UNIVERSITY OF CALIFORNIA, SAN DIEGO

Succinct and Assured Machine Learning: Training and Execution

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

in

Electrical Engineering (Computer Engineering)

by

Bita Darvish Rouhani

Committee in charge:

Professor Farinaz Koushanfar, Chair
Professor Hadi Esmaeilzadeh
Professor Tara Javidi
Professor Truong Nguyen
Professor Bhaskar Rao
Professor Tajana Simunic Rosing

2018

The dissertation of Bita Darvish Rouhani is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2018

DEDICATION

To my beloved parents Mahin and Behrouz.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere appreciation to my advisor Prof. Farinaz Koushanfar for her priceless support, encouragement, and guidance. Her dedication, fondness, and motivation towards doing novel research, as well as exceptional care for her students have taught me invaluable lessons and greatly influenced my life at both academic and personal levels. I will always be candidly grateful for all her advice and support.

I would like to thank my committee members, Prof. Hadi Esmaeilzadeh, Prof. Tara Javidi, Prof. Truong Nguyen, Prof. Bhaskar Rao, and Prof. Tajana Simunic Rosing for taking the time to be part of my committee and for their valuable comments and suggestions. I would also like to sincerely thank my mentors at Microsoft Research, Dr. Doug Burger and Dr. Eric Chung, for their invaluable support and guidance. Working with them was a great opportunity for me to explore the link between research and building first-class technologies.

My experience of Ph.D. was made a lot more delightful because of the brilliant people I was lucky enough to get to know them and/or collaborate with them. In particular, I would like to thank Dr. Azalia Mirhoseini, Dr. Ebrahim Songhori, Mojan Javaheripi, Huili Chen, Mohammad Samragh, Siam Umar Hussain, Mohammad Ghasemzadeh, Salar Yazdjerdi, Fang Lin, Somayeh Imani, Amir Yazdanbakhsh, and Bahar Salimian for all their helps and the happy times we had together. Last but not the least, I wish to express my profound admiration and gratitude to my beloved parents and brothers for their endless love, for believing in me, for inspiring me to dream big and work hard to follow them, and for showing me constant support even when we were physically far apart.

The material in this dissertation is based on the following papers which are either published, under review, or in preparation for submission.

Chapter 1, in part, (i) has been published at IEEE Security and Privacy (S&P) Magazine 2018 as Bita Darvish Rouhani, Mohammad Samragh, Tara Javidid, and Farinaz Koushanfar "Safe Machine Learning and Defeating Adversarial Attacks", and (ii) has been submitted to

Communication of ACM as Bita Darvish Rouhani, Azalia Mirhoseini, and Farinaz Koushanfar "Succinct Training and Execution of Deep Learning on Edge Devices: Depth-First Distributed Graph Traversal, Data Embedding, and Resource Parallelism". The dissertation author was the primary author of this material.

Chapter 2 and 3, in part, has been published at (i) the Proceedings of 2017 International Design Automation Conference (DAC) and appeared as: Bita Darvish Rouhani, Azalia Mirhoseini, and Farinaz Koushanfar "Deep3: Leveraging Three Levels of Parallelism for Efficient Deep Learning", and (ii) the Proceedings of the 2016 International Symposium on Low Power Electronics and Design (ISLPED) as: Bita Darvish Rouhani, Azalia Mirhoseini, and Farinaz Koushanfar "Delight: Adding energy dimension to deep neural networks". The dissertation author was the primary author of this material.

Chpater 4, in part, has been published at (i) the Proceedings of 2018 International Conference On Computer Aided Design (ICCAD) and appeared as: Bita Darvish Rouhani, Mohammad Samragh, Mojan Javaheripi, Tara Javidid, and Farinaz Koushanfar "DeepFense: Online Accelerated Defense Against Adversarial Deep Learning", and (ii) IEEE Security and Privacy (S&P) Magazine 2018 as ita Darvish Rouhani, Mohammad Samragh, Tara Javidid, and Farinaz Koushanfar "Safe Machine Learning and Defeating Adversarial Attacks". The dissertation author was the primary author of this material.

Chapter 5, in part, has been published at arXiv preprint arXiv:1804.00750, 2018 as: Bita Darvish Rouhani, Huili Chen, and Farinaz Koushanfar "Deepsigns: A generic watermarking framework for ip protection of deep learning models". The dissertation author was the primary author of this material.

Chapter 6, in part, has been published at the Proceedings of the 2018 ACM International Symposium on Design Automation Conference (DAC) and appeared as: Bita Darvish Rouhani, Sadegh Riazi, and Farinaz Koushanfar "DeepSecure: Scalable Provably-Secure Deep Learning". The dissertation author was the primary author of this material.

Chapter 2 and 7, in part, has been published at ACM Transactions on Reconfigurable Technology and Systems (TRETS) 2018 as: Bita darvish Rouhani, Siam U Hussain, Kristin Lauter, and Farinaz Koushanfar "ReDCrypt: Real-Time Privacy-Preserving Deep Learning Inference in Clouds Using FPGAs" and the Proceedings of the 2018 ACM International Symposium on Design Automation Conference (DAC) and appeared as: Siam U Hussain, Bita Darvish Rouhani, Mohammad Ghasemzadeh, and Farinaz Koushanfar "MAXelerator: FPGA accelerator for privacy preserving multiply-accumulate (MAC) on cloud servers". The dissertation author was the primary author of the ReDCrypt paper and the secondary author of MAXelerator paper. ReDCrypt is particularly designed for deep learning models and MAXelerator is a generic privacy preserving matrix multiplication framework that is designed in collaboration with Siam U Hussain.

Chapter 2 and 8, This chapter, in part, has been published at the Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA) and appeared as: Bita Darvish Rouhani, Mohammad Ghasemzadeh, and Farinaz Koushanfar "CausaLearn: Automated Framework for Scalable Streaming-based Causal Bayesian Learning using FPGAs". The dissertation author was the primary author of this material.

This dissertation was supported, in parts, by the ONR (N00014-11-1-0885), NSF (CNS-1619261), NSF TrustHub (1649423), and Microsoft Research Ph.D. fellowship grants.

VITA

2013	Bachelor of Science in Electrical Engineering, Sharif University of Technology, Tehran, Iran
2015	Master of Science in Computer Engineering, Rice University, Houston, Texas
2015-2018	Graduate Research Assistant, University of California San Diego, La Jolla, California
2018	Doctor of Philosophy in Electrical Engineering (Computer Engineering), University of California San Diego, La Jolla, California

PUBLICATIONS

B. Rouhani, M. Ghasemzadeh, and F. Koushanfar. "Automated scalable framework for streaming-based causal Bayesian learning using FPGAs." In 26th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), 2018.

B. Rouhani, M. Samragh, T. Javidi, and F. Koushanfar. "Safe Machine Learning and Defeating Adversarial Attacks." IEEE Security and Privacy (S&P) magazine, 2018.

B. Rouhani, M. Samragh, M. Javaheripi, T. Javidi, and F. Koushanfar. "DeepFense: Online Accelerated Defense Against Adversarial Deep Learning." International Conference On Computer Aided Design (ICCAD), 2018.

B. Rouhani, Siam Umar Hussain, Kristin Lauter, and Farinaz Koushanfar. "ReDCrypt: Real-Time Privacy Preserving Deep Learning Using FPGAs." ACM Transactions on Reconfigurable Technology and Systems (TRETS), 2018.

B. Rouhani, H. Chen, and F. Koushanfar. "DeepSigns: A Generic Framework for Watermarking and IP Protection of Deep Learning Models." ArXiv Preprint arXiv:1804.00750, 2018.

H. Chen, **B. Rouhani**, and F. Koushanfar. "DeepMarks: A Digital Fingerprinting Framework for Deep Neural Networks." ArXiv Preprint arXiv:1804.03648, 2018.

S. Hussain, **B. Rouhani**, M. Ghasemzadeh, and F. Koushanfar. "MAXelerator: FPGA Accelerator for Privacy Preserving Multiply-Accumulate (MAC) on Cloud Servers." In Proceedings of Design Automation Conference (DAC), 2018.

M. Ghasemzadeh, F. Lin, **B. Rouhani**, F. Koushanfar, and K. Huang. "AgileNet: Lightweight Dictionary-based Few-shot Learning." ArXiv Preprint 1805.08311, 2018.

B. Rouhani, A. Mirhoseini, and F. Koushanfar. "Succinct Training and Execution of Deep Learning on Edge Devices: Depth-First Distributed Graph Traversal, Data Embedding, and Resource Parallelism." Under Review in Communication ACM Magazine, 2018.

B. Rouhani, S. Riazi, and F. Koushanfar. "DeepSecure: Scalable Provably-Secure Deep Learning." In Proceedings of Design Automation Conference (DAC), 2018.

S. Riazi, **B. Rouhani**, and F. Koushanfar. "Privacy Concerns in Deep Learning." IEEE Security and Privacy (S&P) magazine, 2018.

B. Rouhani, A. Mirhoseini, and F. Koushanfar. "Deep3: Leveraging Three Levels of Parallelism for Efficient Deep Learning." In Proceedings of Design Automation Conference (DAC), 2017.

B. Rouhani, A. Mirhoseini, and F. Koushanfar. "RISE: An Automated Framework for Real-Time Intelligent Video Surveillance on FPGA." ACM Transactions on Embedded Computing Systems (TECS), 2017.

A. Mirhoseini, **B. Rouhani**, E. Songhori, and F. Koushanfar. "ExtDict: Extensible Dictionaries for Data- and Platform-Aware Large-Scale Learning." In Proceedings of International Parallel & Distributed Processing Symposium (IPDPS) ParLearning workshop, 2017.

B. Rouhani, M. Ghasemzadeh, and F. Koushanfar. "Real-time Causal Internet Log Analytics by HW/SW/Projection Co-design." Hardware Demo in Proceedings of IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2017.

B. Rouhani, A. Mirhoseini, and F. Koushanfar. "TinyDL: Just-in-Time Deep Learning Solution for Constrained Embedded Systems." In Proceedings of International Symposium on Circuits & Systems (ISCAS), 2017.

B. Rouhani, A. Mirhoseini, and F. Koushanfar. "DeLight: Adding Energy Dimension to Deep Neural Networks." In Proceedings of International Symposium on Low Power Electronics and Design (ISLPED), 2016.

B. Rouhani, A. Mirhoseini, E. Songhori, and F. Koushanfar. "Automated Real-Time Analysis of Streaming Big and Dense Data on Reconfigurable Platforms." ACM Transactions on Reconfigurable Technology and Systems (TRETS), 2016.

B. Rouhani, A. Mirhoseini, and F. Koushanfar. "Going Deeper than Deep Learning for Massive Data Analytics under Physical Constraints." In Proceedings of International Conference on Hardware/Software Co-design and System Synthesis (CODES+ISSS), 2016.

A. Mirhoseini, **B. Rouhani**, E. Songhori, and F. Koushanfar. "Chime: Checkpointing Long Computations on Intermittently Energized IoT Device."' IEEE Transactions on Multi-Scale Computing Systems (TMSCS), 2016.

A. Mirhoseini, **B. Rouhani**, E. Songhori, and F. Koushanfar. "PerformML: Performance Optimized Machine Learning by Platform and Content Aware Customization." In Proceedings of Design Automation Conference (DAC), 2016.

B. Rouhani, E. Songhori, A. Mirhoseini, and F. Koushanfar. "SSketch: An Automated Framework for Streaming Sketch-based Analysis of Big Data on FPGA." Field-Programmable Custom Computing Machines (FCCM), 2015.

A. Mirhoseini, E. Songhori, **B. Rouhani**, and F. Koushanfar. "Flexible Transformations for Learning Big Data." Short Paper, ACM Special Interest Group for the Computer Systems Performance Evaluation Conference, (SIGMETRICS), 2015.

ABSTRACT OF THE DISSERTATION

Succinct and Assured Machine Learning: Training and Execution

by

Bita Darvish Rouhani

Doctor of Philosophy in Electrical Engineering (Computer Engineering)

University of California, San Diego, 2018

Professor Farinaz Koushanfar, Chair

Contemporary datasets are rapidly growing in size and complexity. This wealth of data is providing a paradigm shift in various key sectors including defense, commercial, and personalized computing. Over the past decade, machine learning and related fields have made significant progress in designing rigorous algorithms with the goal of making sense of this large corpus of available data. Concerns over physical performance (runtime and energy consumption), reliability (safety), and ease-of-use, however, pose major roadblocks to the wider adoption of machine learning techniques. To address the aforementioned roadblocks, a popular recent line of research is focused on performance optimization and machine learning acceleration via hardware/software co-design and automation. This thesis advances the state-of-the-art in this growing field by

advocating a *holistic automated co-design* approach which involves not only *hardware* and *software* but also the *geometry of the data* and *learning model* as well as the *security requirements*. My key contributions include:

- Co-optimizing graph traversal, data embedding, and resource allocation for succinct training and execution of Deep Learning (DL) models. The resource efficiency of my end-to-end automated solutions not only enables compact DL training/execution on edge devices but also facilitates further reduction of the training time and energy spent on cloud data servers.

- Characterizing and thwarting adversarial subspace for robust and assured execution of DL models. I build a holistic hardware/software/algorithm co-design that enables just-in-time defense against adversarial attacks. My proposed countermeasure is robust against the strongest adversarial attacks known to date without violating the real-time response requirement, which is crucial in sensitive applications such as autonomous vehicles/drones.

- Proposing the first efficient resource management framework that empowers coherent integration of robust digital watermarks/fingerprints into DL models. The embedded digital watermarks/fingerprints are robust to removal and transformation attacks and can be used for model protection against intellectual property infringement.

- Devising the first reconfigurable and provably-secure framework that simultaneously enables accurate and scalable DL execution on encrypted data. The proposed framework supports secure streaming-based DL computation on cloud servers equipped with FPGAs.

- Developing the first scalable framework that enables real-time approximation of multi-dimensional probability density functions for causal Bayesian analysis. The proposed solution adaptively fine-tunes the underlying latent variables to cope with the data dynamics as it evolves over time.

Chapter 1

Introduction

Computers and sensors generate data at an unprecedented rate. Analyzing massive and densely-correlated data is an omnipresent trend on different computing platforms. There are (at least) two major sets of challenges that need to be addressed simultaneously to build a learning system that is both sustainable and trustworthy. One set of hurdles is related to the physical resources and/or application-specific constraints such as real-time requirements, available energy, and/or memory bandwidth. The other set of challenges arises due to the entanglement of high-dimensional data. On the one hand, this data entanglement makes it necessary to go beyond traditional linear or polynomial analytics to reach a certain level of accuracy. On the other hand, the complexity of contemporary machine learning models makes them prone to certain degree of nuisance variables that are not necessarily the key features used by human brains. These non-intuitive nuisance variables, in turn, can be leveraged by adversaries to fool the underlying machine learning agent. As such, it is critical to also assure model robustness in the face of adversarial attacks while minimally affecting the underlying system performance.

This thesis addresses the aforementioned two critical aspects of emerging computing scenarios by designing and building holistic solutions and tools that are simultaneously co-optimized for the underlying data geometry, coarse-grained model parallelism, hardware characteristics, and

security requirements (Figure 1.1). My holistic solutions provide a promising avenue to improve the performance of existing cloud-based services. They also unlock new capabilities and services (e.g., significantly longer battery life) for embedded Internet-of-Things (IoT) devices.

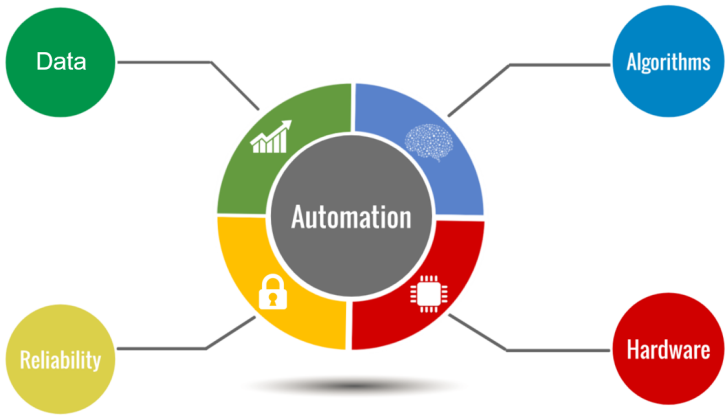


Figure 1.1: Research overview. My work enables the next generation of cyber-physical applications by devising holistic computing frameworks that are simultaneously optimized for the underlying data, learning algorithm, hardware, and security requirements.

My research work is focused on performance optimization for two of the state-of-the-art classes of machine learning namely *Deep Learning (DL)* and *causal Bayesian analysis*. In this section, I highlight the challenges that arise while solving these problems and review prior works. I then discuss my proposed holistic solutions and describe their versatility and broader impact.

1.1 Resource-Efficient and Trusted Deep Learning

Deep learning has become the key methodology for achieving the state-of-the-art accuracy by moving beyond traditional linear or polynomial analytics [DY14]. Despite DL's powerful learning capability, the computational overhead associated with DL methods has hindered their applicability to resource-constrained settings. Examples of such settings range from embedded devices to data centers where physical viability (e.g., runtime and energy efficiency) is a standing challenge. In the following, I first discuss my work on devising resource-efficient DL systems. I then discuss my work to promote reliability and user privacy in DL applications.

1.1.1 Succinct Training and Execution of Deep Neural Networks

Deep learning models are conventionally trained on big data servers. In applications where DL execution on the edge is needed, the trained models are compacted to fit within the constraints of the executing device. Several previous works have demonstrated a significant amount of redundancy in DL models, which allows compacting of pre-trained models on edge devices without considerable loss of accuracy. However, training/fine-tuning of models on large cloud servers assumes constant access to high-performance computing platforms and fast connectivity. Furthermore, in applications where data is collected on edge devices, both the sensor data and model have to be transferred over a remote connection. This would lead to exposure of the data and model in plaintext, a serious security concern for sensitive tasks.

I have introduced, realized, and automated the first resource-aware deep learning framework (called Deep3) that achieves orders of magnitude performance efficiency for training and execution of DL networks on various families of platforms including embedded GPUs, and FPGAs, as well as distributed multi-core CPUs [RMK17a, RMK16]. The rationale in this work is that models which inherently have a high degree of redundancy can be originally trained to be compact and not include unnecessary repetitive parts. Such an approach not only introduces a paradigm shift in model building on edge devices without the overhead of communicating with the cloud but also enables additional reduction of the training time and energy spent on cloud data servers. Even execution time and energy efficiency could be significantly revamped compared to conventional post-facto compacting of large redundant models.

Figure 1.2 compares existing deep learning frameworks in terms of their high-level characteristics. More specifically, the state-of-the-art frameworks for DL optimization and acceleration can be divided into three main categories.

▪ **Data and algorithm optimization.** Data scientists and engineers have primarily focused on DL complexity reduction from an algorithmic/data point of view with limited or no attention to the target hardware characteristics [CHW+13, CSAK14, DCM+12]. Algorithmic and data

	DL Execution	DL Training	DL Model Parallelism	Data Subspace Parallelism	HW/SW Co-design	Non-recurring Engineering Cost	Multi-platform Support
Deep3 (DAC'17)	✓	✓	✓	✓	✓	✓	✓
DistBelief (NIPS'12)	✓	✓	✓	x	x	x	x
Adam (OSDI'14)	✓	✓	✓	x	x	✓	x
DnnWeaver (Micro'16)	✓	x	x	x	✓	✓	x
EIE (ISCA'16)	✓	x	x	x	✓	x	x
FP-DNN (FPGA'17)	✓	x	x	x	✓	✓	x
Xnor-Net (ECCV'16)	✓	x	x	x	x	x	x
NetAdapt (ArXiv'18)	✓	x	x	x	✓	✓	x

Figure 1.2: Comparison of several state-of-the-art deep learning frameworks in terms of their high-level characteristics and features.

optimization without insight from underlying hardware constraints, however, are not sufficient for the succinct realization of DL models. As an example, consider the SqueezeNet network in which the number of weights in the AlexNet model is reduced by a factor of 50 with the goal of an efficient DL inference [IHM+16]. This work and similar DL pruning techniques have been developed based on an implicit assumption that fewer weights result in a more efficient realization. This assumption is not generic across various hardware platforms. In fact, as shown on several embedded devices, SqueezeNet takes approximately 30% more energy compared to the original AlexNet [Mol16, YCS16]. This is particularly because, although a smaller number of weights in a neural network directly reduces the pertinent memory storage requirement, it does not necessarily translate to less energy consumption (i.e., more battery life). In such cases, the size of feature maps and memory access pattern are the dominant factors for energy cost on most embedded platforms. Therefore, it is imperative to include platform-awareness in algorithms to adjust the performance for the target hardware with minimal involvement of human experts.

▪ **Hardware optimization and acceleration.** The development of domain-customized hardware accelerated solutions for efficient implementation of DL models is a key approach taken by several computer engineering researchers [ZLS+15, JGD+14]. Although this line of work has demonstrated significant improvement in deployment of specific DL applications, it has

several restrictions mainly stem from the inflexibility of custom solutions for the realization of other applications. Model and content optimization is often data-dependent and, as such, the methodologies for custom model compaction do not transform into other domains. We believe that by automatic integration of the algorithm and data subspace geometries one can take full advantage of potential opportunities for succinct learning.

- **Algorithm, data, and hardware co-optimization**. Designing automated hardware-aware graph traversal and data embeddings could highly benefit the physical performance of the underlying DL task. I propose Deep3, an automated system that simultaneously leverages three levels of parallelism, namely, data, model, and hardware, for succinct training and execution of neural networks in resource-constrained settings. In particular, I introduce a new extensible and resource-aware graph traversal methodology that (i) allows time multiplexing of the limited resources on the edge and (ii) balances the computation and communication workload for distributed training of large-scale DL models. Deep3 reports the first instance of DL training on embedded GPUs with orders of magnitude runtime and energy improvement achieved by holistic customization to the limits of the hardware resources, data embedding, and DL models [RMK17a, RMK16]. More recently, the NetAdapt framework [YHC+18] is proposed by Google, in which automated customization is performed to adjust the target DL model in accordance with target hardware for efficient DL execution. Unlike Deep3, NetAdapt does not provide DL training.

1.1.2 Assured Deep Neural Networks Against Adversarial Attacks

Reliability and safety consideration is the biggest obstacle to the wide-scale adoption of emerging learning algorithms in sensitive scenarios such as intelligent transportation, healthcare, warfare, and financial systems. Although deep learning models deliver high accuracy in conventional settings with limited simulated input samples, recent research in adversarial DL has shed light on the unreliability of their decisions in real-world scenarios. For instance, consider a traffic sign classifier used in self-driving cars. Figure 1.3 shows an example adversarial sample

where the attacker carefully adds imperceptible perturbation to the input image to mislead the employed DL model, and thus, jeopardizes the safety of the vehicle.

Figure 1.3: The left image is a legitimate "stop" sign sample that is classified correctly by an ML model. The right image, however, is an adversarial input crafted by adding a particular perturbation that makes the same model classify it as a "yield" sign.

I introduce, implement, and automate a novel countermeasure called Modular Robust Redundancy (MRR) to thwart the potential adversarial space and significantly improve the reliability of a victim DL model [RSJ+18, RSJK18a]. Unlike prior defense strategies, MRR methodology is based on unsupervised learning, meaning that no particular adversarial sample is leveraged to build/train the modular redundancies. Instead, my unsupervised learning methodology leverages the structure of the built model and characterizes the distribution of the high dimensional space in the training data. Adopting an unsupervised learning approach, in turn, ensures that the proposed detection scheme can be generalized to a wide class of adversarial attacks. Combined with my resource-efficient DL implementation tool, I build a holistic end-to-end DL system that not only is succinct and accurate but also its integrity is assured against adversarial attacks.

Adversarial samples have already exposed the vulnerability of DL models to malicious attacks; thereby undermining the integrity of autonomous systems built upon deep learning. It is critical to ensure the reliability of DL models in the early development stage instead of looking back with regret when the machine learning systems are compromised by adversaries. My work, in turn, empowers coherent integration of safety consideration into the design process of DL models while minimally affecting the pertinent physical performance in terms of runtime (latency) and/or energy consumption.

1.1.3 Watermarking of Deep Neural Networks

Training a highly accurate DL model requires: (i) having access to a massive collection of mostly proprietary labeled data that furnishes comprehensive coverage of potential scenarios in the target application; (ii) allocating substantial computing resources to fine-tune the underlying model topology (i.e., type and number of hidden layers), hyper-parameters (i.e., learning rate, batch size, etc.), and weights to obtain the most accurate model. Given the costly process of DL training, models are typically considered to be the Intellectual Property (IP) of the model builder.

Model protection against IP infringement is particularly important to preserve the competitive advantage of the owner and ensure the receipt of continuous query requests from clients. Embedding digital watermarks into DL models is a key enabler for reliable technology transfer. Digital watermarks have been immensely leveraged over the past decade to protect the ownership of multimedia and video content, as well as functional artifacts such as digital integrated circuits [FK04, HK99, QP07, CKLS97, Lu04]. Extension of watermarking techniques to DL networks, however, is still in its infancy to enable reliable model distribution. Moreover, adding digital watermarks further presses the already constrained memory for DL training. As such, efficient resource management to minimize the overhead of watermarking is a standing challenge.

Authors in [UNSS17, NUSS18] propose an N-bit ($N > 1$) watermarking approach for embedding the IP information in the *static* content (i.e., weight matrices) of convolutional neural networks. Although this work provides a significant leap as the first attempt to watermark DL networks, it poses (at least) two limitations: (i) It incurs a bounded watermarking capacity due to the use of the static content of the model (weights) as opposed to using dynamic content (activations). The weights of a neural network are invariable (static) during the execution phase, regardless of the data passing through the model. The activations, however, are dynamic and both *data-* and *model-dependent*. We argue that using activations (instead of weights) provides more flexibility for watermarking. (ii) It is not robust against attacks such as overwriting the original embedded watermark by a third party. As such, the original watermark can be removed by an

adversary that is aware of the watermarking method used by the model owner.

More recent studies in [MPT17, ABC+18] propose 1-bit watermarking methodologies for deep learning models. These approaches are built upon model boundary modification and the use of random adversarial samples that lie near decision boundaries. Adversarial samples are known to be statistically unstable, meaning that adversarial samples crafted for a model are not necessarily misclassified by another network [GMP+17, RSJK18b]. Therefore, even though the proposed approaches in [MPT17, ABC+18] yield a high watermark detection rate (true positive rate), they are also too sensitive to hyper-parameter tuning and usually lead to a high false alarm rate. Note that false ownership proofs jeopardize the integrity of the proposed watermarking methodology and render the use of watermarks for IP protection ineffective.

I propose DeepSigns, the first end-to-end IP protection framework that enables developers to systematically insert digital watermarks in the pertinent DL model before distributing the model. DeepSigns is encapsulated as a high-level wrapper that can be leveraged within common deep learning frameworks including TensorFlow, PyTorch, and Theano. Unlike prior works that directly embed the watermark information in the static content (weights) of the pertinent model, DeepSigns works by embedding an arbitrary N-bit ($N \geq 1$) string into the probability density function (pdf) of the activation maps in various layers of a deep neural network. Our proposed watermarking methodology is simultaneously *data-* and *model-dependent*, meaning that the watermark information is embedded in the dynamic content of the DL network and can only be triggered by passing specific input data to the model. DeepSigns' methodology can demonstrably withstand various removal and transformation attacks, including model pruning, model fine-tuning, and watermark overwriting. Figure 1.4 provides a high-level comparison between the state-of-the-art DL watermarking frameworks. As we demonstrate in [CRK18] DeepSigns' methodology can be extended for efficient DL fingerprinting as well.

	DNN Watermarking						
	Output Layer	Hidden Layer	Resource Management API	Data & Model Aware	Reliability	Integrity	Capacity
DeepSigns	✓	✓	✓	✓	✓	✓	N-bit with N≥1
Uchida et al. (2017)	x	✓	x	x	✓	✓	N-bit with N≥1
Merrer&Perez (2017)	✓	x	x	x	✓	x	1-bit
Adi, et al. (2018)	✓	x	x	x	✓	x	1-bit

Figure 1.4: High-level comparison between state-of-the-art watermarking frameworks for deep neural networks. DeepSigns' framework is significantly more robust against removal and transformation attacks, including model pruning, model fine-tuning, and watermark overwriting. DeepSigns' highly-optimized resource management tool, in turn, enables efficient training watermarked neural networks with an extra overhead as low as 2.2%.

1.1.4 Privacy-Preserving Deep Learning

Deep learning models are increasingly incorporated into the *cloud business* to improve the functionality (e.g., accuracy) of the service. A complicating factor in the rush to adopt DL as a cloud service is the data and model privacy. On the one hand, DL models are usually trained by allocating significant computational resources to process massive amounts of training data. As such, the trained models are considered an intellectual property of companies which require confidentiality to preserve the competitive advantage and ensure receiving continuous query requests by clients. On the other hand, clients do not desire to send their private data (e.g., location or financial input) in plain text to cloud servers due to the risk of information leakage.

To incorporate deep learning into the cloud services, it is highly desired to devise privacy-preserving frameworks in which neither of the involving parties is required to reveal their private information. Several research works have been developed to address privacy-preserving computing for DL networks, e.g., [GBDL$^+$16, MZ17]. The existing solutions, however, either: (i) rely on the modification of DL layers (such as non-linear activation functions) to efficiently compute the specific cryptographic protocols. For instance, authors in [GBDL$^+$16, MZ17] have suggested the use of polynomial-based Homomorphic encryption to make the client's data oblivious to the server. Their approach requires changing the non-linear activation functions to some polynomial approximation (e.g., square) during training. Such modification, in turn, can

	Accuracy Preserving	Independency of Third-party Server	Data & Network Embedding	Supporting Distributed Clients	Support for Large Batch Sizes	Scalability for Large DL Models	Security Primitive
DeepSecure	✓	✓	✓	✓	x	✓	GC
CryptoNets (ICML'16)	x	✓	x	x	✓	x	Leveled HE
SecureML (S&P'17)	x	x	x	✓	x	✓	Linearly HE, GC, SS
MiniONN (CCS'17)	✓	✓	x	✓	x	✓	Additively HE, GC, SS

Figure 1.5: High-level characteristics of existing frameworks for privacy-preserving execution of deep learning models and their corresponding cryptographic primitives.

reduce the ultimate accuracy of the model and poses a trade-off between the model accuracy and execution cost of the privacy-preserving protocol. Or (ii) fall in the two-server settings in which data owners distribute their private data among two non-colluding servers to perform a particular DL inference. The two non-colluding server assumption is not ideal as it requires the existence of a trusted third-party which is not always an option in practical scenarios.

I have proposed DeepSecure, the first provably-secure framework for scalable DL-based analysis of data collected by distributed clients [RRK18]. DeepSecure is well-suited for streaming settings where clients need to dynamically analyze their data as it is collected over time without having to queue the samples to meet a certain batch size (e.g., 2600). The secure DL computation in DeepSecure is performed using Yaos Garbled Circuit (GC) protocol. My GC-optimized solution achieves more than 58-fold higher throughput per sample compared with the Microsofts secure DL framework (called CryptoNet). In addition to the GC-optimized DL realization, I introduced a set of inter-domain pre-processing techniques with insights from the data and algorithms to significantly reduce the GC protocol overhead in the context of deep learning. Extensive evaluations of various DL applications demonstrate up to two orders-of-magnitude additional runtime improvement achieved as a result of the proposed pre-processing methodology. DeepSecure also provides support for secure delegation of GC computations to a third party for clients with severe resource constraints such as embedded IoT and wearable devices. Figure 1.5 provides a high-level comparison of existing frameworks for DL execution on encrypted data.

1.2 Real-Time Causal Bayesian Analysis

Probabilistic learning and graphical modeling of time-series data with causal structure is a grand challenge in various scientific fields, ranging from machine learning, neurophysiology, and climatology to economics, medical imaging, and speech processing. In a variety of time-series applications, real-time dynamic updating of random variables is particularly important to enable effective decision making before the system encounters natural changes, rendering much of the collected data irrelevant to the current decision space.

By many estimates, as much as 80 percent of time-series data is semi-structured or even unstructured. Significant theoretical strides have been made to design Bayesian graphical analytics that can be used to effectively capture the causality structure of dynamic data. These set of works, however, are designed at the algorithmic and data abstraction level with complex data flows and are oblivious to the hardware characteristics. As such, they cannot be readily employed for real-time streaming settings in which the memory storage is limited and high-dimensional time-stamped data is collected from multiple sources at a high frequency. A number of accelerated tools have been reported in the literature to facilitate Bayesian graphical analysis on CPUs, GPUs, and FPGAs. The existing tools, however, are either tailored for a specific application with a restrict conjecture about the prior distribution of data (e.g., considering a simple Gaussian distribution) and/or are designed with a predominant assumption that data samples are independently and identically drawn from a certain distribution. As such, they cannot effectively capture dynamic data correlation in casual streaming applications (e.g., complex correlated time-series data). IBM has recently released a streaming system for managing and analyzing time-series data. This tool is built on up of the IBM InfoSphere general purpose platform and does not provide customized hardware accelerated solution for high-frequency applications with memory storage limitation.

As the synopsis of my prior work suggests, the hardware resource allocation and the algorithmic solution should be co-optimized to achieve the best domain-customized performance.

I introduce CausaLearn, a holistic Algorithm/Hardware/Software co-design approach for scalable, real-time, and automated analysis of high-dimensional time-series data with a causal pattern in time [DRGK18]. The proposed solution leverages concurrent parallel resources on reconfigurable hardware platforms to accelerate time-series data analysis in streaming settings. CausaLearn evades the requirement to store the raw data by adaptively learning/updating the underlying probability density function as data evolves over time and only storing the condense extracted knowledge. CausaLearn and its accompanying APIs, in turn, enable knowledge extraction from complex and unstructured raw data sets and overcome the challenges of real-time data analytics on streaming time-series data while minimizing the non-recurring engineering cost.

1.3 Broader Impact and Re-usability

The broader goal of this thesis is to simultaneously capture the best of Computer Architecture, Machine Learning, and Security fields and get one step closer towards realizing the immense potential of big data. To this end, I have implemented and deployed my research with an emphasize on delivering accompanying Application Programming Interfaces. Throughout my projects, I have adopted an end-to-end design methodology to provide abstractions and accompanying interfaces for rapid prototyping, as well as the proof-of-concept implementation of the proposed methodologies on different computing platforms including CPUs, GPUs, and FPGAs. Please refer to https://github.com/Bitadr/ to access my open-source APIs.

Chapter 2

Background

In this chapter, we first introduce the machine learning algorithms we focused on in more detail (Section 2.1). We then discuss the cryptographic methods leveraged in this thesis for secure function evaluation in the context of deep learning (Section 2.2).

2.1 Machine Learning

A machine learning model refers to a function f and its associated parameters θ that are particularly trained to infer/discover the relationship between input samples $x \in \{x_1, x_2, ..., x_N\}$ and the expected labels $y \in \{y_1, y_2, ..., y_N\}$. Each output observation y_i can be either continuous as in most regression tasks or discrete as in classification applications. Machine learning algorithms typically aim to find the optimal parameter set θ such that a loss function $\mathcal{L}$ that captures the difference between the output inference and ground-truth labeled data is minimized:

$$\theta = \underset{\theta}{argmin} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f(x_i, \theta), y_i). \tag{2.1}$$

In this thesis, we particularly focus our evaluations on the state-of-the-art deep learning models and casual Bayesian graphs due to their popularity in the realization of various autonomous learning

systems and time-series applications. Consistent with the literature in this field, we particularly centralize our discussions on the classification tasks. However, we emphasize that the core concept proposed in this thesis is rather more generic and can be used for reliable deployment of different learning techniques such as generalized linear models, regression methods (e.g., regularized regression (Lasso)), and kernel support vector machines.

2.1.1 Deep Learning

Deep learning is an important class of machine learning algorithms that has provided a paradigm shift in our ability to comprehend raw data by showing superb inference accuracy resembling the learning capability of human brain [LBH15, DY14]. A DL model is a hierarchical learning topology consisting of several processing layers stacked on top of one another. Table 2.1 summarizes common layers used in DL neural networks. The state of each neuron (unit) in a DL network is determined in response to the states of the units in the prior layer after applying a nonlinear activation function. In Table 2.1, $x_i^{(l)}$ is the state of unit i in layer l, $z_i^{(l)}$ is the post-nonlinearity value associated with unit i in layer l, $\theta_{ij}^{(l)}$ specifies the parameter connecting unit j in layer l and unit i in the layer $l+1$, and k indicates the kernel size used in 2-dimensional layers.

Training a DL network involves two main steps: (i) forward propagation, and (ii) backward propagation. These steps are iteratively performed for multiple rounds using different batches of known input/output pairs (x_i, y_i) to reach a certain level of accuracy. In the forward propagation, the raw values of data features are fed into the first layer of the network. These raw features are gradually mapped to higher-level abstractions based on the current state of the DL parameters (θ). The state of each neuron (unit) in a DL model is determined in response to the states of the units in the prior layer after applying a non-linear activation function. Commonly used activation functions for hidden layers include logistic sigmoid, Tangent-hyperbolic (Tanh), and Rectified Linear Unit (ReLu). The output layer is an exception for which a Softmax regression is typically used to determine the final inference. Softmax regression (or multinomial logistic regression) is a

Table 2.1: Commonly layers employed in deep neural networks.

DL Layer	Description	Computation
2D Convolution	Multiplying the filter weights ($\theta_{ij}^{(l-1)}$) with the post-nonlinearity values in the preceding layer ($z_{ij}^{(l-1)}$) and summing the results	$x_{ij}^{(l)} = \sum_{s_1=1}^{k}\sum_{s_2=1}^{k}\theta_{s_1 s_2}^{(l-1)} \times z_{(i+s_1)(j+s_2)}^{l-1}$
Fully-Connected	Multiplying the corresponding weights ($\theta_{ij}^{(l-1)}$) with the post-nonlinearity values in the preceding layer ($z_i^{(l-1)}$)	$x_i^{(l)} = \sum_{j=1}^{N_{l-1}}\theta_{ij}^{(l-1)} \times z_j^{(l-1)}$
Max Pooling	Computing the maximum value of $k \times k$ overlapping regions in the $N \times N$ grid of the underneath layer	$x_{ij}^{(l)} = Max(y_{(i+s_1)(j+s_2)}^{l-1})_{s_1 \in \{1,2,...,k\}}$ $s_2 \in \{1,2,...,k\}$
Mean Pooling	Computing the mean value of $k \times k$ non-overlapping regions in the $N \times N$ grid of the underneath layer	$x_{ij}^{(l)} = Mean(z_{(i+s_1)(j+s_2)}^{l-1})_{s_1 \in \{1,2,...,k\}}$ $s_2 \in \{1,2,...,k\}$
L_2 **Normalization**	Normalizing the L_2 norm of feature maps corresponding to each input sample	$x_i^{(l)} = \frac{x_i^{(l)}}{\sqrt{\sum_{j=1}^{N_l}\|x_j^{(l)}\|^2}}$
Batch Normalization	Normalizing feature maps per input batch by adjusting and scaling the activations	$x_i^{(l)} = \frac{x_i^{(l)} - \mu_B^{(l)}}{\sqrt{\frac{1}{b_s}\sum_{j=1}^{b_s}(x_j^{(l)} - \mu_B^{(l)})^2}}$
Non-linearity	Sigmoid	$z_i^{(l)} = \frac{1}{1+e^{-x_i^{(l)}}}$
	Softmax	$z_i^{(l)} = \frac{e^{x_i^{(l)}}}{\sum_{j=1}^{N_l} e^{x_j^{(l)}}}$
	Tangent Hyperbolic (Tanh)	$z_i^{(l)} = \frac{Sinh(x_i^{(l)})}{Cosh(x_i^{(l)})}$
	Rectified Linear Unit (ReLu)	$z_i^{(l)} = max(0, x_i^{(l)})$

generalization of logistic regression that maps a $\mathcal{P}$-dimensional vector of arbitrary real values to a $\mathcal{P}$-dimensional vector of real values in the range of [0, 1). The final inference for each input sample is determined by the output unit that has the largest conditional probability value [DY14].

In the backward propagation, a batch *gradient-based* algorithm (e.g., [Bot10]) is applied to *fine-tune* DL parameters such that a specified loss function is minimized. The loss function captures the difference between the neural network inference (output of forward propagation) and the ground-truth labeled data. In particular, DL parameters are updated per:

$$\theta_{ij}^{(l)} = \theta_{ij}^{(l)} - \eta \frac{1}{b_s} \sum_{k=1}^{b_s} \frac{\partial \mathcal{L}^{(l)}}{\partial \theta_{ij}^{(l)}}, \tag{2.2}$$

where η is the learning rate, b_s is the data batch size, and $(\partial \mathcal{L}^{(l)} / \partial \theta_{ij}^{(l)})$ represents the propagated loss function in the layer l. The forward and backward propagations are *iteratively* applied for multiple rounds of *reprocessing* the input data until the desired accuracy is achieved.

Once the DL network is trained to deliver a desired level of accuracy, the model is employed as a classification oracle in the execution phase (a.k.a., test phase). During the execution phase, the model parameters θ are fixed and prediction is performed through one round of forward propagation for each unknown input sample. Attacks based on adversarial samples target the DL execution phase and do not involve any tampering with the training procedure.

2.1.2 Causal Bayesian Graphical Analysis

Decomposition of time-series data into estimated latent variables provides an important alternative view from the time domain perspective [SMH07, BR10]. Let us denote the input data samples **D** as the pair of (**x**, **y**) values, where $\mathbf{x} = \{x_i = [x_{i1}, ..., x_{id}]\}_{i=1}^{N}$ includes the input data features and $\mathbf{y} = [y_1, ..., y_N]$ are the observation values. Here, d is the feature space size and N specifies the number of data measurements that may grow over time. Each output observation y_i can be either continuous as in most regression tasks, or discrete as in classification applications. The key to performing Bayesian graph analytics is to find a *probabilistic likelihood function* that maps each input feature x_i to its corresponding observation y_i such that:

$$y_i = f(x_i) + \varepsilon_i. \tag{2.3}$$

The variable ε_i is an additive observation noise that determines how different the observation vector y_i can be from the latent function value $f(x_i)$. The observation noise is usually modeled as a Gaussian distribution variable with zero mean and a variance of σ_n^2.

In probabilistic graphical models, all parameters should be represented as random variables. *Gaussian processes* are commonly used as the prior density over the set of latent functions $\{f(x_i)\}_{i=1}^{N}$ for analyzing time-series data. In Gaussian processes, each data point x_i is associated with a Normally distributed random variable f_i. Every finite collection of those random variables

has a multivariate Gaussian distribution. GP is represented as:

$$\mathbf{f}(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), K(\mathbf{x}, \mathbf{x}')), \tag{2.4}$$

where $m(\mathbf{x})$ and $K(\mathbf{x}, \mathbf{x}')$ are the mean and covariance kernels that capture the correlation between data samples. With a GP prior, the observations $\mathbf{y} = [y_1, ..., y_N]$ can be assumed to be conditionally independent given the latent function **f(.)**. Therefore, the likelihood $p(\mathbf{y}|\mathbf{f})$ can be factorized over data samples as $\prod_{i=1}^{N} p(y_i|f_i)$, where $\mathbf{f} = [f(x_1), ..., f(x_N)]$. Note that the observations themselves are not independent (e.g., $p(\mathbf{y}) \neq \prod_{i=1}^{N} p(y_i)$). The mean and covariance kernel of a GP are also random variables with certain hyper-parameters (γ) that should be tuned with respect to the input data. The choice of the mean and covariance kernels determines the smoothness and variability of the latent function **f(.)** to be estimated.

To make our notation explicit, we write the likelihood as $p(\mathbf{y}|\mathbf{f}, \sigma_n^2)$ where σ_n^2 is the parameter of the observation noise, and $p(\mathbf{f}|\gamma)$ is the GP prior. The quantities $\theta = [\gamma, \sigma_n^2]$ are the hyper-parameters of the underlying probabilistic model. The posterior distribution $p(\theta|\mathbf{D})$ must be computed to make predictions for the incoming data samples in different learning tasks including various regression and classification techniques, stochastic optimizations, and neural networks. Let us denote the function of interest to be evaluated with $g(\theta)$. Thereby, the underlying learning task can be expressed as the evaluation of the following integral:

$$E_{p(\theta|\mathbf{D})}[g(\theta)] = \int g(\theta) p(\theta|\mathbf{D}) d\theta. \tag{2.5}$$

For instance, by setting $g(\theta) = p(y^*|\theta)$, one can predict the probability of a future observation y^* based on the previously observed data per $p(y^*|\mathbf{D}) = \int p(\mathbf{y}^*|\theta) p(\theta|\mathbf{D}) d\theta$.

Given the large cardinality of the hyper-parameter set $|\theta|$, and the high dimensionality of input data in real-world applications, it is computationally impractical to analytically evaluate the integral in Eq. (2.5). Thus, estimation algorithms such as MCMC are often the methods of

Table 2.2: Markov Chain Monte Carlo (MCMC) methodologies commonly used for analyzing graphical Bayesian networks.

MCMC Methods	Description
Population-based	Population-based MCMC is a method designed to address the issue of multi-modality using a population of Markov chains. This method is particularly inefficient for analyzing high-dimensional data, due to the high cost of unnecessary space exploration.
State Space Model	State Space Model (SSM) MCMC targets Bayesian applications in which evaluating the closed-form PDF is not feasible. SSMs assumes the availability of unbiased estimators to compute the acceptance ratio in each MCMC step. This assumption does not often hold in practice.
Gibbs Sampling	Gibbs sampling decomposes the proposal distribution into its individual components by computing the full conditional distribution of the variable θ_i conditional on all the remaining ones. Gibbs sampling encounters serious computational inefficiency in solving high-dimensional tasks with highly correlated variables.
Slice Sampling	Slice sampling method uniformly samples from the area under the $p(\theta)$ graph as an equivalent to sampling from the probability distribution. This technique improves mixing performance in learning tasks with highly correlated variables. The complexity of Slice sampling scales exponentially with the data dimensionality.
Hamiltonian	Hamiltonian MCMC method uses the gradient of the target probability distribution to select better movements in each iteration. This method is particularly of interest as it can handle both *strong correlations* and *high-dimensionality* of the probability distribution.
Adaptive	Adaptive MCMC method adjusts the proposal distribution in the execution time to achieve a better sampling efficiency. The adaptive kernel might converge to a non-stationary distribution if not designed carefully.

choice [ADFDJ03]. Table 2.2 summarizes different MCMC algorithms. MCMC methods work sequentially by constructing a Markov chain with each state of the chain corresponding to a new random sample from the posterior distribution $p(\theta|\mathbf{D})$. The selected samples are then leveraged to approximate the answer to Eq. (2.5) as the following:

$$\tilde{E}_{p(\theta|\mathbf{D})}[g(\theta)] = \frac{1}{N}\Sigma_{i=1}^{N} g(\theta^{(i)}). \tag{2.6}$$

2.2 Secure Function Evaluation

Here, we provide a brief description of the cryptographic protocols that we used in this thesis for DL execution on encrypted data.

2.2.1 Oblivious Transfer

Oblivious Transfer (OT) is a cryptographic protocol that runs between a Sender (S) and a Receiver (R). The receiver R obliviously selects one of the potentially many pieces of information provided by S. Particularly, in a 1-out-of-n OT protocol, the sender S has n messages $(x_1, ..., x_n)$

and the receiver R has an index r where $1 \leq r \leq n$. In this setting, R wants to receive x_r among the sender's messages without the sender learning the index r, while the receiver only obtains one of the n possible messages [NP05].

2.2.2 Garbled Circuit

Yao's garbled circuit protocol [Yao86] is a cryptographic protocol in which two parties, Alice and Bob, jointly compute a function $f(x,y)$ on their inputs while keeping the inputs fully private. In GC, the function f should be represented as a Boolean circuit with 2-input gates (e.g., XOR, AND, etc.). The input from each party is represented as input wires to the circuit. All gates in the circuit have to be topologically sorted which creates a list of gates called *netlist*. GC has four different stages: (i) Garbling which is only performed by Alice (a.k.a., Garbler). (ii) Data transfer and OT which involves both parties, Alice and Bob. (iii) Evaluating, only performed by Bob (a.k.a., Evaluator). (iv) Merging the results of the first two steps by either of the parties.

(i) Garbling. Alice garbles the circuit by assigning two random k-bit[1] labels to each wire in the circuit corresponding to semantic values one and zero. For instance, for input wire number 5, Alice creates 128-bit random string l_5^0 as a label for semantic value zero and l_5^1 for semantic value one. For each gate, a garbled table is computed. The very first realization of the garbled table required four different rows, each corresponding to one of the four possible combinations of inputs labels. Each row is the encryption of the correct output key using two input labels as the encryption key [BHKR13]. As an example, assume wire 5 (w_5) and 6 (w_6) are input to an XOR gate and the output is wire 7 (w_7). Then, the second row of the garbled table which corresponds to ($w_5 = 0$) and ($w_6 = 1$) is equivalent to $Enc_{(l_5^0, l_6^1)}(l_7^1)$. To decrypt any garbled table, one needs to possess the associated two input labels. Once Garbler creates all garbled tables, the protocol is ready to move forward to the second step.

(ii) Transferring Data and OT. In this step, Alice sends all the garbled tables along with

[1] k is a security parameter, its value is chosen as 128 in recent works

the correct labels corresponding to her actual input to Bob. For instance, if the input wire 8 belongs to her and her actual input for that wire is zero, she sends l_8^0 to Bob. In order for Bob to be able to decrypt and evaluate the garbled tables (step 3), he needs the correct labels for his input wires as well. This task is not trivial nor easy. On the one hand, Bob cannot send his actual input to Alice to avoid undermining his input privacy. On the other hand, Alice cannot simply send both input labels to Bob since Bob can then infer more information in step 3. To effectively perform this task, OT protocol is utilized. For each input wire that belongs to Bob, both parties engage in a 1-out-of-2 OT protocol where the selection bit is Bob's input and two messages are two labels from Alice. After all required information is received, Bob can start evaluating GC.

(iii) Evaluating. To evaluate the garbled circuit, Bob starts from the first garbled table and uses two input labels to decrypt the correct output key. All gates and their associated dependencies are topologically sorted in the netlist. As such, Bob can perform the evaluation one gate at a time until reaching the output wires without any halts in the process. In order to create the actual plain-text output, both the output mapping (owned by Alice) and final output labels (owned by Bob) are required; thereby, one of the parties needs to send his share to the other party.

(iv) Merging Results. At this point, Alice can compute the final results. To do so, she uses the mapping from output labels to the semantic value for each output wire. The protocol can be considered finished after merging the results or Alice can also share the final results with Bob.

2.2.3 Garbled Circuit Optimizations

During the past decade, several optimization methodologies have been suggested in the literature to minimize the overhead of executing GC protocol. In the following, we summarize the most important cryptographic optimizations techniques that we also leverage in this thesis.

Point and Permute. According to this optimization [BMR90], the label of each wire is appended by a select bit, such that the select bits for the two labels of the same wire are inverse of each other. Even though the select bits are public, the association between select bits and

semantic value of the wire is random and private to the garbler.Besides allowing the use of more efficient encryption, it also makes the evaluation simpler since the evaluator can simply decrypt the appropriate row based on the public select bits of the wire labels.

Row-Reduction. The initial garbled table consists of four rows. Authors in [NPS99] proposed a technique to reduce the number of rows in the garbled table to three. Instead of randomly generating the label for the output wire of a gate, it is computed as a function of the labels of the inputs such that the first row of the garbled table becomes all 0 and no longer needs to be sent to evaluator. This technique results in 25% reduction in communication.

Free-XOR. Perhaps one of the most important optimizations of GC is Free-XOR [KS08]. The Free-XOR methodology enables the evaluation of XOR, XNOR, and NOT gates without costly cryptographic encryption. Therefore, it is highly desirable to minimize the number of non-XOR gates in the deployment of the underlying circuit. In this method, Alice generates a random $(k-1)$-bit value R which is known only to her. For each wire c, she generates the label X_c^0 and sets $X_c^1 = X_c^0 \oplus (R \parallel 1)$[2]. With this convention, the label for the output wire r of an XOR gates with input wires p, q can be simply computed as $X_r = X_p \oplus X_q$. Note that R is appended by 1 to be compatible with the Point and Permute optimization.

Half-Gates. This technique that is proposed in [ZRE15] further reduces the number of rows for AND gates from three to two, resulting in 33% less communication on top of the Row-reduction optimization.

Garbling with Fixed-Key Block Cipher. This methodology [BHKR13] introduces an encryption mechanism for garbled tables based on fixed-key block ciphers (e.g., AES). Many of the modern processors have AES-specific instructions in their instruction set architecture which, in turn, makes the garbling and evaluating process significantly faster using the fixed-key cipher.

Sequential Garbled Circuit. For years the GC protocol could have only been used for Combinational circuits (a.k.a., acyclic graphs of gates). Authors in [SHS+15] suggested a new

[2] $\parallel$ denotes the concatenation operator.

approach that enables garbling/evaluating sequential circuits (cyclic graphs). In their framework, one can garble/evaluate a sequential circuit iteratively for multiple clock cycles.

2.3 Acknowledgements

This chapter, in part, has been published at (i) the Proceedings of 2017 International Design Automation Conference (DAC) and appeared as: Bita Darvish Rouhani, Azalia Mirhoseini, and Farinaz Koushanfar "Deep3: Leveraging Three Levels of Parallelism for Efficient Deep Learning", (ii) ACM Transactions on Reconfigurable Technology and Systems (TRETS) 2018 as: Bita darvish Rouhani, Siam U Hussain, Kristin Lauter, and Farinaz Koushanfar "ReDCrypt: Real-Time Privacy-Preserving Deep Learning Inference in Clouds Using FPGAs", and (iii) the Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA) and appeared as: Bita Darvish Rouhani, Mohammad Ghasemzadeh, and Farinaz Koushanfar "CausaLearn: Automated Framework for Scalable Streaming-based Causal Bayesian Learning using FPGAs". The dissertation author was the primary author of this material.

Chapter 3

Deep3: Leveraging Three Levels of Parallelism for Efficient Deep Learning

This chapter introduces Deep3, an automated platform-aware Deep Learning (DL) framework that brings orders of magnitude performance improvement to DL training and execution. Deep3 is the first to *simultaneously* leverage three levels of parallelism for performing DL: *data, network, and hardware*. It uses platform profiling to abstract physical characterizations of the target platform. The core of Deep3 is a new extensible methodology that enables incorporation of platform characteristics into the higher-level data and neural network transformation. We provide accompanying libraries to ensure automated customization and adaptation to different datasets and platforms. Proof-of-concept evaluations demonstrate 10-100 fold physical performance improvement compared to the state-of-the-art DL frameworks, e.g., TensorFlow.

3.1 Introduction

Deep learning has become the key methodology for achieving the state-of-the-art inference performance by moving beyond traditional linear or polynomial analytical machine

learning [DY14]. Despite DL's powerful learning capability, the computational overhead associated with DL methods has hindered their applicability to resource constrained settings. Examples of such settings range from embedded devices to data centers where physical viability (e.g., runtime and energy efficiency) is a standing challenge.

To optimize DL physical performance, there are at least two sets of challenges that should be addressed simultaneously. The first challenge set has to do with the costly iterative nature of DL training process [DY14]. What signifies the relevance of this cost is the empirical context of DL algorithms: even though DL neural networks have shown superior results compared to their linear machine learning counterparts, their success has been mainly based on experimental evaluations with the theoretical aspects yet to be developed. As such, reducing the DL performance cost, particularly training runtime, enables realization of different DL models within the confine of the available computational resources to empirically identify the best one.

The second challenge set is related to the mapping of computations to increasingly multi-core and/or heterogeneous modern architectures. The cost of computing and moving data to/from the memory and inter-cores is different for each computing platform. The existing solutions for performing DL are divided into two distinct categories: (i) Data scientists, on the one hand, have been mainly focused on optimizing DL efficiency through data and neural network manipulations and pruning with little or no attention to the platform characterization [DCM+12, CHW+13, CSAK14]. (ii) Computer engineers, on the other hand, have developed hardware-accelerated solutions for an efficient domain-customized realization of DL models, e.g., [ZLS+15, JGD+14]. Although these works demonstrate significant improvement in the realization of particular DL models, to the best of our knowledge, none of the prior works have looked into the end-to-end solution on how the higher level data-dependent signal transformation can benefit the physical performance. Our hypothesis is that devising platform aware signal transformations could highly favor the underlying learning task by holistic customization to the limits of the hardware resources, data, and DL neural network.

We propose Deep3, the first automated end-to-end DL framework that explicitly optimizes the physical performance cost by higher-level DL transformation. Deep3 provides performance metrics to quantify DL overhead in terms of (i) the number of arithmetic operations dedicated to training/executing DL networks, and (ii) the number of (inter-core and inter-memory) communicated words. Common performance indicators such as runtime, energy, and memory footprint can be directly deduced from these metrics. To optimize for these costs, we introduce a systematic approach for automated platform customization as well as data projection error tuning while achieving the target accuracy.

Deep3 optimization is devised based on *three inter-linked* thrusts. First, it identifies the intrinsic platform characteristics by running a set of subroutines to find the optimal local neural network size that fits the resource constraints *(Hardware Parallelism)*. Second, it proposes a novel DL graph traversal methodology that leverages the fine- and coarse-grained parallelism in the data, neural network, and physical platform for efficient hardware implementation. Deep3 transforms the global DL training task into the parallel training of multiple smaller size local DL networks while minimally affecting the accuracy. Our approach balances the trade-off between memory communication and local calculations to improve the performance of costly iterative DL computations *(Network Parallelism)*. Third, it leverages a new data and resource aware signal projection as a pre-processing step to reduce the input data feature space size customized to the local neural network topology dictated by the platform *(Data Parallelism)*. Deep3 exploits the degree of freedom in producing several possible projection embeddings to automatically adapt the data and DL circuitry to the physical limitations imposed by the platform. The explicit contributions of this work are as follows:

- Proposing Deep3, the first end-to-end DL framework which is simultaneously data, neural network, and hardware parallel. Deep3 achieves significant performance cost reduction by platform aware signal transformation while delivering the same inference accuracy compared to the state-of-the-art DL approaches.

- Incepting a novel neural network graph traversal methodology for efficient mapping of computations to the limits of the target platform. Our approach is scalable while it greatly reduces costly memory interactions in performing DL training and execution.
- Developing performance cost metrics and automated platform aware signal transformations to optimize the data projection and DL neural network architecture for the underlying resources and physical constraints.
- Devising accompanying libraries to ensure Deep3 ease of adoption on three common classes of platforms including CPUs, GPUs, and FPGAs. Note that our proposed methodology is universal and directly applicable to other families of computing hardware.
- Creating customized approaches to provide support for *streaming scenarios* where data dynamically evolves over time. Proof-of-concept evaluations for both static and dynamic data corroborate 10-100 fold performance improvement compared to the prior art solutions.

3.2 Deep3 Global Flow

Figure 3.1 illustrates the high-level block diagram of Deep3 for training DL networks. Deep3 takes the stream of data samples and the *global DL topology (S_G)* as its input. S_G is a user-defined vector of integers whose components indicate the number of neurons per hidden layer of the global DL model. Deep3 consists of four main modules to learn the global parameters of a DL network customized to the limits of the physical resources and constraints:

(i) Physical profiling: Deep3 characterizes the target platform by running subroutines that contain basic operations involved in the forward and backward propagations (Section 3.3). The characterization is a one-time process and incurs a fixed negligible overhead.

(ii) Parameter coordination: Deep3 uses the output of physical profiling as guidelines to map the global DL training into smaller size local neural networks that fit the platform constraints.

The local networks are formed by subsampling the neurons of the global DL model based on an *extensible depth-first* graph traversal methodology that we introduce to minimize the communication overhead per training iteration (Section 3.4). Each local network is then replicated into multiple computing kernels that are executed in parallel using different data batches.

(iii) Platform aware data transformation: Deep3 leverages a new resource aware signal projection as a pre-processing step. It transforms the stream of input data into multiple lower-dimensional subspaces that fit the local network topology dictated by the platform. Deep3 uses the degree of freedom in producing several possible projection embeddings in order to customize costly DL training/execution to the limits of the platform and data structure (Section 3.5).

(iv) Local network updating kernels: Each local neural network updates a fraction of the global model using different data batches. Each fraction has the same depth as the global network with far fewer edges. The local updates are periodically aggregated through the parameter coordination unit to optimize the weights of the global model (Section 3.4).

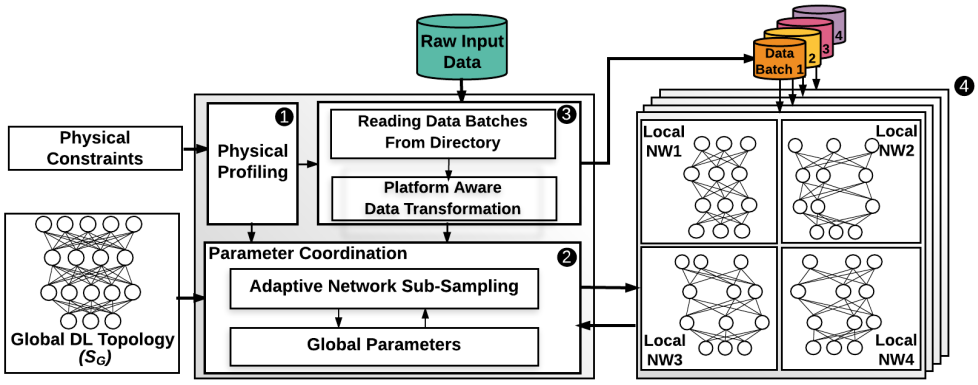


Figure 3.1: Global flow of Deep3 framework. Deep3 leverages the data, neural network, and hardware fine- and coarse-grained parallelism to adaptively customize DL in accordance to the physical resources and constraints.

We provide accompanying libraries to ensure Deep3 ease of use for data scientists and engineers. Our libraries provide support for training/execution of DL models on multi- and many-core CPU, CPU-GPU, and CPU-FPGA platforms.

3.3 Hardware Parallelism

The structure of a local network (i.e., the number of neurons per layer) has a direct impact on the corresponding memory footprint and overall system performance. Deep3 provides a set of subroutines that characterize the impact of neural network size on the subsequent resource consumption. Our subroutines measure the performance of basic operations involved in the DL training/execution. Examples of such operations include convolution, matrix multiplication, non-linearities, and inter-core communication. Note that the realization of DL operations can be highly diverse depending on the target platform. For instance, based on the dimensionality of the matrices being multiplied, a matrix multiplication can be compute-bound, bandwidth-bound, or occupancy-bound on a specific platform. Our subroutines run such operations with varying sizes to spot the target platform constraints.

Deep3 takes a user-defined performance metric such as runtime, energy, energy$\times$delay, and/or memory as its input. To optimize for the target performance metric, Deep3 uses the output of physical profiling as guidelines to break down the global DL model into subsequent local neural networks that fit the pertinent resource provisioning (Section 3.4). The physical profiling unit outputs a vector of integers, S_L, that has the same length as the input global model S_G. Each element of S_L indicates the maximum size of the corresponding layer that fits into the platform. Platform characterization is a one-time process and incurs a negligible overhead compared to DL training. We use vendor supplied libraries to model the target platform as suggested in [Bai16].

3.4 Neural Network Parallelism

Traditionally, a *breadth-first* (a.k.a., stripe-based) model partitioning is used to distribute DL training workload, e.g., TensorFlow. Figure 3.2a illustrates the conventional approach used for neural network parallelism. In this setting, the activations of neural connections that cross the machine boundaries should be exchanged between different nodes to complete one round

of forward and backward propagation. What exacerbates the DL training cost in this context is the variance in the processing time of different nodes. To mitigate the effect of this variation, authors in [DCM$^+$12, CSAK14, AAB$^+$15] suggest the use of asynchronous partial gradient updates for DL training. These works achieve meaningful runtime reduction by parallelism in the training of DL networks with sparse connectivity. However, for fully-connected graphs fine-tuning the parameters is still a bottleneck given the greedy layer-wise nature of DL and the higher communication overhead of fully-connected neural networks.

Deep3, for the first time, proposes a *depth-first graph traversal* methodology to distribute DL training workload customized to the platform constraints. As shown in Figure 3.2b, Network parallelism in Deep3 framework. transforms the global DL model into multiple overlapping local neural networks that are formed by subsampling the neurons of the global network. Each local neural network has the same depth as the global model with far fewer edges. Our approach minimizes the communication overhead per training iteration. This is because each local network can be updated independently without having to wait for partial gradient updates from successive layers to be communicated between different computing nodes. The local updates are periodically aggregated through the parameter coordinator to optimize the weights of the global model. The number of neurons per local network, S_L, is a design configuration dictated by the physical resources (i.e., memory bandwidth, cache size, or available energy).

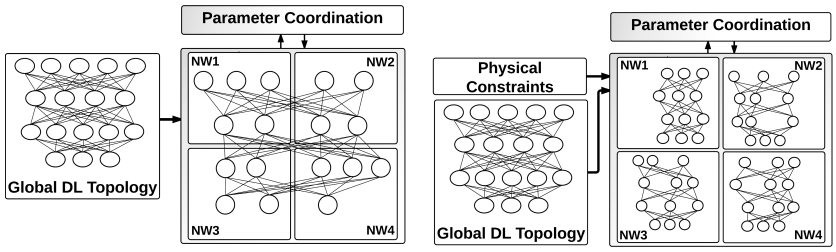


Figure 3.2: Network parallelism in Deep3 framework. The existing breadth-first DL partitioning approaches require communicating the partial updates multiple times for processing each batch of data. On the contrary, our depth-first approach limits such communications to only once after multiple training iterations.

3.4.1 Parameter Coordination

Algorithm 1 outlines the pseudocode of Deep3 parameter coordination unit and local DL computing kernels. Let us denote the parameter coordination unit with $P_{id} = 0$. For each available computing kernel (e.g., an FPGA, GPU, or CPU core), the parameter coordinator initiates a pair of *send- and receive-thread*. The send-thread subsamples the neurons of the global DL model in accordance with the local DL topology, S_L, dictated by the platform (Section 3.3). It communicates the selected subsample of the global model to its associated computing kernel for further processing. Each send-thread keeps track of the selected subsample of DL neurons indices ($index_L$) sent to its associated computing kernel in a list called *history*. Storing the indices is required to later aggregate the updated parameters in the global model.

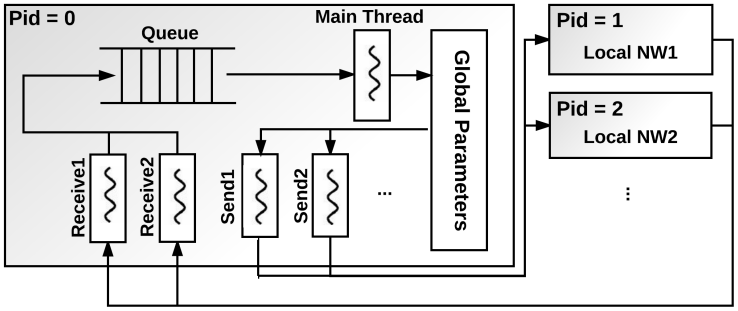


Figure 3.3: Flow of data in Deep3 framework.

The receive-thread reads back the computed gradients from each local computing kernel. It enqueues the local gradients (ΔW_L) along with the *threadID* and the number of communications occurred between the coordinator and that specific kernel so far. This information is then used by the main thread in the coordination unit to retrieve the corresponding global indices from the history list and aggregate the partial local gradients into the global DL model. The send and receive threads keep working until they got interrupted by the main thread once the specified error threshold is met or a certain number of DL iterations have been performed. Figure 3.3 illustrates the flow of data in Algorithm 1.

Algorithm 1 Deep3 Neural Network Parallelism

INPUT: **Global Model (S_G), Error Threshold δ, Local Model (S_L), Training Data Embedding (C_{Tr}), Number of Processors N_p, and Maximum Number of Iterations Max_itr.**

OUTPUT: **Trained Global DL Parameters DL_{glob}.**

0. ***Send_Thread*** $(threadID,\ S_G,\ S_L,\ history)$:
1: $send\_count = 0$
2: $While(!done\_flag)$:
3: $Index_L \leftarrow NetworkSubsampling(S_L)$
4: $DL^{init}_{local} \leftarrow S_G.get\_weights(Index_L)$
5: $comm.send(DL^{init}_{local}, dest = threadID)$
6: $history[threadID].append(Index_L)$
7: $send\_count \leftarrow send\_count + 1$

1. ***Recieve_Thread*** $(threadID,\ Q,\ Q\_Lock)$:
8: $Rcounter = 0$
9: $While(!done\_flag)$:
10: $\Delta W_L = comm.recv(source = threadID)$
11: $lock(Q\_Lock)$
12: $Q.put([\Delta W_L, threadID, Rcounter])$
13: $release(Q\_Lock)$
14: $Rcounter \leftarrow Rcounter + 1$

if $(P_{id} == 0)$: **//Parameter Coordinator**
15: $Q = Queue()$
16: $queue\_Lock = threading.Lock()$
17: $DL_{glob} \leftarrow RandomInitialization()$
18: $itr = 0$
19: $history = []$
20: $done\_flag \leftarrow False$
21: $Creation\ \&\ Initialization\ of\ Send\_Threads\ \&\ Receive\_Threads$
22: $While(\tilde{\delta} \geq \delta\ or\ itr \leq Max\_itr)$: **//Main Thread in Parameter Coordinator**
23: $[\Delta W_L, threadID, Rcounter] \leftarrow Q.get()$
24: $Index_L \leftarrow history[threadID][Rcounter]$
25: $DL_{glob} \leftarrow S_G.get\_weights(S_G)$
26: $S_G.set\_weights(DL_{glob} + \Delta W_L, Index_L)$
27: $\tilde{\delta} \leftarrow UpdateValidationError(S_G)$
28: $itr \leftarrow itr\ +\ 1$
29: $done\_flag \leftarrow True$
30: $DL_{glob} \leftarrow S_G.set\_weights(S_G)$
31: $Broadcasts\ done\_flag\ \&\ Exit$
32: **else: //Local Network Kernels**
33: $While(!done\_flag)$:
34: $DL^{init}_{local} = comm.recv(source = 0)$
35: $S_L.set\_weights(DL^{init}_{local})$
36: $DL_{local} \leftarrow DL^{init}_{local}$
37: $i \leftarrow 0$
38: $While\ (i \leq n_{push})$:
39: $C^i_{Tr} \leftarrow GetNextDataBatch()$
40: $DL_{local} \leftarrow UpdateDL(DL_{local}, C^i_{Tr})$
41: $i \leftarrow i + 1$
42: $\Delta W_L \leftarrow DL_{local} - DL^{init}_{local}$
43: $comm.send(\Delta W_L, dest = 0)$

In Deep3, each local network is independently trained using different data batches generated by the data transformation unit (Section 3.5). A local network might compute its gradients (updates) based on a set of parameters that are slightly out of date. This is because the other local networks have probably updated the global values in the parameter coordination unit in the meantime. The impact of using stale parameters eliminates over time. The reason behind this is that the process of updating DL weights is *associative* and *commutative*, and after several iterations, the DL parameters converge to the desired accuracy.

3.4.2 Computation-Communication Trade-off

Deep3 characterizes time per training iteration for updating the global neural network as:

$$T_{itr} = \underbrace{n_{push} \sum_{i=1}^{N_p} (T_i^{FP} + T_i^{BP})}_{\text{Computation Cost}} + \underbrace{\frac{2}{n_{push}} \sum_{i=1}^{N_p} T_i^{comm}}_{\text{Communication Cost}}, \tag{3.1}$$

where the first term represents the computation cost and the latter term characterizes the inter-core communication overhead. We use T_i^{FP} and T_i^{BP} to denote the forward and backward propagation costs and N_p to represent the number of concurrent processors. The variable n_{push} indicates the frequency of model aggregation (i.e., the number of training data batches that should be processed before a local network pushes back its updates to the coordination unit). Table 3.1 details the computation and communication cost of training fully-connected Deep Neural Networks (DNNs). Similar setup applies to the Convolutional Neural Networks (CNNs) in which a set of convolutions are performed per layer. Deep3 finds an estimation of the physical coefficients listed in Table 3.1 by running a set of subroutines as discussed in Section 3.3. Our libraries provide support for both fully-connected and Convolutional neural networks.

The communication overhead in Deep3 is dominated by the cost of reading/writing the weights of local neural networks back and forth between the parameter coordinator and the

Table 3.1: Local Computation and Communication Costs.

Computation and Communication Costs
$T_i^{FP} = \alpha_{flop} \sum_{s=1}^{S-1} n_i^{(s)} n_i^{(s+1)} + \alpha_{act} \sum_{s=1}^{S} n_i^{(s)}$ S: *total number of DNN layers* α_{flop}: *multipl_add + memory communication cost* α_{act}: *activation function cost*
$T_i^{BP} = 2\alpha_{flop} \sum_{s=1}^{S-1} n_i^{(s)} n_i^{(s+1)} + \alpha_{err} \sum_{s=1}^{S} n_i^{(s)}$ α_{err}: *propagation error cost*
$T_i^{Comm} = \alpha_{net} + \frac{N_{bits} \sum_{s=1}^{S-1} n_i^{(s)} n_i^{(s+1)}}{BW_i}$ α_{net}: *constant network latency* N_{bits}: *number of signal representation bits* BW_i: *operational communication bandwidth*

computing kernels. On the one hand, a high value of n_{push} reduces the communication cost, but it also increases the computational load as the local networks are not frequently combined to optimize the global DL model. On the other hand, a low value of n_{push} degrades the training performance due to a significant increase in the communication overhead. Deep3 tunes the variable n_{push} accordingly to fit the physical limitations while balancing the computation overhead versus communication.

3.5 Data Parallelism

The input layer size of a neural network is conventionally dictated by the feature space size of the input data samples used for DL training. Deep3's data transformation module takes the local DL topology (S_L) imposed by the platform as its input and aims to find the lower-dimensional data embedding that best matches the dictated local model. It works by factorizing the raw input data $A_{m \times n}$ into a *dictionary matrix* $D_{m \times l}$ and a *data embedding* $C_{l \times n}$ such that:

$$\underset{l, D_{m \times l}, C_{l \times n}}{minimize} (\delta_{valid}^{local}) \quad s.t. \quad \|A - DC\|_F \leq \varepsilon \|A\|_F, \; l \leq m, \tag{3.2}$$

where δ_{valid}^{local} is the partial validation error acquired by training the local neural networks using the projected data embedding C instead of the raw data A. $\|\cdot\|_F$ denotes the Frobenius norm and ε is an intermediate approximation error that casts the rank of the input data.

Eq. 3.2 is a part of an overall objective that we aim to optimize in order to train/execute a DL model within the given computational resources. To solve Eq. 3.2, we first initiate the matrices D and C as empty sets. Deep3 gradually updates the corresponding data embeddings by streaming the input data as outlined in Figure 3.4. In particular, for a batch of newly arriving samples (A_i), Deep3 first calculates a projection error, $V(A_i)$, based on the current values of the dictionary matrix D. This error shows how well the newly added samples can be represented in the space spanned by D. If the projection error is less than a user-defined threshold (β), it means the current dictionary D is good enough to represent those new samples (A_i). Otherwise, Deep3 modifies the corresponding data embeddings to include the new data structure imposed by the recently added samples. Our data projection approach is linear in complexity and incurs a negligible overhead as we experimentally verify in Section 3.6. Note that after adding enough samples to the dictionary D, little improvement is observed in the DL accuracy as a result of increasing the size of the DL input layer. While checking the projection error $V(A_i)$ ensures that Deep3 doesn't add linearly-correlated samples to the dictionary, getting feedback from the DL model prevents unnecessary increase in the size of the neural network's input layer.

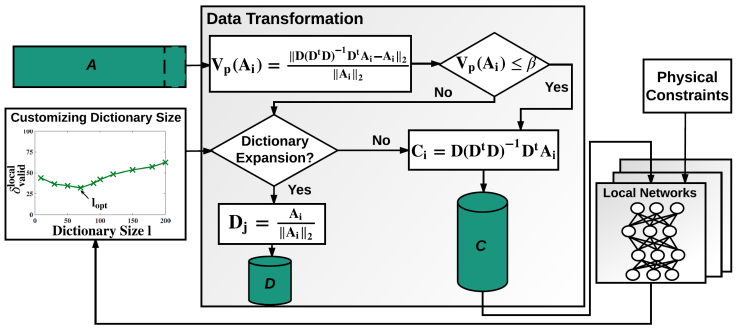


Figure 3.4: Data Parallelism in Deep3 framework. Deep3 maps the stream of input data to a ensemble of lower-dimensional embeddings. The data embedding is used to update the local neural network.

Many complex modern datasets that are not inherently low-rank can be modeled by a composition of multiple lower-rank subspaces [DSB13, MRSK16, RMK16]. Methods such as Principal Component Analysis (PCA) are oblivious to the coarse-grained parallelism existing in the data; they always return a unified subspace of data equal to the rank of the matrix. Deep3 leverages a new composition of high-dimensional dense data as an ensemble of multiple lower-dimensional subspaces by carefully selecting dictionary samples from the data itself rather than using principal vectors. This type of composition has been suggested in the literature to facilitate knowledge extraction [DSB13] or improve the system's physical performance [MRSK16, RMK16]. However, no earlier work has adapted the alignment of data lower-dimensional subspaces as a way to facilitate global training of large-scale DL models customized to the local DL circuitry that best matches the hardware. In addition, traditional projection methods such as PCA incur a quadratic complexity, which makes it a costly choice for projecting large datasets.

3.6 Experiments

We provide accompanying libraries for our realization of Deep3 framework. Our implementations for multi-core CPU and CPU-GPU platforms are built to work with highly popular DL libraries, e.g., TensorFlow [AAB$^+$15] and Theano [BBB$^+$10]. In our FPGA realization, we use 16 bits fixed-point for DL computations. Each variable is represented in two's complement format using 1 sign bit, 3 integer bits, and 12 fraction bits. We choose the fixed-point format for our FPGA implementation for two main reasons: (i) The range of parameters within a DL model is bounded due to applying non-linear activation functions such as Tanh or Sigmoid. Thereby, the use of fixed-point format does not significantly degrade the computational accuracy as shown in [SMA07]; (ii) Fixed-point implementation incurs a smaller area overhead and is significantly faster compared to its floating-point counterpart. We use floating-point format for our multi-core CPU and GPU realizations.

One common approach in training a DL model is the use of Stochastic Gradient Descent (SGD) to fine-tune the neural network parameters. In our realization of SGD, instead of using a fixed learning rate η in computing the local gradients, we used Adagrad technique [DHS11] to adapt the learning rate of each parameter according to the update history of that parameter. In particular, Deep3 computes the learning rate of the i^{th} parameter at iteration K as $\eta_{iK} = \frac{\gamma}{\sum_{j=1}^{K} \Delta W_{ij}^2}$, where ΔW_{ij} is the gradient of parameter i at iteration j.

3.6.1 Deep3 Performance Evaluation

Platform Settings. We evaluate Deep3 on three platforms: (i) **Platform 1** is a CPU-GPU co-processor with 192 CUDA cores and 4-Plus-1 quad-core ARM Cortex A15 CPU [Jet15]. (ii) **Platform 2** is a CPU-FPGA setting in which a Xilinx Virtex-6 FPGA is hosted by an Intel core i7 processor. We use a 1Gbps Ethernet port to send data back and forth between the FPGA and the host. (iii) **Platform 3** is a multi-core CPU with an Intel core i7-2600K processor.

We based our evaluations on perceiving knowledge from (i) visual, (ii) smart-sensing, and (iii) audio data. Table 3.2 shows Deep3 total pre-processing time overhead. The pre-processing overhead can be broken down into the overhead of tuning the algorithmic parameters and data projection. The tuning itself accounts for both platform profiling and setting dictionary size l in accordance to the local neural network size dictated by the platform. We use a small subset of data (e.g., less than 5%) for tuning purposes. After fine-tuning the parameter l, the projection is performed using the customized value of l. In our implementation, all the CPU cores within each specified platform are employed for data projection using Message Passing Interface (MPI).

Table 3.2: Deep3 pre-processing overhead.

Application	Visual[Hyp15] (200×54129)			Smart-Sensing[HAR15] (5625×9120)			Audio[mlr17a] (617×7797)		
Platform ID	1	2	3	1	2	3	1	2	3
Tuning Overhead	49.7s	32.4s	63.5s	91.4s	53.8s	102.6s	31.1s	18.3s	37.5s
Data Projection Overhead	17.1s	9.8s	9.7s	18.9s	10.1s	10.1s	8.3s	4.9s	4.7s
Overal	66.8s	42.2s	73.2s	110.3s	63.9s	112.7s	39.4s	23.2s	42.2s

Table 3.3: Performance improvement achieved by Deep3 over prior-art deep learning approach.

Application	Visual ($200 \times 1000 \times 300 \times 9$)			Smart-Sensing ($5625 \times 3000 \times 500 \times 19$)			Audio ($617 \times 200 \times 26$)		
	CPU/GPU (Platform 1)	CPU/FPGA (Platform 2)	CPU-only (Platform 3)	CPU/GPU (Platform 1)	CPU/FPGA (Platform 2)	CPU-only (Platform 3)	CPU/GPU (Platform 1)	CPU/FPGA (Platform 2)	CPU-only (Platform 3)
DL Circuitry Reduction	4.3×	5.6×	5.4×	40.4×	191.5×	43.3×	6.2×	5.1×	6.2×
Number of Training Iterations	1.1×	1.7×	1.3×	1.8×	3.2×	2.1×	1.1×	1.4×	1.1×
Time Per Iteration Reduction	3.9×	5.7×	3.6×	11.4×	32.7×	9.2×	3.1×	4.1×	2.8×
Training Time Improvement	3.5×	3.3×	2.7×	6.3×	10.2×	4.3×	2.8×	2.9×	2.5×
Execution Time Improvement	1.2×	1.1×	1.2×	10.8×	9.7×	10.3×	5.9×	4.7×	5.6×

As our comparison *baseline*, we implement the state-of-the-art DL methodology in which dropout technique is used to boost the accuracy [SHK$^+$14] and the global DL parameters are updated synchronously. We use Tanh as our activation function for each hidden layer, $\beta = 10\%$ as projection threshold, and $b_s = 100$ for SGD. For each of the visual, audio, and smart-sensing benchmarks, Table 3.3 reports the number of training iterations, time per iteration improvement, and the circuit footprint reduction achieved by Deep3 compared to the baseline approach on each of the platforms. Note that although Deep3's stochastic approach results in a higher *number of iterations* to train the global DL network within a specified accuracy compared to the baseline, it gains significant overall runtime improvement by lessening the required *time per training iteration*. This improvement is achieved by customizing the data and DL circuitry to fit into the fast cache memory, avoiding the costly communication to the main memory of the platform. The training runtime improvement, in turn, also translates to significant savings in the energy consumption. We emphasize that the use of domain-customized DL accelerators such as Tensor Processing Unit (TPU) provide an orthogonal means for performance improvement. As such, Deep3 can achieve even further improvement by leveraging such accelerators.

In Table 3.3, the higher DL circuitry reduction for the CPU-FPGA setting is due to its limited available block RAM budget, which dictates a smaller local network compared to the other two platform settings. As we experimentally verify, although this DL circuitry reduction results in a higher number of iterations to converge to the same accuracy, Deep3 gains significant overall runtime improvement by reducing the required time per training iteration as a result of avoiding the communication with the off-chip memories.

Figure 3.5 compares the runtime performance of Deep3 with the existing parallel DL solutions on a cluster of Intel core-i7 processors. In this experiment, we train Alexnet [KSH12] with 60 million parameters using scaled CIFAR100 dataset. Deep3 shows excellent scaling pattern as the number of training cores increases. This is mainly because of the asynchronous depth-first nature of Deep3 framework that enables independent updating of local networks while balancing communication versus computation. Note that TensorFlow leverages an asynchronous breadth-first approach for distribution of DL workload (Section 3.4), and the baseline is devised based on a synchronous DL updating model as in Theano.

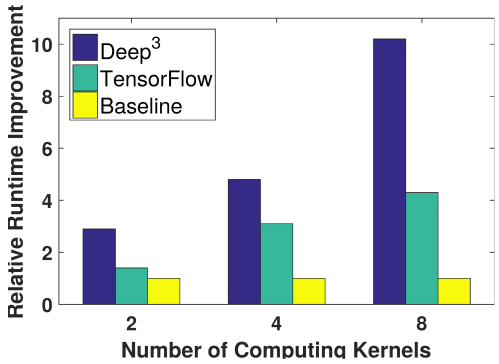


Figure 3.5: Deep3 relative runtime improvement.

Discussion. Our evaluations demonstrate the importance of hardware customization to reduce the cost of expensive training of deep networks. Note that each application demands its own model selection and projection tuning. This means for training DL models, one requires to perform multiple rounds of training with varying parameters including number of hidden layers, number of units per layer, batch size, activation function, etc., until the best ones are empirically identified. The complex and iterative nature of the DL methods rapidly amortizes the customization cost as a pre-processing step.

3.7 Summary

We present Deep3, an automated end-to-end framework that provides holistic data and platform aware solutions and tools to efficiently perform DL training and execution. Deep3 leverages data, network, and platform customization to optimize DL physical performance. It maps the global DL training into smaller size local neural networks based on a novel extensible depth-first graph traversal methodology. Our approach minimizes the inter-core and inter-memory communication overhead while minimally affecting the accuracy. Our accompanying libraries automate adaptation of Deep3 for rapid prototyping of an arbitrary DL task. Our extensive evaluations show that Deep3 achieves significant improvements in DL training and execution.

3.8 Acknowledgements

This chapter, in part, has been published at (i) the Proceedings of 2017 International Design Automation Conference (DAC) and appeared as: Bita Darvish Rouhani, Azalia Mirhoseini, and Farinaz Koushanfar "Deep3: Leveraging Three Levels of Parallelism for Efficient Deep Learning", and (ii) the Proceedings of the 2016 International Symposium on Low Power Electronics and Design (ISLPED) as: Bita Darvish Rouhani, Azalia Mirhoseini, and Farinaz Koushanfar "Delight: Adding energy dimension to deep neural networks". The dissertation author was the primary author of this material.

Chapter 4

DeepFense: Online Accelerated Defense Against Adversarial Deep Learning

Recent advances in adversarial Deep Learning (DL) have opened up a largely unexplored surface for malicious attacks jeopardizing the integrity of autonomous DL systems. With the wide-spread usage of DL in critical and time-sensitive applications, including unmanned vehicles, drones, and video surveillance systems, online detection of malicious inputs is of utmost importance. We propose DeepFense, the *first end-to-end automated* framework that simultaneously enables efficient and safe execution of DL models. DeepFense formalizes the goal of thwarting adversarial attacks as an optimization problem that minimizes the rarely observed regions in the latent feature space spanned by a DL network. To solve the aforementioned minimization problem, a set of complementary but disjoint modular redundancies are trained to validate the legitimacy of the input samples in parallel with the victim DL model. DeepFense leverages hardware/software/algorithm co-design and customized acceleration to achieve just-in-time performance in resource-constrained settings. The proposed countermeasure is unsupervised, meaning that no adversarial sample is leveraged to train modular redundancies. We further provide an accompanying API to reduce the non-recurring engineering cost and ensure automated

adaptation to various platforms. Extensive evaluations on FPGAs and GPUs demonstrate up to two orders of magnitude performance improvement for online adversarial sample detection.

4.1 Introduction

Deep Neural Networks (DNNs) have enabled a transformative shift in various scientific fields ranging from natural language processing and computer vision to health-care and intelligent transportation [MPC16, DY14, Kno15]. Although DNNs demonstrate superb accuracy in controlled settings, it has been shown that they are particularly vulnerable to adversarial samples: carefully crafted input instances which lead machine learning algorithms into misclassifying while the input changes are imperceptible to a naked eye.

In response to the various adversarial attack methodologies proposed in the literature (e.g., [CW17b, GSS14, KGB16, MDFF16]), several research attempts have been made to design DL strategies that are more robust in the face of adversarial examples. The existing countermeasures, however, encounter (at least) two sets of limitations: (i) Although the prior-art methods have reported promising results in addressing adversarial attacks in black-box settings [MC17, ZNR17, SJGZ17], their performance has been shown to significantly drop in white-box scenarios where the adversary has the full knowledge of the defense mechanism [CW17a]. (ii) None of the prior works have provided an automated hardware-accelerated system for online defense against adversarial inputs. Due to the wide-scale adoption of deep learning in sensitive autonomous scenarios, it is crucial to equip all such models with a defense mechanism against the aforementioned adversarial attacks.

We propose DeepFense, the first end-to-end hardware-accelerated framework that enables robust and just-in-time defense against adversarial attacks on DL models. Our key observation is that the vulnerability of DNNs to adversarial samples originates from the existence of rarely-explored sub-spaces spanned by the activation maps in each (hidden) layer. This phenomenon is

particularly caused by (i) the high dimensionality of activation maps and (ii) the limited amount of labeled data to fully traverse/learn the underlying space. To characterize and thwart potential adversarial sub-spaces, we propose a new method called *Modular Robust Redundancy* (MRR). MRR is robust against the state-of-the-art adaptive white-box attacks in which the adversary knows everything about the victim model and its defenders.

Each modular redundancy characterizes the explored subspace in a given layer by learning the Probability Density Function (pdf) of typical data points and marking the complement regions as rarely-explored/risky. Once such characterization is obtained, the checkpointing modules evaluate the input sample in parallel with the victim model and raise alarm flags for data points that lie within the risky regions. The MRRs are trained in unsupervised settings meaning that the training dataset is merely composed of typical benign samples. This, in turn, ensures resiliency against potential new attacks. Our unsupervised countermeasure impacts neither the training complexity nor the final accuracy of the victim DNN.

DeepFense is devised based on a hardware/software/algorithm co-design approach to enable safe DL while customizing system performance in terms of latency, energy consumption, and/or memory footprint with respect to the underlying resource provisioning. There is a trade-off between system performance and robustness against adversarial attacks that is determined by the number of modular redundancies. DeepFense provides an automated tool to adaptively maximize the robustness of the defense model while adhering to the user-defined and/or hardware-specific constraints. We chose FPGAs to provide fine-grained parallelism and just-in-time response by our defender modules. The customized data path for memory access and network schemes on FPGA, in turn, helps to improve the overall system energy efficiency.

Although several hardware-accelerated tools for DL execution have been proposed in the literature, e.g. [ZLS+15, CDS+14, SPA+16, SGK17, RMK17a], none of them have been particularly optimized for in-time defense against adversarial inputs. For instance, defenders require a custom layer to characterize and compare each incoming data sample against the pdf of

legitimate data. These types of custom layers are atypical to conventional DNNs and have not been addressed in prior works. In summary, the contributions of this work are as follows:

- Proposing DeepFense, the first hardware/software/algorithm co-design that empowers online defense against adversarial samples for DNNs. DeepFense methodology is unsupervised and robust against the most challenging attack scenario in real-world applications (white-box attacks).

- Incepting the idea of Modular Robust Redundancy as a viable security countermeasure for adversarial deep learning. DeepFense leverages dictionary learning and probability density functions to statistically detect abnormalities in the inputted data samples. Based on the result of our analysis, we provide new insights on the reason behind the existence of adversarial sample transferrability between different models.

- Devising an automated customization tool to adaptively maximize DL robustness against adversarial samples while complying with the underlying hardware resource constraints in terms of run-time, energy, and memory footprint.

- Providing the first implementation of custom streaming-based DL defense using FPGAs. DeepFense leverages dictionary learning and probability density functions to statistically detect abnormalities in the inputted data samples.

- Performing extensive proof-of-concept evaluations on common DL benchmarks against the state-of-the-art adversarial attacks reported to-date. Thorough performance comparison on various hardware platforms including embedded CPUs, GPUs, and FPGAs corroborates DeepFense's efficiency.

4.2 DeepFense Global Flow

Figure 4.1 illustrates the global flow of DeepFense framework. We consider a system consisting of a single classifier (a.k.a., victim model) and a set of defender modules aiming to detect adversarial samples. DeepFense consists of two main phases to characterize and thwart adversarial attacks: (i) offline pre-processing phase to train defender modules, and (ii) online execution phase in which the legitimacy of each incoming input data is validated on the fly. The one-time pre-processing phase is performed in software while the recurrent execution phase is accelerated using FPGA.

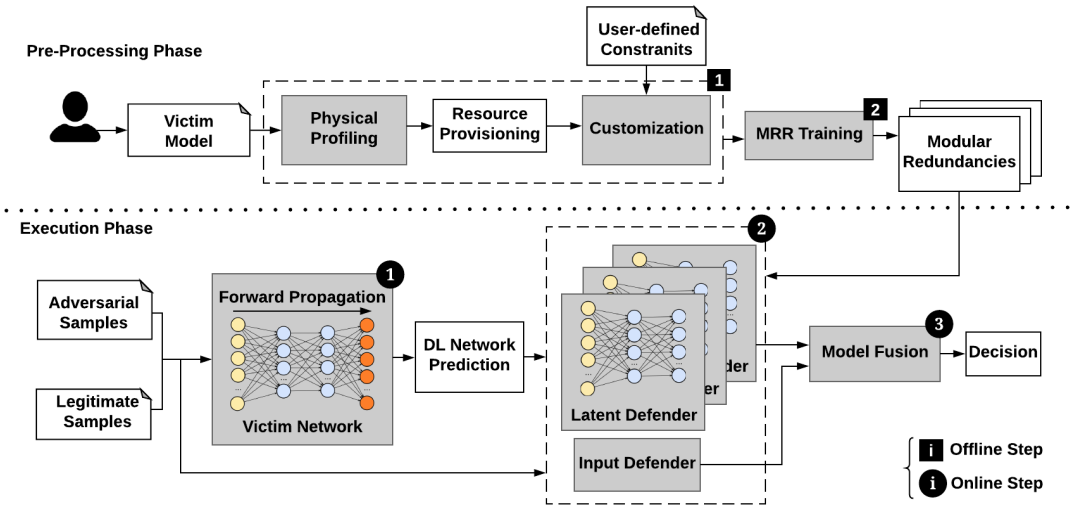


Figure 4.1: Global flow of the DeepFense framework. DeepFense takes as input the high-level description of a DL model together with the proposed defender topologies. Based on the user-provided constraints, DeepFense outputs the best defense layout to ensure maximum throughput and power efficiency, customized for the resource-constrained target hardware platform.

The pre-processing phase consists of two key tasks as explained in the following:

1 Resource Profiling and Design Customization. There is a trade-off between execution run-time and system reliability in terms of successful adversarial detection rate. DeepFense uses physical profiling to estimate resource utilization for the victim model as well as the defender modules. The output of physical profiling along with a set of user-defined constraints (e.g., real-time requirements) is then fed into the design customization unit to determine the viable

number of defenders and their appropriate locations based on the sensitivity of DNN layers (Section 4.4.3). The customization unit analyzes the trade-off between model reliability, resource limitation, and throughput to decide the best combination of defenders suitable to the task and customized for the target hardware.

❷ **Training Modular Redundancies.** DeepFense trains a set of redundancy modules (checkpoints) to isolate potential adversarial sub-spaces. The redundancy modules can be categorized into two classes, namely the *Input Defenders* (Section 4.3.3) and the *Latent (Intermediate) Defenders* (Section 4.3.2). Each defender targets a particular layer in the victim model and is trained with the goal of separating data manifolds and characterizing the underlying pdf by careful realignment of legitimate data within each class.

Once the redundancy modules are trained and customized per hardware and/or user-defined physical constraints, the underlying DL model is ready to be deployed for online execution. DeepFense performs three tasks as follows for the execution phase.

❶ **Forward Propagation.** The predicted class for each incoming sample is acquired through forward propagation in the victim DNN. The predicted output is then fed to the defenders for validation purposes.

❷ **Validation.** DeepFense leverages the checkpoints learned in the pre-processing phase to validate the legitimacy of the input data and the associated label determined in the forward propagation step. In particular, samples that do not lie in the *user-defined* probability interval which we refer to as the *Security Parameter (SP)* are discarded as suspicious samples. SP is a constant number in the range of $[0-100]$ which determines the hardness of adversarial detectors. For applications with excessive security requirements, a high SP value assures full detection of adversarial samples.

❸ **Model Fusion.** The outputs of the redundancy modules are finally aggregated to compute the legitimacy probability of the input data and its associated inference label (Section 4.3.4).

Attack Model. We consider the *adaptive white-box* threat model as the most powerful attacker

that can appear in real-world DL applications. In this scenario, we assume the attacker knows everything about the victim model including the learning algorithm, model topology, and parameters. With the presence of DeepFense parallel defenders, the adversary is required to mislead *all* defenders to succeed in forging an adversarial sample as a legitimate input.

4.3 DeepFense Methodology

DeepFense trains a number of modular redundancies to characterize the data density distribution in the space spanned by the victim model. In this section, we first provide a motivational example for MRR methodology. We then elaborate on the MRR modules that checkpoint the intermediate DL layers (latent defenders) and input space (input defenders). Lastly, we discuss the model fusion to aggregate the MRR outputs and derive the final decision.

4.3.1 Motivational Example

The rationale behind our MRR methodology is not only to thwart adversarial attacks in black-box settings (where the adversary is not aware of the defense mechanism), but also to boost the reliability of the model prediction in presence of adaptive white-box attacks. Table 4.1 compares the success rate of the adaptive white-box Carlini&WagnerL2 attack [CW17a] against MRR methodology with the prior-art countermeasures on MNIST benchmark.[1] We define the *False Positive (FP)* rate as the ratio of legitimate test samples that are mistaken for adversarial samples by DeepFense. The *True Positive (TP)* rate is defined as the ratio of adversarial samples detected by DeepFense. As shown, increasing the number of MRR modules not only decreases the attack success rate but also yields a higher perturbation in the generated adversarial samples. The superior performance of DeepFense is associated with learning the distribution of legitimate samples as opposed to prior works which target altering the decision boundaries.

[1] We used the open-source library https://github.com/carlini/MagNet to implement the Carlini&WagnerL2 adaptive attack.

Table 4.1: Motivational example. We compare the MRR methodology against prior-art works including Magnet [MC17], Efficient Defenses Against Adversarial Attacks [ZNR17], and APE-GAN [SJGZ17] in the white-box setting. For each evaluation, the adversarial perturbation (L_2 distortion) is normalized to that of the attack without the presence of any defense mechanism.

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

4.3.2 Latent Defenders

Each latent defender module placed at the n^{th} layer of the victim model is a neural network architecturally identical to the victim. This homogeneity of topology enables the defenders to suitably address the vulnerabilities of the victim network. We consider a Gaussian Mixture Model (GMM) as the prior probability to characterize the data distribution at each checkpoint location. We emphasize that our proposed approach is rather generic and is not restricted to the GMM. The GMM distribution can be replaced with any other prior depending on the application.

Training a single latent defender. To effectively characterize the explored sub-space as a GMM distribution, one is required to minimize the entanglement between every two Gaussian distributions (corresponding to every two different classes) while decreasing the inner-class diversity. There are three main steps to train one latent defender.

Step 1. Replicating the victim neural network and all its parameters. An L_2 normalization layer is inserted in the desired checkpoint location. The normalization layer maps the latent features (activations), $f(x)$, into the Euclidean space such that the acquired activation maps are bounded to a hyper-sphere, i.e., $\|f(x)\|_2 = 1$. This normalization is crucial as it partially removes the effect of over-fitting to particular data samples that are highly correlated with the underlying DL parameters. The L_2 norm is selected to be consistent with our assumption of GMM prior distribution. This norm can be easily replaced by a user-defined norm through our provided API.

Step 2. Fine-tuning the replicated deep neural network to enforce disentanglement of data features (at a particular checkpoint location) and characterize the pdf of explored sub-spaces. To do so,

we optimize the defender module by adding the following loss function to the conventional cross entropy loss function:

$$\gamma\,[\ \underbrace{\|C^{y^*} - f(x)\|_2^2}_{loss_1}\ -\ \underbrace{\Sigma_{i\neq y^*}\|C^i - f(x)\|_2^2}_{loss_2}\ +\ \underbrace{\Sigma_i(\|C^i\|_2 - 1)^2}_{loss_3}\ ]. \tag{4.1}$$

Here, γ is a trade-off parameter that specifies the contribution of the additive loss term, $f(x)$ is the corresponding feature vector of input sample x at the checkpoint location, y^* is the ground-truth label, and C^i denotes the center corresponding to class i. The center values C^i and intermediate feature vectors $f(x)$ are trainable variables that are learned by fine-tuning the defender module. In our experiments, we set the parameter γ to 0.01 and retrain the defender with the same optimizer used for training the victim DNN. The learning rate is set to 0.1 of that of the victim model as the model is already in a relatively good local minimum.

The first term ($loss_1$) in Eq. (4.1) aims to condense latent data features $f(x)$ that belong to the same class. Reducing the inner-class diversity, in turn, yields a sharper Gaussian distribution per class. The second term ($loss_2$) intends to increase the intra-class distance between different categories and promote separability. If the loss function consists solely of the first two terms in Eq. (4.1), the pertinent model may diverge by pushing the centers to $C^i \mapsto \pm\infty$. We add the term, $loss_3$, to ensure that the pertinent centers lie on a unit hyper-sphere and avoid divergence.

Step 3. After applying Step 2, the latent data features are mapped to discrete GMMs. Each GMM is defined by the first order (mean) and second order statistics (covariance) of the legitimate activations. Using the obtained distributions, DeepFense profiles the percentage of benign samples lying within different L_2 radius of each GMM center. We leverage a security parameter in the range of $[0-100]$ to divide the underlying space into the sub-space where the legitimate data lives and its complementary adversarial sub-space. The acquired percentile profiling is employed to translate the user-defined SP into an L_2 threshold which is later used to detect malicious samples during the online execution phase.

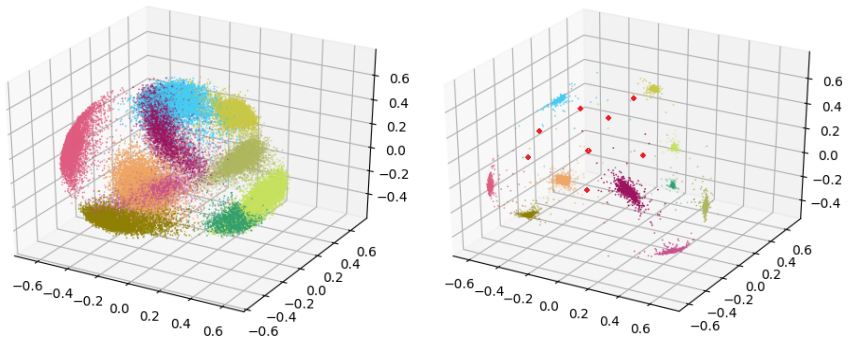


Figure 4.2: Example feature samples in the second-to-last layer of LeNet3 model trained for classifying MNIST data before (left figure) and after (right figure) data realignment performed in Step 2. The majority of adversarial samples (the red dot points) reside in low density regions.

Figure 4.2 illustrates the activation maps in the second-to-last layer of a LeNet3 network trained for classifying MNIST data before and after data realignment. As demonstrated, the majority of adversarial samples reside in the rarely-explored regions; as such these malicious samples that can be effectively detected by DeepFense latent defenders.

Training multiple negatively correlated defenders. The reliability of MRR defense can be increased by training multiple defenders per layer that are negatively correlated, as opposed to using only one latent defender. Consider a defender module that maps a legitimate input x to the feature vector $f(x)$, where $f(x)$ is close (in terms of Euclidean distance) to the corresponding center C^i. An adversary trying to mislead this defender would generate a perturbed input $x+\eta$ such that $f(x+\eta)$ is far from C^i and close to another target center C^j. In other words, the adversary would like to increase the $loss_1$ term in Eq. (4.1). To mitigate such adaptive attacks, we propose to train a Markov chain of latent defenders.

To build the corresponding Markov chain of latent defenders, we start off by training a single defender module as described earlier in this section. Next, we generate a new set of training data that can enforce negative correlations between the current defender module and the next defender. In particular, the n^{th} defender of this chain takes an input data x, generates a perturbation η, and feeds $clip(x+\eta)$ to the $(n+1)^{th}$ defender. The $clip(\cdot)$ operation simply clips

the input sample in a valid range of numerical values, e.g., between 0 and 1. The perturbation η is chosen as $\eta = \frac{\partial loss_1}{\partial x}$, where the $loss_1$ term (See Eq. (4.1)) corresponds to the n^{th} defender. Given this new dataset of perturbed samples, benign data points that deviate from the centers in the n^{th} defender will be close to the corresponding center in the $(n+1)^{th}$ defender. As such, simultaneously deceiving all the defenders requires a higher amount of perturbation.

4.3.3 Input Defender

One may speculate that an adversary can add a structured noise to a legitimate sample such that the data point is moved from one cluster to the center of the other clusters; thus fooling the latent defender modules. The risk of such an attack approach is significantly reduced by leveraging sparse signal recovery techniques. We use dictionary learning to measure the Peak Signal-to-Noise Ratio (PSNR) of each incoming data and filter out atypical samples in the input space. An input checkpoint is configured in two main steps.

Step 1. We learn a separate dictionary for each class by solving:

$$\underset{D^i}{argmin}\ \frac{1}{2}\|Z^i - D^i V^i\|_2^2 + \beta\|V^i\|_1 \quad s.t. \quad \|D_k^i\| = 1,\ 0 \leq k \leq k_{max}. \tag{4.2}$$

Here, Z^i is a matrix whose columns are pixels extracted from different regions of input images belonging to category i. For instance, if we consider 8×8 patches of pixels, each column of Z^i would be a vector of 64 elements. The goal of dictionary learning is to find matrix D^i that best represents the distribution of pixel patches from images belonging to class i. We denote the number of columns in D^i by k_{max}. For a certain D^i, the image patches Z^i are represented with a sparse matrix V^i, and $D^i V^i$ is the reconstructed sample. We leverage Least Angle Regression (LAR) to solve Eq. (4.2). During the execution phase, the input defender module takes the output of the victim DNN (e.g., predicted class i) and uses Orthogonal Matching Pursuit (OMP) [TG07] to sparsely reconstruct the input with the corresponding dictionary D^i. The input sample labeled

as class i should be well-reconstructed as D^iV^* with a high PSNR value, where V^* is the optimal solution obtained by OMP.

Step 2. We profile the PSNR percentiles of legitimate samples within each class and find the corresponding threshold that satisfies the user-defined security parameter. If an incoming sample has a PSNR lower than the threshold (i.e., high perturbation after reconstruction by the corresponding dictionary), it is regarded as malicious data.

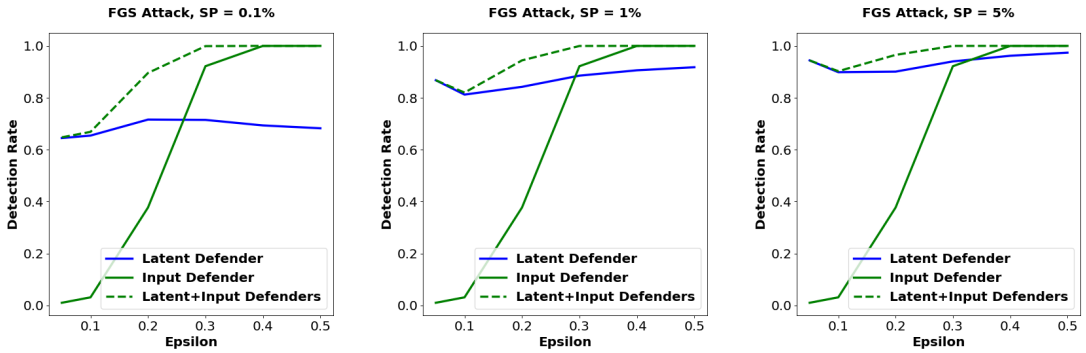


Figure 4.3: Adversarial detection rate of the latent and input defender modules as a function of the perturbation level.

Figure 4.3 demonstrates the impact of perturbation level (ε) on the adversarial detection rate for two different security parameters (cut-off thresholds). In this experiment, we have considered the Fast Gradient Sign (FGS) attack [GSS14] on LeNet3 MNIST benchmark with a single latent defender inserted at the second-to-last layer. As shown, the use of input dictionaries facilitates detection of adversarial samples with relatively high perturbations.

4.3.4 Model Fusion

Each defender module in DeepFense takes as input a sample x and generates a binary output $d_k \in \{0,1\}$ with value 1 denoting an adversarial sample. This binary decision is based on the user-defined security parameter. To aggregate the binary random variables $\{d_1,\ldots,d_N\}$ into a

single decision a, we compute the probability of the input being adversarial as:

$$\begin{aligned} &P(a=1|\{d_1,d_2,\ldots,d_n\}) = 1-\prod_{n=1}^{N}(1-P_n)^{d_n}, \\ &P_n = P(a=1|d_n=1). \end{aligned} \tag{4.3}$$

This formulation resembles the well-known noisy-OR terminology used in statistical learning [Die93]. In MRR methodology, each defender has a parameter P_n which indicates the likelihood of a sample being adversarial given that the n^{th} defender has labeled it as a malicious sample. If all detectors have $P_n = 1$, then the formulation in Eq. (4.3) is equivalent to the logical OR between $\{d_1,\ldots,d_N\}$. The P_n parameters can be estimated by evaluating the performance of each individual defender. For this purpose, we use a subset of training data and create adversarial samples with different attack algorithms. If the defender suspects M_{False} samples of the legitimate training data and M_{True} samples of the adversarial data set, the probability $P(a=1|d_n=1)$ is:

$$P_n = \frac{M_{True}}{M_{False}+M_{True}}. \tag{4.4}$$

The output of the noisy-OR model is in the unit interval. DeepFense raises alarm flags for samples with $P(a=1|\{d_1,d_2,\ldots,d_n\}) \geq 0.5$.

4.4 DeepFense Hardware Acceleration

In this section, we first discuss the hardware architecture of latent and input defenders that enables a high throughput and low energy realization of recurrent execution phase. We, then, discuss the resource profiling and automated design customization unit.

4.4.1 Latent Defenders

During execution, each incoming sample is passed through the latent defender modules that are trained offline (Section 4.3.2). The legitimacy probability of each sample is then approximated by measuring the L_2 distance with the corresponding GMM center. The latent defenders can be situated in any layer of the victim network, therefore, the extracted feature vector from the DNN can be of high cardinality. High dimensionality of the GMM centers may cause shortage of memory as well as increasing the computational cost and system latency. In order to mitigate the curse of dimensionality, we perform Principal Component Analysis (PCA) on the outputs of the latent defenders before measuring the L_2 distance. For the latent defenders in the DeepFense framework, PCA is performed such that more than 99% of the energy is preserved.

The most computationally-intensive operation in DNN execution is matrix-matrix multiplication. Recent FPGAs provide hardened DSP units together with the re-configurable logic to offer a high computation capacity. The basic function of a DSP unit is a multiplication and accumulation (MAC). In order to optimize the design and make use of the efficient DSP slices, we took a parallelized approach to convert the DNN layer computations into multiple operations running simultaneously as suggested in [SPA+16]. Figure 4.4 illustrates the high-level schematic of a latent defender kernel. Two levels of parallelism are applied in the implementation of the DNN layers, controlled by parameters N_{PE} and N_{PU} which denote the parallelism level in the input processing and output generation stage, respectively. The aforementioned parameters are static across all layers of the DNN model. In order to achieve maximum throughput, it is essential to fine-tune the parallelism parameters.

There is a trade-off between the number of parallel employed Processing Units (PU) and hardware complexity in terms of memory access. An increase in the number of parallel computation units will not always result in better throughput since the dimensionality of the data and divisibility into ready-to-process batches highly affects the efficiency of these parallel units. There are two implementation scenarios in DeepFense; each of the Processing Units (PU) can

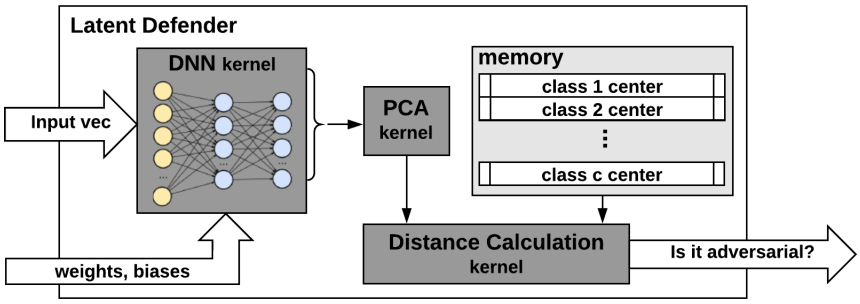


Figure 4.4: Architecture of DeepFense latent defender. The pertinent activations are acquired by propagating the input sample through the defender. PCA is then applied to reduce the dimensionality of the obtained activation. The *L*2 distance with the corresponding GMM center determines the legitimacy of the input.

either be assigned a subset of the layer output features (scenario 1) or the whole feature map for computation (scenario 2). In the first scenario, multiple PUs work in parallel to gradually compute all output features in each DNN layer while in the second scenario, batches of input samples can be processed simultaneously where the batch size is equal to the number of PUs. DeepFense switches between these two scenarios based on the DNN architecture, dimensionality of the layers, and/or available resources. Figure 4.5 shows an example of the design space exploration for the MNIST and SVHN benchmarks [2]. Based on the underlying resource constraints, the number of PUs is uniquely determined by the number of Processing Engines (PE) per PU (the horizontal axis in Figure 4.5).

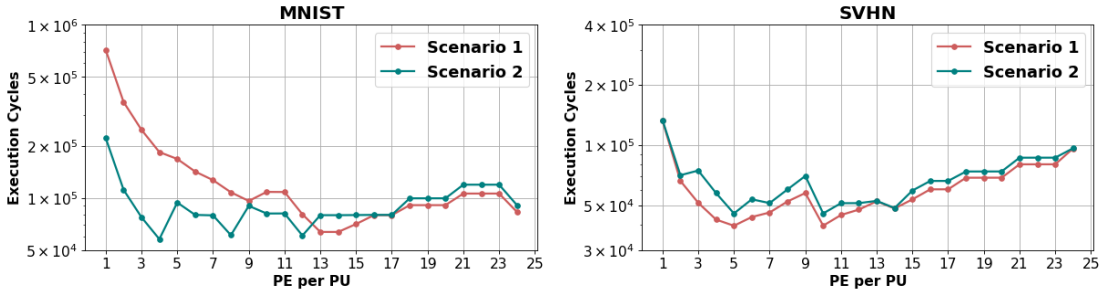


Figure 4.5: Design space exploration for MNIST and SVHN benchmarks on *Xilinx Zynq-ZC702* FPGA. DeepFense finds the optimal configuration of PEs and PUs to best fit the DNN architecture and the available hardware resources.

[2] See Section 4.5 for details of each benchmark.

To minimize the latency of latent defenders, we infuse the PCA kernel into the defender DNNs. Collectively, all transformations from the original input space to the space spanned by principal components can be shown as a vector-matrix multiplication $T = XW_L$ where W_L is a matrix whose columns are Eigenvectors obtained from the legitimate data. The transformation $T = XW_L$ maps a data vector X from an original space of p variables to a new space of L uncorrelated variables. As such, the PCA kernel can be replaced with a *Dense* layer, appended to the defender DNN architecture.

4.4.2 Input Defenders

The input defender module relies on sparse signal reconstruction to detect abnormalities in the victim DNN's input space. Execution of OMP algorithm is the main computational bottleneck in the input defender modules. We provide a scalable implementation of the OMP routine on FPGA to enable low-energy and in-time analysis of input data. Analyzing large chunks of data requires a boost in the computational performance of the OMP core. This can be achieved by modifying the OMP algorithm such that it maximally utilizes the available on-chip resources. Figure 4.6 illustrates the high-level schematic of an input defender's kernel. Here, the *support set* contains columns of the dictionary matrix that have been chosen so far in the routine.

OMP execution includes two computationally expensive steps, namely the matrix-vector multiplication and the LS optimization. Each of these steps includes multiple dot product computations. The sequential nature of the dot product, renders the usage of pipelining inefficient. Thereby, we use a *tree-based* reduction technique to find the final value by adding up the partial results produced by each of the parallel processes. Figure 4.7 outlines the realization of a tree-based reduction module. The reduction module takes an array of size $2M$ as its input (array a) and oscillates between two different modes. In mode 0, the function reduces a by using $temp$ as a temporary array. In mode 1, $temp$ is reduced using a. This interleaving between the two arrays ensures maximal utilization of memory blocks. The final result is returned based on the

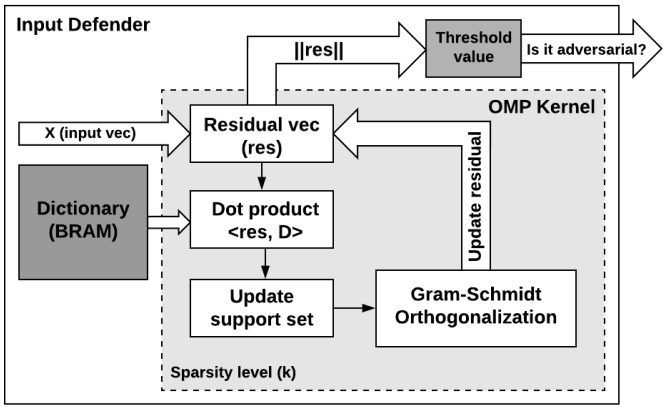


Figure 4.6: Architecture of DeepFense input defender. The OMP core iteratively reconstructs the input vector with the learned dictionary. The reconstruction error is used to determine the input legitimacy.

final mode of the system. We pipeline and unroll the tree-based reduction function to provide a more efficient solution. Cyclic memory array partitioning along with loop unrolling are also leveraged to ensure maximum system throughput.

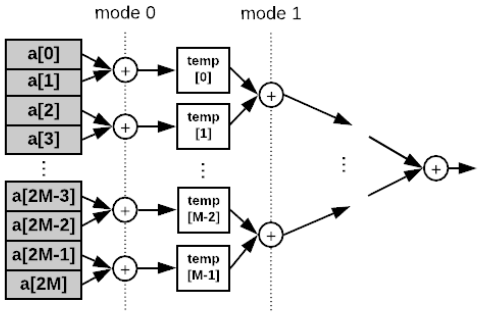


Figure 4.7: Realization of the tree-based vector reduction algorithm.

The LS optimization step is performed using QR decomposition to reduce implementation complexity and make it well-suited for hardware accelerators. The Gram-Schmidt orthogonalization technique gradually forms the orthogonal matrix Q and upper-triangular matrix R to iteratively calculate the decomposition. Algorithm 2 outlines the modified Gram-Schmidt incremental orthogonalization method [GVL12]. Using the Gram-Schmidt methodology, the

residual update can be considerably simplified as the following:

$$r^i \leftarrow r^{i-1} - Q_i (Q_i)^T r^{i-1} \tag{4.5}$$

Algorithm 2 Incremental QR decomposition with modified Gram-Schmidt

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

1: $R^n \leftarrow \begin{bmatrix} R^{n-1} & 0 \\ 0 & 0 \end{bmatrix}$, $\varepsilon^n \leftarrow D_{\Lambda^n}$
2: **for** *j = 1,...,n-1* **do**
3: $\quad R^n_{jn} \leftarrow (Q^{n-1})^T_j \varepsilon^n$
4: $\quad \varepsilon^n \leftarrow \varepsilon^n - R^n_{jn} Q^{n-1}_j$
5: $R^n_{nn} = \sqrt{||\varepsilon^n||_2^2}$
6: $Q^n = Q^{n-1} \varepsilon^n / R^n_{nn}$

The update residual vector r at the end of each iteration is made orthogonal to the selected dictionary samples. As such, none of the columns of matrix D would be selected twice during one call of the OMP algorithm. Based on this observation, we reuse the same set of block memories initially assigned to the dictionary matrix D to store the newly computed columns of the Q matrix, per iteration [RSMK15].

4.4.3 Automated Design Customization

We provide an automated customization unit that maximizes the robustness of a DNN within the limits of the pertinent resource provisioning. Our automated optimization ensures ease of use and reduces the non-recurring engineering cost. DeepFense's customization unit takes as input the high-level description of the defenders in Caffe together with the available resources in terms of storage, DSP units, and run-time. It then outputs the best combination of defender modules to ensure maximum robustness against adversarial attacks while adhering

to the available resources. We thoroughly examined the performance and resource utilization for different building blocks of a DNN. These blocks include the essential hyper-parameters for instantiating the desired DL model, including but not limited to the number of layers and the corresponding input/output sizes. This enables DeepFense to estimate the upper bound for implementation of a DNN on resource-constrained platforms.

Dictionary matrices leveraged in the input defender as well as the weights and biases of the latent defenders are stored in the on-chip DRAM memory to be accessed during the execution phase. Upon computation, data is moved from the DRAM to Block RAMs (BRAM) which enable faster computations. Our evaluations on various FPGAs show that the main resource bottlenecks are the BRAM capacity and the number of DSP units. As such, DeepFense optimizes the configuration of the defenders with regard to these two constraints. In particular, DeepFense solves the following optimization to find the best configuration for the number of defenders N_{def} and the number of processing units N_{PU} per defender.

$$
\begin{aligned}
&\underset{N_{PU},N_{def}}{Maximize}\ (DL_{robustness}) \qquad s.t.: \\
&T_{def}^{max} \leq T_u, \quad N_{def} \times N_{PU} \times DSP_{PU} \leq R_u, \\
&N_{PU} \times [max(size(W^i)) + max(|X^i| + |X^{i+1}|)] \leq M_u,
\end{aligned}
\tag{4.6}
$$

where T_u, M_u, and R_u are user-defined constraints for system latency, BRAM budget, and available DSP resources, respectively. Here, $size(W^i)$ denotes the total number of parameters and $|X^i|$ is the cardinality of the input activation in layer i. DSP_{PU} indicates the number of DSP slices used in one processing unit. Variable T_{def}^{max} is the maximum required latency for executing the defender modules. DeepFense considers both sequential and parallel execution of defenders based on the available resource provisioning and size of the victim DNN. Once the optimization is solved for N_{PU}, N_{PE} is uniquely determined based on available resources.

The OMP unit in DeepFense framework incurs a fixed memory footprint and latency for a

given application. As such, the optimization of Eq. (4.6) does not include this constant overhead. Instead, we exclude this overhead from the user-defined constraints and use the updated upper bounds. In particular, for an OMP kernel, the required computation time can be estimated as $\beta n(kl+k^2)$ where n indicates the number of elements in the input vector, l is the dictionary size, and k represents the sparsity level. β is system-dependant and denotes the number of cycles for one floating point operation. The memory footprint for OMP kernel is merely a function of the dictionary size.

Our customization unit is designed such that it maximizes the resource utilization to ensure maximum throughput. DeepFense performs an exhaustive search over the parameter N_{PU} and solves the equations in 4.6 using the Karush-Kuhn-Tucker (KKT) method to calculate N_{def}. The calculated parameters capture the best trade-off between security robustness and throughput. Our optimization outputs the most efficient layout of defender modules as well as the sequential or parallel realization of defenders. This constraint-driven optimization is non-recurring and incurs a negligible overhead.

4.5 Experiments

We evaluate DeepFense on three different DNN architectures outlined in Table 4.2. Each DNN corresponds to one dataset: MNIST, SVHN, and CIFAR-10. We report the robustness of the aforementioned models against four different attacks. The customized defense layout for each network is implemented on two FPGA platforms. A detailed analysis is provided to compare our FPGA implementation with highly-optimized realizations on CPUs and GPUs.

4.5.1 Attack Analysis and Resiliency

We leverage a wide range of attack methodologies (namely, FGS [GSS14], BIM [KGB16], CarliniL2 [CW17b], and Deepfool [MDFF16]) with varying parameters to ensure DeepFense's

Table 4.2: Architectures of evaluated victim deep neural networks. Here, **20C5** denotes a convolutional layer with 20 output channels and $\mathbf{5 \times 5}$ filters, **MP2** indicates a $\mathbf{2 \times 2}$ max-pooling, **500FC** is a fully-connected layer with **500** neurons, and **GAP** is global average pooling. All hidden layers have a "Relu" activation function.

Benchmark	Architecture
MNIST	$(input)\ 1 \times 28 \times 28 - 20C5 - MP2 - 50C5 - MP2 - 500FC - 10FC$
SVHN	$(input)\ 3 \times 32 \times 32 - 20C5 - MP2 - 50C5 - MP2 - 1000FC - 500FC - 10FC$
CIFAR-10	$(input)\ 3 \times 32 \times 32 - 96C3 - 96C3 - 96C3 - MP2 - 192C3 - 192C3 - 192C3$ $- MP2 - 192C3 - 192C1 - 10C1 - GAP - 10FC$

generalizability. The perturbation levels are selected such that the adversarial noise is undetectable by a human observer (Table 4.3 summarizes the pertinent attack parameters).

Table 4.3: Adversarial attacks' hyper-parameters. For CarliniL2 attack, "C" denotes the confidence, "LR" is the learning rate, "steps" is the number of binary search steps, and "iterations" stands for the maximum number of iterations. Superscripts ($m \rightarrow$ MNIST, $s \rightarrow$ SVHN, $c \rightarrow$ CIFAR-10, $a \rightarrow$ all) are used to indicate the benchmarks for which the parameters are used.

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

There is a trade-off between the false positive and the true positive detection rates that can be controlled using the security parameter (see Section 4.3). The Area Under Curve (AUC) for a TP versus FP plot is a measure of accuracy for adversarial detection. A random decision has an AUC score of 0.5 while an ideal detector will have an AUC score of 1. Figure 4.8 shows the AUC score obtained by DeepFense for different attack configurations where the adversary knows everything about the model but is not aware of the defenders. For a given number of defenders, the AUC score for MNIST is relatively higher compared to more complex benchmarks (e.g., CIFAR-10). This is consistent with our hypothesis since the unexplored sub-space is larger in higher-dimensional benchmarks. Note that using more defenders eventually increases AUC score.

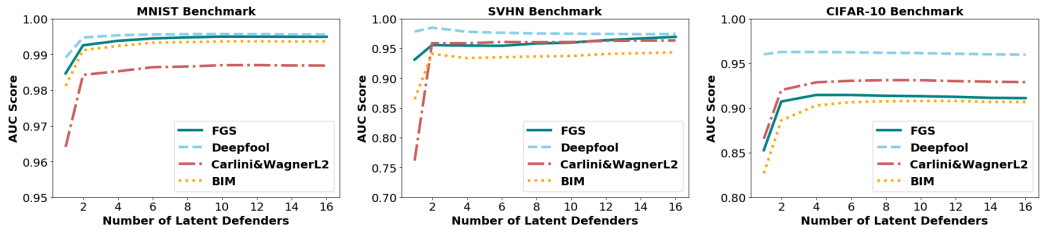


Figure 4.8: AUC score versus the number of defender modules for MNIST, SVHN, and CIFAR-10 datasets.

4.5.2 Performance Analysis

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

Robustness and throughput trade-off. Increasing the number of checkpoints improves the reliability of model prediction in presence of adversarial attacks (Section 4.5.1) at the cost of reducing the effective throughput of the system. In applications with severe resource constraints, it is crucial to optimize system performance to ensure maximum immunity while adhering to the user-defined timing constraints. In scenarios with more flexible timing budget, the customization tool automatically allocates more instances of the defender modules while under strict timing constraints, the robustness is decreased in favor of the throughput. Figure 4.9 demonstrates the throughput versus the number of defender modules for MNIST benchmark on Zynq FPGA. The defender modules are located at the second-to-last layer of the victim DNN. Here the PCA kernel in the defender modules reduces the dimensionality to 10.

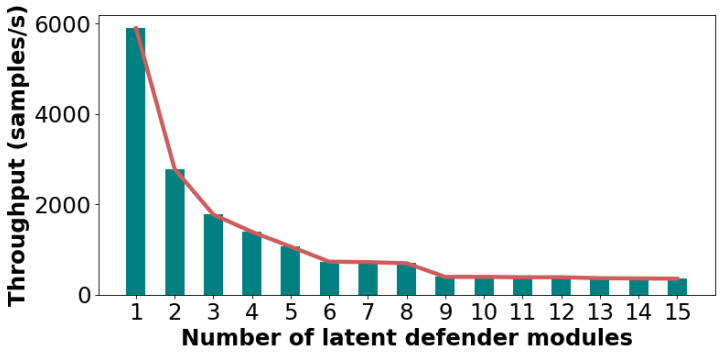


Figure 4.9: Throughput of DeepFense with samples from the MNIST dataset, implemented on the *Xilinx Zync-ZC702* FPGA versus the number of instantiated defenders.

Consider the SVHN benchmark, with the same throughput of 1400 samples per second, DeepFense implementation on UltraScale FPGA can run 8 defenders in parallel while the ARM v8 CPU can maintain the same throughput with only one defender. This directly translates to an improvement in the AUC score from 0.76 to 0.96.

Throughput and energy analysis. To corroborate the efficiency of DeepFense framework, we also evaluate MRR performance on *Jetson TK1* and *Jetson TX2* boards operating in CPU-GPU and CPU-only modes. We define the performance-per-Watt measure as the throughput over the total power consumed by the system. This metric is an effective representation of the system performance since it integrates two influential factors for embedded system applications, namely the throughput and the power consumption. All evaluations in this section are performed with only one instance of the input and latent defenders. Figure 4.10 (left) illustrates the performance-per-Watt for different hardware platforms. Numbers are normalized by the performance-per-Watt for the *Jetson TK1* platform. As shown, DeepFense implementation on Zynq shows an average of 38× improvement over the *Jetson TK1* and 6.2× improvement over the *Jetson TX2* in the CPU mode. The more expensive UltraScale FPGA performs relatively better with an average improvement of 193× and 31.7× over the *Jetson TK1* and *Jetson TX2* boards, respectively.

The comparisons with GPU platforms are delineated in Figure 4.10 (right). All values

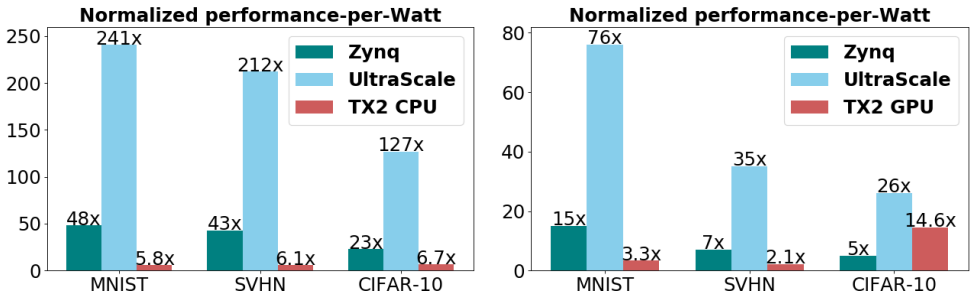


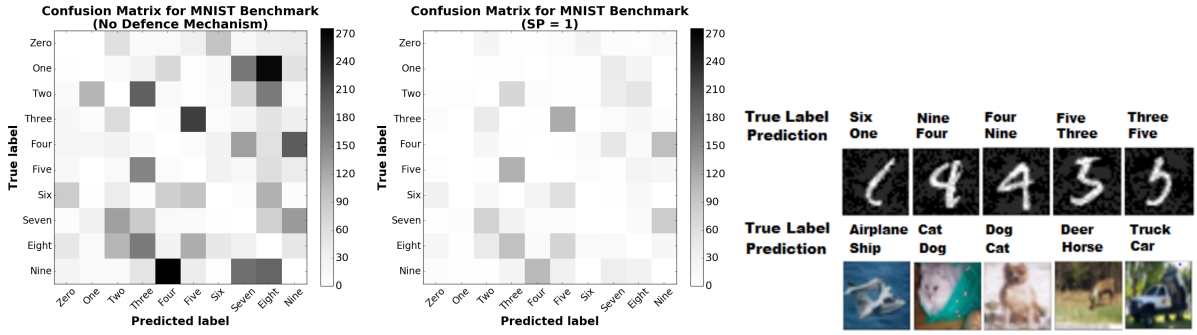
Figure 4.10: Performance-per-Watt comparison with embedded CPU (left) and CPU-GPU (right) platforms. Reported values are normalized by the performance-per-Watt of *Jetson TK1*.

are normalized against the *Jetson TK1* performance-per-Watt in the CPU-GPU mode. The evaluations show an average of 9× and 45.7× improvement over *Jetson TK1* by the Zynq and UltraScale FPGAs, respectively. Comparisons with the *Jetson TX2* demonstrate 2.74× and 41.5× improvement for the Zynq and UltraScale implementations. Note that the UltraScale performs noticeably better than the Zynq FPGA which emphasizes the effect of resource constraints on parallelism and the throughput.

4.5.3 Transferability of Adversarial Samples

Figure 4.11 demonstrates an example of the adversarial confusion matrices for victim neural networks with and without using parallel checkpointing learners. In this example, we set the security parameter to only 1%. As shown, the adversarial sample generated for the victim model **are not transferred** to the checkpointing modules. In fact, the proposed approach can effectively remove/detect adversarial samples by characterizing the rarely explored sub-spaces and looking into the statistical density of data points in the pertinent space.

Note that the remaining adversarial samples that are not detected in this experiment are crafted from legitimate samples that are inherently hard to classify even by a human observer due to the closeness of decision boundaries corresponding to such classes. For instance, in the MNIST application, such adversarial samples mostly belong to class 5 that is misclassified to class 3 or class 4 misclassified as 9. Such misclassifications are indeed the model approximation



(a) (b) (c)

Figure 4.11: Example adversarial confusion matrix (a) without MRR defense mechanism, and (b) with MRR defense and a security parameter of (1%). (c) Example adversarial samples for which accurate detection is hard due to the closeness of corresponding decision boundaries.

error which is well-understood to the statistical nature of the models. As such, a more precise definition of adversarial samples is extremely required to distinguish malicious samples form those that simply lie near the decision boundaries.

4.6 Related Work

In response to the various adversarial attack methodologies proposed in the literature (e.g., [GSS14, MDFF16, CW17b]), several research attempts have been made to design DL strategies that are more robust in the face of adversarial examples. The existing countermeasures can be classified into two categories: **(i)** Supervised strategies which leverage the noise-corrupted inputs [GR14] and/or adversarial examples [SYN15, GSS14, SZS$^+$13] during training of a DL model. These countermeasures are particularly tailored for specific perturbation patterns and can only partially evade adversarial samples generated by other attack scenarios (with different perturbation distributions) from being effective as shown in [GR14]. **(ii)** Unsupervised approaches which aim to address adversarial attacks by smoothing out the gradient space (decision boundaries) [MMK$^+$15, CW17b] or compressing the DL model by removing the nuisance variables [PMW$^+$16]. These set of works have been mainly remained oblivious to the data density

in the latent space and are shown to be vulnerable to adaptive attacks where the adversary knows the defense mechanism [CW16]. More recently, [MC17] proposes an unsupervised manifold projection method called MagNet to reform adversarial samples using auto-encoders. As shown in [CW17a], manifold projection methods including MagNet are not robust to adversarial samples and can approximately increase the required distortion to generate adversarial sample by only 30%. DeepFense methodology (called MRR) is an unsupervised approach that significantly improves the robustness of DL models against best-known adversarial attacks to date. To the best of our knowledge no prior work has addressed resource efficiency or online performance of their defense algorithm.

4.7 Summary

This chapter presents DeepFense, a novel end-to-end framework for online accelerated defense against adversarial samples in the context of deep learning. We introduce modular robust redundancy as a viable unsupervised countermeasure to significantly reduce the risk of integrity attacks. To ensure applicability to various deep learning tasks and FPGA platforms, DeepFense provides an API that takes as input the high-level description of a deep neural network together with the specifications of the underlying hardware platform. Using a software-hardware-algorithm co-design approach, our automated customization tool optimizes the defense layout to maximize model reliability (safety) while complying with the hardware and/or user constraints. Our extensive evaluations corroborate the effectiveness and practicality of DeepFense framework.

4.8 Acknowledgements

This chapter, in part, has been published at (i) the Proceedings of 2018 International Conference On Computer Aided Design (ICCAD) and appeared as: Bita Darvish Rouhani,

Mohammad Samragh, Mojan Javaheripi, Tara Javidid, and Farinaz Koushanfar “DeepFense: Online Accelerated Defense Against Adversarial Deep Learning”, and (ii) IEEE Security and Privacy (S&P) Magazine 2018 as ita Darvish Rouhani, Mohammad Samragh, Tara Javidid, and Farinaz Koushanfar “Safe Machine Learning and Defeating Adversarial Attacks”. The dissertation author was the primary author of this material.

Chapter 5

DeepSigns: Watermarking Deep Neural Networks

Deep Learning (DL) models have created a paradigm shift in our ability to comprehend raw data in various important fields, ranging from intelligence warfare and healthcare to autonomous transportation and automated manufacturing. A practical concern, in the rush to adopt DL models as a service, is protecting the models against Intellectual Property (IP) infringement. DL models are commonly built by allocating substantial computational resources that process vast amounts of proprietary training data. The resulting models are therefore considered to be an IP of the model builder and need to be protected to preserve the owner's competitive advantage. We propose DeepSigns, the first end-to-end IP protection framework that enables developers to systematically insert digital watermarks in the pertinent DL model before distributing the model. DeepSigns is encapsulated as a high-level wrapper that can be leveraged within common deep learning frameworks including TensorFlow, PyTorch, and Theano. The libraries in DeepSigns work by dynamically learning the probability density function (pdf) of activation maps obtained in different layers of a DL model. DeepSigns uses the low probabilistic regions within a deep neural network to gradually embed the owner's signature (watermark) while minimally affecting

the overall accuracy and/or training overhead. DeepSigns can demonstrably withstand various removal and transformation attacks, including model pruning, model fine-tuning, and watermark overwriting. We evaluate DeepSigns performance on a wide variety of DL architectures including Wide Residual Networks, Convolution Neural Networks, and Multi-Layer Perceptrons with MNIST, CIFAR10, and ImageNet data. Our extensive evaluations corroborate DeepSigns' effectiveness and applicability. Our highly-optimized accompanying API further facilitates training watermarked neural networks with an extra overhead as low as 2.2%.

5.1 Introduction

Deep Neural Networks (DNNs) and other deep learning variants have revolutionized vision, speech, and natural language processing and are being applied in many other critical fields [LBH15, DY14, GBC16, RGC15]. Training a highly accurate DNN requires: (i) Having access to a massive collection of mostly proprietary labeled data that furnishes comprehensive coverage of potential scenarios in the target application. (ii) Allocating substantial computing resources to fine-tune the underlying model topology (i.e., type and number of hidden layers), hyper-parameters (i.e., learning rate, batch size, etc.), and DNN weights to obtain the most accurate model. Given the costly process of designing/training, DNNs are typically considered to be the intellectual property of the model builder.

Model protection against IP infringement is particularly important for DNNs to preserve the competitive advantage of the owner and ensure the receipt of continuous query requests by clients if the model is deployed in the cloud as a service. Embedding digital watermarks into DNNs is a key enabler for reliable technology transfer. Digital watermarks have been immensely leveraged over the past decade to protect the ownership of multimedia and video content, as well as functional artifacts such as digital integrated circuits [FK04, HK99, QP07, CKLS97, Lu04]. Extension of watermarking techniques to DNNs, however, is still in its infancy to enable reliable

model distribution. Moreover, adding digital watermarks further presses the already constrained memory for DNN training/execution. As such, efficient resource management to minimize the overhead of watermarking is a standing challenge.

Authors in [UNSS17, NUSS18] propose an N-bit ($N > 1$) watermarking approach for embedding the IP information in the static content (i.e., weight matrices) of convolutional neural networks. Although this work provides a significant leap as the first attempt to watermark DNNs, it poses (at least) two limitations as we discuss in Section 5.5: (i) It incurs a bounded watermarking capacity due to the use of the static content of DNNs (weights) as opposed to using dynamic content (activations). The weights of a neural network are static during the execution phase, regardless of the data passing through the model. The activations, however, are dynamic and both data- and model-dependent. We argue that using activations (instead of weights) provides more flexibility for watermarking. (ii) It is not robust against attacks such as overwriting the original embedded watermark by a third party. As such, the original watermark can be removed by an adversary that is aware of the watermarking method used by the model owner.

More recent studies in [MPT17, ABC$^+$18] propose 1-bit watermarking methodologies for deep learning models. These approaches are built upon model boundary modification and the use of random adversarial samples that lie near decision boundaries. Adversarial samples are known to be statistically unstable, meaning that adversarial samples crafted for a model are not necessarily misclassified by another network [GMP$^+$17, RSJK18b]. Therefore, even though the proposed approaches in [MPT17, ABC$^+$18] yield a high watermark detection rate (true positive rate), they are also too sensitive to hyper-parameter tuning and usually lead to a high false alarm rate. Note that false ownership proofs jeopardize the integrity of the proposed watermarking methodology and render the use of watermarks for IP protection ineffective.

We propose DeepSigns, the first efficient resource management framework that empowers coherent integration of robust digital watermarks into DNNs. DeepSigns is devised based on an Algorithm/Hardware/Software co-design. As illustrated in Figure 5.1, DeepSigns inserts

the watermark information in the host DNN and outputs a protected, functionality-preserved model to prevent the adversary from pirating the ownership of the model. Unlike prior works that directly embed the watermark information in the static content (weights) of the pertinent model, DeepSigns works by embedding an arbitrary N-bit ($N \geq 1$) string into the probability density function (pdf) of the activation maps in various layers of a deep neural network. Our proposed watermarking methodology is simultaneously *data-* and *model-dependent*, meaning that the watermark information is embedded in the dynamic content of the DNN and can only be triggered by passing specific input data to the model. We further provide a comprehensive set of quantitative and qualitative metrics that shall be evaluated to corroborate the effectiveness of current and pending DNN watermarking methodologies that will be proposed in future.

Figure 5.1: DeepSigns is a systematic solution to protect the intellectual property of DNNs.

We provide a highly-optimized implementation of DeepSigns' watermarking methodology which can be readily used as a high-level wrapper within contemporary DL frameworks. Our solution, in turn, reduces the non-recurring engineering cost and enables model designers to incorporate specific Watermark (WM) information during the training of a neural network with minimal changes in their source code and overall training overhead. Extensive evaluation across various DNN topologies confirms DeepSigns' applicability in different settings without requiring excessive hyper-parameter tuning to avoid false alarms or accuracy drop. By introducing DeepSigns, we make the following contributions:

- Enabling effective IP protection for DNNs. A novel watermarking methodology is introduced to encode the pdf of activation maps and effectively trace the IP ownership.

DeepSigns is significantly more robust against removal and transformation attacks compared to prior DNN watermarking methodologies.

- Characterizing the requirements for an effective watermark embedding in the context of deep learning. We provide a comprehensive set of metrics that enables quantitative and qualitative comparison of current and pending DNN-specific IP protection methods.

- Devising a careful resource management and accompanying API. A user-friendly API is devised to minimize the non-recurring engineering cost and facilitate the adoption of DeepSigns within common DL frameworks including TensorFlow, Pytorch, and Theano.

- Analyzing various DNN topologies. Through extensive proof-of-concept evaluations, we investigate the effectiveness of DeepSigns and corroborate the necessity of such solution to protect the IP of an arbitrary DNN and establish the ownership of the model designer.

5.2 DeepSigns Global Flow

Figure 5.2 shows the global flow of DeepSigns framework. DeepSigns consists of two main phases: watermark embedding and watermark extraction. The watermarked DNN can be employed as a service by third-party users either in a white-box or a black-box setting depending on whether the model internals are transparent to the public or not. DeepSigns is the first DNN watermarking framework that is applicable to both white-box and black-box security models.

Watermark Embedding. DeepSigns takes the DNN architecture and the owner-specific watermark signature as its input. The WM signature is a set of arbitrary binary strings that should be generated such that each bit is independently and identically distributed (i.i.d.). DeepSigns, then, outputs a trained DNN that carries the pertinent watermark signature in selected layers along with a set of corresponding WM keys. The WM keys are later used to trigger the embedded WM information during the extraction phase. The WM embedding process is performed in two

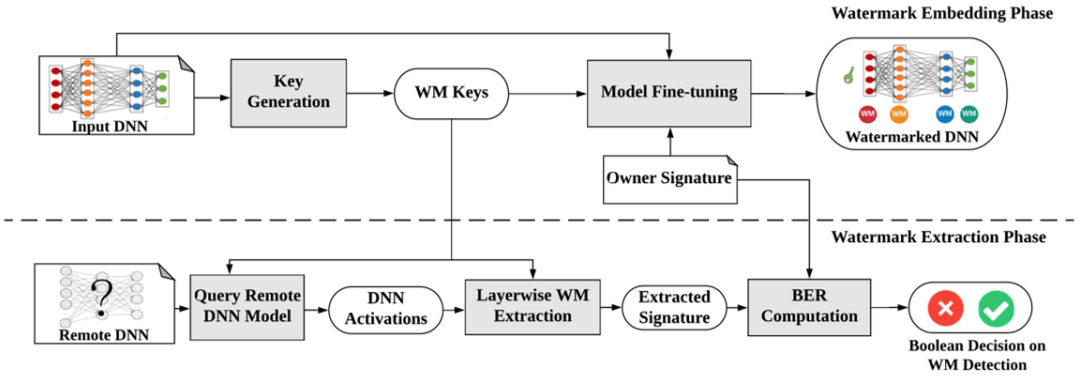


Figure 5.2: Global flow of DeepSigns framework. DeepSigns embeds the owner-specific WM signature in the pdf distribution of activation maps acquired at various DNN layers. A specific set of WM keys are generated to extract the embedded watermarks. The WM keys triggering the ingrained WM are then used for watermark extraction and detection of IP infringement.

steps. First, a set of WM keys are generated as secure parameters for WM embedding. Then, the underlying DNN is trained (fine-tuned) such that the owner-specific WM signature is encoded in the pdf distribution of activation maps obtained at different DNN layers. Note that WM embedding is a one-time task performed by the owner before model distribution. Details of each step are discussed in Section 5.3. The trained watermarked DNN can be securely distributed by the model owner. Model distribution is a common approach in the machine learning field (e.g., the Model Zoo by Caffe Developers, and Alexa Skills by Amazon). Note that even though models are voluntarily shared, it is important to protect the IP of the original owner.

Watermark Extraction. To verify the IP of a remote DNN and detect potential IP infringement, the model owner first needs to query the remote DNN service with WM keys generated in the WM embedding phase and obtain the corresponding activation maps. DeepSigns then extracts the WM signature from the pdf distribution of the acquired activation maps. It next computes the Bit Error Rate (BER) between the extracted signature in each layer and the corresponding true signature. If the BER at any layer is zero, it implies that the owner's IP is deployed in the remote DNN service. Details of each WM extraction step are discussed in Section 5.3.

5.2.1 DNN Watermarking Prerequisites

There are a set of minimal requirements that should be addressed to design a robust digital watermark. Table 5.1 details the prerequisites for an effective DNN watermarking methodology. In addition to previously suggested requirements in [UNSS17, MPT17], we believe reliability and integrity are two other major factors that need to be considered when designing a practical DNN watermarking methodology. *Reliability* is important because the embedded watermark should be accurately extracted using the pertinent keys; the model owner is thereby able to detect any misuse of her model with a high probability. *Integrity* ensures that the IP infringement detection policy yields a minimal number of false alarms, meaning that there is a very low chance of falsely proving the ownership of the model used by a third party. DeepSigns satisfies all the requirements listed in Table 5.1 as shown in Section 5.4.

Table 5.1: Requirements for an effective watermarking of deep neural networks.

Requirements	Description
Fidelity	Accuracy of the target neural network shall not be degraded as a result of watermark embedding.
Reliability	Watermark extraction shall yield minimal false negatives; WM shall be effectively detected using the pertinent keys.
Robustness	Embedded watermark shall be resilient against model modifications such as pruning, fine-tuning, or WM overwriting.
Integrity	Watermark extraction shall yield minimal false alarms (a.k.a., false positives); the watermarked model should be uniquely identified using the pertinent keys.
Capacity	Watermarking methodology shall be capable of embedding a large amount of information in the target DNN.
Efficiency	Communication and computational overhead of watermark embedding and extraction shall be negligible.
Security	The watermark shall be secure against brute-force attacks and leave no tangible footprints in the target neural network; thus, an unauthorized party cannot detect/remove the presence of a watermark.

Potential Attack Scenarios. To validate the robustness of a potential DL watermarking approach, one should evaluate the performance of the proposed methodology against (at least) three types of contemporary attacks: (i) *Model fine-tuning*. This type of attack involves re-training of the original model to alter the model parameters and find a new local minimum while preserving the accuracy. (ii) *Model pruning*. Model pruning is a commonly used approach for efficient execution of neural networks, particularly on embedded devices. We consider model pruning as

another attack approach that might affect the watermark extraction/detection. (iii) *Watermark overwriting*. A third-party user who is aware of the methodology used for DNN watermarking (but is not aware of the owner's private WM keys) may try to embed a new watermark in the model and overwrite the original one. An overwriting attack aims to insert an additional watermark in the model and render the original watermark unreadable. A watermarking methodology should be robust against fine-tuning, pruning, and overwriting for effective IP protection.

5.3 DeepSigns Methodology

Deep learning models possess non-convex loss surfaces with many local minima that are likely to yield an accuracy very close to another [CHM$^+$15, RMK17a]. DeepSigns takes advantage of this phenomenon that there is not a unique solution for modern non-convex optimization problems to embed the WM information in the pdf distribution of activation maps in a DNN. DeepSigns proposes different approaches for watermarking the intermediate layers (*Section 5.3.1*) and the output layer (*Section 5.3.2*) of a DNN model. This is due to the fact that the activation of an intermediate layer is continuous-valued while the one of the output layer is discrete-valued in classification tasks that comprise a large percentage of DL applications.

Furthermore, we analyze the computation and communication overhead of DeepSigns framework in Section 5.3.3. We devise an efficient memory management library to minimize the pertinent WM embedding overhead. DeepSigns accompanying library is compatible with the current popular DL solutions and provides a user-friendly API in Python that supports GPU acceleration. To illustrate the integrability of DeepSigns, we demonstrate how to use DeepSigns as a wrapper to embed/extract WM in Sections 5.3.1 and 5.3.2.

5.3.1 Watermarking Intermediate Layers

DeepSigns embeds the WM information within a transformation of specific activations. We consider a Gaussian Mixture Model (GMM) as the prior pdf to characterize the data distribution at a given layer where the WM shall be inserted. The rationale behind this choice is the observation that GMM provides a reasonable approximation of the activation distribution obtained in hidden layers [IS15, PNB15, LTA16]. DeepSigns is rather generic and is not restricted to the GMM distribution; GMM can be replaced with other prior distributions based on the application. In the following, we discuss the details of WM embedding and extraction procedures for intermediate (hidden) layers.

Watermark Embedding (Hidden Layers)

Algorithm 3 outlines the process of WM embedding in a hidden layer. It consists of two main steps eah of which are explained in the following.

Algorithm 3 Watermark embedding for one hidden layer.

INPUT: Topology of the unmarked DNN $(\mathcal{T})$**; Training data** $(\{X^{train}, Y^{train}\})$**; Owner-specific watermark signature b; Total number of Gaussian classes** (S)**; Length of watermark vector for each selected distribution** (N)**; Dimensionality of the activation map in the embedded layer** (M)**; and Embedding strength hyper-parameters** $(\lambda_1,\ \lambda_2)$**.**

OUTPUT: Watermarked DNN $(\mathcal{T}^*)$**; WM keys.**

1: Key Generation: ❶

$T \leftarrow Select\_Gaussian\_Classes\ ([1,\ S])$

$X^{key} \leftarrow Subset\_Training\_Data\ (T,\ \{X^{train}, Y^{train}\})$

$A^{M\times N} \leftarrow Generate\_Secret\_Matrix\ (M,\ N)$

2: Model Fine-tuning: Two additive loss functions are incorporated to train the DNN: ❷

$$L = \underbrace{cross\_entropy}_{loss_0} + \lambda_1 loss_1 + \lambda_2 loss_2.$$

Return: Marked DNN $\mathcal{T}^*$, WM keys $(s, X^{key}, A^{M\times N})$.

---

❶ **Key Generation.** For a given hidden layer l, DeepSigns first selects one (or more) random indices between 1 and S with no replacement: Each index corresponds to one of the Gaussian distributions in the target mixture model that contains a total of S Gaussians. In our experiments, we set the value S equal to the number of classes in the target application. The mean values of the selected distributions are next used to carry the WM signature. DeepSigns then decides on a subset of the input training data belonging to the selected Gaussian classes (X^{key}). This subset is later used by the model owner to trigger the embedded WM signature within a hidden layer as discussed in Section 5.3.1. We use a subset of 1% of the training data for this purpose.

DeepSigns also generates a projection matrix A to encrypt the selected centers into the binary space. This projection is critical to accurately measure the difference between the embedded WM and the owner-defined binary signature during training. The projection is performed as:

$$\begin{aligned} G_{\sigma}^{s\times N} &= Sigmoid\ (\mu_l^{s\times M} \cdot A^{M\times N}), \\ \tilde{b}^{s\times N} &= Hard\_Thresholding\ (G_{\sigma}^{s\times N},\ 0.5). \end{aligned} \tag{5.1}$$

Here, M is the size of the feature space in the selected hidden layer, s is the number of Gaussian distributions chosen to carry the WM information, and N indicates the desired length of the watermark embedded at the mean value of s selected Gaussian distribution ($\mu_l^{s\times M}$). In our experiments, we use a standard normal distribution $\mathcal{N}(0,1)$ to generate the WM projection matrix (A). Using i.i.d. samples drawn from a normal distribution ensures that each bit of the binary string is embedded into all the features associated with the selected centers. The σ notation in Eq. (5.1) is used as a subscript to indicate the deployment of the Sigmoid function. The output of Sigmoid has a value between 0 and 1. Given the random nature of the binary string, we decide to set the threshold in Eq. (5.1) to 0.5, which is the expected value of Sigmoid function. The *Hard_Thresholding* function in Eq. (5.1) maps the values in G_{σ} that are greater than 0.5 to ones and the values less than 0.5 to zeros. The WM keys comprise the selected Gaussian classes s, trigger keys X^{key} and projection matrix A.

❷ **Model Fine-tuning.** To effectively encode the WM information, two additional constraints need to be considered during DNN training: (i) Selected activations shall be isolated from other activations. (ii) The distance between the owner-specific WM signature and the transformation of isolated activations shall be minimal. We design and incorporate two specific loss functions to address each of these two constraints ($loss_1$ and $loss_2$ in Step 2 of Algorithm 3).

To address the activation isolation constraint, we design an additive loss term that penalizes the activation distribution when activations are entangled and hard to separate. Adhering to our GMM assumption, we add the following term to the cross-entropy loss function conventionally used for DNN training:

$$\lambda_1 (\underbrace{\Sigma_{i \in T} \|\mu_l^i - f_l^i(x,\theta)\|_2^2 \; - \; \Sigma_{i \in T, j \notin T} \|\mu_l^i - \mu_l^j\|_2^2}_{\textbf{loss}_1}). \tag{5.2}$$

Here, λ_1 is a trade-off hyper-parameter that specifies the contribution of the additive loss term, θ is the DNN parameters (weights), $f_l^i(x,\theta)$ is the activation map corresponding to the input sample x belonging to class i at the l^{th} layer, T is the set of s target Gaussian classes selected to carry the WM information, and μ_l^i denotes the mean value of the i^{th} Gaussian distribution at layer l. The additive loss function ($loss_1$) aims to minimize the spreading (variance) of each GMM class used for watermarking (the first term in $loss_1$) while maximizing the distance between the activation centers belonging to different Gaussian classes (the second term in $loss_1$). This loss function, in turn, helps to augment data features so that they better fit a GMM distribution. The mean values μ_l^i and intermediate activations $f_l^i(x,\theta)$ in Eq. (5.2) are *trainable variables* that are iteratively fine-tuned using back-propagation.

To ensure the transformed selected Gaussian centers are as close to the desired WM information as possible, we design the second additive loss term that characterizes the distance between the owner-defined signature and the embedded watermark (see Eq.(5.3)). As such,

DeepSigns adds the following term to the overall loss function used during DNN training:

$$-\lambda_2 \underbrace{\sum_{j=1}^{N}\sum_{k=1}^{s}(b^{kj}\ln(G_{\sigma}^{kj})+(1-b^{kj})\ln(1-G_{\sigma}^{kj}))}_{\textbf{loss}_2}. \tag{5.3}$$

Here, the variable λ_2 is a hyper-parameter that determines the contribution of $loss_2$ during DNN training. The $loss_2$ function resembles a binary cross-entropy loss where the true bit b^{kj} is determined by the owner-defined WM signature and the prediction probability G_{σ}^{kj} is the Sigmoid of the projected Gaussian centers as outlined in Eq. (5.1). The process of computing the vector G_{σ} is differentiable. Thereby, for a selected set of projection matrix (A) and binary WM signature (b), the selected centers (Gaussian mean values) can be adjusted via back-propagation such that the Hamming distance between the binarized projected center $\tilde{b}$ and the actual WM signature b is minimized. In our experiments, we set λ_1 and λ_2 to 0.01.

Watermark Extraction (Hidden Layers)

As the inverse process of watermark embedding, watermark extraction is implemented using the two particular constraints we design in Section 5.3.1. To extract watermark information from intermediate (hidden) layers, the model owner must follow three main steps. **(i)** Acquiring activation maps corresponding to the selected trigger keys X^{key} by submitting a set of queries to the remote DL service provider. **(ii)** Computing the statistical mean value of the activation maps obtained in Step I. The acquired mean values are adopted as an approximation of the Gaussian centers that are supposed to carry the watermark information. The computed mean values together with the owner's private projection matrix A are used to extract the pertinent WM following the procedure in Eq. (5.1). **(iii)** Measuring the bit error rate between the owner's signature and the extracted WM from Step II. Note that if the watermarked DNN in question is not deployed in the remote service, a random WM is extracted which yields a very high BER.

DeepSigns Memory Management and Wrapper

DeepSigns minimizes the required data movement to ensure maximal data reuse and a minimal overhead caused by watermark embedding. To do so, we integrate the computation of additive loss terms to the DNN tensor graph so that the gradients with respect to the GMM centers are computed during regular back-propagation and all computation for WM embedding is performed homogeneously on GPU. Modeling the watermarking graph separately significantly slows than the DNN training process since the activation maps need to be completely dumped from the original DNN graph during the forward pass to compute the WM loss and update the parameters of the WM graph. This approach, in turn, further presses the already constrained memory. Our homogeneous solution reuses the activation values within the original graph with minimal memory overhead.

DeepSigns library provides a customized activity regularizer $WM\_activity\_regularizer$ that computes $loss_1$ and $loss_2$ and returns the total regularized loss value described in Algorithm 1. To extract the WM from the embedded layers, our accompanying library is equipped with functions called $get\_activation$ and $extract\_WM\_from\_activations$ to implement the process outlined in Section 5.3.1. Figure 5.3 shows the prototype of functions used for watermark embedding/extraction in the intermediate layers. The notations are consistent with the definitions in Section 5.3.1 and 5.3.1. DeepSigns' customized library supports acceleration on GPU platforms. Our provided wrapper can be readily integrated within well-known DL frameworks including TensorFlow, Pytorch, and Theano.

5.3.2 Watermarking Output Layer

The final prediction of a DNN shall closely match the ground-truth labels to have the maximum possible accuracy. As such, instead of directly regularizing activations of the output layer, we decide to adjust the tails of decision boundaries to add a statistical bias as a 1-bit

```
import DeepSigns
from DeepSigns import subsample_training_data
from DeepSigns import WM_activity_regularizer
from DeepSigns import get_activations
from DeepSigns import extract_WM_from_activations
from DeepSigns import compute_BER
from utils import create_marked_model

## create WM trigger keys
X^key = subsample_training_data(T, {X^train , Y^train })
## instantiate customized WM activity regularizer
WM_reg = WM_activity_regularizer(λ1, λ2, b, A)
model = create_marked_model(WM_reg , model topology)
## embed WM by standard training of the marked model
model.fit(X^train , Y^train)

## extract WM from the activation maps and compute BER
μ_l^{s*M} = get_activations(model, X^key, l, T)
b̃ = extract_WM_from_activations(μ_l^{s*M}, A)
BER = compute_BER(b̃, b)
```

Figure 5.3: DeepSigns library usage and resource management for WM embedding and extracting in hidden layers.

watermark. The WM key is designed as a set of random (key image, key label) pairs that are used to retrain the DNN. To verify the ownership of a remote DNN, we devise a statistical hypothesis testing method that compares the prediction of the queried DNN with WM key labels. A high consistency implies the existence of the owner's watermark.

Watermarking the output layer is a post-processing step that is performed once the DNN model is converged or the intermediate layers are watermarked as discussed in Section 5.3.1. In the following, we detail the workflow of WM embedding and extraction operating on the output layer. We then provide a corresponding example using DeepSigns' library.

Watermark Embedding (Output Layer)

The WM keys corresponding to the output layer should be carefully crafted such that they reside in low-density regions of the target DNN in order to ensure minimal accuracy drop. To do so, DeepSigns profiles the pdf distribution of different layers in the target DNN. The acquired pdf, in turn, gives us an insight into both the regions that are thoroughly occupied by the training data

and the regions that are only covered by a few inputs, which we refer to as rarely explored regions. Figure 5.4 illustrates a simple example of two clustered activation distributions spreading in a two-dimensional subspace. The procedure of WM embedding in the output layer is summarized in Algorithm 4. In the following, we explicitly discuss each of the steps outlined in Algorithm 4.

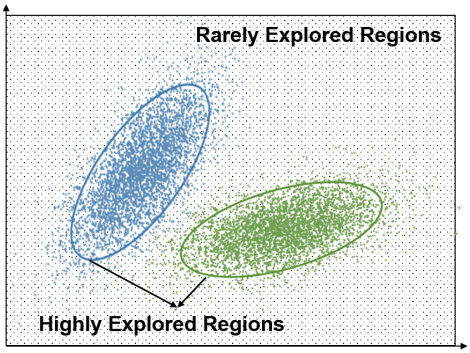


Figure 5.4: Due to the high dimensionality of DNNs and limited access to labeled data (the blue and green dots in the figure), there are regions that are rarely explored. DeepSigns exploits this mainly unused regions for WM embedding while minimally affecting the accuracy.

1 Key Generation 1. DeepSigns generates a set of K unique random input samples to be used as the watermarking keys in step 2. Each random sample is passed through the pre-trained neural network to make sure its intermediate activation lies within the rarely explored regions characterized by the learned pdf. If the number of data activations within an ε-ball of the activation corresponding to the random sample is fewer than a threshold, we accept that sample belongs to the rarely explored regions. Otherwise, a new random sample is generated to replace the previous sample. A corresponding random ground-truth vector is generated and assigned to each input key (e.g., in a classification task, each random input is associated with a randomly selected label).

We set the initial key size to be larger than the owner's desired value $K^{'} > K$ and generate the input keys accordingly. The target model is then fine-tuned (Step 2) using a mixture of the generated keys and a subset of original training data. After fine-tuning, only the keys that are simultaneously correctly classified by the marked model and incorrectly predicted by the unmarked model are appropriate candidates that satisfy both a high detection rate and a low false

Algorithm 4 Watermark embedding for DL output layer.

INPUT: Topology of the partially-marked or unmarked DNN $(\mathcal{T})$ **and its pdf distribution** (pdf)**; Training data** $(\{X^{train}, Y^{train}\})$**; Key size** (K)**.**

OUTPUT: Watermarked DNN $(\mathcal{T}^*)$**; Crafted WM keys** $(\{X^{key}, Y^{key}\})$**.**

1: Key Generation 1: 1

Set the initial key size $K^{'} > K$ (e.g., $K^{'} = 20 \times K$).

$\{X^{key'}, Y^{key'}\} \leftarrow Generate\_Key\_Pairs(K^{'}, pdf)$

2: Model Fine-tuning: 2

$\mathcal{T}^* \leftarrow Train(\mathcal{T}, \{X^{key'}, Y^{key'}\}, \{X^{train}, Y^{train}\})$

3: Key Generation 2: 3

$Y_{\mathcal{T}^*}^{pred'} \leftarrow Predict(\mathcal{T}^*, X^{key'})$

$Y_{\mathcal{T}}^{pred'} \leftarrow Predict(\mathcal{T}, X^{key'})$

$I_{\mathcal{T}^*} \leftarrow Find\_Match\_Index\ (Y^{key},\ Y_{\mathcal{T}^*}^{key'})$

$I_{\mathcal{T}} \leftarrow Find\_Mismatch\_Index\ (Y^{key},\ Y_{\mathcal{T}^{key'}})$

$I^{key'} \leftarrow Find\_Intersection\ (\ I_{\mathcal{T}},\ I_{\mathcal{T}^*})$

$\{X^{key}, Y^{key}\} \leftarrow Select\ (\ \{X^{key'}, Y^{key'}\},\ I^{key'},\ K)$

Return: Marked DNN $\mathcal{T}^*$; WM key $\{X^{key}, Y^{key}\}$.

positive. We set $K^{'} = 20 \times K$ where K is the desired key length selected by the model owner.

2 **Model Fine-tuning.** The pre-trained DNN is fine-tuned with the selected random keys in Step 1. The model shall be retrained such that the neural network has exact predictions (e.g., an accuracy greater than 99%) for chosen key samples. In our experiments, we use the same optimizer setting used for training the original neural network, except that the learning rate is reduced by a factor of 10 to prevent accuracy drop in the prediction of legitimate input data. Note that the model is already converged to a local minimum.

3 **Key Generation 2.** Once the model is fine-tuned in step 2, we first find out the indices of initial WM keys that are correctly classified by the watermarked model. Next, we identify the

indices of WM keys that are not classified correctly by the original DNN before fine-tuning in Step 2. The common keys between these two sets are proper candidates to trigger the embedded WM. A random subset of candidate WM keys is then selected according to the key size (K) defined by the model owner. In the global flow (Figure 5.2), we merge the two key generation steps into one module for simplicity.

Watermark Extraction (Output Layer)

To verify the presence of a watermark in the output layer, DeepSigns performs statistical hypothesis testing on the responses obtained from the remote DNN service. To do so, DeepSigns undergoes three main steps: **(i)** Submitting queries to the remote DNN service provider and acquiring the output labels corresponding to the randomly selected WM keys (X^{key}) as discussed in Section 5.3.2. **(ii)** Computing the number of mismatches between model predictions and WM ground-truth labels. **(iii)** Thresholding the number of mismatches to derive the final decision. If the number of mismatches is less than the threshold, it means the model used by the remote service provider possesses a high similarity to the watermarked DNN.

When the two models are exact duplicates, the number of mismatches will be zero. In practice, the target DNN might be slightly modified by third-party users in both malicious or non-malicious ways. Examples of such modifications are model fine-tuning, pruning, or WM overwriting. As such, the threshold used for WM detection should be greater than zero to withstand DNN modifications. When queried by a random key image, the prediction of the model has probabilities $P(y^{pred} = j) = p_j$ for which $\sum_{j=1}^{C} p_j = 1$ and C is the number of classes in the pertinent application. Since the corresponding key labels are uniformly randomly generated, the

probability that the key sample is correctly classified per generated key label is:

$$\begin{aligned} P(y^{pred} = y^{key}) &= \sum_{j=1}^{C} P(y^{pred} = j, y^{key} = j) \\ &= \sum_{j=1}^{C} P(y^{pred} = j) \times \sum_{j=1}^{C} P(y^{key} = j) \\ &= \frac{1}{C} \times \sum_{j=1}^{C} P(y^{pred} = j) = \frac{1}{C}. \end{aligned}$$

due to the independence between $P(y^{pred})$ and $P(y^{key})$. Note that Eq. (5.4) also holds when the class sizes are unbalanced. Therefore, the probability of an arbitrary DNN to make at least n_k correct decision per owner's private keys is:

$$P(N_k > n_k | O) = 1 - \sum_{k=0}^{n_k} \binom{K}{k} (\tfrac{1}{C})^{K-k} (1 - \tfrac{1}{C})^k. \tag{5.4}$$

Here O is the DNN oracle used in the remote service, N_k is a random variable indicating the number of matched predictions of the two models compared against one another, K is the input key length according to Section 5.3.2. The decision for WM detection in the output layer is made by comparing $P(N_k > n_k | O)$ with an owner-specified probability threshold (p). In our experiments, we use a decision threshold of 0.999.

DeepSigns Wrapper and Memory Management

Figure 5.5 illustrates how to use DeepSigns library for WM embedding and detection in the output layer. DeepSigns automatically designs a set of robust WM key pairs (X^{key}, Y^{key}) for the model owner using the function *key_generation*. The generated WM keys are embedded in the target DNN by fine-tuning. The existence of the WM is determined from the response of the queried model to the key set. Note that DeepSigns can provide various levels of security by setting the hyper-parameters K(key length) and decision policy threshold. A larger key-length, in

turn, induces a higher overhead (see Section 5.3.3). The model owner can explore the trade-off between security and overhead by adopting different hyper-parameters.

DeepSigns wrapper provides a custom layer working for efficient resource management during the identification of the rarely explored regions (Step 1 in Algorithm 4). To do so, we first apply Principal Component Analysis (PCA) on the activation maps acquired by passing the training data through the converged DNN. The computed Eigenvectors are then used to transform the high dimensional activation maps into a lower dimensional subspace. We encode the PCA transformation as a dense layer inserted after the second-to-last layer of the original DNN graph so that the data projection is performed with minimal data movement. The weights of the new dense layer are obtained from the Eigenvectors of the PCA of the pertinent activations. For each randomly generated sample, the density of the activations within an ε Euclidean distance of that sample is then computed. If this density is greater than a threshold that sample will be rejected.

```
import DeepSigns
from DeepSigns import key_generation
from DeepSigns import count_response_mismatch
from DeepSigns import compute_mismatch_threshold
from utils import create_model

## generate WM key pairs
model = create_model(model topology)
model.load_weights('baseline_weights')
(X^key, Y^key) = key_generation(model, K, X^train, pdf)
X^retrain = np.vstack(X^key, X^train)
Y^retrain = np.vstack(Y^key, Y^train)
## embed WM by finetuning the model with the WM key
model.fit(X^retrain, Y^retrain)

## query model with key set to detect WM
Y^pred = model.predict(X^key)
m = count_response_mismatch(Y^pred , Y^key)
θ = compute_mismatch_threshold(p, K, C)
WM_detected = 1 if m < θ else 0
```

Figure 5.5: Using DeepSigns library for WM embedding and extraction in the output layer.

5.3.3 DeepSigns Watermark Extraction Overhead

Here, we analyze the computation and communication overhead of WM extraction. The WM embedding is a one-time offline process that incurs a negligible overhead. We empirically discuss the WM embedding overhead in Section 5.4.

Watermarking Intermediate Layers. From the viewpoint of remote DNN service provider, the computation cost is equivalent to the cost of one forward pass in the DNN model with no extra overhead. From the model owner's viewpoint, the computation cost is divided into two terms. The first term is proportional to $O(M)$ to compute the statistical mean in Step 2 outlined in Section 5.3.1. Here, M denotes the feature space size in the target hidden layer. The second term corresponds to the computation of matrix multiplication in Eq. (5.1), which incurs a cost of $O(MN)$. The communication cost is equivalent to the input key length multiplied by input feature size plus the size of intermediate layers (M) to submit the pertinent queries and obtain the intermediate activations, respectively.

Watermarking Output Layer. For the remote DNN service provider, the computation cost is equal to the cost of one forward pass through the underlying DNN. For the model owner, the computation cost is the cost of performing a simple counting to measure the number of mismatches between the responses of the remote service provider and the WM key labels. In this case, the communication cost is equal to the key length multiplied by the sum of the input feature size and one to submit the queries and read back the predicted labels.

5.4 Evaluations

We evaluate the performance of DeepSigns framework on various datasets including MNIST [LCB98], CIFAR10 [KH09] and ImageNet [DDS$^+$09], with four different neural network architectures. Table 5.2 summarizes DNN topologies used in each benchmark. In Table 5.2, K denotes the key size for watermarking the output layer and N is the length of the owner-specific

Table 5.2: Benchmark neural network architectures. Here, $64C3(1)$ indicates a convolutional layer with 64 output channels and 3×3 filters applied with a stride of 1, $MP2(1)$ denotes a max-pooling layer over regions of size 2×2 and stride of 1, and $512FC$ is a fully-connected layer with 512 output neurons. ReLU is used as the activation function in all benchmarks.

Dataset	Baseline Accuracy	Accuracy of Marked Model		DL Model Type	DL Model Architecture
MNIST	98.54%	K = 20	N = 4	MLP	784-512FC-512FC-10FC
		98.59%	98.13%		
CIFAR10	78.47%	K = 20	N = 4	CNN	3*32*32-32C3(1)-32C3(1)-MP2(1) -64C3(1)-64C3(1)-MP2(1)-512FC-10FC
		81.46%	80.70%		
CIFAR10	91.42%	K = 20	N = 128	WideResNet	Please refer to [ZK16].
		91.48%	92.02%		
ImageNet	74.72%	K = 20	–	ResNet50	Please refer to [DDS+09].
		74.21%	–		

WM signature used for watermarking the hidden layers. In our experiments, we use the second-to-last layer or the output layer for watermarking. DeepSigns library is generic and also supports WM embedding in multiple layers if larger capacity is desired. In the rest of this section, we explicitly evaluate DeepSigns' performance with respect to each requirement listed in Table 5.1. As empirically demonstrated, DeepSigns is effective and applicable across various data sets and DNN architectures.

5.4.1 Fidelity

DeepSigns preserves the DNN overall accuracy after watermark embedding. The accuracy of the target neural network shall not be degraded after embedding the WM information. Table 5.2 summarizes the baseline DNN accuracy (Column 2) and the accuracy of marked models (Column 3 and 4) after WM embedding. As demonstrated, DeepSigns respects the fidelity requirement by simultaneously optimizing for the accuracy of the underlying model (e.g., cross-entropy loss function), as well as the additive WM-specific loss functions as discussed in Section 5.3. In some cases (e.g. WideResNet benchmark), we even observe a slight accuracy improvement compared to the baseline. This improvement is mainly due to the fact that the additive loss functions outlined in Eq. (5.2) and (5.3) act as a form of a regularizer during DNN training. Regularization, in turn, helps the model to mitigate over-fitting by inducing a small amount of noise to DNNs [GBC16].

5.4.2 Reliability and Robustness

DeepSigns enables robust DNN watermarking and reliably extracts the embedded WM for ownership verification. We evaluate the robustness of DeepSigns against three state-of-the-art removal attacks as discussed in Section 5.2.1. These attacks include DNN parameter pruning [HPTD15, HMD15, RMK16], model fine-tuning [SZ14, TSG$^+$16], and watermark overwriting [UNSS17, JDJ01].

Parameter Pruning. We use the pruning approach proposed in [HPTD15] to sparsify the weights in the target watermarked DNN. To prune a specific layer, we first set $\alpha\%$ of the parameters that possess the smallest weight values to zero. The model is then sparsely fine-tuned using cross-entropy loss function to compensate for the accuracy drop caused by pruning. Figure 5.6 demonstrates the impact of pruning on WM extraction/detection in the output and hidden layers. The length of the watermark signature and the key size used in each benchmark are listed in Table 5.2. As shown, DeepSigns can tolerate up to 90% parameter pruning for MNIST benchmark, and up to 99% parameter pruning for the CIFAR10, and ImageNet benchmarks. As illustrated in Figure 5.6, in cases where DNN pruning yields a substantial BER value, the sparse model suffers from a large accuracy loss. As such, one cannot remove DeepSigns' embedded watermark by excessive pruning and attain a comparable accuracy with the baseline.

Model Fine-tuning. Fine-tuning is another form of transformation attack that a third-party user might use to remove the WM information. To perform this type of attack, one needs to retrain the target model using the original training data with the conventional cross-entropy loss function (excluding $loss_1$ and $loss_2$). Table 5.3 summarizes the impact of fine-tuning on the watermark detection rate across all benchmarks. There is a trade-off between model accuracy and the success rate of watermark removal. If a third party tries to disrupt the pdf of activation maps that carry the WM information by fine-tuning the underlying DNN with a high learning rate, she will face a large degradation in DNN accuracy. We use the same learning rate as the one in the final stage of DL training to perform model fine-tuning attack. As shown in Table 5.3, DeepSigns

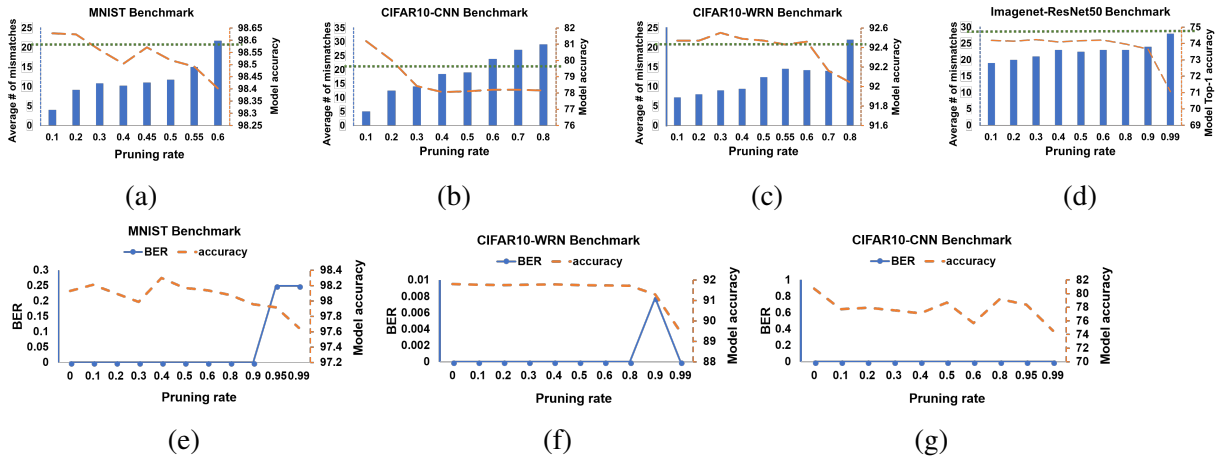


Figure 5.6: Robustness against parameter pruning. The experiments in the first row are related to watermarking the output layer. The horizontal green dotted line is the mismatch threshold obtained from Eq. (5.4). The orange dashed lines show the final accuracy for each pruning rate. Experiments in the second row correspond to watermarking the second-to-last layer.

Table 5.3: DeepSigns is robust against model fine-tuning attack. The reported BER and the detection rate value are averaged over 10 different runs. A value of 1 in the last row of the table indicates that the embedded WM is successfully detected, whereas a value of 0 indicates a false negative. For fine-tuning attacks, the WM-specific loss terms proposed in Section 5.3 are removed from the loss function and the model is retrained using the final learning rate of the original DL model. After fine-tuning, the DL model will converge to another local minimum that is not necessarily a better one (in terms of accuracy) for some benchmarks.

Metrics	Intermediate Layer Watermarking									Output Layer Watermarking										
	MNIST-MLP			CIFAR10-CNN			CIFAR10-WRN			MNIST-MLP			CIFAR10-CNN			CIFAR10-WRN			Imagenet-ResNet50	
Number of epochs	50	100	200	50	100	200	50	100	200	50	100	200	50	100	200	50	100	200	10	20
Accuracy	98.21	98.20	98.18	70.11	62.74	59.86	91.79	91.74	91.8	98.57	98.57	98.59	98.61	98.63	98.60	87.65	89.74	88.35	74.06	74.14
BER	0	0	0	0	0	0	0	0	0	-	-	-	-	-	-	-	-	-	-	-
Detection success	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

can successfully detect the embedded WM even after fine-tuning the DNN for various number of epochs. We set the number of fine-tuning epochs to 10 and 20 epochs for ImageNet benchmark and 50, 100, 200 epochs for other benchmarks. The reason for this selection is that training the ImageNet benchmark from scratch takes 90 epochs whereas other benchmarks take around 300 epochs to be trained.

Watermark Overwriting. Assuming the attacker is aware of the watermarking methodology, she may attempt to corrupt the original watermark by embedding a new WM. In practice, the attacker does not have any knowledge about the location of the watermarked layers. In our experiments, we consider the worst-case scenario in which the attacker knows where the WM

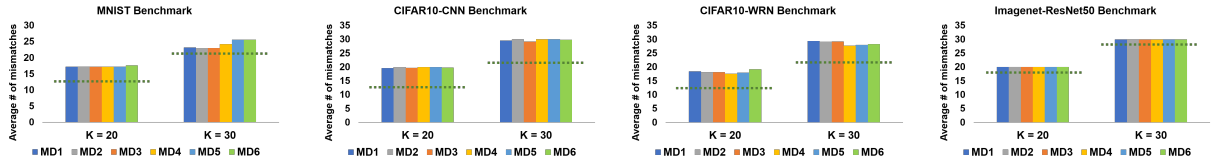
is embedded but does not know the WM key. To perform the overwriting attack, the attacker follows the procedure in Section 5.3 to embed a new WM signature with her new keys. Table 5.4 summarizes the results of WM overwriting for all four benchmarks in which the output layer is watermarked. As shown, DeepSigns is robust against the overwriting attack and can successfully detect the original embedded WM in the overwritten DNN. The decision thresholds shown in Table 5.4 for different key lengths are computed based on Eq. (5.4) as discussed in Section 5.3.2. A BER of zero is also observed in the overwritten DNNs where the WM is embedded in the second-to-last layer. This further confirms the DeepSigns' reliability and robustness against malicious attacks.

Table 5.4: DeepSigns is robust against overwriting attack. The reported number of mismatches is the average value of 10 runs for the same model using different WM key sets.

	Average # of mismatches		**Decision threshold**		**Detection**
	K = 20	K = 30	K = 20	K = 30	**success**
MNIST	8.3	15.4	13	21	**1**
CIFAR10-CNN	9.2	16.7	13	21	**1**
CIFAR10-WRN	8.5	10.2	13	21	**1**
Imagenet-ResNet50	10.5	18.5	19	29	**1**

5.4.3 Integrity

DeepSigns avoids claiming the ownership of unmarked DNNs and yields low false positive rates. Let us assume the third-party user does not share her model internals and only returns the output prediction for each submitted query. In this case, it is critical to incur the minimal number of false alarms in WM extraction. Figure 5.7 illustrates DeepSigns integrity-related performance. In this experiment, six different unmarked models (with the same and different architectures) are queried by DeepSigns. As corroborated, DeepSigns satisfies the integrity criterion and has no false positives across different benchmarks, which means the ownership of unmarked DNNs will not be falsely proved. We use the same set of hyper-parameters (e.g., detection policy threshold) across all the benchmarks with no particular fine-tuning.

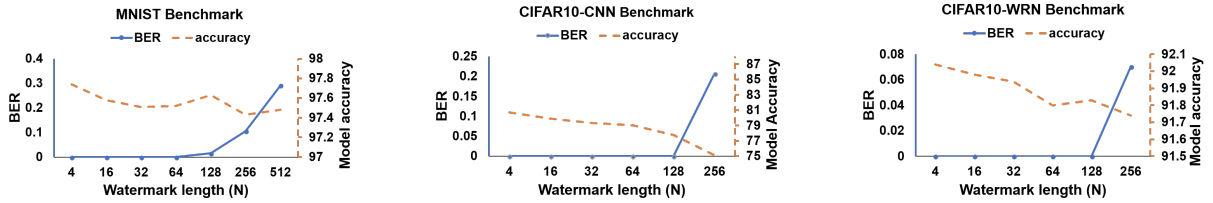


(a) (b) (c) (d)

Figure 5.7: Integrity analysis of DeepSigns framework. The green dotted horizontal lines indicate the detection threshold for various WM lengths. The first three models (MD 1-3) are neural networks with the same topology but different parameters compared with the watermarked model. The last three models (MD 4-6) are neural networks with different topologies ([SDBR14], [LH15], [ZK16]).

5.4.4 Capacity

DeepSigns has a high watermarking capacity. We embed WM signatures with different lengths in a single DNN layer to assess the capacity of DeepSigns watermarking methodology. As shown in Figure 5.8, DeepSigns allows up to 64 bits WM embedding for MNIST and up to 128 bits WM embedding in the second-to-last layer of CIFAR10-CNN, and CIFAR10-WRN benchmarks. Note that there is a trade-off between the capacity and accuracy which can be used by the IP owner to embed a larger watermark in her DNN model if desired. For IP protection purposes, capacity is not an impediment criterion as long as there is sufficient capacity to contain the necessary WM information. Nevertheless, we include this property in Table 5.1 to have a comprehensive list of requirements.



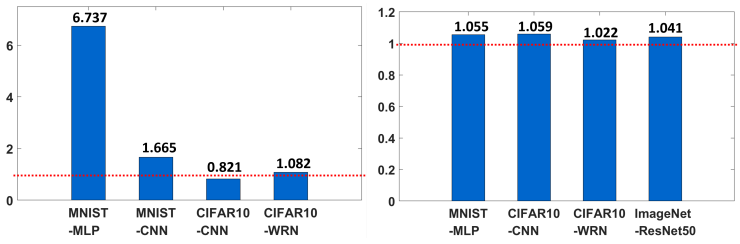
(a) (b) (c)

Figure 5.8: There is a trade-off between the length of the WM signature (capacity) and the bit error rate of WM extraction. As the number of the embedded bits (N) increases, the test accuracy of the watermarked model decreases and the BER of WM extraction increases. This trend indicates that embedding excessive amount of WM information impairs fidelity and reliability.

5.4.5 Efficiency

DeepSigns incurs a negligible overhead. The WM extraction overhead is discussed in Section 5.3.3. Here, we analyze the overhead incurred by the WM embedding phase. The computation overhead to embed a watermark using DeepSigns is a function of the DNN topology (i.e., the number of parameters/weights in the pertinent DNN), the key length (K), and the length of watermark signature N. DeepSigns has no communication overhead for WM embedding since the embedding process is performed locally by the model owner.

To quantify the computation overhead for embedding a watermark, we measure the normalized training time of the baseline model without WM and the marked model with WM to reach the same accuracy level. The results are shown in Figure 5.9, where the x-axis denotes various benchmarks and the y-axis denotes the runtime ratio of training a DNN model with/without a WM. As shown, DeepSigns incurs a reasonable overhead for WM embedding (normalized runtime overhead around 1), suggesting a high efficiency. The overhead of embedding a watermark in the hidden layer of MNIST-MLP benchmark is higher than others since this benchmark is so compact with a relatively small rarely-explored region. The low-dimensionality of this model, in turn, makes it harder to reach the same accuracy while adding noise to the system by incorporating the WM-specific regularization.

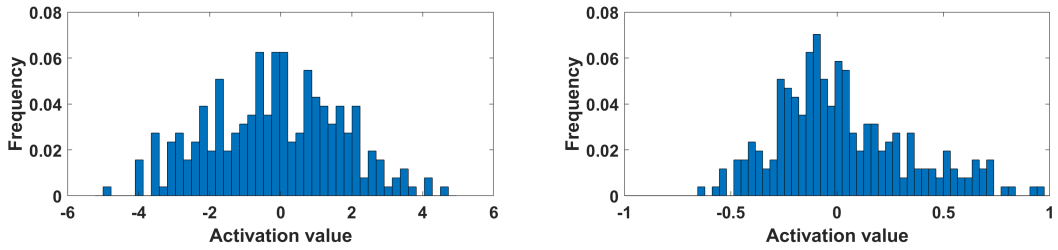


(a) (b)

Figure 5.9: Normalized WM embedding runtime overhead in (a) the second-to-last layer and (b) the output layer. The desired runtime ratio is denoted by the red dashed line in the figure.

5.4.6 Security

DeepSigns leaves an imperceptible footprint in the watermarked DNN and is secure against brute-force attacks. As mentioned in Table 5.1, embedding a watermark should not leave noticeable changes in the pdf distribution spanned by the target DNN. DeepSigns satisfies the security requirement by preserving the intrinsic distribution of weights/activations. Figure 5.10 shows the distribution of activations in WM embedded layer of a marked DNN and the ones in the same layer of the unmarked DNN for CIFAR10-WRN benchmark. The range of activations is not deterministic in different models and cannot be used by malicious users to detect WM existence.



(a) (b)

Figure 5.10: Distribution of the activation maps for (a) marked and (b) unmarked models. DeepSigns preserves the intrinsic distribution while securely embedding the WM information.

DeepSigns is secure against brute-force attacks. If the WM is embedded in the output layer, the searching space of an adversary to find the exact WM keys designed by the model owner is $O(d^{I\times K})$ where d is the size of input data, I is the number of possible elements in the input domain (e.g., 256 for image data as each pixel can get an integer value in the range of $[0-255]$), and K is the WM key size. Note that each WM key is generated based on i.i.d. random distribution. If the WM is embedded in hidden layers, the search space for the attacker to find WM keys is $O(\prod_{l\in L}\binom{S}{s_l}R^{M_l\times N_l})$. Here, L is the set of hidden layers used for watermarking, s_l, M_l and N_l denote the number of WM-related Gaussian centers, the dimension of activation maps, and the length of the WM signature in the l^{th} layer, respectively. R is the number of values in the domain used for creating projection matrix A ($A \in \mathbb{R}$ so R is infinity).

5.5 Comparison With Prior Works

DeepSigns provides the first automated resource management tool and the accompanying API for efficient DNN watermarking. Unlike prior works, DeepSigns uses dynamic statistics of DNN models for watermark embedding by encoding the WM information in the pdf distribution of activation maps. Note that the weights of a DNN model are static during the inference phase while activation maps are dynamic features that are dependent on the input keys and the DNN parameters simultaneously. Our dynamic watermarking approach is significantly more robust against potential attacks compared with the prior art in which static weights are explored for watermarking (Section 5.5.1). None of the previous output layer watermarking frameworks consider overwriting attacks in their experiments. As such, no quantitative comparison is feasible. Nevertheless, we demonstrate DeepSigns robustness against overwriting attacks in Table 5.4. In the rest of this section, we explicitly compare DeepSigns performance against three state-of-the-art DNN watermarking frameworks existing in the literature.

5.5.1 Intermediate Layer Watermarking

The works in [UNSS17, NUSS18] encode the WM information in the weights of convolution layers, as opposed to the activation maps proposed by DeepSigns. As shown in [UNSS17], watermarking the weights is not robust against overwriting attacks. Table 5.5 provides a side-by-side robustness comparison between our approach and these prior works for different dimensionality ratio (defined as the ratio of the length of the attacker's WM signature to the size of weights or activations). As demonstrated, DeepSigns' dynamic data- and model-aware approach is significantly more robust compared to prior art [UNSS17, NUSS18]. As for its robustness against pruning attacks, our approach is tolerant of higher pruning rates. Consider the CIFAR10-WRN benchmark as an example, DeepSigns is robust up to 80% pruning rate, whereas the works in [UNSS17, NUSS18] are only robust up to 65% pruning rate.

Table 5.5: Robustness comparison against overwriting attacks. The WM information embedded by DeepSigns can withstand overwriting attacks for a wide of range of $\frac{N}{M}$ ratio. In this experiment, we use the CIFAR10-WRN since this benchmark is the only model evaluated by [UNSS17, NUSS18].

N to M Ratio	Bit Error Rate (BER)	
	Uchida et.al [UNSS17, NUSS18]	DeepSigns
1	0.309	0
2	0.41	0
3	0.511	0
4	0.527	0

5.5.2 Output Layer Watermarking

There are two prior works targeting watermarking the output layer [ABC$^+$18, MPT17]. Even though the works by [ABC$^+$18, MPT17] provide a high WM detection rate (reliability), they do not address the integrity requirement, meaning that these approaches can lead to a high false positive rate in practice. Table 5.6 provides a side-by-side comparison between DeepSigns and the work in [MPT17]. As shown, DeepSigns has a significantly lower probability of falsely claiming the ownership of a remote DNN.

Table 5.6: Integrity comparison between DeepSigns and the prior work (PW) [MPT17]. For each benchmark, the WM key size is set to $K = 20$ and 10 different sets of WM keys are generated to query six unmarked DNNs. The average false positive rates of querying each DNN model are reported.

False Positive Rate	Model 1		Model 2		Model 3		Model 4		Model 5		Model 6	
WM Method	PW	Ours	PW	Ours	PW	Ours	PW	Ours	PW	Ours	PW	Ours
MNIST	0.5	0.0	0.3	0.0	0	0.0	0.1	0.0	1.0	0.0	1.0	0.0
CIFAR10-CNN	0.0	0.0	0.1	0.0	0.1	0.0	0.0	0.0	1.0	0.0	0.0	0.0
CIFAR10-WRN	0.5	0.0	0.8	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

The work in [ABC$^+$18] uses the accuracy on the test set as the decision policy to detect WM information. It is well-known that there is no unique solution to DL problems [CHM$^+$15, DY14, GBC16]. In other words, there are various models with even different topologies that yield approximately the same test accuracy for a particular data application. Besides high false positive rates, another drawback of using test accuracy for WM detection is the high overhead of communication and computation [ABC$^+$18]; therefore, their watermarking approach suffers from

a low efficiency. DeepSigns uses a small WM key size ($K = 20$) to trigger the WM information, whereas a typical test set in DL problems can be two to three orders of magnitude larger.

5.6 Summary

Deep learning is facilitating breakthroughs in various fields such as medical, aerospace, business, and education. While the commercialization of DNNs is so popular, efficient IP protection for pre-trained, ready-to-deploy models has been a standing challenge. It is timely to devise a systematic solution for DNN IP protection. DeepSigns takes the first step towards this goal by providing an efficient end-to-end framework that enables WM embedding in activations of a neural network while minimally affecting the overall runtime and resource utilization. Our accompanying API paves the way for model designers to achieve a reliable technology transfer.

5.7 Acknowledgements

This chapter, in part, has been published at arXiv preprint arXiv:1804.00750, 2018 as: Bita Darvish Rouhani, Huili Chen, and Farinaz Koushanfar "Deepsigns: A Generic Watermarking Framework for IP Protection of Deep Learning Models". The dissertation author was the primary author of this material.

Chapter 6

DeepSecure: Scalable Provably-Secure Deep Learning

This chapter presents DeepSecure, the an scalable and provably secure deep learning framework that is built upon automated design, efficient logic synthesis, and optimization methodologies. DeepSecure targets scenarios in which neither of the involved parties including the cloud servers that hold the DL model parameters or the delegating clients who own the data is willing to reveal their information. Our framework is the first to empower *accurate* and *scalable* DL analysis of data generated by distributed clients without sacrificing the security to maintain efficiency. The secure DL computation in DeepSecure is performed using *Yao's Garbled Circuit* (GC) protocol. We devise GC-optimized realization of various components used in DL. Our optimized implementation achieves up to *58-fold* higher throughput per sample compared with the best prior solution. In addition to the optimized GC realization, we introduce a set of novel low-overhead pre-processing techniques which further reduce the GC overall runtime in the context of DL. Our extensive evaluations demonstrate up to *two orders-of-magnitude* additional runtime improvement achieved as a result of our pre-processing methodology.

6.1 Introduction

Technology leaders such as Microsoft, Google, IBM, and Facebook are devoting millions of dollars to devise accurate deep learning models for various artificial intelligence and data inference applications ranging from social networks and transportations to environmental monitoring and health care [Jon14, Kir16, Efr17]. The applicability of DL models, however, is hindered in settings where the risk of data leakage raises serious privacy concerns. Examples of such applications include cloud-based applications where clients hold sensitive private information, e.g., medical records, financial data, or location.

To employ DL models in a privacy-preserving setting, it is imperative to devise computing frameworks in which neither of the involving parties is required to reveal their information. Several research works have been developed to address privacy-preserving computing for DL networks, e.g., [GBDL$^+$16, MZ17]. The existing solutions, however, either: (i) rely on the modification of DL layers (such as non-linear activation functions) to efficiently compute the specific cryptographic protocols. For instance, authors in [GBDL$^+$16, MZ17] have suggested the use of polynomial-based Homomorphic encryption to make the client's data oblivious to the server. Their approach requires changing the non-linear activation functions to some polynomial approximation (e.g., square) during training. Such modification, in turn, can reduce the ultimate accuracy of the model and poses a trade-off between the model accuracy and execution cost of the privacy-preserving protocol. Or (ii) fall in the two-server settings in which data owners distribute their private data among two non-colluding servers to perform a particular DL inference. The two non-colluding server assumption is not ideal as it requires the existence of a trusted third-party which is not always an option in real-world settings.

We introduce DeepSecure, the first provably-secure framework for *scalable* DL-based analysis of data collected by distributed clients. DeepSecure enables applying the state-of-the-art DL models on sensitive data without sacrificing the accuracy to obtain security. Consistent

with the literature, we assume an *honest-but-curious* adversary model for a generic case where both DL parameters and input data must be kept private. DeepSecure proposes the use of Yao's Garbled Circuit (GC) protocol to securely perform DL execution. We empirically corroborate that our framework is significantly efficient for scenarios in which the number of samples collected by each distributed client is less than 2600 samples; the clients send the data to the server for processing with the least possible delay. Our approach is well-suited for streaming settings where clients need to dynamically analyze their data as it is collected over time without having to queue the samples to meet a certain batch size. In DeepSecure framework, we further provide mechanisms based on secret sharing to securely delegate GC computations to a third party for constrained embedded devices.

The function to be securely evaluated in GC should be represented as a list of Boolean logic gates (a.k.a., *netlist*). We generate the netlists required for deep learning using logic synthesis tools with GC-optimized custom libraries as suggested in [SHS+15]. The computation and communication workload of GC protocol in DeepSecure framework is explicitly governed by the number of neurons in the target DL network and input data size. We further introduce a set of novel low-overhead pre-processing techniques to reduce data and DL network footprint without sacrificing neither the accuracy nor the data confidentiality. Our pre-processing approach is developed based on two sets of innovations: (i) transformation of input data to an ensemble of lower-dimensional subspaces, and (ii) avoiding the execution of neutral (inactive) neurons by leveraging the sparsity structure inherent in DL models. The explicit contributions of this work are as follows:

- Proposing DeepSecure, the first provably-secure framework that simultaneously enables *accurate* and *scalable* privacy-preserving DL execution for distributed clients.
- Devising new custom libraries to generate *GC-optimized netlists* for required DL network computations. Our custom library is built upon automated design, efficient logic synthesis, and optimization methodologies.

- Incepting the idea of *data and DL network* pre-processing for secure function evaluation. Our approach leverages the fine- and coarse-grained data and DL network parallelism to avoid unnecessary computation/communication in the execution of Yao's GC protocol.
- Providing proof-of-concept evaluations of various visual, audio, and smart-sensing benchmarks. Our evaluations corroborate DeepSecure's scalability and practicability for distributed users compared with the HE-based solution.

6.2 DeepSecure Framework

Figure 6.1 demonstrates the overall flow of DeepSecure framework. DeepSecure consists of two main components to securely perform data inference in the context of deep learning: (i) GC-optimized execution of the target DL model (Section 6.2.1), and (ii) data and DL network transformation (Section 6.2.2).

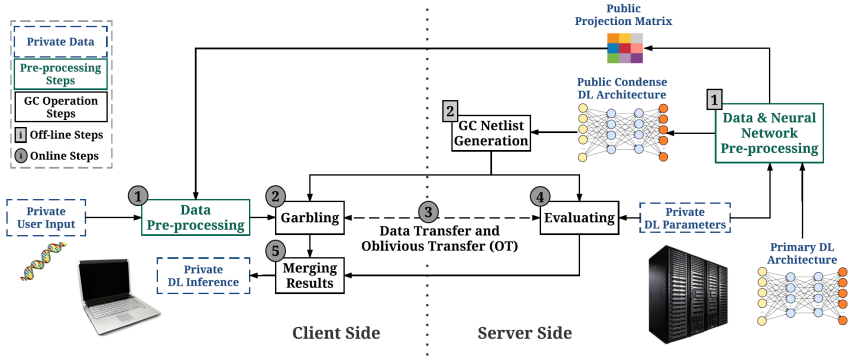


Figure 6.1: Global flow of DeepSecure framework including both off-line (indexed by rectangular icons) and online (indexed by oval icons) steps. The operations shown in the left hand side of the figure are executed by the client (Alice) while the operations on the right hand side are performed by the server (Bob).

6.2.1 DeepSecure GC Core Structure

In a DL-based cloud service, a cloud server (Bob) holds the DL model parameters trained for a particular application, and a delegated client (Alice) owns a data sample for which she wants

to securely find the corresponding classified label (a.k.a., inference label).

DeepSecure enables computing the pertinent data inference label in a provably-secure setting while keeping both the DL model's parameters and data sample private. To perform a particular data inference, the netlist of the publicly known DL architecture[1] should be generated prior to the execution of the GC protocol. The execution of the GC protocol involves four main steps: (i) the client (data owner) garbles the Boolean circuit of the DL architecture. (ii) The client sends the computed garbled tables from the first step to the cloud server along with her input wire labels. Both client and the cloud server then engage in a 1-out-of-2 Oblivious Transfer [NP05] protocol to obliviously transfer the wire labels associated with cloud server's inputs. (iii) The cloud server evaluates the garbled circuit and computes the corresponding encrypted data inference. (iv) The encrypted result is sent back to the client to be decrypted using the garbled keys so that the true inference label is revealed.

GC Communication and Computation Overhead. Table 6.1 details the computation and communication cost for execution of a fully-connected DNN. A similar setup applies to the CNN models in which a set of convolutions are performed per layer. The total communication needed between client and the server is proportional to the number of non-XOR gates since only the garbled tables for non-XOR gates need to be transferred. In DeepSecure framework, garbling/evaluating each non-XOR and XOR gate requires 164 and 62 CPU clock cycles (*clks*) on average, respectively.

As shown in Table 6.1, the computation and communication overhead of DL execution using GC protocol is explicitly governed by the number of neurons (units) per DL layer. As such, we suggest a set of data and DL network transformation as an arbitrarily pre-processing step to reduce the computation and communication overhead of GC protocol for DL inference.

[1] DL architecture refers to the number and type of layers and not the values of the pertinent private DL parameters.

Table 6.1: GC Computation and Communication Costs for realization of a DNN model.

Computation and Communication Costs
$T_{comp} = \beta_{mult} \sum_{l=1}^{n_l-1} n^{(l)} n^{(l+1)} + \beta_{add} \sum_{l=2}^{n_l} n^{(l)} + \beta_{act} \sum_{l=2}^{n_l} n^{(l)}$ $\beta_{opr} = \frac{N_{opr}^{XOR} \times C_{opr}^{XOR} + N_{opr}^{Non\_XOR} \times C_{opr}^{Non\_XOR}}{f_{CPU}}$ n_l: *total number of DL layers* β_{mult}: *computational cost of a multiply operation in GC protocol* β_{add}: *computational cost of an add operation in GC protocol* β_{act}: *computational cost of a non-linearity operation in GC protocol* N^{XOR}: *number of XOR gates* N^{Non_XOR}: *number of non_XOR gates* C^{XOR}: *garbling/evaluating cost of a XOR gate* C^{Non_XOR}: *garbling/evaluating cost of a Non_XOR gate* f_{CPU}: *CPU clock frequency*
$T_{comm} = \frac{\alpha_{mult} \sum_{l=1}^{n_l-1} n^{(l)} n^{(l+1)} + \alpha_{add} \sum_{l=2}^{n_l} n^{(l)} + \alpha_{act} \sum_{l=2}^{n_l} n^{(l)}}{BW_{net}}$ $\alpha_{opr} = N_{opr}^{Non\_XOR} \times 2 \times N_{bits}$ BW_{net}: *operational communication bandwidth* α_{mult}: *communication cost of a multiply operation in GC protocol* α_{add}: *communication cost of an add operation in GC protocol* α_{act}: *communication cost of a non-linearity operation in GC protocol* N_{bits}: *GC security parameter*

6.2.2 Data and DL Network Pre-processing

DeepSecure pre-processing consists of two main steps: (i) data projection (Section 6.2.2), and (ii) DL network distillation (Section 6.2.2).

Data Projection

The input layer size of a neural network is conventionally dictated by the feature space size of the input data samples. Many complex modern data matrices that are not inherently low-rank can be modeled by a composition of multiple lower-rank subspaces. This type of composition of high-dimensional dense (non-sparse but structured) data as an ensemble of lower dimensional subspaces has been used earlier by data scientists and engineers to facilitate knowledge extraction or achieve resource efficiency [RMK17b, RMK16, RMK17d]. DeepSecure, for the first time,

introduces, implements, and automates the idea of data pre-processing as a way to achieve performance optimization for GC execution of a DL model.

As we empirically demonstrate in Section 6.3, the main bottleneck in the execution of GC protocol is the communication overhead between the server (Bob) and client (Alice). Therefore, we focus our pre-processing optimization to minimize the GC communication workload (T_{comm}) customized to the application data and DL model (see Table 6.1 for the characterization of communication overhead in accordance with the data and DL network size). In order to perform data pre-processing in DeepSecure framework, the server needs to re-train its private DL parameters according to the following objective function:

$$\begin{aligned} \underset{D,\ \tilde{DL}_{param}}{Minimize}\ (T_{comm})\ \ s.t.,\ \ & ||A - DC||_F \leq \varepsilon ||A||_F \\ & \delta(\tilde{DL}_{param}) \leq \delta(DL_{param}) \\ & l \leq m, \end{aligned} \tag{6.1}$$

where $A_{m\times n}$ is the raw input training data (owned by the server) that we want to factorize into a *dictionary matrix* $D_{m\times l}$ and a low-dimensional *data embedding* $C_{l\times n}$ that is used to re-train the DL model. Here, $\delta(.)$ is the partial validation error corresponding to the pertinent DL parameters. We use DL_{param} to denote the initial DL parameters acquired by training the target DL model using raw data features (A). Whereas, $\tilde{DL}_{param}$ indicates the updated parameters after re-training the underlying DL model using the projected data embedding C. $||\cdot||_F$ denotes the Frobenius norm and ε is an intermediate approximation error that casts the rank of the input data. We leveraged the streaming-based data sketching methodology suggested in [RMK17b] to solve the optimization objective outlined in Eq. (6.1).

Once the DL model is re-trained using the projected data embedding C, we define a projection matrix ($W = DD^{+}$)[2] which is publicly released to be used by the clients during DL

[2] D^{+} indicates the pseudo-inverse of the matrix D.

execution phase. As we will discuss in Section 6.2.4, W does not reveal any information regarding the training data nor the DL parameters. We emphasize that re-training of conventional DL model is a *one-time off-line* process performed by the server. As such, the data pre-processing overhead during GC execution only involves a matrix-vector multiplication, $Y_i = WX_i$, that is performed prior to garbling on the client side. Here, X_i indicates the raw data owned by the client (Alice), W denotes the projection matrix, and Y_i is the projected data in the space spanned by columns of W.

DL Network Pre-processing

Recent theoretical and empirical advances in DL has demonstrated the importance of sparsity in training DL models, e.g., [HPTD15]. Sparsity inducing techniques such as rectifying non-linearities and $\mathcal{L}_1$ penalty are key techniques used to boost the accuracy in training neural networks with millions of parameters.

To eliminate the unnecessary garbling/evaluation of non-contributing neurons in a DL model, we suggest pruning the underlying DL network prior to netlist generation for the GC protocol. In our DL network pre-processing, the connections with a weight below a certain threshold are removed from the network. The condensed network is re-trained as suggested in [HPTD15] to retrieve the accuracy of the initial DL model. DeepSecure network pre-processing step is a one-time off-line process performed by the server. Our approach is built upon the fact that DL models are usually over-parameterized and can be effectively represented with a sparse structure without a noticeable drop in the accuracy.

Note that using conventional GC protocol, it is not feasible to skip the multiplication/addition in evaluating a particular neuron in a DL model. Our network pre-processing cuts out the non-contributing connections/neurons per layer of a DL network. It, in turn, enables using the sparse nature of DL models to significantly reduce the computation and communication workload of executing the GC protocol. The off-line step 1 indicated in Figure 6.1 corresponds to both data pre-processing and neural network pruning.

6.2.3 GC-Optimized Circuit Components Library

As we explained earlier, the GC protocol requires the function of interest being represented as a Boolean circuit. Following the Free-XOR optimization [KS08], the XOR gates are almost free of cost and the garbled table needs to be generated and transferred only for the non-XOR gates. Therefore, to optimize the computation and communication costs, one needs to minimize the number of non-XOR gates. We leverage the industrial synthesis tool to optimize the resulting netlist by setting the area overhead for XOR gates to zero and for all the other non-XOR gates to one. We design optimized fundamental blocks e.g., multiplexer (MUX), comparator (CMP), adder (ADD), and multiplier (MULT) such that they incur the least possible non-XOR gates. We add our optimized blocks to the library of the synthesis tool.

Our custom synthesis library includes the GC-optimized realization of all the necessary computation modules in a neural network. The results of our synthesized circuits in terms of the number of XOR and non-XOR are summarized in Table 6.2. To compute DL non-linearity functions, we evaluate COordinate Rotation DIgital Computer (CORDIC) circuit. Each iteration of computing CORDIC improves the final accuracy by one bit. For instance, in order to achieve 12 bit accuracy, we need to iteratively evaluate the circuit 12 times. To operate CORDIC in hyperbolic mode, one needs to evaluate iterations $(3 \times i + 1)$ twice, which in turn, results in an overall 14 iterations per instance computation. CORDIC outputs Cosine-Hyperbolic (Cosh) and Sine-Hyperbolic (Sinh). The synthesized result provided in table 6.2 shows the total number of gates for 14 iterations of evaluation plus one DIV operation for $Tanh_{\mathrm{CORDIC}}$ with an additional two ADD operations for Sigmoid computation.

Softmax is a monotonically increasing function. Therefore, applying this function to a given input vector does not change the index of the maximum value (inference label index). As such, we use optimized CMP and MUX blocks to implement Softmax in DeepSecure framework.

Table 6.2: Number of XOR and non-XOR gates for each operation of DL networks.

Name	#XOR	#Non-XOR
$\text{Tanh}_{\text{CORDIC}}$	8415	3900
$\text{Sigmoid}_{\text{CORDIC}}$	8447	3932
$\text{Softmax}_{\text{n}}$	$(n-1)\cdot 48$	$(n-1)\cdot 32$
ReLu	30	15
ADD	16	16
MULT	381	212
DIV	545	361
$\text{A}_{1\times m}\cdot\text{B}_{m\times n}$	$397\cdot m\cdot n-16\cdot n$	$228\cdot m\cdot n-16\cdot n$

6.2.4 Security Proof

In this section, we provide a comprehensive security proof of DeepSecure in the Honest-but-Curious (HbC) adversary model. Our core secure function evaluation engine is the GC protocol. GC is proven to be secure in HbC adversary model [BHR12]; thereby, any input from either client or server(s) to GC will be kept private during the protocol execution. The Garbled circuit optimization techniques (Section 6.2.3) that we leverage in DeepSecure to reduce the number of non-XOR gates of circuit components do not affect the security of our framework. This is because the security of the GC protocol is independent of the topology of the underlying Boolean circuit. As such, we only need to provide the security proof of the three modifications that are performed outside of the GC protocol:

(i) Data Pre-processing. In step one of off-line processing depicted in Figure 6.1, the projection matrix (W) is computed and released publicly. Projection Matrix (W) reveals nothing but the subspace of dictionary matrix (D) from which the matrix D cannot be reconstructed. **Proof:** Let $D_{m\times l}=U_{m\times r}\times\Sigma_{r\times r}\times V^{T}_{r\times l}$ denote the Singular Value Decomposition (SVD) of the dictionary

matrix D, where r is the rank of D ($r = min(m,l)$). As such,

$$\begin{aligned} W &= DD^{+} \\ &= D(D^T D)^{-1} D^T \\ &= U\Sigma V^T (V\Sigma^T U^T U\Sigma V^T)^{-1} V\Sigma^T U^T \\ &= U\Sigma (V^T V^{T^{-1}})(\Sigma^T I_r \Sigma)^{-1}(V^{-1}V)\Sigma^T U^T \\ &= U\Sigma I_r (\Sigma^T \Sigma)^{-1} I_r \Sigma^T U^T \\ &= UU^T, \end{aligned} \tag{6.2}$$

where I_r indicates the identity matrix of size $r \times r$. As such, the projection matrix W does not reveal any information regarding the actual values of the dictionary matrix D but the subspace spanned by its column space U (a.k.a., *left-singular vectors*). Note that for a given set of left-singular vectors U, there exist infinite possible data matrices that reside in the same column space. As such, the dictionary matrix D cannot be reconstructed without having access to corresponding right-singular vectors V and the singular value set Σ. If revealing the subspace spanned by U is not tolerable by the server, this pre-processing step can be skipped.

(ii) DL Network Pre-processing. In this pre-processing stage, the inherent sparsity of the neural network is utilized to produce a new model which requires less computation. In order to avoid garbling/evaluating unnecessary components in the Boolean circuit, the server needs to modify the netlist used in the GC protocol. Therefore, the sparsity map of the network is considered as a public knowledge and will be revealed to the Garbler as well (Garbler needs to garble the circuit based on the netlist). However, the sparsity map only contains information regarding which part of the network does not contribute to the output and reveals nothing about the private network parameters. Just like the data pre-processing step, if revealing the sparsity map is not acceptable by the server (DL model owner), this step can be skipped.

6.3 Evaluations

We use Synopsys Design Compiler 2010.03-SP4 to generate the Boolean circuits. The timing analysis is performed by two threads on an Intel Core i7-2600 CPU working at $3.4GHz$ with $12GB$ RAM and Ubuntu 14.04 operating system. In all of the experiments, the GC security parameter is set to 128-bit. In our target DL setting, DeepSecure achieves an effective throughput of $2.56M$ and $5.11M$ gates per second for non-XOR and XOR gates, respectively.

Table 6.3: Number of XOR and non-XOR gates, communication, computation time, and overall execution time for our benchmarks without involving the data and DL network pre-processing.

Name	Data	Network Architecture	#XOR	#Non-XOR	Comm. (MB)	Comp. (s)	Execution (s)
Benchmark 1	MNIST [LCB17]	28×28-5C2-ReLu-100FC-ReLu-10FC-Softmax	4.31E7	2.47E7	7.91E2	1.98	9.67
Benchmark 2	MNIST [LCB17]	28×28-300FC-Sigmoid-100FC-Sigmoid-10FC-Softmax	1.09E8	6.23E7	1.99E3	4.99	24.37
Benchmark 3	Audio [mlr17a]	617-50FC-Tanh-26FC-Softmax	1.32E7	7.54E6	2.41E2	0.60	2.95
Benchmark 4	Smart-sensing [mlr17b]	5625-2000FC-Tanh-500FC-Tanh-19FC-Softmax	4.89E9	2.81E9	8.98E4	224.50	1098.3

Table 6.3 details DeepSecure performance in the realization of four different DL benchmarks without including the data and DL network pre-processing. Our benchmarks include both DNN and CNN models for analyzing visual, audio, and smart-sensing datasets. The topology of each benchmark is outlined in the third column of the Table 6.3. For instance, the first benchmark is a five-layer CNN model to classify MNIST data including: (i) a convolutional layer with a kernel of size 5×5, a stride of $(2,\ 2)$, and a map-count of 5 (indicated as layer $5C2$ in Table 6.3). This layer outputs a matrix of size $5 \times 13 \times 13$. (ii) A ReLu layer as the non-linearity activation function. (iii) A fully-connected layer that maps the $(5 \times 13 \times 13 = 865)$ units computed in the previous layers to a 100-dimensional vector (indicated as layer $100FC$ in Table 6.3). (iv) Another ReLu non-linearity layer, followed by (v) a final fully-connected layer of size 10 to compute the probability of each class. Same notation is used to describe the architecture of other benchmarks.

DeepSecure Pre-processing Effect. Table 6.4 shows DeepSecure performance for each benchmark after including the data and DL network pre-processing. Our pre-processing customization is an arbitrary step that can be used to minimize the number of required XOR and non-XOR gates for the realization of a particular DL model. As illustrated, our pre-processing

Table 6.4: Number of XOR and non-XOR gates, communication, computation time, and overall execution time for our benchmarks after considering the pre-processing steps. The last column of the table denotes the improvement achieved as a result of applying our pre-processing methodology.

Name	Data and Network Compaction	#XOR	#Non-XOR	Comm. (MB)	Comp. (s)	Execution (s)	Improvement
Benchmark 1	9-fold	4.81E6	2.76E6	8.82E1	0.22	1.08	8.95×
Benchmark 2	12-fold	1.21E7	6.57E6	2.10E2	0.54	2.57	9.48×
Benchmark 3	6-fold	2.51E6	1.40E6	4.47E1	0.11	0.56	5.27×
Benchmark 4	120-fold	6.28E7	3.39E7	1.08E3	2.78	13.26	82.83×

approach reduces GC execution time by up to 82-fold without any drop in the accuracy.

Comparison with Prior Art Framework. Table 6.5 details the computation and communication overhead per sample in DeepSecure framework compared with the prior art privacy-preserving DL system [GBDL$^+$16]. Our result shows more than 58-fold improvement in terms of overall execution time per sample even without considering the pre-processing steps. For instance, it takes 570.11 seconds to run a single instance on the pertinent MNIST network using [GBDL$^+$16] while DeepSecure reduces this time to 9.67 seconds with no data and network pre-processing. Our data and DL network pre-processing further reduces the processing time per sample to only 1.08 seconds with no drop in the target accuracy. Authors in [GBDL$^+$16] have only reported one benchmark (which we used as *benchmark 1*) for proof-of-concept evaluation. As such, we did not include the comparison for the other three benchmarks used for evaluation.

Table 6.5: Communication and computation overhead per sample in DeepSecure vs. CryptoNet [GBDL$^+$16] for benchmark 1.

Framework	Comm.	Comp. (s)	Execution (s)	Improvement
DeepSecure without pre-processing	791MB	1.98	9.67	58.96 ×
DeepSecure with pre-processing	88.2MB	0.22	1.08	527.88×
CryptoNets	74KB	570.11	570.11	-

Figure 6.2 shows the expected processing time as a function of data batch size from the client's point of view. The reported runtime for CryptoNets corresponds to implementing benchmark 1 using 5-10 bit precision on a Xeon E5-1620 CPU running at 3.5GHz as presented in [GBDL$^+$16]. Whereas, DeepSecure is prototyped using 16 bit number representation on

an intel Core-i7 processor that has a slightly less computing power compared to the Xeon processor [Pro17].

As illustrated in Figure 6.2, DeepSecure's computational cost scales linearly with respect to the number of samples. As such, DeepSecure is particularly ideal for scenarios in which *distributed clients stream* small batches of data (e.g., $N_{client} \leq 2590$) and send them to the server to find the corresponding inference label with *minimal delay*. However, CryptoNet is better-suited for settings where one client has a large batch of data (e.g., $N_{client} = 8192$) to process at once. This is because CryptoNets incurs a constant computational cost up to a certain number of samples depending on the choice of the polynomial degree. To mitigate the cost, authors in [GBDL+16] suggest processing data in batches as opposed to individual samples using scalar encoding. The data batch size, in turn, is dictated by the polynomial degree used in the realization of a particular DL model. Therefore, to acquire a higher security level one might need to use larger data batch sizes in the CryptoNet framework.

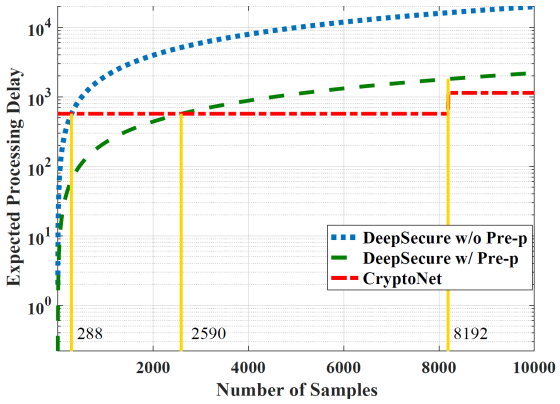


Figure 6.2: Expected processing time from client's point of view as a function of data batch size. In this figure, the *y* axis is illustrated in logarithmic scale.

6.4 Related Work

Authors in [BOP06] have suggested the use of secure function evaluation protocols to securely evaluate a DL model. Their proposed approach, however, is an interactive protocol in

which the data owner needs to first encrypt the data and send it to the cloud. Then, the cloud server should multiply the encrypted data with the weights of the first layer, and send back the results to the data owner. The data owner decrypts, applies the pertinent non-linearity, and encrypts the result again to send it back to the cloud server for the evaluation of the second layer of the DL model. This process continues until all the layers are computed. There are several limitations with this work [BOP06]: (i) it leaks partial information embedded in the weights of the DL model to the data owner. (ii) It requires the data owner to have a constant connection with the cloud server while evaluating the DL network. To address the issue of information leakage as a result of sharing the intermediate results, [OPB07] and [POC+08] enhance the protocol initially proposed in [BOP06] to obscure the weights. However, even these works [OPB07, POC+08] still need to establish a constant connection with the client to delegate the non-linearity computations after each hidden layer to the data owner and do not provide the same level of security as DeepSecure.

Perhaps the closest work to DeepSecure is [GBDL+16] in which homomorphic encryption is used as the primary tool for privacy-preserving DL computation. Unlike DeepSecure, the inherent noise in HE yields a trade-off between privacy and accuracy in evaluating the DL model which in turn translates to a lower accuracy level for obtaining a higher degree of privacy. In addition, the relatively high computation overhead of HE bounds the applicability of such approach to the use of low-degree polynomials and limited-precision numbers (e.g., 5-10 bits). SecureML [MZ17] is another secure DL framework that proposes to use additive secret-sharing and the GC protocol. Similar to CryptoNets [GBDL+16], they approximate the non-linear activation functions with polynomials. The MiniONN framework [LJLA17] is also based on additive secret-sharing and GC but has lower latency compared to SecureML approach.

A number of earlier works have shown the usability of data projection and sparsity regularization techniques to facilitate feature extraction and accelerate the execution of particular DL models [RMK17b, HPTD15, SGK17]. These set of works have been mainly focused on the functionality of DL models in terms of the accuracy and physical performance (e.g., energy

consumption, memory footprint, etc.) with no attention to the data privacy. To the best of our knowledge, DeepSecure is the first framework that introduces, implements, and automates the idea of data and DL network transformation as a way to minimize the number of required non-XOR gates for the privacy-preserving realization of DL models using GC.

6.5 Summary

We present DeepSecure, a novel practical and provably-secure DL framework that enables distributed clients and cloud servers, jointly evaluate a DL network on their private data. DeepSecure leverages automated design, efficient logic synthesis tools, and optimization methodologies to provide scalable realization of functions required for DL evaluation optimized for Yao's GC protocol. Our GC-optimized realization of hierarchical non-linear DL models demonstrates up to 58 times higher throughput per sample compared with the prior art privacy-preserving DL solution. We further propose a set of data and DL network transformation techniques as a pre-processing step to explicitly optimize the computation and communication overhead of GC protocol in the context of deep learning. Proof-of-concept evaluations using different DL benchmarks shows up to two orders-of-magnitude additional improvements achieved as a result of our pre-processing methodology.

6.6 Acknowledgements

This chapter, in part, has been published at the Proceedings of the 2018 ACM International Symposium on Design Automation Conference (DAC) and appeared as: Bita Darvish Rouhani, Sadegh Riazi, and Farinaz Koushanfar "DeepSecure: Scalable Provably-Secure Deep Learning". The dissertation author was the primary author of this material.

Chapter 7

ReDCrypt: Real-Time Privacy-Preserving Deep Learning Inference in Clouds

Deep Learning (DL) is increasingly incorporated into the *cloud business* in order to improve the functionality (e.g., accuracy) of the service. The adoption of DL as a cloud service raises serious privacy concerns in applications where the *risk of data leakage* is not acceptable. Examples of such applications include scenarios where clients hold potentially sensitive private information such as medical records, financial data, and/or location. We propose ReDCrypt, the first *reconfigurable* hardware-accelerated framework that empowers privacy-preserving inference of deep learning models in cloud servers. ReDCrypt is well-suited for *streaming (a.k.a., real-time AI)* settings where clients need to dynamically analyze their data as it is collected over time without having to queue the samples to meet a certain batch size. Unlike prior work, ReDCrypt neither requires to change how AI models are trained nor relies on two non-colluding servers to perform. The privacy-preserving computation in ReDCrypt is executed using *Yao's Garbled Circuit* (GC) protocol. We break down the deep learning inference task into two phases: (i) privacy-insensitive (local) computation, and (ii) privacy-sensitive (interactive) computation. We devise a high-throughput and power-efficient implementation of GC protocol on FPGA for the

privacy-sensitive phase. ReDCrypt's accompanying API provides support for seamless integration of ReDCrypt into any deep learning framework. Proof-of-concept evaluations for different DL applications demonstrate up to 57-fold higher throughput per core compared to the best prior solution with no drop in the accuracy.

7.1 Introduction

To incorporate deep learning into the cloud services, it is highly desired to devise privacy-preserving frameworks in which neither of the involving parties is required to reveal their private information. Several research works have been developed to address privacy-preserving computing for DL networks, e.g., [GBDL$^+$16, MZ17]. The existing solutions, however, either: (i) Rely on the modification of DL layers (such as non-linear activation functions) to efficiently compute the specific cryptographic protocols. For instance, authors in [GBDL$^+$16, MZ17] have suggested the use of polynomial-based Homomorphic encryption to make the client's data oblivious to the server. Their approach requires changing the non-linear activation functions to some polynomial approximation (e.g., square) during training. Such modification, in turn, can reduce the ultimate accuracy of the model and poses a trade-off between the model accuracy and execution cost of the privacy-preserving protocol. Or (ii) fall in the two-server settings in which data owners distribute their private data among two non-colluding servers to perform a particular DL inference. The two non-colluding server assumption is not ideal as it requires the existence of a trusted third-party which is not always an option in practical scenarios.

We propose ReDCrypt, the first provably-secure framework for *scalable* inference of DL models in cloud servers using FPGA platforms. ReDCrypt employs Yao's Garbled Circuit (GC) protocol to securely perform DL inference. GC allows any two-party function to be computed without revealing the respective inputs. In GC protocol, the underlying function shall be represented as a Boolean circuit, called a *netlist*. The truth tables of the netlist are encrypted

to ensure privacy. In contrast with the prior work, e.g., [GBDL+16, MZ17], the methodology of ReDCrypt neither requires any modification to the existing DL architecture nor relies on two non-colluding servers to ensure privacy. We designed a customized FPGA implementation for efficient execution of GC protocol on cloud servers. Our implementation provides support for the privacy-preserving realization of the computational layers used in the context of deep learning. To the best of our knowledge, no prior hardware acceleration has been reported in the literature for high-throughput and power-efficient inference of deep learning in a privacy-preserving setting.

We demonstrate that in various deep learning networks, the majority of the privacy-sensitive computations boils down to matrix-vector multiplications (Section 7.3). To achieve the highest efficiency for privacy-preserving DL computation, we devise a customized hardware architecture for this operation via GC protocol. Our custom solution is designed to maximize system throughput by incorporation of both algorithmic and hardware parallelism. Our design explicitly benefits from the precise control in synchronization with the system clock supported by the FPGA platform as opposed to a generic processor.

To leverage ReDCrypt framework, the cloud server that owns the DL model topology and weights needs to reformat the convolution operations into matrix multiplication by inserting a custom reshaping layer that unrolls the patches as discussed in Section 7.3. This pre-processing is a one-time step before generating the corresponding Boolean circuit (netlist). DL inference workload in ReDCrypt is divided into local privacy-insensitive and interactive privacy-sensitive computations. The input sample and all the intermediate features of the pertinent neural network are computed in a privacy-sensitive setting. The encoded output layer's activation is then sent to the client. The client *locally* apply a Softmax to the final encoded features to obtain the pertinent data label. The Softmax execution is considered a privacy-insensitive local computation.

Given the importance of GC protocol for secure function evaluation, it is not surprising that several GC realizations on FPGA have been reported in literature [JKSS10, SZD+16, FIL17]. The existing works, however, are devised for a generic realization of GC protocol and are not

particularly tailored for deep learning computations. By carefully designing the control flow of our customized architecture for matrix-vector multiplications through GC, we ensure minimal idle cycle due to dependency issues. In particular, we demonstrate a 57-fold improvement in throughput per core compared to the best prior-art solution. This improvement directly translates to providing support for 57 times more clients using the same number of computational cores.

The architecture of the GC accelerator embraces two recent approaches in the literature: (1) the TinyGarble [SHS+15] framework introduced sequential GC, where the same netlist is garbled for multiple rounds with updated encryption keys, (2) the work in [WGMK16] performed static analysis on the underlying function (given the control path is independent of the data path) to determine the most optimized netlist to garble in every round. In our design, there are two levels of nested loops: one outer loop and multiple inner loops. The outer loop garbles the netlist of the smallest unit of the matrix-vector multiplication operation in every round, analogous to the sequential GC of [SHS+15]. The inner loop breaks the operation of the unit into segments such that in every clock cycle we can ensure optimal utilization of the implemented encryption units. Unlike the typical GC approach, the underlying netlist is embedded in a finite state machine (FSM) that governs the transfer of the keys between gates. This approach allows us to employ a parallel architecture for the multiplication operation as we can precisely control the garbling operation in every clock cycle and ensure accurate synchronization among the gates being garbled in parallel. Unlike software parallelization, our approach does not incur any synchronization overhead. As a result, we can ensure the minimal idle cycle of the encryption units. The explicit contributions of this paper are:

- Introducing ReDCrypt, the first reconfigurable and provably-secure framework that simultaneously enables *accurate* and *scalable* privacy-preserving DL execution. ReDCrypt supports secure streaming (real-time) DL computation in cloud servers using efficient FPGA platforms.

- Designing customized hardware architecture for *GC-optimized* DL computations (e.g., matrix-vector multiplication and non-linearities) to maximize parallelism. Our architecture seamlessly supports scalability in dimension and data bit-width of the matrices.

- Presenting the first GC architecture with precise gate level control per clock cycle. Instead of conventional netlist based GC execution, we design our custom hardware accelerator as an FSM that controls the operation and communication among parallel GC cores, ensuring minimal (highest 2) idle cycles.

- Providing up to 57-fold improvement in garbling operation compared to the state-of-the-art software GC framework. This translates to the capability of the cloud to support 57 times more clients simultaneously.

- Providing proof-of-concept evaluations of various visual, audio, and smart-sensing benchmarks. Our evaluations corroborate ReDCrypt's scalability and practicability for distributed users compared to the prior-art solution.

7.2 ReDCrypt Global Flow

Figure 7.1 illustrates the global flow of ReDCrypt framework. In ReDCrypt, a cloud server (Alice) holds the DL model parameters trained for a particular inference application, and a delegated client (Bob) owns a data sample for which he wants to securely find the corresponding classified label (a.k.a., inference result). ReDCrypt empowers the cloud server (Alice) to provide a power-efficient and high-throughput concurrent service to multiple clients. The evaluation of the encrypted circuit is then performed offline on the client side. Note that to ensure privacy, the server requires to garble a new encrypted model for each client and/or input sample.

DL models have become a well-established machine learning technique. Commonly employed DL topologies and cost functions are well-known to the public. Indeed, what needs to

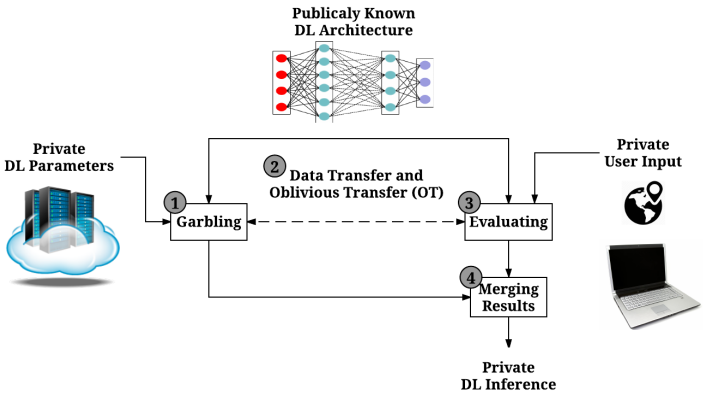


Figure 7.1: Global flow of ReDCrypt framework.

be kept private from the cloud server's perspective is the DL model parameters that have been tuned for a particular task using massive statistical databases and devoting large amounts of computing resources for several weeks/months. Data owners, on the other hand, tend to leverage off-the-shelf models to figure out the inference label for their private data while keeping their data fully private.

ReDCrypt enables computing the pertinent data inference label in a provably-secure setting while keeping both the DL model's parameters and data sample private. To perform a particular data inference in ReDCrypt framework, the netlist of the publicly known DL architecture[1] should be generated prior to the execution of the GC protocol. As shown in Figure 7.1, the execution of the GC protocol involves four main steps: (i) The server garbles the Boolean circuit of the DL architecture. (ii) The server then sends the computed garbled tables from the first step to the client along with her input wire labels. Both client and the cloud server then engage in a 1-out-of-2 Oblivious Transfer (OT) [NP05] protocol to obliviously transfer the wire labels associated with cloud server's inputs. (iii) The client evaluates (executes) the garbled circuit and computes the corresponding encrypted data inference. (iv) The server shares the mapping for the output labels with the client so that he can learn the inference results.

To provide service to a large number of distributed clients simultaneously, ReDCrypt

[1]DL architecture refers to the number and type of layers and not the values of the pertinent private DL parameters.

provides a high-throughput and power-efficient realization of the garbling (step 1) on FPGA. The garbled tabled are then sent out to the clients through multiple communication channels, which is a rational assumption for cloud servers. The cloud server in ReDCrypt acts as the garbler while the client acts as the evaluator. The motivations behind this setting are as the following:

- The server possesses the deep learning model parameters (the set of weight matrices) and the client holds the input data (a single vector). In GC, the evaluator receives his inputs through OT. Thus it is more efficient to have the client, who has less private data, as the evaluator. Even though it is possible to send all the inputs at once through OT extension [IKNP03], a resource-constrained client may not have enough memory to store all the labels together. With the recent development of sequential GC [SHS$^+$15], it is feasible to perform OT every round and store only the labels required for that round; making our approach amenable to a wide range of clients including those who are using resource-constrained embedded systems such as smart-phones.

- The garbling operation does not require any input from any party. It is only during the evaluation that the weight matrices and the client data are required. The garbling accelerator keeps generating the garbled tables independently and sends them to the host CPU. The host in the meantime dynamically updates her model if required, and when requested by the client simply sends one of the stored garbled circuits along with the input labels. Note that even if the model does not change, new labels are required for every garbling operation to ensure the security of the pertinent DL execution.

7.2.1 Security Model

We assume an *Honest-but-Curious* (HbC) security model in which the participating parties follow the protocol they agreed on, but they may want to deduce more information from the data at hand. We focus our evaluations on this security model because of the following reasons:

(i) HbC is a standard security model in the literature [KSMB13, BHKR13] and is the first step towards security against malicious adversaries. Our solution can be modified to support *malicious* models following the methods presented in [LP07, NO09, SS11, LP12]. Note that stronger security models rely on multiple rounds of HbC with varying parameters; thereby, the efficiency of ReDCrypt is carried out to those models as well. (ii) Many privacy-preserving DL execution settings naturally fit well in HbC security. For instance, when all parties have the incentive to produce the correct result (perhaps when the DL inference task is a paid service). In these settings, both data provider and the server that holds the DL model will follow the protocol in order to produce the correct outcome.

ReDCrypt's core secure function evaluation engine is the GC protocol. GC is proven to be secure in HbC adversary model [BHR12]; thereby, any input from either client or server(s) to GC will be kept private during the protocol execution. Our hardware accelerator does not alter the protocol execution and thus is as secure as any GC realization. In the rest of the paper, we will describe in details the ReDCrypt's architecture for enabling efficient privacy-preserving DL execution in cloud servers that support FPGA platforms.

7.3 System Architecture

Table 2.1 summarizes common layers used in different DL networks. Each of these layers can be effectively represented as a *Boolean circuit* used in GC. An end-to-end DL model is formed by stacking different layers on top of each other. Note that many of the computations involved in DL inference, such as non-linearity layers, cannot be accurately represented by polynomials used in Homomorphic encryption. For instance, approximating a Rectified Linear unit (ReLu) using Homomorphic encryption [GBDL$^+$16] requires leveraging high-order polynomials, whereas a ReLu can be accurately represented by a Multiplexer in Boolean circuits.

The main computational workload of DL models is related to the realization of convolu-

tional and fully-connected layers [ZWS+16]. Fully-connected layers are essentially built based on matrix-vector multiplication (see Table 2.1). The convolutional layers can also be evaluated in the form of a matrix multiplication [SLWW17]. Figure 7.2 demonstrates how a convolution operation can be effectively transformed into a matrix multiplication. As such, it is highly required to design our hardware-accelerated GC engine such that it supports the efficient realization of this operation in privacy-preserving setting. The non-linearity layers in a neural network, including Tanh, Sigmoid, and ReLu can be easily implemented as a Boolean circuit using COordinate Rotation DIgital Computer (CORDIC) and the multiplexer (MUX).

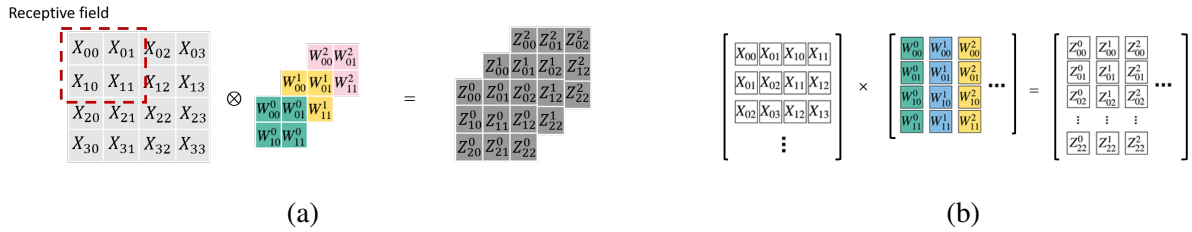


Figure 7.2: Convolution operation can be mapped into matrix multiplication: (a) convolution operation can be mapped into (b) matrix multiplication. Here, W represents weight kernels, X is the input tensor, and Z indicates the results of convolution. In this figure, the padding and stride are set to 0 and 1 with 1 channel.

DL inference workload in ReDCrypt framework is divided into two phases: (i) interactive privacy-sensitive, and (ii) local privacy-insensitive computations. The interactive privacy-sensitive phase includes computing the intermediate activation sets that rely on both the input data and DL model weights. In the very last layer of a neural network, a Softmax layer is typically used to convert the last layer activations into a probability vector and find the ultimate class. Evaluating the Softmax function in ReDCrypt framework is considered a local privacy-insensitive operation as the client is in charge of evaluating the garbled circuit and should ultimately know his corresponding data label. Nevertheless, we want to emphasize that Softmax is a monotonically increasing function. Therefore, applying this function to a given input vector does not change the index of the maximum value (inference label index). Therefore, one can also use logic building blocks such as MUX and comparator for the realization of the Softmax functionality in a

privacy-preserving setting if required by another application.

The architecture of ReDCrypt is presented in Figure 7.3. The cloud server includes a Central Processing Unit (CPU) as the host and an FPGA-based accelerator to perform the garbling operation. The GC accelerator on FPGA generates the garbled tables along with the labels for both garbler (server) and evaluator (client) and sends them to the host CPU. The CPU stores them in a buffer (not shown in the figure) and reads back when requested for an inference task by a client. Note that ReDCrypt only accelerates the GC computation on the server side and is independent of the GC realization on the client side. Moreover, the operation on the client side is transparent to the presence of ReDCrypt on the server. In general, the bottleneck of GC protocol evaluation is communication as also shown in the prior works [SHS+15]. As such, acceleration of GC evaluation on the client side is not effectual to reduce the overall latency. However, in the cloud server setting, where a single server is simultaneously communicating with a large number of clients via multiple channels, generating the garbled tables becomes the bottleneck. Therefore, accelerating this process is beneficial on the server side, but not on the client side. In the following sections, we describe in details the operation of the host CPU, followed by the GC accelerator on FPGA.

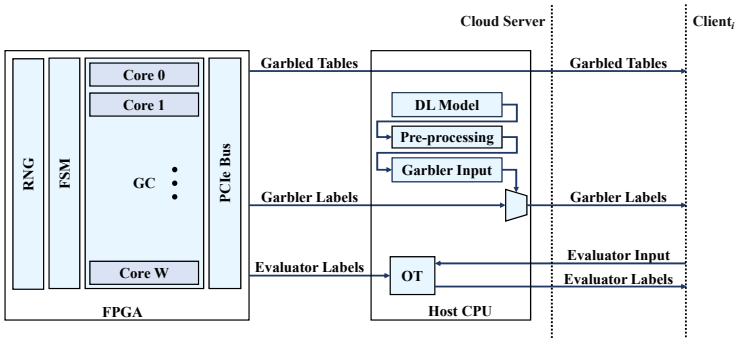


Figure 7.3: ReDCrypt system architecture on the server side.

7.3.1 Host CPU

The host CPU in ReDCrypt framework is in charge of two main tasks: **(i)** Transforming the traditional convolution layers into a matrix multiplication. To do so, ReDCrypt inserts a set of reshaping layers at the input of each convolution to unroll the input activation set as shown in Figure 2. It then reformats the 3-dimensional convolution weight tensors into a 2-dimensional matrix based on the corresponding stride and kernel size in the target convolution layer. ReDCrypt provides a set of software codes based on TensorFlow to facilitate the conversion of convolution layers into matrix multiplication format and ensure ease of use by data scientist and engineers. The transformation of the underlying neural network into a set of subsequent matrix multiplication is a one-time pre-processing step before generating the corresponding netlist. The server only needs to repeat this step if and only if it decided to update its DL model topology/weights. **(ii)** Sending the garbled labels to the client. The host CPU also gets involved in an oblivious transfer to communicate the evaluator labels/input with the client (Bob).

7.3.2 FPGA Accelerator

The garbling accelerator consists of the following components:

- Multiple garbling cores to generate the garbled tables in parallel. Each core incorporates a GC engine and a memory block to store the generated garbled tables.
- Label generator to create the wire labels required by the garbling operation. It incorporates a hardware Random Number Generator (RNG) to generate the random bit stream.
- An FSM to govern the operation of the garbling cores. The FSM replaces the netlist in the conventional GC. This approach allows us to precisely control the garbling operation customized for the matrix-vector multiplication. Note that the netlist is embedded in the FSM. Therefore, the hardware acceleration is transparent to the evaluator (client) except for the speedup in service.

- A PCI Bus to transfer the generated garbled tables to the CPU.

Motivation behind the Hardware Accelerator. In ReDCrypt, we chose to perform the parallel garbling operation on an FPGA-based accelerator rather than on a processor with multiple cores. There are several advantages of this approach. In a processor, the threads communicate among themselves through shared memory. To ensure that there are no race conditions or the threads do not read stale variables, barriers are created both before and after a thread accessing that memory. The time overhead of the creation and release of the barriers is so high as compared to the time of generating one garbling table that eventually parallelizing the GC operation do not result in an improved timing. Parallelization of garbling operation on GPU is presented in [PDL11, HMSG13], but these works precede the row reduction optimization described in Section 2.2. Therefore, they do not manage the dependency among gates. FPGA allows us to precisely control the operation in sync with the clock. In the conventional GC, the underlying function (in this case the matrix-vector multiplication) is described as a Boolean circuit (the netlist) and stored in the memory of both the garbler and the evaluator. In our implementation, instead of storing the circuit description in a file, we designed an FSM to generate the garbled tables according to the netlist. Thus we can precisely schedule the operations to make sure that all the variables (in this case the labels) are written and read in order *without the use of a barrier*.

7.4 Configuration of the GC Cores

We now present the design of the GC cores generating the garbled tables for privacy-preserving matrix-vector multiplication on the hardware accelerator. The product z of the weight matrix $W_{M\times N}$ and an input data vector $x_{N\times 1}$ is:

$$z[m] = \sum_{n=0}^{N-1} W[m,n] \times x[n], m = 0 to M-1. \tag{7.1}$$

As such, the smallest unit of the matrix-vector multiplication comprises of a multiplier followed by an accumulator, i.e., a MAC. The control flow of the FPGA accelerator comprises two nested loops. Following the methodology presented in [SHS+15], we design the MAC unit and garble (and evaluate) this unit sequentially for N rounds to compute one element of z. This forms the outer loop of the control flow. As described above, multiple parallel garbling cores are employed to generate the garbled tables for the MAC. The number of cores to be used in parallel depends on the input bit-width and available resource provisioning on the FPGA platform. In the inner loop, we break down the operation of the MAC unit such that (1) only one non-XOR operation is performed per core per clock cycle, (2) at each cycle, no core is idle due to dependency issues. Note that the cores also contain 1 to 4 XOR gates at every cycle. However, due to the free-XOR optimization they do not need costly encryption operations.

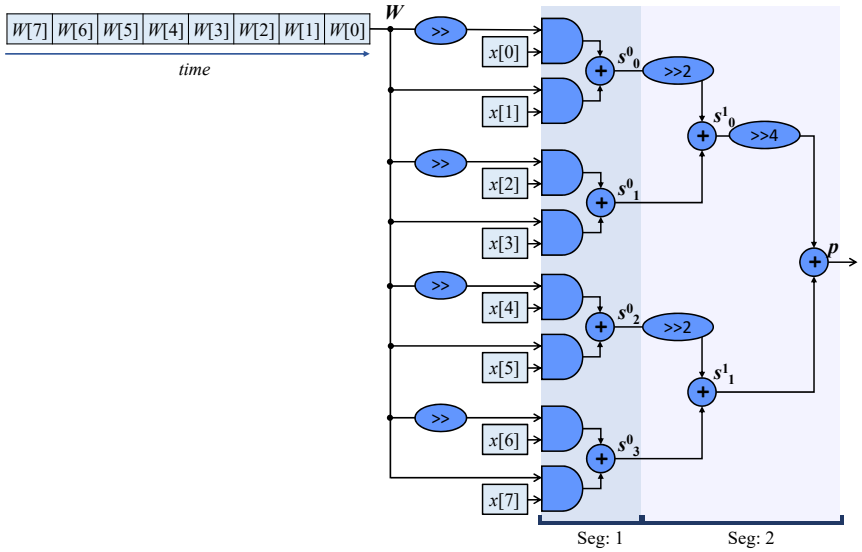


Figure 7.4: Schematic depiction of the tree-base multiplication.

For the addition operation, we employ the implementation with the minimum number of non-XOR gates (1 AND gate per input bit) provided in [SHS+15]. However, the implementation of the multiplication operation in [SHS+15] follows a serial nature that does not allow parallelism. We leverage a tree-based structure for multiplication to maximize parallelism. Figure 7.4 shows the multiplication operation of two unsigned numbers with bit-width $b = 8$. The operation for

signed numbers is discussed later in this section. The bits of x (i.e., their corresponding labels in GC operation) are constant over time for one multiplication, and the bits of W (i.e., their corresponding labels) are input to the system serially over time. The addition operations in Figure 7.4 represents one-bit full adder where the carry is transferred internally for the next cycle. In the following, we first describe the configuration of the parallel GC cores in the two segments marked in Figure 7.4 to coherently generate the necessary garbled tables. We then describe the accumulation operation and how we handle signed numbers.

7.4.1 Segment 1: MUX_ADD

The configuration of the parallel GC cores in segment 1 is displayed in Figure 7.5. Each block in the figure represents the logic operations performed by one core in every three clock cycles as shown in Figure7.6. Henceforth, we refer to every three clock cycles as one *stage*. Each GC core in this segment handles two AND gates and one adder per stage. The adder itself contains one AND and four XOR gates. The garbling engine of ReDCrypt can generate one garbled table per clock cycle, as described later in Section 7.5.1. Thus generating the three garbled tables requires three clock cycles, i.e., one stage. Note that the XOR gates do not require the generation of garbled tables thanks to the free-XOR optimization [KS08]. Instead, the output labels of an XOR gate are generated by simply XORing the input labels.

Each core m receives the labels for the two corresponding bits of x: $x[m]$ and $x[m+1]$. These labels remain unchanged for the computation of one product. All the cores receive the labels of two bits of W: $W[n]$ and $W[n+1]$ at each stage n. However, since the garbled table for only one gate is generated every clock cycle, each core needs to import one label per cycle, thus one k-bit input port is sufficient for getting the labels. The label for one bit of the W input is required for two consecutive stages; thereby at each stage after the first, one label is ported and the other one is shifted internally.

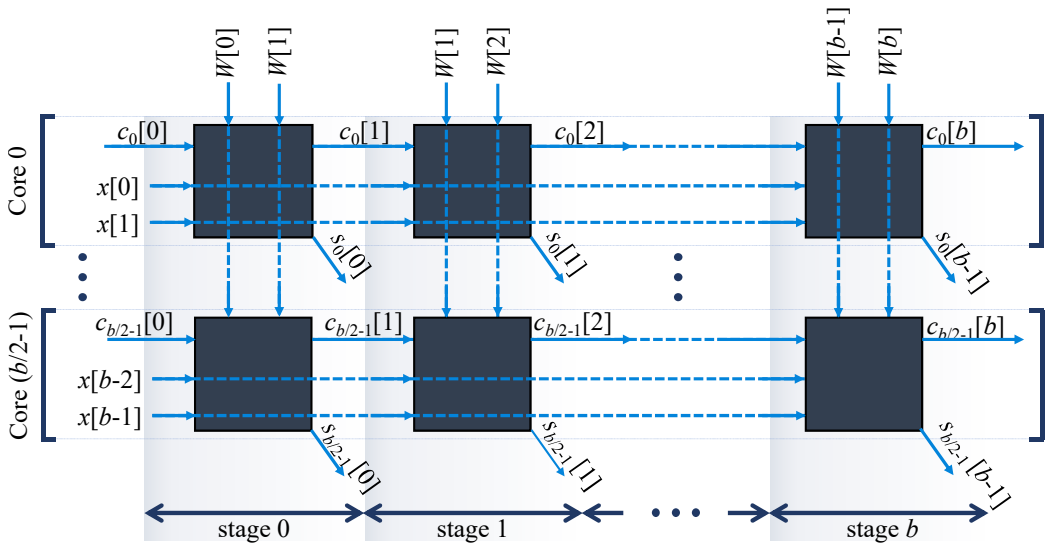


Figure 7.5: The high-level configuration and functionality of the parallel GC cores in segment 1 (MUX_ADD).

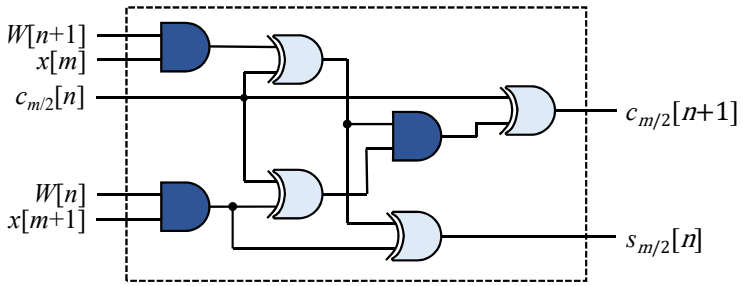


Figure 7.6: Logic operations performed in one GC core.

7.4.2 Segment 2: TREE

At each stage n, a single GC core in segment 1 generates the garbled table (and wire labels) for one bit of the sums: $s_0[n]$, ..., $s_{b/2-1}[n]$. At the next stage, these sums are added up in segment 2 according to the tree structure. The shift operations in Figure 7.4 translate to delay operations as all the cores in segment 1 perform in parallel. The number of additions performed in this segment per stage is $b/2+1$. The GC cores in this segment are designed to perform three additions per core (three garbled tables, one per each addition) so that they are synchronized with segment 1. As such, there are $\lceil((b/2-1)/3)\rceil$ GC cores in this segment.

7.4.3 Accumulator, Support for Signed Inputs and Relu

The final step of the MAC is the accumulator which requires one addition per cycle. To support signed inputs, two MUX-2's complement pairs are placed at both input and output of the multiplier. Each pair incorporates two AND gates. Finally, the ReLu operation (if needed) is basically one MUX where the selector bit is connected to the sign-bit. Each MUX contains one AND gate. Our garbling accelerator operates in a pipelined fashion, which allows us to integrate these nine AND operations (ten with the ReLu) into segment 2. Even though this approach results in an increased number of the shift registers, it ensures the minimum number of idle cycles for the GC cores. Our implementation results (see Section 7.5) show that the bottleneck of the resources is the number or Look-Up Tables (LUT), not the number of registers (Flip-Flops). Therefore, our approach results in the most optimized design.

7.4.4 Scalability Analysis

Scalability in terms of bit-with b. For bit-width b, the accelerator requires $b/2+\lceil(b/2+8)/3\rceil$ cores per MAC. Thus the maximum number of idle cores per MAC is 2. The computation of the output of one MAC takes $b+log(b)+2$ stages. However, since the operations are pipelined, the throughput is 1 MAC per b stages per $b/2+\lceil(b/2+8)/3\rceil$ cores.

Scalability in terms of the weight matrix dimensions $M \times N$ and depth of the network. The number of MAC operations increases linearly with the matrix dimension or the number of layers in the DL model (which dictates the number of matrix multiplication operation). For a constant number of cores, the computation time for one matrix-vector product increases linearly with either of M or N or the number of layers. For a constant computation time, the number of required cores increases linearly with either of M or N or the number of layers. Note that the configuration of the parallel cores dictates that the smallest increment in the number of cores is $b/2+\lceil(b/2+8)/3\rceil$.

Unlike Homomorphic encryption based frameworks, ReDCrypt is not sensitive to the depth of the underlying neural networks. HE protocol has a privacy/utility trade-off. In other words, to obtain a higher privacy level the utility of the system can decrease significantly. In particular, in the HE protocol, the noise introduced to the DL model as a result of securely encrypting data samples gets accumulated in each layer towards the output activations [GBDL$^+$16]. As such, the accuracy might degrade in deep neural networks with a large number of layers. GC protocol, on the other hand, does not induce such noise in the system due to the exact representation of the model as a Boolean circuit.

7.5 Hardware Setting and Results

We implement the prototype of ReDCrypt on a Virtex UltraSCALE VCU108 (XCVU095) FPGA. A system with Ubuntu 16.04, 128 GB memory, and Intel Xeon E5-2600 CPU @ 2.2GHz is employed as the general purpose processor hosting the FPGA. The software realization for comparison purposes is executed on the same CPU. We leverage PCIe library provided by [XIL17] to interconnect the host and FPGA platforms. Vivado 2017.3 is used to synthesize our GC units.

7.5.1 GC Engine

Each GC core incorporates one GC engine that generates one garbled table per clock cycle. The GC engine adopts all the optimizations described in Section 2.2. Our implementation involves only two logic gates: AND and XOR. The GC engine takes as its input the labels for the two input wires of the AND gate and outputs the output label and the two rows of the corresponding garbled tables. According to the methodology presented in [BHKR13], the encryption is performed by fixed-key block cipher instantiated with AES. We employ four instantiations of a single stage AES implementation to perform the four required AES encryption in parallel. The s-boxes [DR13] inside the AES algorithm are implemented efficiently by utilizing the LUTRAMs on the Virtex

UltraSCALE FPGA. The unique gate identifier T is generated by concatenating n (see Eq. 7.1), core id, stage index and and gate id (see Figure 7.5). Due to the free XOR optimization, XOR gates just require XORing the two input labels and are handle outside while the GC engine is designed to generate garbled tables only for the AND gates. This approach ensures that there is no mismatch in the timing for executing different gates as in [SZD+16] and therefore no stalling caused by dependency issues.

The labels and garbled tables are stored in the on-chip memory of the FPGA. The memory is divided into blocks with one input port per block and one output port for the entire memory. The output port is used by the PCIe Bus to transfer the generated input labels and garbled tables to the general purpose processor hosting the FPGA. Since each core has its own block in the memory with an individual input port, logically it can be visualized as each core having its own memory block.

7.5.2 Label Generator

To generate the wire labels for GC we implement on-chip hardware Random Number Generators (RNG). We adopt the Ring Oscillator (RO) based RNG suggested in [WT09]. Each RO contains 3 inverters and a single RNG XORs the output of 16 ROs. The entropy of the implemented RNG on our evaluation platform is thoroughly evaluated by NIST battery of randomness tests [RSN+01]. In the worst-case scenario, the GC accelerator requires $k \times (b/2)$ random bits/cycle. However, for a large portion of the operation, it requires only k bits/cycle on average. The label generator incorporates $k \times (b/2)$ RNGs such that it can support the worst-case setting. The FSM that synchronizes the garbling operation fully or partially turns off the operation of the RNGs to conserve energy, when possible.

7.5.3 Resource Utilization

The FPGA resource utilization of one MAC unit is shown in Table 7.1 for different bit-widths b. It can be seen from the table that the underlying resource utilization of our design increases linearly with b. We do not compare the resource utilization with the prior-art GC implementation on FPGA [FIL17] for two reasons: (i) [FIL17] being a generic GC implementation, it is difficult to estimate the resource it would require only to perform the MAC operation in similar number of clock cycles as this work, (ii) it employs SHA-1 for encryption (the most resource consuming part of the implementation), while we employ AES. SHA-1 is not considered secure anymore and all the current GC realizations in both software and hardware employ AES.

Table 7.1: Resource usage of one MAC unit.

Bit-width (b)	8	16	32
LUT	2.95E+04	5.91E+04	1.11E+05
LUTRAM	1.28E+02	3.84E+02	6.40E+02
Flip-Flop	2.44E+04	4.88E+04	8.40E+04

We plot the percentage of resources utilized per bit-width for one MAC unit in Figure 7.7. It can be seen from the plot that the bottleneck of the design on this platform is the number of LUTs, which reaches a peak of around 10% for $b = 32$. The maximum clock frequency supported by this implementation is 200MHz on the Virtex UltraSCALE.

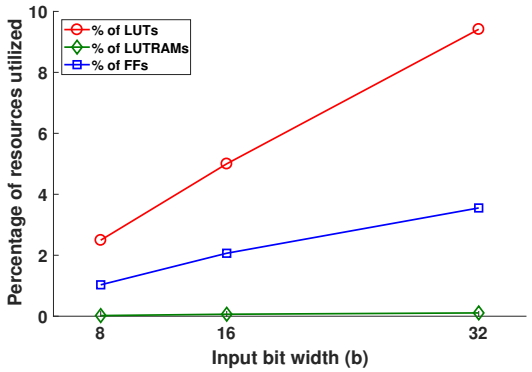


Figure 7.7: Percentage resource utilization per MAC for different bit-widths.

7.5.4 Performance Comparison with the Prior-art GC Implementation

To the best of our knowledge, ReDCrypt is the first FPGA implementation of privacy-preserving deep learning. Table 7.2 compares the throughput of ReDCrypt against the fastest available software GC framework TinyGarble [SHS+15] and the FPGA GC solution presented in [FIL17]. Both ReDCrypt and [FIL17] employ parallel GC cores to accelerate the operation. In ReDCrypt, the maximum number of parallel cores depends on the available resources in the FPGA while in [FIL17] it depends on the latency of garbling one AND gate and available BRAMs on FPGA. Considering all these, we believe that reporting the overall throughput would be ambiguous and somewhat unfair to the software framework [SHS+15]. Therefore, we report the throughput of all the frameworks per core.

Table 7.2: Throughput Comparison of ReDCrypt with state-of-the-art GC frameworks.

	TinyGarble [SHS+15] on CPU			FPGA Overlay Architecture [FIL17]			ReDCrypt on FPGA		
Bit-width	8	16	32	8	16†	32	8	16	32
Clock Cycle per MAC	1.44E+05	5.45E+05	2.24E+06	4.40E+03	1.20E+04	3.60E+04	24	48	96
Time per MAC (μs)	42.29	160.35	657.65	22.00	60.00	180.00	0.12	0.24	0.48
Throughput (MAC per sec)	2.36E+04	6.24E+03	1.52E+03	4.55E+04	1.67E+04	5.56E+03	8.33E+06	4.17E+06	2.08E+06
No of cores	1	1	1	43	43	43	8	14	24
Throughput per core (MAC per sec)	2.36E+04	6.24E+03	1.52E+03	1.06E+03	3.88E+02	1.29E+02	1.04E+06	2.98E+05	8.68E+04
$\times$ Throughput of ReDCrypt per core	1/44	1/48	1/57	1/985	1/768	1/672	-	-	-

†Interpolated from the results provided in [FIL17] for 8, 32 and 64 bits.

As shown Table 7.2, ReDCrypt accelerates the garbling operation by up to 57 times compared to [SHS+15] and at least 985 times compared to [FIL17]. Another recent GC realization on FPGA, GarbledCPU [SZD+16] do not report timing results for multiplication and addition. However, they report 2$\times$ improvement in throughput compared to JustGarbled [BHKR13] (which is the back-end of [SHS+15]) on an Intel Core i7-2600 CPU @ 3.4GHz. We estimate at least 37 times improvement over [SZD+16] in throughput per core (this work does not attempt parallelization). Due to pipeline stalls caused by dependency issues, the throughput of [SHS+15] is likely to go down further while garbling a complete netlist.

To be fair, we should state that a major factor behind the lower throughput of [FIL17, SHS+15] is their focus on general purpose GC computing while ReDCrypt is custom made for

performing DL inference only. However, the large enhancement in throughput establishes the practicality of the custom solution.

7.6 Practical Design Experiments

In this section, we evaluate ReDCrypt framework for realization of both deep learning and generic matrix-based machine learning applications.

7.6.1 Deep Learning Benchmarks

We evaluate ReDCrypt performance for the realization of four different DL benchmarks. Our benchmarks include both DNN and CNN models for analyzing visual, audio, and smart-sensing data. Table 7.3 details the computation on the server side and the transferred Bytes for each client in each benchmark. The topology of our benchmarks is outlined in the following.

Table 7.3: Number of XOR and non-XOR gates, amount of communication and computation time for each benchmark.

Id	DL Architecture	#non-XOR		#XOR		Comm. (GB)		Comp. (ms)	
		b = 8	b = 16	b = 8	b = 16	b = 8	b = 16	b = 8	b = 16
1	28×28-5C2-ReLu-100FC-ReLu-10FC-Softmax	2.09E+07	6.95E+07	5.56E+07	1.67E+08	0.68	2.25	13.04	26.07
2	28×28-300FC-ReLu-100FC-ReLu-10FC-Softmax	5.11E+07	1.70E+08	1.36E+08	4.09E+08	1.67	5.52	31.94	63.89
3	617-50FC-ReLu-26FC-Softmax	6.17E+06	2.06E+07	1.65E+07	4.94E+07	0.20	0.67	3.86	7.72
4	5625-2000FC-ReLu-500FC-ReLu-19FC-Softmax	2.35E+09	7.85E+09	6.28E+09	1.88E+10	76.90	254.22	1471.14	2942.28

Benchmark 1. Detecting objects in an image is a key enabler in devising various artificial intelligence and learning tasks. We evaluate ReDCrypt practicability in analyzing MNIST dataset [LCB17] using two different DL architectures. This data contains hand-written digits represented as 28×28 pixel grids, where each pixel is denoted by a gray level value in the range of 0-255. In this experiment, we train and use a 5-layer convolutional neural network for document classification as suggested in [GBDL$^+$16]. The five layers include: (i) a convolutional layer with a kernel of size 5×5, a stride of $(2,\ 2)$, and a map-count of 5. This layer outputs a matrix of size $5 \times 13 \times 13$. (ii) A ReLu layer as the non-linearity activation function. (iii) A fully-connected layer

that maps the $(5 \times 13 \times 13 = 865)$ units computed in the previous layers to a 100-dimensional vector. (iv) Another ReLu non-linearity layer, followed by (v) a final fully-connected layer of size 10 to compute the probability of each inference class.

Benchmark 2. We train and use LeNet-300-100 as described in [LBBH98] for the MNIST dataset [LCB17]. LeNet-300-100 is a classical feed-forward neural network consisting of three fully-connected layers interleaved with two non-linearity layers with total $267K$ parameters.

Benchmark 3. Processing audio data is an important step in devising different voice activated learning tasks that appear in mobile sensing, robotics, and autonomous applications. Our audio data collection consists of approximately 1.25 hours of speech collected by 150 speakers [mlr17a]. In this experiment, we train and use a 3-layer DNN of size $(617 \times 50 \times 26)$ with ReLu as the non-linear activation function to analyze data within 5% inference error.

Benchmark 4. Analyzing smart-sensing data collected by embedded sensors such as accelerometers and gyroscopes is a common step in the realization of various learning tasks. In our smart-sensing data analysis, we train and use a 4-layer fully-connected DNN of size $(5625 \times 2000 \times 500 \times 19)$ with ReLu as the non-linear activation function to classify 19 different activities [mlr17b] within 5% inference error.

7.6.2 Generic ML Applications

Even though ReDCrypt is designed to accelerate deep learning inference, it can greatly enhance the performance of many other machine learning applications. In this section, we analyze a number of well-known ML applications to assess the speedup provided by the custom FPGA realization of a GC based MAC operation. We assume a 32 bit fixed point system with 24 cores on ReDCrypt. Note that the throughput can be increased linearly by adding more GC cores to the FPGA. For example, about 25 times more GC cores can fit in our current implementation.

Recommendation System. The movie recommendation system in [NIW+13] presents

Table 7.4: Ridge Regression Runtime Improvement.

Name	n	d	Time [NWI$^+$13] (s)	Time Ours (s)	Runtime Improvement
forestFires	517	12	46	1.8	24.5 ×
winequality-red	1599	11	39	1.7	22.6 ×
autompg	398	9	21	1.1	18.7 ×
concreteStrength	1030	8	17	1.0	16.8 ×

an efficient implementation of privacy-preserving matrix factorization which has been widely adopted by many other works such as [KSS14, WHC$^+$14]. More than 2/3rd of the execution time in [NIW$^+$13] is spent on matrix-vector multiplication for gradient computations. The complexity of the proposed matrix factorization is $O(Mlog^2M)$ where M is the total number of ratings while the complexity of the pertinent MAC operations in each operation is $O(Sd)$ where S is summation of total number of ratings and total number of movies, and d is the dimension of user/item profile. On the MovieLens dataset, each iteration of [NIW$^+$13] takes 2.9hr. Incorporating our hardware accelerated MAC into the approach of [NIW$^+$13] significantly reduces the gradient computation time, decreasing the total runtime per iteration from 2.9hr to 1hr (69% improvement).

Ridge Regression. This method is used to find the best-fit curve through a set of data points. The work in [NWI$^+$13] combines both homomorphic encryption and Yao garbled circuits to efficiently perform privacy-preserving ridge regression. Their approach has $O(d^3)$ MACs, $O(d)$ square roots, and $O(d^2)$ divisions in the first phase and $O(d^2)$ MAC operations in the second phase. As such, accelerating the MAC operations would significantly improve the runtime as shown in Table 7.4 for selected datasets used in [NWI$^+$13]. n and d are number of samples and feature size respectively.

Portfolio Analysis. To calculate the risk to return ratio based on the stock portfolio of the investor, the client stock weight vector w (which contains relative weight of stocks in the investor's portfolio) and the financial institution stock covariance matrix cov (which is the result of financial institution's research on the market) are required. The risk to return ratio is then obtained by performing $w \times cov \times w'$ where w' is the transpose of w [CGK10]. In [VW17], the authors reported 20μs to perform 252 rounds of risk to return analysis for a portfolio of size 2 on

an Nvidia-k80 GPU. According to our evaluation, the same computation with privacy-preserving would take 1.33 seconds using TinyGarble and 15.23ms using ReDCrypt.

In the above analysis, we assumed that the cloud server has sufficient number of communication channels and bandwidth. However, after a certain threshold, communication capability of the server may become the bottleneck of the operation. Note that ReDCrypt does not affect the pertinent accuracy of the model in any of the benchmarks described above.

7.7 Related Work

Authors in [BOP06] have suggested the use of secure function evaluation protocols to securely evaluate a DL model. Their proposed approach, however, is an interactive protocol in which the data owner needs to first encrypt the data and send it to the cloud. Then, the cloud server should multiply the encrypted data with the weights of the first layer, and send back the results to the data owner. The data owner decrypts, applies the pertinent non-linearity, and encrypts the result again to send it back to the cloud server for the evaluation of the second layer of the DL model. This process continues until all the layers are computed. There are several limitations with this work [BOP06]: (i) it leaks partial information embedded in the weights of the DL model to the data owner. (ii) It requires the data owner to have a constant connection with the cloud server while evaluating the DL network. To address the issue of information leakage as a result of sharing the intermediate results, [OPB07] and [POC+08] enhance the protocol initially proposed in [BOP06] to obscure the weights. However, even these works [OPB07, POC+08] still need to establish a constant connection with the client to delegate the non-linearity computations after each hidden layer to the data owner and do not provide the same security level as ReDCrypt.

Most recently, several Cryptographic frameworks have been reported in the literature to enable DL computations in a privacy-preserving setting, e.g., [GBDL+16, MZ17, LJLA17, RRK18]. The existing frameworks, however, are not amenable to be used for real-time DL cloud

services. In particular, the existing secure DL frameworks either rely on large batch sizes for an optimized performance [GBDL+16], or require a second non-colluding server to execute their security primitives [MZ17]. More importantly, to the best of our knowledge, none of the prior work have provided hardware accelerated solution to optimize system performance for secure execution of DL models.

A number of recent works [SZD+16, FIL17] have presented an implementation of GC on FPGA. However, their primary focus is on the versatility of the framework rather than computational efficiency. In [SZD+16], the underlying netlist is always that of a MIPS processor where the secure function is loaded as a set of instructions. They report 2 times improvement in throughput (by employing a single core) over the fastest software realization at that time, while our ReDCrypt achieves 57 times increase in throughput per core. The work in [FIL17], which targets privacy-preserving data mining applications, presents an FPGA overlay architecture [BL12] where the overlay circuit contains implementations of garbled components (logic gates) upon which the netlist of the secure function is loaded. Overlay architectures in general require 40× to 100× more LUTs compared to the conventional design approach [BL12]. Even though [FIL17] achieves increased throughput by employing parallel cores, its throughput per core is about 17 times smaller compared to the software realization of [SHS+15]. As explained above, in the majority of the DL computations the privacy-sensitive computation boils down to a matrix-vector multiplication. Therefore our customized architecture on FPGA results in orders of magnitude higher throughput compared to the generic GC realizations.

7.8 Summary

We present ReDCrypt, a novel practical and provably-secure DL framework that enables the cloud servers to provide high-throughput and power-efficient service to distributed clients. The security primitive of our framework does not involve any trade-off between accuracy and

privacy. ReDCrypt targets streaming settings where the DL inference results should be computed in real time (batch size = 1). We devise custom hardware acceleration on FPGA that provide 57-fold higher throughput compared to the state-of-the-art GC solution for DL inference.

7.9 Acknowledgements

This chapter, in part, has been published at ACM Transactions on Reconfigurable Technology and Systems (TRETS) 2018 as: Bita darvish Rouhani, Siam U Hussain, Kristin Lauter, and Farinaz Koushanfar "ReDCrypt: Real-Time Privacy-Preserving Deep Learning Inference in Clouds Using FPGAs" and the Proceedings of the 2018 ACM International Symposium on Design Automation Conference (DAC) and appeared as: Siam U Hussain, Bita Darvish Rouhani, Mohammad Ghasemzadeh, and Farinaz Koushanfar "MAXelerator: FPGA accelerator for privacy preserving multiply-accumulate (MAC) on cloud servers". The dissertation author was the primary author of the ReDCrypt paper and the secondary author of MAXelerator paper. ReDCrypt is particularly designed for deep learning models and MAXelerator is a generic privacy preserving matrix multiplication framework that is designed in collaboration with Siam U Hussain.

Chapter 8

CausaLearn: Scalable Streaming-based Causal Bayesian Learning using FPGAs

In this chapter, we propose CausaLearn, the first automated framework that enables real-time and scalable approximation of Probability Density Function (pdf) in the context of causal Bayesian graphical models. CausaLearn targets complex streaming scenarios in which the input data evolves over time and independence cannot be assumed between data samples (e.g., continuous time-varying data analysis). Our framework is devised using a Hardware/Software/Algorithm co-design approach. We provide the first implementation of Hamiltonian Markov Chain Monte Carlo on FPGA that can efficiently sample from the steady state probability distribution at scales while considering the correlation between the observed data. CausaLearn is customizable to the limits of the underlying resource provisioning in order to maximize the effective system throughput. It uses physical profiling to abstract high-level hardware characteristics. These characteristics are integrated into our automated customization unit in order to tile, schedule, and batch the pdf approximation workload corresponding to the pertinent platform resources and constraints. We benchmark the design performance for analyzing various massive time-series data on three FPGA platforms with different computational budgets. Our extensive evaluations demonstrate up to *two orders-of-magnitude* runtime and energy improvements compared to the

best-known prior solution. We provide an accompanying API that can be leveraged by data scientists and practitioners to automate and abstract hardware design optimization.

8.1 Introduction

Probabilistic learning and graphical modeling of time-series data with causal structure is a challenging task in various scientific fields, ranging from machine learning [SMH07] and stochastic optimization [BR10] to economics [FS11] and medical imaging [BGJM11]. Bayesian networks are an important class of directed graph analytics used to model dynamic systems. Unlike undirected graphical networks such as Markov Random Field, Bayesian networks are capable of learning causal structure in time-series data. In a Bayesian network, the posterior probability density function over the model parameters should be continuously updated to accommodate for the newly added structural trends as data evolves over time. Dynamic (a.k.a., *streaming*) learning of random variables is particularly important in *time-series data analysis* to enable effective decision making before the system encounters natural changes, rendering much of the collected data irrelevant to the current decision space.

Energy and runtime efficiency play a key role in building viable computing systems for analyzing massive and densely correlated data. Several recent theoretical works have shown the importance of data and model parallelism in analyzing Bayesian graphical networks [WT11, CFG14, NWX13, SDB$^+$17]. These set of works, however, are designed at the algorithmic and data abstraction level and are oblivious to the hardware characteristics. Given the diminishing benefits of technology scaling, it is important to devise specialized hardware accelerators for efficient realization of different learning models [ABP$^+$16, LLW10]. A number of prior research works have provided FPGA accelerators for Bayesian networks, e.g., [MB16, LMB15]. Although these works demonstrate significant improvement for deployment of specific Bayesian models, their predominant assumption is that data samples are independently and identically drawn from

a certain distribution. As such, they cannot effectively capture dynamic data correlation in causal streaming applications (e.g., correlated time-series data).

We propose CausaLearn, the first *scalable FPGA* framework to compute on and update *continuous* random variables and their associated pdfs for *streaming-based* causal Bayesian analysis. Our key observation is that without simultaneous optimization of hardware resource allocation and algorithmic solution, the best performance efficiency cannot be achieved. To fulfill this objective, CausaLearn incorporates hardware characteristics into the higher-level hierarchy of the algorithmic solution and enables automated customization per application data and/or physical constraints. In particular, CausaLearn performs a one-time hardware physical profiling to find the pertinent resource constraints (e.g., memory bandwidth, computing power, and available energy). This information is automatically integrated into CausaLearn's resource-aware customization unit to tile, schedule, and batch the pertinent computational workload such that it best fits the platform. CausaLearn's automated compilation disengages users from hardware resource optimization task while providing synthesizable solutions that are co-optimized for the underlying hardware architecture and execution schedule.

CausaLearn leverages Gaussian processes (GP) to capture data dynamics in streaming settings. GP form a core methodology in probabilistic machine learning [Ras04, Ras04, TLR08] to model the causality structure of time-series data. Markov Chain Monte Carlo (MCMC) is the mainstream method that is used in practice to explore the state space of probabilistic models such as GP. Given the wide range of MCMC applications, it is thus not surprising that a number of implementations on CPUs [MA14], GPUs [THL11, HWSN12, BGD16, MvdWN$^+$17, TKD$^+$16], and FPGAs [MB16, LMB15, AMW08, BAFG$^+$10] have been reported in the literature. MCMC incurs a complex data flow consisting of various sequential computing kernels to construct the pertinent Markov chain. As such, FPGAs provide a more flexible programmable substrate for MCMC acceleration compared to GPU accelerators that are particularly designed for Single Instruction Multiple Data (SIMD) operations. The existing works on FPGA, however, have mainly

focused on the acceleration of MCMC for analyzing independent and identically distributed (i.i.d.) samples that are drawn from a simple multivariate Gaussian distribution, e.g., [MB16, LMB15]. Such assumption, however, does not hold for dynamic Bayesian analytics with causal structure as we illustrate in our practical design experiments. Perhaps, the only prior works on FPGA that have considered causal data dependency in the context of Bayesian networks are [AMW08, BAFG$^+$10]. Authors in [AMW08, BAFG$^+$10] have used Dirichlet processes in *discrete space* to facilitate human T-cell analysis. We emphasize that due to the discrete nature of Dirichlet processes these works are inapplicable to the analysis of dynamic *continuous* random variables.

CausaLearn adopts Hamiltonian Markov Chain Monte Carlo (H_MCMC) to effectively explore the state space of GP parameters by moving toward the gradient of the associated pdf given the observed data samples. The prior MCMC acceleration works on FPGA, e.g., [MB16, LMB15, AMW08, BAFG$^+$10] leverage random walks to sample from the target density function. Exploration of the parameters' space using random walks is particularly inefficient in analyzing *high-dimensional streaming* data due to the high cost of mitigating the impact of an unnecessary movement in constructing the Markov chain. CausaLearn overcomes this inefficiency by moving toward the gradient of the model using Hamiltonian dynamics. Computing the gradient of the target density function involves a variety of operations with complex data flows. CausaLearn provides a set of novel algorithmic and hardware optimization techniques to enable real-time execution of H_MCMC algorithm using FPGAs. In particular, our optimization includes: (i) Revising the conventional H_MCMC routine to iteratively update the corresponding gradients of the probability function using incremental data decomposition. Our algorithmic modification effectively reduces the hardware implementation complexity of computing the inverse of large matrices with no drops in the output's accuracy. (ii) Devising an automated tree-based memory management system that facilitates multiple concurrent loads/stores in order to effectively increase the system throughput by enabling data parallelism to the limits of the hardware resources. (iii) Designing an automated compilation tool to tile and schedules matrix-based computations to best

fits the data dimensionality and the available resource provisioning.

We provide an accompanying API to make CausaLearn available to a broader community who rely on probabilistic data analysis and often have a limited hardware design expertise. Our API libraries can be leveraged for deployment of widely used classes of data analytics such as various regression and classification methods, belief propagation, expectation maximization, and neural networks. In summary, our explicit contributions are as follows:

- Introducing CausaLearn, the first *scalable* framework that enables *automated real-time* multi-dimensional pdf approximation for *causal* Bayesian analysis. CausaLearn supports streaming settings in which latent variables should be updated as data evolves over time.

- Developing a resource-aware customization tool to optimize system performance. Our automated optimization attains a balance between parallel operations and data reuse by slicing the computation and configuring the design to best fit the intrinsic physical resources.

- Devising the first scalable floating-point realization of causal Gaussian processes on FPGA by adopting stochastic Hamiltonian Markov Chain Monte Carlo (H_MCMC).

- Designing an accompanying API to facilitate automation and adaptation for rapid prototyping of an arbitrary causal Bayesian data analysis. Our API minimizes the required user interaction while providing high performance and efficiency gains for FPGA acceleration.

- Providing proof-of-concept evaluations by analyzing large time-series data on three FPGAs with different computational budgets. Our evaluations demonstrate up to 320-fold runtime and 770-fold energy improvement compared to a highly-optimized software deployment. Such improvements, in turn, empower real-time data analysis in streaming settings.

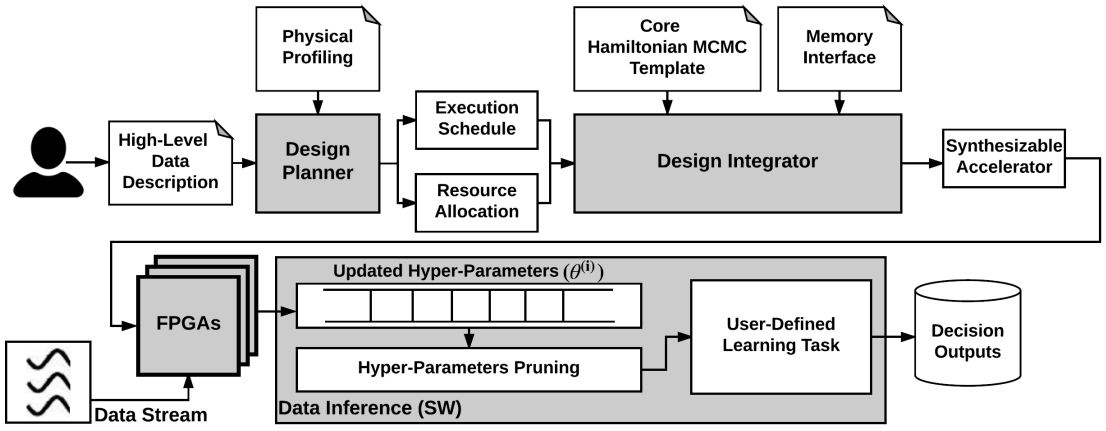


Figure 8.1: Global flow of CausaLearn framework. CausaLearn takes the stream of data samples as its input and learns the hyper-parameters of the corresponding posterior probability density function $P(\theta|\mathbf{D})$ using Hamiltonian MCMC. Our proposed Hamiltonian MCMC template is adaptively customized to the limits of the underlying platform and data structure. The updated hyper-parameters are used to perform a particular user-defined Bayesian learning task (e.g., regression or classification).

8.2 CausaLearn Global Flow

Figure 8.1 illustrates the high-level block diagram of CausaLearn framework. CausaLearn leverages Hamiltonian MCMC to devise a generic scalable framework that can be directly applied to different Bayesian applications. Hamiltonian technique is particularly of interest due to two main reasons: (i) It can handle both strong correlation and high-dimensionality in real-world applications by stochastically computing the gradient of the posterior distribution. (ii) It evades the requirement to compute the costly Metropolis-Hastings ratio commonly used in the alternative MCMC methods. This is because the acceptance rate tends to be high in the Hamiltonian method by moving toward the gradient of the target density function at each MCMC iteration as opposed to the use of random walks. CausaLearn involves two automated steps to schedule and customize the underlying data flow (Section 8.5). An accompanying API is also devised (Section 8.5.3) to ensure ease of use by users who do not necessarily possess a certain level of hardware-design knowledge. Please refer to Section 2.1 for details on different MCMC methods.

(i) Design Planner. The design planner takes the high-level description of data from the user

as its input. This description includes the rate of data arrival and feature space size in the target application. CausaLearn adopts platform profiling to abstract the physical characteristic of the target FPGA. The platform characteristics include the Block-RAM (BRAM) budget, available Digital Signal Processing (DSP) units, and memory bandwidth. The acquired physical characteristics along with the data description are fed into the design planner unit to find the optimal execution schedule and resource allocation (Section 8.5.1).

(ii) Design Integrator. The design integrator employs our core Hamiltonian MCMC (Section 8.4) as a template and customizes it according to the data schedule and resource allocation provided by the design planner. The integrator converters the acquired execution schedule into state machines and microcodes embedded in the target hardware design. CausaLearn tiles, batches, and pipelines the subsequent computational workload such that it best fits the target platform and application data (Section 8.5.2). The final synthesizable code is created after adding the memory interface to the design.

CausaLearn leverages a HW/SW co-design methodology. Bayesian analysis of streaming data involves: (i) Fine-tuning the pertinent hyper-parameters priors, and (ii) Performing a particular inference task (e.g., regression or classification) using the updated hyper-parameters. CausaLearn leverages FPGA as the primary hardware accelerator to enable real-time updating of the corresponding random variables and their associated pdfs inline with the data arrival. The FPGA is programmed with the Verilog code automatically generated as the output of the design integrator unit. The inference phase is performed on the general purpose processor that hosts the FPGA board. This is because data inference is a one-time process per input data sample and incurs a much lower computational overhead compared to that of updating the posterior distribution [CFG14]. We use Peripheral Component Interconnect Express (PCIe) port to load the data to the FPGA and write back the updated parameters to the host. All computations are performed using IEEE 754 single precision floating-point format. Floating-point representation enables CausaLearn to be readily adopted in different learning tasks without requiring the user to

modify the core implementation. It is worth noting that the fixed-point solutions are of limited applicability due to the variant nature of ultimate learning tasks and the unpredictability of data range in different applications.

8.3 CausaLearn Framework

CausaLearn leverages a three-level model hierarchy to capture the causality structure of time-series data. In particular, CausaLearn solves the following *objective function* to model the complex correlation of data samples:

$$\begin{aligned} &\textit{Observation model}: \ \mathbf{y}|\mathbf{f},\sigma_n^2 \sim \prod_{i=1}^{N} p(y_i|f_i,\sigma_n^2), \\ &\textit{GP prior}: \ \mathbf{f(x)}|\gamma \sim \mathcal{GP}(m(\mathbf{x}),K(\mathbf{x},\mathbf{x}'|\gamma)), \\ &\textit{Hyper parameters prior}: \ \theta = [\gamma,\sigma_n^2] \sim p(\gamma)p(\sigma_n^2), \end{aligned} \tag{8.1}$$

where σ_n^2 is the variance of the observation noise per Eq. (2.3) and γ is the hyper-parameter set of the predictive function **f(.)** defined as GP. All hyper-parameters $\theta = [\sigma_n^2, \gamma]$ are iteratively updated in CausaLearn framework as data evolves over time to dynamically approximate the posterior distribution $p(\theta|\mathbf{D})$. A GP model is fully defined by its second order statistics (i.e., mean and covariance). A common prior density choice for the GP covariance kernel is the squared-exponential function [TLR08]:

$$K_{ij}(\mathbf{x}) = \sigma_k^2 e^{(-\frac{1}{2}(x_i-x_j)^T\Sigma^{-1}(x_i-x_j))}. \tag{8.2}$$

Here, σ_k^2 is the variance of the kernel function and Σ is a diagonal positive definite matrix, $\Sigma = diag[\mathcal{L}_1^2,...,\mathcal{L}_d^2]$, in which each diagonal element is the length-scale parameter indicating the importance of a particular input dimension in deriving the ultimate output.

Algorithm 5 outlines the pseudocode of CausaLearn framework. The hyper-parameter

set includes the variances of the observation noise and covariance kernel along with the length-scales variables ($\theta = [\sigma_n^2, \sigma_k^2, \mathcal{L}_1, ..., \mathcal{L}_d]$). We further assume a log-uniform prior for the variance parameter σ_k^2 and a multivariate Gaussian prior for the length-scale parameters. Algorithm 5 involves four main steps, each of which are explained in detail as follows:

❶ **Platform Profiling:** CausaLearn provides a set of automated subroutine that characterize the available resource provisioning. Our subroutines measure the performance of the following four basic operations involved in the H_MCMC algorithm: matrix-matrix multiplication, dot-product, back-substitution, and random number generation. Our subroutines run the operations with varying sizes to find the target platform constraints. Note that the realization of each operation can be highly diverse depending on the target platform. For instance, based on the sizes of the matrices being multiplied, a matrix multiplication can be compute-bound, bandwidth-bound, or occupancy-bound on a specific platform.

❷ **Automated Customization:** CausaLearn design customization uses the output of physical profiling along with a set of user-defined constraints to schedule and balance the computational workload. The user-defined physical constraints can be expressed in terms of runtime (T_u), memory (M_u), and power consumption (P_u). The building blocks of the customization unit are design planner and design integrator. The details of these blocks are discussed in Section 8.5.

❸ **Dynamic Parameter Updating on FPGA:** CausaLearn takes the stream of data as its input and adaptively updates the pertinent pdf model using H_MCMC. We discuss the template H_MCMC accelerator architecture and its detailed hardware implementation in Section 8.4. Note that our proposed accelerator architecture is the first realization of Hamiltonian MCMC on the FPGA platform.

❹ **Parameter Pruning and Data Inference:** CausaLearn leverages the hyper-parameter samples drawn form the posterior distribution $p(\theta|\mathbf{D})$ to perform a user-defined data inference task (e.g., Eq. (2.6)). We use autocorrelation metric $\rho(.)$ to evaluate the mixing property of the

Algorithm 5 CausaLearn Pseudocode

Inputs: Stream of input data ($D = [X, Y]$), Initial parameters $\theta^{(1)}$, Desired Markov Chain length (C_{len}), discretization factor dt, number of discretization steps n_{step}, Updating frequency n_u, Mass matrix (M), Constant friction term (F), Portion of newly arrived data in each data batch η, Physical constraints $C_u = [T_u, M_u, P_u]$.

Outputs: Posterior Distribution Samples $\theta^{(i)}$, and output decision set O.

❶
$HW_{spec} \leftarrow PlatformProfiling()$

❷
$[b_s, HW_{code}] \leftarrow Customization(HW_{spec}, C_u)$

$ProgramingFPGA(HW_{code})$

for $i = 1, 2, ..., C_{len}$ **do**

  if $(i \mod n_u) == 0$ **then**

    $[\tilde{X}, \tilde{Y}] \leftarrow DataPartitioning(X, Y, b_s, \eta)$

    $Transferring\ Data\ Batch\ \tilde{D}\ to\ FPGA$

❸
  $r^{(i)} \sim \mathcal{N}(0, M)$

  $(\theta_1, r_1) \leftarrow (\theta^{(i)}, r^{(i)})$

  $B = \frac{1}{2}\sigma_n^2 dt$

  $E = \sqrt{2|F - B|dt}$

  for $t = 2, ..., n_{step}$ **do**

    $\theta_t \leftarrow \theta_{t-1} + M^{-1} r_{t-1} dt$

    $\bigtriangledown \tilde{U}(\theta_t) \leftarrow gradient(\tilde{D}, \theta_t)$

    $r_t \leftarrow r_{t-1} - \bigtriangledown \tilde{U}(\theta_t) dt - FM^{-1} r_{t-1} dt + \mathcal{N}(0, E)$

  end for

  $(\theta^{(i+1)}, r^{(i+1)}) \leftarrow (\theta_{n_{step}}, r_{n_{step}})$

  $Sending\ Back\ \theta^{(i+1)}\ to\ the\ Host$

❹
  $\tilde{\theta} = HyperParameterPrunning(\theta)$

  $O = UserDefinedDataInference(\tilde{\theta})$

end for

generated samples:

$$\rho_k = \frac{\Sigma_i^{N-k}(\theta_i - \bar{\theta})(\theta_{i+k} - \bar{\theta})}{\Sigma_i^N(\theta_i - \bar{\theta})}, \tag{8.3}$$

where $\bar{\theta}$ is the running average of the previous hyper-parameter samples and k is a user-defined constant that denotes the desired lag in computing the autocorrelation. CausaLearn prunes the correlated hyper-parameter samples to further reduce the computational overhead of the inference phase while providing an effective exploration of the parameters' space. We provide extensive evaluations for both regression and classification tasks in Section 8.7.

8.4 Accelerator Architecture

CausaLearn leverages batch data processing to update the hyper-parameters of the probability density function. The size of data batch to be evaluated at each MCMC iteration explicitly governs the computational workload of the underlying task. As we will discuss in Section 8.5, CausaLearn performs physical profiling and resource-aware customization to adjust the data batch size (b_s) and schedule the subsequent computations such that it best fits the target platform and application data requirements.

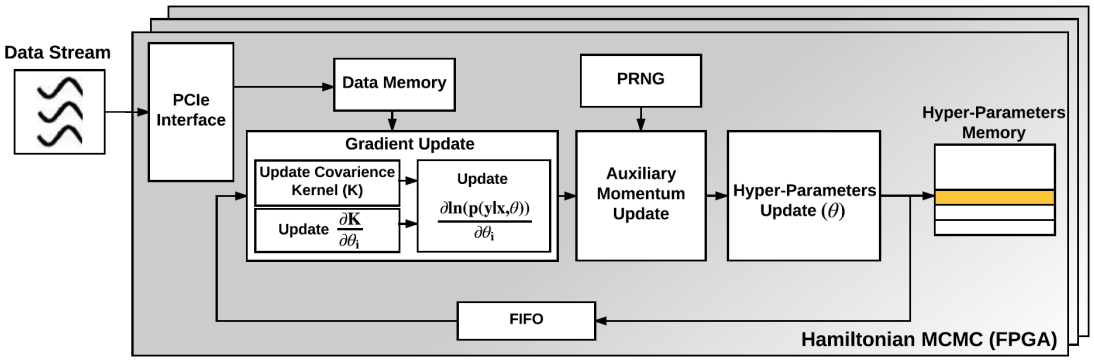


Figure 8.2: High-level block diagram of Hamiltonian MCMC.

Figure 8.2 illustrates the high-level block diagram of the H_MCMC methodology. At each H_MCMC iteration, a data batch consisting of both newly arrived data samples and a random subset of previous samples are loaded into the FPGA through PCIe to be processed using Hamiltonian dynamics (*Lines 5-7 in Algorithm 5*). We use η to denote the portion of new data in each data batch ($0 < \eta \leq 1$). Performing Hamiltonian MCMC includes three main steps:

(i) Computing the gradient of posterior distribution given the prior density function of each hyper-parameter (Line 14 of Algorithm 5). In H_MCMC, the posterior distribution of θ given a set of independent observations $\mathbf{y} \in \mathbf{D}$ is represented as $p(\theta|\mathbf{D}) \propto e^{(-U(\theta))}$, where the energy function U is equivalent to:

$$U = -\Sigma_{\mathbf{y}\in\mathbf{D}}\ ln\ p(\mathbf{y}|\mathbf{x},\theta) - ln\ p(\theta). \tag{8.4}$$

(ii) Updating the auxiliary momentum variable r. CausaLearn adds a friction term to the momentum updating step as suggested in [CFG14] to minimize the impact of injected noise as a result of bypassing the Metropolis-Hastings correction step in conventional MCMC. CausaLearn includes a scaled Pseudo Random Number Generator (PRNG) to sample from $\mathcal{N}(0,E)$ (Line 15 of Algorithm 5).

(iii) Drawing new hyper-parameter samples based on the currently computed gradients and momentum values. The Mass matrix, M, in Line 13 of Algorithm 5 is used to precondition the MCMC sampler when specific information about the target pdf is available. In many applications, the matrix M is set to the identity matrix I.

The main computational workload in Algorithm 5 is associated with computing the gradient of density function. Algorithm 6 outlines the process of computing the gradient vector $\bigtriangledown\tilde{U}(\theta_t)$ to perform H_MCMC with GP prior. Evaluating the $\frac{\partial ln(p(\mathbf{y}|\mathbf{x},\theta))}{\partial\theta_i}$ term in Line 11 of Algorithm 6 requires computing the inverse of the covariance kernel ($K_{b_s\times b_s}$). Computing the inverse of a dense $b_s \times b_s$ matrix with $b_s \gg 2$ involves a variety of operations with complex data flow. As such, we suggest adopting QR decomposition in the MCMC routine to reduce the hardware implementation complexity and make the algorithm well-suited for FPGA acceleration.

Algorithm 7 details the incremental QR decomposition by modified Gram-Schmitt technique. QR decomposition returns an *orthogonal* matrix Q and an *upper-triangular* matrix R. Utilizing QR decomposition facilitates the gradient computing step by transforming the inversion

Algorithm 6 GP Gradient Computing

Inputs: Batch of input data ($\tilde{D} = [\tilde{X}, \tilde{Y}]$), Hyper-parameter set $\theta = [\sigma_n^2, \sigma_k^2, \mathcal{L}_1, ...\mathcal{L}_d]$
Outputs: Gradient of energy function $\bigtriangledown\tilde{U}(\theta)$.

$Q^{(0)} \leftarrow [\,]$
$R^{(0)} \leftarrow [\,]$
$H \leftarrow [0,0,...,0]^T_{1\times b_s}$
for $i = 1,2,...,b_s$ **do**
    for $j = 1,2,...,b_s$ **do**
        $v^2 \leftarrow \Sigma_{k=1}^{d} \frac{(\tilde{X}_{ik}-\tilde{X}_{jk})^2}{\mathcal{L}_k^2}$
        $H_j \leftarrow \sigma_k^2 exp(\frac{-v^2}{2})$
    end for
    $H_i \leftarrow H_i + \sigma_n^2$
    $[Q^{(i)}, R^{(i)}] \leftarrow QR\_Update(Q^{(i-1)}, R^{(i-1)}, H)$
end for
$Z_i \leftarrow R^{-1}Q^T \frac{\partial K}{\partial \theta_i}$
$\frac{\partial ln(p(Y|X,\theta))}{\partial \theta_i} \leftarrow -\frac{1}{2}(Tr(Z_i) + Y^T Z_i R^{-1} Q^T Y)$
$\bigtriangledown\tilde{U}(\theta_i) \leftarrow \frac{|D|}{|\tilde{D}|}(\frac{\partial ln(p(Y|X,\theta))}{\partial \theta_i} - \bigtriangledown ln(p(\theta_i)))$

of the dense kernel matrix into the inversion of an upper-triangular matrix ($K^{-1} = R^{-1}Q^T$), which is performed using simple back substitution (Section 8.4.1).

8.4.1 Hardware Implementation

In this section, we explain the realization of H_MCMC module step by step. We leverage both algorithmic and hardware optimization techniques to implement H_MCMC efficiently.

Memory Management

To effectively pipeline the data flow in Algorithm 5 and optimize the system throughput, it is necessary to perform multiple concurrent loads and stores from a particular RAM. To cope with the concurrency, we suggest having multiple smaller-sized block memories to store particular data matrices instead of using a unified large BRAM. We devise and automate a memory management

Algorithm 7 Incremental QR decomposition by modified Gram-Schmidt

Inputs: New column H, Last iteration Q^{s-1} and R^{s-1}.
Output: Q^s and R^s.

1: $R^s \leftarrow \begin{pmatrix} R^{s-1} & 0 \\ 0 & 0 \end{pmatrix}$
2: **for** $j = 1,...,s\text{-}1$ **do**
3: $\quad R^s_{js} \leftarrow (Q^{s-1}_j)^T H$
4: $\quad H \leftarrow H - R^s_{js} Q^{s-1}_j$
end for
5: $R^s_{ss} \leftarrow \|H\|_2$
6: $Q^s \leftarrow [Q^{s-1}, \frac{H}{R^s_{ss}}]$

system to tile and schedule the matrix computations such that it best fits the data geometry and the physical hardware resources.

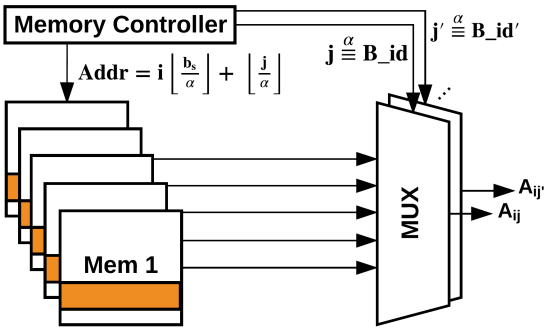


Figure 8.3: CausaLearn uses cyclic interleaving to facilitate concurrent load/store in performing matrix computations.

Figure 8.3 illustrates the schematic depiction of the memory management unit in CausaLearn framework. The block memories corresponding to a specific data matrix share the same address signal (*addr*) generated by the memory controller. The block identification index (*B_id*) is used in conjunction with the address signal to locate a certain element of the pertinent data matrix. To perform a matrix-based operation, one requires having access to sequential matrix indexes. CausaLearn's memory controller fills the corresponding memory blocks using *cyclic interleaving*. Employing cyclic interleaving enables accessing multiple successive elements of a

matrix simultaneously which, in turn, facilitates parallelizing matrix-matrix multiplications.

For a given data batch size (b_s), the number of concurrent floating-point adders/multipliers used to perform a matrix operation is directly controlled by the unrolling factor by which the data matrices are partitioned into smaller blocks. Let us denote the pertinent unroll factor with α. CausaLearn provides a set of subroutines that characterize the impact of unrolling factor α on the subsequent resource consumption. Our automated subroutines take the available resource provisioning into account and provide guidelines for an efficient hardware mapping. These guidelines are leveraged to customize the matrix-based computational workloads to the resource limits of the target platform while avoiding mapping of the data matrices into registers due to an excessive array partitioning. For instance, in Xilinx vendor libraries, every 10 floating-point numbers or less will be mapped to registers during the design synthesis. As such, α should take an integer value less than or equal to $\alpha \leq \frac{b_s}{10}$ to avoid excessive data partitioning. Note that mapping of large data matrices into the registers exhausts the LUT units on the target FPGA resulting in a complex control logic. This, in turn, translates to a larger critical path to accommodate for the underlying computations.

Tree-based Reduction

Performing matrix-vector and matrix-matrix multiplication results in frequent appearance of dot product operations similar to $c+ = A[i] \times B[i]$. Due to the sequential nature of dot products (Figure 8.4a), simple use of pipelining/unrolling does not significantly reduce the Initiation Interval (II) between two successive operations. As such, we suggest to transform such sequential operations ($c+ = A[i] \times B[i]$) into a series of operations that can be independently run in parallel (e.g., $W[i] = A[i] \times B[i]$). In particular, we implement a tree-based adder to find the final sum value c by adding up the values stored in a BRAM called W. We use cyclic interleaving for storing all the involving arrays including A, B, and W to facilitate pipelining the subsequent multiplications and additions (Figure 8.4b).

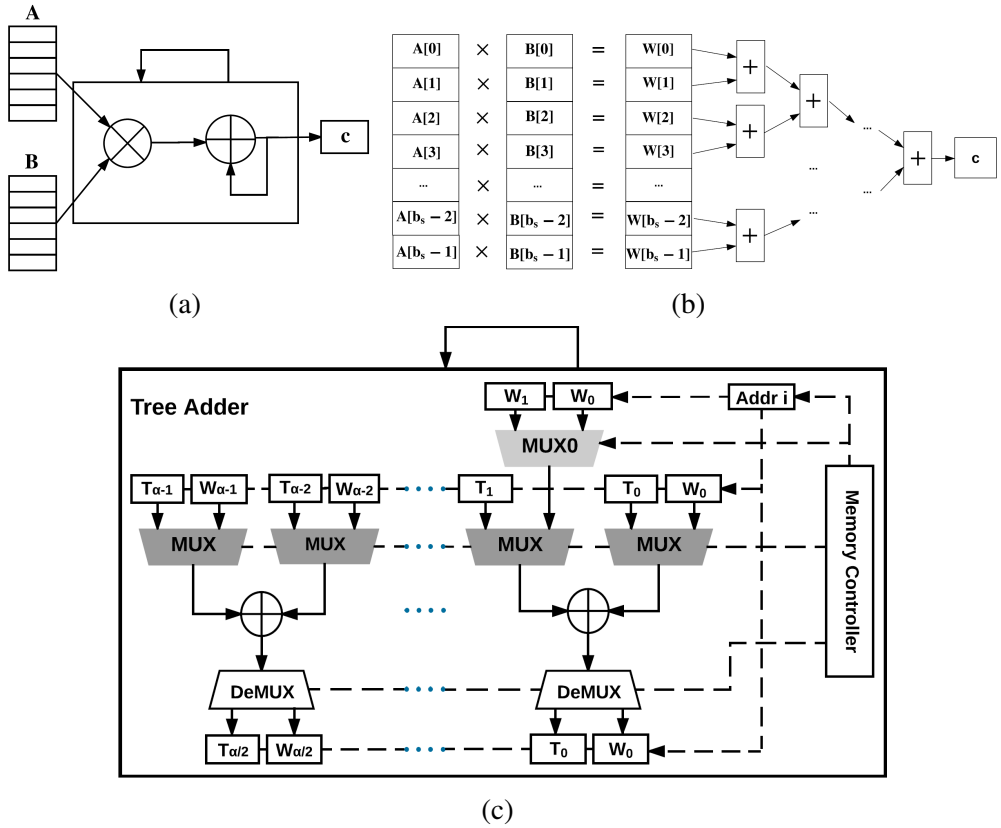


Figure 8.4: Facilitating matrix multiplication and dot product using tree structure. (a) Conventional sequential approach. (b) Proposed tree-based model. Our approach reduces the II of dot product operations to 1. (c) The inner structure of tree-based adder.

Figure 8.4c illustrates the inner structure of CausaLearn tree-based adder. We utilize a temporary array T within the tree adder module to store the intermediate results. In our tree adder module, the number of additions performed at each stage is halved and the result is stored in the other array. E.g., in the even stages, the values in the array W are summed up and the results are stored in the array T. CausaLearn's memory controller generates the appropriate source/destination addresses to load/store the intermediate results at each stage of the tree. The number of floating-point adders/multipliers in the tree-based reduction module is equivalent to the unrolling factor α used to partition data matrices (assuming dual port memory blocks). Let us index the elements of arrays W and T with variable k. Each sub-block W_i and T_i in Figure 8.4c is filled such that $k \equiv i \mod \alpha$ where $k \in \{0, 1, ..., b_s\}$. In the tree adder structure, the multiplexer

denoted by "MUX0" is necessary for performing the last b_s/α additions on the remaining values in W_{+1}. The final result (c) is stored in the address 0 of memory assigned to array W. As will be discussed in Section 8.4.1, CausaLearn attains a balance between parallel operations and data reuse by scheduling a slice of operations to be performed at each clock cycle.

Matrix Inverse Computation

Computing the inverse of the covariance kernel K is a key step in finding the gradient direction in the H_MCMC routine. Employing QR decomposition within the H_MCMC routine facilitates such operations given that K^{-1} can be computed as $R^{-1}Q^T$. For instance, to solve an equation similar to $V = K^{-1}B$, one needs to find the vector V such that $RV = Q^TB$. Given the upper-triangular structure of matrix R, the latter equation can be solved using back-substitution [RMSK16, RSMK15, RMK17c] in which (starting from the last row index) each element of the vector V can be uniquely recovered by solving a linear equation as illustrated in Figure 8.5. Let us denote the product of Q^TB with vector C. The Processing Element (PE) in Figure 8.5 is a multiply-add accumulator that computes:

$$V_i = \frac{C_i - \Sigma_{j=i+1}^{b_s} R_{ij}V_j}{R_{ii}}. \tag{8.5}$$

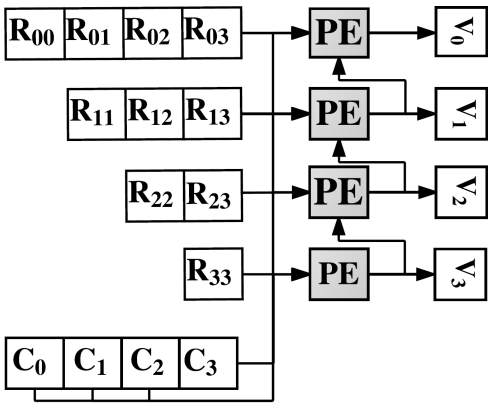


Figure 8.5: Schematic depiction of back-substitution.

CausaLearn performs back-substitution by parallelizing the computations as shown in Figure 8.6. Cyclic interleaving along the second dimension (matrix columns) is used to store the Q and R matrices. This enables us to pipeline the design and reduce the II between two successive operations into only 1 clock cycle. Indices of vectors and the second dimension of matrices in Figure 8.6 correspond to their actual values modulo α. We batch the operations in the back-substitution module to parallelize computations that share the same variables. E.g., in computing Line 10 of Algorithm 2, multiple columns of matrix $Z_{b_s \times b_s}$ may be batched together to facilitate computations given that the columns of matrix Z can be computed independently using the same set of Q and R values.

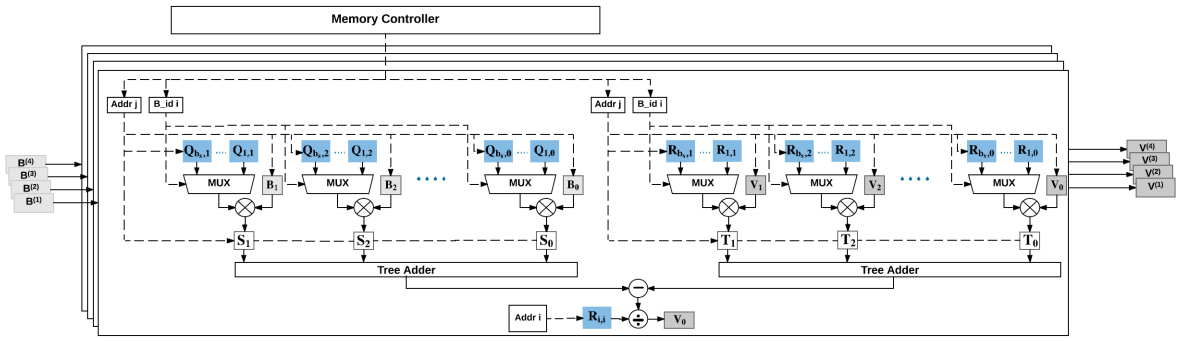
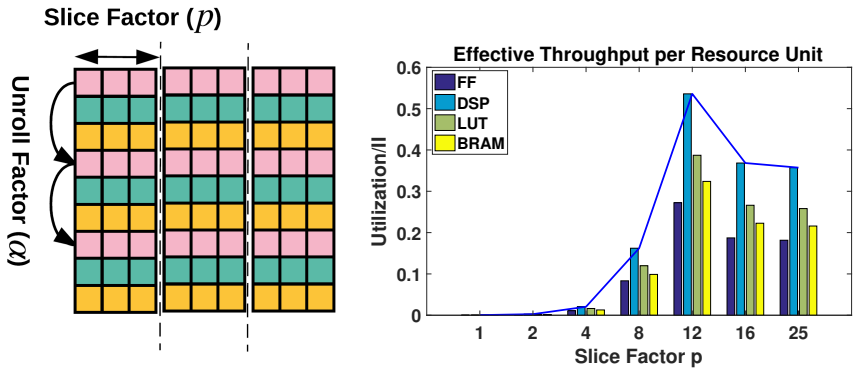


Figure 8.6: CausaLearn architecture for computing back-substitution. The operations in the right and left side of the equation, $R_{b_s \times b_s} V_{b_s \times b_s} = Q^T_{b_s \times b_s} B_{b_s \times b_s}$, are parallelized to optimized system throughput per iteration. We use cyclic interleaving along the second dimension to store the Q and R matrices. Each column of matrix $B^{(i)}$ is partitioned into smaller blocks to further accommodate parallelism. In this figure, we used dash lines to indicate the control signals.

Data Parallelism

CausaLearn gains a balance between parallel operations and data reuse by partitioning matrix-based computations into smaller slices of operations that best match the available computational resources such as DSP units. Figure 8.7a shows an example where multiple columns of matrix V are scheduled as a slice of operations to be evaluated in parallel in the matrix inversion unit ($R_{b_s \times b_s} V_{b_s \times b_s} = Q^T_{b_s \times b_s} B_{b_s \times b_s}$). As shown in Figure 8.7b, there is a trade-off between the number of samples per slice of computations and resource utilization. CausaLearn leverages this

trade-off to optimize the template design such that the throughput per resource unit is maximized. The effective throughput per resource unit decreases for large values of slice factor p. This performance drop is due to the saturation of the pertinent resource provisioning which, in turn, makes it infeasible to perform more operations in parallel. We leverage batch data parallelism within different parts of the framework (e.g., tree-based reduction module, matrix inversion unit, etc.) to improve the efficiency of the system.



(a) (b)

Figure 8.7: Example data parallelism in CausaLearn matrix inversion unit: (a) Partitioning of matrix computations into slices of operation. (b) Resource utilization divided by the pertinent initiation interval as a function of the number of samples per operation slice on Virtex-7-XC7VX485T FPGA.

8.5 CausaLearn Customization

The architecture discussed in Section 8.4 serves as a template for the accelerator's micro-architecture. Here, we outline our design customization methodology to adapt the H_MCMC routine to the resource boundaries of the target platform.

8.5.1 Design Planner

Table 8.1 details the memory footprint and runtime cost in CausaLearn framework. Memory constraint on computing platforms is one of the main limitations in big data regime.

CausaLearn updates the posterior distribution samples of a dynamic data collection by breaking down the input data into data batches that best fit the memory budget. CausaLearn's memory footprint outlined in Table 8.1 specifies the storage requirement for the gradient matrices corresponding to each GP hyper-parameter, the covariance kernel $K = QR$, and the intermediate matrices Z_i (Line 10 in Algorithm 6).

The runtime requirement for data analysis in CausaLearn can be approximated as:

$$T_{CausaLearn} \propto T^{comm} + T^{Comp}. \tag{8.6}$$

The T^{comm} term denotes the communication overhead of sending a data batch of size $b_s \times d$ from host to the FPGA platform and reading back the updated posterior distribution parameters θ. The T^{Comp} term represents the runtime cost of updating the covariance matrix K and computing the gradients as outlined in Algorithms 5, 6, and 7. The computation and communication costs in CausaLearn framework are detailed in Table 8.1. As we demonstrate in Section 8.7, CausaLearn's overall runtime is mainly dominated by the computational workload while the communication cost contributes to a small fraction of the overall runtime (e.g., $\leq 0.03\%$).

Table 8.1: CausaLearn memory and runtime characterization.

Physical Performance of CausaLearn Framework	
Memory Footprint	$M_{CausaLearn} \approx N_{bits} n_k (4+d) b_s^2$ N_{bits}: *Number of signal representation bits* n_k: *Number of H_MCMC units working in parallel* b_s: *Number of samples per data batch-size* d: *Feature space size of the incoming data samples*
Computation Runtime	$T^{comp} \approx \beta_{flop} C_{len} n_{step} (6b_s^2 d + b_s d)$ β_{flop}: *Computational cost per floating-point operation* C_{len}: *Desired Markov chain's length* n_{step}: *Number of discretization steps in H_MCMC*
Communication Runtime	$T^{comm} \approx \beta_{net} + \frac{N_{bit} C_{len} [b_s d + (d+2)]}{BW}$ β_{net}: *Constant network latency* BW: *Operational communication bandwidth*

There is a trade-off between the selected data batch size b_s and the required runtime

to reach the Markov chain steady state distribution, a.k.a., *mixing time* [CFG14, BDH14]. On the one hand, a high value of b_s reduces the number of iterations to reach the steady state distribution. However, it also reduces the throughput of the system as data can no longer fit in the fast BRAM of the target board. On the other hand, a low value of b_s may degrade the overall performance due to the significant increase in the number of required posterior samples to compute a steady approximation of Eq. (2.5). CausaLearn carefully leverages this trade-off to customize computations to the limits of the physical resources and constraints while minimally affecting the mixing time in the target application.

To deliver the most accurate approximation within the given resource provisioning, CausaLearn solves the optimization objective described in Eq. (8.7). CausaLearn's constraint-driven optimization can be expressed as:

$$\begin{aligned} &\underset{b_s,\, n_k}{\textit{minimize}}\ (\textit{MC mixing time}), \\ &\textit{subject to: } T^{comm} + T^{Comp} \leq T_u, \\ &\eta n_k b_s \leq f_{data} T_u, \\ &M_{CausaLearn} \leq M_u, \\ &P_{CausaLearn} \leq P_u, \\ &n_k \in \mathbb{N}, \end{aligned} \tag{8.7}$$

where T_u, P_u, and M_u are a set of user-defined parameters that imply the application constraints in terms of runtime, power, and memory respectively. The maximum number of newly arrived samples that should be processed in each time unit is either dictated by the arriving rate of data samples (f_{data}) or the buffer size for storing incoming samples (M_u). Here, η is the proportion of newly arrived samples versus the old ones in each data batch. For a fixed set of parameters, power consumption ($P_{CausaLearn}$) has a linear correlation with the number of MCMC modules that are run in parallel. CausaLearn tunes the number of concurrent MCMC modules accordingly

to adapt to possible power limitations imposed by the target setting.

CausaLearn approximates the solution of Eq. (8.7) by fixing the number of parallel H_MCMC units (n_k) and solving for data batch size (b_s) using the Karush-Kuhn-Tucker (KKT) conditions. To facilitate automation, we provide a solver for our optimization approach. The solver gets the constraints from the user as inputs and uses our Mathematica-based computational software program to solve the optimization. Note that the constraint-driven optimization is a one-time process and incurs a constant, negligible overhead.

8.5.2 Design Integrator

The design integrator unit in CausaLearn framework takes the acquired execution schedule into consideration and generates the corresponding state machines and microcodes to manage the memory controller and data parallelisms discussed in Section 8.4.1. The customized synthesizable code is generated after embedding the microcodes within the template H_MCMC architecture. In our prototype designs, we leverage PCIe to transfer data back and forth between the FPGA and the general purpose processor hosting the FPGA. The PCIe interface can be replaced by any other data transfer link such as Ethernet depending on the application.

8.5.3 CausaLearn API

CausaLearn API consists of a set of high-level automated subroutines which perform the subsequent steps outlined in Figure 8.1. Programmers interact with our API only through providing the input data stream and pertinent physical constraints in terms of the available memory, runtime, and/or power inside a bash file. CausaLearn finds the optimal batch size (b_s) using our Mathematica-based optimizer as discussed in Section 8.5.1. The API then calls Vivado-HLS to search for optimal values of various design directives including unroll factor, slice factor, and pipeline depth that yield the maximum throughput while complying with the user-defined

constraints. Eventually, the customized H_MCMC core along with the required I/O interface modules are generated to be implemented on FPGA.

In CausaLearn, API follows specific steps to find the optimal values for each HLS directive in an automated manner. For instance, the optimal value of slice factor is obtained by synthesizing the design using different values of slice factor and collecting utilization and initiation interval from the synthesis report. The optimal value is either the local optima of the effective throughput per resource unit as depicted in Figure 8.7b or the maximum value that allows the design to fit user-specific constraints (when using the local optima exceeds the user constraints). After setting the slice factor, unroll factor is determined to increase data parallelism while maintaining the design metrics below the specific physical constraints. It is noteworthy that the whole customization process is automated so that data practitioners with different scientific backgrounds that do not necessarily possess any particular hardware design knowledge can benefit from CausaLearn end-to-end design. Depending on the synthesis speed on the host machine and data dimensionality, profiling can take 5 to 30 minutes on commodity personal computers. Note that profiling is performed once per application/platform and its cost is amortized over-time as the system is used for processing data streams.

8.6 Hardware Setting and Results

We evaluate CausaLearn using three off-the-shelf FPGA evaluation boards namely Zynq ZC702 (XC7Z020), Virtex VC707 (XC7VX485T), and Virtex UltraScale VCU108 (XCVU095) as the primary hardware accelerator. We use an Intel core-i5 CPU with $8GB$ memory running on the Windows OS at $2.40GHz$ as the general purpose processor hosting the FPGA. The software realization of CausaLearn is employed for comparison purposes. We leverage PCIe library provided by [XIL17] to interconnect the host and FPGA platforms. Vivado HLS 2016.4 is used to synthesize and simulate our MCMC units. All FPGA platforms work at $100MHz$ frequency.

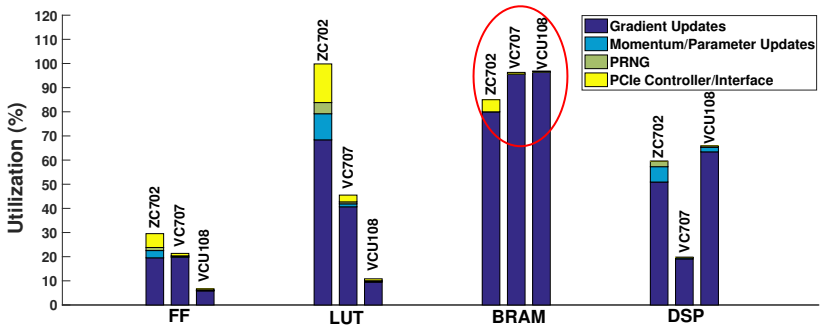


Figure 8.8: Resource utilization of CausaLearn framework on different FPGA platforms assuming a hyper-parameter set of size $|\theta| = 10$. The output of our automated customization characterizes the hardware accelerator which, in turn, helps us to fully exploit the available on-chip memory. As shown, the resource utilization is mainly dominated by the Gradient update unit.

Figure 8.8 shows the breakdown resource utilization of CausaLearn deployed on three FPGAs. Each FPGA platform has a different computational budget. The total resource utilization accounts for both the H_MCMC unit (including the gradient update, momentum and parameter update, and PRNG modules) as well as the PCIe controller. Table 8.2 details CausaLearn performance per iteration of H_MCMC for processing different number of data samples (n). The earlier MCMC hardware accelerators are developed based on the assumption that input data samples are independent and identically distributed. These works cannot handle time-series data with causal structure as shown in Figure 8.9. As such, we opt to compare CausaLearn runtime (with $n_k = 1$) and energy consumption against a highly optimized C++ software solution. The software baseline is optimized using Eigen and OpenMP libraries. Eigen library exploits Intel Stream SIMD Extension (SSE) instructions to enhance the performance of intensive matrix computation. All the available cores on the Intel Core-i5 CPU (with 8GB memory running at 2.40GHz) were used to execute the H_MCMC routine.

FPGA power is simulated using Vivado power analyzer which accounts for both static and dynamic power. We use Intel Power Gadget 3.0.7. to measure CPU execution power. The power consumption for the H_MCMC unit is 0.95, 3.74, and 3.84 *Watts* for ZC702, VC707,

Table 8.2: Relative runtime/energy improvement per H_MCMC iteration achieved by CausaLearn on different platforms compared to the optimized software implementation for $|\theta| = 10$ and $n_{step} = 100$. The conventional H_MCMC algorithm incurs $O(n^2)$ runtime complexity, whereas, our batch optimization approach scales linearly with $[\frac{n}{b_s}]$.

n	Communication Overhead	Runtime per Iteration SW	CausaLearn Runtime per Iteration			Runtime Improvement			Energy Improvement		
			ZC702	VC707	VCU108	ZC702	VC707	VCU108	ZC702	VC707	VCU108
256	16.92 *msec*	113.01 *sec*	96.18 *sec*	47.10 *sec*	76.71 *sec*	1.2×	2.4×	1.5×	8.1×	4.1×	2.4×
512	33.65 *msec*	902.98 *sec*	192.37 *sec*	94.23 *sec*	153.42 *sec*	4.7×	9.6×	5.9×	31.8×	16.5×	9.8×
1024	67.18 *msec*	8601.37 *sec*	384.72 *sec*	188.61 *sec*	230.13 *sec*	22.4×	42.7×	37.4×	136.3×	65.9×	56.3×
2048	134.19 *msec*	33.52 *hr*	769.44 *sec*	376.81 *sec*	460.26 *sec*	156.8×	320.2×	262.2×	769.1×	398.9×	318.2×

and VCU108, respectively. As illustrated, the computational time in CausaLearn grows linearly with respect to the number of data samples. In this experiment, the optimal batch size for each platform is used to maximize the on-chip memory usage as shown in Figure 8.8. The optimal data batch sizes (output of CausaLearn customization) are 88, 256, and 360 on ZC702, VC707, and VCU108, respectively. In cases where the number of data samples is not divisible by the data batch, $[\frac{n}{b_s}]$ iterations are performed to analyze all data samples.

8.7 Practical Design Experiences

We use CausaLearn to analyze three large time-series data with strong causal structure. In particular, we analyze the following datasets:

(i) Dow Jones Index stock's change over time. This data [UCI16a] includes daily stock data of 30 different companies collected over 6 months. Each data sample x_i contains 8 features including different statistics of the stock price during the previous week (e.g., the highest and lowest price). The task is to predict the percentage of return for each stock in the following week.

(ii) Sensor data to classify different daily human activities. The dataset [UCI16b] comprises body motion and vital signs recordings for ten volunteers while performing different activities. Each data sample x_i includes 23 features. In this experiment, we use the data collected for two subjects to distinguish jogging and running activities. Each activity is recorded for 1 minute with a sampling rate of $50Hz$ resulting in more than $6K$ samples per subject.

(iii) Time-variant data for regression purposes [VRH+13]. The data is generated using a time-variant (unknown) function where the task is to predict the function's output given the previously observed samples. Figure 8.9a shows the regression's output using the posterior distribution samples learned by CausaLearn (Figure 8.11c). In Figure 8.9, we compare the regression's output using MCMC samples learned by assuming a causal GP prior versus i.i.d. data measurements with multivariate Gaussian prior (e.g., [MB16, MB12]). The data points denoted by star signs are the training observations **y**.

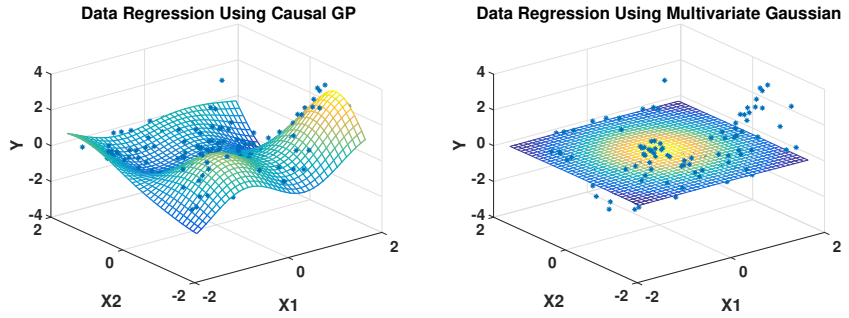
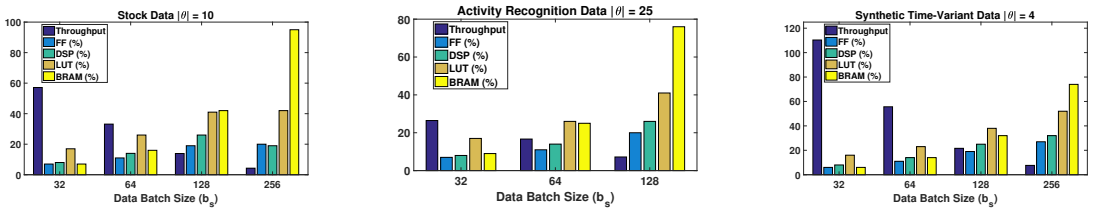


Figure 8.9: Time-variant data analysis using MCMC samples by assuming (a) causal GP prior (CausaLearn), vs. (b) i.i.d. assumption with multivariate Gaussian prior (e.g., [MB16, MB12]).

Data batch size, b_s, is a key tunable parameter that characterizes CausaLearn's resource utilization and runtime performance as outlined in Table 8.1. Figure 8.10 demonstrates the impact of data batch size b_s on the subsequent resource utilization and system throughput per H_MCMC unit in each application. Multiple H_MCMC units can work in parallel within the confine of the resource provisioning to further boost the system throughput for smaller data batch sizes.

Figure 8.11 shows CausaLearn's posterior distribution samples obtained with a batch size of $b_s = 128$ in each application. The red cross sign on each graph demonstrates the maximum a posterior (MAP) estimate obtained by solving:

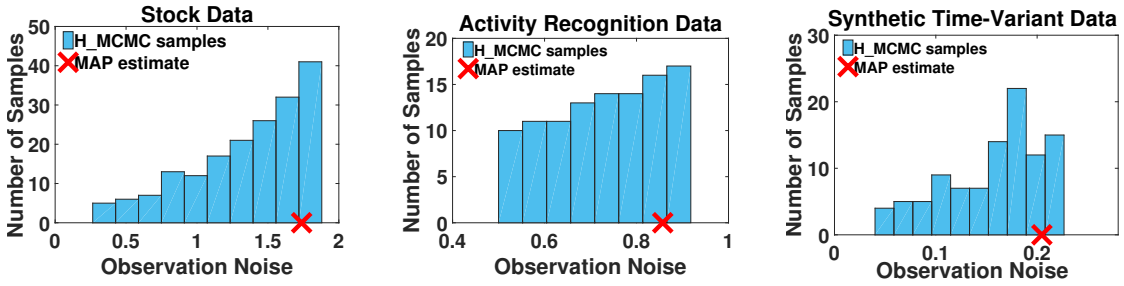
$$\underset{\theta}{argmax}\ ln(p(\mathbf{y}|\mathbf{x},\theta)) + ln(p(\theta)). \tag{8.8}$$



(a) (b) (c)

Figure 8.10: VC707 resource utilization and system throughput per H_MCMC unit ($n_k = 1$) as a function of data batch size b_s in different applications. The reported throughputs indicate batch per second processing rate corresponding to $n_{step} = 100$.

Due to the space limit and high dimensionality of the target datasets, Figure 8.11 selectively shows the MCMC samples obtained for the observation noise variance (σ_n^2). The same trend is observed for the other hyper-parameters (e.g., σ_k^2 and $\mathcal{L}_i$).



(a) (b) (c)

Figure 8.11: Example CausaLearn's posterior distribution samples. The red cross sign on each graph demonstrates the maximum a posterior estimate in each experiment.

8.8 Related Work

Bayesian network is a key method to model dynamic systems in various statistical and machine learning tasks. Significant theoretical strides have been made to design Bayesian graphical analytics that can be used at scales by exploiting task and data level parallelism [WT11, CFG14, NWX13, SDB+17, MCF15]. Available Bayesian inference tools on CPUs [MA14], GPUs [THL11, HWSN12, BGD16, HJBB14], and FPGAs [AMW08, BAFG+10], however, are either application specific or include direct mappings of algorithms to hardware. As such, the

idea of customizing the Bayesian networks to make them well-suited for the underlying platform is unexplored. Recently, authors in [MvdWN+17, TKD+16] have introduced a generic GPU-accelerated framework for Bayesian inference. Even these works are built based the assumption that input data samples are i.i.d; thus lack the capability to capture the inherent causal structure of time series data. To the best of our knowledge, CausaLearn is the first automated framework that enables end-to-end prototyping of complex causal Bayesian analytics with continuous random variables. CausaLearn is capable of handling both strong correlation and high-dimensionality in streaming scenarios with severe resource constraints.

FPGAs have been used to accelerate computationally expensive MCMC methods. Recent works in [MB16, LMB16, MB12] have proposed reconfigurable architectures with custom precision for efficient realization of population-based MCMC routine applied to Bayesian graphical models. Authors in [MB16, LMB16, MB12] targets simple multivariate Gaussian densities where observations are assumed to be independent and identically distributed. Thus, these works cannot be readily employed in more sophisticated streaming scenarios where independence cannot be assumed between data samples. To the best of our knowledge, CausaLearn is the first to provide a scalable FPGA realization of generic H_MCMC routine applied to streaming applications with large and densely correlated data samples. We emphasize that the use of data precision optimization technique proposed in [MB12, MRB13] provide an orthogonal means to our resource-aware customization for performance improvement. Therefore, CausaLearn can achieve even greater improvement by leveraging such optimizations.

8.9 Summary

This chapter presents CausaLearn, the first automated reconfigurable framework to compute on and continuously update time-varying probability density functions for causal Bayesian analysis. CausaLearn targets probabilistic learning in streaming scenarios in which the number

of data samples grows over time and computational resources are severely limited. To boost the computational efficiency, CausaLearn provides a scalable implementation of Hamiltonian MCMC on FPGA. We modify the conventional MCMC algorithm using QR decomposition to make it amenable for hardware-based acceleration performed by FPGA platforms. We further provide novel memory management, tree-based reduction, and data parallelism techniques to effectively pipeline and balance the underlying matrix computations on FPGA. CausaLearn is devised with an automated constraint-driven optimization unit to customize H_MCMC workload to the limits of the resource provisioning while minimally affecting the MC mixing time. An accompanying API ensures automated applicability of CausaLearn for an end-to-end realization of complex Bayesian graphical analysis on massive datasets with densely correlated samples.

8.10 Acknowledgements

This chapter, in part, has been published at the Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA) and appeared as: Bita Darvish Rouhani, Mohammad Ghasemzadeh, and Farinaz Koushanfar "CausaLearn: Automated Framework for Scalable Streaming-based Causal Bayesian Learning using FPGAs". The dissertation author was the primary author of this material.

Chapter 9

Summary and Future Work

Physical viability and safety consideration are key factors in devising machine learning systems that are both sustainable and trustworthy. Learning and analysis of massive data is a trend that is ubiquitous among different computing platforms ranging from smartphones and Internet-of-Thing devices to personal computers and many-core cloud servers. Concerns over the accuracy, physical performance, and reliability are major challenges in building automated learning systems that can be employed in real-world settings. This thesis addresses these three critical aspects of emerging computing scenarios by providing holistic automated solutions that simultaneously capture the best of Computer Architecture, Machine Learning, and Security fields. The research contained in this thesis opens new interesting directions including but not limited to:

Developing holistic content and platform aware solutions and tools for scalable massive data analytics. This type of research, in turn, enables designing powerful computing systems that can automatically evolve and adapt to new data, algorithm, and hardware platforms. I believe that multi-disciplinary research empowered by tools and ideas from data science and algorithms, as well as engineering perspectives, paves the way for new discoveries that will have a significant impact on society at large.

Creating new customizable architectures and hardware acceleration platforms that

are able to efficiently perform iterative and communication intensive tasks. As the synopsis of my previous work suggests, the signal geometry, in terms of the ensemble of lower dimensional components, as well as the algorithmic properties, can be leveraged for finding the best customized domain-specific architecture. Benchmarking common applications on various computing fabrics including CPUs, CPU-GPUs, and CPU-FPGAs guides the formation of the best composition of accelerating components for each case. The constraints on the hardware system will be power, processor utilization, reduction in the data movement, and the memory bandwidth efficiency.

Building end-to-end computing systems for efficient deployment of various autonomous cyber-physical applications. Examples of such applications include but are not limited to sensor fusion, augmented reality, and autonomous sensing applications. My solutions shall provide a progressive computing system that leverages machine learning and customized hardware to autonomously evolve and adapt to the pertinent data and domain dynamics/constraints.

Devising assured and privacy-preserving machine learning systems. Machine learning systems should be designed such that they protect user information and privacy in the emerging Internet-of-Thing era and they are robust in the face of adversarial samples. Despite the plethora of work in the secure computing field, there is an inevitable need for the development of practical and scalable solutions that are amenable to resource-limited settings. The synopsis of my research work suggests that inter-domain optimization with insights from the hardware, data, and algorithms significantly reduce security protocols' overhead.

Bibliography

[AAB+15] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, and Matthieu Devin. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015. *Software available from tensorflow. org*, 1, 2015.

[ABC+18] Yossi Adi, Carsten Baum, Moustapha Cisse, Benny Pinkas, and Joseph Keshet. Turning your weakness into a strength: Watermarking deep neural networks by backdooring. *Usenix Security Symposium*, 2018.

[ABP+16] Dominik Auras, Sebastian Birke, Tobias Piwczyk, Rainer Leupers, and Gerd Ascheid. A flexible mcmc detector asic. In *SoC Design Conference (ISOCC), 2016 International*, pages 285–286. IEEE, 2016.

[ADFDJ03] Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I Jordan. An introduction to mcmc for machine learning. *Machine learning*, 50(1):5–43, 2003.

[AMW08] Narges Bani Asadi, Teresa H Meng, and Wing H Wong. Reconfigurable computing for learning bayesian networks. In *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 203–211. ACM, 2008.

[BAFG+10] Narges Bani Asadi, Christopher W Fletcher, Greg Gibeling, Eric N Glass, Karen Sachs, Daniel Burke, Zoey Zhou, John Wawrzynek, Wing H Wong, and Garry P Nolan. Paralearn: a massively parallel, scalable system for learning interaction networks on fpgas. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 83–94. ACM, 2010.

[Bai16] Baidu. Deepbench suit. 2016.

[BBB+10] James Bergstra, Olivier Breuleux, Guillaume Bastien, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, 2010.

[BDH14] Rémi Bardenet, Arnaud Doucet, and Chris Holmes. An adaptive subsampling approach for mcmc inference in large datasets. In *Proceedings of The 31st International Conference on Machine Learning*, pages 405–413, 2014.

[BGD16] Andrew L Beam, Sujit K Ghosh, and Jon Doyle. Fast hamiltonian monte carlo using gpu computing. *Journal of Computational and Graphical Statistics*, 25(2):536–548, 2016.

[BGJM11] Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng. *Handbook of Markov Chain Monte Carlo*. CRC press, 2011.

[BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 478–492. IEEE, 2013.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 784–796. ACM, 2012.

[BL12] Alexander Brant and Guy GF Lemieux. Zuma: An open fpga overlay architecture. In *Field-Programmable Custom Computing Machines*, pages 93–96. IEEE, 2012.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 503–513. ACM, 1990.

[BOP06] Mauro Barni, Claudio Orlandi, and Alessandro Piva. A privacy-preserving protocol for neural-network-based computation. In *Proceedings of the 8th workshop on Multimedia and security*, pages 146–151. ACM, 2006.

[Bot10] Léon Bottou. Large-scale machine learning with stochastic gradient descent. *Proceedings of COMPSTAT'2010*, pages 177–186, 2010.

[BR10] Leonard Bottolo and Sylvia Richardson. Evolutionary stochastic search for bayesian model exploration. *Bayesian Analysis*, 5(3):583–618, 2010.

[CDS+14] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM Sigplan Notices*, 49(4):269–284, 2014.

[CFG14] Tianqi Chen, Emily B Fox, and Carlos Guestrin. Stochastic gradient hamiltonian monte carlo. In *ICML*, pages 1683–1691, 2014.

[CGK10] Gregory Connor, Lisa R Goldberg, and Robert A Korajczyk. *Portfolio risk analysis*. Princeton University Press, 2010.

[CHM+15] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surfaces of multilayer networks. In *Artificial Intelligence and Statistics*, 2015.

[CHW+13] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In *International Conference on Machine Learning*, pages 1337–1345, 2013.

[CKLS97] Ingemar J Cox, Joe Kilian, F Thomson Leighton, and Talal Shamoon. Secure spread spectrum watermarking for multimedia. *IEEE transactions on image processing*, 6(12), 1997.

[CRK18] Huili Chen, Bita Darvish Rohani, and Farinaz Koushanfar. Deepmarks: A digital fingerprinting framework for deep neural networks. *arXiv preprint arXiv:1804.03648*, 2018.

[CSAK14] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, volume 14, pages 571–582, 2014.

[CW16] Nicholas Carlini and David Wagner. Defensive distillation is not robust to adversarial examples. *arXiv preprint*, 2016.

[CW17a] Nicholas Carlini and David Wagner. Magnet and" efficient defenses against adversarial attacks" are not robust to adversarial examples. *arXiv preprint arXiv:1711.08478*, 2017.

[CW17b] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy (SP)*, pages 39–57. IEEE, 2017.

[DCM+12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, and Quoc V Le. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[DDS+09] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[Die93] Francisco Javier Diez. Parameter adjustment in bayes networks. the generalized noisy or–gate. In *Uncertainty in Artificial Intelligence, 1993*, pages 99–105. Elsevier, 1993.

[DR13] Joan Daemen and Vincent Rijmen. The rijndael block cipher, 2013.

[DRGK18] Bita Darvish Rouhani, Mohammad Ghasemzadeh, and Farinaz Koushanfar. Causalearn: Automated framework for scalable streaming-based causal bayesian learning using fpgas. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 1–10. ACM, 2018.

[DSB13] Eva L Dyer, Aswin C Sankaranarayanan, and Richard G Baraniuk. Greedy feature selection for subspace clustering. *Journal of Machine Learning Research*, 14(1):2487–2517, 2013.

[DY14] Li Deng and Dong Yu. Deep learning: methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387, 2014.

[Efr17] Amir Efrati. How "deep learning" works at apple, beyond. `https://www.theinformation.com/How-Deep-Learning-Works-at-Apple-Beyond`, 2017.

[FIL17] Xin Fang, Stratis Ioannidis, and Miriam Leeser. Secure function evaluation using an fpga overlay architecture. In *FPGA*, pages 257–266, 2017.

[FK04] Borko Furht and Darko Kirovski. *Multimedia security handbook*. CRC press, 2004.

[FS11] Thomas Flury and Neil Shephard. Bayesian inference based only on simulated likelihood: particle filter analysis of dynamic economic models. *Econometric Theory*, 27(05):933–956, 2011.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[GBDL+16] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 201–210, 2016.

[GMP+17] Kathrin Grosse, Praveen Manoharan, Nicolas Papernot, Michael Backes, and Patrick McDaniel. On the (statistical) detection of adversarial examples. *arXiv preprint arXiv:1702.06280*, 2017.

[GR14] Shixiang Gu and Luca Rigazio. Towards deep neural network architectures robust to adversarial examples. *arXiv preprint arXiv:1412.5068*, 2014.

[GSS14] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[GVL12] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. John Hopkins University Press, 2012.

[HAR15] Uci machine learning repository. 2015.

[HJBB14] Clifford Hall, Weixiao Ji, and Estela Blaisten-Barojas. The metropolis monte carlo method with cuda enabled graphic processing units. *Journal of Computational Physics*, 258:871–879, 2014.

[HK99] Frank Hartung and Martin Kutter. Multimedia watermarking techniques. *Proceedings of the IEEE*, 87(7), 1999.

[HMD15] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[HMSG13] Nathaniel Husted, Steven Myers, Abhi Shelat, and Paul Grubbs. Gpu and cpu parallelization of honest-but-curious secure two-party computation. In *Computer Security Applications Conference*. ACM, 2013.

[HPTD15] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems (NIPS)*, 2015.

[HWSN12] Soren Henriksen, Adrian Wills, Thomas B Schön, and Brett Ninness. Parallel implementation of particle mcmc methods on a gpu. *IFAC Proceedings Volumes*, 45(16):1143–1148, 2012.

[Hyp15] Remote sensing. 2015.

[IHM+16] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *Crypto*, volume 2729, pages 145–161. Springer, 2003.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[JDJ01] Neil F Johnson, Zoran Duric, and Sushil Jajodia. *Information Hiding: Steganography and Watermarking-Attacks and Countermeasures: Steganography and Watermarking: Attacks and Countermeasures*, volume 1. Springer Science & Business Media, 2001.

[Jet15] Jetson tk1. 2015.

[JGD+14] Jonghoon Jin, Vinayak Gokhale, Aysegul Dundar, Bharadwaj Krishnamurthy, Berin Martini, and Eugenio Culurciello. An efficient implementation of deep convolutional neural networks on a mobile coprocessor. In *Circuits and Systems (MWSCAS), 2014 IEEE 57th International Midwest Symposium on*, pages 133–136. IEEE, 2014.

[JKSS10] Kimmo Järvinen, Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Garbled circuits for leakage-resilience: Hardware implementation and evaluation of one-time programs. In *CHES*, volume 10, pages 383–397. Springer, 2010.

[Jon14] Nicola Jones. The learning machines. *Nature*, 505(7482):146–148, 2014.

[KGB16] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533*, 2016.

[KH09] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[Kir16] Jeremy Kirk. Ibm join forces to build a brain-like computer. `http://www.pcworld.com/article/2051501/universities-join-ibm-in-cognitive-computing-researchproject.html`, 2016.

[Kno15] Eric Knorr. How paypal beats the bad guys with machine learning, 2015.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In *International Colloquium on Automata, Languages, and Programming*, pages 486–498. Springer, 2008.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[KSMB13] Ben Kreuter, Abhi Shelat, Benjamin Mood, and Kevin Butler. Pcf: A portable circuit format for scalable two-party secure computation. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 321–336, 2013.

[KSS14] Florian Kerschbaum, Thomas Schneider, and Axel Schröpfer. Automatic protocol selection in secure two-party computations. In *International Conference on Applied Cryptography and Network Security*. Springer, 2014.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553), 2015.

[LCB98] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998.

[LCB17] Yann LeCun, Corinna Cortes, and Christopher Burges. Mnist dataset. `http://yann.lecun.com/exdb/mnist/`, 2017.

[LH15] Ming Liang and Xiaolin Hu. Recurrent convolutional neural network for object recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[LJLA17] Jian Liu, Mika Juuti, Yao Lu, and N Asokan. Oblivious neural network predictions via MiniONN transformations. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[LLW10] Mingjie Lin, Ilia Lebedev, and John Wawrzynek. High-throughput bayesian computing machine with reconfigurable hardware. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 73–82. ACM, 2010.

[LMB15] Shuanglong Liu, Grigorios Mingas, and Christos-Savvas Bouganis. An exact mcmc accelerator under custom precision regimes. In *Field Programmable Technology (FPT), 2015 International Conference on*, pages 120–127. IEEE, 2015.

[LMB16] Shuanglong Liu, Grigorios Mingas, and Christos Bouganis. An unbiased mcmc fpga-based accelerator in the land of custom precision arithmetic. *IEEE Transactions on Computers*, 2016.

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT'07*, volume 4515 of *LNCS*, pages 52–78. Springer, 2007.

[LP12] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *Journal of cryptology*, 25(4):680–722, 2012.

[LTA16] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning (ICML)*, 2016.

[Lu04] Chun-Shien Lu. *Multimedia Security: Steganography and Digital Watermarking Techniques for Protection of Intellectual Property: Steganography and Digital Watermarking Techniques for Protection of Intellectual Property*. Igi Global, 2004.

[MA14] Dougal Maclaurin and Ryan P Adams. Firefly monte carlo: Exact mcmc with subsets of data. *arXiv preprint arXiv:1403.5693*, 2014.

[MB12] Grigorios Mingas and Christos-Savvas Bouganis. A custom precision based architecture for accelerating parallel tempering mcmc on fpgas without introducing sampling error. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 153–156. IEEE, 2012.

[MB16] Grigorios Mingas and Christos-Savvas Bouganis. Population-based mcmc on multi-core cpus, gpus and fpgas. *IEEE Transactions on Computers*, 65(4):1283–1296, 2016.

[MC17] Dongyu Meng and Hao Chen. Magnet: a two-pronged defense against adversarial examples. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 135–147. ACM, 2017.

[MCF15] Yi-An Ma, Tianqi Chen, and Emily Fox. A complete recipe for stochastic gradient mcmc. In *Advances in Neural Information Processing Systems*, pages 2917–2925, 2015.

[MDFF16] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2574–2582, 2016.

[mlr17a] UCI machine learning repository. `https://archive.ics.uci.edu/ml/datasets/isolet`, 2017.

[mlr17b] UCI machine learning repository. `https://archive.ics.uci.edu/ml/datasets/Daily+and+Sports+Activities`, 2017.

[MMK+15] Takeru Miyato, Shin-ichi Maeda, Masanori Koyama, Ken Nakae, and Shin Ishii. Distributional smoothing with virtual adversarial training. *arXiv preprint arXiv:1507.00677*, 2015.

[Mol16] David Moloney. Embedded deep neural networks:the cost of everything and the value of nothing. In *2016 IEEE Hot Chips 28 Symposium (HCS)*, pages 1–20. IEEE, 2016.

[MPC16] Patrick McDaniel, Nicolas Papernot, and Z Berkay Celik. Machine learning in adversarial settings. *IEEE Security & Privacy*, 14(3):68–72, 2016.

[MPT17] Erwan Le Merrer, Patrick Perez, and Gilles Trédan. Adversarial frontier stitching for remote neural network watermarking. *arXiv preprint arXiv:1711.01894*, 2017.

[MRB13] Grigorios Mingas, Farhan Rahman, and Christos-Savvas Bouganis. On optimizing the arithmetic precision of mcmc algorithms. In *Field-Programmable Custom Computing Machines (FCCM), 21st Annual International Symposium on*, pages 181–188. IEEE, 2013.

[MRSK16] Azalia Mirhoseini, Bita Darvish Rouhani, Ebrahim M Songhori, and Farinaz Koushanfar. Perform-ml: Performance optimized machine learning by platform and content aware customization. In *Proceedings of the 53rd Annual Design Automation Conference*, page 20. ACM, 2016.

[MvdWN+17] Alexander G de G Matthews, Mark van der Wilk, Tom Nickson, Keisuke Fujii, Alexis Boukouvalas, Pablo León-Villagrá, Zoubin Ghahramani, and James Hensman. Gpflow: A gaussian process library using tensorflow. *Journal of Machine Learning Research*, 18(40):1–6, 2017.

[MZ17] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. *IACR Cryptology ePrint Archive*, 2017:396, 2017.

[NIW+13] Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. Privacy-preserving matrix factorization. In *Conference on Computer & communications security*. ACM, 2013.

[NO09] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In *TCC'09*, volume 5444 of *LNCS*, pages 368–386. Springer, 2009.

[NP05] Moni Naor and Benny Pinkas. Computationally secure oblivious transfer. *Journal of Cryptology*, 18(1):1–35, 2005.

[NPS99] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM conference on Electronic commerce*, pages 129–139. ACM, 1999.

[NUSS18] Yuki Nagai, Yusuke Uchida, Shigeyuki Sakazawa, and Shinichi Satoh. Digital watermarking for deep neural networks. *International Journal of Multimedia Information Retrieval*, 7(1), 2018.

[NWI+13] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *Symposium on S & P*. IEEE, 2013.

[NWX13] Willie Neiswanger, Chong Wang, and Eric Xing. Asymptotically exact, embarrassingly parallel mcmc. *arXiv preprint arXiv:1311.4780*, 2013.

[OPB07] Claudio Orlandi, Alessandro Piva, and Mauro Barni. Oblivious neural network computing via homomorphic encryption. *EURASIP Journal on Information Security*, 2007:18, 2007.

[PDL11] Shi Pu, Pu Duan, and Jyh-Charn Liu. Fastplay-a parallelization model and implementation of smc on cuda based gpu cluster architecture. *IACR Cryptology ePrint Archive*, 2011.

[PMW+16] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. *IEEE Symposium on Security and Privacy (SP)*, pages 582–597, 2016.

[PNB15] Ankit B Patel, Tan Nguyen, and Richard G Baraniuk. A probabilistic theory of deep learning. *arXiv preprint arXiv:1504.00641*, 2015.

[POC+08] Alessandro Piva, Claudio Orlandi, M Caini, Tiziano Bianchi, and Mauro Barni. Enhancing privacy in remote data classification. In *IFIP International Information Security Conference*, pages 33–46. Springer, 2008.

[Pro17] Intel Processors. `http://www.velocitymicro.com/blog/xeon-vs-i7i5-whats-difference/`, 2017.

[QP07] Gang Qu and Miodrag Potkonjak. *Intellectual property protection in VLSI designs: theory and practice*. Springer Science & Business Media, 2007.

[Ras04] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Advanced lectures on machine learning*, pages 63–71. Springer, 2004.

[RGC15] Mauro Ribeiro, Katarina Grolinger, and Miriam AM Capretz. Mlaas: Machine learning as a service. In *IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, 2015.

[RMK16] Bita Darvish Rouhani, Azalia Mirhoseini, and Farinaz Koushanfar. Delight: Adding energy dimension to deep neural networks. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*. ACM, 2016.

[RMK17a] Bita Darvish Rouhani, Azalia Mirhoseini, and Farinaz Koushanfar. Deep3: Leveraging three levels of parallelism for efficient deep learning. In *Proceedings of ACM 54th Annual Design Automation Conference (DAC)*, 2017.

[RMK17b] Bita Darvish Rouhani, Azalia Mirhoseini, and Farinaz Koushanfar. Deep3: Leveraging three levels of parallelism for efficient deep learning. In *Proceedings of the 54rd Annual Design Automation Conference*. ACM, 2017.

[RMK17c] Bita Darvish Rouhani, Azalia Mirhoseini, and Farinaz Koushanfar. Rise: An automated framework for real-time intelligent video surveillance on fpga. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):158, 2017.

[RMK17d] Bita Darvish Rouhani, Azalia Mirhoseini, and Farinaz Koushanfar. Tinydl: Just-in-time deep learning solution for constrained embedded systems. In *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on*, pages 1–4. IEEE, 2017.

[RMSK16] Bita Darvish Rouhani, Azalia Mirhoseini, Ebrahim M Songhori, and Farinaz Koushanfar. Automated real-time analysis of streaming big and dense data on reconfigurable platforms. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 10(1):8, 2016.

[RRK18] Bita Darvish Rouhani, M Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. *Design Automation Conference (DAC)*, 2018.

[RSJ+18] Bita Rouhani, Mohammad Samragh, Mojan Javaheripi, Tara Javidi, and Farinaz Koushanfar. Deepfense: Online accelerated defense against adversarial deep learning. *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018.

[RSJK18a] Bita Rouhani, Mohammad Samragh, Tara Javidi, and Farinaz Koushanfar. Safe machine learning and defeating adversarial attacks. *IEEE Security and Privacy (S&P) Magazine*, 2018.

[RSJK18b] Bita Darvish Rouhani, Mohammad Samragh, Tara Javidi, and Farinaz Koushanfar. Safe machine learning and defeat-ing adversarial attacks. *IEEE Security and Privacy (S&P) Magazine*, 2018.

[RSMK15] Bita Darvish Rouhani, Ebrahim M Songhori, Azalia Mirhoseini, and Farinaz Koushanfar. Ssketch: An automated framework for streaming sketch-based analysis of big data on fpga. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 187–194. IEEE, 2015.

[RSN+01] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, and Elaine Barker. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical Report 800-22, NIST, 2001.

[SDB+17] Umut Simsekli, Alain Durmus, Roland Badeau, Gaël Richard, Eric Moulines, and Taylan Cemgil. Parallelized stochastic gradient markov chain monte carlo algorithms for non-negative matrix factorization. In *42nd International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017.

[SDBR14] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[SGK17] Mohammad Samragh, Mohammad Ghasemzadeh, and Farinaz Koushanfar. Customizing neural networks for efficient fpga implementation. In *IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[SHS+15] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy*, pages 411–428. IEEE, 2015.

[SJGZ17] Shiwei Shen, Guoqing Jin, Ke Gao, and Yongdong Zhang. Ape-gan: Adversarial perturbation elimination with gan. *ICLR Submission, available on OpenReview*, 2017.

[SLWW17] Zhourui Song, Zhenyu Liu, Chunlu Wang, and Dongsheng Wang. Computation error analysis of block floating point arithmetic oriented convolution neural network accelerator design. *arXiv preprint arXiv:1709.07776*, 2017.

[SMA07] Antony W Savich, Medhat Moussa, and Shawki Areibi. The impact of arithmetic representation on implementing mlp-bp on fpgas: A study. *Neural Networks, IEEE Transactions on*, 18(1):240–252, 2007.

[SMH07] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted boltzmann machines for collaborative filtering. In *Proceedings of the 24th international conference on Machine learning*, pages 791–798. ACM, 2007.

[SPA+16] Hardik Sharma, Jongse Park, Emmanuel Amaro, Bradley Thwaites, Praneetha Kotha, Anmol Gupta, Joon Kyung Kim, Asit Mishra, and Hadi Esmaeilzadeh. Dnnweaver: From high-level deep network models to fpga acceleration. In *The Workshop on Cognitive Architectures*, 2016.

[SS11] Abhi Shelat and Chih-hao Shen. Two-output secure computation with malicious adversaries. In *EUROCRYPT'11*, volume 6632 of *LNCS*, pages 386–405. Springer, 2011.

[SYN15] Uri Shaham, Yutaro Yamada, and Sahand Negahban. Understanding adversarial training: Increasing local stability of neural nets through robust optimization. *arXiv preprint arXiv:1511.05432*, 2015.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[SZD+16] Ebrahim M Songhori, Shaza Zeitouni, Ghada Dessouky, Thomas Schneider, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. Garbledcpu: a mips processor for secure computation in hardware. In *Proceedings of the 53rd Annual Design Automation Conference (DAC)*, page 73. ACM, 2016.

[SZS+13] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[TG07] Joel Tropp and Anna C Gilbert. Signal recovery from random measurements via orthogonal matching pursuit. *IEEE Transactions on Information Theory*, 53(12):4655–4666, 2007.

[THL11] Matthew M Tibbits, Murali Haran, and John C Liechty. Parallel multivariate slice sampling. *Statistics and Computing*, 21(3):415–430, 2011.

[TKD+16] Dustin Tran, Alp Kucukelbir, Adji B Dieng, Maja Rudolph, Dawen Liang, and David M Blei. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*, 2016.

[TLR08] Michalis K Titsias, Neil Lawrence, and Magnus Rattray. Markov chain monte carlo algorithms for gaussian processes. *Inference and Estimation in Probabilistic Time-Series Models*, 9, 2008.

[TSG+16] Nima Tajbakhsh, Jae Y Shin, Suryakanth R Gurudu, R Todd Hurst, Christopher B Kendall, Michael B Gotway, and Jianming Liang. Convolutional neural networks for medical image analysis: Full training or fine tuning? *IEEE transactions on medical imaging*, 35(5), 2016.

[UCI16a] UCI Machine Learning Repository. `https://archive.ics.uci.edu/ml/datasets/Dow+Jones+Index`. 2016.

[UCI16b] UCI Machine Learning Repository. `https://archive.ics.uci.edu/ml/datasets/MHEALTH+Dataset`. 2016.

[UNSS17] Yusuke Uchida, Yuki Nagai, Shigeyuki Sakazawa, and Shin'ichi Satoh. Embedding watermarks into deep neural networks. In *Proceedings of the ACM on International Conference on Multimedia Retrieval*, 2017.

[VRH+13] Jarno Vanhatalo, Jaakko Riihimäki, Jouni Hartikainen, Pasi Jylänki, Ville Tolvanen, and Aki Vehtari. Gpstuff: Bayesian modeling with gaussian processes. *Journal of Machine Learning Research*, 14(Apr):1175–1179, 2013.

[VW17] Javier Alejandro Varela and Norbert Wehn. Near real-time risk simulation of complex portfolios on heterogeneous computing systems with opencl. In *International Workshop on OpenCL*. ACM, 2017.

[WGMK16] Xiao Wang, S Dov Gordon, Allen McIntosh, and Jonathan Katz. Secure computation of mips machine code. In *ESORICS*. Springer, 2016.

[WHC+14] Xiao Shaun Wang, Yan Huang, TH Hubert Chan, Abhi Shelat, and Elaine Shi. Scoram: oblivious ram for secure computation. In *CCS*. ACM, 2014.

[WT09] Knut Wold and Chik How Tan. Analysis and enhancement of random number generator in fpga based on oscillator rings. *International Journal of Reconfigurable Computing*, 2009:4, 2009.

[WT11] Max Welling and Yee W Teh. Bayesian learning via stochastic gradient langevin dynamics. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 681–688, 2011.

[XIL17] XILLYBUS. `http://xillybus.com/`, 2017.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162–167. IEEE, 1986.

[YCS16] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. *arXiv preprint arXiv:1611.05128*, 2016.

[YHC+18] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Vivienne Sze, and Hartwig Adam. Netadapt: Platform-aware neural network adaptation for mobile applications. *arXiv preprint arXiv:1804.03230*, 2018.

[ZK16] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[ZLS+15] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.

[ZNR17] Valentina Zantedeschi, Maria-Irina Nicolae, and Ambrish Rawat. Efficient defenses against adversarial attacks. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 39–49. ACM, 2017.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 220–250. Springer, 2015.

[ZWS+16] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. Energy-efficient cnn implementation on a deeply pipelined fpga cluster. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pages 326–331. ACM, 2016.