing scenes." `http://www.ehu.eus/ccwintco/index.php?title=Hyperspectral_Remote_Sensing_Scenes`.

[104] "Uci machine learning repository." `http://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones`.

[105] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database of handwritten digits," 1998.

[106] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, pp. 1139–1147, PMLR, 2013.

[107] A. Esteva, A. Robicquet, B. Ramsundar, V. Kuleshov, M. DePristo, K. Chou, C. Cui, G. Corrado, S. Thrun, and J. Dean, "A guide to deep learning in healthcare," *Nature medicine*, vol. 25, no. 1, p. 24, 2019.

[108] The HIPAA Privacy Rule. `https://www.hhs.gov/hipaa/for-professionals/privacy/index.html`.

[109] E. Hesamifard, H. Takabi, and M. Ghasemi, "Cryptodl: Deep neural networks over encrypted data," *arXiv preprint arXiv:1711.05189*, 2017.

[110] A. Brutzkus, R. Gilad-Bachrach, and O. Elisha, "Low latency privacy preserving inference," in *International Conference on Machine Learning*, pp. 812–821, PMLR, 2019.

[111] F. Bourse, M. Minelli, M. Minihold, and P. Paillier, "Fast homomorphic evaluation of deep discretized neural networks," in *Annual International Cryptology Conference*, pp. 483–512, Springer, 2018.

[112] E. Chou, J. Beal, D. Levy, S. Yeung, A. Haque, and L. Fei-Fei, "Faster cryptonets: Leveraging sparsity for real-world encrypted inference," *arXiv preprint arXiv:1811.09953*, 2018.

[113] A. Sanyal, M. Kusner, A. Gascon, and V. Kanade, "Tapas: Tricks to accelerate (encrypted) prediction as a service," in *International Conference on Machine Learning*, pp. 4490–4499, PMLR, 2018.

[114] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "Chet: an optimizing compiler for fully-homomorphic neural-network inferencing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 142–156, 2019.

[115] M. Ball, B. Carmer, T. Malkin, M. Rosulek, and N. Schimanski, "Garbled neural networks are practical.," *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 338, 2019.

[116] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious neural network predictions via minionn transformations," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 619–631, 2017.

[117] Z. Ghodsi, A. Veldanda, B. Reagen, and S. Garg, "Cryptonas: Private inference on a relu budget," in *Advances in Neural Information Processing Systems*, 2020.

[118] C. Gentry, “Fully homomorphic encryption using ideal lattices,” in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pp. 169–178, 2009.

[119] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, “CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy,” in *International Conference on Machine Learning*, 2016.

[120] B. Li and D. Micciancio, “On the security of homomorphic encryption on approximate numbers,” *IACR Cryptol. ePrint Arch*, vol. 2020, p. 1533, 2020.

[121] S. Bian, M. Hiromoto, and T. Sato, “Darl: Dynamic parameter adjustment for lwe-based secure inference,” in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1739–1744, IEEE, 2019.

[122] N. Agrawal, A. Shahin Shamsabadi, M. J. Kusner, and A. Gascón, “Quotient: two-party secure neural network training and prediction,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1231–1247, 2019.

[123] A. Dalskov, D. Escudero, and M. Keller, “Secure evaluation of quantized neural networks,” *Proceedings on Privacy Enhancing Technologies*, vol. 2020, no. 4, pp. 355–375, 2020.

[124] S. Wu, G. Li, F. Chen, and L. Shi, “Training and inference with integers in deep neural networks,” in *International Conference on Learning Representations*, 2018.

[125] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2704–2713, 2018.

[126] L. Xie and A. Yuille, “Genetic cnn,” in *Proceedings of the IEEE international conference on computer vision*, pp. 1379–1388, 2017.

[127] P. Moengin, “Exponential penalty methods for solving linear programming problems,” in *Proceedings of the World Congress on Engineering and Computer Science*, vol. 2, 2011.

[128] T. Bäck, D. B. Fogel, and Z. Michalewicz, *Handbook of evolutionary computation*. CRC Press, 1997.

[129] X. Wang, A. J. Malozemoff, and J. Katz, “EMP-toolkit: Efficient MultiParty computation toolkit.” `https://github.com/emp-toolkit`, 2016.

[130] S. U. Hussain, B. Li, F. Koushanfar, and R. Cammarota, “TinyGarble2: Smart, Efficient, and Scalable Yao’s Garble Circuit,” in *ACM Workshop on Privacy-Preserving Machine Learning in Practice(PPMLP)*, 2020.

[131] G. Guennebaud, B. Jacob, *et al.*, “Eigen v3.” http://eigen.tuxfamily.org, 2010.

[132] J. Liu, M. Juuti, Y. Lu, and N. Asokan, “Oblivious neural network predictions via MiniONN transformations,” in *CCS*, ACM, 2017.

[133] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor, “Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation,” in *EuroS&P)*, pp. 112–127, IEEE, 2016.

[134] J. Bethge, C. Bartz, H. Yang, Y. Chen, and C. Meinel, “Meliusnet: Can binary neural networks achieve mobilenet-level accuracy?,” *arXiv preprint arXiv:2001.05936*, 2020.

[135] Z. Liu, Z. Shen, M. Savvides, and K.-T. Cheng, “Reactnet: Towards precise binary neural network with generalized activation functions,” in *European Conference on Computer Vision*, pp. 143–159, Springer, 2020.

[136] W. Tang, G. Hua, and L. Wang, “How to train a compact binary neural network with high accuracy?,” in *AAAI*, pp. 2625–2631, 2017.

[137] L. Liu and J. Deng, “Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution,” in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[138] J. Yu, L. Yang, N. Xu, J. Yang, and T. Huang, “Slimmable neural networks,” *arXiv preprint arXiv:1812.08928*, 2018.

[139] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1,” *arXiv preprint arXiv:1602.02830*, 2016.

[140] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, “Chameleon: A hybrid secure computation framework for machine learning applications,” in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pp. 707–721, 2018.

[141] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi, “Ezpc: programmable, efficient, and scalable secure two-party computation for machine learning,” *ePrint Report*, vol. 1109, 2017.

[142] FaceScrub, *The FaceScrub dataset*, 2020. `http://engineering.purdue.edu/~mark/puthesis`, (accessed July 3, 2020).

[143] H.-W. Ng and S. Winkler, “A data-driven approach to cleaning large face datasets,” in *IEEE international conference on image processing*, 2014.

[144] “Malaria Cell Images, accessed on 01/20/2019.” `https://www.kaggle.com/iarunava/cell-images-for-detecting-malaria`.

[145] Y. Liu, Y. Xie, and A. Srivastava, “Neural trojans,” in *2017 IEEE International Conference on Computer Design (ICCD)*, pp. 45–48, IEEE, 2017.

[146] M. Charikar, J. Steinhardt, and G. Valiant, "Learning from untrusted data," in *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pp. 47–60, 2017.

[147] C. Liu, B. Li, Y. Vorobeychik, and A. Oprea, "Robust linear regression against training data poisoning," in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pp. 91–102, 2017.

[148] B. Tran, J. Li, and A. Madry, "Spectral signatures in backdoor attacks," in *Advances in Neural Information Processing Systems*, pp. 8000–8010, 2018.

[149] 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*, pp. 273–294, Springer, 2018.

[150] 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.

[151] 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.

[152] S. Ma and Y. Liu, "Nic: Detecting adversarial samples with neural network invariant checking," in *Proceedings of the 26th Network and Distributed System Security Symposium (NDSS 2019)*, 2019.

[153] 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 *Proceedings of the 35th Annual Computer Security Applications Conference*, pp. 113–125, 2019.

[154] 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*, vol. 29, no. 7, pp. 2717–2730, 2017.

[155] G. M. Davis, S. G. Mallat, and Z. Zhang, "Adaptive time-frequency decompositions," *Optical engineering*, vol. 33, no. 7, pp. 2183–2192, 1994.

[156] D. Salomon, *Data compression: the complete reference*. Springer Science & Business Media, 2004.

[157] B. Stellato, B. P. Van Parys, and P. J. Goulart, "Multivariate chebyshev inequality with estimated mean and variance," *The American Statistician*, vol. 71, no. 2, pp. 123–127, 2017.

[158] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition," *Neural networks*, vol. 32, pp. 323–332, 2012.

[159] O. M. Parkhi, A. Vedaldi, and A. Zisserman, “Deep face recognition,” 2015.

[160] M. Fredrikson, S. Jha, and T. Ristenpart, “Model inversion attacks that exploit confidence information and basic countermeasures,” in *ACM CCS*, 2015.

[161] G. B. Huang, M. Mattar, H. Lee, and E. Learned-Miller, “Learning to align from scratch,” in *NIPS*, 2012.