

Old Dominion University

ODU Digital Commons

Electrical & Computer Engineering Theses & Dissertations

Electrical & Computer Engineering

Fall 12-2021

Joint Linear and Nonlinear Computation with Data Encryption for Efficient Privacy-Preserving Deep Learning

Qiao Zhang

Old Dominion University, qzhan002@odu.edu

Follow this and additional works at: https://digitalcommons.odu.edu/ece_etds



Part of the [Artificial Intelligence and Robotics Commons](#), and the [Computer Engineering Commons](#)

Recommended Citation

Zhang, Qiao. "Joint Linear and Nonlinear Computation with Data Encryption for Efficient Privacy-Preserving Deep Learning" (2021). Doctor of Philosophy (PhD), Dissertation, Electrical & Computer Engineering, Old Dominion University, DOI: 10.25777/6hgt-cv97
https://digitalcommons.odu.edu/ece_etds/230

This Dissertation is brought to you for free and open access by the Electrical & Computer Engineering at ODU Digital Commons. It has been accepted for inclusion in Electrical & Computer Engineering Theses & Dissertations by an authorized administrator of ODU Digital Commons. For more information, please contact digitalcommons@odu.edu.

**JOINT LINEAR AND NONLINEAR COMPUTATION WITH
DATA ENCRYPTION FOR EFFICIENT
PRIVACY-PRESERVING DEEP LEARNING**

by

Qiao Zhang

B.Sc. June 2014, Chongqing University of Posts and Telecommunications

M.Sc. June 2017, Chongqing University of Posts and Telecommunications

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

ELECTRICAL & COMPUTER ENGINEERING

OLD DOMINION UNIVERSITY

December 2021

Approved by:

Hongyi Wu (Director)

Chunsheng Xin (Member)

Jiang Li (Member)

Sachin Shetty (Member)

ABSTRACT

JOINT LINEAR AND NONLINEAR COMPUTATION WITH DATA ENCRYPTION FOR EFFICIENT PRIVACY-PRESERVING DEEP LEARNING

Qiao Zhang
Old Dominion University, 2021
Director: Dr. Hongyi Wu

Deep Learning (DL) has shown unrivalled performance in many applications such as image classification, speech recognition, anomalous detection, and business analytics. While end users and enterprises own enormous data, DL talents and computing power are mostly gathered in technology giants having cloud servers. Thus, data owners, i.e., the clients, are motivated to outsource their data, along with computationally-intensive tasks, to the server in order to leverage the server’s abundant computation resources and DL talents for developing cost-effective DL solutions. However, trust is required between the server and the client to finish the computation tasks (e.g., conducting inference for the newly-input data from the client, based on a well-trained model at the server) otherwise there could be the data breach (e.g., leaking data from the client or the proprietary model parameters from the server).

Privacy-preserving DL takes data privacy into account where various data-encryption based techniques are adopted. However, the efficiency of linear and nonlinear computation for each DL layer remains a fundamental challenge in practice due to the intrinsic intractability and complexity of privacy-preserving primitives (e.g., Homomorphic Encryption (HE) and Garbled Circuits (GC)). As such, this dissertation targets deeply optimizing state-of-the-art frameworks as well as newly designing efficient modules by joint linear and nonlinear computation, with data encryption, to further boost the overall performance of privacy-preserving DL. Four contributions are made.

First, deep optimization on the HE-based linear computation in HE-GC-based privacy-preserving DL inference, GALA, is presented that features a row-wise weight matrix encoding, a combination of the share generation and a first-Add-second-Perm approach to reduce the most expensive permutation operations. GALA demonstrates an inference runtime boost by $2.5\times$ to $8.3\times$ over the state-of-the-art frameworks.

Second, the GC-based nonlinear calculation is replaced with a newly-designed joint linear and non-linear computation for each DL layer, in HE-GC-based privacy-preserving DL inference, based on the Homomorphic Secret Sharing. This construction achieves an inference speedup as high as $48\times$ compared with state-of-the-art frameworks.

Third, the nonlinear calculation of each layer is completed for free by a carefully partitioned DL framework, GELU-Net, where the server performs linear computation on encrypted data utilizing a less complex homomorphic cryptosystem, while the client securely executes non-polynomial computation in plaintext without approximation. GELU-Net demonstrates $14\times$ to $35\times$ inference speedup compared to the classic systems.

Finally, we propose the WISE framework to completely eliminate the most expensive HE permutation in HE-based linear calculation and reduce the communication cost from 4.5 rounds to only half round by a joint permutation-free computation between the nonlinear transformation in the current layer and the linear transformation in the next layer. WISE achieves $2\times$ to $13\times$ speedup over one of the most recent works through various widely-adopted neural layers and demonstrates a speedup up to $5.3\times$ on practical DL models.

Copyright, 2021, by Qiao Zhang, All Rights Reserved.

ACKNOWLEDGEMENTS

I cannot accomplish my Ph.D. degree without a great deal of support and assistance from many people. I would like to take a moment to thank them as they helped me to keep persistent and deal with problems in my research and daily life. This precious kindness lets me learn a lot along the way on this journey.

First and foremost, I would like to express the special appreciation to my advisor, Dr. Hongyi Wu, for providing me with graduate scholarship and guiding me into the research topic about privacy-preserving deep learning. Dr. Wu gave me enough freedom to explore what I was interested in and instructed me to think critically and write elegantly. Meanwhile, Dr. Wu always got around to discussing with me whenever I had some questions or ideas and he also encouraged me to keep myself on the right path of my destination when I got stuck. I am honored to have such a kind and patient person as my Ph.D. advisor.

Then I want to give the sincere gratitude to my co-advisor, Dr. Chunsheng Xin, for helping me with countless technical discussions and guidance. I also want to thank Dr. Cong Wang for the research discussion and help at my early study at ODU. Meanwhile, I appreciate the committee members for their valuable time and consideration to review my dissertation. Furthermore, I would like to give my special thanks to my roommates and labmates for their kind help in my daily life at Norfolk, and I am thankful to people in the ODU community for extending their helping hands when I needed.

Last but not least, I wish to acknowledge the great love of my grandparents, mother, and sweet sister. Studying abroad as an international student was challenging for me as everything came from scratch, but they always acted as my strongest backend and supported me through the whole journey. They always encouraged and listened to me whenever I had difficulties, and I would never finish my Ph.D. study without their endless and unconditioned care and love.

TABLE OF CONTENTS

	Page
LIST OF TABLES	viii
LIST OF FIGURES	x
Chapter	
1. INTRODUCTION	1
1.1 PRIVACY-PRESERVING DEEP LEARNING (PPDL)	2
1.2 CHALLENGES IN PPDL	6
1.3 DISSERTATION CONTRIBUTIONS	8
1.4 ORGANIZATION OF THE DISSERTATION	10
2. PRELIMINARIES	12
2.1 SYSTEM MODELS	12
2.2 THREAT MODEL	15
2.3 PRIVACY-PRESERVING TOOLS	17
2.4 CHAPTER SUMMARY	20
3. GREEDY COMPUTATION FOR LINEAR ALGEBRA IN PPDL	21
3.1 MOTIVATION	22
3.2 PRELIMINARIES	26
3.3 SYSTEM DESCRIPTION	28
3.4 SECURITY ANALYSIS	43
3.5 PERFORMANCE EVALUATION	43
3.6 CHAPTER SUMMARY	56
4. OBSCURE COMPUTATION WITH PIGGYBACK PROPAGATION IN PPDL	57
4.1 MOTIVATION	57
4.2 PRELIMINARIES	60
4.3 SYSTEM DESCRIPTION	63
4.4 SECURITY ANALYSIS	78
4.5 PERFORMANCE EVALUATION	80
4.6 CHAPTER SUMMARY	89
5. GLOBALLY ENCRYPTED AND LOCALLY UNENCRYPTED PPDL	90
5.1 MOTIVATION	90
5.2 PRELIMINARIES	92
5.3 SYSTEM DESCRIPTION	92
5.4 SECURITY ANALYSIS	101
5.5 PERFORMANCE EVALUATION	102

Chapter	Page
5.6 CHAPTER SUMMARY	106
6. PERMUTATION ELIMINATION AND OT REDUCTION IN PPDL	107
6.1 MOTIVATION	107
6.2 PRELIMINARIES	110
6.3 SYSTEM DESCRIPTION	111
6.4 SECURITY ANALYSIS	127
6.5 PERFORMANCE EVALUATION	130
6.6 CHAPTER SUMMARY	135
7. CONCLUSIONS AND FUTURE WORKS	138
BIBLIOGRAPHY	143
APPENDICES	
A. SUPPLEMENTARY MATERIALS FOR SECURETRAIN	166
A.1 CONSTRUCTION OF (2,2)-THRESHOLD SCHEME	166
A.2 BACK PROPAGATION EXCEPT LAST LAYER	166
A.3 COMPLEXITY ANALYSIS	168
A.4 SECURITY AGAINST A SEMI-HONEST SERVER	171
VITA	173

LIST OF TABLES

Table	Page
1. Data-encryption Based Techniques used in PDDL.	5
2. Quantitative Evaluation for Different PDDL Schemes.	7
3. Cost of matrix-vector multiplication (time in millionsecond).	25
4. Complexity comparison of three methods.	36
5. Complexity comparison of convolution.	42
6. Comparison of noise management.	43
7. Computation complexity of matrix-vector multiplication.	46
8. Runtime cost of matrix-vector multiplication.	47
9. Computation complexity of convolution.	48
10. Runtime cost of convolution.	49
11. Computation complexity of state-of-the-art neural network models.	50
12. Runtime cost of classic model.	51
13. Runtime cost of state-of-the-art models.	52
14. Percentages of linear computation in state-of-the-art neural network models. . . .	53
15. Accuracy with floating and fixed point in state-of-the-art neural network models. Top-1 accuracy: only the prediction with the highest probability is a true label; Top-5 accuracy: any one of the five model predictions with higher probability is a true label.	53
16. Key notations.	60
17. Computation complexity.	78
18. Communication complexity in each part.	78
19. Inference performance comparison.	80
20. Comparison of training time.	88
21. Computation time of activation in different schemes (ms).	94
22. Complexity comparison (per iteration).	101
23. Comparison of accuracy.	104
24. Computation speed in different networks (s).	105
25. GELU-Net speed in different environments (s).	105
26. Running time and communication cost of convolution layers.	130
27. Running time and communication cost on modern DL models.	131

LIST OF FIGURES

Figure	Page
1. PPDL: (a) Privacy-preserving MLaaS, (b) Privacy-preserving training.	4
2. Overview of Works in the Dissertation.	9
3. Basic structure of CNN.	13
4. Simulator for the (a) corrupted client and (b) corrupted server in MLaaS.	17
5. Naive matrix-vector multiplication.	30
6. Hybrid matrix-vector multiplication.	32
7. Row-encoding-share-RaS multiplication.	34
8. SISO convolution.	37
9. MIMO convolution.	39
10. Kernel grouping based MIMO convolution.	40
11. Layer-wise accumulated runtime and GALA speedup over GAZELLA on different networks: (a) AlexNet; (b) VGG; (c) ResNet-18; (d) ResNet-50; (e) ResNet-101; (f) ResNet-152. The bar with values on the left y-axis indicates speedup, and the curve with values on the right y-axis indicates the accumulated runtime. The layers with speedup of 1 are nonlinear layers.	54
12. A framework of privacy-preserving inference.	58
13. Forward propagation.	65
14. Softmax calculation.	70
15. Back propagation diagram.	72
16. Performance comparison for ReLU calculation: (a) Runtime and (b) Communication cost with different output dimensions.	81
17. Comparison of the last layer output of different approximations over 10 epochs: (a) Piecewise linear approximation [1]; (b) Maclaurin approximation [2]; (c) ReLU based softmax approximation [3,4]; (d) ReLU based sigmoid approximation [5]; (e) Polynomial based sigmoid approximation [6]; (f) Non-approximation in SecureTrain.	81
18. Training loss and testing accuracy: (a) Loss during training and (b) Testing accuracy with different approximation approaches. The softmax is non-approximation in SecureTrain while [3,4] are with ReLU based softmax approximation, [2] is with Maclaurin approximation, [1] is with Piecewise linear approximation, [6] is with Polynomial based sigmoid approximation, and [5] is with ReLU based sigmoid approximation.	83
19. Probability density distribution of training loss with different approximations: (a) Non approximation in SecureTrain; (b) ReLU based softmax approximation [3,4]; (c) Maclaurin approximation [2]; (d) Piecewise linear approximation [1]; (e) Polynomial based sigmoid approximation [6]; (f) ReLU based sigmoid approximation [5].	86

20.	Probability density distribution of accuracy with different approximations: (a) Non-approximation in SecureTrain; (b) ReLU based softmax approx. [3, 4]; (c) Maclaurin approx. [2]; (d) Piecewise linear approx. [1]; (e) Polynomial based sigmoid approx. [6]; (f) ReLU based sigmoid approx. [5].	86
21.	Output probability distribution of each approach with different network structures in terms of the number of hidden neurons: (a) Piecewise linear approximation [1]; (b) Maclaurin approximation [2]; (c) ReLU based softmax approximation [3, 4]; (d) ReLU based sigmoid approximation [5]; (e) Polynomial based sigmoid approximation [6]; (f) Non-approximation in SecureTrain.	87
22.	Training loss of different network structures under different approximations: (a) Piecewise linear approximation [1]; (b) Maclaurin approximation [2]; (c) ReLU based softmax approximation [3, 4]; (d) ReLU based sigmoid approximation [5]; (e) Polynomial based sigmoid approximation [6]; (f) Non-approximation in SecureTrain.	87
23.	Testing accuracy of different network structures under different approximations: (a) Piecewise linear approximation [1]; (b) Maclaurin approximation [2]; (c) ReLU based softmax approximation [3, 4]; (d) ReLU based sigmoid approximation [5]; (e) Polynomial based sigmoid approximation [6]; (f) Non-approximation in SecureTrain.	88
24.	Server reconstructs training data via activations.	97
25.	Server reconstruction with/without gradient protection.	103
26.	Training stability of different schemes.	104
27.	WISE protocol compared with CryptFlow2 [7].	109
28.	Data transformation for convolution.	112
29.	Perm-free convolution.	115
30.	Data transformation for dot product.	121
31.	Comparison of layer-wise computation flow.	124
32.	Layer-wise accumulated running time and WISE speedup over CryptFlow2 on different networks: (a) and (b) LeNet; (c) and (d) AlexNet; (e) and (f) VGG-11; (g) and (h) VGG-13; (i) and (j) VGG-16; (k) and (l) VGG-19; (m) and (n) ResNet-18; (o) and (p) ResNet-34. The bar with values on the left y-axis indicates speedup in log scale, and the curve with values on the right y-axis indicates the accumulated running time. The layers with speedup of 1 are pooling layers.	136
33.	Layer-wise accumulated communication cost (in log scale) on different networks: (a) LeNet; (b) AlexNet; (c) VGG-11; (d) VGG-13; (e) VGG-16; (f) VGG-19; (g) ResNet-18; (h) ResNet-34.	137

CHAPTER 1

INTRODUCTION

Large-volume data is generated in daily life due to the rapid evolution and utilization of information technology such as the Internet of Things (IoT) [8, 9]. Global data creation is projected to grow to more than 180 zettabytes by 2025 [10], which has become a driving force for the development of Deep Learning (DL) technologies [11–14]. DL has shown its unrivalled performance in many applications such as image classification [15, 16], speech recognition [17–19], anomalous detection [20–22], and business analytics [23, 24]. The success of DL largely depends on three key ingredients: massive amount of high-quality training data, high-performance computation resources and well-designed model structures [25, 26]. These ingredients are often owned by different entities. For instance, end users and enterprises possess enormous data, while DL talents and computing power are mostly gathered in technology giants such as Google, Facebook and Microsoft. Data owners are motivated to outsource their data, along with computationally intensive tasks, to the clouds (e.g., Microsoft Azure and Google Cloud) in order to leverage abundant resources and DL talents for developing cost-effective DL solutions.

DL includes the training phase and inference phase where the former utilizes the training data to produce a well-trained model, based on which the latter conducts the prediction for newly-input data. For example, assume Alice owns her data, while Bob owns the cloud platform. In the training phase, Alice would like to have her data processed by Bob to create a well-trained DL model. In the inference phase, Bob, the cloud server with the trained model and computational resources, performs predictions for clients is often known as Machine Learning as a Service (MLaaS) [27, 28].

However, both phases require trust between the (cloud) servers (i.e., service providers) and clients (i.e., data owners). Due to different trust domains, privacy issues arise from

exposure to servers the private information in the outsourced data and from the intermediate data through the interactions between clients and servers. Both involve data breaches corresponding to either the original data or model parameters. There are various instances where the original data or model parameters should not be public [29–32]. As the data breach becomes a critical concern, more and more governments have established regulations for protecting users’ data, such as General Data Protection Regulation (GDPR) in the European Union [33], Personal Data Protection Act (PDPA) in Singapore [34], California Consumer Privacy Act (CCPA) [35] and the Health Insurance Portability and Accountability Act (HIPAA) [36] in the US. The cost of the data breach is high. For instance, in the breach of 600,000 drivers’ personal data in 2016, Uber paid \$148 million to settle the investigation [37]; SingHealth was fined about \$750,000 by the Singapore government for a breach of PDPA data [38]; Google was fined about \$57 million for a breach of GDPR data [39], which is the largest penalty as of March 18, 2020 under the European Union privacy law.

Besides the protection for processed (usually personal) data (from clients), the servers that created/computed the DL models in MLaaS, also wish not to make these highly-valued model parameters publicly available. By releasing the parameters of their DL models, the servers are worried about losing their intellectual property. Furthermore, DL models are shown to memorize information about their training data [40]. In particular, the parameters of DL models could lead to exposure of training data [41], which are considered confidential in many cases.

1.1 PRIVACY-PRESERVING DEEP LEARNING (PPDL)

Privacy-Preserving Deep Learning (PPDL) takes ethical and legal privacy concerns into account, and various privacy-preserving techniques are adopted in the computation process of DL in order to protect sensitive data (e.g., private data from the client and model parameters from the server in MLaaS). In this dissertation, we focus on multi-party privacy-preserving

techniques based on data encryption, which are widely studied in PPDL as shown in Table 1¹. Specifically, the Homomorphic Encryption (HE) [48–53], Garbled Circuits (GC) [54–58], Oblivious Transfer (OT) [59, 60] and Secret Sharing (SS) [61–63] are four dominant data-encryption based techniques². Meanwhile, there are two dominant privacy-preserving DL scenarios in multi-party computation, i.e., data-encryption based privacy-preserving MLaaS and training. In data-encryption based privacy-preserving MLaaS as shown in Figure 1 (a), the data owner is the client and the cloud server has a well-trained deep learning model. The cloud server provides the inference service based on the private data from the client. For example, an encrypted medical image (such as a chest X-ray) is sent by a doctor to the cloud server, which runs the DL model and returns the encrypted prediction to the doctor. The prediction is then decrypted by the doctor into a plaintext result to assist diagnosis and health care planning. In this case, besides the protection of the outsourced data from clients, it is required that neither the clients nor any other parties learn anything about the model parameters at the cloud servers, other than the final predictions. In this dissertation, we mainly focus on privacy-preserving MLaaS.

As for data-encryption based privacy-preserving training shown in Figure 1 (b), multiple clients securely outsource their partial data and jointly communicate with the cloud server to obtain a well-trained model. For instance, multiple health care providers (such as hospitals) send their encrypted medical data (such as the chest X-ray) to the cloud server, and the final model is obtained by interactions among them. Health care providers can utilize the more comprehensive model to improve accuracy. The cloud servers or any other involved parties should not learn anything about clients’ outsourced data other than its size. Meanwhile, the updated model parameters are supposed to be respectively blind to data owners and cloud servers [4] during the training process.

¹There are another two branches of research for privacy-preserving DL with Differential Privacy (DP) [42–45] and Trusted Execution Environment (TEE) [46, 47] which mainly deal with user data and computation in plaintext and we make it orthogonal to the dissertation scope as we mainly focus on privacy-preserving DL with various data-encryption based techniques.

²The detailed introduction about these four techniques is described in Section 2.1 of Chapter 2.

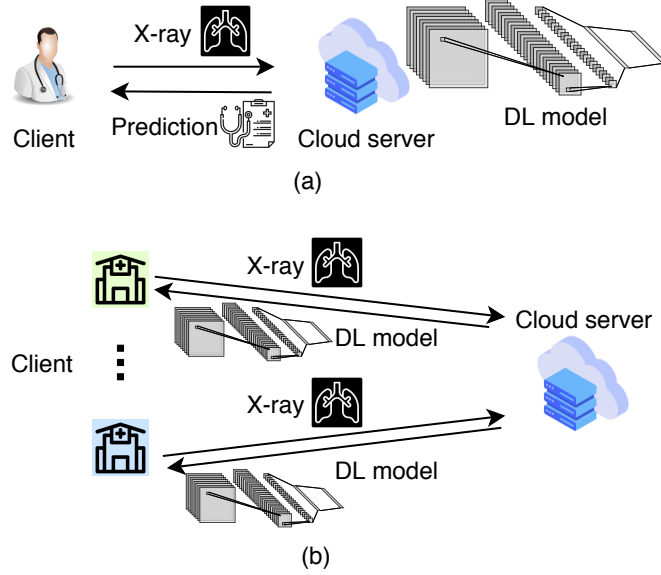


Figure 1. PPDL: (a) Privacy-preserving MLaaS, (b) Privacy-preserving training.

The DL models are cascaded with layers, each of which features linear and nonlinear computation. In this dissertation, we mainly focus on the Convolutional Neural Network (CNN), which is widely used in many PPDL frameworks [4, 64–66]. The linear computation in CNN includes dot product and convolution while the nonlinear computation contains various activation functions³. Either a single technique, i.e., single computation in Table 1, or a combination of techniques, i.e., hybrid computation in Table 1, is adopted in the two building-block computation to realize PPDL.

Among the privacy-preserving techniques shown in Table 1, single-HE schemes [6, 77–85] need no extra trusted party while they need to approximate the nonlinear functions into polynomials. Single-SS schemes [67–73] generally involve more parties, e.g., an extra trusted party or multiple non-colluding cloud servers, and a larger number of interactions, e.g., communication round among the parties. Single-GC schemes [74–76] aim to minimize the non-XOR gates in the constructed Boolean circuits or to adopt more GC-friendly models such as Binary Neural Networks [76].

³The detailed introduction for the CNN model is described in Section 2.1 of Chapter 2.

Table 1. Data-encryption Based Techniques used in PPDL.

Schemes	Data-encryption Based Techniques		Number of parties		
	Single computation	Hybrid computation	2	3	≥ 4
Liu [67]	SS	○	○	●	○
Falcon [68]	SS	○	○	●	○
Swift [69]	SS	○	○	●	●
Securenn [70]	SS	○	○	●	○
Trident [71]	SS	○	○	○	●
Huang [72]	SS	○	○	●	○
Leia [73]	SS	○	●	○	○
Soteria [74]	GC	○	●	○	○
Deepsecure [75]	GC	○	●	○	○
Xonn [76]	GC	○	●	○	○
Zhu [77]	HE	○	○	○	●
Cryptonets [78]	HE	○	●	○	○
Faster [79]	HE	○	●	○	○
E2dm [80]	HE	○	●	○	○
Homopai [81]	HE	○	●	○	○
Spindle [82]	HE	○	○	○	●
Badawi [83]	HE	○	●	○	○
Chen [84]	HE	○	○	○	●
Cryptodl [6]	HE	○	●	○	○
Hervé [85]	HE	○	●	○	○
Privedge [86]	○	SS+GC	●	○	○
Secureml [3]	○	SS+GC	●	○	○
Chameleon [87]	○	SS+GC	○	●	○
Minionn [65]	○	SS+GC	●	○	○
DElphi [66]	○	SS+GC	●	○	○
Quotient [88]	○	OT+GC	●	○	○
Bayhenn [89]	○	HE+SS	●	○	○
Gelunet [90]	○	HE+SS	●	○	○
Cheetah [91]	○	HE+SS	●	○	○
Xu [92]	○	HE+SS	●	○	○
Cheetah [93]	○	HE+GC	●	○	○
Falcon [94]	○	HE+GC	●	○	○
Gazelle [64]	○	HE+GC	●	○	○
Autoprivacy [95]	○	HE+GC	●	○	○
Ensei [96]	○	HE+GC	●	○	○
Helen [97]	○	HE+GC	○	○	●

“●” and “○” denote adopted and unadopted item, respectively. Each item in “Hybrid computation” is listed with linear primitive + nonlinear primitive.

Generally, the hybrid-primitive computation achieves better overall performance compared with the single-primitive counterpart. As one of the most preferred combinations in hybrid-primitive computation, HE-GC schemes [64,93–97] utilize HE’s capability for efficient linear computation and GC’s advantage in computing the comparison function. Meanwhile, no extra trusted party is needed, which is more suitable for scenarios where parties are mutually distrusted. Specifically, the HE-GC framework, GAZELLE [64], has demonstrated three orders of magnitude faster than the single-HE framework, CryptoNets [78], which is one of the classic privacy-preserving systems.

1.2 CHALLENGES IN PDDL

Although PDDL has witnessed encouraging performance improvement, there are still two main challenges.

(1) Dedicated effort is still needed to make the data-encryption based privacy-preserving schemes more practical.

As one of the most efficient PDDL frameworks, GAZELLE’s running time for AlexNet [15] and VGG [16], which are two of the state-of-the-art DL models, is over one hundred and over one thousand seconds, respectively [98]. This cost is prohibitive in many applications. For example, the time constraints in many real-time speech recognition systems (such as Alexa and Google Assistant) are within 10 seconds [99,100] while autonomous cars demand an immediate response time less than one second [101]. Meanwhile, several gigabytes and tens of gigabytes for AlexNet and VGG also pose a challenge to the network traffic [98].

(2) There is a tradeoff between efficiency and model accuracy in an effort to reduce the communication/computation cost in PDDL.

Specifically, the model accuracy could drop by adopting the approximation mechanism for nonlinear computation, while the communication/computation cost is reduced. To have a more quantitative estimation for such tradeoff in data-encryption based privacy-preserving

Table 2. Quantitative Evaluation for Different PPDL Schemes.

Schemes	Number of layers	Accuracy drop (%)	Efficiency level			
			Days	Hrs	Mins	Ss
All-HE schemes						
Homopai [81]	~2	~1	○	○	●	○
Spindle [82]	~2	~1	○	○	●	○
Cryptodl [6]	~3	~4	○	●	○	○
Cryptonets [78]	~3	~1	○	○	●	○
E2dm [80]	~3	~1	○	○	○	●
Chen [84]	~3	~1	○	○	○	●
Zhu [77]	~8	~1	○	○	○	●
Hervé [85]	~8	~1	○	○	●	○
Badawi [83]	~11	~4	○	○	●	○
Faster [79]	~50	~4	○	○	●	○
All-SS schemes						
Securenn [70]	~4	~1	○	●	○	○
Liu [67]	~6	~2	○	●	○	○
Falcon [68]	~16	~1	●	○	○	○
Swift [69]	~3	~1	○	○	○	●
Trident [71]	~3	~1	○	○	○	●
Huang [72]	~10	~1	○	○	○	●
Leia [73]	~13	~7	○	○	○	●
All-GC schemes						
Deepsecure [75]	~3	~1	○	○	○	●
Soteria [74]	~13	~10	○	○	○	●
Xonn [76]	~16	~13	○	○	○	●
HE-GC schemes						
Helen [97]	~2	~1	○	●	○	○
Gazelle [64]	~5	~1	○	○	○	●
Falcon [94]	~10	~1	○	○	○	●
Ensei [96]	~18	~1	○	○	○	●
Autoprivacy [95]	~32	~1	○	○	○	●
Cheetah [93]	~50	~1	○	○	○	●

Systems marked with orange indicate they are training-enabled.

“●” and “○” denote adopted and unadopted item, respectively.

DL, Table 2 shows model accuracy and efficiency with respect to different network sizes⁴, i.e., number of model layers, in single-primitive and hybrid-primitive schemes, e.g., single-HE schemes [6, 77–85], single-SS schemes [67–73], single-GC schemes [74–76], and HE-GC schemes [64, 93–97].

For single-HE training schemes [6, 81, 82], back propagation is explored in small-size

⁴The statistics are reported in the respective paper.

networks, e.g., the 2-layer linear model or a 3-layer neural network. The massive iterations in back propagation result in significant computation cost even in plaintext DL models [26], which cannot be easily addressed with deeper networks. While the efficiency decreases from minutes (Mins) to days with more layers, the model accuracy of trained models also drops [6] since the nonlinear functions are approximated. For single-HE inference schemes, the classic CryptoNets [78] tackles a small CNN model with an efficiency level of Mins. The efficiency is further improved to seconds (Ss) by optimizations to reduce the complexity of HE operations [77, 80, 84, 85]. Meanwhile, the inference accuracy drops [79, 83] with more layers due to the approximated nonlinear functions.

As for the single-SS training schemes, the model accuracy is kept with negligible loss as the network goes deeper, while the tradeoff lies in the efficiency from hours (Hrs) to days or even to weeks. For single-SS inference schemes, the inference accuracy is maintained by introducing more parties, i.e., the extra trusted party [72] or multiple servers [73], to generate the desired shares. Furthermore, special computation-efficient models, e.g., BNNs in [73], could result in the accuracy drop with deeper networks.

As for single-GC schemes, the tradeoff tends to maintain efficiency (at the level of seconds) while accepting the certain loss of model accuracy. One of the reasons is that the networks are appropriately modified to keep the efficiency [76] as complexity increases with deeper models.

As for HE-GC training schemes, small-size networks, e.g., a simple 2-layer linear model, are explored to have a good balance between model accuracy and efficiency. For HE-GC inference schemes, model accuracy is maintained as the nonlinear functions, e.g., ReLU, can be exactly computed by GC. Meanwhile, efficiency is improved by deep optimizations of respective HE and GC calculation [64, 93–96].

1.3 DISSERTATION CONTRIBUTIONS

In order to shorten the gap to practical usability and mitigate the efficiency-accuracy

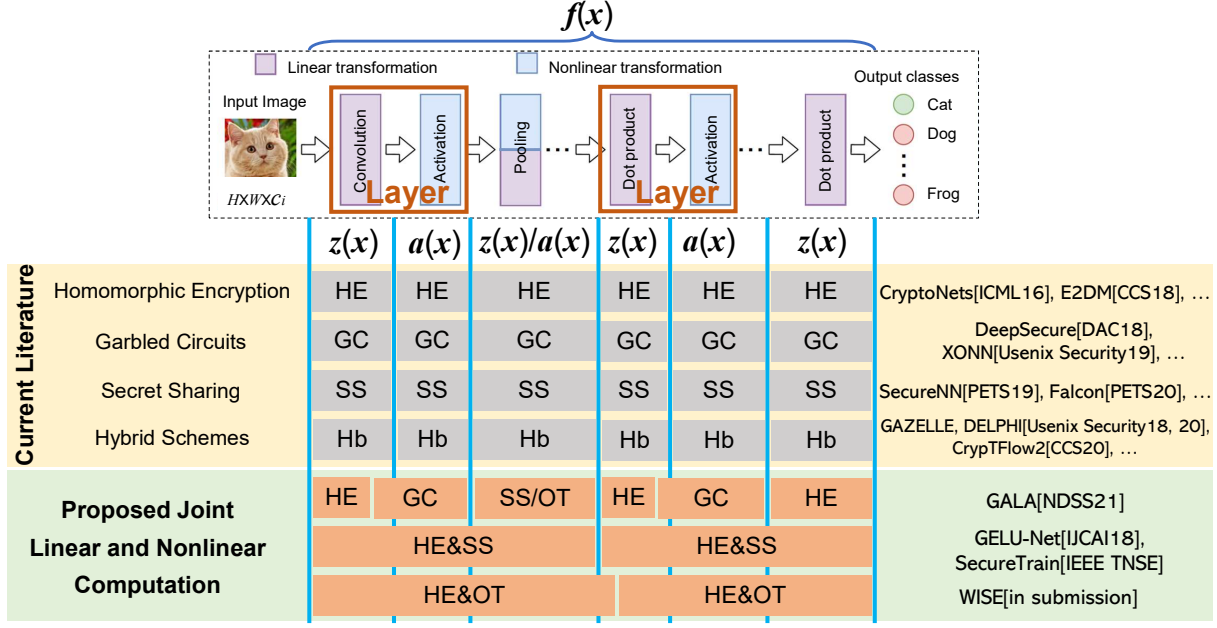


Figure 2. Overview of Works in the Dissertation.

tradeoff in PPD, by considering the joint linear and nonlinear computation to reduce or eliminate the expensive computing load in traditional function-wise computation (see Figure 2⁵), this dissertation targets deeply optimizing state-of-the-art frameworks and designing computation modules with data encryption, to fasten the overall performance. Four contributions are made as follows.

First, a deep optimization for the HE-based linear computation in HE-GC-based privacy-preserving DL inference, GALA [102], is presented. It features a row-wise weight matrix encoding, a combination of the share generation and a first-Add-second-Perm approach to reduce the most expensive permutation operations. GALA demonstrates an inference runtime boost by $2.5\times$ to $8.3\times$ over the state-of-the-art frameworks.

Second, the GC-based nonlinear calculation is replaced with a newly-designed joint linear and non-linear computation for each DL layer [103], in HE-GC-based privacy-preserving DL

⁵Here we use $f(x)$ to denote the whole network as a function while $z(x)$ and $a(x)$ are the linear and nonlinear functions, respectively. The term “Hb” denotes specific privacy-preserving primitive such as HE, GC, SS, or OT, etc.

inference, based on the Homomorphic Secret Sharing. This construction achieves an inference speedup as high as $48\times$ compared with state-of-the-art frameworks.

Third, the nonlinear calculation of each layer is completed for free by a carefully partitioned DL framework, GELU-Net [90], where the server performs linear computation on encrypted data utilizing a less complex homomorphic cryptosystem, while the client securely executes non-polynomial computation in plaintext without approximation. GELU-Net demonstrates $14\times$ to $35\times$ inference speedup compared to the classic systems.

Finally, we propose WISE framework to completely eliminate the most expensive HE permutation in HE-based linear calculation and reduce the communication cost from 4.5 rounds to only a half round, by a joint permutation-free computation between the nonlinear transformation in the current layer and the linear transformation in the next layer. WISE achieves $2\times$ to $13\times$ speedup over one of the most recent works through various widely-adopted neural layers and demonstrates a speedup up to $5.3\times$ on practical DL models.

1.4 ORGANIZATION OF THE DISSERTATION

The remainder of the dissertation is organized as follows. In Chapter 2, we introduce the necessary preliminaries that are adopted in this dissertation. They include the system models, threat model and privacy-preserving tools. The system models describe in detail the DL architecture, i.e, CNN. The threat model defines the semi-honest adversaries that are considered in this dissertation as well as shows the diagram to prove the security against such adversaries. Privacy-preserving tools explain the HE, SS, GC and OT techniques.

Chapter 3 presents GALA, which focuses on a deep optimization of the HE-based linear computations to minimize the most expensive permutation operations in privacy-preserving MLaaS, thus substantially reducing the overall computation time. It views the HE-based linear computation as a series of Homomorphic Add, Mult and Perm operations and chooses the least expensive operation in each linear computation step to reduce the overall cost.

SecureTrain is detailed in Chapter 4 where the GC-based nonlinear calculation is replaced

with a newly-designed joint linear and non-linear computation for each DL layer, based on the Homomorphic Secret Sharing. Furthermore, it explores a piggyback design for privacy-preserving training by carefully devising the share set and integrating the dataflow of the whole training process.

In Chapter 5, we elaborate the GELU-Net architecture, which securely completes the nonlinear calculation of each layer for free by carefully partitioning a deep neural network to two non-colluding parties. One party performs linear computation on encrypted data utilizing a less complex homomorphic cryptosystem, while the other executes non-polynomial computation in plaintext but in a privacy-preserving manner.

Chapter 6 proposes WISE, a hybrid privacy-preserving MLaaS protocol that features a permutation-free scheme which completely eliminates the most expensive HE permutation operations in the linear transformation and a joint permutation-free computation between the nonlinear transformation in the current layer and the linear transformation in the next layer, which reduces the communication cost from 4.5 rounds to only a half round. Finally, Chapter 7 concludes the dissertation.

CHAPTER 2

PRELIMINARIES

In this chapter, we introduce the necessary preliminaries that are adopted in this dissertation. They include the system models in Section 2.1, threat model in Section 2.2 and privacy-preserving tools in Section 2.3. The system models describe in detail the DL architecture, i.e, CNN. The threat model defines the semi-honest adversaries that are considered in this dissertation as well as shows the diagram to prove the security against such adversaries. Privacy-preserving tools explain the HE, SS, GC and OT techniques.

2.1 SYSTEM MODELS

There are typically two PPDL scenarios, i.e., privacy-preserving MLaaS and privacy-preserving training, as shown in Figure 1. Specifically, in privacy-preserving MLaaS shown in Figure 1 (a), there are two parties: the client \mathcal{C} and the server \mathcal{S} (or service provider). \mathcal{S} provides MLaaS using its internal DL model, e.g., CNN, that is well trained with massive data. \mathcal{C} owns private input and requests to make prediction by sending its data, in a privacy-preserving way, to \mathcal{S} . \mathcal{S} runs prediction using its proprietary DL model (parameters) and sends the prediction result back to \mathcal{C} . For example, a doctor, i.e., \mathcal{C} , sends an encrypted medical image (such as a chest X-ray) to \mathcal{S} , which runs the DL model and returns the encrypted prediction to the doctor. The prediction is then decrypted into a plaintext result to assist diagnosis and health care planning. In privacy-preserving training shown in Figure 1 (b), a set of clients, e.g., hospitals, own the private data and prefer to jointly train a more comprehensive DL model for better diagnosis performance, e.g., medical diagnosis. Clients send the data to and interact with \mathcal{S} in a privacy-preserving way to finally produce a well-trained model.

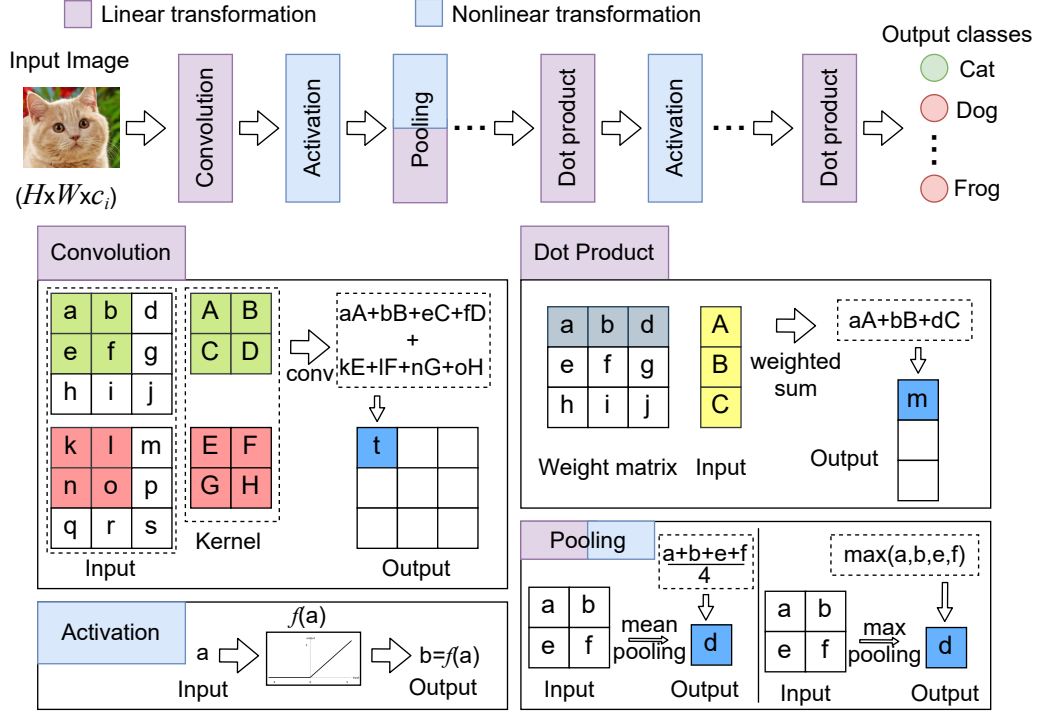


Figure 3. Basic structure of CNN.

Meanwhile, the computational flow of a DL model includes a cascade of layers. Each layer is composed of linear function that is followed by nonlinear function as shown in Figure 3. The two typical linear functions are *dot product* and *convolution*. The dot product takes as input a $n_i \times 1$ vector and a weight matrix of the size of $n_o \times n_i$. The output is a vector with the size of $n_o \times 1$. The j -th element of the output vector is calculated as a sum of element-wise product between the j -th row of the weight matrix and the input vector. The input of convolution is the feature maps with the size of $H \times W \times c_i$ and c_o kernels with the size of $f_h \times f_w \times c_i$, where H , W and c_i are respectively the height, width and number of channels of the input feature maps; f_h and f_w are the height and width of each kernel. For the first layer, the feature maps are simply the input image(s) as shown in Figure 3.

The convolution will produce the feature output, with a size of $H' \times W' \times c_o$. Specifically, the process of convolution can be visualized as placing each of the c_o kernels at different

locations of the input feature maps. At each location, a sum of element-wise product is computed between the kernel and corresponding data values within the kernel window. An example of convolution is shown in Figure 3 where $c_i = 2$, $c_o = 1$, $f_h = f_w = 2$, $H = W = 3$, and $H' = W' = 3$. Note that a *bias* value can be added to the output of the dot product and convolution. The output of the linear function is fed into the nonlinear function in an element-wise manner. In this dissertation, we mainly focus on ReLU function, $\text{ReLU}(x) = \max\{0, x\}$, which is one of the most widely used nonlinear functions in the state-of-the-art DL models [15, 16, 25, 26].

The layer containing convolution is called the convolution layer while the one with dot product is the fully-connected (dense) layer. Another type of layer is the pooling layer, which typically operates on the output channels of a convolution layer. The *pooling* operation (in the pooling layer) slides a window on the input channels and aggregates the elements within the window into a single output value. In this dissertation, we mainly adopt mean pooling, which is one of the most common pooling operations in DL. It takes the mean of elements within the sliding window as the output value and can be easily performed on the shares of secret [78]. Furthermore, pooling layers typically reduce the input size, i.e., H and W , but do not affect the number of channels, i.e., c_i .

Given the input data, the *forward propagation* repeats the linear transformation, i.e., dot product or convolution, and nonlinear transformation, i.e., activation functions or pooling functions, until the last layer where the nonlinear transformation is **softmax**¹ and the output is the prediction result. In data-encryption based privacy-preserving MLaaS, the cloud server receives encrypted data from the client and then completes the forward propagation such that the prediction result is finally returned back to the client. Meanwhile, the **softmax** can be ignored in data-encryption based privacy-preserving MLaaS as it is monotone and thus does not affect the prediction result. This dissertation mainly focuses on privacy-preserving

¹With the form $f(z_j) = \frac{e^{z_j}}{\sum_j e^{z_j}}$.

MLaaS. We firstly present an optimized privacy-preserving MLaaS, GALA, in Chapter 3. Then we replace the GC-based nonlinear calculation with a newly-designed joint linear and non-linear computation for each DL layer in Chapter 4. Chapter 5 elaborates the GELU-Net architecture, which securely completes the nonlinear calculation for free. Chapter 6 proposes WISE with a permutation-free scheme and a joint permutation-free computation between the nonlinear transformation in the current layer and the linear transformation in the next layer.

In order to do training, the output of forward propagation is fed into the *back propagation* to update the model parameters, i.e, weight matrix, bias and kernels, through gradient descent where the derivatives of activation functions and pooling functions are involved [15]. The goal of data-encryption based privacy-preserving training is to make the process of parameter update blind to all involved parties, given that all the clients send their encrypted data to the cloud servers, which obviously update the model parameters by multi-party interactions [4, 67]. Chapter 4 and Chapter 5, respectively, extend the privacy-preserving MLaaS to privacy-preserving training with a piggyback design that combines the dataflow in forward and back propagation and the plaintext computation for nonlinear functions.

2.2 THREAT MODEL

There are mainly two adversarial behaviors considered in data-encryption based PPDL: 1) a participant is defined to be semi-honest (SH) or passive if this participant executes the pre-defined protocols correctly but tries to learn as much private information as possible by analyzing the messages exchanged during the protocol execution, and 2) a participant is defined to be malicious (MA) or active if he arbitrarily deviates from the protocol specifications. Concretely, in data-encryption based PPDL with SH participants, the clients (data owners) and cloud servers are assumed to follow all protocols during the entire learning process, while the individual client tries to learn the model parameters or private data of other clients (in training), and cloud servers try to infer clients' input data, or the model

parameters (in training). As for data-encryption based PPDL with MA participants, the individual client arbitrarily deviates from the protocols by sending cloud servers incorrect input or intermediate data, while the cloud servers deviate from the protocols by applying incorrect model parameters (in MLaaS) or by sending clients incorrect intermediate data. The two adversarial behaviors result in either misleading prediction in MLaaS or meaningless model parameters in training. One should prove the security against either SH or MA adversaries in the designed PPDL framework.

The security proofs against SH adversaries are given in Yao’s protocol [54, 104] and the Goldreich-Micali-Wigderson (GMW) protocol [62, 63], while the security proofs against MA adversaries are based on zero-knowledge protocols [62, 105]. The SH threat model is weaker than the MA counterpart, but it allows us to build highly efficient protocols and is therefore widely adopted in data-encryption based PPDL [68]. In this dissertation, we mainly focus on the privacy-preserving MLaaS with SH threat model². Specifically, the security lies in the existence of two *simulators* respectively for the corrupted client and the corrupted server, such that the *view* of each simulator in the *ideal world* is computationally indistinguishable from that of each simulated party in the *real world*. As shown in Figure 4, the *view* of a party includes its input, randomness and received message(s). In the ideal world, the simulator interacts with a Trust Third Party (TTP) while in the real world, it communicates with the simulated party according to the specific protocol. As for a corrupted client, the simulator abstracts its (private) input data, sends it to the TTP and receives the final result in the ideal world, while it also interacts with the client in the real world based on the available data and the same randomness (with the client). As for a corrupted server, the simulator abstracts its (proprietary) model parameters, sends it to the TTP and receives none in the ideal world, while it also interacts with the server in the real world based on the available data and the same randomness (with the server).

Meanwhile, as the data-encryption based PPDL tackles the privacy issues during the

²The analysis for privacy-preserving training with SH threat model follows similar diagram.

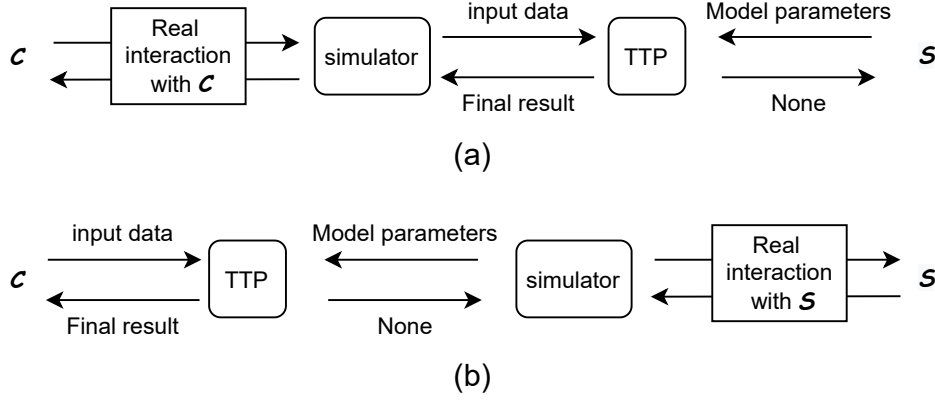


Figure 4. Simulator for the (a) corrupted client and (b) corrupted server in MLaaS.

learning process, i.e., inference or training, the black-box attacks that utilize the prediction results, e.g., model extraction [106, 107], model inversion [41], membership inference [108], detection evasion [109, 110], are out of the scope of this dissertation.

2.3 PRIVACY-PRESERVING TOOLS

In this section, we review four data-encryption based privacy-preserving primitives used for linear and nonlinear computation in the dissertation.

2.3.1 HOMOMORPHIC ENCRYPTION

Homomorphic Encryption (HE) [48, 49, 51–53] is a kind of public-key encryption that additionally supports linear operations, e.g., addition and multiplication, over the ciphertexts. Conventional HE scheme operates on individual ciphertext [48, 49], while the packed homomorphic encryption (PHE) packs multiple values into a single ciphertext and performs element-wise operation in a Single Instruction Multiple Data (SIMD) manner [111] to parallel the HE computation³. Specifically, an HE scheme involves a tuple of algorithms $\text{HE} = (\text{KeyGen}, \text{Enc}, \text{Dec}, \text{Eval})$ with the following syntax:

- $\text{HE.KeyGen} \rightarrow (\text{pk}, \text{sk})$. HE.KeyGen randomly outputs a public key pk and a secret key

³We recommend the readers to [112, 113] for a more basic and concrete description about PHE.

\mathbf{sk} .

- $\text{HE.Enc}(\mathbf{pk}, \mathbf{m}) \rightarrow [\mathbf{m}]_{\mathbf{pk}}$. HE.Enc takes as input the public key \mathbf{pk} and plaintext vector $\mathbf{m} \in \mathbb{Z}^n$ and outputs a ciphertext $[\mathbf{m}]_{\mathbf{pk}}$. We denote $[\mathbf{m}]_{\mathcal{C}}$ and $[\mathbf{m}]_{\mathcal{S}}$ as the ciphertext where \mathbf{m} is respectively encrypted by the public key of client \mathcal{C} and server \mathcal{S} .
- $\text{HE.Dec}(\mathbf{sk}, [\mathbf{m}]_{\mathbf{pk}}) \rightarrow \mathbf{m}$. HE.Dec takes as input the secret key \mathbf{sk} and ciphertext $[\mathbf{m}]_{\mathbf{pk}}$, and outputs plaintext vector \mathbf{m} .
- $\text{HE.Eval}(\mathbf{pk}, [\mathbf{m}_1]_{\mathbf{pk}}, [\mathbf{m}_2]_{\mathbf{pk}}, L(\mathbf{m}_1, \mathbf{m}_2)) \rightarrow [\mathbf{m}_3]_{\mathbf{pk}}$. With the input of public key \mathbf{pk} , two ciphertexts $[\mathbf{m}_1]_{\mathbf{pk}}$ and $[\mathbf{m}_2]_{\mathbf{pk}}$ ⁴, and a linear function L , HE.Eval outputs a new ciphertext $[\mathbf{m}_3]_{\mathbf{pk}}$ where $\mathbf{m}_3 = L(\mathbf{m}_1, \mathbf{m}_2)$. Here the three basic HE operations in HE.Eval are addition, multiplication, and permutation (Perm). The addition (multiplication) between $[\mathbf{m}_1]_{\mathbf{pk}}$ and $[\mathbf{m}_2]_{\mathbf{pk}}$ adds (multiplies) \mathbf{m}_1 and \mathbf{m}_2 in element-wise manner, while permutation transforms the element order in \mathbf{m}_1 (or \mathbf{m}_2) such that its j -th element becomes its first element. Among the three basic HE operations, Perm is the most expensive one [64]. Specifically, in Chapter 3, we aim to minimize the Perm operation for the linear computation and a Perm-free scheme is proposed in Chapter 6 such that the overall computation cost is noticeably reduced. Furthermore, the involved multiplication in our designs is *scalar multiplication* between one ciphertext and one plaintext, which is faster than the ciphertext-ciphertext counterpart.

2.3.2 SECRET SHARING

In the 2-of-2 additive secret sharing protocol⁵, a value x is shared between two parties, e.g, client \mathcal{C} and server \mathcal{S} , such that combining the two secrets yields the true value. Specifically, a party that wants to share a secret x selects a random number r to form a pair $(\langle x \rangle_1, \langle x \rangle_2) = (x - r, r)$. Here, x can be either plaintext or ciphertext. That party then sends one of the shares (either $\langle x \rangle_1$ or $\langle x \rangle_2$) to the other party. To reconstruct the secret x , one needs to

⁴It can also be one ciphertext $[\mathbf{m}_1]_{\mathbf{pk}}$ and one plaintext \mathbf{m}_2 .

⁵This type of secret sharing is used in this dissertation.

only add the two shares $x = \langle x \rangle_1 + \langle x \rangle_2$. Given a share $\langle x \rangle_1$ or $\langle x \rangle_2$, the secret x is perfectly hidden. In this dissertation, we apply the 2-of-2 additive secret sharing to share the linear result at \mathcal{S} as well as the nonlinear result at \mathcal{C} such that the computation over ciphertext is transformed by the counterpart over shares (in plaintext), which efficiently reduces the computation overhead⁶.

2.3.3 GARBLED CIRCUITS

In Yao’s garbled circuits [54], a garbled version of a Boolean circuit is interactively evaluated by two parties. One party, called garbler, creates the garbled circuit and encodes its inputs based on the garbled circuit. The other party, called evaluator, receives the garbled circuit and obtains encodings of its inputs via Oblivious Transfer (OT) [59]. The evaluator then evaluates the circuit gate by gate to finally compute the encoding of the output, which is decoded by garbler. Formally, a garbling scheme is a tuple of algorithms $\text{GC} = (\text{Garble}, \text{Eval})$ with the following syntax [66]:

- $\text{GC.Garble}(\mathcal{C}) \rightarrow (\tilde{\mathcal{C}}, \{\text{label}_{i,0}, \text{label}_{i,1}\})$. On input a boolean circuit \mathcal{C} , Garble outputs a garbled circuit $\tilde{\mathcal{C}}$ and a set of labels $\{\text{label}_{i,0}, \text{label}_{i,1}\}$. Here $\text{label}_{i,b}$ represents assigning the value $b \in \{0, 1\}$ to the i -th input label.
- $\text{GC.Eval}(\tilde{\mathcal{C}}, \{\text{label}_{i,x_i}\}_{i \in [n]}) \rightarrow y$. On input a garbled circuit $\tilde{\mathcal{C}}$ and labels $\{\text{label}_{i,x_i}\}_{i \in [n]}$ corresponding to an input $x = \{x_i\} \in \{0, 1\}^n$, Eval outputs a string $y = \mathcal{C}(x)$.

The GC tuple must be complete: the output of Eval must equal $\mathcal{C}(x)$. Second, it must be private: given $\tilde{\mathcal{C}}$ and $\{\text{label}_{i,x_i}\}$, the evaluator should not learn anything about \mathcal{C} or x except the size of $|\mathcal{C}|$ (i.e., the number of gates in \mathcal{C} denoted by $1^{|\mathcal{C}|}$) and the output $\mathcal{C}(x)$. Meanwhile, Yao’s protocol has a constant number of communication rounds and the main overhead stems from the total number of AND gates in the circuit, as XOR gates can be

⁶In Chapter 4, we detail another type of secret sharing called Homomorphic Secret Sharing (HSS), which shares a desired function output (corresponding to x) by a homomorphic evaluation algorithm (corresponding to the generation of the pair $(x - r, r)$).

evaluated for free [55]. Other state-of-the-art GC optimizations are point-and-permute [56], fixed-key AES garbling [57], and half-gates [58]. The **ReLU** result can be obtained by the GC with a comparison circuit [64], which is combined in Chapter 3 with an optimization of HE based linear computation to reduce the overall cost.

2.3.4 OBLIVIOUS TRANSFER

In the 1-out-of- k Oblivious Transfer (OT) [114], denoted as $\binom{k}{1}$ -OT $_\ell$, the sender's inputs are the k strings, $m_0, m_1, \dots, m_{k-1} \in \{0, 1\}^\ell$, and the receiver's input is a value $i \in \{0, 1, \dots, k-1\}$. At the end of the OT execution, the receiver obtains m_i from the functionality, and the sender receives no output. Here, the OT protocol guarantees that 1) the receiver learns nothing about $m_{j, j \neq i}$, and 2) the sender learns nothing about i . An advancement in the practicality of OT protocols is the OT extension [60, 115], which is further optimized such as in [116]. The state-of-the-art approaches rely on OT-based multiplication in **ReLU** computation [7], which involves 4 rounds of communication. In contrast, the proposed WISE protocol in Chapter 6 *totally eliminates* this overhead by reconstructing the multiplication formula in **ReLU** computation, thus contributing to the reduction of the overall computation cost.

2.4 CHAPTER SUMMARY

This chapter has introduced the necessary preliminaries that are adopted in this dissertation. They have included the system models in Section 2.1, threat model in Section 2.2 and privacy-preserving tools in Section 2.3. The system models have described in detail the DL architecture, i.e, CNN. The threat model has defined the semi-honest adversaries that are considered in this dissertation and shown the diagram to prove the security against such adversaries. Privacy-preserving tools have explained the HE, SS, GC and OT techniques.

CHAPTER 3

GREEDY COMPUTATION FOR LINEAR ALGEBRA IN PDDL

Recall from Section 1.1 that the schemes that exploit Homomorphic Encryption (HE)-based linear computations and Garbled Circuit (GC)-based nonlinear computations have demonstrated superior performance to enable privacy-preserving MLaaS. Nevertheless, there is still a significant gap in the computation speed. Our investigation has found that the HE-based linear computation dominates the total computation time for state-of-the-art deep neural networks. Furthermore, the most time-consuming component of the HE-based linear computation is a series of Permutation (Perm) operations that are imperative for dot product and convolution in privacy-preserving MLaaS. This chapter focuses on a deep optimization of the HE-based linear computations to minimize the Perm operations, thus substantially reducing the overall computation time.

To this end, we propose *GALA: Greedy computation for Linear Algebra* in privacy-preserving neural networks, which views the HE-based linear computation as a series of Homomorphic Add, Mult and Perm operations and chooses the least expensive operation in each linear computation step to reduce the overall cost. GALA makes the following contributions: (1) It introduces a row-wise weight matrix encoding and combines the share generation that is needed for the GC-based nonlinear computation, to reduce the Perm operations for the dot product; (2) It designs a first-Add-second-Perm approach (named *kernel grouping*) to reduce Perm operations for convolution. As such, GALA efficiently reduces the cost for the HE-based linear computation, which is a critical building block in almost all of the recent frameworks for privacy-preserving neural networks, including GAZELLE (Usenix Security’18), DELPHI (Usenix Security’20), and CrypTFlow2 (CCS’20). With its deep optimization of the HE-based linear computation, GALA can be a plug-and-play module integrated into these systems to further boost their efficiency.

Our experiments show that it achieves a significant speedup up to $700\times$ for the dot product and $14\times$ for the convolution computation under different data dimensions. Meanwhile, GALA demonstrates an encouraging runtime boost by $2.5\times$, $2.7\times$, $3.2\times$, $8.3\times$, $7.7\times$, and $7.5\times$ over GAZELLE and $6.5\times$, $6\times$, $5.7\times$, $4.5\times$, $4.2\times$, and $4.1\times$ over CrypTFlow2, on AlexNet, VGG, ResNet-18, ResNet-50, ResNet-101, and ResNet-152, respectively. The rest of this chapter is organized as follows. Section 3.1 details the motivation of GALA. Section 3.2 introduces the primitives that GALA relies on. Section 3.3 describes the design details of GALA. Section 3.4 presents the security analysis of GALA. The experimental results are illustrated and discussed in Section 3.5. Finally, Section 3.6 concludes this chapter.

3.1 MOTIVATION

Since designing and training a deep neural network model requires intensive resource and DL talents, cloud providers began to offer Machine Learning as a Service (MLaaS) [117], where a proprietary DL model is trained and hosted on a cloud. Clients can utilize the service by simply sending queries (inference) to the cloud and receiving results through a web portal. While this emerging cloud service is embraced as an important tool for efficiency and productivity, the interaction between clients and cloud servers leads to new vulnerabilities. This chapter focuses on the development of privacy-preserving and computationally efficient MLaaS.

Although communication can be readily secured from end to end, privacy still remains a fundamental challenge. On the one hand, the clients must submit their data to the cloud for inference, but they want the data privacy well protected, preventing the curious cloud provider or attacker with access to the cloud from mining valuable information. In many domains such as health care [118] and finance [119], data are extremely sensitive. For example, when patients transmit their physiological data to the server for medical diagnosis, they do not want anyone (including the cloud provider) to see it. Regulations such as the Health Insurance Portability and Accountability Act (HIPAA) [36] and the General Data

Protection Regulation (GDPR) in Europe [120] are in place to impose restrictions on sharing sensitive user information. On the other hand, cloud providers do not want users to be able to extract their proprietary model that has been trained with significant resource and efforts [107]. Furthermore, the trained model contains private information about the training data set and can be exploited by malicious users [40, 108, 121]. To this end, there is an urgent need to develop effective and efficient schemes to ensure that in MLaaS a cloud server does not have access to users' data and a user cannot learn the server's model.

A series of efforts have been made to enable privacy-preserving MLaaS, by leveraging cryptographic techniques as summarized in Section 1.1. The first is the *Homomorphic Encryption (HE)-Based Approach*. For example, in CryptoNets [78], Faster CryptoNets [79] and CryptoDL [6], the client encrypts data using HE and sends the encrypted data to the server. The server performs polynomial computations (e.g., addition and multiplication) over encrypted data to calculate an encrypted inference result. The client finally obtains the inference outcome after decryption. E2DM [80] adopts a more efficient HE (i.e., packed HE [111]) which packs multiple messages into one ciphertext and thus improves computation efficiency. The second approach is based on *Garbled Circuit (GC)* [104]. DeepSecure [75] and XONN [76] binarize the computations in neural networks and employ GC to obviously obtain the prediction without leaking sensitive client data. The third approach exploits *Secret Sharing (SS)*. SS is used in [122] and [4] to split the client data into shares. The server only owns one share of the data. The computations are completed by interactive share exchanges. In addition, Differential Privacy (DP) [123–125] and Trusted Execution Environment (TEE) [126–129] are also explored to protect data security and privacy in neural networks. In order to deal with different properties of linearity (weighted sum and convolution functions) and nonlinearity (activation and pooling functions) in neural network computations, several efforts have been made to orchestrate multiple cryptographic techniques to achieve better performance [1–3, 5, 7, 64–66, 87, 90, 97, 130–134]. Among them, the schemes

with HE-based linear computations and GC-based nonlinear computations (called the HE-GC neural network framework hereafter) demonstrate superior performance [64–66, 130]. Specifically, the GAZELLE framework [64] represents the state-of-the-art design for the HE-based linear computation and achieves a speedup of three orders of magnitude faster than the classic CryptoNets inference system [78].

Despite rapid improvement, there is still a significant gap in computation speed, rendering the existing schemes infeasible for practical applications. For example, the time constraints in many real-time applications (such as speech recognition) are within a few seconds [99, 100]. In contrast, our benchmark has shown that GAZELLE takes 43 seconds and 659 seconds to run the well-known deep neural networks **ResNet-18** and **ResNet-152** [26] on an Intel i7-8700 3.2GHz CPU (see detailed experimental settings in Section 3.5), which renders it impractical in real-world applications.

This performance gap motivates us to further improve the efficiency of the HE-GC neural network frameworks. In the deep neural network, both the fully-connected and convolutional layers are based on the linear computation, while the activation functions perform nonlinear computation. The former dominates the total computation time in state-of-the-art deep neural networks. For example, the runtime of the nonlinear computation in GAZELLE is merely 2.3%, 1.8%, 1.7%, 1.5%, 1.6%, and 2%, respectively, on **AlexNet** [15], **VGG** [16], **ResNet-18** [26], **ResNet-50** [26], **ResNet-101** [26], and **ResNet-152** [26]. The nonlinear cost in the original plaintext models is even lower (averaged 1.7%). This indicates a great potential to speed up the overall system through optimizing linear computations. Although a few recent approaches, e.g., DELPHI [66] and CrypTFlow2 [7], perform better than GAZELLE in terms of the overall system runtime, they all inherit the HE-based linear computation in GAZELLE. This work contributes a solid optimization on the HE-based linear computation (i.e., dot product and convolution), which can be integrated into those systems (including GAZELLE, DELPHI and CrypTFlow2) to further improve their overall system performance. The HE-based computation consists of three basic operations: Homomorphic

Addition (Add), Multiplication (Mult), and Permutation (Perm). Our investigation has shown that the most time-consuming part of the HE-based computation is a series of Perm operations that are imperative to enable dot product and convolution. Our experiments show that Perm is 56 times slower than Add and 34 times slower than Mult. As shown in Table 3, in the dot product by multiplying a 2×2048 matrix with a length-2048 vector, the cost in GAZELLE is dominated by Perm, which takes about *98% of the computation time*. This observation motivates the proposed linear optimization, which aims to minimize the Perm operations, thus substantially reducing the overall computation time. With smaller number of Perm operations, the proposed approach demonstrates $10\times$ speedup in the above matrix-vector computation.

Table 3. Cost of matrix-vector multiplication (time in millionsecond).

Method	Total (ms)	Perm		Mult		Add	
		#	time	#	time	#	time
GAZELLE	2	11	1.96	2	0.01	11	0.037
Proposed	0.2	1	0.17	2	0.01	1	0.003

This significant speedup lies in a simple and efficient idea to choose the least expensive operation in each linear computation step to reduce the overall cost. We name the proposed approach *GALA: Greedy computation for Linear Algebra* in privacy-preserving neural networks. We view the HE-based linear computation as a series of Homomorphic Add, Mult and Perm operations. The two inputs are the encrypted vector (or channels) from the client and the plaintext weight matrix (or kernel) from the server. The output is the encrypted dot product (or convolution). The objective in each step is to choose the most efficient operations in the descending priorities of Add, Mult and Perm. To this end, we (1) design

a row-wise weight matrix encoding with combined share generation¹ (i.e., a row-encoding-share-RaS (Rotated and Sum) approach) to reduce the number of Perm operations in the dot product by $\log_2 \frac{n}{n_o}$ where n is the number of slots in a ciphertext and n_o is the output dimension of dot product and (2) propose a first-Add-second-Perm approach (named *kernel grouping*) to reduce the number of Perm operations of convolution by a factor of $\frac{c_i}{c_n}$ where c_i and c_n are respectively the number of channels in input data, and the number of channels that can be packed in a ciphertext. n is always greater than and can be up to 8192 times of n_o depending on the dimension of dataset [135] and HE implementation [136].

At the same time, $\frac{c_i}{c_n}$ is at least one and can be up to 256 for state-of-the-art neural network architectures such as **ResNets** [26] where the large channels, i.e., 1024 and 2048, and small kernel size, i.e., 1×1 and 3×3 , are adopted. The larger input data from users will result in smaller c_n , which accordingly contributes to higher speedup especially in the state-of-the-art CNNs. As such, GALA efficiently boosts the performance of HE-based linear computation, which is a critical building block in almost all of the recent frameworks for privacy-preserving neural networks, e.g., GAZELLE, DELPHI, and CrypTFlow2. Furthermore, GALA’s deep optimization of the HE-based linear computation can be integrated as a plug-and-play module into these systems to further improve their overall efficiency. For example, GALA can serve as a computing module in the privacy-preserving DL platforms, MP2ML [133] and CrypTFlow [134], which are compatible with the user-friendly TensorFlow [137] DL framework. Our experiments show that GALA achieves a significant speedup up to $700\times$ for the dot product and $14\times$ for the convolution computation under various data dimensions. Meanwhile, GALA demonstrates an encouraging runtime boost by $2.5\times$, $2.7\times$, $3.2\times$, $8.3\times$, $7.7\times$, and $7.5\times$ over GAZELLE and $6.5\times$, $6\times$, $5.7\times$, $4.5\times$, $4.2\times$, and $4.1\times$ over CrypTFlow2, on AlexNet, VGG, ResNet-18, ResNet-50, ResNet-101, and ResNet-152, respectively. More details are given in Section 3.5.

¹The resultant linear output will be shared between server and client as the input of GC-based nonlinear computation.

3.2 PRELIMINARIES

We consider an MLaaS system shown in Figure 1 (a). The client owns private data. The server is in the cloud and has a well-trained deep learning model to provide the inference service based on the received client’s data. For example, a doctor sends an encrypted medical image (such as a chest X-ray) to the server, which runs the neural network model and returns the encrypted prediction to the doctor. The prediction is then decrypted into a plaintext result to assist diagnosis and health care planning. Meanwhile, we focus on the Convolutional Neural Network (CNN) with the `ReLU` activation function as described in Section 2.1, and GALA mainly addresses privacy-preserving linear optimization (i.e., convolution and dot product). The privacy-preserving nonlinear optimizations (especially activations) are based on GC as introduced in other HE-GC approaches such as GAZELLE [64]. Furthermore, similar to GAZELLE [64] and other previous works, namely the SecureML [3], MiniONN [65], DeepSecure [75] and XONN [76], GALA adopts the semi-honest model introduced in Section 2.2, in which both parties try to learn additional information from the message received (assuming they have a bounded computational capability). That is, the client \mathcal{C} and server \mathcal{S} will follow the protocol, but \mathcal{C} wants to learn model parameters and \mathcal{S} attempts to learn the client’s data.

GALA mainly adopts two privacy-preserving tools (see Section 2.3) namely Brakerski-Fan-Vercauteren (BFV) scheme [52] which is one of the PHE techniques, and Secret Sharing (SS). Recall in Section 2.3.1 that the run-time complexity of Add and ScMult² is significantly lower than that of Perm among the three basic HE operations. From our experiments, a Perm operation is 56 times slower than an Add operation and 34 times slower than a ScMult operation³. This observation motivates the proposed linear optimization, which aims to minimize the number of Perm operations, thus substantially reducing the overall computation time. Meanwhile, PHE introduces noise in the ciphertext which theoretically

²We use it to denote the scalar multiplication.

³It is based on GAZELLE’s implementation at https://github.com/chiraag/gazelle_mpc.

hides the original message [51, 64]. Assume the noise of two PHE-encrypted ciphertext are η_0 and η_1 , then the noise after the Add operation is approximately $\eta_0 + \eta_1$. The noise after a ScMult operation is $\eta_{mult}\eta_0$ where η_{mult} is the *multiplicative noise growth of the SIMD scalar multiplication operation* [64]. The noise after a Perm operation is $\eta_0 + \eta_{rot}$ where η_{rot} is the *additive noise growth of a permutation operation* [64]. Roughly, we have $\eta_{rot} > \eta_{mult} \gg \eta_0 \gg 1$. If the noise goes beyond a certain level, the decryption would fail. Thus, it is also important to have good noise management over the ciphertext. We will show in Section 3.3.3 that GALA has better noise control than GAZELLE, which further guarantees the overall success for the linear computations. On the other hand, while the overall idea of secret sharing (SS) is straightforward, creative designs are often required to enable its effective application in practice. Specifically, in the HE-GC neural network framework, the linear result from the dot product or convolution is encrypted at the server side and needs to be shared with the client to enable the following GC-based nonlinear computation. Assume m is the resulted ciphertext of a linear computation at the server, GAZELLE then generates the share $\langle m \rangle_0 = r$ and sends $\langle m \rangle_1 = m - r$ to the client. The two shares act as the input of the GC-based nonlinear computation. Here the computation of m involves a series of Perm operations, which is time-consuming. Instead of directly generating the share $\langle m \rangle_0 = r$ for m , we develop a *share-RaS (Rotate and Sum) computing* for dot product which lets the server generate an indirect share r' for the incomplete m , m' , while the true r is easy to be derived from r' and the true $\langle m \rangle_1 = m - r$ is easy to be derived from $m' - r'$. The computation of m' eliminates a large number of Perm operations thus reducing the computation complexity. Specifically, our result shows that the proposed *share-RaS computing* demonstrates $19\times$ speedup for the dot product by multiplying a 16×128 matrix with a length-128 vector (the detailed benchmarks are shown in Section 3.5).

3.3 SYSTEM DESCRIPTION

In this section, we introduce the proposed system, GALA, for streamlining the linear

computations (i.e., matrix-vector multiplication and convolution) in privacy-preserving neural network models. The HE-based linear computation consists of three basic operations: Homomorphic Addition (Add), Multiplication (Mult), and Permutation (Perm). Our investigation has shown that the linear computation dominates the total computation cost and the most time-consuming part of HE-based linear computation is a series of Perm operations that are imperative to enable dot product and convolution. GALA aims to minimize the Perm operations, thus substantially reducing the overall computation time. We view the HE-based linear computation as a series of Add, Mult and Perm. The two inputs to linear computation are the encrypted vector (or channels) from the client and the plaintext weight matrix (or kernel) from the server. The output is the encrypted dot product (or convolution). The objective in each step is to choose the most efficient operations in the descending priorities of Add, Mult and Perm. Therefore, the overhead for the HE-based linear computation can be efficiently reduced by GALA. The recent privacy-preserving neural network frameworks can integrate GALA as a plug-and-play module to further boost their efficiency. We also analyze the (better) noise management and (guaranteed) system security of GALA.

3.3.1 ROW-ENCODING-SHARE-RAS MATRIX-VECTOR MULT

We first focus on matrix-vector multiplication (dot product) which multiplies a plaintext matrix at the server with an encrypted vector from the client. We first discuss a naive method followed by the mechanism employed in the state-of-the-art framework (i.e., GAZELLE [64]), and then introduce the proposed optimization of GALA that significantly improves the efficiency in matrix-vector multiplication.

For a lucid presentation of the proposed GALA and comparison with the state-of-the-art framework, we adopt the same system model used in [64]. More specifically, we consider a Fully Connected (FC) layer with n_i inputs and n_o outputs. The number of slots in one ciphertext is n . We also adopt the assumptions used in [64]: n , n_i and n_o are powers of two, and n_o and n_i are smaller than n . If they are larger than n , the original $n_o \times n_i$ matrix can

be split into $n \times n$ sized blocks that are processed independently.

1) Naive Method: The naive calculation for matrix-vector multiplication is shown in Figure 5, where \mathbf{w} is the $n_o \times n_i$ plaintext matrix on the server and $[\mathbf{x}]_c$ is the HE-encrypted vector provided by the client. The server encodes each row of \mathbf{w} into a separate plaintext vector (see step (a) in Figure 5). The length of each encoded vector is n (including padded 0's if necessary). We denote these encoded plaintext vectors as $\mathbf{w}_0, \mathbf{w}_1, \dots, \mathbf{w}_{(n_o-1)}$. For example, the yellow and green rows in step (a) of Figure 5 are \mathbf{w}_0 and \mathbf{w}_1 , respectively.

The server intends to compute the dot product between \mathbf{w} and $[\mathbf{x}]_c$. To this end, it first uses ScMult to compute the elementwise multiplication between \mathbf{w}_i and the encrypted input vector $[\mathbf{x}]_c$ to get $[\mathbf{u}_i]_c = [\mathbf{w}_i \odot \mathbf{x}]_c$ (see step (b) in Figure 5). The sum of all elements in \mathbf{u}_i will be the i -th element of the desired dot product between \mathbf{w} and $[\mathbf{x}]_c$. However, as discussed in Section 2.3.1, it is not straightforward to obtain the sum under the packed HE.

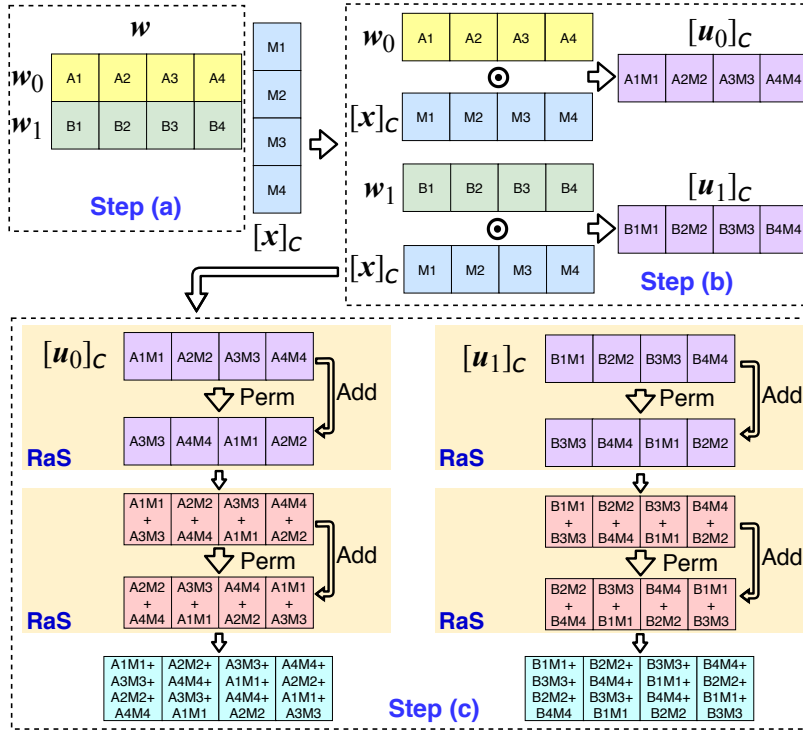


Figure 5. Naive matrix-vector multiplication.

A *rotate-and-sum* (RaS) calculation must be used here, as illustrated in step (c) of Figure 5. Specifically, the entries in $[\mathbf{u}_i]_C$ are first rotated through Perm by $\frac{n_i}{2}$ positions such that the first $\frac{n_i}{2}$ entries of the rotated $[\mathbf{u}_i]_C$ are actually the second $\frac{n_i}{2}$ entries of the original $[\mathbf{u}_i]_C$.

Then the server uses Add to conduct elementwise addition between the rotated $[\mathbf{u}_i]_C$ and the original $[\mathbf{u}_i]_C$, which results in a ciphertext whose first $\frac{n_i}{2}$ entries contain the elementwise sum of the first and second $\frac{n_i}{2}$ entries of \mathbf{u}_i . The server conducts this RaS process for $\log_2 n_i$ iterations. Each iteration acts on the resulted ciphertext from the previous iteration, and rotates by half of the previous positions, as shown in Step (c) of Figure 5. Finally, the server gets a ciphertext where the first entry is the i -th element in \mathbf{wx} . By applying this procedure on each of the n_o rows (i.e., $\mathbf{w}_0, \mathbf{w}_1, \dots, \mathbf{w}_{(n_o-1)}$), the server obtains n_o ciphertext. Altogether, the first entries of those ciphertext correspond to \mathbf{wx} .

We now analyze the complexity of the above linear computation process, in terms of the number of operations and output ciphertext. We consider the process starting from the server's reception of $[\mathbf{x}]_C$ (i.e., the encrypted input data from the client) until it obtains the to-be-shared ciphertext⁴ (i.e., the n_o ciphertext after RaS). There are a total of n_o scalar multiplications (ScMult) operations, $n_o \log_2 n_i$ Perm operations and $n_o \log_2 n_i$ Add operations. It yields n_o output ciphertext, each of which contains one element of the linear result \mathbf{wx} . This inefficient use of the ciphertext space results in a low efficiency for linear computations.

2) Hybrid Calculation (GAZELLE): In order to fully utilize the n slots in a ciphertext and further reduce the complexity, the state-of-the-art scheme is to combine the diagonal encoding [138] and RaS, by leveraging the fact that n_o is usually much smaller than n_i in FC layers. This hybrid method shows that the number of expensive Perm operations is a function of n_o rather than n_i , thus accelerating the computation of FC layers [64]. The basic idea of the hybrid method is shown in Figure 6.

⁴In HE-GC neural network computing, the resultant ciphertext from linear calculation are shared between client and server as the input of GC-based nonlinear function.

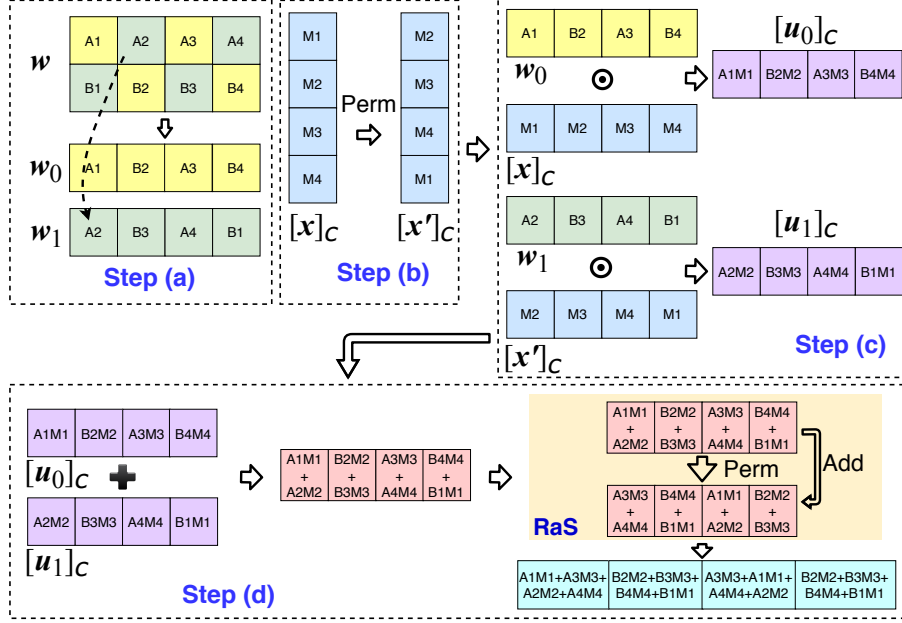


Figure 6. Hybrid matrix-vector multiplication.

Specifically, the server encodes w into n_o plaintext vectors through a diagonal manner. For example, in step (a) of Figure 6, the first plaintext vector w_0 consists of the yellow elements of matrix w , ($A1, B2, A3, B4$), and the second plaintext vector w_1 consists of the green elements ($A2, B3, A4, B1$). Note that the w_0 in this method is different from the w_0 in the naive method of Figure 5. So is w_1 .

The server then rotates $[x]_C$ by i positions, shown in step (b), and uses ScMult to perform elementwise multiplication with w_i . For example, in step (c) of Figure 6, w_0 is multiplied with the encrypted data $[x]_C$ and w_1 is multiplied with the input that is rotated by one position (i.e., $[x']_C$). As a result, the server gets n_o multiplied ciphertext, $\{[u_i]_C\}$. The entries in each of $\{[u_i]_C\}$ are partial sums of the elements in the matrix-vector multiplication wx . For example, as shown in step (c) of Figure 6, the server obtains two multiplied ciphertext (i.e., $[u_0]_C$ and $[u_1]_C$) whose elements are partial sums of the first and second elements of wx (i.e., $(A1M1 + A2M2 + A3M3 + A4M4)$ and $(B1M1 + B2M2 + B3M3 + B4M4)$). Then the server sums them up elementwise, to form another ciphertext, which is

the vector in the middle of step (d) in Figure 6. At this point, similar to the naive method, the server proceeds with $\log_2 \frac{n_i}{n_o}$ RaS iterations and finally obtains a single ciphertext whose first n_o entries are the corresponding n_o elements of $\mathbf{w}\mathbf{x}$ (see the first two elements of the vector after RaS in step (d)).

Furthermore, as the number of slots n in a ciphertext is always larger than the dimension of the input vector, n_i , the computation cost is further reduced by packing copies of input \mathbf{x} as much as possible to form $[\mathbf{x}_{\text{pack}}]_C$. Thus $[\mathbf{x}_{\text{pack}}]_C$ has $\frac{n}{n_i}$ copies of \mathbf{x} and the server is able to multiply $\frac{n}{n_i}$ encoded vectors with $[\mathbf{x}_{\text{pack}}]_C$ by one ScMult operation. Therefore, the server gets $\frac{n_i n_o}{n}$ rather than n_o multiplied ciphertext. The resultant single ciphertext now has $\frac{n}{n_o}$ rather than $\frac{n_i}{n_o}$ blocks. The server then applies $\log_2 \frac{n}{n_o}$ RaS iterations to get the final ciphertext, whose first n_o entries are the n_o elements of $\mathbf{w}\mathbf{x}$.

The hybrid method requires $\frac{n_i n_o}{n}$ scalar multiplications (ScMult), $\frac{n_i n_o}{n} - 1$ HstPerm rotations for $[\mathbf{x}_{\text{pack}}]_C$, $\log_2 \frac{n}{n_o}$ Perm rotations, and $\frac{n_i n_o}{n} + \log_2 \frac{n}{n_o} - 1$ additions (Add). There is only one output ciphertext, which efficiently improves the slot utilization compared to the naive method.

3) Row-encoding-share-RaS Multiplication (GALA): The proposed GALA framework is motivated by two observations on the hybrid method. First, the hybrid method essentially strikes a tradeoff between Perm and HstPerm operations, where the number of Perms (which is the most expensive HE operation) is proportional to the number of slots in a ciphertext. This is not desired as we prefer a large n to pack more data for efficient SIMD HE. GALA aims to make the number of Perm operations disproportional to the number of slots and eliminate all HstPerm operations on the input ciphertext.

The second observation is the $\log_2 \frac{n}{n_o}$ RaS operations. We discover that this is actually unnecessary. Specifically, the unique feature in the HE-GC neural network framework is that the resultant single ciphertext from linear computing is shared between the client and server, to be the input for the nonlinear computing in the next phase. As the shares are

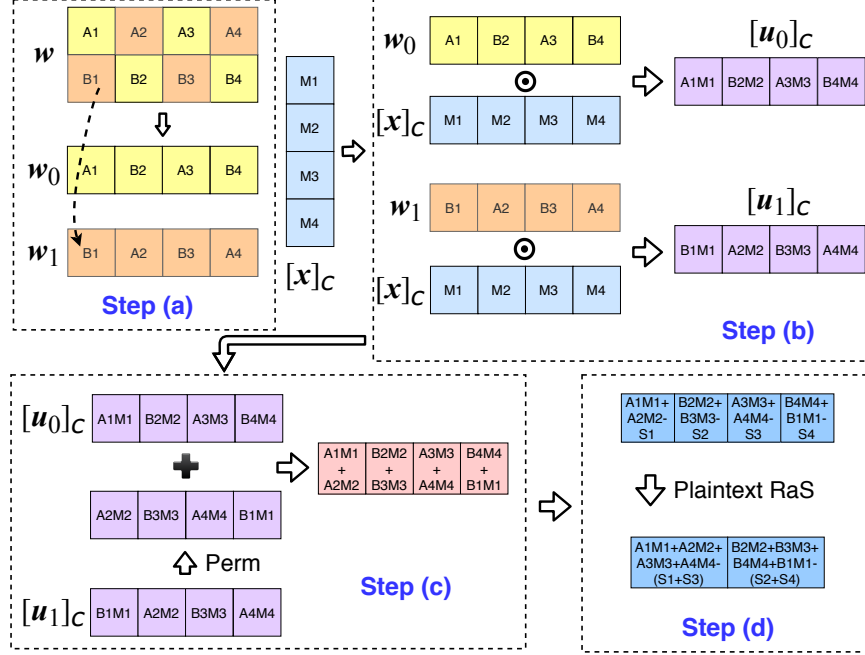


Figure 7. Row-encoding-share-RaS multiplication.

in plaintext, we propose to transfer the final $\log_2 \frac{n}{n_o}$ RaS operations in the HE domain to $\log_2 \frac{n}{n_o}$ RaS operations in plaintext. This significantly reduces expensive Perm operations. For example, multiplying a 16×128 matrix with a length-128 vector by our proposed scheme shows about $19\times$ speedup compared with the hybrid method [64] on a commodity machine (see detailed benchmarks in Section 3.5).

Figure 7 illustrates GALA’s matrix-vector calculation. The server first conducts the row-wise weight matrix encoding which encodes w into n_o plaintext vectors in a diagonal manner, as shown in step (a) in Figure 7. Compared with the hybrid method, the row-wise weight matrix encoding of GALA enables the server to directly multiply w_i and $[x]_c$, eliminating the Perm operations on $[x]_c$ in step (b). Furthermore, the encoding also benefits the noise management in the resultant to-be-shared ciphertext as to be analyzed in Section 3.3.3.

As a result, the server gets n_o multiplied ciphertext, $\{[u_i]_c\}$, such that the first entry of $[u_i]_c$ is a partial sum of the i -th element of the matrix-vector multiplication wx . For example, in step (b) of Figure 7, the first element A1M1 in $[u_0]_c$ is a partial sum of the

first element of \mathbf{wx} (i.e., $A1M1 + A2M2 + A3M3 + A4M4$), and the first element in $[\mathbf{u}_1]_C$ is a partial sum of the 2nd element of \mathbf{wx} (i.e., $B1M1 + B2M2 + B3M3 + B4M4$). Then, the server conducts rotations on each $[\mathbf{u}_i]_C$, with totally $(n_o - 1)$ Perm operations excluding the trivial rotation by zero, to make the first entry of $[\mathbf{u}_i]_C$ to be a partial sum of the first element of \mathbf{wx} . Next, the server adds all of the rotated $[\mathbf{u}_i]_C$ to obtain a single ciphertext whose entries are repeatedly a partial sum of the elements of \mathbf{wx} . For example, in step (c) of Figure 7, $[\mathbf{u}_1]_C$ is rotated by one position and then added the $[\mathbf{u}_0]_C$ to get one ciphertext, whose entries are the partial sum of the first and second elements of \mathbf{wx} .

Till now, the natural next step is to conduct $\log_2 \frac{n_i}{n_o}$ RaS iterations to get a final ciphertext whose first n_o entries are the n_o elements of \mathbf{wx} , i.e., the approach used by the hybrid method [64, 65]. With GALA, we propose to eliminate the $\log_2 \frac{n_i}{n_o}$ time-consuming RaS iterations by integrating it with the generation of shares for the GC-based nonlinear computing.

As introduced in the hybrid method [64, 65], in order to do the GC-based nonlinear computing, the encrypted linear output is shared as follows: (1) the server generates a random vector; (2) the server subtracts the random vector from the ciphertext (the encrypted linear output); (3) the subtracted ciphertext is sent to the client, which subsequently decrypts it and obtains its share.

Here we let the server encode a similar random vector and subtract it from the ciphertext obtained in step (c) of Figure 7. The subtracted ciphertext is sent to the client, which decrypts ciphertext, and then applies $\log_2 \frac{n_i}{n_o}$ RaS iterations on the plaintext, as illustrated in step (d) of Figure 7. Similarly, the server gets its share by $\log_2 \frac{n_i}{n_o}$ plaintext RaS iterations on its encoded random vector. Hence, in GALA, the server replaces the ciphertext RaS operations by much faster plaintext RaS operations. This significantly improves the computation efficiency.

Furthermore, in order to make use of all slots in a ciphertext, the client packs $\frac{n}{n_i}$ input \mathbf{x} to form a packed vector $[\mathbf{x}_{\text{pack}}]_C$. Then the server multiplies $\frac{n}{n_i}$ encoded weight vectors with

$[\mathbf{x}_{\text{pack}}]_{\mathcal{C}}$ by one ScMult operation. As a result, the server obtains $\frac{n_i n_o}{n}$ multiplied ciphertext, which are respectively rotated to enable the elementwise sum, finally producing a single ciphertext that has $\frac{n}{n_o}$ to-be-accumulated blocks. Without any further HE RaS iterations, the server then starts to encode the random vector for the share generation. The only extra computation is the plaintext RaS iteration(s) at both the client and server, which is much faster compared to the ones in the HE domain.

As a result, GALA needs $\frac{n_i n_o}{n}$ ScMult operations, $(\frac{n_i n_o}{n} - 1)$ Perm operations, and $(\frac{n_i n_o}{n} - 1)$ Add operations. It yields one output ciphertext and makes efficient utilization of ciphertext slots. Table 4 compares the complexity among the naive method, the hybrid method (i.e., GAZELLE) and the proposed row-encoding-share-RaS matrix-vector multiplication (GALA). We can see that the proposed method completely eliminates the HstPerm operations⁵ and significantly reduces the Perm operations.

Table 4. Complexity comparison of three methods.

Method	# Perm	# HstPerm	# ScMult	# Add
Naive	$n_o \log_2 n_i$	0	n_o	$n_o \log_2 n_i$
GAZELLE	$\log_2 \frac{n}{n_o}$	$\frac{n_i n_o}{n} - 1$	$\frac{n_i n_o}{n}$	$\log_2 \frac{n}{n_o} + \frac{n_i n_o}{n} - 1$
GALA	$\frac{n_i n_o}{n} - 1$	0	$\frac{n_i n_o}{n}$	$\frac{n_i n_o}{n} - 1$

3.3.2 KERNEL GROUPING BASED CONVOLUTION

In this subsection, we introduce GALA’s optimization for convolution. Similar to the discussion on the matrix-vector multiplication, we first begin with the basic convolution for the Single Input Single Output (SISO), then go through the state-of-the-art scheme for

⁵Based on GAZELLE’s implementation, the computation cost for a series of Perm operations on the same ciphertext can be optimized by first conducting one Perm Decomposition (DecPerm) on the ciphertext and then doing the corresponding series of Hoisted Perm (HstPerm) operations [64]. Since only one DecPerm is involved, it can amortize the total permutation time.

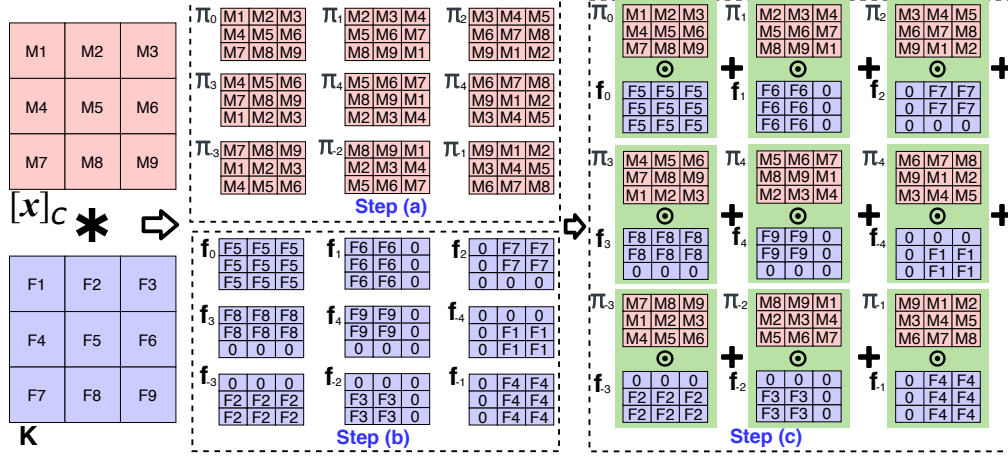


Figure 8. SISO convolution.

the Multiple Input Multiple Output (MIMO) (i.e., the GAZELLE framework [64]). Finally we elaborate GALA's first-Add-second-Perm (*kernel grouping*) scheme that achieves more efficient convolution computation. We assume the server has c_o plaintext kernels with a size of $k_w \times k_h \times c_i$ and the client sends to the server the encrypted data in the size of $u_w \times u_h$ with c_i channels. The server needs to homomorphically convolve the encrypted data from the client with its plaintext kernels to produce the encrypted output.

1) Basic SISO convolution: SISO is a special case of MIMO where $c_i = c_o = 1$. In this case, the encrypted data from the client has a size of $u_w \times u_h$ with one channel (i.e., a 2D image) and there is only one kernel with size $k_w \times k_h$ (i.e., a 2D filter) at the server. The SISO convolution is illustrated by an example in Figure 8 where $[x]_C$ is the encrypted data from the client and K is the plaintext kernel at the server. The process of convolution can be visualized as placing the kernel K at different locations of the input data $[x]_C$. At each location, a sum of an element-wise product between the kernel and corresponding data values within the kernel window is computed. For example, in Figure 8, the first value of the convolution between $[x]_C$ and kernel K is $(M1F5 + M2F6 + M4F8 + M5F9)$. It is obtained by first placing the center of K , i.e., $F5$, at $M1$ and then calculating the element-wise product between K and the part of $[x]_C$ that is within K 's kernel window (i.e., $M1, M2, M4$ and $M5$).

The final result is the sum of the element-wise product. The rest of convolution values are calculated similarly by placing F5 at M2 to M9.

We now elaborate the convolution by an example when F5 is placed at M5 (i.e., the central element of $[\mathbf{x}]_C$). In this example, the kernel size is $k_w k_h = 9$. The convolution is derived by summing the element-wise product between the 9 values in K and the corresponding 9 values around M5. This can be achieved by rotating $[\mathbf{x}]_C$ in a raster scan fashion [64]. Specifically, $[\mathbf{x}]_C$ is converted to a vector by concatenating all rows. Then, it is rotated by $(k_w k_h - 1)$ rounds, with half of them in the forward direction and the other half in the backward direction. We denote π_j as the rotation by j positions, where a positive sign of j indicates the forward direction and negative the backward direction, as shown in step (a) of Figure 8.

The convolution is obtained by (1) forming the kernel coefficients according to the partial sum at the corresponding location as shown in step (b) of Figure 8, (2) scaling the 9 rotated π_j with the corresponding kernel coefficients, and (3) summing up all scaled π_j (see step (c)). The rotation for $[\mathbf{x}]_C$ is completed by HstPerm⁶. The scaling is done by ScMult and the summation is achieved by Add. Therefore, the SISO convolution requires a total of $(k_w k_h - 1)$ HstPerm operations (excluding the trivial rotation by zero), $k_w k_h$ ScMult operations and $(k_w k_h - 1)$ Add operations. The output is one ciphertext⁷ which contains the convolution result.

2) Output Rotation based MIMO convolution (GAZELLE): We now consider the more general case, i.e., MIMO, where c_i or c_o is not one. The naive approach is to directly apply SISO convolution by first encrypting the c_i input channels into c_i ciphertext, $\{[\mathbf{x}_i]_C\}$. Each of the c_o kernels includes c_i filters. Each $[\mathbf{x}_i]_C$ is convolved with one of the c_i filters by SISO and the final convolution is obtained by summing up all of the c_i SISO convolutions. As a result, the naive approach requires $c_i(k_w k_h - 1)$ HstPerm operations (for c_i input

⁶With a common DecPerm operation.

⁷We assume the input size $u_w u_h$ is smaller than the ciphertext size n .

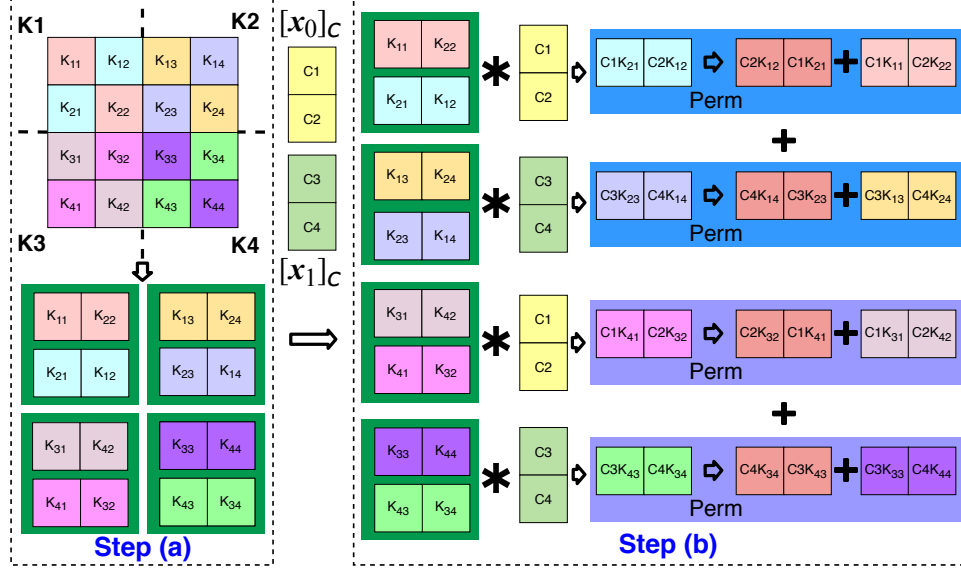


Figure 9. MIMO convolution.

channels), $c_i c_o k_w k_h$ ScMult operations and $c_o(c_i k_w k_h - 1)$ Add operations. There are c_o output ciphertext.

Given the number of slots n in a ciphertext is usually larger than the channel size $u_w u_h$, the ciphertext utilization (i.e., the meaningful slots that output desired results) in the c_o output ciphertext is low.

In order to improve the ciphertext utilization and computation efficiency for MIMO convolution, the state-of-the-art method (i.e., the output rotation [64]) first packs c_n channels of input data into one ciphertext, which results in $\frac{c_i}{c_n}$ input ciphertext (see Figure 9 where the four input channels form two ciphertext, each of which includes two channels). Meanwhile, the c_o kernels are viewed as a $c_o \times c_i$ kernel block and each row of the block includes c_i 2D filters for one kernel. Then the MIMO convolution is viewed as a matrix-vector multiplication where the element-wise multiplication is replaced by convolution. As each ciphertext holds c_n channels, the kernel block is divided into $\frac{c_o c_i}{c_n^2}$ blocks (see step (a) in Figure 9, where the kernel block is divided into K1 to K4).

Next, each divided block is diagonally encoded into c_n vectors such that the first filters in

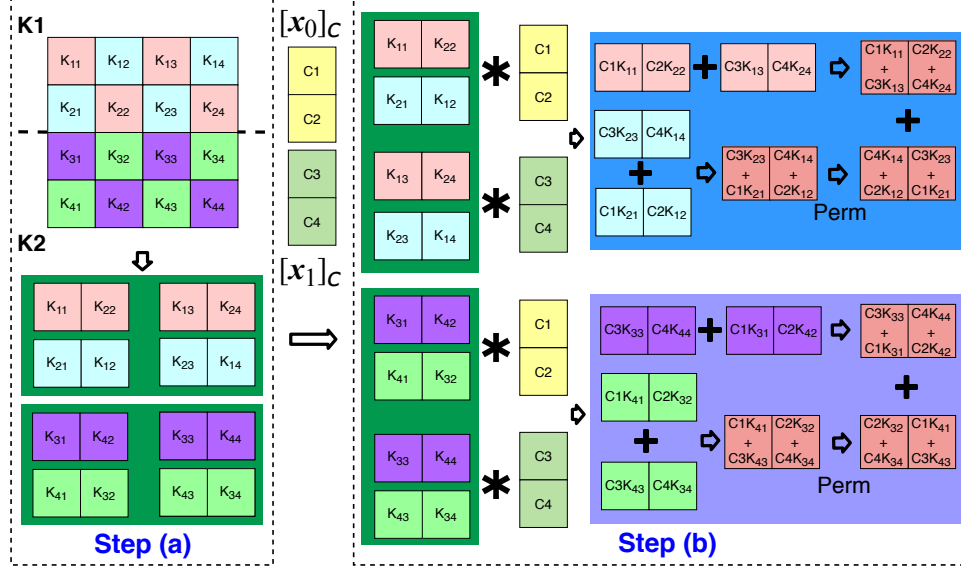


Figure 10. Kernel grouping based MIMO convolution.

all vectors are in the first column of the kernel block (see the four groups of vectors in step (a) of Figure 9). In this way, each input ciphertext can directly convolve with the vectors in each divided block by SISO, and the convolution for each divided block is obtained by rotating the c_n convolved vectors to the same kernel order as the diagonal one and summing them up (see step (b)).

Finally, the convolution for c_n kernels is calculated by adding the convolution of $\frac{c_i}{c_n}$ blocks associated with the same kernels as illustrated in step (b) of Figure 9.

Clearly, there are $\frac{c_o}{c_n}$ output ciphertext, as expected. For each of the $\frac{c_o c_i}{c_n^2}$ blocks, there are totally c_n SISO-like convolutions, requiring $c_n k_w k_h$ ScMult operations, $(c_n - 1)$ Perm operations and $(c_n k_w k_h - 1)$ Add operations. Next, there are $\frac{c_i}{c_n}$ block convolutions which are associated with the same kernel order. Thus, they are added up to obtain the final convolution result. Meanwhile, the rotation group for each input ciphertext is reused to convolve with different kernel blocks. Thus, there are $\frac{c_i(k_w k_h - 1)}{c_n}$ HstPerm operations with $\frac{c_i}{c_n}$ common DecPerm operations. In all, the MIMO convolution needs a total of $\frac{c_i c_o}{c_n^2}(c_n - 1)$ Perm, $\frac{c_i}{c_n}(k_w k_h - 1)$ HstPerm, $k_w k_h \frac{c_i c_o}{c_n}$ ScMult and $\frac{c_o}{c_n}(c_i k_w k_h - 1)$ Add operations.

3) Kernel Grouping Based MIMO convolution (GALA): One key observation on the above MIMO convolution is that, each of the $\frac{c_o c_i}{c_n^2}$ blocks needs $(c_n - 1)$ expensive Perm operations in order to get the convolution for that block. However, we actually do not need to get the convolution for each block. As our goal is to get the convolution for each kernel, the blocks that are associated with the same kernel are combined in our proposed first-Add-second-Perm approach (*kernel grouping*) to reduce the Perm cost. Specifically, in step (a) of Figure 10, the whole kernel block is divided into two blocks K1 and K2 such that each block is the combination of $\frac{c_i}{c_n}$ c_n -by- c_n divided blocks, which correspond to the same kernels (i.e., the first and second kernel in K1 and the third and fourth kernel in K2).

For each newly formed block, all of the vectors are first convolved with the corresponding input ciphertext by SISO-like convolution. Then the convolved vectors that are associated with the same kernel order are first added together (see the addition of convolved vectors before rotation in step (b) of Figure 10). Finally, these added vectors are rotated to the same kernel order and summed up to obtain the convolution result (see the rotation and final addition for each block in step (b) of Figure 10).

This kernel grouping calculation results in $(c_n - 1)$ Perm operations for each of $\frac{c_o}{c_n}$ newly formed blocks, which reduces the Perm complexity by a factor of $\frac{c_i}{c_n}$ compared with GAZELLE's MIMO convolution. This reduction is nontrivial especially for the state-of-the-art neural networks such as ResNets [26], where $\frac{c_i}{c_n}$ can be 256. This is because these neural networks contain a large number of large-size feature maps in order to capture the complex input features [15, 16, 26].

Similar to the output rotation based MIMO convolution discussed above, there are $\frac{c_o}{c_n}$ output ciphertext in the proposed scheme. For each of the $\frac{c_o}{c_n}$ newly formed blocks, there are c_i SISO-like convolutions. Then for each of the c_n kernel orders, there are $\frac{c_i}{c_n}$ convolutions to be summed up, which results in c_n added convolutions. These added convolutions are further rotated to the same kernel order and summed up to get the final convolution. Therefore, the proposed MIMO convolution requires a total of $\frac{c_o}{c_n}(c_n - 1)$ Perm, $\frac{c_i}{c_n}(k_w k_h - 1)$ HstPerm,

$k_w k_h \frac{c_i c_o}{c_n}$ ScMult, and $\frac{c_o}{c_n}(c_i k_w k_h - 1)$ Add operations.

Table 5 compares the overall complexity for convolution computations. GALA’s kernel grouping approach reduces the expensive Perm operations by a factor of $\frac{c_i}{c_o}$ without increasing other operations compared with the output rotation based MIMO convolution (i.e., the GAZELLE framework). The reduction in Perm operations leads to a significant speedup. Specifically, GALA shows about $14\times$ speedup compared with GAZELLE in the convolution between input data with a size of 16×16 with 2048 channels, and 512 kernels with a size of $1\times 1@2048$ on a commodity machine (see detailed benchmarks in Section 3.5).

Table 5. Complexity comparison of convolution.

Method	# Perm	# HstPerm [‡]	# ScMult	# Add
GAZELLE	$\frac{c_i c_o (c_n - 1)}{c_n^2}$	$\frac{c_i (k_w k_h - 1)}{c_n}$	$\frac{c_i c_o k_w k_h}{c_n}$	$\frac{c_o (c_i k_w k_h - 1)}{c_n}$
GALA	$\frac{c_o (c_n - 1)}{c_n}$	$\frac{c_i (k_w k_h - 1)}{c_n}$	$\frac{c_i c_o k_w k_h}{c_n}$	$\frac{c_o (c_i k_w k_h - 1)}{c_n}$

[‡]Rotations of the input with $\frac{c_i}{c_n}$ common DecPerm operations.

3.3.3 NOISE MANAGEMENT

The packed HE (e.g., the BFV scheme) introduces noise in the ciphertext which theoretically hides the original message [51, 64]. However, the noise management is critical to the correct decryption of ciphertext after a series of HE operations. We will show that GALA has better noise management compared with GAZELLE.

Based on the computation complexity of matrix-vector multiplication and convolution, along with the noise change for HE operations as described in Section 3.2, Table 6 shows the noise growth of different schemes. As for the matrix-vector multiplication, GALA has a lower noise growth while keeping the number of output ciphertext as small as one⁸. As for

⁸Note that the noise in Table 6 is calculated by assuming $(\frac{n_i n_o}{n} - 1) \geq 0$. The noise of GALA is still lower than that of GAZELLE when $(\frac{n_i n_o}{n} - 1) < 0$ as it means one ciphertext can hold data with size $n_o \times n_i$,

Table 6. Comparison of noise management.

Matrix-vector Multiplication		
Method	Noise after computation	# Cipher
Naive	$n_i \eta_0 \eta_{mult} + (n_i - 1) \eta_{rot}$	n_o
GAZELLE	$n_i \eta_0 \eta_{mult} + [\frac{n_i n_o - n}{n_o} \eta_{mult} + \frac{n - n_o}{n_o}] \eta_{rot}$	1
GALA	$\frac{n_i n_o}{n} \eta_0 \eta_{mult} + (\frac{n_i n_o}{n} - 1) \eta_{rot}$	1
Convolution Computation		
Method	Noise after computation	# Cipher
GAZELLE	$c_i \eta_{\Delta} + \frac{c_i}{c_n} (c_n - 1) \eta_{rot}$	$\frac{c_o}{c_n}$
GALA	$c_i \eta_{\Delta} + (c_n - 1) \eta_{rot}$	$\frac{c_o}{c_n}$
$\eta_{\Delta} = k_w k_h \eta_{mult} \eta_0 + (k_w k_h - 1) \eta_{rot} \eta_{mult}$		

the convolution computation, GALA reduces the noise term associated with rotation by a factor of $\frac{c_i}{c_n}$ compared to GAZELLE. This is nontrivial especially for state-of-the-art neural networks such as **ResNets** [26], where $\frac{c_i}{c_n}$ can be 256. The number of output ciphertext is also maintained as small as $\frac{c_o}{c_n}$. Overall, GALA features a lower noise growth and lower computation complexity compared with GAZELLE.

3.4 SECURITY ANALYSIS

GALA is based on the same security framework as GAZELLE [64]. The security proof is based on the simulation for the ideal world and the real world as described in Section 2.2⁹. Generally, the security of linear computation in GALA is fully protected by the security of HE (e.g., the BFV scheme [51, 52]). The nonlinear computation (which is not the focus of this chapter) is protected by Garbled Circuits (GC) [104] or its alternatives. The security of GC-based nonlinear computation has been proven in TASTY [139] and MP2ML [133].

3.5 PERFORMANCE EVALUATION

We conduct the experiments in both LAN and WAN settings. The LAN setting is which only involves one ScMult operation in GALA, and GAZELLE needs to subsequently conduct a series of RaS operations.

⁹We refer the readers to Section 6.4 for a more concrete and specific elaboration.

implemented on a Gigabit Ethernet in our lab between two workstations as the client and server, respectively. Both machines run Ubuntu, and have an Intel i7-8700 3.2GHz CPU with 12 threads and 16 GB RAM. The WAN setting is based on a connection between a local PC and an Amazon AWS server with an average bandwidth about 200Mbps and round-trip time around 13ms. We have downloaded the codes released by GAZELLE¹⁰, DELPHI¹¹ and CrypTFlow2¹², and have run all experiments on the same hardware devices and network settings. We conducted a series of experiments under various neural network architectures. In each experiment, we first ran the baseline algorithm (i.e., GAZELLE, DELPHI or CrypTFlow2) to obtain the baseline total runtime (including online runtime and offline runtime), and then replaced the linear computation of the baseline algorithm by GALA to get a new total runtime, which was then used to compute the speedup.

While the codes for GAZELLE, DELPHI and CrypTFlow2 are implemented in different ways (for example, GAZELLE is based on its crypto platform while DELPHI and CrypTFlow2 are based on the Microsoft SEAL library), we focus on the speedup of GALA on top of each of them. We also set the cryptographic parameters in line with GAZELLE: 1) Parameters for both HE and GC schemes were selected for a 128-bit security level. 2) A plaintext modulus p of 20 bits is enough to store all the intermediate values in the network computation. 3) The ciphertext modulus q was chosen to be a 60-bit pseudo-Mersenne prime that is slightly smaller than the native machine word on a 64-bit machine to enable lazy modular reductions. 4) The selection of the number of slots is the smallest power of two that allows for a 128-bit security which in our case is $n = 2048$. We refer readers to [64] for more details about the parameter selection.

3.5.1 MICROBENCHMARKS

In this section, we benchmark and compare the runtime of GALA’s linear optimization

¹⁰Available at https://github.com/chiraag/gazelle_mpc

¹¹Available at <https://github.com/mc2-project/delphi>

¹²Available at <https://github.com/mpc-msri/EzPC/tree/master/SCI>

(i.e., matrix-vector multiplication and convolution computation) with state-of-the-art approaches. We claim the same communication cost and inference accuracy with GAZELLE and achieve improved computation efficiency.

1) Matrix-Vector Multiplication: Table 7 compares the computation complexity of GALA’s matrix-vector optimization with GAZELLE and two other optimization schemes (i.e., a diagonal method (Diagonal) [138] and an extended method (Extended) [84]). We can see that GALA largely reduces the expensive Perm operation to zero in our cases (including the HstPerm) while GAZELLE needs up to 11 Perm and Extended [84] needs up to 520 Perm (including HstPerm). On the other hand, GALA also maintains a light overhead for HE multiplication/addition, i.e., only one multiplication, compared with the other three optimizations, e.g., Diagonal [138] and Extended [84] involve up to 2048 multiplications/additions.

The runtime results for matrix-vector multiplication are summarized in Table 8, which includes the original runtime of GAZELLE, DELPHI and CrypTFlow2, and the speedup of GALA on top of each. We take the share-RaS calculation cost (see the plaintext computing for final share at the client in step (d) of Figure 7) as part of the runtime cost of GALA for fair comparison. Meanwhile, as multiple copies are packed in one ciphertext, the HstPerm operation includes a common DecPerm to enable hoist optimization for rotation (see the details in [64]). As can be seen from Table 8, GALA’s optimization gains a large speedup due to the row-encoding-share-RaS module, which reduces the costly Perm, Mult, and Add operations for a series of RaS calculations. Specifically, GALA achieves the speedup of $1795\times$, $208\times$ and $57\times$ over the Diagonal [138] under different matrix dimensions in the LAN setting. This benefit stems from the fact that the computation complexity of the Diagonal is related to the input dimension n_i , which is always large in the state-of-the-art neural networks such as AlexNet [15], VGG [16] and ResNets [26]. For a similar reason, GALA significantly outperforms the Extended method [84].

Meanwhile, GALA has a speedup of $59\times$, $13\times$ and $19\times$ over GAZELLE under different

Table 7. Computation complexity of matrix-vector multiplication.

Dimension ($n_o \times n_i$): 1×2048				
Metric	Diagonal [138]	GAZELLE	Extended [84]	GALA
# Perm	0	11	0	0
# HstPerm [‡]	2047	0	2047	0
# ScMult	2048	1	2048	1
# Add	2047	11	2047	0
Dimension ($n_o \times n_i$): 2×1024				
Metric	Diagonal [138]	GAZELLE	Extended [84]	GALA
# Perm	0	10	9	0
# HstPerm [‡]	1023	0	511	0
# ScMult	1024	1	512	1
# Add	1023	10	520	0
Dimension ($n_o \times n_i$): 16×128				
Metric	Diagonal [138]	GAZELLE	Extended [84]	GALA
# Perm	0	7	4	0
# HstPerm [‡]	127	0	7	0
# ScMult	128	1	8	1
# Add	127	7	11	0

[‡]Rotations of the input with a common DecPerm

matrix dimensions in the LAN setting. This computation gain comes from the HstPerm-free scheme (i.e., row-encoding) and elimination of RaS computation (i.e., share-RaS scheme) compared to GAZELLE, which is particularly effective for large $\frac{n_i}{n_o}$ ratio and large ciphertext slots (see the superior performance for the neural network with a dimension of 1×2048). These features well suit the current convolutional neural networks which have tens of thousands of values to be fed into the fully connected layers [16, 26].

Compared with DELPHI and CrypTFlow2, GALA achieves a speedup for weight matrix multiplication up to $700\times$ in the LAN setting. This is largely due to GALA’s deep optimization for HE computation. We also notice that GALA’s speedup slows down in WAN which is due to the communication rounds needed for conversions between HE and GC. Therefore it leads to significant round time in total compared with the light HE computation overhead. For example, the round-trip time is around 13 milliseconds while the GALA’s optimized HE cost is within one millisecond.

2) Convolution Computation: We benchmark and compare the computation complexity

Table 8. Runtime cost of matrix-vector multiplication.

Dimension ($n_o \times n_i$): 1×2048					
Approach	Comm. (MB)	LAN (ms)		WAN (ms)	
		Time	Speedup	Time	Speedup
Diagonal [138]	0.03	57	1795 \times	75	4 \times
Extended [84]	0.03	57.5	1796 \times	77	4 \times
GAZELLE [64]	0.03	1.9	59 \times	19.3	1 \times
DELPHI [66]	0.14	28	700 \times	59.5	3.2 \times
CrypTFlow2 [7]	0.13	28	700 \times	46.2	2.5 \times
Dimension ($n_o \times n_i$): 2×1024					
Diagonal [138]	0.03	28	208 \times	47	2.5 \times
Extended [84]	0.03	16	116 \times	36	1.9 \times
GAZELLE [64]	0.03	1.8	13 \times	19	1 \times
DELPHI [66]	0.13	26.5	176 \times	57.8	3.1 \times
CrypTFlow2 [7]	0.13	26.5	176 \times	44.7	2.4 \times
Dimension ($n_o \times n_i$): 16×128					
Diagonal [138]	0.03	3.7	57 \times	21	1 \times
Extended [84]	0.03	1	16 \times	20.4	1 \times
GAZELLE [64]	0.03	1.2	19 \times	21	1 \times
DELPHI [66]	0.13	20.5	292 \times	51.7	2.8 \times
CrypTFlow2 [7]	0.13	20.5	292 \times	38.7	2.1 \times

and runtime of GALA with GAZELLE, DELPHI and CrypTFlow2 for convolution calculation. The details are illustrated in Tables 9 and 10. As for the computation complexity, we compare GALA with GAZELLE whose privacy-preserving convolution calculation over HE is one of the most optimized methods in current literature. While introducing no extra HE multiplication/addition, GALA reduces the most expensive Perm, i.e., DecPerm and HstPerm, by up to $59\times$ for input size of $16 \times 16 @ 2048$ with kernel size of $1 \times 1 @ 512$. This block with large channels and small kernel size is featured in state-of-the-art neural networks such as **ResNets** [26], which makes GALA suitable to boost the modern networks.

As for runtime comparison shown in Table 10, GALA demonstrates $9\times$, $14\times$ and $2.6\times$ speedup over GAZELLE with different input and kernel dimensions in LAN setting. As analyzed in Section 3.3.2, due to the fundamental complexity reduction by GALA’s kernel grouping approach, GALA reduces the expensive Perm operation by a factor of $\frac{c_i}{c_n}$. As we mention above, the large speedup is achieved under large input channels and small

Table 9. Computation complexity of convolution.

Input [†]	Kernel [‡]	Metric	GAZELLE [64]	GALA
16×16@128	1×1@128	# DecPerm	1792	112
		# HstPerm	1792	112
		# ScMult	2048	2048
		# Add	2032	2032
16×16@2048	1×1@512	# DecPerm	114944	2048
		# HstPerm	114688	1792
		# ScMult	131072	131072
		# Add	130944	130944
16×16@128	3×3@128	# DecPerm	1808	128
		# HstPerm	1920	240
		# ScMult	18432	18432
		# Add	18416	18416
16×16@2048	5×5@64	# DecPerm	14592	312
		# HstPerm	20480	6200
		# ScMult	409600	409600
		# Add	409592	409592

[†]Dim. is in the form of $u_w \times u_h @ c_i$

[‡]Dim. is in the form of $k_w \times k_h @ c_o$ with c_i channels per kernel

kernel sizes, so the proposed approach fits very well with state-of-the-art networks such as **ResNets** [26], where the feature maps are always with large channels (which results in large c_i while c_n is fixed) and small kernels (that are usually 1×1, 3×3 and 5×5 at most, which benefit small HE multiplication/addition). Meanwhile, the speedup over DELPHI and CrypTFlow2 is up to 7.4× in the LAN setting. On the other hand, the speedup of GALA in the WAN setting is also decent, up to 8.7×, 6.3× and 6.5× for GAZELLE, DELPHI and CrypTFlow2, respectively. This is because the computation cost of convolution increases accordingly with regard to the communication cost, compared with the case of matrix-vector multiplication.

3.5.2 PERFORMANCE WITH CLASSIC NETWORKS

In this section, we benchmark the GALA performance on a 4-layer Multi Layer Perceptron (MLP)¹³ which is also adopted in other privacy preserving frameworks including

¹³MPL consists of FC layers. The adopted network structure is 784-128-128-10.

Table 10. Runtime cost of convolution.

Dimension (Input Dim. [†] , Kernel Dim. [‡]): 16×16@128, 1×1@128					
Approach	Comm. (MB)	LAN (ms)		WAN (ms)	
		Time	Speedup	Time	Speedup
GAZELLE	0.5	321	9×	408	3.2×
DELPHI	2.1	391	3.1×	502	2.3×
CrypTFlow2	2	389	3.1×	482	2.2×
Dimension (Input Dim. [†] , Kernel Dim. [‡]): 16×16@2048, 1×1@512					
GAZELLE	8	20583.5	14×	21784	8.7×
DELPHI	31	17939	4.4×	19205	3.7×
CrypTFlow2	29	17928	4.4×	19101	3.6×
Dimension (Input Dim. [†] , Kernel Dim. [‡]): 16×16@128, 3×3@128					
GAZELLE	0.5	457	2.6×	547	2.1×
DELPHI	2	2563.6	5.8×	2671	5×
CrypTFlow2	1.9	2559	5.8×	2648	5×
Dimension (Input Dim. [†] , Kernel Dim. [‡]): 16×16@2048, 5×5@64					
GAZELLE	8	5875.2	1.7×	7073	1.5×
DELPHI	31	56499	7.4×	57765	6.3×
CrypTFlow2	29	56409	7.4×	57582	6.5×

[†]Dim. is in the form of $u_w \times u_h @ c_i$

[‡]Dim. is in the form of $k_w \times k_h @ c_o$ with c_i channels per kernel

GAZELLE, SecureML [3] and MiniONN [65] as a baseline network, as well as state-of-the-art neural network models including AlexNet [15], VGG [16], ResNet-18 [26], ResNet-50 [26], ResNet-101 [26], and ResNet-152 [26]. We use MNIST dataset [140] for the MLP and CIFAR-10 dataset [141] for state-of-the-art networks.

Table 11 shows computation complexity of the proposed GALA compared with GAZELLE. We can see that GALA reduces GAZELLE’s Perm by 34×, 31×, 30×, 47×, 39×, and 36× for AlexNet, VGG, ResNet-18, ResNet-50, ResNet-101, and ResNet-152, respectively. The fundamental base for this speedup lies in GALA’s deep optimization for HE-based linear computation. We also notice that GALA achieves limited reduction of Perm in MLP (from 70 to 55). This is due to the small ratio between the number of slots in the ciphertext and output dimension in each layer, i.e, $\frac{n}{n_o}$, which limits the performance gain. The limited gain is also observed in Table 12 which shows the system speedup of GALA over GAZELLE, CrypTFlow2, DELPHI, SecureML and MiniONN. Specifically, GALA boosts

Table 11. Computation complexity of state-of-the-art neural network models.

Net. Frameworks	Metric	GAZELLE [64]	GALA
MLP	# Perm	70	55
	# ScMult	56	56
	# Add	70	55
AlexNet	# Perm	40399	1157
	# DecPerm	143	142
	# HstPerm	1493	1492
	# ScMult	481298	481298
	# Add	481096	481089
VGG	# Perm	66055	2115
	# DecPerm	161	160
	# HstPerm	1283	1280
	# ScMult	663556	663556
	# Add	663370	663363
ResNet-18	# Perm	180375	5921
	# DecPerm	483	482
	# HstPerm	3467	3464
	# ScMult	1399363	1399363
	# Add	1398778	1398771
ResNet-50	# Perm	1464119	30615
	# DecPerm	2819	2818
	# HstPerm	3863	3848
	# ScMult	2935408	2935408
	# Add	2931734	2931727
ResNet-101	# Perm	2560823	64887
	# DecPerm	6083	6082
	# HstPerm	8215	8200
	# ScMult	5302896	5302896
	# Add	5294326	5294319
ResNet-152	# Perm	3463991	95127
	# DecPerm	8963	8962
	# HstPerm	12055	12040
	# ScMult	7252592	7252592
	# Add	7239894	7239887

CrypTFlow2 by $2.3\times$ in the LAN setting. SecureML also gains $2.6\times$ in the LAN setting. Meanwhile, GALA’s performance is similar to GAZELLE and MiniONN. This is due to the relatively small network size and noticeable communication overhead (i.e., the large round time in total compared with computation cost). Nevertheless, none of the competing schemes achieves a better performance than GALA.

It is worth pointing out that the MLP network is not widely adopted in practical scenarios.

Table 12. Runtime cost of classic model.

Network Model: MLP					
Approach	Comm. (MB)	LAN (ms)		WAN (ms)	
		Time	Speedup	Time	Speedup
SecureML	0.21	31.9	2.6 \times	79.3	1.5 \times
MiniONN	4.4	14.1	1 \times	227.6	1 \times
GAZELLE	0.21	15	1 \times	84.9	1 \times
DELPHI	84	204.5	3.1 \times	3658.3	1 \times
CrypTFlow2	12.4	246	2.3 \times	780.6	1.2 \times

On the other hand, as the state-of-the-art deep neural networks utilize large channels and small-size kernels to capture data features while the size of feature maps is large, GALA is especially effective for accelerating such large state-of-the-art network models.

Table 13 shows the runtime of GAZELLE, DELPHI and CrypTFlow2, and the speedup of GALA on top of each. By reducing HE operations, especially Perm operations, GALA achieves noticeable boost over the GAZELLE, DELPHI and CrypTFlow2 frameworks. Specifically, the results show that GALA boosts GAZELLE by 2.5 \times (from 11s to 4.3s), 2.7 \times (from 18s to 6.5s), 3.2 \times (from 43s to 13s), 8.3 \times (from 276s to 33s), 7.7 \times (from 486s to 62s), and 7.5 \times (from 659s to 87s) in LAN setting, on AlexNet, VGG, ResNet-18, ResNet-50, ResNet-101, and ResNet-152, respectively.

CrypTFlow2 (CCS’20) is the latest framework for privacy preserved neural networks. It optimizes the nonlinear operations of DELPHI, and adopts a similar HE scheme of DELPHI for linear operations. GALA is an efficient plug-and-play module to optimize the linear operations of CrypTFlow2. As shown in Tables 8 and 10, GALA’s optimization of linear operations can further boost CrypTFlow2 by 700 \times and 7.4 \times for matrix-vector multiplication and convolution in the LAN setting, respectively. This speedup stems from GALA’s streamlined HE calculation compared with the one of CrypTFlow2. Slow-down is observed in the WAN setting, but CrypTFlow2 can still gain up to 6.5 \times speedup for convolution due to the computation-intensive nature for large input channels with small kernel dimensions featured in state-of-the-art network models. As for the overall system speedup, GALA can

Table 13. Runtime cost of state-of-the-art models.

Network Model: AlexNet					
Approach	Comm. (MB)	LAN (ms)		WAN (ms)	
		Time	Speedup	Time	Speedup
GAZELLE	17.45	11,019.2	2.5 ×	13,669.6	1.9 ×
DELPHI	617	90,090.1	2.9 ×	114,955	2 ×
CrypTFlow2	116.6	69,133.6	6.5 ×	73,876.8	4.8 ×
OT-based CrypTFlow2	2,108	226,431.7	21 ×	310,985.6	20 ×
Network Model: VGG					
GAZELLE	22.8	18,067.4	2.7 ×	21,566.2	2 ×
DELPHI	718.5	123,198.4	2.9 ×	152,176.4	1.5 ×
CrypTFlow2	150	97,038.9	6 ×	103,169.1	4.6 ×
OT-based CrypTFlow2	5,063.7	340,342.9	21 ×	543,242	24 ×
Network Model: ResNet-18					
GAZELLE	54	42,748.3	3.2 ×	51,032.7	2.3 ×
DELPHI	2,033.9	250,618.4	2.6 ×	332,524.2	1.9 ×
CrypTFlow2	354	190,684.7	5.7 ×	205,146.8	4.3 ×
OT-based CrypTFlow2	6,292.1	650,989.7	19.5 ×	903,492.6	19 ×
Network Model: ResNet-50					
GAZELLE	297.1	276,886.8	8.3 ×	321,600.2	4 ×
DELPHI	10,489	746,568.8	1.7 ×	1167,566.8	1.4 ×
CrypTFlow2	1,831	425,454.4	4.5 ×	499,429.6	2.9 ×
OT-based CrypTFlow2	13,104	1364,463.2	14.4 ×	3307,902.6	19 ×
Network Model: ResNet-101					
GAZELLE	603.1	486,745.2	7.7 ×	577,454.9	3.7 ×
DELPHI	22,199.4	1411,383.8	1.7 ×	2302,091.8	1.3 ×
CrypTFlow2	3,582.8	777,057.4	4.2 ×	921,735.6	2.8 ×
OT-based CrypTFlow2	23,857	2467,606.1	13.3 ×	6006,071.4	18.2 ×
Network Model: ResNet-152					
GAZELLE	873.1	659,833.7	7.5 ×	786,587	3.6 ×
DELPHI	29,433	1975,798.9	1.6 ×	3157,176.8	1.3 ×
CrypTFlow2	5,141	1065,103.4	4.1 ×	1272,772.6	2.7 ×
OT-based CrypTFlow2	32,804	3379,188.7	13 ×	8245,124.5	17.5 ×

boost CrypTFlow2 by 6.5×, 6×, 5.7×, 4.5×, 4.2×, and 4.1× in LAN, and by 4.8×, 4.6×, 4.3×, 2.9×, 2.8×, and 2.7× in WAN, based on the aforementioned network architectures.

It might appear counter-intuitive that while CrypTFlow2 is a more recent system than DELPHI, the speedup of GALA over DELPHI is smaller than its speedup over CrypTFlow2. This is because CrypTFlow2 has optimized the nonlinear part of DELPHI, significantly reducing its runtime. As a result, the runtime of linear operations in CrypTFlow2 accounts

Table 14. Percentages of linear computation in state-of-the-art neural network models.

Networks	GAZELLE	DELPHI	CrypTFlow2	Plaintext
AlexNet	97.7	76.9	98.7	98.5
VGG	98.2	77.9	98.8	98.1
ResNet-18	98.3	75.1	98.6	98.9
ResNet-50	98.5	55.2	96.8	97.9
ResNet-101	98.4	53.2	96.5	98.3
ResNet-152	98	52	96.4	98.4

Table 15. Accuracy with floating and fixed point in state-of-the-art neural network models. Top-1 accuracy: only the prediction with the highest probability is a true label; Top-5 accuracy: any one of the five model predictions with higher probability is a true label.

Network Models	Floating-point		Fix-point	
	Top1	Top5	Top1	Top5
AlexNet	78.89%	97.32%	78.43%	97.26%
VGG	92.09%	99.72%	92.05%	99.68%
ResNet-18	93.33%	99.82%	93.21%	99.81%
ResNet-50	93.86%	99.85%	93.86%	99.84%
ResNet-101	94.16%	99.79%	94.12%	99.79%
ResNet-152	94.23%	99.81%	94.15%	99.79%

for a very high percentage as illustrated in Table 14. Hence, CrypTFlow2 can benefit more from GALA’s optimization of linear computation, resulting in a higher speedup in terms of the overall runtime. It is worth pointing out that the ability to accelerate CrypTFlow2 is highly desirable since it is the latest privacy-preserving framework. Meanwhile, we also show GALA’s speedup on top of the OT-based CrypTFlow2 which relies on OT to complete the linear computation. As significant communication cost, including round cost, is involved in OT, the overhead of linear computation, especially in the WAN setting, increases compared with HE-based CrypTFlow2, which results in greater speedup achieved by GALA.

Next we examine the runtime breakdown of different layers for those six state-of-the-art networks as shown in Figure 11, which allows detailed observation. Note that the layer indexing here is slightly different from the original plaintext model for the sake of HE operations,

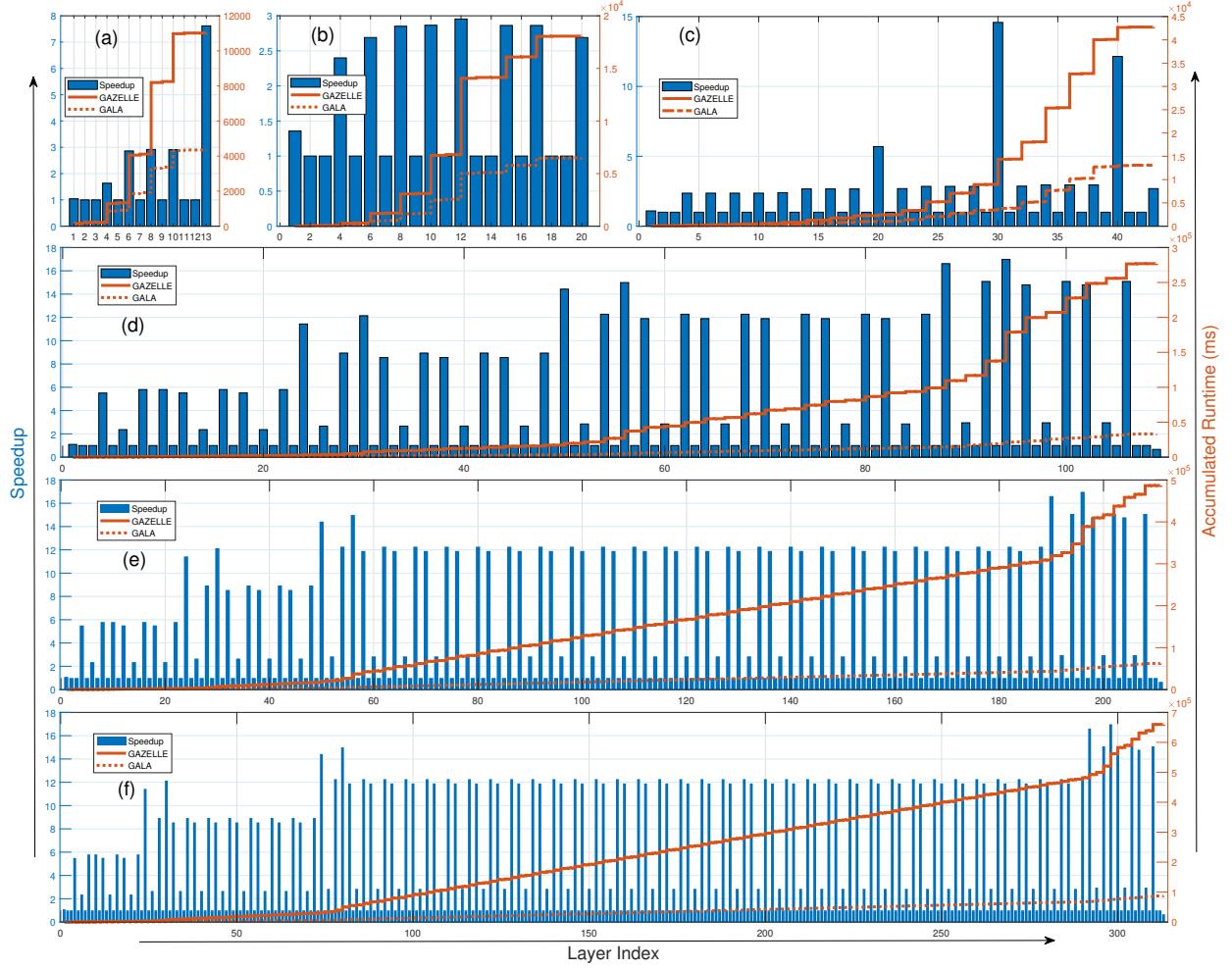


Figure 11. Layer-wise accumulated runtime and GALA speedup over GAZELLA on different networks: (a) AlexNet; (b) VGG; (c) ResNet-18; (d) ResNet-50; (e) ResNet-101; (f) ResNet-152. The bar with values on the left y-axis indicates speedup, and the curve with values on the right y-axis indicates the accumulated runtime. The layers with speedup of 1 are nonlinear layers.

e.g., the nonlinear activation or pooling following a convolution operation is counted as a separate layer. The x -axis of each subfigure in Figure 11 shows the layer index of a sequence of linear (convolution or matrix-vector multiplication) and nonlinear (activation or pooling) layers that constitute each network model. The y -axis plots the accumulated running time (milliseconds) up to a layer, and the speedup of GALA over GAZELLE in each layer.

For example, Figure 11 (a) illustrates the result for **AlexNet**. The most time-consuming computation in GAZELLE is in layers “6”, “8” and “10”, which are all convolution computations. This is evidenced by the large jump of runtime from these layers to the next layer. GALA decreases the time for these linear computations by nearly $3\times$. Meanwhile, the nonlinear layers (activation/pooling) have a speedup of 1, as GALA has the same computation cost as GAZELLE in those layers. Since the nonlinear computation contributes to only a small portion of the total cost, it does not significantly affect the overall performance gain of GALA that focuses on accelerating the linear computation. Note that GALA does not benefit much in the first layer of **AlexNet**, i.e., the first convolution, as the input has only three channels. However, the speedup for the following more costly convolutions allows GALA to effectively reduce the overall cost. A similar observation can be seen from the result on **VGG**. As for the four **ResNets** frameworks, the most significant performance gain stems from the convolution with 1×1 kernels. As **ResNets** repeat the blocks with multiple 1×1 convolution kernels, GALA effectively accelerates this type of convolution due to its deeply optimized linear computation mechanism (see details in Section 3.3.2), thus reducing the overall runtime. The similar trend is observed for **DELPHI** and **CryptFlow2**.

It is also worth mentioning that GALA focuses on optimizing the HE-based linear operations only and can be integrated into a baseline model (such as GAZELLE, **CryptFlow2**, or **DELPHI**). The proposed approach does not introduce approximation. Hence, it does not result in any accuracy loss compared to the baseline privacy preserved model. Furthermore, compared with the original plaintext model, the only possible accuracy loss in GALA comes from the quantification of floating point numbers to fixed point numbers in the HE

operations. Such quantification is indispensable in all HE-based frameworks including CryptFlow2. From our experiments, the model accuracy loss due to quantification is negligible, as shown in Table 15.

3.6 CHAPTER SUMMARY

This chapter has focused on a deep optimization on the HE-based linear computation in privacy-preserving neural networks. It aims to minimize the Perm operations, thus substantially reducing the overall computation time. To this end, we have proposed *GALA: Greedy computation for Linear Algebra*, which views the HE-based linear computation as a series of Homomorphic Add, Mult and Perm operations and chooses the least expensive operation in each linear computation step to reduce the overall cost. GALA can be a plug-and-play module integrated into HE-based systems to further boost their efficiency. GALA demonstrates an encouraging runtime boost by $2.5\times$, $2.7\times$, $3.2\times$, $8.3\times$, $7.7\times$, and $7.5\times$ over GAZELLE and $6.5\times$, $6\times$, $5.7\times$, $4.5\times$, $4.2\times$, and $4.1\times$ over CryptFlow2, on AlexNet, VGG, ResNet-18, ResNet-50, ResNet-101, and ResNet-152, respectively.

CHAPTER 4

OBSCURE COMPUTATION WITH PIGGYBACK

PROPAGATION IN PPDL

Recall from Section 2.1 that one layer of the CNN model includes linear and nonlinear computation. A deep optimization for the privacy-preserving linear computation, i.e., GALA, is presented in the previous chapter, which has demonstrated noticeable performance improvement over various DL models. In this chapter, we explore the optimization for a whole layer by replacing the GC-based nonlinear calculation with a newly-designed joint linear and non-linear computation based on the Homomorphic Secret Sharing. Furthermore, it expands the forward computation to enable backward propagation with a piggyback design that carefully devises the share set and integrates the dataflow of the whole training process. As such, we propose SecureTrain that achieves an inference speedup as high as $48\times$ compared with state-of-the-art inference frameworks. The rest of this chapter is organized as follows. Section 4.1 details the motivation of SecureTrain. Section 4.2 introduces the primitives that SecureTrain is based on. Section 4.3 describes the design details of SecureTrain. Section 4.4 presents the security analysis of SecureTrain. The experimental results are illustrated and discussed in Section 4.5. Finally, Section 4.6 concludes this chapter.

4.1 MOTIVATION

The success of DL relies on three core elements: massive computing power, expertise to construct good DL models, and large datasets for model training [15, 16, 142]. It is common that the entities possessing data are different than the organizations that own the computing power and DL expertise. For example, end users, enterprises, and regional Internet Service Providers (ISPs) possess a large volume of data, while the DL talent and

computing power are mostly gathered in technology giants such as Google and Microsoft. The former has a strong motivation to utilize the computing power and DL talent of the latter to solve challenging problems in networks, e.g., to optimize network design. However, a fundamental challenge is data privacy. For example, those data can have precious business value and need to be protected. Moreover, they can be sensitive and protected by laws from disclosure [33, 41, 107, 108, 143–145]. To this end, there has been a great interest in the research community to develop privacy-preserving DL systems, such as the CryptoNets [78], SecureML [3], MiniONN [65], EzPC [132], and Xonn [76].

Fig. 12 illustrates a privacy-preserving DL system to provide real-time, networked diagnosis to patients who are covered by Wireless Body Area Networks (WBANs). Each patient is monitored by various sensors, such as the ear temperature sensor, electrocardiograph and pulse oximeter. The health data of patients, e.g., temperature, heart rate and blood oxygen, is collected by WBANs and reported to a health provider such as a hospital. The latter sends the health data to a server in a cloud (e.g., the Microsoft cloud), which hosts a trained DL model to provide the health diagnosis service. To protect the privacy of sensitive data, the health provider (as a client) encrypts the data before sending to the server. The encrypted data is processed by the server on the crypto domain, and the diagnosis result, which is also on the crypto domain, is returned to the client. The client decrypts the result into plaintext, and uses it for better patient treatment. Data privacy is fully protected in this process by the underlying encryption scheme, such that sensitive patient information is not

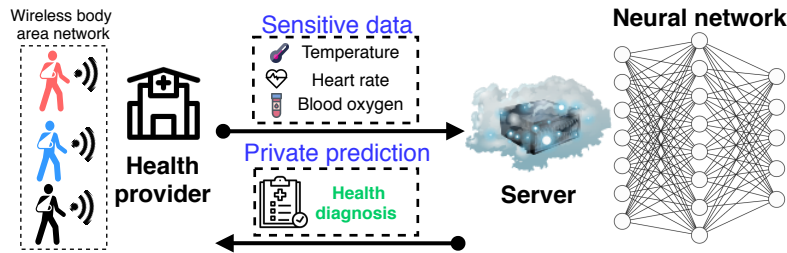


Figure 12. A framework of privacy-preserving inference.

leaked. Furthermore, the privacy-preserving DL system also aims to protect the intellectual property of the DL model on the server to ensure the users (such as health care providers) cannot learn the server’s proprietary model.

While there has been good progress on privacy-preserving DL, the current systems are primarily designed for inference only, and face great challenges for model training. Due to the intractability of privacy-preserving non-polynomial computation (e.g., `softmax`), the current approaches have chosen to approximate the non-polynomial functions to enable computation over the crypto domain. However, such approximation comes with a price that leads to the drop of system accuracy [1, 2, 146]. Moreover, applying it to model training (back propagation) results in unwarranted stability [3–6]. Although one can infinitely approximate a function to alleviate those problems, e.g., using piece-wise linear functions, the resultant large-size approximation function hinders the system efficiency and usability. Furthermore, the current systems use a large number of Homomorphic *permutation operation* (Perm) to achieve inference over the crypto domain, since it is needed to compute the weighted sum and convolution, two critical operations in DL. As discussed in 2.3.1, the Perm operation is time-consuming, which hinders the system efficiency of current systems.

In this chapter, we propose a novel framework, called *secure model training* (SecureTrain), to address the two fundamental challenges faced by privacy-preserving DL model training: (1) model accuracy loss and training instability due to use of function approximation, and (2) computation efficiency. The overarching goal is to eliminate the use of function approximation to *carry out training without accuracy loss and instability*, and reduce the use of Perm operation to improve *computation efficiency*. First of all, in order to achieve approximation-free computation, SecureTrain features an innovative design that enables joint linear and non-linear computation based on the Homomorphic Secret Sharing (HSS) [147–149]. Second, it eliminates the time consuming Perm operations by carefully designing the share set. Moreover, SecureTrain exploits the data flow in both forward and back propagation to enable an efficient piggybacking, thus further accelerating the overall

Table 16. Key notations.

Symbol	Definition
\mathcal{C}/\mathcal{S}	Client/Server
\mathbf{x}	Input data from client
\mathbf{w}/\mathbf{b}	Initial model weight and bias
$\mathbf{w}^{\mathcal{C}}/\mathbf{w}^{\mathcal{S}}$	Weight share at client/server
$\mathbf{w}_1^{\mathcal{C}}/\mathbf{w}_2^{\mathcal{C}}$	Weight shares of $\mathbf{w}^{\mathcal{C}}$
$\mathbf{b}^{\mathcal{C}}/\mathbf{b}^{\mathcal{S}}$	Bias share at client/server
$\widehat{\mathbf{w}}/\widehat{\mathbf{b}}$	Updated model weight/bias
$[\]_{\mathcal{C}}/[\]_{\mathcal{S}}$	Ciphertext encrypted by client/server
\mathbf{z}	Linear output of neural network
r, r_1, r_2, h_1, h_2	Random numbers
δ	back propagation error

computation and reducing communication cost.

We analyze the computation and communication complexity of SecureTrain and prove its security using the standard simulation approach [62, 150]. The proposed SecureTrain is benchmarked with well-known datasets for both inference and training. For inference, our results show that SecureTrain not only ensures privacy-preserving inference but also achieves an inference speedup of $48\times$, $10\times$, $7\times$ and $1.3\times$, respectively, compared with state-of-the-art privacy-preserving inference systems: SecureML [3], MiniONN [65], EzPC [132], and Xonn [76]. For training, SecureTrain achieves the training accuracy and stability comparable to plaintext learning.

4.2 PRELIMINARIES

The system model of SecureTrain includes both MLaaS and training as shown in Figure 1. We now elaborate the neural network training, and the cryptographic background for HSS. Table 16 summaries the key notations that we use in the rest of the chapter. Similar to GALA in Chapter 3, we adopt PHE to enable computation over ciphertext at the server, and the threat model is semi-honest.

4.2.1 NEURAL NETWORK TRAINING

A neural network consists of multiple computation layers that represent a complex relation between the high-dimensional input and the output. Training a neural network is to fit model parameters to a training dataset. A typical training process consists of both *forward propagation* and *back propagation*. Consider a multiclass classification problem to classify m -dimension input $\mathbf{x} = (x_1, x_2, \dots, x_m)$ into a number of l classes, i.e., $\mathbf{y}_n = (y_1, y_2, \dots, y_l)$. Assume that there are totally n layers except for the input layer. \mathbf{w}_i and \mathbf{b}_i are the weight and bias matrices corresponding to the i -th ($1 \leq i \leq n$) layer. Generally, the input \mathbf{x} is denoted as the 0-th layer.

Forward Propagation: The forward propagation calculates weighted-sums¹, i.e., dot product, layer by layer. The output of the i -th layer is activation $\mathbf{a}_i = f(\mathbf{z}_i)$, where \mathbf{z}_i is the weighted-sum $\mathbf{a}_{i-1}\mathbf{w}_i + \mathbf{b}_i$; $f(\cdot)$ is the activation function, i.e., ReLU. The last layer adopts the **softmax** to map a high-dimensional vector into a list of prediction probabilities, $\mathbf{y}_n = e^{\mathbf{z}_n} / \sum_{j=1}^l e^{z_{nj}}$, where $\mathbf{z}_n = \mathbf{a}_{n-1}\mathbf{w}_n + \mathbf{b}_n$.

Back propagation: Once the forward propagation derives the prediction probability \mathbf{y}_n , the distance between the prediction and the true label $\mathbf{t} = (t_1, t_2, \dots, t_l)$ is calculated as the cost. SecureTrain adopts the widely used cross entropy cost function, $C = -\sum_{j=1}^l (t_j \ln y_j + (1 - t_j) \ln(1 - y_j))$ [15, 16, 26]. The weight and bias are then updated based on the backward error propagation as $\hat{\mathbf{w}}_i = \mathbf{w}_i - \eta \Delta \mathbf{w}_i$ and $\hat{\mathbf{b}}_i = \mathbf{b}_i - \eta \Delta \mathbf{b}_i$ where η is the learning rate. The gradients $\Delta \mathbf{w}_i, \Delta \mathbf{b}_i$ are calculated as follows,

$$\Delta \mathbf{w}_i = \mathbf{a}_{i-1} \boldsymbol{\delta}_i, \Delta \mathbf{b}_i = \boldsymbol{\delta}_i, \quad (1)$$

where $\boldsymbol{\delta}_n = \mathbf{y}_n - \mathbf{t}$ that is based on the cross-entropy cost, $\boldsymbol{\delta}_i = \boldsymbol{\delta}_{i+1} \mathbf{w}_{i+1} \odot \frac{\partial \mathbf{a}_i}{\partial \mathbf{z}_i}$, and $\mathbf{a}_0 = \mathbf{x}$. As for the ReLU, it is straightforward to compute $\frac{\partial \mathbf{a}_i}{\partial \mathbf{z}_{ij}} = 1$ if $\mathbf{z}_{ij} \geq 0$, and $\frac{\partial \mathbf{a}_i}{\partial \mathbf{z}_{ij}} = 0$ otherwise.

Since the learning rate η is a constant pre-determined by the client and server, we simplify

¹The convolution operation in Convolutional Neural Network (CNN) can also be transformed into weighted-sum operation [151].

the notations for updating the weight and bias as follows, assuming η has been multiplied into δ_i ,

$$\hat{\mathbf{w}}_i = \mathbf{w}_i - \mathbf{a}_{i-1}\delta_i, \quad \hat{\mathbf{b}}_i = \mathbf{b}_i - \delta_i. \quad (2)$$

4.2.2 HOMOMORPHIC SECRET SHARING

In the secret sharing protocol, a value is shared between two parties, such that combining the two secrets yields the true value. SecureTrain is developed with an efficient secret share mechanism based on the Homomorphic Secret Sharing (HSS) [147–149]. Specifically, a two-party HSS scheme for a class of programs \mathbf{P} consists of algorithms (**Gen**, **Enc**, **Eval**) with the following syntax: 1) **Gen**(1^λ): On input a security parameter 1^λ , the key generation algorithm outputs a public key \mathbf{pk} and a pair of evaluation keys ($\mathbf{ek}_0, \mathbf{ek}_1$). 2) **Enc**(\mathbf{pk}, \mathbf{x}): Given public key \mathbf{pk} and secret input value \mathbf{x} , the encryption algorithm outputs a ciphertext \mathbf{ct} . 3) **Eval**($b, \mathbf{ek}_b, (\mathbf{ct}_1, \dots, \mathbf{ct}_n), P$): On input party index $b \in \{0, 1\}$, evaluation key \mathbf{ek}_b , vector of n ciphertext, a program $P \in \mathbf{P}$ with n inputs, the homomorphic evaluation algorithm outputs y_b , constituting party b 's share of an output $y = P(\mathbf{x})$.

For different functions (i.e., different programs P) with different homomorphic encryption (i.e., the different encryption algorithms **Enc**(\mathbf{pk}, \mathbf{x})), the **Eval** function should be specifically designed, which, in our case is to develop the **Eval** function for the linear and nonlinear computation in neural networks with the packed homomorphic encryption. Here creative designs are required to enable its effective application in practice. This is because in many applications the two parties need to securely obtain \mathbf{x} and perform computation on their respective shares to produce correct results. How to compute \mathbf{x} (i.e., construct a set of data $\mathbf{ct}_1, \dots, \mathbf{ct}_n$) and reconstruct the results (i.e., get the y_b) with the **Eval** function at each party is non-trivial, particularly for encrypted nonlinear computation in neural networks.

4.3 SYSTEM DESCRIPTION

A key design challenge to enable secure and privacy-preserving training is to develop a secure training framework that is *accurate* and *efficient*. The accuracy is imperative to ensure the success of training, while the computation efficiency is critical for practical applicability. In this chapter, we propose a novel secure training framework, *SecureTrain*, that features the following design principles. First, SecureTrain is developed based on the Homomorphic Secret Sharing (HSS) approach [147–149] that enables secure and approximation-free computation for linear and non-linear functions, in order to achieve stable neural network training without accuracy loss. Second, SecureTrain is carefully designed according to the neural network architecture, by piggybacking part of the computation of the back propagation into the forward propagation, and by combining linear and non-linear computation in both the forward and back propagation to accelerate the overall computation and minimize the total communication cost.

4.3.1 SYSTEM OVERVIEW

The proposed SecureTrain framework supports multiple clients to work with a server to collaboratively train a neural network. The server sequentially interacts with each client to complete the training process. In the following discussion, we will focus on one client and one server only. Once the training with one client is finished, the client passes its share of the neural network model parameters to the next client. Note that the randomness of the share does not reveal user data or model parameters to the next client.

To start the training process, the weight and bias (i.e., \mathbf{w} and \mathbf{b}) of each layer are initialized randomly. Without loss of generality, we consider the operation of one layer in the neural network in the discussions since the operations of different layers are similar. Moreover, for the ease of description, we omit the subscript or superscript to denote the network layer, and simply refer to \mathbf{w} and \mathbf{b} unless specified otherwise.

A novel secret share scheme is carefully crafted to protect both the user data and the neural network model parameters. More specifically, the client \mathcal{C} and server \mathcal{S} respectively keep their weight and bias shares $\mathbf{w}^{\mathcal{C}}$, $\mathbf{w}^{\mathcal{S}}$, $\mathbf{b}^{\mathcal{C}}$, and $\mathbf{b}^{\mathcal{S}}$, subject to $\mathbf{w} = \mathbf{w}^{\mathcal{C}} + \mathbf{w}^{\mathcal{S}}$ and $\mathbf{b} = \mathbf{b}^{\mathcal{C}} + \mathbf{b}^{\mathcal{S}}$.

Initially, the client \mathcal{C} has the input data and the random share of weights and bias, while the server \mathcal{S} has the other share of weights and bias. During training, the client and server update their shares of the weights and bias, respectively. Each training round consists of three stages, namely forward propagation, `softmax` calculation, and back propagation.

4.3.2 FORWARD PROPAGATION

During the forward propagation, the input data is fed in the forward direction through the network layers, as introduced in 4.2.1. Each layer takes a vector of data, \mathbf{x} , as input to compute a linear transformation (i.e., the weighted sum, $\mathbf{z} = \mathbf{x}\mathbf{w} + \mathbf{b}$) followed by the nonlinear activations (i.e., $\mathbf{a} = f(\mathbf{z})$). The output (i.e., the activations \mathbf{a}) is then fed to the next layer, serving as the input to continue the forward propagation. The challenge is how to perform such computation in a secure and privacy-preserving manner based on the shares owned by \mathcal{C} and \mathcal{S} .

The overall design principle of SecureTrain is based on the Homomorphic Secret Sharing (HSS). In this research, we apply HSS to develop an efficient approach to enable secure linear and non-linear computation. In particular, the key contribution is to devise innovative evaluation algorithms, i.e., `Eval`, based on the HSS masked paring scheme to ensure $P(\chi)$, i.e., the linear and non-linear functions in each layer, can be efficiently reconstructed from $\text{Eval}(\chi_1, P)$ and $\text{Eval}(\chi_2, P)$. The computation in all layers is essentially similar, but the treatment for the first layer and the rest of layers is slightly different. In the following discussion, we introduce Layer 1 first and then highlight the difference in computing Layer k when $k \geq 2$.

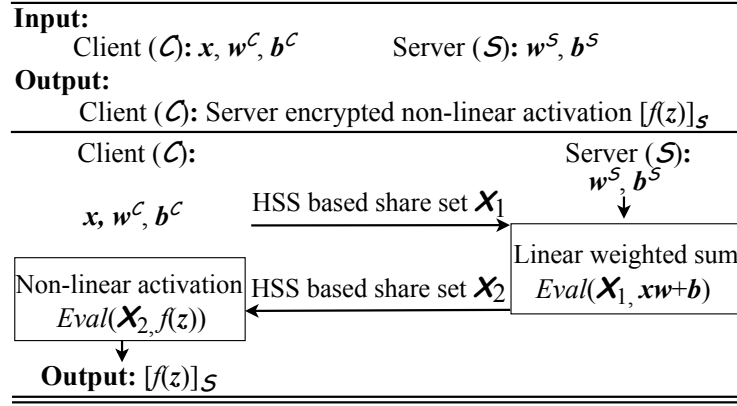


Figure 13. Forward propagation.

Calculation for The First Layer

As illustrated in Fig. 13, the input of the first layer is the client's data \mathbf{x} . In order to protect \mathbf{x} , \mathcal{C} generates two random vectors \mathbf{x}_1 and \mathbf{x}_2 where $\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2$. It also generates two random numbers r_1 and r_2 . \mathcal{C} sends a tuple $(r_1\mathbf{x}_1, r_2\mathbf{x}_2)$ to \mathcal{S} . This design is based on the (t, w) -threshold scheme [61] tailored by data splintering [152]. The detailed security analysis is given in Appendix A.1. Furthermore, \mathcal{C} computes and sends $[r_2]_{\mathcal{C}}$, $[\frac{r_2}{r_1}]_{\mathcal{C}}$ and $[r_2(\mathbf{x}\mathbf{w}^{\mathcal{C}} + \mathbf{b}^{\mathcal{C}})]_{\mathcal{C}}$ to \mathcal{S} . Hereafter, the subscription $[\cdot]_{\mathcal{C}}$ denotes ciphertext encrypted by the client's private key, while $[\cdot]_{\mathcal{S}}$ denotes ciphertext encrypted by the private key of the server. All encryptions, unless specified otherwise, are realized by packed HE (e.g., CKKS [53]).

Here $(r_1\mathbf{x}_1, r_2\mathbf{x}_2)$, $[r_2]_{\mathcal{C}}$, $[\frac{r_2}{r_1}]_{\mathcal{C}}$, and $[r_2(\mathbf{x}\mathbf{w}^{\mathcal{C}} + \mathbf{b}^{\mathcal{C}})]_{\mathcal{C}}$ form the HSS based share set χ_1 , which will be used by \mathcal{S} for calculating the linear weighted sum $\text{Eval}(\chi_1, \mathbf{x}\mathbf{w} + \mathbf{b})$ that will be introduced next.

Calculation of linear weighted sum at server. The server \mathcal{S} has its share of the neural network model parameters, i.e., $\mathbf{w}^{\mathcal{S}}$ and $\mathbf{b}^{\mathcal{S}}$. Upon receiving $(r_1\mathbf{x}_1, r_2\mathbf{x}_2)$, $[r_2(\mathbf{x}\mathbf{w}^{\mathcal{C}} + \mathbf{b}^{\mathcal{C}})]_{\mathcal{C}}$,

$[r_2]_{\mathcal{C}}$ and $[\frac{r_2}{r_1}]_{\mathcal{C}}$ from \mathcal{C} , \mathcal{S} computes the following:

$$\begin{cases} r_1 \mathbf{x}_1 \mathbf{w}^{\mathcal{S}} \odot [\frac{r_2}{r_1}]_{\mathcal{C}} = [r_2 \mathbf{x}_1 \mathbf{w}^{\mathcal{S}}]_{\mathcal{C}}, \\ \mathbf{b}^{\mathcal{S}} \odot [r_2]_{\mathcal{C}} = [r_2 \mathbf{b}^{\mathcal{S}}]_{\mathcal{C}}, \end{cases} \quad (3)$$

where ‘ \odot ’ denotes element-wise multiplication, which is based on the packed HE if it involves ciphertext². Then \mathcal{S} computes $[r_2 \mathbf{x}_1 \mathbf{w}^{\mathcal{S}}]_{\mathcal{C}} + r_2 \mathbf{x}_2 \mathbf{w}^{\mathcal{S}} + [r_2 \mathbf{b}^{\mathcal{S}}]_{\mathcal{C}} = [r_2(\mathbf{x} \mathbf{w}^{\mathcal{S}} + \mathbf{b}^{\mathcal{S}})]_{\mathcal{C}}$, and finally obtains the following which is essentially the weighted sum, but scrambled by r_2 and encrypted by \mathcal{C} :

$$[r_2(\mathbf{x} \mathbf{w}^{\mathcal{S}} + \mathbf{b}^{\mathcal{S}})]_{\mathcal{C}} + [r_2(\mathbf{x} \mathbf{w}^{\mathcal{C}} + \mathbf{b}^{\mathcal{C}})]_{\mathcal{C}} = [r_2 \mathbf{z}]_{\mathcal{C}}. \quad (4)$$

Calculation of non-linear ReLU activation at client. The next step is to calculate the activation. Here, we focus on ReLU, which is predominantly used in state-of-the-art deep neural networks due to its superior performance [15]. \mathcal{S} cannot perform the non-linear activation calculation directly by HE. A naive approach is to let \mathcal{S} send $[r_2 \mathbf{z}]_{\mathcal{C}}$ to \mathcal{C} , which then recovers \mathbf{z} and calculates the ReLU function. However, releasing the weighted sum \mathbf{z} to \mathcal{C} can leak the model parameters as shown in [90, 107].

To securely perform the activation calculation, \mathcal{S} scrambles each element in $r_2 \mathbf{z}$ by a random vector $\mathbf{v}^{\mathcal{S}}$:

$$[r_2 \mathbf{z}]_{\mathcal{C}} \odot \mathbf{v}^{\mathcal{S}} = [r_2 \mathbf{z} \odot \mathbf{v}^{\mathcal{S}}]_{\mathcal{C}}. \quad (5)$$

Meanwhile, \mathcal{S} generates a vector $\mathbf{u}^{\mathcal{S}}$ satisfying $\mathbf{u}^{\mathcal{S}} \odot \mathbf{v}^{\mathcal{S}} = \{\mathbf{1}\}$, and constructs two vectors \mathbf{g}_1 and \mathbf{g}_2 with the same dimension as \mathbf{z} :

$$\begin{aligned} \mathbf{g}_1 &= [g_{11}, g_{12}, \dots, g_{1j}, \dots], \\ \mathbf{g}_2 &= [g_{21}, g_{22}, \dots, g_{2j}, \dots], \end{aligned}$$

²The addition between two ciphertext (or between one ciphertext and one plaintext) is also in element-wise manner.

where (g_{1j}, g_{2j}) is a pair of *polar indicators*, given below:

$$(g_{1j}, g_{2j}) = \begin{cases} (0, u_j^S), & \text{if } v_j^S > 0, \\ (u_j^S, -u_j^S), & \text{if } v_j^S < 0. \end{cases} \quad (6)$$

\mathcal{S} encrypts \mathbf{g}_1 and \mathbf{g}_2 into $[\mathbf{g}_1]_{\mathcal{S}}$ and $[\mathbf{g}_2]_{\mathcal{S}}$, and sends them along with $[r_2 \mathbf{z} \odot \mathbf{v}^S]_{\mathcal{C}}$ to \mathcal{C} . These three items form the HSS based share set χ_2 , which will be used by \mathcal{C} for calculating the non-linear ReLU activation $\text{Eval}(\chi_2, f(\mathbf{z}))$. Note that, $[\mathbf{g}_1]_{\mathcal{S}}$ and $[\mathbf{g}_2]_{\mathcal{S}}$ can be transmitted offline since \mathbf{g}_1 and \mathbf{g}_2 are pre-generated by \mathcal{S} . Upon receiving the inputs from \mathcal{S} , \mathcal{C} first obtains $\mathbf{y} = \mathbf{z} \odot \mathbf{v}^S$ by decrypting $[r_2 \mathbf{z} \odot \mathbf{v}^S]_{\mathcal{C}}$ and canceling the r_2 term. We now show how the client \mathcal{C} can compute ReLU based on \mathbf{y} , $[\mathbf{g}_1]_{\mathcal{S}}$ and $[\mathbf{g}_2]_{\mathcal{S}}$.

Lemma 1): $[\mathbf{g}_1]_{\mathcal{S}} \odot \mathbf{y} + [\mathbf{g}_2]_{\mathcal{S}} \odot f(\mathbf{y})$ recovers the server-encrypted true ReLU function outcome, i.e., $[f(\mathbf{z})]_{\mathcal{S}}$. ■

Proof. If \mathcal{C} had the true weighted sum outcome, i.e., \mathbf{z} , the corresponding ReLU function would be calculated as follows:

$$f(z_j) = \begin{cases} z_j, & \text{if } z_j \geq 0 \\ 0, & \text{if } z_j < 0, \end{cases} \quad (7)$$

for each element z_j in \mathbf{z} , as introduced in 4.2.1.

However, \mathcal{C} only has $y_j = v_j^S \times z_j$. Since v_j^S is a random number that could be positive or negative, it is infeasible to obtain the correct activation directly. Instead, \mathcal{C} computes

$$[\mathbf{g}_1]_{\mathcal{S}} \odot \mathbf{y} + [\mathbf{g}_2]_{\mathcal{S}} \odot f(\mathbf{y}). \quad (8)$$

Since $y_j = v_j^S \times z_j$, $f(y_j)$ may yield four possible outputs, depending on the signs of v_j^S

and z_j .

$$f(y_j) = \begin{cases} y_j, & \text{if } v_j^S > 0 \ \& \ z_j > 0 \\ y_j, & \text{if } v_j^S < 0 \ \& \ z_j < 0 \\ 0, & \text{if } v_j^S > 0 \ \& \ z_j \leq 0 \\ 0, & \text{if } v_j^S < 0 \ \& \ z_j \geq 0. \end{cases} \quad (9)$$

For example, when $v_j^S > 0$ and $z_j > 0$, we have $g_{1j} = 0$ and $g_{2j} = u_j^S$ according to Eq. (6).

Therefore,

$$[g_{1j}]_S \odot y_j = [0]_S, [g_{2j}]_S \odot f(y_j) = [u_j^S \times v_j^S \times z_j]_S.$$

Note that we have chosen $v_j^S u_j^S = 1$. Therefore, Eq. (8) should yield $[z_j]_S$. This is clearly the server-encrypted ReLU output, i.e., the correct result of $[f(z_j)]_S$. Similarly, it can be shown that Eq. (8) always produces the server-encrypted ReLU outcome for other cases of v_j^S and z_j in Eq. (9). The lemma is thus proven. \blacksquare

By Lemma 1, \mathcal{C} has successfully obtained $[f(\mathbf{z})]_S$. This ends the computation in the first layer.

Calculation for the k -th Layer ($2 \leq k \leq n$)

In a neural network, the activations will be fed into the next layer as the input to continue the forward propagation. So, we essentially want to let $\mathbf{x} = f(\mathbf{z})$ and repeat the calculations discussed above for all layers.

However, we are facing a new challenge because \mathcal{C} only has the encrypted $[f(\mathbf{z})]_S$, but not the plaintext data as in the first layer. \mathcal{C} could still let $\mathbf{x} = [f(\mathbf{z})]_S$. As discussed in the first layer, it is not an option to provide \mathbf{x} directly to \mathcal{S} , since \mathcal{S} would recover $f(\mathbf{z})$ and accordingly derive the user data [41, 90, 107]. As a result, \mathcal{C} generates two shares, \mathbf{x}_1 and \mathbf{x}_2 where \mathbf{x}_1 is a random vector and $\mathbf{x}_2 = \mathbf{x} - \mathbf{x}_1$, as discussed before. Note that, \mathbf{x}_2 is essentially encrypted by \mathcal{S} since \mathbf{x} is in the PHE domain. This leads to a fundamental challenge in calculating $r_2(\mathbf{x}\mathbf{w}^C + \mathbf{b}^C)$, because it would require a vector multiplication namely the dot

product, which is computationally expensive in the PHE domain as discussed in 3.1. This renders it impractical to implement the envisioned secure training framework for modern neural networks.

We take a different approach by again adopting the (t, w) -threshold splintering strategy. More specifically, \mathcal{C} constructs the tuple $(h_1\mathbf{w}_1^c, h_2\mathbf{w}_2^c)$, where $\mathbf{w}_1^c + \mathbf{w}_2^c = \mathbf{w}^c$, and h_1 and h_2 are two random numbers. It sends the tuple along with $[\frac{1}{h_1}]_c$ and $[\frac{1}{h_2}]_c$ to \mathcal{S} . Similar to the discussion for the first layer, \mathcal{C} also sends $(r_1\mathbf{x}_1, r_2\mathbf{x}_2)$, $[r_2(\mathbf{x}_1\mathbf{w}^c + \mathbf{b}^c)]_c$, $[r_2]_c$ and $[\frac{r_2}{r_1}]_c$ to \mathcal{S} . The above two tuples and five ciphertext form the HSS based share set χ_1 , which will be used by \mathcal{S} for calculating the linear weighted sum $\text{Eval}(\chi_1, \mathbf{x}\mathbf{w} + \mathbf{b})$.

Upon receiving the inputs from \mathcal{C} , \mathcal{S} performs the following calculation using the received two tuples and five ciphertext:

- Similar to the discussion in the first layer, \mathcal{S} computes $r_1\mathbf{x}_1\mathbf{w}^s \odot [\frac{r_2}{r_1}]_c = [r_2\mathbf{x}_1\mathbf{w}^s]_c$.
- Similar to the first layer, \mathcal{S} computes $r_2\mathbf{x}_2\mathbf{w}^s$.
- Similar to the first layer, \mathcal{S} computes $\mathbf{b}^s \odot [r_2]_c = [r_2\mathbf{b}^s]_c$.
- \mathcal{S} computes $r_2\mathbf{x}_2h_1\mathbf{w}_1^c \odot [\frac{1}{h_1}]_c = [r_2\mathbf{x}_2\mathbf{w}_1^c]_c$.
- \mathcal{S} computes $r_2\mathbf{x}_2h_2\mathbf{w}_2^c \odot [\frac{1}{h_2}]_c = [r_2\mathbf{x}_2\mathbf{w}_2^c]_c$.

Summing up the above five terms along with $[r_2(\mathbf{x}_1\mathbf{w}^c + \mathbf{b}^c)]_c$ received from \mathcal{C} , \mathcal{S} obtains the following:

$$\begin{aligned}
& [r_2(\mathbf{x}_1\mathbf{w}^c + \mathbf{b}^c)]_c + [r_2\mathbf{x}_1\mathbf{w}^s]_c + r_2\mathbf{x}_2\mathbf{w}^s \\
& + [r_2\mathbf{x}_2\mathbf{w}_1^c]_c + [r_2\mathbf{x}_2\mathbf{w}_2^c]_c + [r_2\mathbf{b}^s]_c \\
& = [r_2(\mathbf{x}\mathbf{w} + \mathbf{b})]_c = [r_2\mathbf{z}]_c.
\end{aligned} \tag{10}$$

Thus, \mathcal{S} has obtained the weighted sum that is scrambled by r_2 and encrypted by \mathcal{C} . This is the same as Eq. (4) introduced in the first layer. The same method can be applied to continue the calculation of the non-linear activation. The process repeats until it reaches the last layer, which is followed by **softmax** to be discussed next.

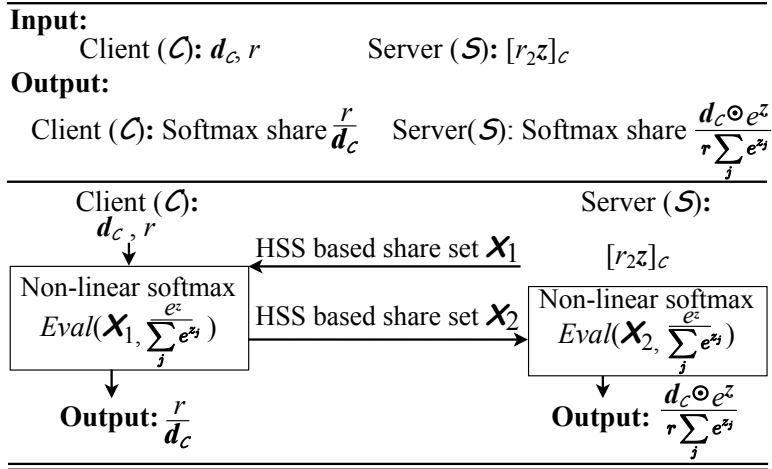


Figure 14. Softmax calculation.

4.3.3 SOFTMAX CALCULATION

After \mathcal{S} obtains the masked weighted sum $[r_2 \mathbf{z}]_{\mathcal{C}}$ for the last layer, it starts to calculate the non-linear **softmax** function for back propagation. As discussed in 4.1, **softmax** is critical to the training process.

It is fundamentally challenging to efficiently calculate **softmax** under the secure training framework because the two mainstream approaches for secure computation (i.e., HE and GC) have limitations to calculate non-linear functions as discussed in 1.2. Fig. 14 illustrates the **softmax** calculation under the secure training framework. The goal is to let \mathcal{C} and \mathcal{S} each obtain a secret share of the true **softmax** value, i.e., $\frac{e^{\mathbf{z}}}{\sum_{j=1}^l e^{z_j}}$.

In order to precisely and securely calculate the **softmax** shares, random vectors are introduced at three occasions to protect the true value of \mathbf{z} . First, \mathcal{S} generates a random vector $\mathbf{d}_{\mathcal{S}}$ with the same dimension as \mathbf{z} , and constructs $[e^{-\mathbf{d}_{\mathcal{S}}}]_{\mathcal{S}}$ which will be used for noise cancellation later. Recall that \mathcal{S} has obtained $[r_2]_{\mathcal{C}}$ from \mathcal{C} in the forward propagation, so \mathcal{S} can compute

$$[r_2 \mathbf{z}]_{\mathcal{C}} + \mathbf{d}_{\mathcal{S}} \odot [r_2]_{\mathcal{C}} = [r_2(\mathbf{z} + \mathbf{d}_{\mathcal{S}})]_{\mathcal{C}}, \quad (11)$$

where \mathbf{z} is scrambled by \mathbf{d}_S , and thus even \mathcal{C} decrypts the above, it would not know \mathbf{z} . \mathcal{S} sends both $[r_2(\mathbf{z} + \mathbf{d}_S)]_{\mathcal{C}}$ and $[e^{-\mathbf{d}_S}]_S$ to \mathcal{C} . The above two ciphertext form the HSS based share set χ_1 , which will be used by \mathcal{C} for calculating the non-linear **softmax** $\text{Eval}(\chi_1, \frac{e^{\mathbf{z}}}{\sum_{j=1}^l e^{z_j}})$. Upon receiving them, \mathcal{C} decrypts the former and cancels r_2 to obtain $\mathbf{z} + \mathbf{d}_S$, and then computes the following:

$$re^{(\mathbf{z} + \mathbf{d}_S)} \odot [e^{-\mathbf{d}_S}]_S + \mathbf{o} = [re^{\mathbf{z}} + \mathbf{o}]_S, \quad (12)$$

where r is a random number and \mathbf{o} is a random zero-sum vector with $\sum_{j=1}^l o_j = 0$. r and \mathbf{o} are introduced here to protect \mathbf{z} . \mathcal{C} further generates a random vector \mathbf{d}_C and computes:

$$\mathbf{d}_C \odot e^{(\mathbf{z} + \mathbf{d}_S)}, \quad (13)$$

where \mathbf{d}_C is introduced to protect \mathbf{z} . \mathcal{C} sends the results of Eqs. (12) and (13) to \mathcal{S} , which form the HSS based share set χ_2 that will be used by \mathcal{S} for calculating the non-linear **softmax** $\text{Eval}(\chi_2, \frac{e^{\mathbf{z}}}{\sum_{j=1}^l e^{z_j}})$.

\mathcal{S} decrypts $[re^{\mathbf{z}} + \mathbf{o}]_S$ to obtain $re^{\mathbf{z}} + \mathbf{o}$, and subsequently sums up all elements of the vector to compute $r \sum_{j=1}^l e^{z_j}$. At the same time, since \mathcal{S} has \mathbf{d}_S , it obtains $\mathbf{d}_C \odot e^{\mathbf{z}}$ by cancelling $e^{\mathbf{d}_S}$ in Eq. (13). Therefore, \mathcal{S} computes its **softmax** share as follows:

$$\frac{\mathbf{d}_C \odot e^{\mathbf{z}}}{r \sum_{j=1}^l e^{z_j}}. \quad (14)$$

Meanwhile, \mathcal{C} constructs $\frac{r}{\mathbf{d}_C}$ as its share of **softmax**. Clearly, the true **softmax** value can be recovered by multiplying the two shares:

$$\frac{\mathbf{d}_C \odot e^{\mathbf{z}}}{r \sum_{j=1}^l e^{z_j}} \odot \frac{r}{\mathbf{d}_C} = \frac{e^{\mathbf{z}}}{\sum_{j=1}^l e^{z_j}}.$$

The shares at \mathcal{C} and \mathcal{S} serve as the input for back propagation.

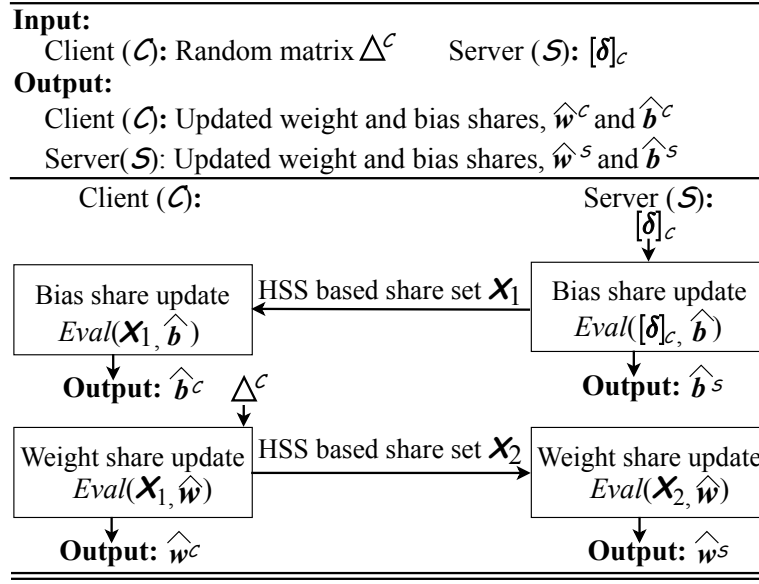


Figure 15. Back propagation diagram.

4.3.4 BACK PROPAGATION

As introduced in 4.2.1, the back propagation begins from the last layer to recursively update the network parameters. The weights and bias in the i -th layer are updated as follows where \mathbf{x} is the activation from the previous layer, and \mathbf{w} , \mathbf{b} , and δ are the weight, bias and error in the current layer.

$$\hat{\mathbf{w}} = \mathbf{w} - \mathbf{x}\delta, \quad \hat{\mathbf{b}} = \mathbf{b} - \delta, \quad (15)$$

Fig. 15 shows the back propagation. According to Eq. (15), the weight \mathbf{w} , bias \mathbf{b} and error δ in the current layer, as well as non-linear activation \mathbf{x} in the previous layer are needed to update the weight and bias for the current layer. As discussed earlier, \mathbf{w} , \mathbf{b} and \mathbf{x} are shared between \mathcal{C} and \mathcal{S} as \mathbf{w}^c , \mathbf{b}^c , $r_1\mathbf{x}_1$ and \mathbf{w}^s , \mathbf{b}^s , $r_2\mathbf{x}_2$, respectively.

Update of Weight/Bias in the Last Layer

In order to update the weights and bias for the last layer, we first introduce how to calculate δ . Recall that, after the **softmax** calculation, \mathcal{C} and \mathcal{S} respectively have the shares $\frac{r}{d_{\mathcal{C}}}$ and $\frac{d_{\mathcal{C}} \odot e^z}{r \sum_{j=1}^l e^{z_j}}$. For back propagation, \mathcal{C} sends $[\frac{r}{d_{\mathcal{C}}}]_{\mathcal{C}}$ and $[t]_{\mathcal{C}}$ to \mathcal{S} (piggybacked to the transmission of the results of Eqs. (12) and (13)), where $[t]_{\mathcal{C}}$ is the \mathcal{C} -encrypted label vector. \mathcal{S} computes the following \mathcal{C} -encrypted ciphertext, i.e., $[\delta]_{\mathcal{C}}$, which essentially shows the difference between the output of **softmax** and the label vector:

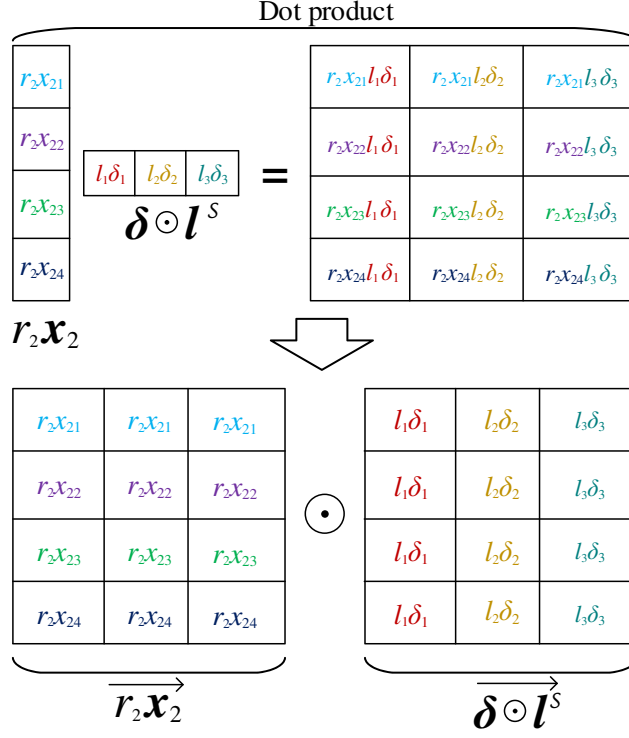
$$\frac{d_{\mathcal{C}} \odot e^z}{r \sum_{j=1}^l e^{z_j}} \odot [\frac{r}{d_{\mathcal{C}}}]_{\mathcal{C}} - [t]_{\mathcal{C}} = [\frac{e^z}{\sum_{j=1}^l e^{z_j}} - t]_{\mathcal{C}} = [\delta]_{\mathcal{C}}. \quad (16)$$

Next, three steps are followed within one communication round to update the weights and bias at \mathcal{C} and \mathcal{S} .

Step 1: Bias Share Update at \mathcal{S} . Once \mathcal{S} obtains $[\delta]_{\mathcal{C}}$ by Eq. (16), it generates its share of δ as a random vector $\delta^{\mathcal{S}}$ and updates its bias share by $\hat{\mathbf{b}}^{\mathcal{S}} = \mathbf{b}^{\mathcal{S}} - \delta^{\mathcal{S}}$. Note that this update involves no communication as $\delta^{\mathcal{S}}$ is self-generated by \mathcal{S} . Meanwhile, four ciphertext are created by \mathcal{S} , which form the HSS based share set χ_1 and will be used to update weights and bias shares at \mathcal{C} . The first ciphertext is the other share of δ for \mathcal{C} : $[\delta]_{\mathcal{C}} - \delta^{\mathcal{S}} = [\delta - \delta^{\mathcal{S}}]_{\mathcal{C}} = [\delta^{\mathcal{C}}]_{\mathcal{C}}$. The second ciphertext is generated by masking δ elementwisely with a random noise vector $\mathbf{l}^{\mathcal{S}}$: $[\delta]_{\mathcal{C}} \odot \mathbf{l}^{\mathcal{S}} = [\delta \odot \mathbf{l}^{\mathcal{S}}]_{\mathcal{C}}$.

The third ciphertext, $[\overrightarrow{r_2 \mathbf{x}_2}]_{\mathcal{S}}$, is transformed and encrypted by \mathcal{S} based on $r_2 \mathbf{x}_2$, which is the share of \mathcal{S} for the activation function in the previous layer. As to be introduced in Step 2, \mathcal{C} will need to compute the arithmetic multiplication (dot product) between $r_2 \mathbf{x}_2$ and $\delta \odot \mathbf{l}^{\mathcal{S}}$. However, the arithmetic multiplication is computationally expensive if directly done in HE as discussed in 3.1. The transformation converts it into element-wise multiplication, which is significantly more efficient for HE computation. As illustrated in the figure below, the transformation essentially expands the original vector $r_2 \mathbf{x}_2$ to a matrix $(\overrightarrow{r_2 \mathbf{x}_2})$ by row filling, i.e., duplicating the element of each row, such that the dot product can be realized

by element-wise multiplication. The fourth ciphertext, $[\vec{l}^S]_S$, is transformed and encrypted by \mathcal{S} according to \mathbf{l}^S . It is generated in a way similar to the third ciphertext, but by column filling, i.e., duplicating the element of each column of \mathbf{l}^S .



Step 2: Weight/Bias Share Update at \mathcal{C} . Upon receiving the four ciphertext generated by \mathcal{S} , \mathcal{C} decrypts $[\delta^c]_C$ and $[\delta \odot \mathbf{l}^S]_C$. Note that since δ is perturbed by \mathbf{l}^S , \mathcal{C} cannot deduce δ . Then, \mathcal{C} updates its bias share as:

$$\hat{b}^c = b^c - \delta^c. \quad (17)$$

\mathcal{C} generates a random matrix Δ^c and updates its weight share as

$$\hat{w}^c = w^c - \Delta^c. \quad (18)$$

We will show later that the shares at \mathcal{C} and \mathcal{S} together result in a correct update of the weights and bias of the neural network.

\mathcal{C} then calculates two terms that will enable \mathcal{S} to update its weight share in Step 3. Specifically, the first term is

$$\mathbf{x}_1(\boldsymbol{\delta} \odot \mathbf{l}^{\mathcal{S}}). \quad (19)$$

The second term is

$$[\overrightarrow{r_2 \mathbf{x}_2}]_{\mathcal{S}} \odot \overrightarrow{\boldsymbol{\delta} \odot \mathbf{l}^{\mathcal{S}}}, \quad (20)$$

where $\overrightarrow{\boldsymbol{\delta} \odot \mathbf{l}^{\mathcal{S}}}$ is transformed from $\boldsymbol{\delta} \odot \mathbf{l}^{\mathcal{S}}$ by column filling as illustrated in the former figure.

As \mathcal{C} has r_2 , it can cancel r_2 to obtain

$$[\overrightarrow{\mathbf{x}_2} \odot \overrightarrow{\boldsymbol{\delta} \odot \mathbf{l}^{\mathcal{S}}}]_{\mathcal{S}}. \quad (21)$$

Adding Eq. (19) and Eq. (21) results in

$$\mathbf{x}_1(\boldsymbol{\delta} \odot \mathbf{l}^{\mathcal{S}}) + [\overrightarrow{\mathbf{x}_2} \odot \overrightarrow{\boldsymbol{\delta} \odot \mathbf{l}^{\mathcal{S}}}]_{\mathcal{S}} = [\mathbf{x}(\boldsymbol{\delta} \odot \mathbf{l}^{\mathcal{S}})]_{\mathcal{S}},$$

where $\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2$ as discussed in 4.3.2.

Finally, \mathcal{C} calculates the corresponding weight share for \mathcal{S} as:

$$[\mathbf{x}(\boldsymbol{\delta} \odot \mathbf{l}^{\mathcal{S}})]_{\mathcal{S}} - \Delta^c \odot [\overrightarrow{\mathbf{l}^{\mathcal{S}}}]_{\mathcal{S}} = [\mathbf{x}(\boldsymbol{\delta} \odot \mathbf{l}^{\mathcal{S}}) - \Delta^c \odot \overrightarrow{\mathbf{l}^{\mathcal{S}}}]_{\mathcal{S}}, \quad (22)$$

which forms the HSS based share set $\boldsymbol{\chi}_2$ and is then sent to \mathcal{S} for weight update in Step 3.

Step 3: Weight Share Update at \mathcal{S} . By decrypting the ciphertext from Eq. (22), \mathcal{S} gets $\mathbf{x}(\boldsymbol{\delta} \odot \mathbf{l}^{\mathcal{S}}) - \Delta^c \odot \overrightarrow{\mathbf{l}^{\mathcal{S}}}$. As $\mathbf{l}^{\mathcal{S}}$ is known by \mathcal{S} , it can be cancelled, yielding $\mathbf{x}\boldsymbol{\delta} - \Delta^c$. \mathcal{S} finally updates its weight share by

$$\widehat{\mathbf{w}}^{\mathcal{S}} = \mathbf{w}^{\mathcal{S}} - (\mathbf{x}\boldsymbol{\delta} - \Delta^c). \quad (23)$$

It is easy to verify that the sum of updated weight and bias at \mathcal{C} and \mathcal{S} are

$$\hat{\mathbf{b}}^{\mathcal{C}} + \hat{\mathbf{b}}^{\mathcal{S}} = \mathbf{b} - \boldsymbol{\delta} \text{ and } \hat{\mathbf{w}}^{\mathcal{C}} + \hat{\mathbf{w}}^{\mathcal{S}} = \mathbf{w} - \mathbf{x}\boldsymbol{\delta},$$

which are exactly the updated weights and bias as shown in Eq. (15). By now the update of weights and bias at \mathcal{C} and \mathcal{S} is completed for the last layer. The communication is within one round.

Update of Weight/Bias in the k -th Layer ($k \leq (n - 1)$)

The back propagation in the k -th layer is very similar to that in the last layer as introduced above. The only difference is the calculation of $[\boldsymbol{\delta}]_{\mathcal{C}}$. In the last layer, $[\boldsymbol{\delta}]_{\mathcal{C}} = [\frac{e^{\mathbf{z}}}{\sum_{j=1}^t e^{z_j}} - \mathbf{t}]_{\mathcal{C}}$ as shown in Eq. (16), which is simply the difference between the output of **softmax** and the label vector. In the k -th layer, $[\boldsymbol{\delta}]_{\mathcal{C}}$ depends on the derivative of the activation function, i.e., $\frac{\partial \mathbf{x}}{\partial \mathbf{z}}$. More specifically, $\boldsymbol{\delta}$ in the k -th layer should be computed as:

$$\boldsymbol{\delta} = \mathring{\boldsymbol{\delta}} \mathring{\mathbf{w}} \odot \frac{\partial \mathbf{x}}{\partial \mathbf{z}} = (\mathring{\boldsymbol{\delta}}^{\mathcal{C}} + \mathring{\boldsymbol{\delta}}^{\mathcal{S}})(\mathring{\mathbf{w}}^{\mathcal{C}} + \mathring{\mathbf{w}}^{\mathcal{S}}) \odot \frac{\partial \mathbf{x}}{\partial \mathbf{z}},$$

where $\mathring{\mathbf{w}}$ and $\mathring{\boldsymbol{\delta}}$ are the weight and error of the $(k + 1)$ -th layer, while $\frac{\partial \mathbf{x}}{\partial \mathbf{z}}$ is the derivative of the current layer's activation function.

The key challenge is to securely compute the derivative. This can be achieved by embedding the computation into the forward propagation. Recall that $[\mathbf{g}_1]_{\mathcal{S}}$ and $[\mathbf{g}_2]_{\mathcal{S}}$ have been introduced in the forward propagation to enable \mathcal{C} to obtain the \mathcal{S} -encrypted ReLU by Eq. (8). To compute the derivative, \mathcal{S} introduces another vector \mathbf{g}_3 and sends $[\mathbf{g}_3]_{\mathcal{S}}$ to \mathcal{C} , where

$$g_{3j} = \begin{cases} 0, & \text{if } v_j^{\mathcal{S}} > 0 \\ 1, & \text{if } v_j^{\mathcal{S}} < 0. \end{cases} \quad (24)$$

Accordingly, while \mathcal{C} calculates the \mathcal{S} -encrypted ReLU by Eq. (8), it also computes the

\mathcal{S} -encrypted ReLU derivative as follows:

$$f'_R(\mathbf{y}) + (1 - 2f'_R(\mathbf{y})) \odot [\mathbf{g}_3]_{\mathcal{S}}, \quad (25)$$

where \mathbf{y} is the masked weighted sum as discussed in 4.3.2 and $f'_R(y_j)$ denotes the derivative of ReLU, which is 1 if $y_j > 0$ or 0 otherwise as introduced in 4.2.1.

Lemma 2): *Eq. (25) yields the \mathcal{S} -encrypted ReLU derivative, i.e., $[\frac{\partial \mathbf{x}}{\partial \mathbf{z}}]_{\mathcal{S}}$. ■*

Proof. If $v_j^{\mathcal{S}} > 0$, then $g_{3j} = 0$ and accordingly Eq. (25) results in $[f'_R(y_j)]_{\mathcal{S}}$. Since $y_j = v_j^{\mathcal{S}} \times z_j$ and $v_j^{\mathcal{S}} > 0$, it is straightforward to show that $f'_R(y_j) = f'_R(z_j)$. Therefore, Eq. (25) yields the \mathcal{S} -encrypted ReLU derivative. On the other hand, if $v_j^{\mathcal{S}} < 0$, we have $g_{3j} = 1$, and thus Eq. (25) results in $[1 - f'_R(y_j)]_{\mathcal{S}}$. It is again easy to show that $f'_R(y_j) = 1 - f'_R(z_j)$ when $v_j^{\mathcal{S}} < 0$. Therefore, Eq. (25) still yields the \mathcal{S} -encrypted ReLU derivative. ■

Till now, \mathcal{C} has obtained the ReLU derivative in a way piggybacked to the forward propagation with marginal computation cost. A secret share approach can then follow to compute the back propagation using a method similar to the last layer as discussed in Sec. 4.3.4. The detailed design is presented in Appendix A.2.

4.3.5 COMPLEXITY ANALYSIS

The computation and communication complexities in a layer of SecureTrain are summarized in Tables 17 and 18, where n_i is the input dimension at a layer; n_o is the output dimension; n_s is the number of slots in a CKKS ciphertext; s_c is the size of a CKKS ciphertext in bit; and s_p is the size of a plaintext value in bit. We assume $n_s \gg n_i, n_o$, which is also adopted in [64].

The detailed analysis can be found in Appendix A.3. Table 17 summarizes the computation complexity of SecureTrain in the forward propagation (i.e., inference), softmax, and

Table 17. Computation complexity.

Methodology	Perm	Mult	Add
Halevi-Shoup [138]	$O(n_i)$	$O(n_i)$	$O(n_i)$
GAZELLE [64]	$O(\log \frac{n_s}{n_o} + \frac{n_i n_o}{n_s})$	$O(\frac{n_i n_o}{n_s})$	$O(\log \frac{n_s}{n_o} + \frac{n_i n_o}{n_s})$
SecureTrain(Inf.)	0	$O(1)$	$O(1)$
SecureTrain(Sof.)	0	$O(1)$	$O(1)$
SecureTrain(Bac.)	0	$O(\frac{n_i n_o}{n_s})$	$O(\frac{n_i n_o}{n_s})$

Table 18. Communication complexity in each part.

Comput. part	Commu. cost in bit	Commu. round
Forward Prop.	$8s_c + 2n_i s_p(1 + n_o)$	1
Softmax	$3s_c + n_o s_p$	1
Backprop.	$4n_o s_p + (11 + \frac{3n_i n_o}{n_s})s_c$	1

back propagation. It also compares with classic methodology in [138] and the state-of-the-art approach in GAZELLE [64]. Note that [138] and [64] focus on inference only. SecureTrain reduces the layer-wise forward calculation to constant complexity by integrating secret-share-based plaintext calculation and the HE-based non-permutation computation. It is worth pointing out that SecureTrain finishes the linear and non-linear calculation for each layer with above complexity while [138] and [64] only compute the linear part. Meanwhile, we give the analytical communication complexity of SecureTrain in Table 18. The quantitative performance comparison is given in 4.5.

4.4 SECURITY ANALYSIS

We prove the security of SecureTrain using the simulation approach [62]. Specifically, the semi-honest adversary \mathcal{A} can compromise any one of the client or server, but not both (i.e., the client and server do not collude). Here, security means that the adversary only learns

the inputs from the party that it has compromised, but nothing else beyond that. It is modeled by two interactions. The first is an interaction in the real world that parties follow the protocol in the presence of a simulator \mathbf{sim} which constructs the messages to the target parties, based on their inputs and randomness; the second is an ideal interaction that \mathbf{sim} forward inputs of target parties to a functionality machine \mathcal{F} that acts as the Trusted Third Party (TTP). To prove security, we demonstrate that no simulator \mathbf{sim} can distinguish the real and ideal interactions. In other words, we want to show that the real-world simulator achieves the same effect in the ideal interaction.

(1) *Security against a semi-honest client.* We define a simulator \mathbf{sim} that simulates an admissible adversary \mathcal{A} which has compromised the client in the real world. As for forward propagation (see Figure 13), \mathbf{sim} conducts the following: 1) receives from client the HSS based share set χ_1 ; 2) sends χ_1 to \mathcal{F} and receives the HSS based share set χ_2 , including three ciphertext (see from Eq. (5)); 3) constructs another HSS based share set $\tilde{\chi}_1$, which has the same data structure as χ_1 ; and 4) sends $\tilde{\chi}_1$ to \mathcal{S} and receives the HSS based share set $\tilde{\chi}_2$. Here, $\tilde{\chi}_2$ is indistinguishable from χ_2 due to the randomness of \mathbf{v}^S in Eq. (5)³ and the security of CKKS. Thus the forward propagation is secure against a semi-honest client.

In **softmax** calculation as shown in Figure 14, \mathbf{sim} conducts as follows: 1) abstracts the randomness of client and forms a random number r and a random vector \mathbf{d}_C ; 2) sends r and \mathbf{d}_C to \mathcal{F} and receives the HSS based share set χ_1 , including two ciphertext (see from Eq. (11)); 3) constructs another random number \tilde{r} and random vector $\tilde{\mathbf{d}}_C$, which have the same structure as r and \mathbf{d}_C ; and 4) receives from \mathcal{S} the HSS based share set $\tilde{\chi}_1$. Here χ_1 is indistinguishable from $\tilde{\chi}_1$ due to the randomness of \mathbf{d}_S in Eq. (11) and the security of CKKS. Thus the **softmax** calculation is secure against a semi-honest client.

In back propagation as shown in Figure 15, \mathbf{sim} conducts as follows: 1) abstracts the randomness of client and forms the random matrix Δ^C ; 2) sends Δ^C to \mathcal{F} and gets HSS

³The approximation calculation in CKKS makes the linear result \mathbf{z} non-zero with high probability, thus the randomness of \mathbf{v}^S makes \mathbf{z} always blind to client.

Table 19. Inference performance comparison.

Framework	Runtime (s)	Communication	Accuracy (%)
SecureML [3]	4.88	-	93.1
Minionn [65]	1.04	15.8MB	97.6
EzPC [132]	0.7	76MB	97.6
Xonn [76]	0.13	4.29MB	97.6
SecureTrain	0.1	1.89MB	97.6

based share set χ_1 ; 3) constructs another random matrix $\tilde{\Delta}^c$; and 4) receives from \mathcal{S} the HSS based share set $\tilde{\chi}_1$. Here $\tilde{\chi}_1$ is indistinguishable from χ_1 due to randomness of δ^S and l^S , and the security of CKKS.

Furthermore, the calculation for $[\delta]_c$ is piggybacked in the weight/bias update for the previous layer to enable the next round of weight/bias update. In such case, **sim** conducts as follows: 1) abstracts the randomness of client and forms a random vector \mathbf{p}^c ; 2) sends \mathbf{p}^c to \mathcal{F} and gets a ciphertext according to Eq. (33) and a plaintext tuple $(r_3\delta_1^S, r_4\delta_2^S)$; 3) constructs another random vector $\tilde{\mathbf{p}}^c$; 4) receives from \mathcal{S} the ciphertext $[\mathbf{g}_3]_S$ and calculates the **ReLU** derivative by Eq. (25); 5) calculates a ciphertext $[\tilde{\mathbf{p}}^S]_S$ by Eq. (32) and sends $[\tilde{\mathbf{p}}^c]_c$ and $[\tilde{\mathbf{p}}^S]_S$ to \mathcal{S} ; and 6) receives from \mathcal{S} the ciphertext of Eq. (33) and the plaintext tuple with the same structure as $(r_3\delta_1^S, r_4\delta_2^S)$. Here the ciphertext in step 2) are indistinguishable from these in step 6) due to the randomness of \mathbf{q}^S . The plaintext tuple in step 2) is also indistinguishable from the one in step 6) due to randomness of δ^S . Thus, the back propagation is secure against a semi-honest client.

(2) *Security against a semi-honest server.* The proof is similar to the security against a semi-honest client, as detailed in Appendix A.4.

4.5 PERFORMANCE EVALUATION

We implement SecureTrain using a C++ backend. The source code is published at

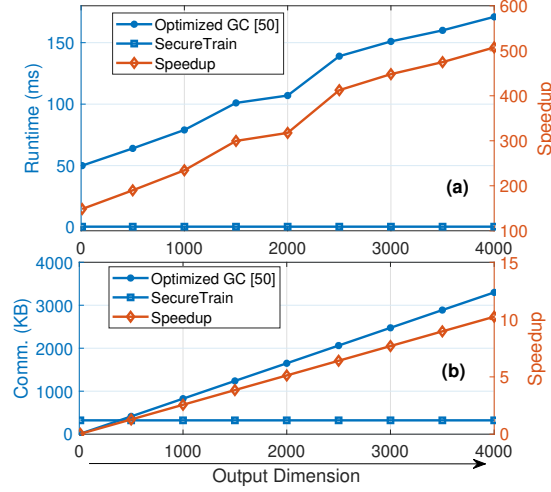


Figure 16. Performance comparison for ReLU calculation: (a) Runtime and (b) Communication cost with different output dimensions.

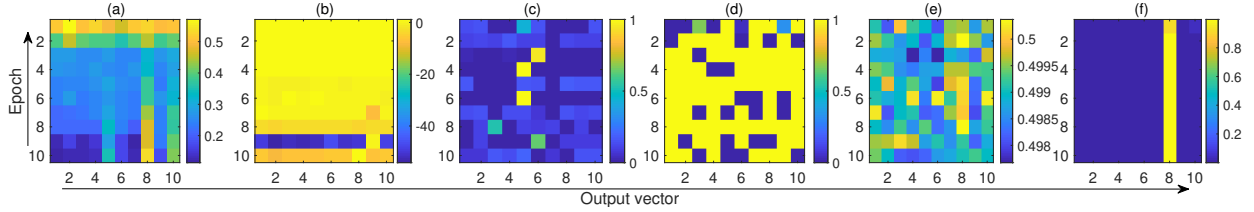


Figure 17. Comparison of the last layer output of different approximations over 10 epochs: (a) Piecewise linear approximation [1]; (b) Maclaurin approximation [2]; (c) ReLU based softmax approximation [3, 4]; (d) ReLU based sigmoid approximation [5]; (e) Polynomial based sigmoid approximation [6]; (f) Non-approximation in SecureTrain.

GitHub⁴. Both the client and server run Ubuntu and have an Intel i7-8700 3.2GHz CPU with 12 threads and 16 GB RAM. The network link between them is a Gigabit Ethernet. This experiment setting is similar to the ones adopted in existing works such as [76]. The Microsoft SEAL package is used for HE computation [136]. The CKKS scheme is adopted in SecureTrain. Note that CKKS directly supports floating point encryption/decryption/operations. That is, it does not need to encode floating point numbers in NN computation into integers for encryption/decryption as many other encryption schemes. The five parameters of

⁴<https://github.com/ChiaoThon/SecureTrain>

CKKS, i.e., the polynomial modulus degree, coefficient modulus size, noise standard deviation, number of slots in the ciphertext, and the precision of floating point in bits, are set as 8192, 200, 3.2, 4096, and 40, respectively. Such parameter selection can guarantee the correct decryption of the 0-multiplicative-depth ciphertexts in SecureTrain (e.g., the randomized ciphertexts from server in Eq. (5)) on the crypto domain, as demonstrated in the SEAL library [136]. We test SecureTrain on the widely used MNIST dataset [140] with 60K training images and 10K testing images. We adopt a classic neural network model that has been widely used in previous works including SecureML [3], GELU-Net [90], CryptoDL [6], SecureNN [4] and ABY [5]. It includes four layers including two hidden layers. The input dimension is 784 which corresponds to the total number of pixels in a MNIST image. The dimension for two hidden layers is 128. The output dimension is 10 which corresponds to 10 digit classes of the MNIST dataset.

4.5.1 PERFORMANCE IN INFERENCE

SecureTrain is able to conduct both inference and training. We first look at the inference performance compared with four state-of-the-art privacy-preserving inference frameworks. Table 19 illustrates their inference performance including the runtime, communication cost, and the inference accuracy. The runtime is the duration from the moment when the client sends an image to the server, to the moment when the client receives the inference result from the server. SecureTrain achieves an inference speedup of $48\times$, $10\times$, $7\times$, and $1.3\times$, respectively, compared with SecureML [3], MiniONN [65], EzPC [132], and Xonn [76]. SecureTrain achieves the same inference accuracy, 97.6%, as Minionn, EzPC and Xonn. With regard to the communication cost, SecureTrain outperforms other schemes by $2\times$ to $40\times$. This is because all those schemes adopted GC for non-linear activation calculations, while SecureTrain uses a highly efficient approach based on HSS.

Next, as shown in Figure 16, we illustrate the performance gain of SecureTrain in non-linear computation by comparing the performance of ReLU calculation using GC and the

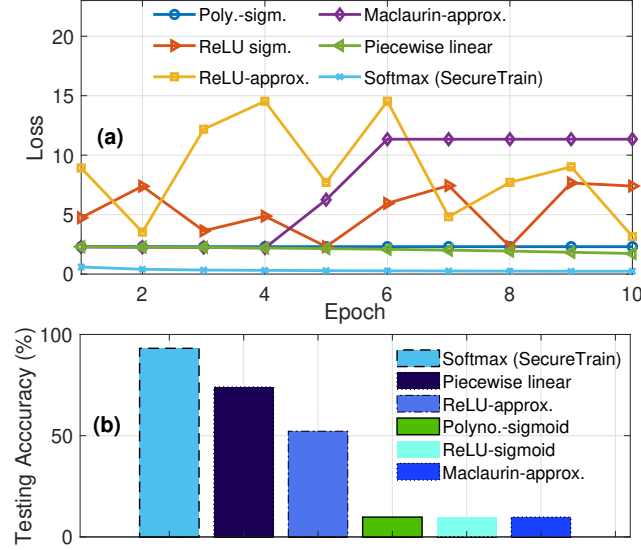


Figure 18. Training loss and testing accuracy: (a) Loss during training and (b) Testing accuracy with different approximation approaches. The **softmax** is non-approximation in SecureTrain while [3, 4] are with ReLU based **softmax** approximation, [2] is with Maclaurin approximation, [1] is with Piecewise linear approximation, [6] is with Polynomial based **sigmoid** approximation, and [5] is with ReLU based **sigmoid** approximation.

scheme used by SecureTrain. Specifically, the SecureTrain scheme saves time about two orders of magnitude compared with GC. In these cases, the runtime of GC ranges from 50 to 171 milliseconds while the SecureTrain scheme can complete it for around 0.4 millisecond. The communication cost reduction reaches up to one order of magnitude, thanks to both significantly reduced computation complexity and the scalability of data processing of the packed HE technique.

4.5.2 PERFORMANCE IN TRAINING

In neural network training, a critical step is the computation of **softmax**, which is needed by the backpropagation. It is more difficult than other nonlinear functions, due to the specific form of the function which involves the exponential normalization of the input. The existing schemes mainly use approximation, e.g., the piecewise linear **sigmoid**⁵ approximation [1],

⁵This function with the form $f(x) = \frac{1}{1+e^{-x}}$ is adopted to approximate the **softmax** for the last layer.

Maclaurin `sigmoid` approximation [2], polynomial `sigmoid` approximation by CryptoDL [6], ReLU based `sigmoid` approximation by ABY [5], and ReLU based approximation by SecureML [3] and SecureNN [4]. Note that the first four schemes did not directly approximate the `softmax` function, but used an approximated `sigmoid` to substitute the `softmax` function.

Figs. 17(a)–(f) illustrate the training output over 10 epochs by the above five approximation approaches compared with SecureTrain that implements the original `softmax` function. Each row in a subfigure is the output vector corresponding to the 10 digit classes at a given epoch by the corresponding approach, while each column is the output value for a given class over 10 epochs. The training image is a digit ‘7’. Fig. 17(f) indicates that the training using SecureTrain that implements the original `softmax` function efficiently learns the input feature even at the early epochs, with a dominant value in the 8-th column (which corresponds to the class ‘7’) and much smaller values in other columns. In contrast, all five approximation approaches (Fig. 17(a)–(e)) have poor performance. Among them, the piecewise linear approximation (Fig. 17(a)) performs better and converges, while other approximation approaches cannot learn the input feature ‘7’ well and do not converge.

Next we examine the overall training loss and testing accuracy, as illustrated in Fig. 18. Among the approximation approaches, the piecewise linear approximation has a converged training loss. However, there is still a significant performance gap compared with SecureTrain, which converges significantly faster and achieves a 93.17% testing accuracy after 10 epochs. Other approximation approaches cannot even converge and have a poor testing accuracy. Fig. 18(a) illustrates that these approximation approaches have an unstable loss. ReLU based `sigmoid` approximation and Maclaurin approximation have about 10% accuracy, which is equivalent to a random guess, and indicates the trained models actually have not learned the features effectively. The polynomial based `sigmoid` approximation has an almost flat loss curve and a poor 10% accuracy, which indicates the model does not become better with the training. This happens when the input has a relatively wide range, which results in the polynomial approximation significantly deviating from the original `softmax` function.

To examine the training stability of each approximation approach, we train the network 500 times using each approach, and record the loss and testing accuracy for each experiment. Fig. 19 plots the *probability density distribution* (PDF) of the loss from the 500 experiments. While SecureTrain keeps the training loss around 0.23, the approximation based approaches have large loss. This indicates the poor training stability of those approaches. The polynomial based `sigmoid` approximation has a loss around 2.3. While it is relatively small, the problem of this approach is that the loss does not reduce or converge throughout the training process. Fig. 20 plots the PDF of the testing accuracy over the 500 trainings for each approach. The testing accuracies of SecureTrain are all very close, centering around 93%. The piecewise linear approximation has an accuracy of around 73% to 77%. In contrast, other approximations have highly diverse accuracies among different experiments. This again indicates the poor training stability. The polynomial based `sigmoid` has a consistently poor accuracy around 10%, as the training process does not really converge. In summary, the training by SecureTrain is consistently stable, and the accuracy is much better than the approximation based approaches.

Next, we explore the performance under different network structures. We change the number of hidden neurons in each hidden layer from as small as 8 to 1024, where 1024 is widely used in modern neural network structures. All network structures are trained for 10 epochs. Fig. 21 illustrates the output distribution of SecureTrain and the five approximation approaches, as a function of the number of neurons in the hidden layer. Each row in a subfigure is the 10-dimension output vector for a network with the given number of hidden neurons. As can be seen, SecureTrain effectively learns the data feature under different network structures, which has a large output value for class ‘7’ (corresponding to the 8-th column in the subfigure). Among the approximation schemes, the piecewise linear approximation performs relatively better than others. Nevertheless, it still has a significant performance gap compared with SecureTrain. It needs 128 or more hidden neurons to effectively recognize the input image. The remaining approximation approaches show unstable

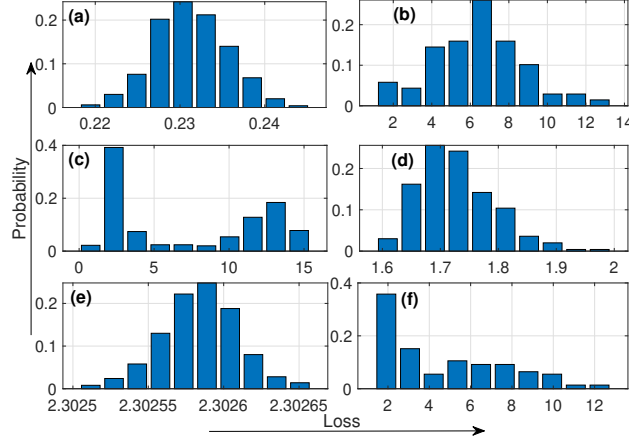


Figure 19. Probability density distribution of training loss with different approximations: (a) Non approximation in SecureTrain; (b) ReLU based `softmax` approximation [3, 4]; (c) Maclaurin approximation [2]; (d) Piecewise linear approximation [1]; (e) Polynomial based `sigmoid` approximation [6]; (f) ReLU based `sigmoid` approximation [5].

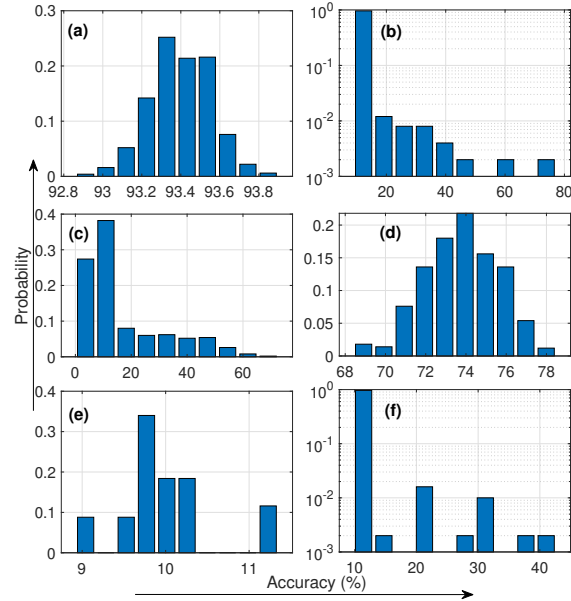


Figure 20. Probability density distribution of accuracy with different approximations: (a) Non-approximation in SecureTrain; (b) ReLU based `softmax` approx. [3, 4]; (c) Maclaurin approx. [2]; (d) Piecewise linear approx. [1]; (e) Polynomial based `sigmoid` approx. [6]; (f) ReLU based `sigmoid` approx. [5].

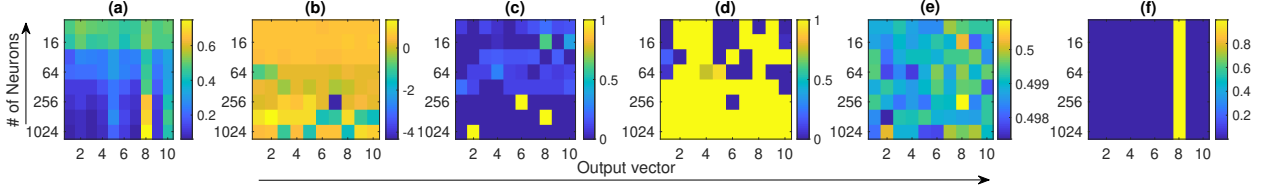


Figure 21. Output probability distribution of each approach with different network structures in terms of the number of hidden neurons: (a) Piecewise linear approximation [1]; (b) Maclaurin approximation [2]; (c) ReLU based **softmax** approximation [3, 4]; (d) ReLU based **sigmoid** approximation [5]; (e) Polynomial based **sigmoid** approximation [6]; (f) Non-approximation in SecureTrain.

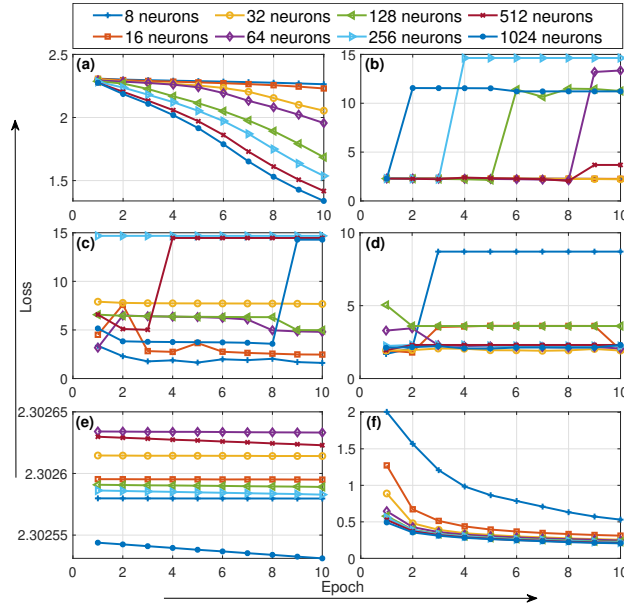


Figure 22. Training loss of different network structures under different approximations: (a) Piecewise linear approximation [1]; (b) Maclaurin approximation [2]; (c) ReLU based **softmax** approximation [3, 4]; (d) ReLU based **sigmoid** approximation [5]; (e) Polynomial based **sigmoid** approximation [6]; (f) Non-approximation in SecureTrain.

output distributions under different network structures. This illustrates the instability of these approximations as they work for certain network structures, but do not achieve the consistent stability for general larger networks.

Fig. 22 plots the training loss over 10 epochs under different network structures. We

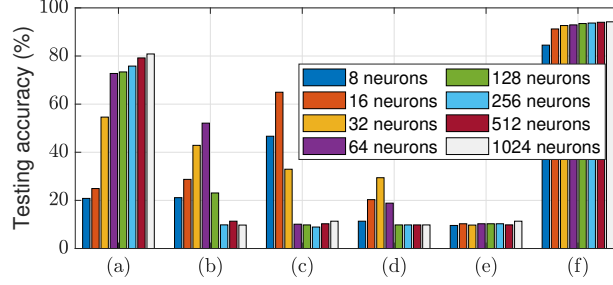


Figure 23. Testing accuracy of different network structures under different approximations: (a) Piecewise linear approximation [1]; (b) Maclaurin approximation [2]; (c) ReLU based `softmax` approximation [3, 4]; (d) ReLU based `sigmoid` approximation [5]; (e) Polynomial based `sigmoid` approximation [6]; (f) Non-approximation in SecureTrain.

have the following observations: 1) the piecewise linear approximation approach performs better with a larger network (more hidden neurons); 2) the loss of the Maclaurin based approximation increases under all network settings; 3) the ReLU based approximation approach has loss decreasing for the network with 128 hidden neurons, while the loss bumps up and down or stays flat for all other networks; 4) the ReLU based `sigmoid` approximation approach performs similarly as the ReLU based approximation approach; 5) the loss of the polynomial based `sigmoid` approximation stays flat or decreases slightly (by about 10^{-5} for 10 epochs), which is technically not trainable. In contrast, SecureTrain converges for all network structures at a fast pace, thanks to its novel implementation of the original `softmax`.

The testing accuracies under different network structures for each approach are illustrated in Fig. 23. SecureTrain significantly outperforms all other approaches. Besides the low accuracy, another serious issue for the approximation based approaches (except the piecewise linear approximation) is that their accuracy decreases under a larger, more sophisticated

Table 20. Comparison of training time.

Framework	Accuracy (%)	Time per batch (s)	TTP
SecureNN [4]	73.8	263.6	✓
SecureTrain	92.9	25.6	✗

network structures. The piecewise linear approximation approach is better than other approximation based approaches in that its accuracy increases under a larger network and reaches around 80% accuracy for the network with 1024 hidden neurons. However, it is still significantly lower than the 94% accuracy of SecureTrain.

Furthermore, Table 20 shows the time cost of the state-of-the-art scheme and SecureTrain in training phase with batch size 128. We can see that SecureTrain keeps over 19% higher accuracy and a $10\times$ training speedup. Meanwhile, SecureTrain does not need a Trust Third Party (TTP), which is another sharp contrast in terms of practical usability as the TTP is not preferred in practice.

In summary, the `softmax` function in the last layer is critical for the backpropagation in training. Most existing approaches for privacy preserved neural network training must reply on approximation. However, most of them result in unstable training, which finally leads to an unusable model. The proposed SecureTrain uses a creative design to enable the secure implementation of the original `softmax` function as well as the updating process in a cost-efficient manner. Therefore, it maintains all the good features as the plaintext version of training, including the same model accuracy, the convergence of training, and better accuracy with a larger, more sophisticated model structure. SecureTrain also significantly outperforms all existing approaches in training speed, testing accuracy, and convergence speed.

4.6 CHAPTER SUMMARY

In this chapter, we have explored the optimization for a whole layer by replacing the GC-based nonlinear calculation with a newly-designed joint linear and non-linear computation based on the Homomorphic Secret Sharing. Furthermore, the forward computation has been expanded to enable backward propagation with a piggyback design that carefully devises the share set and integrates the dataflow of the whole training process. As such, we have proposed SecureTrain that achieves an inference speedup as high as $48\times$ compared with state-of-the-art inference frameworks.

CHAPTER 5

GLOBALLY ENCRYPTED AND LOCALLY UNENCRYPTED **PPDL**

In previous chapter, we explored the optimization for each layer by replacing the GC-based nonlinear calculation with a newly-designed joint linear and non-linear computation based on the Homomorphic Secret Sharing. While the nonlinear computation is over ciphertext in Chapter 4, the nonlinear calculation is completed for free in this chapter by a carefully partitioned DL framework, GELU-Net, where the server performs linear computation on encrypted data utilizing a less complex homomorphic cryptosystem, while the client securely executes non-polynomial computation in plaintext without approximation. GELU-Net demonstrates $14\times$ to $35\times$ inference speedup compared to the classic systems. The rest of this chapter is organized as follows. Section 5.1 details the motivation of GELU-Net. Section 5.2 introduces the primitives that GELU-Net adopts. Section 5.3 describes the design details of GELU-Net. Section 5.4 presents the security analysis of GELU-Net. The experimental results are illustrated and discussed in Section 5.5. Finally, Section 5.6 concludes this chapter.

5.1 MOTIVATION

Privacy is a fundamental challenge for many smart applications that depend on data aggregation and collaborative learning across different entities. Existing endeavors take different directions to address the privacy issue. Two major directions are differential privacy [123] and fully homomorphic encryption [78]. Differential privacy injects noise into query results to avoid inferring information about any specific record. However, it needs careful calibration to balance privacy and model usability. Further, private attributes still remain in

plaintext so users may still have security concerns. A more promising solution comes from the recent advance in fully homomorphic encryption (FHE) [153]. It allows users to encrypt data with the public key and offload computation to the cloud. The cloud computes over the encrypted data and generates encrypted results. Without the secret key, the cloud simply serves as a computation platform but cannot access any user information. This powerful technique has been integrated with deep learning in the pioneering work of [78], known as CryptoNets, which built a convolutional neural network on FHE to process inference queries. However, it faces three fundamental problems: P1) FHE is extremely costly in computation, thus unsuitable for large-scale neural networks; P2) the activation functions are not cryptographically computable, hence, they have to be approximated by polynomials, leading to degraded model accuracy; P3) only inference is supported, but training is unstable due to approximated polynomial activation functions. privacy-preserving training is considered in [2], which also utilizes polynomial approximation (e.g., Taylor expansion) to circumvent the difficulty of activations. Thus, it suffers from the same problems of CryptoNets including accuracy loss and training instability.

In this chapter, we propose a novel privacy-preserving learning architecture that resolves three problems of existing FHE-based approaches such as CryptoNets. It is dubbed *Globally Encrypted, Locally Unencrypted Deep Neural Network* (GELU-Net). The intrinsic strategy is to split each neuron into linear and nonlinear components and implement them separately on non-colluding parties. Linear computations are conducted based on a partially homomorphic cryptosystem, i.e., Paillier [48]. It offers sufficient security strength to keep data *globally encrypted*, and is significantly more efficient than FHE used in CryptoNets. As such, it solves P1. Note that it would be impossible to use Paillier without the novel design of separating the two components, because Paillier does not support nonlinear polynomials. The cryptographically incomputable activations are resolved in a *locally unencrypted* yet still privacy-preserving manner to retain the original accuracy, which solves P2 and P3. GELU-Net can effectively perform model training without the stability and accuracy loss issues. We

apply techniques such as random masking to surgically inject privacy-preserving components into the backpropagation algorithm, at minimal computation and communication cost while ensuring loss-free model accuracy.

Our contributions are summarized as follows: 1) we propose a novel privacy-preserving, computationally efficient, homomorphic encryption-based learning architecture, GELU-Net, which successfully resolves the three major problems of CryptoNets and other similar approaches; 2) we carry out security analysis and compare the complexity of GELU-Net with existing approaches; 3) we conduct extensive experiments on common datasets and demonstrate that GELU-Net achieves 14 to 35 times speed-up compared to CryptoNets in different environments.

5.2 PRELIMINARIES

Similar to Securetrain, the system model of GELU-Net includes both MLaaS and training as shown in Figure 1, and the threat model is semi-honest. We adopt a well-known partially homomorphic encryption system called *Paillier* [48]. *Paillier* supports unlimited number of additions between ciphertext, and multiplication between a ciphertext and a scalar constant.

5.3 SYSTEM DESCRIPTION

We first give an overview of GELU-Net and then elaborate the proposed protocol.

5.3.1 OVERVIEW OF GELU-NET ARCHITECTURE

In CryptoNets, the entire neural network is implemented on the server based on FHE operations. The private data are encrypted by the clients using FHE. Each client sends the encrypted data to the server, which runs the model and returns the encrypted inference result to the client. In GELU-Net, the overall neural network model is still implemented on the *server*. However, the nonlinear activation is securely outsourced and resolved in an unencrypted form. More specifically, each *client* uses *Paillier* to encrypt its private data (it is

referred as “Globally Encrypted”). Similar to CryptoNets, the encrypted data are sent to the neural network model on the *server*. The *server* is able to perform most computation based on the partially homomorphic encrypted data. However, it cannot compute the activation function, which is nonlinear and thus unsupported by the *Paillier* cryptosystem. To this end, the input for the activation (i.e., the intermediate weighted-sum in encrypted form) is sent back to the *client*, which, as the corresponding data owner, has the key and thus can decrypt the input (referred as “Locally Unencrypted”), execute the activation, re-encrypt the result, and send it to the *server* for the next layer.

The proposed GELU-Net has two prominent advantages as summarized below. The first advantage of this design is to enable activation without approximation, because it is now computed by the client in plaintext form. This ensures free of accuracy loss and the desired stability in training, thus addressing problems P2 and P3 introduced in 5.1. The second advantage is the significantly improved computation efficiency. The neural network runs much faster than CryptoNets, solving problem P1.

While the first advantage is obvious, the second seems counter-intuitive at the first glance. Given the proposed GELU-Net requires communication between server and client as well as decryption and encryption for computing each activation, would it become a performance bottleneck? Surprisingly, not only is it not a performance bottleneck, it also contributes significant performance gain. To fully understand such potential, we conduct a set of initial experiments on a commodity desktop using the Paillier package¹, and compare it with FHE implemented by Microsoft’s SEAL Library². Table 21 shows the different computation times of an activation function, approximated by square (that is used in CryptoNets and involves FHE multiplication between two ciphertext), 5-th order Taylor expansion (a better approximation using FHE), and our proposed approach where the activation is securely

¹Paillier Cryptosystem (in Python), <https://github.com/n1analytics/python-paillier>

²Simple Encrypted Arithmetic Library, <https://www.microsoft.com/en-us/research/project/simple-encrypted-arithmetic-library>

Scheme	Communication	Crypto	Activation	Total
Square	0	0	90.6	90.6
5-th order	0	0	1619.6	1619.6
GELU-Net	5	3.7	0.2	8.9

Table 21. Computation time of activation in different schemes (ms).

outsourced. The results show that, despite the cost paid for communication and encryption/decryption, the computation of an activation in GELU-Net is 10 times faster than the square approximation used in CryptoNets and about 180 times faster than the 5-th order approximation.

The above discussion is based on activation only. As discussed in 5.5, the overall performance gain is even higher, because other functions of GELU-Net are also implemented by *Paillier*, which enjoys significantly lower complexity than FHE as shown in [154,155]. To this end, GELU-Net aims to avoid FHE as long as *Paillier* is sufficient to meet privacy requirement. This would significantly improve computation efficiency and accordingly boost the overall performance. It is worth mentioning that running the entire neural network model on a client is not an option since we aim to perform collaborative learning, i.e., building a model utilizing the data from all clients.

5.3.2 PRIVACY-PRESERVING LEARNING ALGORITHMS

In this section, we elaborate the proposed privacy-preserving learning algorithms. For lucid presentation, the following description is based on training between a client and a server. The same process repeats for all clients. In *Paillier*, given a public key pair $(\mathbf{pk}_u, \mathbf{sk}_u)$ from party u , a vector of ciphertext is denoted as $[\mathbf{x}_i]_u$ encrypted by public key \mathbf{pk}_u . Initially, the client and server generate key pairs $(\mathbf{pk}_c, \mathbf{sk}_c)$ and $(\mathbf{pk}_s, \mathbf{sk}_s)$ respectively and publish their public keys. The proposed scheme consists of privacy-preserving forward propagation (Algorithm 1) and back propagation (Algorithm 2) as described below.

Algorithm 1: Privacy Preserved Forward Propagation

Input: *Client:* Data \mathbf{x} , set gradient/random accumulator $\Delta \mathbf{w}_i^c, \Delta \mathbf{b}_i^c, \mathbf{r}_i^{wc}, \mathbf{r}_i^{bc}$ to $\mathbf{0}$. *Server:* Initialize model, training bound d_{max} . Record initial parameters \mathbf{w}_i^I and \mathbf{b}_i^I (Section 4)

Output: Softmax output \mathbf{y}

```

1 for  $d = 1, 2, 3, \dots, d_{max}$  do
2   Client:  $\mathbf{a}_0 \leftarrow \mathbf{x}_d$ 
3   for  $i = 1, 2, \dots, n - 1$  do
4     Client: Encrypt  $\mathbf{a}_{i-1}$  with  $\mathbf{pk}_c$  as  $[\mathbf{a}_{i-1}]_c$  and send it to server
5     Server: Compute  $[\tilde{\mathbf{z}}_i]_c \leftarrow (\tilde{\mathbf{w}}_i \otimes [\mathbf{a}_{i-1}]_c) \oplus \tilde{\mathbf{b}}_i$  and send  $[\tilde{\mathbf{z}}_i]_c$  to client
6     Client: Decrypt  $[\tilde{\mathbf{z}}_i]_c$  with  $\mathbf{sk}_c$ , call Algorithm 4 to remove randomness in  $\tilde{\mathbf{z}}_i$ 
7     if  $i = n - 1$  then
8       Client:  $\mathbf{y} \leftarrow e^{\mathbf{z}_i} / \sum_j e^{\mathbf{z}_j}$  (softmax)
9     else
10      Client:  $\mathbf{a}_i \leftarrow f(\mathbf{z}_i)$  (next layer)
11  Call Algorithm 2 for back propagation

```

Privacy-preserving Forward Propagation

The forward propagation is summarized in Algorithm 1. The client first encrypts the data with \mathbf{pk}_c and sends it to the server. The weighted sum is homomorphically calculated by the server, $[\tilde{\mathbf{z}}_i]_c = (\tilde{\mathbf{w}}_i \otimes [\mathbf{a}_{i-1}]_c) \oplus \tilde{\mathbf{b}}_i$, which can be carried out by Paillier, since only one quantity is in the encrypted form. To prevent the server from inferring activations and data during the back propagation (which will be discussed next), random masks are applied on \mathbf{w}_i and \mathbf{b}_i (denote by $\tilde{\mathbf{w}}_i$ and $\tilde{\mathbf{b}}_i$, respectively). The encrypted weighted-sum $[\tilde{\mathbf{z}}_i]_c$ with random masks is sent back to the client for computing activation. The client calls Algorithm 4 to remove randomness in $\tilde{\mathbf{z}}_i$ and compute the activation for the next layer. The process repeats until the final layer is reached.

Note that in each layer i , the client can accumulate \mathbf{a}_{i-1} and \mathbf{z}_i over several iterations to solve the linear equation $\mathbf{w}_i \mathbf{a}_{i-1} + \mathbf{b}_i = \mathbf{z}_i$ for \mathbf{w}_i and \mathbf{b}_i . In an iteration, a number of m linear equations can be established (where m is the number of neuron in the layer). There are $m^2 + m$ unknowns including m^2 weighted connections and m biases. In the next iteration, an additional m equations are established while there is only one more unknown, i.e., the

Algorithm 2: Privacy Preserved Back Propagation

```

1 Server: Encrypt  $\frac{1}{\eta}$  with  $\text{pk}_s$  as  $[\frac{1}{\eta}]_s$  and send it to client
2 Client: For the last layer  $(n-1)$ , compute  $\delta_{n-1} \leftarrow \mathbf{y} - \mathbf{t}$ ,  $\Delta \mathbf{w}_{n-1} \leftarrow \mathbf{a}_{n-2} \delta_{n-1}$ ,  $\Delta \mathbf{b}_{n-1} \leftarrow \delta_{n-1}$ ,
3  $\Delta \mathbf{w}_{n-1}^c \leftarrow \Delta \mathbf{w}_{n-1}^c + \Delta \mathbf{w}_{n-1}$ ,  $\Delta \mathbf{b}_{n-1}^c \leftarrow \Delta \mathbf{b}_{n-1}^c + \Delta \mathbf{b}_{n-1}$ 
4 for  $i = n-2, n-3, \dots, 1$  do
5   Client: Encrypt  $\delta_{i+1}$  with  $\text{pk}_c$  as  $[\delta_{i+1}]_c$  and send it to server
6   Server: Compute  $[\tilde{\mathbf{q}}_{i+1}]_c \leftarrow [\delta_{i+1}]_c \otimes \tilde{\mathbf{w}}_{i+1}$ , and send it to client
7   Client: Decrypt  $[\tilde{\mathbf{q}}_{i+1}]_c$  with  $\text{sk}_c$ , call Algorithm 4 to remove randomness in  $\tilde{\mathbf{q}}_{i+1}$ , and calculate
       $\delta_i \leftarrow \delta_{i+1} \mathbf{w}_{i+1} \mathbf{a}_i (1 - \mathbf{a}_i)$ ,  $\Delta \mathbf{w}_i \leftarrow \mathbf{a}_{i-1} \delta_i$ ,  $\Delta \mathbf{b}_i \leftarrow \delta_i$ .
8   Update  $\Delta \mathbf{w}_i^c \leftarrow \Delta \mathbf{w}_i^c + \Delta \mathbf{w}_i$ ,  $\Delta \mathbf{b}_i^c \leftarrow \Delta \mathbf{b}_i^c + \Delta \mathbf{b}_i$ 
9   if  $d < d_{max}$  then
10     Client: Call Algorithm 3 to mask  $\Delta \mathbf{w}_i$  and  $\Delta \mathbf{b}_i$  as  $[\Delta \tilde{\mathbf{w}}_i]_s$  and  $[\Delta \tilde{\mathbf{b}}_i]_s$ , send to server
11     Server: Decrypt  $[\Delta \tilde{\mathbf{w}}_i]_s$  and  $[\Delta \tilde{\mathbf{b}}_i]_s$  with  $\text{sk}_s$ , and update  $\tilde{\mathbf{w}}_i \leftarrow \tilde{\mathbf{w}}_i - \eta \Delta \tilde{\mathbf{w}}_i$  and
            $\tilde{\mathbf{b}}_i \leftarrow \tilde{\mathbf{b}}_i - \eta \Delta \tilde{\mathbf{b}}_i$ 
12 Call Algorithm 1 for the next iteration or call Algorithm 5 to update model parameter on server
    when finish
  
```

learning rate η . This is because $\mathbf{w}_i = \mathbf{w}_i - \eta \Delta \mathbf{w}_i$ and $\Delta \mathbf{w}_i$ is known by the client during back propagation. Thus, the client can extract the model after $m+2$ iterations, which can accordingly cause leakage of the training data [107]. To address this problem, the server imposes a bound d_{max} randomly selected between $(1, m+2)$ so that a client's data can be used continuously for training. If d_{max} is reached, the next client is selected. The server can always return to the same client for training at a later time but not continuously exceeding d_{max} (see Section 5.4 for detailed analysis).

Privacy-preserving Back Propagation

As illustrated in Algorithm 2, back propagation starts from the last layer $i = n-1$ to compute the error δ_i between **softmax** prediction \mathbf{y} and true label \mathbf{t} . Note that Paillier can be used because all computations involve at most one quantity in the encrypted form. Then the error is propagated backward throughout the network via gradients $\Delta \mathbf{w}_i$ and $\Delta \mathbf{b}_i$ for all the layers. In order to correctly update weights on the server, the client must send private gradients to the server. Revealing such private gradients to the server can cause privacy leaks. To this end, the client calls Algorithm 3 to protect the gradients by random masking

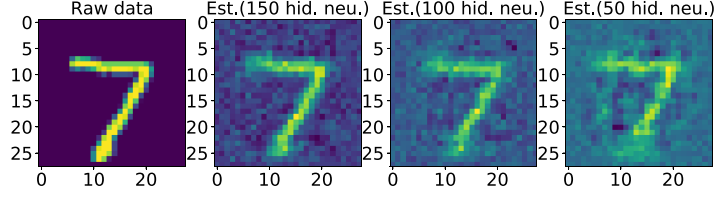


Figure 24. Server reconstructs training data via activations.

before sending them to the server.

On the other hand, the random mask should be removed by the client before the nonlinear activation; otherwise, it would be difficult to recover the original value after activation. To achieve this, the client calls Algorithm 4 to recover \mathbf{q}_i from masked $\tilde{\mathbf{q}}_i = \delta_i \tilde{\mathbf{w}}_i$. The client does this in each iteration so the error does not accumulate and there is no need to keep track of it. The client only needs to track the sum of the correct gradients $\Delta \mathbf{w}_i^c$ and $\Delta \mathbf{b}_i^c$, as well as the sum of injected randomness \mathbf{r}_i^{wc} and \mathbf{r}_i^{bc} in each gradient update for the final model update in Algorithm 5 when training finishes.

Secure Gradient Updates

The gradients should be protected during back propagation. Otherwise, the server can similarly establish $\Delta \mathbf{w}_i = \mathbf{a}_{i-1} \delta_i$, $\Delta \mathbf{b}_i = \delta_i$ from the received gradients and quickly derive the activations. From those private activations, the server can further invert the neural network to reconstruct user data [156]. Fig. 24 shows an example to reconstruct private data of handwritten digits. For fully connected networks, the server can simply utilize the Moore–Penrose inverse [157] to estimate data \mathbf{x} by, $\hat{\mathbf{x}} = \mathbf{w}_1^T (\mathbf{w}_1 \mathbf{w}_1^T)^{-1} (\mathbf{z}_1 - \mathbf{b}_1)$, where $\mathbf{z}_1 = f^{-1}(\mathbf{a}_1)$ is the inverse of the activation function from the first layer. To protect the gradients, random vectors are introduced to prevent the server from deriving activation and user data. For layer i , random vectors \mathbf{r}_i^w and \mathbf{r}_i^b (uniformly distributed over \mathbb{Z}_N) are generated by the client. Using the learning rate encrypted by the server $[\frac{1}{\eta}]_s$, the client injects the randomness into the encrypted gradients by homomorphically computing $[\Delta \tilde{\mathbf{w}}_i]_s = \Delta \mathbf{w}_i \oplus ([\frac{1}{\eta}]_s \otimes \mathbf{r}_i^w)$ and

Algorithm 3: Prevent Gradient Leakage by Client

Input: $\Delta \mathbf{w}_i, \Delta \mathbf{b}_i, [\frac{1}{\eta}]_s$, and $\mathbf{r}_i^w, \mathbf{r}_i^b \in \mathbb{Z}_N$
Output: $[\Delta \tilde{\mathbf{w}}_i]_s, [\Delta \tilde{\mathbf{b}}_i]_s, \mathbf{r}_i^{wc}$ and \mathbf{r}_i^{bc}
 1 $[\Delta \tilde{\mathbf{w}}_i]_s \leftarrow \Delta \mathbf{w}_i \oplus ([\frac{1}{\eta}]_s \otimes \mathbf{r}_i^w)$
 2 $[\Delta \tilde{\mathbf{b}}_i]_s \leftarrow \Delta \mathbf{b}_i \oplus ([\frac{1}{\eta}]_s \otimes \mathbf{r}_i^b)$
 3 $\mathbf{r}_i^{wc} \leftarrow \mathbf{r}_i^{wc} + \mathbf{r}_i^w, \mathbf{r}_i^{bc} \leftarrow \mathbf{r}_i^{bc} + \mathbf{r}_i^b$

Algorithm 4: Randomness Cancellation by Client

1 **if** *Forward propagation* **then**
 Input: $\tilde{\mathbf{z}}_i, \mathbf{a}_{i-1}, \mathbf{r}_i^{wc}$ and \mathbf{r}_i^{bc}
 Output: Recovered weighted sum \mathbf{z}_i
 2 $\mathbf{z}_i \leftarrow \tilde{\mathbf{z}}_i + \mathbf{r}_i^{wc} \mathbf{a}_{i-1} + \mathbf{r}_i^{bc}$
 3 **if** *Backpropagation* **then**
 Input: $\tilde{\mathbf{q}}_{i+1}, \boldsymbol{\delta}_{i+1}$, and \mathbf{r}_{i+1}^{wc}
 Output: Recovered error \mathbf{q}_{i+1}
 4 $\mathbf{q}_{i+1} \leftarrow \tilde{\mathbf{q}}_{i+1} + \boldsymbol{\delta}_{i+1} \mathbf{r}_{i+1}^{wc}$

$[\Delta \tilde{\mathbf{b}}_i]_s = \Delta \mathbf{b}_i \oplus ([\frac{1}{\eta}]_s \otimes \mathbf{r}_i^b)$ for weights and biases. The server decrypts the masked gradients by \mathbf{sk}_s and blindly updates the parameters as,

$$\begin{aligned}\tilde{\mathbf{w}}_i &= \tilde{\mathbf{w}}_i - \eta(\Delta \mathbf{w}_i + \mathbf{r}_i^w/\eta) = \tilde{\mathbf{w}}_i - \eta \Delta \mathbf{w}_i - \mathbf{r}_i^w, \\ \tilde{\mathbf{b}}_i &= \tilde{\mathbf{b}}_i - \eta(\Delta \mathbf{b}_i + \mathbf{r}_i^b/\eta) = \tilde{\mathbf{b}}_i - \eta \Delta \mathbf{b}_i - \mathbf{r}_i^b.\end{aligned}\tag{26}$$

In this way, the server is oblivious of the actual weights and has no way to figure out the activations (detailed proofs in Section 5.4). Note that random errors are accumulated at the server in each iteration. To perform activation on the actual weighted-sum, the client needs to remove randomness in $[\tilde{\mathbf{z}}_i]_c$ during forward propagation and $[\boldsymbol{\delta}_{i+1}]_c \otimes \tilde{\mathbf{w}}_{i+1}$ in back propagation. Eq. (26) shows that the actual weights/biases on server are $\mathbf{w}_i - \mathbf{r}_i^{wc}$ and $\mathbf{b}_i - \mathbf{r}_i^{bc}$ after each update. In forward propagation, to recover \mathbf{z}_i from $\tilde{\mathbf{z}}_i = (\mathbf{w}_i - \mathbf{r}_i^{wc})\mathbf{a}_{i-1} + \mathbf{b}_i - \mathbf{r}_i^{bc}$, the client adds $\mathbf{r}_i^{wc} \mathbf{a}_{i-1} + \mathbf{r}_i^{bc}$ to $\tilde{\mathbf{z}}_i$. Similarly, in back propagation, it adds $\boldsymbol{\delta}_{i+1} \mathbf{r}_{i+1}^{wc}$ to $[\tilde{\mathbf{q}}_{i+1}]_c$. These steps are summarized in Algorithm 4.

Algorithm 5: Final Parameter Updates

Input: Final values of $\Delta \mathbf{w}_i^c, \Delta \mathbf{b}_i^c$, initial weights \mathbf{w}_i^I and \mathbf{b}_i^I
Output: Final weights \mathbf{w}_i^F and \mathbf{b}_i^F

- 1 **for** $i = 1, 2, 3, \dots, n - 1$ **do**
- 2 Client: Encrypt $\Delta \mathbf{w}_i^c, \Delta \mathbf{b}_i^c$ with PK_s as $[\Delta \mathbf{w}_i^c]_s$ and $[\Delta \mathbf{b}_i^c]_s$ and send them to server
- 3 Server: Decrypt $[\Delta \mathbf{w}_i^c]_s$ and $[\Delta \mathbf{b}_i^c]_s$ with sk_s , update $\mathbf{w}_i^F = \mathbf{w}_i^I - \eta \Delta \mathbf{w}_i^c$ and $\mathbf{b}_i^F = \mathbf{b}_i^I - \eta \Delta \mathbf{b}_i^c$

Final Parameter Update

Once the training is completed, the final weights are updated on the server in one shot by subtracting the cumulative sum of actual gradients as shown in Algorithm 5.

While the above discussion is based on fully connected networks, a convolutional neural network (CNN)-based GELU-Net can be implemented in a similar way, since convolution is a linear operation and thus can be computed homomorphically. Max pooling can be adapted by mean pooling, thus handled by the server. Feature activations are returned to the clients and gradients are securely updated. Due to space limitation we skip the details but present its results in 5.5.

5.3.3 COMPLEXITY ANALYSIS

Communication Cost: The communication cost is analyzed as the total number of messages transmitted between the client and server. We assume a unit message size for encrypted data. For an n -layer network, in the forward propagation, the communication cost is $2m(n - 1)$, where m is the number of activations in a layer. This is because total m ciphertexts need to be transmitted by the client and the server, for the sum of inputs \mathbf{z}_i and activations \mathbf{a}_i , respectively, at each layer. In the back propagation, the client needs $m^2 + m + 1$ messages for model updates between two consecutive layers. Except the final layer, the client interacts with the server to calculate the gradients, which requires transmitting encrypted error $[\delta_{i+1}]_c$ in m messages between client and server for each layer. Summing up cost from the

forward and back propagation, the entire network requires $\mathcal{O}(nm^2)$ communication messages for an iteration.

Computation Cost: Arithmetic multiplications and additions are mapped to modular exponentiations and modular multiplications over ciphertext, respectively. Here, we denote such cost of conducting homomorphic arithmetic in Paillier by p . For n layers, both forward and back propagations take $\mathcal{O}(nm^2p)$ so the total computation cost is $\mathcal{O}(nm^2p)$.

Numerical Comparison: Two previous studies have considered privacy-preserving training for DNN. [2] uses a doubly homomorphic encryption called BGN [50] that supports one multiplication between ciphertext and unlimited additions. We call this scheme *BGN-Net* henceforth. [1] adopts ElGamal for homomorphic encryption [49], which supports either additive or multiplicative computation but not both. The arithmetic costs of BGN and ElGamal are denoted by b and e , respectively. Our tests show that the running time is $e \leq p \ll b$. BGN is at least 15 times slower than Paillier and Paillier is comparable with ElGamal. Note that CryptoNets do not support training, and thus is not comparable here.

Table 22 compares the complexity between different schemes in terms of computation and communication for an n -layer fully connected network. GELU-Net achieves over an order of magnitude improvements in the computation cost compared to ElGamal-Net, which requires all Z parties to participate in each iteration. In BGN-Net, since the 5-th order polynomial is employed to approximate activation, a large number of C homomorphic computations ($C > 30$) is needed using BGN. This actually gives GELU-Net a leverage. As long as the number of neurons per layer (m) is smaller than 450, GELU-Net is faster. Furthermore, a drawback of both schemes is that they are built on vertically divided data among users where the partial update of plaintext parameters will involve global activation values, from which the data distribution of other users can be derived.

Approach	Computation	Communication
GELU-Net	$\mathcal{O}(nm^2p)$	$\mathcal{O}(nm^2)$
BGN-Net	$\mathcal{O}(Cnmb)$	$\mathcal{O}(nm^2)$
ElGamal-Net	$\mathcal{O}(Z^2m^3ne)$	$\mathcal{O}(Z^2m^2n)$

Table 22. Complexity comparison (per iteration).

5.4 SECURITY ANALYSIS

In this section, we perform security analysis of GELU-Net against the well-known equation solving attack in the semi-honest model [107]. Recently, there are also other attacks to the neural networks, such as the membership attack [108] and adversarial examples [158,159]. These attacks target the vulnerabilities of the neural network itself, but are not directly relevant to the inherent privacy issues studied in this chapter.

Proposition 1. (Gradients protection in backpropagation) *The server cannot learn true values of $\Delta\mathbf{w}_i$ and $\Delta\mathbf{b}_i$ in order to reconstruct activations and private user data.*

Proof. The prove follows the simulation method [62]. The basic idea is to construct simulators given the input to a party and global output, and show that it learns nothing except the final result. During training, the server attempts to remove the randomness from the received gradients. Given a value r_j selected by the client and an attempt r_k from the server, both in the space of \mathbb{Z}_N , the probability that r_j equals r_k is $Pr\{r_j = r_k\} \leq 1 - e^{-2/|\mathbb{Z}_N|}$ [160]. $|\mathbb{Z}_N|$ is the size of a finite field identical to the cipher space of Paillier. Since the elements of the random mask is independent, the server can correctly yield matrices of \mathbf{r}_i^w and \mathbf{r}_i^b with probabilities $Pr\{\mathbf{r} = \mathbf{r}_i^w\} \leq (1 - e^{-2/|\mathbb{Z}_N|})^{m^2}$ and $Pr\{\mathbf{r} = \mathbf{r}_i^b\} \leq (1 - e^{-2/|\mathbb{Z}_N|})^m$. Because $|\mathbb{Z}_N|$ is a large number, the probability that the server can successfully derive the gradients is close to zero. ■

Proposition 2. (Model protection in forward propagation) *The accumulated function groups $\{\mathbf{z}_i = \mathbf{w}_i\mathbf{a}_{i-1} + \mathbf{b}_i\}_d$ reveal nothing but the subspaces of weights and bias from which*

the matrices \mathbf{w}_i and \mathbf{b}_i cannot be reconstructed by client.

Proof. Let $\mathbf{z}_{m \times 1}^{(i)} = \mathbf{w}_{m \times m}^{(i)} \mathbf{a}_{m \times 1}^{(i-1)} + \mathbf{b}_{m \times 1}^{(i)}$ denote the function group obtained by client after one forward propagation. Since a client is continuously trained for $d = d_{max}$ (less than the bound of $m + 2$ in Section 5.3.2), the function group does not reveal any information regarding the actual values of the matrices \mathbf{w}_i and \mathbf{b}_i but the subspaces linearly combined by infinitely many possible matrices solutions. Hence, model weights \mathbf{w}_i and \mathbf{b}_i cannot be successfully reconstructed by the client with $d = d_{max}$. ■

Proposition 3. (Gradients protection in final model update) *The accumulated parameter update groups $\{\mathbf{w}_i^F = \mathbf{w}_i^I - \eta \Delta \mathbf{w}_i^c, \mathbf{b}_i^F = \mathbf{b}_i^I - \eta \Delta \mathbf{b}_i^c\}_i$ reveal nothing but the subspaces of gradients from which the matrices $\Delta \mathbf{w}_i$ and $\Delta \mathbf{b}_i$ in the previous back propagations cannot be reconstructed.*

Proof. In the final model update, the client sends $\Delta \mathbf{w}_i^c$ and $\Delta \mathbf{b}_i^c$ to server. Since the client is allowed for d_{max} training iterations, the server ultimately obtains $d_{max} - 1$ pairs of randomized gradients $\Delta \tilde{\mathbf{w}}_i$ and $\Delta \tilde{\mathbf{b}}_i$. For each element in weight/bias matrices, there are totally d_{max} linear equations with $2d_{max} - 1$ unknown parameters ($d_{max} - 1$ random numbers and d_{max} gradients for each backward propagation). Since $d_{max} > 1$ and there is no way the server can add extra equations, the function group does not reveal any information regarding the actual values of $\Delta \mathbf{w}_i$ and $\Delta \mathbf{b}_i$ but the subspaces linearly combined by infinitely many possible matrices solutions. Therefore, the intermediate gradients cannot be reconstructed by the server. Fig. 25 shows an example when gradient protection is in place. We can see that the server can no longer reconstruct training data during back propagation. ■

5.5 PERFORMANCE EVALUATION

To evaluate the performance of GELU-Net, we use commodity workstations to implement the clients and server. The workstations have 2.8 GHz Intel Core i7 CPU and 8GB RAM

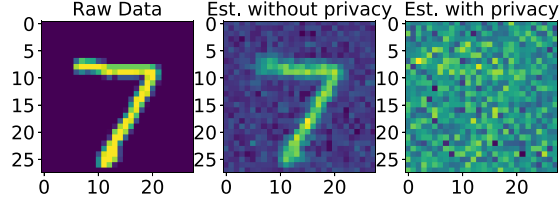


Figure 25. Server reconstruction with/without gradient protection.

connected by 1 Gbps LAN. The Paillier package is integrated with *Numpy* and *Theano* to build the neural network. We run experiments based on Iris, Diabetes, kr-vs-kp and MNIST datasets to compare with CryptoNets (implemented in Microsoft’s SEAL library) and BGN-Net in terms of training stability, accuracy and computation speed.

5.5.1 TRAINING STABILITY AND ACCURACY

Square activation is proposed in [161] to train convex objectives and inject hidden neurons in a gradual manner. Inspired by this idea, polynomial activation is adopted in secure neural networks. CryptoNets uses square function and BGN-Net uses the k -th order polynomial (Taylor expansion) to approximate the activation. However, polynomials may incur instability on non-convex objectives. Our results indicate that they make the network hard to train. In contrast, GELU-Net leaves the activation function unscathed so model parameters are still learnable in a privacy-preserving manner.

To compare the training stability, we implement the BGN-Net network architecture of 1 densely connected hidden layer with 5, 12, 15, 300 neurons. As shown in Fig. 26, the square activation of CryptoNets fails quickly. A higher order approximation (e.g., 3rd or 5th order) used by BGN-Net is better, but still unsuccessful as training terminates prematurely.

Next we compare the accuracy of these approaches in Table 23. Since CryptoNets and BGN-Net are unstable in training, they are pre-trained with plaintext data. Encrypted data are used for inference only. GELU-Net is able to retain the original model accuracy while other two approaches suffer an accuracy loss ranging from 2% to 7%. This makes GELU-Net

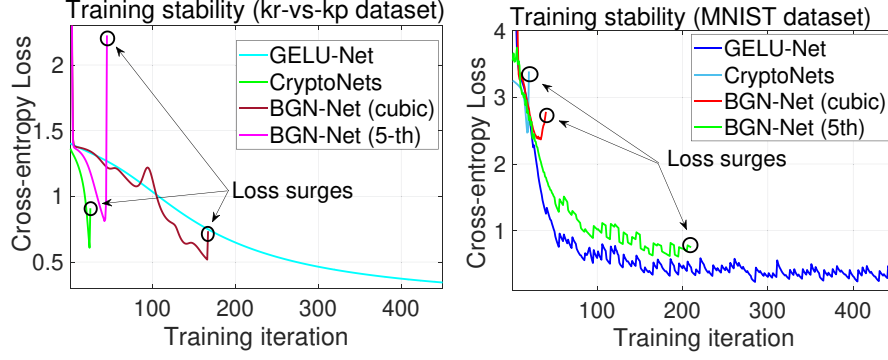


Figure 26. Training stability of different schemes.

Datasets	GELU-Net	CryptoNets	BGN-Net
Iris	0.986±0.004	0.966±0.012	0.96±0.007
Diabetes	0.760±0.011	0.741±0.023	0.723±0.028
kr-vs-kp	0.967±0.008	0.948±0.015	0.944±0.014
MNIST	0.969±0.004	0.919±0.0009	0.901±0.006

Table 23. Comparison of accuracy.

especially appealing in many smart applications on large datasets when model accuracy is the key consideration.

5.5.2 COMPUTATION SPEED

We compare the computation speed between GELU-Net and CryptoNets on MNIST. Note that our testing shows that BGN encryption is even slower than FHE, so we only compare GELU-Net with CryptoNets here. First, we adopt the CNN architecture used in CryptoNets (denoted by Conv-1) and then stack more convolutional layers to form an architecture identical to LeNet-5.

- Conv-1: Conv(5×5, stride 2, 5 filters)-ReLU (square)-Mean Pooling-ReLU (square)- Softmax.
- LeNet-5: Conv(5×5, stride 1, 6 filters)-Mean Pooling-ReLU (square)-Conv(5×5, stride 1, 16 filters)-Mean Pooling-ReLU (square)-Dense(120)-Dense(84)-Softmax.

Architecture	Time (s)	Accuracy
GELU-Net (Conv-1)	67.5±2.8	0.936±0.006
CryptoNets (Conv-1)	1271.8±1.9	0.909±0.002
GELU-Net (LeNet-5)	85.5±2.1	0.989±0.001
CryptoNets (LeNet-5)	3009.6±1.7	0.967±0.003

Table 24. Computation speed in different networks (s).

Architecture	Cloud-Local	Cloud-Cloud	Local-Local
Dense(12)	0.381±0.002	0.156±0.005	0.281±0.01
Dense(300)	147.5±9.8	59.7±1.9	107.4±3.5
Conv1	91.3±5.4	37.5±1.6	67.5±2.8
LeNet-5	126.7±6.3	47.5±1.2	85.5±2.1

Table 25. GELU-Net speed in different environments (s).

Since CryptoNets only supports inference, the model is loaded with pre-trained weights on MNIST. Table 24 shows the computation time (for one inference) and the accuracy. We observe that GELU-Net achieves 18 to 35× speed-up over CryptoNet with no accuracy loss. The performance gain is more obvious when the network gets deeper because more expensive homomorphic multiplications over ciphertext (for square activations) are required in CryptoNets. Since the network communication is an integral part of GELU-Net, we further evaluate its performance in two different environments. We first deploy the server in a cloud computing infrastructure while clients on workstations with different network domains. This scenario is denoted as Cloud-Local. Then we put both the server and clients on different virtual machines in the cloud, denoted as Cloud-Cloud. Table 25 shows the computation time for one inference. We observe that GELU-Net achieves optimal performance in the data center since the propagation delay is minimal. The communication cost increases when the client and server reside in different network domains. In the worst case, GELU-Net still achieves 14× speed-up compared to CryptoNets.

5.6 CHAPTER SUMMARY

In this chapter, the nonlinear calculation has been completed for free by a carefully partitioned DL framework, GELU-Net, where the server performs linear computation on encrypted data utilizing a less complex homomorphic cryptosystem, while the client securely executes non-polynomial computation in plaintext without approximation. GELU-Net has demonstrated $14\times$ to $35\times$ inference speedup compared to the classic systems.

CHAPTER 6

PERMUTATION ELIMINATION AND OT REDUCTION IN PPDL

While it is encouraging to witness the recent development in PPDL including works in previous chapters, there still exists a significant performance gap for its deployment in real-world applications. We have considered optimizing either the linear computation (in Chapter 3) or the linear and nonlinear computation of each layer (in Chapter 4 and Chapter 5). In this chapter, we further jointly consider the computation of two consecutive layers to optimize system efficiency. Specifically, we propose WISE¹, a novel hybrid protocol that features (1) a permutation-free scheme which completely eliminates the most expensive ciphertext permutation operations in the linear transformation and (2) a joint permutation-free computation between the nonlinear transformation in the current layer and the linear transformation in the next layer, which reduces the communication cost from 4.5 rounds to only a half round. As such, WISE achieves 2× to 13× speedup over CrypTFlow2 (ACM CCS’20) for various neural layers used in the state-of-the-art DL architectures. Furthermore, WISE demonstrates a speedup of 5.3×, 2×, 1.97×, 1.95×, 1.94×, 1.93×, 3.63×, 2.94× over CrypTFlow2 on practical DL models LeNet, AlexNet, VGG-11, VGG-13, VGG-16, VGG-19, ResNet-18, and ResNet-34. The rest of this chapter is organized as follows. Section 6.1 details the motivation of WISE. Section 6.2 introduces the primitives that WISE adopts. Section 6.3 describes the design details of WISE. Section 6.4 presents the security analysis of WISE. The experimental results are illustrated and discussed in Section 6.5. Finally, Section 6.6 concludes this chapter.

¹Under submission.

6.1 MOTIVATION

To achieve usable privacy-preserving MLaaS, a series of recent works have made inspiring progress towards system efficiency [3, 5, 7, 64–66, 75, 76, 78, 87, 90, 102, 133, 162–164]. Specifically, the inference speed has gained several orders of magnitude from CryptoNets [78] to the most recent frameworks such as CrypTFlow2 [7]. At a high level, several cryptographic primitives, e.g., the Homomorphic Encryption (HE) [51–53] and Multi-Party Computation (MPC) techniques [150] (such as Oblivious Transfer (OT) [114], Secret Sharing (SS) [61] and Garbled Circuits (GC) [104, 165]), are carefully considered and adopted in those systems to compute the linear (e.g., dot product and convolution) and nonlinear (e.g., activation) functions, which are the building blocks in a DL model. For example, CrypTFlow2 is currently one of the leading frameworks for privacy-preserving DL and has shown significant speedup compared with other schemes such as GAZELLE [64] and DELPHI [66]. The input of CrypTFlow2 is the private data from the client. It is encrypted and sent to the server which performs HE-based computation for the linear function. Therein the HE addition, multiplication, and permutation, which are three basic operators over encrypted data, are performed between encrypted data from the client and model parameters at the server. The output is the respective shares (in plaintext) of the linear function at the client and server, which serve as the input of the following OT-based computation for the nonlinear function. The corresponding output (shares) acts as the input for the next layer’s linear computation. The computation is repeated layer by layer until the final output.

While it is encouraging to witness the recent development in privacy-preserving DL, there still exists a significant performance gap for its deployment in real-world applications. For example, our benchmark has shown that CrypTFlow2 takes 115 seconds and 147 seconds to run the well-known DL networks **VGG-19** [16] and **ResNet-34** [26] on the Intel(R) Xeon(R) E5-2666 v3 @ 2.90GHz CPU (see the detailed experimental settings and results in Section 6.5). It is worth pointing out that the response time constraints in many practical applications

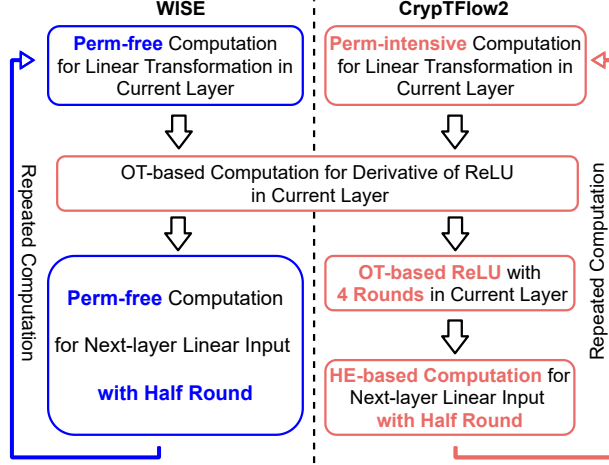


Figure 27. WISE protocol compared with CryptFlow2 [7].

(such as speech recognition and wearable health monitoring) are within a few seconds or up to one minute [99, 166]. This performance gap motivates us to further improve the efficiency of privacy-preserving DL.

Our Contributions. In this chapter, we propose WISE (tWeaking prIvacy-preSeRving DL inference through permutation Elimination and OT reduction), which features a novel hybrid protocol to reduce the prediction latency by replacing expensive operations with faster ones throughout the linear and nonlinear computation. More specifically, WISE proposes the following new techniques:

1. a permutation-free scheme, which completely eliminates the most expensive ciphertext permutation operations in the HE-based linear transformation, and
2. a joint permutation-free computation between the nonlinear transformation in the current layer and the linear transformation in the next layer, which reduces the communication cost from 4.5 rounds to only a half round², introducing negligible computation overhead.

²One round is the communication trip from source node to sink node and then from sink node back to source node, while 0.5 round is either communication trip from source node to sink node or the one from sink node to source node.

The above two key techniques enable WISE to achieve significant improvement in prediction latency over CrypTFlow2. Figure 27 compares WISE with CrypTFlow2 in terms of the computation and communication in each neural layer. WISE involves no permutations in the linear transformation, while CrypTFlow2 relies on a series of permutations to obtain the result that is required for the subsequent nonlinear computation. For instance, 2048 permutations are needed to calculate one of the convolutions³ in **ResNet** [26]. In contrast, WISE enables a permutation-free calculation that effectively reduces the computation complexity. After the linear transformation, WISE and CrypTFlow2 both follow an OT-based computation to obtain the derivative of **ReLU**⁴. After that, CrypTFlow2 involves another OT-based computation with 4 rounds of communication between the client and server to finally get the **ReLU** output plus another half-round communication to support the HE computation to form the input for the next layer. On the other hand, WISE features a joint permutation-free strategy that efficiently integrates the linear and nonlinear computation. It results in a total of only a half round communication to construct the input for the next-layer’s linear transformation. As a result, WISE achieves over $2\times$ to $13\times$ speedup for various neural layers used in state-of-the-art DL architectures.

6.2 PRELIMINARIES

We consider a privacy-preserving MLaaS system shown in Figure 1 (a). The threat model in WISE is similar to that of the prior privacy-preserving frameworks such as MiniONN [65], GAZELLE [64], DELPHI [66] and CrypTFlow2 [7]. WISE relies on PHE, additive Secret Sharing (SS) and Oblivious Transfer (OT)⁵. Among the three basic PHE operations, Perm

³Convolution [167] is one of the typical linear functions in modern DL models and the detailed permutation complexity is analyzed in Section 6.3.

⁴One can obtain the **ReLU** output based on its derivative as described in Section 6.3.

⁵As described in Section 2.3.

is the most expensive [64]. WISE totally eliminates the Perm such that the overall computation cost is noticeably reduced. Furthermore, the multiplication involved in WISE is *scalar multiplication* between one ciphertext and one plaintext, which is faster than the ciphertext-ciphertext counterpart. Meanwhile, we apply the 2-of-2 additive secret sharing to share the linear result at \mathcal{S} as well as the nonlinear result at \mathcal{C} such that the computation over ciphertext is replaced by the counterpart over shares (in plaintext), which efficiently reduces the computation overhead. Finally, the state-of-the-art approaches rely on OT-based multiplication in ReLU computation [7], which involves 4 rounds of communication. In a contrast, the proposed WISE protocol *totally eliminates* this overhead by reconstructing the multiplication formula in ReLU computation (see details in Section 6.3), thus contributing to the reduction of the overall computation cost.

6.3 SYSTEM DESCRIPTION

The proposed WISE protocol introduces two novel techniques to reduce the prediction latency in privacy-preserving DL. First, it eliminates the most expensive operation in HE, i.e., permutation for linear transformation. Second, the joint computation for nonlinear (in the current layer) and linear (in the next layer) transformations cuts down the communication cost from 4.5 rounds to only a half round, introducing the negligible computation overhead.

In this section, we elaborate WISE based on the system model introduced in Section 6.2. We use a simple two-layer DL model for a lucid presentation, but note that WISE is applicable to the state-of-the-art DL models with more complex layer structures and input data sizes. Specifically, the example DL model is expressed as:

$$\mathbf{z} = \mathbf{w} \cdot f(\mathbf{k} * \mathbf{x}), \quad (27)$$

where $f(\cdot)$ is the ReLU activation function⁶, \mathbf{x} is the 2×2 input data, \mathbf{k} is a 3×3 kernel for

⁶We use $f(\cdot)$ to specifically denote the ReLU function through out the chapter.

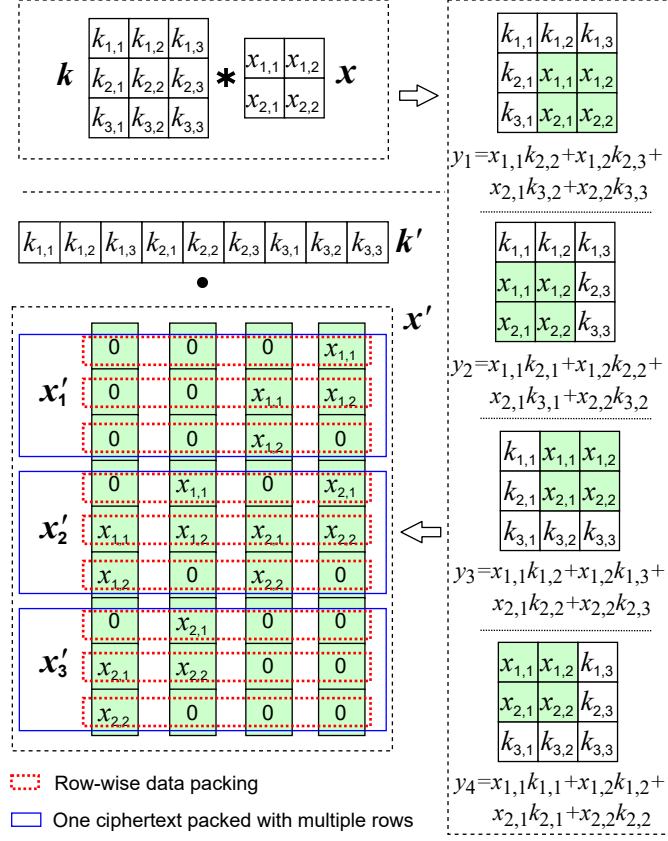


Figure 28. Data transformation for convolution.

the convolution layer, “ $*$ ” stands for convolution, “ \cdot ” stands for dot product and w is the weight matrix for the fully-connected (dense) layer:

$$x = \begin{bmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{bmatrix}, \quad k = \begin{bmatrix} k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix} \quad \text{and also}$$

$$w = \begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \\ w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} \end{bmatrix}.$$

6.3.1 PERM-FREE COMPUTATION FOR LINEAR TRANSFORMATION

Let's start with the linear computation for the first layer where the client \mathcal{C} encrypts its private input \mathbf{x} by HE, e.g., BFV algorithm [52], into $[\mathbf{x}]_{\mathcal{C}}$ and sends it to the cloud server \mathcal{S} . \mathcal{S} then performs perm-free operations over the encrypted input $[\mathbf{x}]_{\mathcal{C}}$ based on its model parameters \mathbf{k} and sends the result back to \mathcal{C} , which decrypts the linear result and uses it as the input for the subsequent nonlinear computation. In the later layers, the input to \mathcal{S} 's linear transformation is the encrypted output of the nonlinear transformation from the previous layer (i.e., the encrypted $f(\mathbf{k} * \mathbf{x})$ in our case), which will be discussed in Section 6.3.2. As such the \mathcal{S} similarly conducts perm-free HE operations for linear computation and sends the result (i.e., the final result of $\mathbf{w} \cdot f(\mathbf{k} * \mathbf{x})$ in our case) back to the client. The client accordingly conducts the decryption and repeats the process until the last layer.

Perm-free Computation for Linear Transformation in the First Layer

If both the input \mathbf{x} and the kernel \mathbf{k} are plaintext, the convolution between \mathbf{x} and \mathbf{k} should yield a vector \mathbf{y} with four elements as $\mathbf{y} = \mathbf{k} * \mathbf{x} = (y_1, y_2, y_3, y_4)$:

$$\begin{cases} y_1 = x_{1,1}k_{2,2} + x_{1,2}k_{2,3} + x_{2,1}k_{3,2} + x_{2,2}k_{3,3} \\ y_2 = x_{1,1}k_{2,1} + x_{1,2}k_{2,2} + x_{2,1}k_{3,1} + x_{2,2}k_{3,2} \\ y_3 = x_{1,1}k_{1,2} + x_{1,2}k_{1,3} + x_{2,1}k_{2,2} + x_{2,2}k_{2,3} \\ y_4 = x_{1,1}k_{1,1} + x_{1,2}k_{1,2} + x_{2,1}k_{2,1} + x_{2,2}k_{2,2}, \end{cases}$$

which is actually a dot product $\mathbf{y} = \mathbf{k}' \cdot \mathbf{x}'^7$ as shown in Figure 28. Here \mathbf{k}' is the flattened vector of \mathbf{k} in row wise, and \mathbf{x}' is constructed in a way that the dot product of its i -th column and \mathbf{k}' produces y_i . For example, it is easy to verify that the dot product between \mathbf{k}' and the first column of \mathbf{x}' results in y_1 , so on and so forth.

Clearly, the above dot product of $\mathbf{k}' \cdot \mathbf{x}'$ can also be expressed as follows: the values in

⁷Note that any convolution can be similarly converted into dot product computation as shown in [151].

j -th row of \mathbf{x}' are firstly multiplied with the j -th value in \mathbf{k}' , and these multiplied rows are then added up to get the final result. For example, the first row in \mathbf{x}' is multiplied with the first element in \mathbf{k}' , i.e., $k_{1,1}$, and the second row in \mathbf{x}' is multiplied with the second element in \mathbf{k}' , i.e., $k_{1,2}$, , so on and so forth. After all rows are multiplied with the corresponding kernel values, the result \mathbf{y} is calculated by adding all multiplied rows in \mathbf{x}' . Therefore, we observe the following result.

Observation 1. If the client \mathcal{C} encrypts each row of \mathbf{x}' as one ciphertext, then the convolution can be achieved by the server \mathcal{S} with only homomorphic multiplication and addition, as analogue with the above plaintext computation, which *totally eliminates the most expensive permutation*.

While the above observation is promising, it is not directly applicable in privacy-preserving DL, because encrypting each row of \mathbf{x}' would result in a significant encryption and communication cost at the client. This cost is proportional to the size of \mathbf{k}' , i.e., the number of rows of \mathbf{x}' , that can be over one thousand in state-of-the-art DL networks such as AlexNet [15], VGG [16] and ResNet [26].

In order to make the permutation-free computation a true advantage, WISE combines the encryption of packed HE and the computation over plaintext shares to reduce the overall cost for convolution. Specifically, multiple rows are firstly packed into one ciphertext. For example, let's assume that three rows in \mathbf{x}' can be packed in one ciphertext by \mathcal{C} , which forms three ciphertexts: $[\mathbf{x}'_1]_{\mathcal{C}}$, $[\mathbf{x}'_2]_{\mathcal{C}}$, and $[\mathbf{x}'_3]_{\mathcal{C}}$, as shown in Figure 28. These three ciphertexts are sent to \mathcal{S} , which first conducts three HE multiplications⁸:

$$[\mathbf{x}'_1]_{\mathcal{C}} \otimes \mathbf{k}'_1, \quad [\mathbf{x}'_2]_{\mathcal{C}} \otimes \mathbf{k}'_2, \quad [\mathbf{x}'_3]_{\mathcal{C}} \otimes \mathbf{k}'_3,$$

where \mathbf{k}'_i ($i \in \{1, 2, 3\}$) are transformed from \mathbf{k}' to make sure that each row (from \mathbf{x}') in

⁸We denote “ \otimes ” as homomorphic multiplication in element wise manner if any ciphertext is involved. Otherwise “ \otimes ” is plaintext multiplication in element manner. Similar logic is applied to addition denoted as “ \oplus ” or “ \ominus ”.

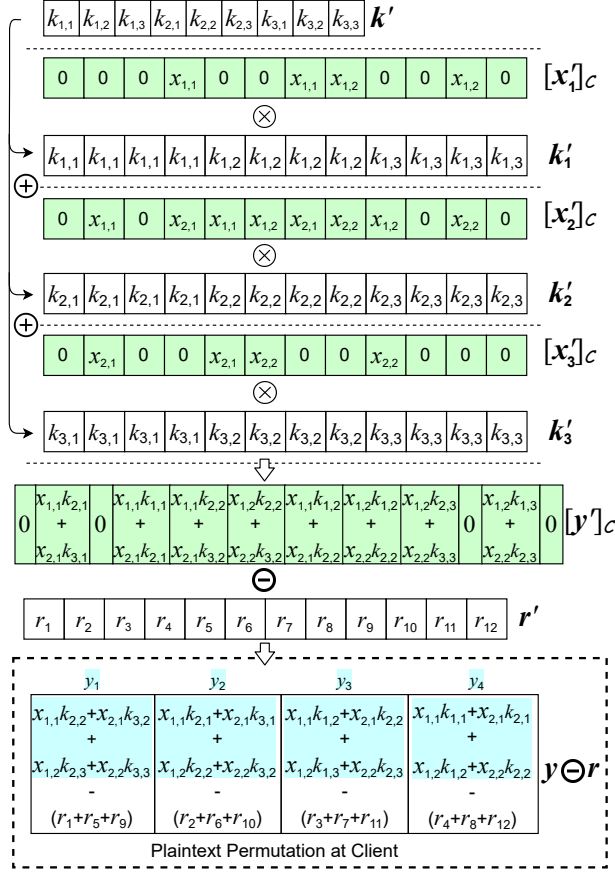


Figure 29. Perm-free convolution.

\mathbf{x}'_i is multiplied with the corresponding value in \mathbf{k}' . For example, as shown in Figure 29, $[\mathbf{x}'_1]_C$ is the encryption of \mathbf{x}'_1 , which includes the first three rows from \mathbf{x}' . Therefore, \mathbf{k}'_1 is constructed such that the first three values from \mathbf{k}' are respectively multiplied with the first three rows contained in \mathbf{x}'_1 . Similar construction is applied to \mathbf{k}'_2 and \mathbf{k}'_3 .

Since each multiplied row in \mathbf{x}' (i.e., the row that is multiplied with corresponding value in \mathbf{k}') can be added as shown in Figure 28, the resultant three ciphertext are then added up to form

$$[\mathbf{y}']_C = ([\mathbf{x}'_1]_C \otimes \mathbf{k}'_1) \oplus ([\mathbf{x}'_2]_C \otimes \mathbf{k}'_2) \oplus ([\mathbf{x}'_3]_C \otimes \mathbf{k}'_3).$$

As shown in Figure 29, $[\mathbf{y}']_C$ contains the encrypted partial sum of convolution. At this point, a straightforward approach is to conduct a series of permutations and additions

over $[\mathbf{y}']_{\mathcal{C}}$ to get an encrypted \mathbf{y} . This approach is adopted in the state-of-the-art privacy-preserving DL frameworks such as [7, 64, 66, 102]. More specifically, two other ciphertexts are formed by respectively rotating $[\mathbf{y}']_{\mathcal{C}}$ such that the first element in each rotated ciphertext is the fifth and ninth element in \mathbf{y}' . By adding the above two rotated ciphertexts as well as the original $[\mathbf{y}']_{\mathcal{C}}$, the first four elements in the resultant ciphertext are exactly the four elements in \mathbf{y} . Actually, this permutation and addition is analogous to adding the three four-element blocks in $[\mathbf{y}']_{\mathcal{C}}$ that have partial sum of the convolution. After \mathcal{S} gets the encrypted \mathbf{y} , it then generates a share \mathbf{r} and sends $[\mathbf{y}]_{\mathcal{C}} \ominus \mathbf{r} = [\mathbf{y} \ominus \mathbf{r}]_{\mathcal{C}}$ to the client, which decrypts $[\mathbf{y} \ominus \mathbf{r}]_{\mathcal{C}}$ into $(\mathbf{y} \ominus \mathbf{r})$. As a result, the client and server respectively own a share of \mathbf{y} . The \mathbf{r} from \mathcal{S} and $(\mathbf{y} \ominus \mathbf{r})$ from \mathcal{C} are the inputs of the subsequent nonlinear computation, e.g., ReLU function.

This conventional approach is obviously not permutation-free. In contrast, we discover an interesting observation, which shows that it is actually unnecessary to compute $[\mathbf{y}]_{\mathcal{C}}$.

Observation 2. Given $[\mathbf{y}']_{\mathcal{C}}$, there are two equivalent ways to generate the shares of \mathbf{y} , i.e., \mathbf{r} at \mathcal{S} and $(\mathbf{y} \ominus \mathbf{r})$ at \mathcal{C} . One approach (i.e., the conventional approach) is to first derive $[\mathbf{y}]_{\mathcal{C}}$ over $[\mathbf{y}']_{\mathcal{C}}$ and then generate the shares. The other approach is to first share $[\mathbf{y}']_{\mathcal{C}}$ between \mathcal{S} and \mathcal{C} , and then each party uses its share to derive its corresponding share of \mathbf{y} .

Based on the above observation, we propose a novel, permutation-free scheme. It does not compute $[\mathbf{y}]_{\mathcal{C}}$ over $[\mathbf{y}']_{\mathcal{C}}$, and thus completely eliminates the expensive permutation over ciphertext $[\mathbf{y}']_{\mathcal{C}}$. More specifically, as shown in Figure 29, \mathcal{S} directly generates the share of \mathbf{y}' , i.e., \mathbf{r}' , and sends $[\mathbf{y}']_{\mathcal{C}} \ominus \mathbf{r}' = [\mathbf{y}' \ominus \mathbf{r}']_{\mathcal{C}}$ to \mathcal{C} . \mathcal{C} decrypts $[\mathbf{y}' \ominus \mathbf{r}']_{\mathcal{C}}$ into $(\mathbf{y}' \ominus \mathbf{r}')$, and get $(\mathbf{y} \ominus \mathbf{r})$ by a series of rotation and addition in *plaintext*. Specifically, \mathcal{C} rotates $(\mathbf{y}' \ominus \mathbf{r}')$ to form two plaintexts such that the first element in each rotated plaintext is the fifth and ninth element in $(\mathbf{y}' \ominus \mathbf{r}')$. As such, $(\mathbf{y} \ominus \mathbf{r})$ is obtained by adding the two rotated plaintexts with $(\mathbf{y}' \ominus \mathbf{r}')$, as shown in Figure 29. Meanwhile, \mathcal{S} rotates \mathbf{r}' to form two plaintexts such that the first element in each rotated plaintext is the fifth and ninth element in \mathbf{r}' , and \mathbf{r} is then obtained by adding the two rotated plaintexts with \mathbf{r}' .

The rotation and addition calculations are similar to the method discussed earlier to obtain $[\mathbf{y}]_{\mathcal{C}}$ from $[\mathbf{y}']_{\mathcal{C}}$ at \mathcal{S} , but it is much faster because it is based on plaintext with no permutation over ciphertext. We will provide detailed complexity analysis in Section 6.3.3 to show WISE’s fundamental improvement for speeding up privacy-preserving DL inference computation. It is easy to verify that the proposed scheme achieves the same result as the conventional approach: \mathcal{C} and \mathcal{S} respectively owns a share of \mathbf{y} , i.e., \mathbf{r} from \mathcal{S} and $(\mathbf{y} \ominus \mathbf{r})$ from \mathcal{C} , that will be the input of the subsequent nonlinear computation.

Now let’s put all pieces together to summarize the perm-free computation for linear transformation in the first layer. Client \mathcal{C} transforms input \mathbf{x} into \mathbf{x}' and respectively encrypts \mathbf{x}'_1 , \mathbf{x}'_2 and \mathbf{x}'_3 as $[\mathbf{x}'_1]_{\mathcal{C}}$, $[\mathbf{x}'_2]_{\mathcal{C}}$ and $[\mathbf{x}'_3]_{\mathcal{C}}$ (see Figure 28). These three ciphertexts are sent to \mathcal{S} . \mathcal{S} computes a ciphertext $[\mathbf{y}' \ominus \mathbf{r}']_{\mathcal{C}}$ by HE multiplication and addition based on its transformed kernels \mathbf{k}'_1 , \mathbf{k}'_2 and \mathbf{k}'_3 (see Figure 29). The ciphertext is subsequently sent back to \mathcal{C} , which decrypts $[\mathbf{y}' \ominus \mathbf{r}']_{\mathcal{C}}$ into $(\mathbf{y}' \ominus \mathbf{r}')$ to obtain its share of the linear result, i.e., $(\mathbf{y} \ominus \mathbf{r})$, by plaintext rotation and addition. $(\mathbf{y} \ominus \mathbf{r})$ and \mathbf{r} (obtained from \mathbf{r}' by \mathcal{S} as shown in Figure 29) are fed into the nonlinear computation module, i.e., $f(\mathbf{y})$, to be discussed in Section 6.3.2. Note that the bias, \mathbf{b} , which is another model parameter as an additive term to the convolution as $(\mathbf{y} \oplus \mathbf{b})$, is easily added to \mathbf{r}' such that \mathcal{C} and \mathcal{S} will obtain the shares of $(\mathbf{y} \oplus \mathbf{b})$.

Perm-free Computation for Linear Transformation in the l -th Layer ($l > 1$)

In the first layer, the input of linear transformation at \mathcal{S} is the ciphertext from \mathcal{C} (e.g., $[\mathbf{x}'_1]_{\mathcal{C}}$, $[\mathbf{x}'_2]_{\mathcal{C}}$, and $[\mathbf{x}'_3]_{\mathcal{C}}$ in Figure 28). Similarly, the input to \mathcal{S} for the linear transformation in other layers is the (encrypted) output of nonlinear computation, i.e., $f(\mathbf{k} * \mathbf{x})$ or $f(\mathbf{y})$ in our example DL model shown in Eq. (27). As will be discussed in Section 6.3.2, the nonlinear output is similar to $[\mathbf{x}'_i]_{\mathcal{C}}$ ($i \in \{1, 2, 3\}$), which is ready to be processed by \mathcal{S} in a way as shown in Figure 29. As such, the linear calculation at \mathcal{S} for other layers is in line with the calculation in the first layer. As a result, the client and server respectively obtains

a share, i.e., \mathbf{r} (at \mathcal{S}) and $(\mathbf{y} \ominus \mathbf{r})$ (at \mathcal{C}) that will be the input of the subsequent nonlinear computation.

6.3.2 PERM-FREE JOINT LINEAR AND NONLINEAR COMPUTATION

Recall that at the end of the linear transformation, \mathcal{C} and \mathcal{S} respectively gets the share of the convolution $\mathbf{y} = \mathbf{k} * \mathbf{x}$, i.e., $(\mathbf{y} \ominus \mathbf{r})$ and \mathbf{r} . The next step is to compute the nonlinear ReLU function $f(\mathbf{y})$, based on the shares at \mathcal{C} and \mathcal{S} .

As shown in [4], $f(\mathbf{y})$ can be calculated as:

$$f(\mathbf{y}) = f_D(\mathbf{y}) \otimes \mathbf{y}, \quad (28)$$

where $f_D(\cdot)$ is the derivative of the ReLU function:

$$f_D(y_i) = \begin{cases} 1, & \text{if } y_i > 0 \\ 0, & \text{others} \end{cases},$$

where $i \in \{1, 2, 3, 4\}$ for our example.

It has been shown in [7] that $f_D(\mathbf{y})$ can be efficiently computed where the input is the shares of linear transformation, e.g., $(\mathbf{y} \ominus \mathbf{r})$ and \mathbf{r} in our case, and the output is the Boolean shares⁹ of $f_D(\mathbf{y})$ denoted as $\hat{\mathbf{a}}_{\mathcal{C}} \in \{0, 1\}^{410}$ (at \mathcal{C}) and $\hat{\mathbf{a}}_{\mathcal{S}} \in \{0, 1\}^4$ (at \mathcal{S}) such that the XOR between $\hat{\mathbf{a}}_{\mathcal{C}}$ and $\hat{\mathbf{a}}_{\mathcal{S}}$ is $f_D(\mathbf{y})$ ¹¹. However, it is worth pointing out that the computation of the final multiplication, $f_D(\mathbf{y}) \otimes \mathbf{y}$, needs two OT processes with four rounds of communication between \mathcal{C} and \mathcal{S} , which account for about half of the total communication rounds as shown in the state-of-the-art framework CrypTFlow2 [7]. Specifically, the input of this OT module is the shares of $f_D(\mathbf{y})$, i.e., $\hat{\mathbf{a}}_{\mathcal{C}}$ and $\hat{\mathbf{a}}_{\mathcal{S}}$, and the shares of \mathbf{y} , i.e., $(\mathbf{y} \ominus \mathbf{r})$ and \mathbf{r} , while the

⁹The shares are arithmetic shares if not specified.

¹⁰The dimension of 4 is for our example.

¹¹We refer the readers to [7] for more details about the privacy-preserving calculation for derivative of ReLU.

output is the shares of nonlinear result $f(\mathbf{y})$ denoted as $\mathbf{a}_\mathcal{C}$ (at \mathcal{C}) and $\mathbf{a}_\mathcal{S}$ (at \mathcal{S}) such that $\mathbf{a}_\mathcal{C} \oplus \mathbf{a}_\mathcal{S} = \mathbf{a} = f(\mathbf{y})$. After that, \mathcal{C} encrypts $\mathbf{a}_\mathcal{C}$ as $[\mathbf{a}_\mathcal{C}]_\mathcal{C}$ and sends it to \mathcal{S} . \mathcal{S} adds $[\mathbf{a}_\mathcal{C}]_\mathcal{C}$ with $\mathbf{a}_\mathcal{S}$ to obtain

$$[\mathbf{a}_\mathcal{C}]_\mathcal{C} \oplus \mathbf{a}_\mathcal{S} = [\mathbf{a}_\mathcal{C} \oplus \mathbf{a}_\mathcal{S}]_\mathcal{C} = [f(\mathbf{y})]_\mathcal{C},$$

which forms the encrypted input for the linear computation in the next layer. Therefore, there is a total amount of 4.5 communication rounds to get the next-layer linear input after \mathcal{C} and \mathcal{S} obtain the shares of derivative of ReLU.

Perm and OT-Free Computation after $f_\mathbf{D}$

We now show that the computation and communication costs can be further reduced by combining the two separate processes, i.e., the OT-based multiplication to get the shares of $f(\mathbf{y})$ and the HE-based data encryption and computation to get $[f(\mathbf{y})]_\mathcal{C}$. Specifically, given the shares of linear transformation, $(\mathbf{y} \ominus \mathbf{r})$ (at \mathcal{C}) and \mathbf{r} (at \mathcal{S}), and the shares of derivative of ReLU $f_\mathbf{D}(\mathbf{y})$, $\hat{\mathbf{a}}_\mathcal{C}$ (at \mathcal{C}) and $\hat{\mathbf{a}}_\mathcal{S}$ (at \mathcal{S}), we aim to reconstruct Eq. (28) such that \mathcal{C} and \mathcal{S} directly get the respective share of $f(\mathbf{y})$ without the four-round OTs.

To this end, we rewrite Eq. (28) as follows to aggregate the terms that can be calculated locally by \mathcal{C} and \mathcal{S} , respectively:

$$\begin{aligned} f(\mathbf{y}) &= f_\mathbf{D}(\mathbf{y}) \otimes \mathbf{y} \\ &= \underbrace{(\hat{\mathbf{a}}_\mathcal{C} \oplus \hat{\mathbf{a}}_\mathcal{S} \ominus (\mathbf{2} \otimes \hat{\mathbf{a}}_\mathcal{C} \otimes \hat{\mathbf{a}}_\mathcal{S}))}_{\mathbf{1}} \otimes ((\mathbf{y} \ominus \mathbf{r}) \oplus \mathbf{r}) \\ &= \underbrace{((\mathbf{y} \ominus \mathbf{r}) \otimes \hat{\mathbf{a}}_\mathcal{C})}_{\mathbf{2}} \oplus \underbrace{((\mathbf{y} \ominus \mathbf{r}) \otimes (\mathbf{1} \ominus (\mathbf{2} \otimes \hat{\mathbf{a}}_\mathcal{C})) \otimes \hat{\mathbf{a}}_\mathcal{S})}_{\mathbf{3}} \oplus \\ &\quad \underbrace{((\mathbf{r} \ominus (\mathbf{2} \otimes \mathbf{r} \otimes \hat{\mathbf{a}}_\mathcal{S})) \otimes \hat{\mathbf{a}}_\mathcal{C})}_{\mathbf{4}} \oplus \underbrace{(\mathbf{r} \otimes \hat{\mathbf{a}}_\mathcal{S})}_{\mathbf{5}}, \end{aligned}$$

where $\mathbf{1} = \{1\}^{412}$, $\mathbf{2} = \{2\}^4$, and the term $\mathbf{1}$ obtains the value of $f_\mathbf{D}(\mathbf{y})$ from Boolean shares

¹²The dimension of 4 is for our example.

$\hat{\mathbf{a}}_{\mathcal{C}}$ and $\hat{\mathbf{a}}_{\mathcal{S}}$. Recall that \mathcal{C} owns $(\mathbf{y} \ominus \mathbf{r})$ and $\hat{\mathbf{a}}_{\mathcal{C}}$, and \mathcal{S} owns \mathbf{r} and $\hat{\mathbf{a}}_{\mathcal{S}}$. It is also worth pointing out that both \mathbf{r} and $\hat{\mathbf{a}}_{\mathcal{S}}$ can be pregenerated because \mathbf{r} is independent of the input \mathbf{x} and one of the shares for $f_{\mathcal{D}}(\mathbf{y})$ can be predetermined [7].

Observation 3. Based on this newly transformed $f(\mathbf{y})$ expression, it is easy to observe that terms ② and ③ can be locally calculated by \mathcal{C} while terms ④ and ⑤ by \mathcal{S} . This allows \mathcal{C} and \mathcal{S} to subsequently generate their shares of $f(\mathbf{y})$.

More specifically, \mathcal{S} pregenerates \mathbf{r} and $\hat{\mathbf{a}}_{\mathcal{S}}$, readily calculates term ⑤ as its share $\mathbf{a}_{\mathcal{S}}$ of $f(\mathbf{y})$, and sends $\hat{\mathbf{a}}_{\mathcal{S}}$ along with term ④ to \mathcal{C} in the offline phase¹³ then \mathcal{C} is able to calculate all but term ⑤ of $f(\mathbf{y})$, i.e., the share $\mathbf{a}_{\mathcal{C}}$ of $f(\mathbf{y})$. However, directly sending the above two terms raises the privacy issue. In order to protect privacy, \mathcal{S} encrypts them with its public key as $[\hat{\mathbf{a}}_{\mathcal{S}}]_{\mathcal{S}}$ and $[\mathbf{r} \ominus (2 \otimes \mathbf{r} \otimes \hat{\mathbf{a}}_{\mathcal{S}})]_{\mathcal{S}}$. As such, \mathcal{C} correspondingly gets

$$\begin{aligned} [\mathbf{a}_{\mathcal{C}}]_{\mathcal{S}} = & \underbrace{((\mathbf{y} \ominus \mathbf{r}) \otimes \hat{\mathbf{a}}_{\mathcal{C}})}_{\textcircled{2}} \oplus \underbrace{((\mathbf{y} \ominus \mathbf{r}) \otimes (1 \ominus (2 \otimes \hat{\mathbf{a}}_{\mathcal{C}})))}_{\textcircled{3}} \otimes [\hat{\mathbf{a}}_{\mathcal{S}}]_{\mathcal{S}} \\ & \oplus \underbrace{[\mathbf{r} \ominus (2 \otimes \mathbf{r} \otimes \hat{\mathbf{a}}_{\mathcal{S}})]_{\mathcal{S}}}_{\textcircled{4}} \otimes \hat{\mathbf{a}}_{\mathcal{C}}. \end{aligned}$$

Next, \mathcal{C} is supposed to send $[\mathbf{a}_{\mathcal{C}}]_{\mathcal{S}}$ to \mathcal{S} such that \mathcal{S} gets the nonlinear output $[f(\mathbf{y})]_{\mathcal{S}} = [\mathbf{a}_{\mathcal{C}}]_{\mathcal{S}} \oplus \mathbf{a}_{\mathcal{S}}$, which is the input for the next layer. However, $[\mathbf{a}_{\mathcal{C}}]_{\mathcal{S}}$ and $[f(\mathbf{y})]_{\mathcal{S}}$ are \mathcal{S} -encrypted, which can be decrypted by \mathcal{S} and thus $f(\mathbf{y})$ is exposed to \mathcal{S} . We resolve this problem by letting \mathcal{C} randomly generate another term \mathbf{g} and calculates its share for $f(\mathbf{y})$ as

$$\begin{aligned} [\mathbf{a}_{\mathcal{C}}]_{\mathcal{S}} = & \underbrace{(((\mathbf{y} \ominus \mathbf{r}) \otimes \hat{\mathbf{a}}_{\mathcal{C}}) \ominus \mathbf{g})}_{\textcircled{2}} \oplus \underbrace{((\mathbf{y} \ominus \mathbf{r}) \otimes (1 \ominus (2 \otimes \hat{\mathbf{a}}_{\mathcal{C}})))}_{\textcircled{3}} \otimes \\ & [\hat{\mathbf{a}}_{\mathcal{S}}]_{\mathcal{S}} \oplus \underbrace{[\mathbf{r} \ominus (2 \otimes \mathbf{r} \otimes \hat{\mathbf{a}}_{\mathcal{S}})]_{\mathcal{S}}}_{\textcircled{4}} \otimes \hat{\mathbf{a}}_{\mathcal{C}}, \end{aligned} \tag{29}$$

¹³The offline phase is independent of the input \mathbf{x} while the online phase is the process dependant of input \mathbf{x} .

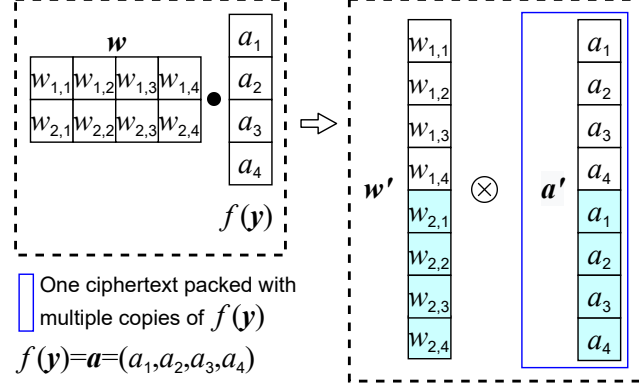


Figure 30. Data transformation for dot product.

where the term ② is changed from $((\mathbf{y} \ominus \mathbf{r}) \otimes \hat{\mathbf{a}}_{\mathcal{C}})$ to $((\mathbf{y} \ominus \mathbf{r}) \otimes \hat{\mathbf{a}}_{\mathcal{C}}) \ominus \mathbf{g}$. Meanwhile, \mathcal{C} encrypts \mathbf{g} with its public key as $[\mathbf{g}]_{\mathcal{C}}$, and sends it along with $[\mathbf{a}_{\mathcal{C}}]_{\mathcal{S}}$ to \mathcal{S} . In this way, \mathcal{S} decrypts $[\mathbf{a}_{\mathcal{C}}]_{\mathcal{S}}$ into $\mathbf{a}_{\mathcal{C}}$ and gets \mathcal{C} -encrypted nonlinear output as

$$\mathbf{a}_{\mathcal{S}} \oplus \mathbf{a}_{\mathcal{C}} \oplus [\mathbf{g}]_{\mathcal{C}} = [f(\mathbf{y})]_{\mathcal{C}}. \quad (30)$$

Here, as $\mathbf{a}_{\mathcal{C}}$ is masked with \mathbf{g} and $[\mathbf{g}]_{\mathcal{C}}$ is semantic secure to \mathcal{S} , \mathcal{S} only gets an encrypted $f(\mathbf{y})$ without other information. Note that the randomness of \mathbf{g} and \mathbf{r} is guaranteed as $\hat{\mathbf{a}}_{\mathcal{C}}$ and $\hat{\mathbf{a}}_{\mathcal{S}}$ are Boolean shares. The concrete security proof is given in Section 6.4. Meanwhile, \mathbf{g} can be pregenerated, encrypted by \mathcal{C} as $[\mathbf{g}]_{\mathcal{C}}$, and sent to \mathcal{S} in the *offline phase*, which introduces no HE cost for the encryption and communication in the online phase. As such, we make the whole process for nonlinear computation after $f_{\mathcal{D}}$ perm-free, and the communication is reduced to 0.5 round, i.e., \mathcal{C} sends $[\mathbf{a}_{\mathcal{C}}]_{\mathcal{S}}$ to \mathcal{S} , which is a sharp contrast to 4.5 rounds in CrypTFlow2 [7]. We will formally analyze the computation and communication costs in Section 6.3.3.

Data Transformation

While $[f(\mathbf{y})]_{\mathcal{C}}$ is obtained by Eq. (30), a simple but necessary data transformation is needed to ensure that \mathcal{S} can repeat the perm-free linear computation proposed in Section 6.3.1, i.e., \mathcal{S} multiplies $[f(\mathbf{y})]_{\mathcal{C}}$ with its associated kernel data (like \mathbf{k}'_i ($i \in \{1, 2, 3\}$) in Figure 29), forms the partial share (like $[\mathbf{y}' \ominus \mathbf{r}']_{\mathcal{C}}$ in Figure 29), and sends it back to \mathcal{C} to get the share of the linear result namely $\mathbf{w} \cdot f(\mathbf{y})$ in plaintext (like the way to get $(\mathbf{y} \ominus \mathbf{r})$ in Figure 29). As we transform the convolution into dot product, the dot product $\mathbf{w} \cdot f(\mathbf{y})$ is similarly processed as shown in Figure 30. Specifically, multiple copies of $f(\mathbf{y})$, i.e., \mathbf{a} in Figure 30, are packed in a single ciphertext, which forms a \mathcal{C} -encrypted ciphertext $[\mathbf{a}']_{\mathcal{C}}$. $[\mathbf{a}']_{\mathcal{C}}$ is sent to \mathcal{S} . \mathcal{S} flattens \mathbf{w} to \mathbf{w}' in row wise and multiplies it with $[\mathbf{a}']_{\mathcal{C}}$. It results in a ciphertext containing the partial sum of the linear result $\mathbf{w} \cdot f(\mathbf{y})$ which is in line with $[\mathbf{y}']_{\mathcal{C}}$ in convolution. Thus, \mathcal{S} similarly generates \mathbf{r}' , forms a partial share (like $[\mathbf{y}' \ominus \mathbf{r}']_{\mathcal{C}}$ in Figure 29), and sends it to \mathcal{C} . \mathcal{C} does the decryption and obtains the exact share of $\mathbf{w} \cdot f(\mathbf{y})$ with plaintext computation. Note that the random term \mathbf{r}' for the last layer is not applied or is filled with a single value since \mathcal{C} must get the prediction output.

Based on the above analysis, before Eq. (30) is performed, \mathcal{C} packs multiple copies of \mathbf{g} in one ciphertext to form a new $[\mathbf{g}]_{\mathcal{C}}$ and sends it to \mathcal{S} *in offline phase*. As $(\mathbf{a}_{\mathcal{S}} \oplus \mathbf{a}_{\mathcal{C}})$ is in plaintext at \mathcal{S} according to Eq. (30), \mathcal{S} simply forms the plaintext that has multiple copies of $(\mathbf{a}_{\mathcal{S}} \oplus \mathbf{a}_{\mathcal{C}})$, and then Eq. (30) finally outputs a new ciphertext $[f(\mathbf{y})]_{\mathcal{C}}$ that actually contains multiple copies of $f(\mathbf{y})$. Till now, \mathcal{S} is able to repeat the perm-free linear computation. Note that if the next linear computation is convolution, \mathbf{g} and $(\mathbf{a}_{\mathcal{S}} \oplus \mathbf{a}_{\mathcal{C}})$ in Eq. (30) are treated as \mathbf{x} (in convolution) and similar transformation is performed as shown in Figure 28, which enables \mathcal{S} to get the new $[f(\mathbf{y})]_{\mathcal{C}}$ that is similar to the linear input $[\mathbf{x}'_i]_{\mathcal{C}}$ ($i \in \{1, 2, 3\}$).

We now summarize the overall process for the proposed perm-free joint linear and non-linear computation to generate the input for the linear computation of the next layer. It includes offline and online phases. In the offline phase, \mathcal{S} pregenerates \mathbf{r} and $\hat{\mathbf{a}}_{\mathcal{S}}$ and gets two

ciphertext $[\mathbf{r} \ominus (\mathbf{2} \otimes \mathbf{r} \otimes \hat{\mathbf{a}}_{\mathcal{S}})]_{\mathcal{S}}$ and $[\hat{\mathbf{a}}_{\mathcal{S}}]_{\mathcal{S}}$, which are then sent to \mathcal{C} . Meanwhile, \mathcal{C} pregenerates \mathbf{g} , encrypts it into $[\mathbf{g}]_{\mathcal{C}}$, and sends it to \mathcal{S} . In the online phase, right after \mathcal{C} gets the share of $f_{\mathcal{D}}(\mathbf{y})$, i.e., $\hat{\mathbf{a}}_{\mathcal{C}}$, it calculates $[\mathbf{a}_{\mathcal{C}}]_{\mathcal{S}}$ as shown in Eq. (29) and sends $[\mathbf{a}_{\mathcal{C}}]_{\mathcal{S}}$ to \mathcal{S} . Upon receiving $[\mathbf{a}_{\mathcal{C}}]_{\mathcal{S}}$, \mathcal{S} decrypts it into $\mathbf{a}_{\mathcal{C}}$ and gets $[f(\mathbf{y})]_{\mathcal{C}}$ by Eq. (30). Here $[f(\mathbf{y})]_{\mathcal{C}}$ has been transformed to the data format according to Section 6.3.2, serving as the input for the next-layer linear computation. Note that the offline process in WISE is totally *non-interactive*, which does not need the involved parties to synchronously exchange the calculated data and then finish the process. Therefore, it eliminates the interactive offline computation that is often used in state-of-the-art frameworks such as DELPHI [66] and MiniONN [65].

6.3.3 COMPLEXITY ANALYSIS

In this section, we formally show that WISE is computationally advantageous to state-of-the-art perm-involved frameworks [7, 64, 66]. To ease our analysis, we reiterate some necessary variables as follows.

- c_i, c_o : number of input and output channels in the current convolution layer;
- f_h, f_w : height and width of kernel in the current convolution layer;
- f'_h, f'_w : height and width of kernel in the next convolution layer;
- c'_o : number of output channels in the next convolution layer;
- c_n : number of input channels that can be packed into one ciphertext¹⁴.

Computation Complexity

At the very beginning, as the convolution is transformed into dot product, e.g., the input \mathbf{x} is transformed into \mathbf{x}' in Figure 28, the size of the input to be packed by \mathcal{C} increases $f_h f_w$ times compared with the original. Thus, the number of resultant ciphertext transmitted from \mathcal{C} to \mathcal{S} is $f_h f_w c_i / c_n$. After \mathcal{S} receives the $f_h f_w c_i / c_n$ ciphertexts, it conducts $f_h f_w c_i / c_n$

¹⁴We uniformly analyze the convolution layers as the fully connected layer can also be transformed into convolution layer [94, 168]. And we always start with the calculation for the current layer.

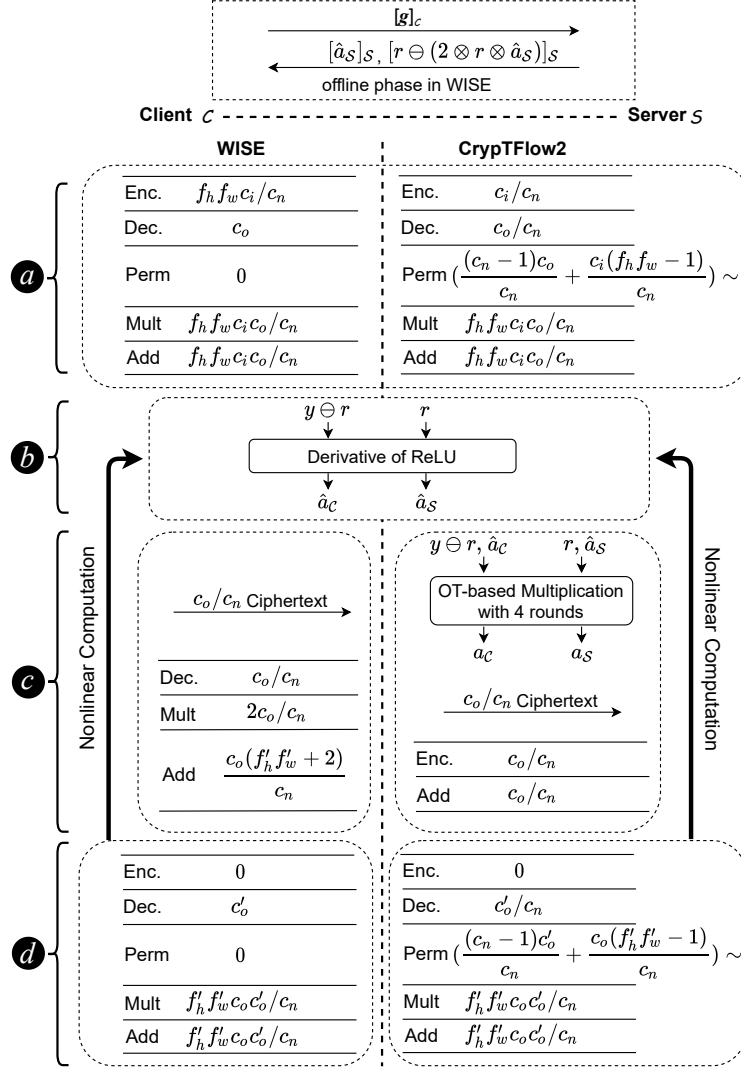


Figure 31. Comparison of layer-wise computation flow.

multiplications and $(f_h f_w c_i / c_n - 1)$ additions to get one intermediate ciphertext for one kernel (like $[y']_{\mathcal{S}}$ for kernel \mathbf{k} in Figure 29). As the number of output channels is c_o , i.e., there are c_o kernels, \mathcal{S} obtains c_o intermediate ciphertexts with a total of $f_h f_w c_i c_o / c_n$ multiplications and $(f_h f_w c_i / c_n - 1)c_o$ additions. Then another c_o additions are performed on the c_o intermediate ciphertexts to get c_o ciphertexts that have the partial share of the convolution (like $[y' \ominus r']_{\mathcal{C}}$ in Figure 29). Therefore, the total number of additions is $f_h f_w c_i c_o / c_n$. These c_o intermediate ciphertexts are sent back to \mathcal{C} , which does the decryption and obtains the share of convolution

in plaintext.

At this point, \mathcal{C} and \mathcal{S} finish the linear computation in the first layer, where \mathcal{C} has the linear share $\mathbf{y} \ominus \mathbf{r}$ and \mathcal{S} has the linear share \mathbf{r} . The comparison of computation complexity¹⁵ is summarized in block **a** of Figure 31. It is interesting to observe that WISE essentially converts $(c_n - 1)c_o/c_n$ permutations in CrypTFlow2 to decryption, and replaces $c_i(f_h f_w - 1)/c_n$ permutations in CrypTFlow2 by encryption. Since the cost for permutation is generally more expensive than decryption and encryption, this demonstrates the lighter computation complexity of WISE compared with CrypTFlow2.

Next, \mathcal{C} and \mathcal{S} collaborate to calculate the derivative of ReLU based on their respective linear shares, and the output is the corresponding Boolean share $\hat{\mathbf{a}}_{\mathcal{C}}$ and $\hat{\mathbf{a}}_{\mathcal{S}}$ as shown in block **b** of Figure 31. Note that the data size of $\hat{\mathbf{a}}_{\mathcal{C}}$ and $\hat{\mathbf{a}}_{\mathcal{S}}$ is c_o/c_n (in ciphertext). Then, \mathcal{C} computes the \mathcal{S} -encrypted nonlinear share $[\mathbf{a}_{\mathcal{C}}]_{\mathcal{S}}$ with $2c_o/c_n$ multiplications and $2c_o/c_n$ additions, as shown in Eq. (29). $[\mathbf{a}_{\mathcal{C}}]_{\mathcal{S}}$ (with c_o/c_n ciphertext) is sent to \mathcal{S} . Upon receiving $[\mathbf{a}_{\mathcal{C}}]_{\mathcal{S}}$, \mathcal{S} performs c_o/c_n decryption to obtain $\mathbf{a}_{\mathcal{C}}$ and gets $[f(\mathbf{y})]_{\mathcal{C}}$ with $c_o f'_h f'_w / c_n$ additions according to Eq. (30). This is because $(\mathbf{a}_{\mathcal{S}} \oplus \mathbf{a}_{\mathcal{C}})$ and \mathbf{g} in Eq. (30) is transformed in a way similar to transforming \mathbf{x} into \mathbf{x}' in Figure 28, which forms the input for linear computation of the next layer.

The comparison of the computation process to get the linear input of the next layer is shown in block **c** of Figure 31. As decryption is generally faster than encryption [133], WISE's c_o/c_n decryption is more advantageous than CrypTFlow2's c_o/c_n encryption. Meanwhile, WISE has extra $c_o(f'_h f'_w + 1)/c_n$ additions (as another c_o/c_n additions offset with c_o/c_n additions in CrypTFlow2) in block **c**, which only introduce a light cost as addition is the cheapest operation for HE, e.g., within microseconds per operation in our experiments. We also show in Section 6.5 that these additions are negligible. Furthermore, we will show next that WISE's remaining $2c_o/c_n$ multiplications are also cheaper by combining with the linear

¹⁵Note that we use the lower bound of permutation cost for the HE-based linear computation among the state-of-the-art frameworks [7, 64, 66, 102] to show WISE's fundamental computation strength.

computation for the next layer, as shown in block **a** of Figure 31.

Note that \mathcal{S} obtains $[f(\mathbf{y})]_{\mathcal{C}}$ with $c_o f'_h f'_w / c_n$ ciphertexts at the end of block **c** then in block **a**, \mathcal{S} respectively conducts $f'_h f'_w c_o c'_o / c_n$ multiplications and additions to get c'_o ciphertexts that have the partial share of the convolution (like $[\mathbf{y}' \ominus \mathbf{r}']_{\mathcal{C}}$ in Figure 29). These c'_o ciphertexts are sent back to \mathcal{C} , which performs c'_o decryptions and gets the share for the linear computation in the next layer. In contrast, the corresponding computation of CrypTFlow2 requires $(c_n - 1)c'_o / c_n + c_o(f'_h f'_w - 1) / c_n$ permutations. While converting CrypTFlow2's $(c_n - 1)c'_o / c_n$ permutations to an equal number of decryption results in WISE's decryption complexity, the other $c_o(f'_h f'_w - 1) / c_n$ permutations in CrypTFlow2 are replaced by WISE's $2c_o / c_n$ multiplications in block **c**. On the one hand, the permutation is more expensive than decryption and multiplication. On the other hand, $(f'_h f'_w - 1)$ is larger than 2 as long as the kernel size, i.e., the kernel width f'_w and height f'_h , is larger than 1, which usually happens in many state-of-the-art DL networks such as AlexNet and VGG. As such, the overall computation cost of WISE is systematically reduced.

The process repeats from block **b** to block **d** until the last layer. We will show experimental data in Section 6.5 that WISE is able to speed up CrypTFlow2 over various practical DL networks.

Communication Complexity

Since block **a** is for the first layer only where the communication is similar to block **d**, and block **b** involves the same communication cost, our analysis focuses on blocks **c** and **d**.

In block **c**, WISE has lower communication cost. It requires 0.5 communication round while CrypTFlow2 needs 4.5 rounds. Overall, WISE transmits c_o / c_n ciphertexts while CrypTFlow2 needs to transmit not only c_o / c_n ciphertexts but also extra plaintext data of $2n(\lambda + 2\eta)$ bits for the OT-based multiplication, where n is the number of elements in \mathbf{y} (e.g., 4 in our example DL model), λ is the security parameter, and η is the logarithm of plaintext modulus.

In block **d**, both WISE and CrypTFlow2 need 0.5 round to transmit data from \mathcal{S} to \mathcal{C} . The amount of data transmitted in WISE is c_n times larger than that of CrypTFlow2. Note that, however, this overhead can be mitigated by adequate bandwidth, which is a relatively low cost in today's transmission link. As to be shown in Section 6.5, WISE has noticeable overall performance gain (due to the reduced computation cost as analyzed above and the lower communication cost in block **c**) to offset the communication cost in block **d**.

6.4 SECURITY ANALYSIS

The proposed WISE protocol consists of four blocks, i.e., blocks **a**, **b**, **c**, and **d**, as shown in Figure 31. We show the security of blocks **a**, **b**, and the pairing processing of (block **c**, block **d**) where the output of each block/processing is randomly shared between \mathcal{C} and \mathcal{S} . As stated in Lemma 2 of [169], a protocol that ends with secure re-sharing of output is universally composable. Thus, the overall WISE is secure after compositing blocks **a**, **b**, and the pairing processing of (block **c**, block **d**).

Specifically, Our security proof follows the ideal-world/real-world paradigm [150]: in the real world, both \mathcal{C} and \mathcal{S} interact with each other according to the protocol specification while in the ideal world, parties have access to a Trusted Third Party (TTP) that implements each block/processing. The executions in both worlds are coordinated by the environment ϵ who chooses the inputs to the parties and plays the role of a distinguisher between the real and ideal executions. We aim to show that the view of a semi-honest adversary in the real world is indistinguishable to that in the ideal world.

Theorem 1. *The protocol in block **a** is secure in the presence of semi-honest adversaries, if the underlying HE scheme is semantically secure.*

Proof. We first prove the security against a semi-honest server by constructing an ideal-world simulator **Sim** that performs as follows:

- (1) receives \mathbf{k} (model parameters) from ϵ , and sends it to TTP;

- (2) starts running \mathcal{S} on input \mathbf{k} ;
- (3) constructs $[\tilde{\mathbf{x}}'_i]_{\mathbf{pk}'} \leftarrow \text{HE.Enc}(\mathbf{pk}', \mathbf{0})$ where \mathbf{pk}' is randomly generated by Sim via HE.KeyGen ;
- (4) sends $[\tilde{\mathbf{x}}'_i]_{\mathbf{pk}'}$ to \mathcal{S} ;
- (5) outputs whatever \mathcal{S} outputs.

Here \mathcal{S} 's view in real execution is $[\mathbf{x}'_i]_{\mathcal{C}}$ (data encrypted by the client), which is computationally indistinguishable from its view in the ideal execution i.e., $[\tilde{\mathbf{x}}'_i]_{\mathbf{pk}'}$, due to the semantic security of HE.Enc . Therefore, the output distribution of ϵ in the real world is computationally indistinguishable from that in the ideal world.

Next, we prove security against a semi-honest client by constructing an ideal-world simulator Sim that works as follows:

- (1) receives \mathbf{x} (\mathcal{C} 's private data) from ϵ ; Sim sends it to TTP and gets the result $(\mathbf{y} \ominus \mathbf{r})$;
- (2) starts running \mathcal{C} on input \mathbf{x} , and receives $[\mathbf{x}'_i]_{\mathcal{C}}$ from \mathcal{C} ;
- (3) randomly splits $(\mathbf{y} \ominus \mathbf{r})$ into a vector $\tilde{\mathbf{y}}$ such that it has the same dimension as $(\mathbf{y}' \ominus \mathbf{r}')$, and $(\mathbf{y} \ominus \mathbf{r})$ can be obtained from $\tilde{\mathbf{y}}$ by rotation and addition calculations (in plaintext), which is similar to compute $(\mathbf{y} \ominus \mathbf{r})$ from $(\mathbf{y}' \ominus \mathbf{r}')$ as explained in Section 6.3.1;
- (4) encrypts $\tilde{\mathbf{y}}$ into $[\tilde{\mathbf{y}}]_{\mathcal{C}}$ using \mathcal{C} 's public key and returns $[\tilde{\mathbf{y}}]_{\mathcal{C}}$ to \mathcal{C} ;
- (5) outputs whatever \mathcal{C} outputs.

Here \mathcal{C} 's view in the real execution is $(\mathbf{y}' \ominus \mathbf{r}')$ while its view in the ideal execution is $\tilde{\mathbf{y}}$. We only need to show that any element in $(\mathbf{y}' \ominus \mathbf{r}')$ is indistinguishable from a random number in $\tilde{\mathbf{y}}$. This is clearly true since $(\mathbf{y} \ominus \mathbf{r})$ is randomly split to form $\tilde{\mathbf{y}}$. At the end of the simulation, \mathcal{C} outputs $(\mathbf{y} \ominus \mathbf{r})$, which is the same as real execution. Thus, we claim that the output distribution of ϵ in the real world is computationally indistinguishable from that in the ideal world, completing the proof.

Theorem 2. *The protocol in block 6 is secure in the presence of semi-honest adversaries,*

if the underlying OT scheme is secure.

Proof. The security of block **b** has been proven in [7].

Theorem 3. *The protocol in the pairing processing of (block **c**, block **d**) is secure in the presence of semi-honest adversaries, if the underlying HE scheme is secure.*

Proof. We first prove security against a semi-honest server by constructing an ideal-world simulator **Sim** that performs as follows:

- (1) receives \mathbf{k}, \mathbf{r} and $\hat{\mathbf{a}}_{\mathcal{S}}$ from ϵ , and sends it to TTP;
- (2) starts running \mathcal{S} on input \mathbf{k}, \mathbf{r} and $\hat{\mathbf{a}}_{\mathcal{S}}$;
- (3) constructs $[\tilde{\mathbf{g}}]_{\mathbf{pk}'} \leftarrow \text{HE.Enc}(\mathbf{pk}', \mathbf{0})$ where \mathbf{pk}' is randomly generated by **Sim** via HE.KeyGen ;
- (4) randomly forms a vector $\tilde{\mathbf{a}}_{\mathcal{C}}$ and encrypts it into $[\tilde{\mathbf{a}}_{\mathcal{C}}]_{\mathcal{S}}$ using \mathcal{S} 's public key;
- (5) sends $[\tilde{\mathbf{a}}_{\mathcal{C}}]_{\mathcal{S}}$ and $[\tilde{\mathbf{g}}]_{\mathbf{pk}'}$ to \mathcal{S} ;
- (6) outputs whatever \mathcal{S} outputs.

Here \mathcal{S} 's view in the real execution is $\mathbf{a}_{\mathcal{C}}$ and $[\mathbf{g}]_{\mathcal{C}}$ while its view in the ideal execution is $\tilde{\mathbf{a}}_{\mathcal{C}}$ and $[\tilde{\mathbf{g}}]_{\mathbf{pk}'}$. On the one hand, any element in $\mathbf{a}_{\mathcal{C}}$ is indistinguishable from a random number in $\tilde{\mathbf{a}}_{\mathcal{C}}$. On the other hand, $[\mathbf{g}]_{\mathcal{C}}$ and $[\tilde{\mathbf{g}}]_{\mathbf{pk}'}$ are indistinguishable due to the semantic security of HE.Enc . Therefore, the output distribution of ϵ in the real world is computationally indistinguishable from that in the ideal world.

Next, we prove security against a semi-honest client by constructing an ideal-world simulator **Sim** that works as follows:

- (1) receives $(\mathbf{y} \ominus \mathbf{r})$ and $\hat{\mathbf{a}}_{\mathcal{C}}$ from ϵ ; **Sim** sends it to TTP and gets the result $(\bar{\mathbf{y}} \ominus \bar{\mathbf{r}})$, i.e., the share of linear transformation in the next layer;
- (2) starts running \mathcal{C} on input $(\mathbf{y} \ominus \mathbf{r})$ and $\hat{\mathbf{a}}_{\mathcal{C}}$;
- (3) constructs $[\tilde{\mathbf{r}}_1]_{\mathbf{pk}'} \leftarrow \text{HE.Enc}(\mathbf{pk}', \mathbf{0})$ and $[\tilde{\mathbf{r}}_2]_{\mathbf{pk}'} \leftarrow \text{HE.Enc}(\mathbf{pk}', \mathbf{0})$ where \mathbf{pk}' is randomly generated by **Sim** via HE.KeyGen ;
- (4) sends $[\tilde{\mathbf{r}}_1]_{\mathbf{pk}'}$ and $[\tilde{\mathbf{r}}_2]_{\mathbf{pk}'}$ to \mathcal{C} , and receives $[\mathbf{a}_{\mathcal{C}}]_{\mathbf{pk}'}$ and $[\mathbf{g}]_{\mathcal{C}}$ from \mathcal{C} ;

Input Dimension $H \times W@_{c_i}$	Kernel Dimension $f_h \times f_w@_{c_o}$	Stride & Padding	Framework	Time (ms)		Speedup		Comm. (MB)
				LAN	WAN	LAN	WAN	
$14 \times 14@6$	$5 \times 5@16$	(1, 0)	WISE	82	353	5 \times	2 \times	4.39
			CrypTFlow2	414	713	-	-	2.2
$2 \times 2@512$	$3 \times 3@512$	(1, 1)	WISE	1990	2231	2.5 \times	2.4 \times	97
			CrypTFlow2	5173	5367	-	-	3
$4 \times 4@256$	$3 \times 3@512$	(2, 1)	WISE	1943	2173	13 \times	11 \times	98
			CrypTFlow2	25566	25851	-	-	5.52
$2 \times 2@512$	$1 \times 1@512$	(1, 0)	WISE	1886	2120	5.9 \times	5.3 \times	98
			CrypTFlow2	11174	11385	-	-	2.6
$32 \times 32@3$	$11 \times 11@96$	(4, 5)	WISE	635	899	12.3 \times	9.1 \times	24
			CrypTFlow2	7866	8218	-	-	16

Table 26. Running time and communication cost of convolution layers.

(5) randomly splits $(\bar{\mathbf{y}} \ominus \bar{\mathbf{r}})$ into a vector $\tilde{\mathbf{y}}$ such that it has the same dimension as $(\mathbf{y}' \ominus \mathbf{r}')$ (for the next layer), and $(\bar{\mathbf{y}} \ominus \bar{\mathbf{r}})$ can be obtained from $\tilde{\mathbf{y}}$ by rotation and addition calculations (in plaintext), which is similar to compute $(\mathbf{y} \ominus \mathbf{r})$ from $(\mathbf{y}' \ominus \mathbf{r}')$ as explained in Section 6.3.1;

(6) encrypts $\tilde{\mathbf{y}}$ into $[\tilde{\mathbf{y}}]_{\mathcal{C}}$ using \mathcal{C} 's public key and returns $[\tilde{\mathbf{y}}]_{\mathcal{C}}$ to \mathcal{C} ;

(7) outputs whatever \mathcal{C} outputs.

Here \mathcal{C} 's view in the real execution is $[\hat{\mathbf{a}}_{\mathcal{S}}]_{\mathcal{S}}$, $[\mathbf{r} \ominus (\mathbf{2} \otimes \mathbf{r} \otimes \hat{\mathbf{a}}_{\mathcal{S}})]_{\mathcal{S}}$ and $(\mathbf{y}' \ominus \mathbf{r}')$ (for the next layer) while its view in the ideal execution is $[\tilde{\mathbf{r}}_1]_{\text{pk}'}$, $[\tilde{\mathbf{r}}_2]_{\text{pk}'}$ and $\tilde{\mathbf{y}}$. First, any element in $(\mathbf{y}' \ominus \mathbf{r}')$ is indistinguishable from a random number in $\tilde{\mathbf{y}}$ since $(\bar{\mathbf{y}} \ominus \bar{\mathbf{r}})$ is randomly split to form $\tilde{\mathbf{y}}$. Second, $[\hat{\mathbf{a}}_{\mathcal{S}}]_{\mathcal{S}}$ and $[\mathbf{r} \ominus (\mathbf{2} \otimes \mathbf{r} \otimes \hat{\mathbf{a}}_{\mathcal{S}})]_{\mathcal{S}}$ are indistinguishable from $[\tilde{\mathbf{r}}_1]_{\text{pk}'}$ and $[\tilde{\mathbf{r}}_2]_{\text{pk}'}$ due to the semantic security of HE.Enc . At the end of the simulation, \mathcal{C} outputs $(\bar{\mathbf{y}} \ominus \bar{\mathbf{r}})$, which is the same as real execution. Thus, we claim that the output distribution of ϵ in the real world is computationally indistinguishable from that in the ideal world, completing the proof.

6.5 PERFORMANCE EVALUATION

In this section, we present the performance evaluation and experimental results. We first introduce the experimental setup in Section 6.5.1, and then discuss the following two questions in Sections 6.5.2 and 6.5.3, respectively:

- How efficient is WISE to speed up the layer-wise computation?

Dataset Input Dim. ($H \times W@c_i$)	DL Architecture	Framework	Time (ms)		Speedup		Comm. (MB)	
			LAN	WAN	LAN	WAN	Online	Offline
MNIST ($28 \times 28@1$)	LeNet	WISE	977	1845.5	5.3 \times	3.3 \times	52.5	8
		CrypTFlow2	5243	6221.4	-	-	10.7	-
CIFAR10 ($32 \times 32@3$)	AlexNet	WISE	35921	37688	2 \times	2 \times	1827	41
		CrypTFlow2	75196	76928	-	-	44.2	-
	VGG-11	WISE	44366	47129	1.97 \times	1.9 \times	2195.3	107.8
		CrypTFlow2	87464	90359	-	-	168.4	-
	VGG-13	WISE	47518	50749	1.95 \times	1.89 \times	2310.2	157.3
		CrypTFlow2	92429	95842	-	-	269.5	-
	VGG-16	WISE	53499	57619	1.94 \times	1.88 \times	2571.9	171
		CrypTFlow2	103577	107983	-	-	299.3	-
	VGG-19	WISE	59480	64489	1.93 \times	1.86 \times	2833.6	184.7
		CrypTFlow2	114725	120124	-	-	329.1	-
	ResNet-18	WISE	26551	32896	3.63 \times	3.18 \times	1306.1	122.4
		CrypTFlow2	96175.6	104308	-	-	267.7	-
	ResNet-34	WISE	50360	63146	2.94 \times	2.6 \times	2479.7	190.6
		CrypTFlow2	147740.6	163527	-	-	441.8	-

Table 27. Running time and communication cost on modern DL models.

- What is the prediction latency and communication cost on practical DL models using WISE compared with the state-of-the-art frameworks such as CrypTFlow2 [7]¹⁶?

6.5.1 EXPERIMENTAL SETUP

We ran all experiments on two Amazon AWS **c4.xlarge** instances possessing the Intel(R) Xeon(R) CPU E5-2666 v3 @ 2.90GHz, with 7.5GB of system memory. In the LAN setting, the client \mathcal{C} and server \mathcal{S} were executed on such two instances both located in the **us-east-1d** (Northern Virginia) availability zone. In the WAN setting, \mathcal{C} and \mathcal{S} were executed on such two instances respectively located in the **us-east-1d** (Northern Virginia) and **us-east-2c** (Ohio) availability zone. \mathcal{C} and \mathcal{S} each used an 4-thread execution. These experiential settings are similar with those used for the evaluation of the state-of-the-art frameworks [64, 66]. Furthermore, the evaluation used the following datasets and architectures:

1. **MNIST** [140], a basic dataset for the image classification task. It contains a set of 28×28 grayscale images of handwritten digits from 0 to 9. Given such an image as

¹⁶CrypTFlow2 is currently one of the leading frameworks for privacy-preserving DL and it shows significant speedup than other works such as GAZELLE [64] (USENIX Security’18) and DELPHI [66] (USENIX Security’20).

one input, we aim to correctly predict the handwritten digit it represents, in a privacy-preserving manner. We evaluate this task using the classic **LeNet** network [167].

2. **CIFAR10** [170], another commonly used dataset for image classification benchmark that is substantially more complicated than the MNIST-based classification task. The dataset includes a set of 32×32 RGB images for 10 classes such as automobiles, birds, cats, etc. For the privacy-preserving classification task with CIFAR10 dataset, we adopt various DL topologies that are used in practice: **AlexNet** [15], **VGG-11** [16], **VGG-13** [16], **VGG-16** [16], **VGG-19** [16], **ResNet-18** [26], and **ResNet-34** [26].

When we compare WISE with CrypTFlow2¹⁷, we estimate the cost of WISE’s and CrypTFlow2’s protocols by summing the costs of the relevant subprotocols for all linear and non-linear transformations (including all computation and communication costs) as shown in all blocks of Figure 31. We do this as a protocol decomposition with all costs counted, and this methodology is also used in other privacy-preserving frameworks.

6.5.2 MICROBENCHMARKS

We first benchmark the performance of the convolution layer¹⁸, which is the basic building block in state-of-the-art DL architectures. Note that each convolution layer includes both linear and nonlinear transformations. In Table 26, we evaluate the cost of various convolution layers used in **LeNet**, **AlexNet**, **VGG-11**, **VGG-13**, **VGG-16**, **VGG-19**, **ResNet-18**, and **ResNet-34**. If the convolution layer is the first layer of the relevant network, the cost includes all computation and communication overhead from block **a** to block **c** of Figure 31, which is the time duration starting from \mathcal{C} transforms and encrypts its input, until \mathcal{S} obtains the input for linear computation of the next layer. If the convolution layer is not the first layer, the cost includes all computation and communication overhead from block **d** back

¹⁷Code available at <https://github.com/mpc-msri/EzPC>.

¹⁸Note that the dense layer can also be transformed into convolution layer [94, 168].

to block **c** of Figure 31, which is the duration between the time when \mathcal{S} performs the linear computation for the current layer to the time when \mathcal{S} obtains the input for the linear computation of the next layer.

The key takeaway from Table 26 is that our online time is about $2\times$ to $13\times$ smaller than CrypTFlow2’s. Note that our online communication cost is higher than CrypTFlow2’s, e.g., WISE needs 98MB while CrypTFlow2 involves 5.5MB. However, WISE has noticeable overall performance gain (due to the reduced computation cost as analyzed in Section 6.3.3 and the lower communication cost in block **c** of Figure 31) to offset the communication cost.


6.5.3 PERFORMANCE ON MODERN DL MODELS

We test the performance of WISE on various DL models used in practice. The overall evaluation is shown in Table 27. Specifically, in the LAN setting, WISE has a speedup of $5.3\times$, $2\times$, $1.97\times$, $1.95\times$, $1.94\times$, $1.93\times$, $3.63\times$, $2.94\times$ over CrypTFlow2 on LeNet, AlexNet, VGG-11, VGG-13, VGG-16, VGG-19, ResNet-18, and ResNet-34. The speedup is respectively $3.3\times$, $2\times$, $1.9\times$, $1.89\times$, $1.88\times$, $1.86\times$, $3.18\times$, $2.6\times$ in the WAN setting. As for the communication cost, the online overhead includes all transmitted data in blocks **a**, **b**, **c**, and **d** of Figure 31, while the offline overhead includes all transmitted data in the offline phase as shown in Figure 31. Here we can see that WISE has a larger data load to be transmitted in the online phase and the transmission overhead can be over $10\times$ compared with CrypTFlow2. As the bandwidth is a relatively low cost in today’s transmission link, e.g., Amazon AWS can easily keep a bandwidth around one Gigabit, the reduction of computation and communication round in WISE brings an overall performance boost that offsets the transmission overhead. Meanwhile, the communication overhead of WISE in the offline phase is lighter and comparable to that of CrypTFlow2 (in the online phase), which has proved to be communication-reduced for the involved parties [7]. Furthermore, it is worth reiterating that WISE’s offline phase is *totally non-interactive* and does not need the involved parties to synchronously exchange any calculated data. Therefore, it eliminates the interactive offline

computation that is often used in the state-of-the-art frameworks such as DELPHI [66] and MiniONN [65].

Next we test the runtime breakdown of each layer in our evaluated eight DL models as shown in Fig. 32. It allows us to have detailed observations. Specifically, the runtime for each layer includes all overhead for linear and nonlinear transformations, which is similar to the one examined in Section 6.5.2. Meanwhile, the layer index also includes the pooling layer, i.e., mean pooling. In **LeNet**, WISE has noticeable speedup in each convolution layer and the speedup is larger at the last layer as WISE only needs one HE multiplication while CrypTFlow2 involves a series of HE permutations. Similar observations are found in the last layer of other networks. In **AlexNet**, the large kernel width and height in the first layer, i.e., $f_h = f_w = 11$, result in more permutations needed in CrypTFlow2 while the counterpart in WISE is the same amount of multiplications, which is more efficient (see more details in Section 6.3.3). At the same time, the stride of 4 in the first layer involves 16 non-stride convolutions in CrypTFlow2 while WISE benefits from the decreased data size to be transmitted and from the smaller computation overhead for strided convolution. As such, WISE gains a larger speedup. In **VGG-11**, **VGG-13**, **VGG-16** and **VGG-19**, the speedup is larger in layers 11, 13, 15 and 17 (except the last layer) respectively. This is because large c_o (c'_o in block **d** of Figure 31), i.e., the number of output channels, makes the gap between permutation and decryption more significant. As decryption is generally cheaper than multiplication, the speedup correspondingly increases. In **ResNet-18** and **ResNet-34**, WISE’s speedup is larger in layers 9, 15, 21 (except the last layer) with strided convolution, which is in line with the speedup for the first layer in **AlexNet**. Similar observations are found in the WAN setting.

Finally, we show in Figure 33 the breakdown of communication overhead in each layer in our evaluated eight DL models. Here the layer index doesn’t include the pooling layer as no communication cost is involved in pooling. As demonstrated in Figure 31, the gap in communication cost (i.e., the amount of data to be transmitted) between WISE and

CrypTFlow2 is proportional to the number of output channels, i.e., c_o . As such, we can see an increased difference between their communication costs as c_o increases along the layers in all networks. As we discussed earlier, despite the larger amount of data for communication, WISE has noticeable overall performance gain (due to the reduced computation cost as analyzed in Section 6.3.3 and the lower communication cost in block ) to offset the increased communication costs.

6.6 CHAPTER SUMMARY

In this chapter, we have jointly considered the computation of two consecutive layers to optimize system efficiency. Specifically, we have proposed WISE, a novel hybrid protocol that features (1) a permutation-free scheme which completely eliminates the most expensive ciphertext permutation operations in the linear transformation and (2) a joint permutation-free computation between the nonlinear transformation in the current layer and the linear transformation in the next layer, which reduces communication cost from 4.5 rounds to only a half round. As such, WISE has achieved about $2\times$ to $13\times$ speedup over CrypTFlow2 (ACM CCS'20) for various neural layers used in the state-of-the-art DL architectures and demonstrates a speedup up to $5.3\times$ on practical DL models.

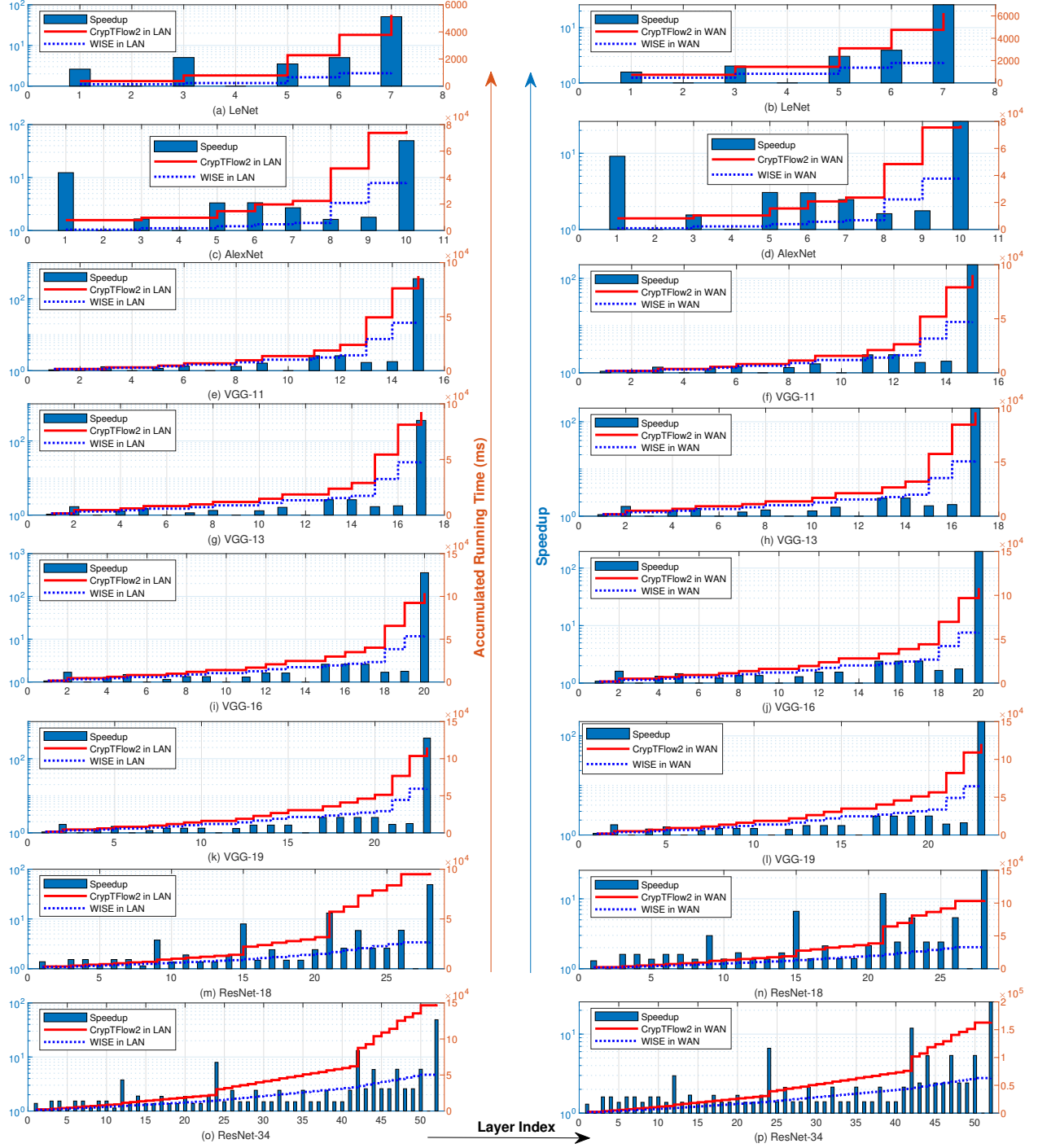


Figure 32. Layer-wise accumulated running time and WISE speedup over CryptFlow2 on different networks: (a) and (b) LeNet; (c) and (d) AlexNet; (e) and (f) VGG-11; (g) and (h) VGG-13; (i) and (j) VGG-16; (k) and (l) VGG-19; (m) and (n) ResNet-18; (o) and (p) ResNet-34. The bar with values on the left y-axis indicates speedup in log scale, and the curve with values on the right y-axis indicates the accumulated running time. The layers with speedup of 1 are pooling layers.

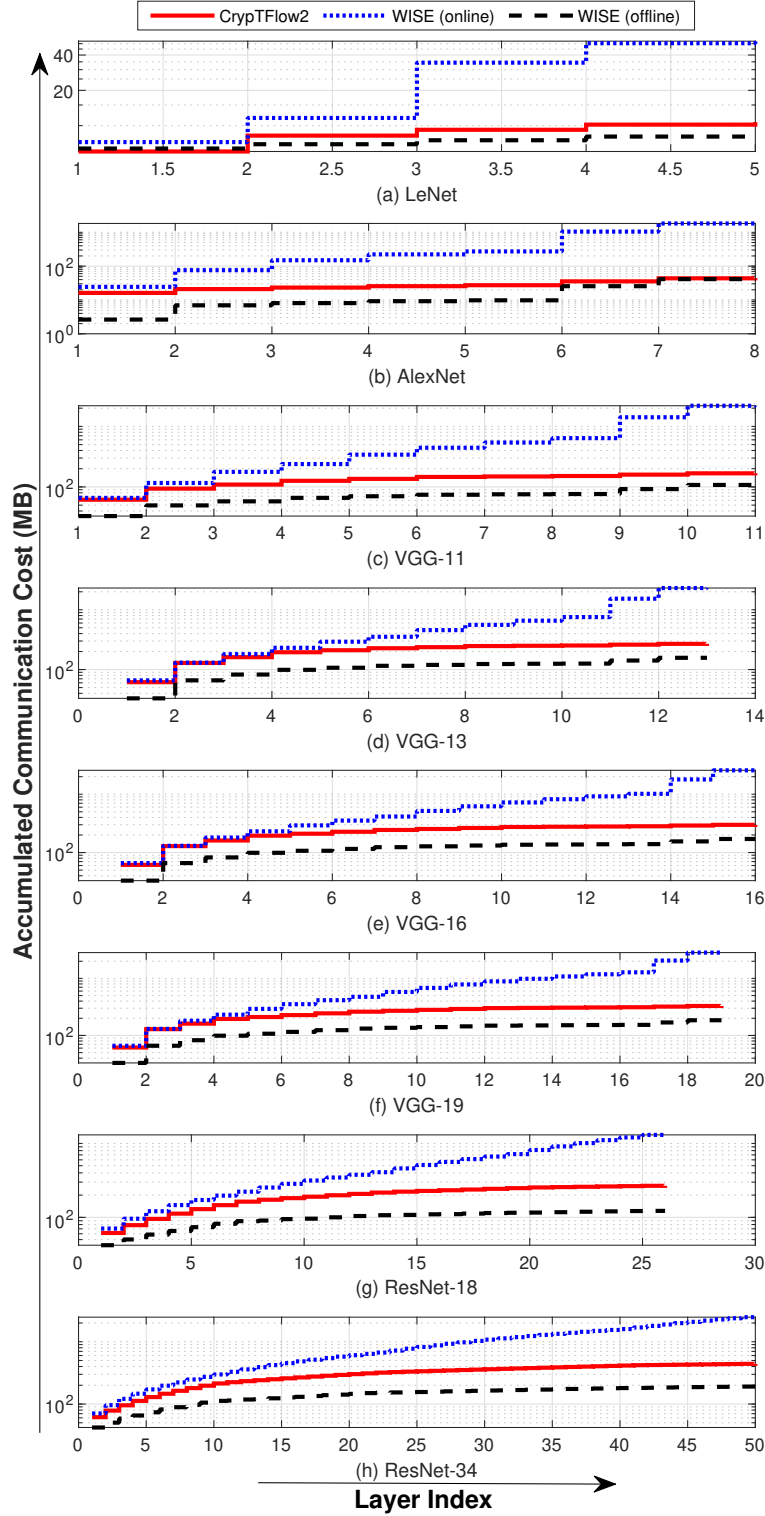


Figure 33. Layer-wise accumulated communication cost (in log scale) on different networks: (a) LeNet; (b) AlexNet; (c) VGG-11; (d) VGG-13; (e) VGG-16; (f) VGG-19; (g) ResNet-18; (h) ResNet-34.

CHAPTER 7

CONCLUSIONS AND FUTURE WORKS

In order to shorten the gap to practical usability and mitigate the efficiency-accuracy tradeoff in PPD, this dissertation has deeply optimized state-of-the-art frameworks and designed efficient modules by joint linear and nonlinear computation, with data encryption, to boost the overall performance. The four contributions that have been made in the dissertation are listed below.

- First, a deep optimization for the HE-based linear computation in HE-GC-based privacy-preserving DL inference, GALA [102], has been presented that features a row-wise weight matrix encoding, a combination of the share generation and a first-Add-second-Perm approach to reduce the most expensive permutation operations. GALA has demonstrated an inference runtime boost by $2.5\times$ to $8.3\times$ over state-of-the-art frameworks.
- Second, the GC-based nonlinear calculation has been replaced with a newly-designed joint linear and non-linear computation for each DL layer [103], in HE-GC-based privacy-preserving DL inference, based on the Homomorphic Secret Sharing. This construction has achieved an inference speedup as high as $48\times$ compared with state-of-the-art frameworks.
- Third, the nonlinear calculation of each layer has been completed for free by a carefully partitioned DL framework, GELU-Net [90], where the server performs linear computation on encrypted data utilizing a less complex homomorphic cryptosystem, while the client securely executes non-polynomial computation in plaintext without approximation. GELU-Net has demonstrated $14\times$ to $35\times$ inference speedup compared to the classic systems.

- Finally, we have proposed the WISE framework to completely eliminate the most expensive HE permutation in HE-based linear calculation and reduce the communication cost from 4.5 rounds to only a half round, by a joint permutation-free computation between the nonlinear transformation in the current layer and the linear transformation in the next layer. WISE has achieved $2\times$ to $13\times$ speedup over one of the most recent works through various widely-adopted neural layers, and demonstrates a speedup up to $5.3\times$ on practical DL models.

It is worth pointing out that even with significant progress toward privacy-preserving machine learning in recent years (including the work in this dissertation), there still exists a large performance gap between the plaintext system (generally below a second) and the privacy-preserving system (ranging from seconds to hundreds of seconds). Nevertheless, it is still promising to attain the long-term goal for practical implementation of privacy-preserving machine learning. First, the privacy-preserving machine learning system is to be deployed on clouds with abundant computation power. Hence, even though it takes significantly more time than the plaintext system on the same local hardware, running it on clouds with parallel computing infrastructure can significantly reduce the gap. Second, the research on the in-depth optimization of the privacy-preserving computation further help to close the runtime gap. Altogether, the combination of advanced algorithms and cloud computation resources may enable the privacy-preserving system to achieve a response time well suited for some practical applications in the near future.

Specifically, we suggest several promising directions to possibly shorten the gap.

1) *Conversion among shares in hybrid-primitive schemes:* Generally, different properties of linear and nonlinear computation make hybrid-primitive schemes outperform single-primitive counterparts. In hybrid-primitive schemes, it is inevitable to conduct particular conversion among different data types (e.g., arithmetic shares, Boolean shares and Yao’s GC shares), which are resulted from different primitives. The conversion is an important factor

that affects the overall system cost, especially for large-size input and models. While the hybrid-primitive computation is involving and being optimized [162, 164], adopting single-primitive methodology can circumvent the share conversion [91]. However, it may also incur challenges for efficient and accurate computation, i.e., linear and nonlinear computation, as linear functions are suitable for arithmetic values while the nonlinear functions are suitable for Boolean values. Thus, designing more efficient and accurate modules with a single primitive for both linear and nonlinear functions may boost overall system performance.

2) *Reconsidering the linear-and-nonlinear logic for privacy-preserving DL:* The fundamental working flow in DL is the repetition of linear and nonlinear computation, which is also a golden rule in privacy-preserving DL. Specifically, all privacy-preserving schemes come up with designs to first tackle the linear computation, and then solve the computation of nonlinear functions by taking the linear output as nonlinear input. As shown in [91], this seemingly logical rule may hinder improvement for the overall system. For example, given the input to one layer in the DL model, the intermediate data (i.e., the output of linear function) is calculated for the final output (i.e., the nonlinear result). Obviously, each layer does not necessarily need that specific intermediate data, i.e., the linear output, if the nonlinear output can be obtained by efficiently calculating another intermediate data. While it is interesting, the construction of that intermediate data remains challenging and needs more insights.

3) *Parallel computing based hardware-software codesign for larger and deeper networks:* State-of-the-art DL models have massive layers, e.g., over one hundred layers [26], and large-size input, e.g., three-channel images with 2-D size of 227×227 [171]. Besides optimization for computation algorithms, how to pipeline such large networks with large-volume input remains another hurdle for privacy-preserving DL, as privacy-preserving primitives, e.g., HE and GC, may process data by big modulus which are not efficiently fit for current parallel computing techniques. The batching technique that computes privacy-preserving data, e.g., encrypted input, in parallel is mostly used, e.g., SIMD for HE [64, 153] and circuit pipeline

for GC [76]. Meanwhile, several protocols for the linear and nonlinear functions are recently proposed [68, 172] that involve a large amount of plaintext computation, e.g., computation for arithmetic numbers to benefit more from the current parallel computing techniques.

On the other hand, there are some emerging schemes that involve specifically designed hardware to accelerate the primitive, e.g., speed up the Number Theoretic Transform (NTT) for HE, and thus improve system efficiency [173–176]. Overall, the designs for algorithmic parallelism and hardware acceleration are still relatively disjoint, and the performance may be further improved by hardware-software codesign towards better pipelining for encrypted data.

4) *Network architecture selection on primitive-integrated platform:* As pointed out in many works [142, 177–180], the DL models always contain redundancy. Therefore, lots of networks are compressed to boost computation efficiency. Meanwhile, advanced computation algebra [181–184] further speed ups the plaintext-level computation. However, how to apply these optimizations into privacy-preserving computation remains challenging as plaintext-level computation should be transformed into crypto arithmetic with big modulus, which makes the plaintext-level acceleration infeasible.

In other words, there is a need to search crypto-friendly computation architecture that is both efficient and accuracy-guaranteed. A few works have considered the network structure search in privacy-preserving DL, i.e., nonlinear function selection [66] and BNN selection [74]. Searching for the network architecture includes finding 1) the fit kernel sizes for convolution, 2) the pooling methods, 3) the nonlinear activation functions, and 4) the connections for the above three elements and all should consider the properties of the primitives, e.g., HE based square is more efficient than GC based ReLU. Meanwhile, a recent work shows an efficient Integer-Arithmetic-Only CNN structure [185], which may be used as a searching option for the integer-in-nature primitives such as HE.

Furthermore, given the optimized and modularized DL platforms, e.g., tensorflow [137], integrating the privacy-preserving primitives in those platforms can utilize the well developed

computation advantage and thus help to improve the efficiency of model search. A few works have embedded some primitives, e.g., HE, into tensorflow [133, 134, 186–189] while the comprehensive integration into DL platforms needs more efforts to facilitate the network search for optimal privacy-preserving DL architectures.

BIBLIOGRAPHY

- [1] T. Chen and S. Zhong, “Privacy-preserving backpropagation neural network learning,” *IEEE Transactions on Neural Networks*, vol. 20, no. 10, pp. 1554–1564, 2009.
- [2] J. Yuan and S. Yu, “Privacy preserving back-propagation neural network learning made practical with cloud computing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 212–221, 2013.
- [3] P. Mohassel and Y. Zhang, “Secureml: A system for scalable privacy-preserving machine learning,” in *2017 IEEE Symposium on Security and Privacy*. IEEE, 2017, pp. 19–38.
- [4] S. Wagh, D. Gupta, and N. Chandran, “Securenn: 3-party secure computation for neural network training,” *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 3, pp. 26–49, 2019.
- [5] P. Mohassel and P. Rindal, “Aby3: A mixed protocol framework for machine learning,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 35–52.
- [6] E. Hesamifard, H. Takabi, M. Ghasemi, and R. N. Wright, “Privacy-preserving machine learning as a service,” *Proceedings on Privacy Enhancing Technologies*, vol. 2018, no. 3, pp. 123–142, 2018.
- [7] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, “CrypTFlow2: Practical 2-party secure inference,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 325–342.

- [8] W. Li, H. Song, and F. Zeng, "Policy-based secure and trustworthy sensing for internet of things in smart cities," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 716–723, 2017.
- [9] G. Li, J. He, S. Peng, W. Jia, C. Wang, J. Niu, and S. Yu, "Energy efficient data collection in large-scale internet of things via computation offloading," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4176–4187, 2018.
- [10] *Total data volume worldwide 2010-2025 by Statista*. [Online]. Available: <https://www.statista.com/statistics/871513/worldwide-data-created/>
- [11] M. S. Hossain, M. Al-Hammadi, and G. Muhammad, "Automatic fruit classification using deep learning for industrial applications," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 2, pp. 1027–1034, 2018.
- [12] C. Esposito, X. Su, S. A. Aljawarneh, and C. Choi, "Securing collaborative deep learning in industrial applications within adversarial scenarios," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 11, pp. 4972–4981, 2018.
- [13] A. Bashar, "Survey on evolving deep learning neural network architectures," *Journal of Artificial Intelligence*, vol. 1, no. 02, pp. 73–82, 2019.
- [14] W. G. Hatcher and W. Yu, "A survey of deep learning: Platforms, applications and emerging research trends," *IEEE Access*, vol. 6, pp. 24 411–24 432, 2018.
- [15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.
- [16] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

- [17] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, “Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups,” *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82–97, 2012.
- [18] G. E. Dahl, D. Yu, L. Deng, and A. Acero, “Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition,” *IEEE Transactions on audio, speech, and language processing*, vol. 20, no. 1, pp. 30–42, 2011.
- [19] L. Deng, G. Hinton, and B. Kingsbury, “New types of deep neural network learning for speech recognition and related applications: An overview,” in *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013, pp. 8599–8603.
- [20] M. Sabokrou, M. Fayyaz, M. Fathy, and R. Klette, “Deep-cascade: Cascading 3d deep neural networks for fast anomaly detection and localization in crowded scenes,” *IEEE Transactions on Image Processing*, vol. 26, no. 4, pp. 1992–2004, 2017.
- [21] W. Luo, W. Liu, D. Lian, J. Tang, L. Duan, X. Peng, and S. Gao, “Video anomaly detection with sparse coding inspired deep neural networks,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019.
- [22] D. Kwon, K. Natarajan, S. C. Suh, H. Kim, and J. Kim, “An empirical study on network anomaly detection using convolutional neural networks,” in *2018 IEEE 38th International Conference on Distributed Computing Systems*. IEEE, 2018, pp. 1595–1598.
- [23] M. Mohammadi, A. Al-Fuqaha, S. Sorour, and M. Guizani, “Deep learning for iot big data and streaming analytics: A survey,” *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 2923–2960, 2018.

- [24] F. Tang, Z. M. Fadlullah, B. Mao, and N. Kato, "An intelligent traffic load prediction-based adaptive channel assignment algorithm in sdn-iot: A deep learning approach," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 5141–5154, 2018.
- [25] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.
- [26] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [27] M. Ribeiro, K. Grolinger, and M. A. Capretz, "Mlaas: Machine learning as a service," in *2015 IEEE 14th International Conference on Machine Learning and Applications*. IEEE, 2015, pp. 896–902.
- [28] H. Assem, L. Xu, T. S. Buda, and D. O’Sullivan, "Machine learning as a service for enabling internet of things and people," *Personal and Ubiquitous Computing*, vol. 20, no. 6, pp. 899–914, 2016.
- [29] T. C. Rindfleisch, "Privacy, information technology, and health care," *Communications of the ACM*, vol. 40, no. 8, pp. 92–100, 1997.
- [30] M. Meingast, T. Roosta, and S. Sastry, "Security and privacy issues with health care information technology," in *2006 International Conference of the IEEE Engineering in Medicine and Biology Society*. IEEE, 2006, pp. 5453–5458.
- [31] M. A. Rothstein, "Is deidentification sufficient to protect health privacy in research?" *The American Journal of Bioethics*, vol. 10, no. 9, pp. 3–11, 2010.

- [32] W. Gao, W. Yu, F. Liang, W. G. Hatcher, and C. Lu, “Privacy-preserving auction for big data trading using homomorphic encryption,” *IEEE Transactions on Network Science and Engineering*, 2018.
- [33] P. Voigt and A. Von dem Bussche, “The eu general data protection regulation (gdpr),” *A Practical Guide, 1st Ed., Cham: Springer International Publishing*, 2017.
- [34] W. B. Chik, “The singapore personal data protection act and an assessment of future trends in data privacy reform,” *Computer Law & Security Review*, vol. 29, no. 5, pp. 554–575, 2013.
- [35] E. Goldman, “An introduction to the california consumer privacy act (ccpa),” *Santa Clara Univ. Legal Studies Research Paper*, 2020.
- [36] A. Act, “Health insurance portability and accountability act of 1996,” *Public law*, vol. 104, p. 191, 1996.
- [37] *Uber to Pay 148 Million Penalty*. [Online]. Available: <https://www.wsj.com/articles/uber-to-pay-148-million-penalty-to-settle-2016-data-breach-1537983127>
- [38] *Data breach news: Singapore*. [Online]. Available: <https://www.channelnewsasia.com/news/singapore/ihis-singhealth-fined-1-million-data-breach-cyberattack-11124156>
- [39] *France fines Google 57 million*. [Online]. Available: <https://www.businessinsider.com/france-fines-google-57-million-for-gdpr-breach-2019-1>
- [40] C. Song, T. Ristenpart, and V. Shmatikov, “Machine learning models that remember too much,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 587–601.
- [41] M. Fredrikson, S. Jha, and T. Ristenpart, “Model inversion attacks that exploit confidence information and basic countermeasures,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1322–1333.

- [42] C. Dwork, “Differential privacy: A survey of results,” in *International conference on theory and applications of models of computation*. Springer, 2008, pp. 1–19.
- [43] C. Dwork, A. Roth *et al.*, “The algorithmic foundations of differential privacy.” *Foundations and Trends in Theoretical Computer Science*, vol. 9, no. 3-4, pp. 211–407, 2014.
- [44] C. Dwork, G. N. Rothblum, and S. Vadhan, “Boosting and differential privacy,” in *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*. IEEE, 2010, pp. 51–60.
- [45] C. Dwork, F. McSherry, K. Nissim, and A. Smith, “Calibrating noise to sensitivity in private data analysis,” in *Theory of cryptography conference*. Springer, 2006, pp. 265–284.
- [46] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, “Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave,” in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, 2016, pp. 1–9.
- [47] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, “Innovative instructions and software model for isolated execution.” *Hasp@ isca*, vol. 10, no. 1, 2013.
- [48] P. Paillier, “Public-key cryptosystems based on composite degree residuosity classes,” in *International conference on the theory and applications of cryptographic techniques*. Springer, 1999, pp. 223–238.
- [49] T. ElGamal, “A public key cryptosystem and a signature scheme based on discrete logarithms,” *IEEE transactions on information theory*, vol. 31, no. 4, pp. 469–472, 1985.

- [50] D. Boneh, E.-J. Goh, and K. Nissim, “Evaluating 2-dnf formulas on ciphertexts,” in *Theory of cryptography conference*. Springer, 2005, pp. 325–341.
- [51] Z. Brakerski, “Fully homomorphic encryption without modulus switching from classical gapsvp,” in *Annual Cryptology Conference*. Springer, 2012, pp. 868–886.
- [52] J. Fan and F. Vercauteren, “Somewhat practical fully homomorphic encryption.” *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 144, 2012.
- [53] J. H. Cheon, A. Kim, M. Kim, and Y. Song, “Homomorphic encryption for arithmetic of approximate numbers,” in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 409–437.
- [54] A. C. Yao, “Protocols for secure computations,” in *23rd annual symposium on foundations of computer science*. IEEE, 1982, pp. 160–164.
- [55] V. Kolesnikov and T. Schneider, “Improved garbled circuit: Free xor gates and applications,” in *International Colloquium on Automata, Languages, and Programming*. Springer, 2008, pp. 486–498.
- [56] D. Beaver, S. Micali, and P. Rogaway, “The round complexity of secure protocols,” in *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, 1990, pp. 503–513.
- [57] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, “Efficient garbling from a fixed-key blockcipher,” in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 478–492.
- [58] S. Zahur, M. Rosulek, and D. Evans, “Two halves make a whole,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 220–250.

- [59] Y. Lindell and B. Pinkas, “Secure two-party computation via cut-and-choose oblivious transfer,” *Journal of cryptology*, vol. 25, no. 4, pp. 680–722, 2012.
- [60] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, “Extending oblivious transfers efficiently,” in *Annual International Cryptology Conference*. Springer, 2003, pp. 145–161.
- [61] A. Shamir, “How to share a secret,” *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.
- [62] O. Goldreich, *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.
- [63] O. Goldreich, S. Micali, and A. Wigderson, “How to play any mental game,” in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, 1987, pp. 218–229.
- [64] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, “Gazelle: A low latency framework for secure neural network inference,” in *27th USENIX Security Symposium*, 2018, pp. 1651–1669.
- [65] J. Liu, M. Juuti, Y. Lu, and N. Asokan, “Oblivious neural network predictions via minionn transformations,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 619–631.
- [66] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, “Delphi: A cryptographic inference service for neural networks,” in *29th USENIX Security Symposium*, 2020.
- [67] Z. Liu, I. Tjuawinata, C. Xing, and K. Y. Lam, “Mpc-enabled privacy-preserving neural network training against malicious attack,” *arXiv preprint arXiv:2007.12557*, 2020.

- [68] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin, “Falcon: Honest-majority maliciously secure framework for private deep learning,” *arXiv preprint arXiv:2004.02229*, 2020.
- [69] N. Koti, M. Pancholi, A. Patra, and A. Suresh, “Swift: Super-fast and robust privacy-preserving machine learning,” *arXiv preprint arXiv:2005.10296*, 2020.
- [70] S. Wagh, D. Gupta, and N. Chandran, “Securenn: Efficient and private neural network training.” *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 442, 2018.
- [71] H. Chaudhari, R. Rachuri, and A. Suresh, “Trident: Efficient 4pc framework for privacy preserving machine learning,” in *27th Annual Network and Distributed System Security Symposium*, 2020, pp. 23–26.
- [72] K. Huang, X. Liu, S. Fu, D. Guo, and M. Xu, “A lightweight privacy-preserving cnn feature extraction framework for mobile sensing,” *IEEE Transactions on Dependable and Secure Computing*, 2019.
- [73] X. Liu, B. Wu, X. Yuan, and X. Yi, “Leia: A lightweight cryptographic neural network inference system at the edge.” *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 463, 2020.
- [74] A. Aggarwal, T. E. Carlson, R. Shokri, and S. Tople, “Soteria: In search of efficient neural networks for private inference,” *arXiv preprint arXiv:2007.12934*, 2020.
- [75] B. D. Rouhani, M. S. Riazi, and F. Koushanfar, “Deepsecure: Scalable provably-secure deep learning,” in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.
- [76] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. Lauter, and F. Koushanfar, “{XONN}: Xnor-based oblivious deep neural network inference,” in *28th USENIX Security Symposium*, 2019, pp. 1501–1518.

- [77] R. Zhu, C. Ding, and Y. Huang, “Practical mpc+ fhe with applications in secure multi-party neural network evaluation.” *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 550, 2020.
- [78] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, “Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy,” in *International Conference on Machine Learning*, 2016, pp. 201–210.
- [79] E. Chou, J. Beal, D. Levy, S. Yeung, A. Haque, and L. Fei-Fei, “Faster cryptonets: Leveraging sparsity for real-world encrypted inference,” *arXiv preprint arXiv:1811.09953*, 2018.
- [80] X. Jiang, M. Kim, K. Lauter, and Y. Song, “Secure outsourced matrix computation and application to neural networks,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1209–1222.
- [81] Q. Li, Z. Huang, W.-j. Lu, C. Hong, H. Qu, H. He, and W. Zhang, “Homopai: A secure collaborative machine learning platform based on homomorphic encryption,” in *2020 IEEE 36th International Conference on Data Engineering*. IEEE, 2020, pp. 1713–1717.
- [82] D. Froelicher, J. R. Troncoso-Pastoriza, A. Pyrgelis, S. Sav, J. S. Sousa, J.-P. Bossuat, and J.-P. Hubaux, “Scalable privacy-preserving distributed learning,” *arXiv preprint arXiv:2005.09532*, 2020.
- [83] A. A. Badawi, J. Chao, J. Lin, C. F. Mun, J. J. Sim, B. H. M. Tan, X. Nan, K. M. M. Aung, and V. R. Chandrasekhar, “The alexnet moment for homomorphic encryption: Hcnn, the first homomorphic cnn on encrypted data with gpus,” *arXiv preprint arXiv:1811.00778*, 2018.

- [84] H. Chen, W. Dai, M. Kim, and Y. Song, “Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 395–412.
- [85] H. Chabanne, A. de Wargny, J. Milgram, C. Morel, and E. Prouff, “Privacy-preserving classification on deep neural network.” *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 35, 2017.
- [86] A. S. Shamsabadi, A. Gascón, H. Haddadi, and A. Cavallaro, “Privedge: From local to distributed private training and prediction,” *IEEE Transactions on Information Forensics and Security*, 2020.
- [87] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, “Chameleon: A hybrid secure computation framework for machine learning applications,” in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 707–721.
- [88] N. Agrawal, A. Shahin Shamsabadi, M. J. Kusner, and A. Gascón, “Quotient: two-party secure neural network training and prediction,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1231–1247.
- [89] P. Xie, B. Wu, and G. Sun, “Bayhenn: combining bayesian deep learning and homomorphic encryption for secure dnn inference,” *arXiv preprint arXiv:1906.00639*, 2019.
- [90] Q. Zhang, C. Wang, H. Wu, C. Xin, and T. V. Phuong, “Gelu-net: A globally encrypted, locally unencrypted deep neural network for privacy-preserved learning.” in *IJCAI*, 2018, pp. 3933–3939.

- [91] Q. Zhang, C. Wang, C. Xin, and H. Wu, “Cheetah: An ultra-fast, approximation-free, and privacy-preserved neural network framework based on joint obscure linear and nonlinear computations,” *arXiv preprint arXiv:1911.05184*, 2019.
- [92] X. Ma, X. Chen, and X. Zhang, “Non-interactive privacy-preserving neural network prediction,” *Information Sciences*, vol. 481, pp. 507–519, 2019.
- [93] B. Reagen, W. Choi, Y. Ko, V. Lee, G.-Y. Wei, H.-H. S. Lee, and D. Brooks, “Cheetah: Optimizations and methods for privacy-preserving inference via homomorphic encryption,” *arXiv preprint arXiv:2006.00505*, 2020.
- [94] S. Li, K. Xue, B. Zhu, C. Ding, X. Gao, D. Wei, and T. Wan, “Falcon: A fourier transform based approach for fast and secure convolutional neural network predictions,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 8705–8714.
- [95] Q. Lou, B. Song, and L. Jiang, “Autoprivacy: Automated layer-wise parameter selection for secure neural network inference,” *arXiv preprint arXiv:2006.04219*, 2020.
- [96] S. Bian, T. Wang, M. Hiromoto, Y. Shi, and T. Sato, “Ensei: Efficient secure inference via frequency-domain homomorphic convolution for privacy-preserving visual recognition,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 9403–9412.
- [97] W. Zheng, R. A. Popa, J. E. Gonzalez, and I. Stoica, “Helen: Maliciously secure cooperative learning for linear models,” in *2019 IEEE Symposium on Security and Privacy*. IEEE, 2019, pp. 724–738.
- [98] Q. Zhang, C. Xin, and H. Wu, “Privacy preserving deep learning based on multi-party secure computation: A survey,” *IEEE Internet of Things Journal*, 2021.

- [99] *Developers' Alexa interface for device APIs*. [Online]. Available: <https://developer.amazon.com/zh/docs/device-apis/alexa-interface.html>
- [100] *The Google developer interface for web conversation*. [Online]. Available: <https://developers.google.com/actions/reference/rest/conversation-webhook>
- [101] V. V. Dixit, S. Chand, and D. J. Nair, "Autonomous vehicles: disengagements, accidents and reaction times," *PLoS one*, vol. 11, no. 12, p. e0168054, 2016.
- [102] Q. Zhang, C. Xin, and H. Wu, "Gala: Greedy computation for linear algebra in privacy-preserved neural networks," *arXiv preprint arXiv:2105.01827*, 2021.
- [103] —, "Securetrain: An approximation-free and computationally efficient framework for privacy-preserved neural network training," *IEEE Transactions on Network Science and Engineering*, 2020.
- [104] A. C.-C. Yao, "How to generate and exchange secrets," in *27th Annual Symposium on Foundations of Computer Science*. IEEE, 1986, pp. 162–167.
- [105] Y. Lindell, "How to simulate it—a tutorial on the simulation proof technique," *Tutorials on the Foundations of Cryptography*, pp. 277–346, 2017.
- [106] Y. Zhu, Y. Cheng, H. Zhou, and Y. Lu, "Hermes attack: Steal dnn models with lossless inference accuracy," *arXiv preprint arXiv:2006.12784*, 2020.
- [107] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction apis," in *25th USENIX Security Symposium*, 2016, pp. 601–618.
- [108] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *2017 IEEE Symposium on Security and Privacy*. IEEE, 2017, pp. 3–18.

- [109] X. Yuan, P. He, Q. Zhu, and X. Li, “Adversarial examples: Attacks and defenses for deep learning,” *IEEE transactions on neural networks and learning systems*, vol. 30, no. 9, pp. 2805–2824, 2019.
- [110] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli, “Evasion attacks against machine learning at test time,” in *Joint European conference on machine learning and knowledge discovery in databases*. Springer, 2013, pp. 387–402.
- [111] Z. Brakerski, C. Gentry, and S. Halevi, “Packed ciphertexts in lwe-based homomorphic encryption,” in *International Workshop on Public Key Cryptography*. Springer, 2013, pp. 1–13.
- [112] *Homomorphic Encryption*. [Online]. Available: <https://bit-ml.github.io/blog/post/homomorphic-encryption-toy-implementation-in-python/>
- [113] S. Erabelli, “pyfhe-a python library for fully homomorphic encryption,” Ph.D. dissertation, Massachusetts Institute of Technology, 2020.
- [114] G. Brassard, C. Crépeau, and J.-M. Robert, “All-or-nothing disclosure of secrets,” in *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1986, pp. 234–238.
- [115] D. Beaver, “Correlated pseudorandomness and the complexity of private computations,” in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 479–488.
- [116] V. Kolesnikov and R. Kumaresan, “Improved OT extension for transferring short secrets,” in *Annual Cryptology Conference*. Springer, 2013, pp. 54–70.

- [117] W. Wang, J. Gao, M. Zhang, S. Wang, G. Chen, T. K. Ng, B. C. Ooi, J. Shao, and M. Reyad, “Rafiki: Machine learning as an analytics service system,” vol. 12, no. 2. VLDB Endowment, 2018, pp. 128–140.
- [118] M. Mozaffari-Kermani, S. Sur-Kolay, A. Raghunathan, and N. K. Jha, “Systematic poisoning attacks on and defenses for machine learning in healthcare,” *IEEE Journal of Biomedical and Health Informatics*, vol. 19, no. 6, pp. 1893–1905, 2015.
- [119] S. Sohangir, D. Wang, A. Pomeranets, and T. M. Khoshgoftaar, “Big data: Deep learning for financial sentiment analysis,” *Journal of Big Data*, vol. 5, no. 1, p. 3, 2018.
- [120] I. File, “Proposal for a regulation of the european parliament and of the council on the protection of individuals with regard to the processing of personal data and on the free movement of such data (general data protection regulation),” *General Data Protection Regulation*, 2012.
- [121] B. Wang and N. Z. Gong, “Stealing hyperparameters in machine learning,” in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 2018, pp. 36–52.
- [122] L. Wan, W. K. Ng, S. Han, and V. Lee, “Privacy-preservation for gradient descent methods,” in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2007, pp. 775–783.
- [123] R. Shokri and V. Shmatikov, “Privacy-preserving deep learning,” in *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, 2015, pp. 1310–1321.
- [124] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, “Deep learning with differential privacy,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 308–318.

- [125] N. Phan, Y. Wang, X. Wu, and D. Dou, “Differential privacy preservation for deep auto-encoders: an application of human behavior prediction,” in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.
- [126] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, “Innovative instructions and software model for isolated execution,” in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.
- [127] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, “Oblivious multi-party machine learning on trusted processors,” in *Proceedings of the 25th USENIX Security Symposium*, 2016, pp. 619–636.
- [128] S. P. Bayerl, T. Frassetto, P. Jauernig, K. Riedhammer, A.-R. Sadeghi, T. Schneider, E. Stapf, and C. Weinert, “Offline model guard: Secure and private ml on mobile devices,” 2020.
- [129] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, “Opaque: An oblivious and encrypted distributed analytics platform,” in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, 2017, pp. 283–298.
- [130] S. Li, K. Xue, B. Zhu, C. Ding, X. Gao, D. Wei, and T. Wan, “Falcon: A fourier transform based approach for fast and secure convolutional neural network predictions,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 8705–8714.
- [131] R. Xu, J. B. Joshi, and C. Li, “Cryptonn: Training neural networks over encrypted data,” in *Proceedings of the IEEE 39th International Conference on Distributed Computing Systems*. IEEE, 2019, pp. 1199–1209.

- [132] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi, “Ezpc: Programmable, efficient, and scalable secure two-party computation for machine learning,” Cryptology ePrint Archive, Report 2017/1109, Tech. Rep., 2017.
- [133] F. Boemer, R. Cammarota, D. Demmler, T. Schneider, and H. Yalame, “Mp2ml: a mixed-protocol machine learning framework for private inference,” in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, 2020, pp. 1–10.
- [134] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, “Cryptflow: Secure tensorflow inference,” in *2020 IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 336–353.
- [135] *UCI Iris dataset*. [Online]. Available: <https://archive.ics.uci.edu/ml/datasets/iris>
- [136] “Microsoft seal (release 3.2),” <https://github.com/Microsoft/SEAL>, Feb. 2019, microsoft Research, Redmond, WA.
- [137] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “Tensorflow: A system for large-scale machine learning,” in *12th USENIX symposium on operating systems design and implementation*, 2016, pp. 265–283.
- [138] S. Halevi and V. Shoup, “Algorithms in helib,” in *Proceedings of the Annual Cryptology Conference*. Springer, 2014, pp. 554–571.
- [139] W. Henecka, S. K ögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, “Tasty: Tool for automating secure two-party computations,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010, pp. 451–462.
- [140] *MNIST dataset*. [Online]. Available: <http://yann.lecun.com/exdb/mnist/>

- [141] *CIFAR-10 dataset*. [Online]. Available: <https://www.cs.toronto.edu/~kriz/cifar.html>
- [142] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, “Model compression and hardware acceleration for neural networks: A comprehensive survey,” *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.
- [143] G. J. Annas *et al.*, “Hipaa regulations-a new era of medical-record privacy?” *New England Journal of Medicine*, vol. 348, no. 15, pp. 1486–1490, 2003.
- [144] A. A. Abd EL-Latif, B. Abd-El-Atty, S. E. Venegas-Andraca, and W. Mazurczyk, “Efficient quantum-based security protocols for information sharing and data protection in 5g networks,” *Future Generation Computer Systems*, vol. 100, pp. 893–906, 2019.
- [145] H. Qiu, K. Kapusta, Z. Lu, M. Qiu, and G. Memmi, “All-or-nothing data protection for ubiquitous communication: Challenges and perspectives,” *Information Sciences*, vol. 502, pp. 434–445, 2019.
- [146] K. Garimella, N. K. Jha, and B. Reagen, “Sisyphus: A cautionary tale of using low-degree polynomial activations in privacy-preserving deep learning,” *arXiv preprint arXiv:2107.12342*, 2021.
- [147] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, and M. Orrù, “Homomorphic secret sharing: optimizations and applications,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2105–2122.
- [148] E. Boyle, N. Gilboa, and Y. Ishai, “Breaking the circuit size barrier for secure computation under ddh,” in *Annual International Cryptology Conference*. Springer, 2016, pp. 509–539.
- [149] ———, “Group-based secure computation: optimizing rounds, communication, and computation,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2017, pp. 163–193.

- [150] O. Goldreich, “Secure multi-party computation,” *Manuscript. Preliminary version*, vol. 78, 1998.
- [151] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” in *Proceedings of the 22nd ACM international conference on Multimedia*, 2014, pp. 675–678.
- [152] P. Vepakomma, J. Balla, and R. Raskar, “Splintering with distributions: A stochastic decoy scheme for private computation,” *arXiv preprint arXiv:2007.02719*, 2020.
- [153] C. Gentry, “Fully homomorphic encryption using ideal lattices,” in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.
- [154] N. P. Smart and F. Vercauteren, “Fully homomorphic encryption with relatively small key and ciphertext sizes,” in *Proceedings of International Workshop on Public Key Cryptography*, 2010, pp. 420–443.
- [155] Y. Hu, “Improving the efficiency of homomorphic encryption schemes,” Ph.D. dissertation, Worcester Polytechnic Institute, 2013.
- [156] A. Mahendran and A. Vedaldi, “Understanding deep image representations by inverting them,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 5188–5196.
- [157] S. L. Campbell and C. D. Meyer, *Generalized inverses of linear transformations*. SIAM, 2009.
- [158] B. Hitaj, G. Ateniese, and F. Perez-Cruz, “Deep models under the gan: Information leakage from collaborative deep learning,” in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1310–1321.

- [159] A. Nguyen, J. Yosinski, and J. Clune, “Deep neural networks are easily fooled: High confidence predictions for unrecognizable images,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 427–436.
- [160] B. Schneier, *Applied cryptography: protocols, algorithms, and source code in C*. John Wiley & Sons, 2007.
- [161] R. Livni, S. Shalev-Shwartz, and O. Shamir, “On the computational efficiency of training neural networks,” in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, 2014, pp. 855–863. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2968826.2968922>
- [162] A. Patra, T. Schneider, A. Suresh, and H. Yalame, “ABY2.0: Improved mixed-protocol secure two-party computation,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
- [163] S. Tan, B. Knott, Y. Tian, and D. J. Wu, “CRYPTGPU: Fast privacy-preserving machine learning on the gpu,” *arXiv preprint arXiv:2104.10949*, 2021.
- [164] D. Demmler, T. Schneider, and M. Zohner, “ABY-a framework for efficient mixed-protocol secure two-party computation.” in *NDSS*, 2015.
- [165] M. Bellare, V. T. Hoang, and P. Rogaway, “Foundations of garbled circuits,” in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 784–796.
- [166] F. Mohammadzadeh, C. S. Nam, and E. Lobaton, “Prediction of physiological response over varying forecast lengths with a wearable health monitoring platform,” in *2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. IEEE, 2018, pp. 437–440.

- [167] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
- [168] J. Long, E. Shelhamer, and T. Darrell, “Fully convolutional networks for semantic segmentation,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440.
- [169] D. Bogdanov, S. Laur, and J. Willemsen, “Sharemind: A framework for fast privacy-preserving computations,” in *European Symposium on Research in Computer Security*. Springer, 2008, pp. 192–206.
- [170] [Online]. Available: <https://www.cs.toronto.edu/~kriz/cifar.html>
- [171] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.
- [172] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, “Cryptflow2: Practical 2-party secure inference.”
- [173] R. Agrawal, L. Bu, A. Ehret, and M. A. Kinsy, “Fast arithmetic hardware library for rlwe-based homomorphic encryption,” *arXiv preprint arXiv:2007.01648*, 2020.
- [174] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, “Heax: An architecture for computing on encrypted data,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1295–1309.
- [175] T. Morshed, M. M. A. Aziz, and N. Mohammed, “Cpu and gpu accelerated fully homomorphic encryption,” *arXiv preprint arXiv:2005.01945*, 2020.

- [176] D. Reis, J. Takeshita, T. Jung, M. Niemier, and X. S. Hu, “Computing-in-memory for performance and energy efficient homomorphic encryption,” *arXiv preprint arXiv:2005.03002*, 2020.
- [177] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” *arXiv preprint arXiv:1510.00149*, 2015.
- [178] S. Han, J. Pool, J. Tran, and W. Dally, “Learning both weights and connections for efficient neural network,” in *Advances in neural information processing systems*, 2015, pp. 1135–1143.
- [179] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “Eie: efficient inference engine on compressed deep neural network,” *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.
- [180] A. Kozlov, I. Lazarevich, V. Shamporov, N. Lyalyushkin, and Y. Gorbachev, “Neural network compression framework for fast model inference,” *arXiv preprint arXiv:2002.08679*, 2020.
- [181] Z. Zhang, H. Wang, S. Han, and W. J. Dally, “Sparch: Efficient architecture for sparse matrix multiplication,” in *2020 IEEE International Symposium on High Performance Computer Architecture*. IEEE, 2020, pp. 261–274.
- [182] F. Le Gall, “Powers of tensors and fast matrix multiplication,” in *Proceedings of the 39th international symposium on symbolic and algebraic computation*, 2014, pp. 296–303.
- [183] R. Yuster and U. Zwick, “Fast sparse matrix multiplication,” *ACM Transactions On Algorithms*, vol. 1, no. 1, pp. 2–13, 2005.

- [184] X. Huang and V. Y. Pan, “Fast rectangular matrix multiplication and applications,” *Journal of complexity*, vol. 14, no. 2, pp. 257–299, 1998.
- [185] H. Zhao, D. Liu, and H. Li, “Efficient integer-arithmetic-only convolutional neural networks,” *arXiv preprint arXiv:2006.11735*, 2020.
- [186] T. Ryffel, A. Trask, M. Dahl, B. Wagner, J. Mancuso, D. Rueckert, and J. Passerat-Palmbach, “A generic framework for privacy preserving deep learning,” *arXiv preprint arXiv:1811.04017*, 2018.
- [187] M. Dahl, J. Mancuso, Y. Dupis, B. Decoste, M. Giraud, I. Livingstone, J. Patriquin, and G. Uhma, “Private machine learning in tensorflow using secure computation,” *arXiv preprint arXiv:1810.08130*, 2018.
- [188] F. Boemer, A. Costache, R. Cammarota, and C. Wierzynski, “ngraph-he2: A high-throughput framework for neural network inference on encrypted data,” in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2019, pp. 45–56.
- [189] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski, “ngraph-he: a graph compiler for deep learning on homomorphically encrypted data,” in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, 2019, pp. 3–13.
- [190] S. C. Kothari, “Generalized linear threshold scheme,” in *Workshop on the Theory and Application of Cryptographic Techniques*. Springer, 1984, pp. 231–241.

APPENDIX A

SUPPLEMENTARY MATERIALS FOR SECURETRAIN

A.1 CONSTRUCTION OF (2,2)-THRESHOLD SCHEME

We tailor the Shamir threshold scheme [61, 190] from \mathbb{Z} to \mathbb{R} by data splintering [152] to enable secure and efficient training in SecureTrain. Formally, in a (t, w) -threshold scheme, the message m is split among w parties such that t of them are needed to reconstruct m , but no subset of smaller size than t can reconstruct m . The first thing is to randomly select $(t - 1)$ coefficients s_1, s_2, \dots, s_{t-1} and form a polynomial $s(x) = m + s_1x + \dots + s_{t-1}x^{t-1}$. In order to recover m, s_1, \dots, s_{t-1} , t pairs of $(x, s(x))$ are needed to solve the equation group.

The design of tuple $(r_1\mathbf{x}_1, r_2\mathbf{x}_2)$ is derived from the $(2, 2)$ -threshold scheme. Specifically, the $(2, 2)$ -threshold scheme forms a polynomial $s(x) = m + s_1x$ and in order to recover m and s_1 , 2 pairs of $(x, s(x))$ are needed to solve the equation group. Then we construct

$$r_2\mathbf{x}_2 = \mathbf{x} + \left(\frac{(r_2 - 1)\mathbf{x}}{r_1\mathbf{x}_1} - \frac{r_2}{r_1}\right)r_1\mathbf{x}_1 \quad (31)$$

which has a corresponding relation to $s(x) = m + s_1x$ that $s(x)$ is $r_2\mathbf{x}_2$, m is \mathbf{x} , s_1 is $\left(\frac{(r_2-1)\mathbf{x}}{r_1\mathbf{x}_1} - \frac{r_2}{r_1}\right)$ and x is $r_1\mathbf{x}_1$. The randomness of \mathbf{x}_1, r_1, r_2 means the randomness of $\left(\frac{(r_2-1)\mathbf{x}}{r_1\mathbf{x}_1} - \frac{r_2}{r_1}\right)$ thus correspondingly resulting in the randomness of s_1 . In this way, \mathcal{S} cannot obtain \mathbf{x} by tuple $(r_1\mathbf{x}_1, r_2\mathbf{x}_2)$.

A.2 BACK PROPAGATION EXCEPT LAST LAYER

\mathcal{C} obtains the \mathcal{S} -encrypted ReLU derivative, i.e., $[\frac{\partial \mathbf{x}}{\partial \mathbf{z}}]_{\mathcal{S}}$ as discussed in 4.3.4. Recall that, in forward propagation, \mathcal{C} creates the tuple $(r_1\mathbf{x}_1, r_2\mathbf{x}_2)$ and sends it to \mathcal{S} for the calculation

of weighted sum. To enable the backpropagation, \mathcal{C} additionally generates two terms and sends them along with the tuple $(r_1\mathbf{x}_1, r_2\mathbf{x}_2)$ to \mathcal{S} .

First, \mathcal{C} generates its share of ReLU derivative \mathbf{p}^c randomly and encrypts it as $[\mathbf{p}^c]_c$. Second, it creates the share for \mathcal{S} :

$$[\mathbf{p}^s]_s = [\frac{\partial \mathbf{x}}{\partial \mathbf{z}}]_s - \mathbf{p}^c = [\frac{\partial \mathbf{x}}{\partial \mathbf{z}} - \mathbf{p}^c]_s, \quad (32)$$

where $[\frac{\partial \mathbf{x}}{\partial \mathbf{z}}]_s$ is obtained by \mathcal{C} using Eq. (25). Upon receiving $[\mathbf{p}^s]_s$ and $[\mathbf{p}^c]_c$, \mathcal{S} decrypts $[\mathbf{p}^s]_s$ into \mathbf{p}^s and computes

$$(\mathbf{p}^s + [\mathbf{p}^c]_c) \odot \mathbf{q}^s = [\mathbf{p}^s + \mathbf{p}^c]_c \odot \mathbf{q}^s = [\frac{\partial \mathbf{x}}{\partial \mathbf{z}}]_c \odot \mathbf{q}^s, \quad (33)$$

where \mathbf{q}^s is a mask vector for protecting the elements in $\frac{\partial \mathbf{x}}{\partial \mathbf{z}}$.

Furthermore, \mathcal{S} creates $(r_3\delta_1^s, r_4\delta_2^s)$, where r_3 and r_4 are two random numbers generated by \mathcal{S} while δ_1^s and δ_2^s are two random shares satisfying $\delta_1^s + \delta_2^s = \hat{\delta}^s$. Note that $\hat{\delta}^s$ has been obtained by \mathcal{S} in the $(k+1)$ -th layer. The tuple along with the result of Eq. (33) are sent to \mathcal{C} . This transmission is piggybacked to the HSS based share set χ_1 in back propagation from \mathcal{S} to \mathcal{C} for $(k+1)$ -th layer.

\mathcal{C} subsequently computes the following five ciphertexts and a plaintext tuple:

$$\left\{ \begin{array}{l} C_1 = [\hat{\delta}^c \hat{\mathbf{w}}^c \odot \frac{\partial \mathbf{x}}{\partial \mathbf{z}} \odot \mathbf{q}^s]_c \\ C_2 = [r_3\delta_1^s \hat{\mathbf{w}}^c \odot \frac{\partial \mathbf{x}}{\partial \mathbf{z}} \odot \mathbf{q}^s]_c \\ C_3 = [r_4\delta_2^s \hat{\mathbf{w}}^c \odot \frac{\partial \mathbf{x}}{\partial \mathbf{z}} \odot \mathbf{q}^s]_c \\ C_4 = [\frac{\frac{\partial \mathbf{x}}{\partial \mathbf{z}} \odot \mathbf{q}^s}{r_5}]_c, C_5 = [\frac{\frac{\partial \mathbf{x}}{\partial \mathbf{z}} \odot \mathbf{q}^s}{r_6}]_c \\ (r_5\delta_1^c, r_6\delta_2^c), \end{array} \right. \quad (34)$$

where $\hat{\delta}^c$ and $\hat{\mathbf{w}}^c$ are available to \mathcal{C} as the share of δ and \mathbf{w} in the $(k+1)$ -th layer. The tuple $(r_5\delta_1^c, r_6\delta_2^c)$ is generated by \mathcal{C} such that δ_1^c and δ_2^c are two random shares satisfying

$\delta_1^C + \delta_2^C = \mathring{\delta}^C$ while r_5 and r_6 are two random numbers. The results of Eq. (34) are piggybacked to the message when \mathcal{C} sends the ciphertext in Eq. (22) to \mathcal{S} for $(k+1)$ -th layer. Then \mathcal{S} conducts:

$$\left\{ \begin{array}{l} C_1 \odot \frac{1}{q^S}, C_2 \odot \frac{1}{r_3 q^S}, C_3 \odot \frac{1}{r_4 q^S} \\ C_4 \odot \frac{r_5 \delta_1^C \hat{w}^S}{q^S}, C_5 \odot \frac{r_6 \delta_2^C \hat{w}^S}{q^S} \\ \mathring{\delta}^S \hat{w}^S \odot [\frac{\partial x}{\partial z}]_C, \end{array} \right. \quad (35)$$

and subsequently adds the six items in Eq. (35) to obtain the \mathcal{C} -encrypted δ , i.e., $[\delta]_C$:

$$\begin{aligned} & C_1 \odot \frac{1}{q^S} + C_2 \odot \frac{1}{r_3 q^S} + C_3 \odot \frac{1}{r_4 q^S} + C_4 \odot \frac{r_5 \delta_1^C \hat{w}^S}{q^S} \\ & + C_5 \odot \frac{r_6 \delta_2^C \hat{w}^S}{q^S} + \mathring{\delta}^S \hat{w}^S \odot [\frac{\partial x}{\partial z}]_C \\ & = [\mathring{\delta}^C \hat{w}^C \odot \frac{\partial x}{\partial z}]_C + [\delta_1^S \hat{w}^C \odot \frac{\partial x}{\partial z}]_C + [\delta_2^S \hat{w}^C \odot \frac{\partial x}{\partial z}]_C \\ & + [\delta_1^C \hat{w}^S \odot \frac{\partial x}{\partial z}]_C + [\delta_2^C \hat{w}^S \odot \frac{\partial x}{\partial z}]_C + [\mathring{\delta}^S \hat{w}^S \odot \frac{\partial x}{\partial z}]_C \\ & = [\mathring{\delta} \hat{w} \odot \frac{\partial x}{\partial z}]_C = [\delta]_C. \end{aligned} \quad (36)$$

Since \mathcal{S} now owns \mathcal{C} -encrypted δ , \mathcal{C} and \mathcal{S} can continue the update of weight and bias in a way similar to that in the last layer as introduced in 4.3.4.

A.3 COMPLEXITY ANALYSIS

This section analyzes the computation as well as communication complexity of Secure-Train.

Computation Complexity. As for forward propagation in Section 4.3.2¹, \mathcal{C} firstly conducts one Add and one Mult to get $r_2 \mathbf{x}_2$ based on encrypted $\mathbf{x} = [\mathbf{a}]_S$. Then, after \mathcal{S} receives the HSS based share set χ_1 , as shown in Figure 13, \mathcal{S} conducts four Mult to respectively obtain $[r_2 \mathbf{x}_1 \mathbf{w}^S]_C$, $[r_2 \mathbf{b}^S]_C$, $[r_2 \mathbf{x}_2 \mathbf{w}_1^C]_C$ and $[r_2 \mathbf{x}_2 \mathbf{w}_2^C]_C$. Then, five Add in Eq. (10) results in the client-encrypted linear sum $[r_2 \mathbf{z}]_C$ at \mathcal{S} .

Next, the non-linear calculation starts from Eq. (5), which involves one Mult to scramble the linear result into $[r_2 \mathbf{z} \odot \mathbf{v}^S]_C$. Then, \mathcal{S} sends \mathcal{C} the HSS based share set χ_2 (see Figure

¹We analyze the situation except first layer as the complexity at the first layer is lighter.

13) and \mathcal{C} get server-encrypted non-linear result by Eq. (8), which needs two Mult and one Add. Therefore, the total complexity for each layer in forward propagation is seven Add ($O(1)$) and eight Mult ($O(1)$).

As for **softmax** calculation in Section 4.3.3, SecureTrain features with non-approximation design within one-round communication. Specifically, one Mult and one Add are firstly needed for \mathcal{S} to disturb \mathbf{z} by Eq. (11). Then, \mathcal{S} sends \mathcal{C} HSS based share set χ_1 (see Figure 14). \mathcal{C} accordingly calculates by Eq. (12) with one Mult and one Add. As the following computation only involves plaintext, **softmax** needs two Mult ($O(1)$) and two Add ($O(1)$).

As for back propagation in Section 4.3.4², \mathcal{S} firstly refreshes its bias share in plaintext by $\widehat{\mathbf{b}}^{\mathcal{S}}$. After that, \mathcal{S} sends \mathcal{C} HSS based share set χ_1 (see Figure 15), which involves one Add and one Mult.

After \mathcal{C} receives χ_1 , it updates its bias share by Eq. (17). To update the weight, \mathcal{C} refreshes its weight share by Eq. (18). Then, \mathcal{C} forms HSS based share set χ_2 (see Figure 15). Specifically, $\frac{n_i n_o}{n_s}$ Mult are needed to obtain Eq. (21). This is because the size of $\overrightarrow{\delta \odot \mathbf{l}^{\mathcal{S}}}$ or $\overrightarrow{\mathbf{x}_2}$ is $n_i n_o$, which needs $\frac{n_i n_o}{n_s}$ ciphertexts. Then, another $\frac{n_i n_o}{n_s}$ Add is needed to sum up Eq. (19) and Eq. (21). After that, \mathcal{C} gets the weight share for \mathcal{S} with $\frac{n_i n_o}{n_s}$ Mult and $\frac{n_i n_o}{n_s}$ Add by Eq. (22). Upon receiving the set χ_2 , \mathcal{S} updates its weight share by Eq. (23).

Moreover, specific computation is piggybacked in the previous process to enable the next round of weight/bias update. In concrete terms, \mathcal{C} firstly gets ReLU derivative by Eq. (25) in forward propagation, with one Mult and one Add. Next, \mathcal{C} forms the share of ReLU derivative for \mathcal{S} with another one Add by Eq. (32). \mathcal{S} then gets the scrambled ReLU derivative by Eq. (33) with one Add and one Mult. Finally, \mathcal{S} conducts six Mult by Eq. (35) and five Add by Eq. (36) to finally get the $[\delta]_{\mathcal{C}}$.

Thus the computation complexity for backpropagation is $(9 + \frac{2n_i n_o}{n_s})$ Mult (with complexity of $O(\frac{n_i n_o}{n_s})$) and $(9 + \frac{2n_i n_o}{n_s})$ Add (with complexity of $O(\frac{n_i n_o}{n_s})$).

Communication Complexity. Similar to the analysis for computation complexity, there

²We analyze the situation from $(n - 1)$ -th layer as the complexity at the last layer is lighter.

are three communication parts that are respectively involved in forward propagation, softmax and back propagation.

As for forward propagation in Section 4.3.2³ shown in Figure 13, to calculate the linear weighted sum, \mathcal{C} firstly sends \mathcal{S} the HSS based share set χ_1 , including five ciphertexts ($[r_2]_{\mathcal{C}}$, $[\frac{r_2}{r_1}]_{\mathcal{C}}$, $[r_2(\mathbf{x}_1\mathbf{w}^{\mathcal{C}} + \mathbf{b}^{\mathcal{C}})]_{\mathcal{C}}$, $[\frac{1}{h_1}]_{\mathcal{C}}$ and $[\frac{1}{h_2}]_{\mathcal{C}}$), two plaintext vector ($r_1\mathbf{x}_1$ and $r_2\mathbf{x}_2$) with dimension of n_i , and two matrices ($h_1\mathbf{w}_1^{\mathcal{C}}$ and $h_2\mathbf{w}_2^{\mathcal{C}}$) with dimension of $n_i \times n_o$.

Then, \mathcal{S} sends \mathcal{C} the HSS based share set χ_2 (see Figure 13), which contains three ciphertexts ($[r_2\mathbf{z} \odot \mathbf{v}^{\mathcal{S}}]_{\mathcal{C}}$, $[\mathbf{g}_1]_{\mathcal{S}}$ and $[\mathbf{g}_2]_{\mathcal{S}}$) for computation of non-linear ReLU activation. Thus the total interactive data includes eight ciphertext with $8s_c$ in bit, two plaintext vector with $2n_i s_p$ in bit, and two plaintext matrices with $2n_i n_o s_p$ in bit. One communication round is involved.

As for **softmax** shown in Figure 14, two ciphertexts ($[r_2(\mathbf{z} + \mathbf{d}_{\mathcal{S}})]_{\mathcal{C}}$ and $[e^{-\mathbf{d}_{\mathcal{S}}}]_{\mathcal{S}}$) are firstly transmitted from \mathcal{S} to \mathcal{C} as the HSS based share set χ_1 for the calculation of Eq. (12) and Eq. (13). Then, one ciphertext ($[re^{\mathbf{z}} + \mathbf{o}]_{\mathcal{S}}$) and one plaintext vector ($(\mathbf{d}_{\mathcal{C}} \odot e^{\mathbf{z} + \mathbf{d}_{\mathcal{S}}})$) with dimension of n_o are sent from \mathcal{C} to \mathcal{S} as the HSS based share set χ_2 to accurately get the share of **softmax** function. Therefore, the total transmitted data during **softmax** calculation is three ciphertexts (with $3s_c$ in bit) and one plaintext vector (with $n_o s_p$ in bit). One communication round is involved.

In back propagation⁴ shown in Figure 15, \mathcal{S} begins the update of bias by $\widehat{\mathbf{b}}^{\mathcal{S}}$, After that, \mathcal{S} sends \mathcal{C} the HSS based share set χ_1 for the update of weight and bias, including two individual ciphertext, $\frac{n_i n_o}{n_s}$ ciphertext for $[\overrightarrow{r_2 \mathbf{x}_2}]_{\mathcal{S}}$ and $\frac{n_i n_o}{n_s}$ ciphertext for $[\overrightarrow{\mathbf{t}^{\mathcal{S}}}]_{\mathcal{S}}$. Then, another $\frac{n_i n_o}{n_s}$ ciphertext by Eq. (22) are transmitted from \mathcal{C} to \mathcal{S} as HSS based share set χ_2 for the update of weight at \mathcal{S} .

Similar to the analysis for computation complexity, specific communication is piggy-backed in the previous layer to get the error $[\delta]_{\mathcal{C}}$ for the current layer and then enable the

³We also consider the layer except first layer, as the communication complexity for the first layer is lighter.

⁴We analyze the situation without last layer as the communication complexity at the last layer is lighter.

next round of weight/bias update. Specifically, \mathcal{S} firstly sends \mathcal{C} one ciphertext, $[g_3]_{\mathcal{S}}$, to get server-encrypted ReLU derivative at \mathcal{C} by Eq. (25). Then, two ciphertexts, $[p^c]_{\mathcal{C}}$ and $[p^s]_{\mathcal{S}}$, are sent from \mathcal{C} to \mathcal{S} for the calculation of scrambled ReLU derivative by Eq. (33). The scrambled ReLU derivative along with two plaintext vectors, $r_3\delta_1^S$ and $r_4\delta_2^S$, with dimension of n_o , are transmitted to \mathcal{C} . \mathcal{C} then forms five ciphertext (C_1 to C_5 in Eq. (34)) and two plaintext vectors ($r_5\delta_1^C$ and $r_6\delta_2^C$, with dimension of n_o) to enable \mathcal{S} obtain the client-encrypted error $[\delta]_{\mathcal{C}}$ by Eq. (36).

In all, back propagation involves $(11 + \frac{3n_i n_o}{n_s})$ ciphertext (with $(11 + \frac{3n_i n_o}{n_s})s_c$ bit) and four plaintext vectors (with $4n_o s_p$ bit). As the calculation for error $[\delta]_{\mathcal{C}}$ involves no extra communication round (see 4.3.4), SecureTrain needs only one communication round for weight/bias update in each layer.

A.4 SECURITY AGAINST A SEMI-HONEST SERVER

We now prove the security against a semi-honest server. We define a simulator **sim** that simulates an admissible adversary \mathcal{A} which has compromised the server in the real world. In forward propagation as shown in Figure 13, **sim** conducts the following: 1) abstracts the randomness of server and forms a random vector \mathbf{b}^S and a random matrix \mathbf{w}^S ; 2) sends \mathbf{b}^S and \mathbf{w}^S to \mathcal{F} and gets HSS based share set χ_1 ; 3) constructs a random vector $\tilde{\mathbf{b}}^S$ and a random matrix $\tilde{\mathbf{w}}^S$; and 4) receives from \mathcal{C} the HSS based share set $\tilde{\chi}_1$. Here χ_1 and $\tilde{\chi}_1$ are indistinguishable due to the randomness of the tuple in χ_1 and the security of CKKS. Thus the forward propagation is secure against a semi-honest server.

In **softmax** calculation as shown in Figure 14, **sim** conducts as follows: 1) abstracts the randomness of the server and forms the HSS based share set χ_1 ; 2) sends χ_1 to \mathcal{F} and gets HSS based share set χ_2 according to equations from Eq. (12); 3) constructs another HSS based share set $\tilde{\chi}_1$, which have the same structure as χ_1 ; and 4) sends $\tilde{\chi}_1$ to \mathcal{C} and receives from \mathcal{C} the HSS based share set $\tilde{\chi}_2$ based on equations from Eq. (12). Here χ_2 and $\tilde{\chi}_2$ are indistinguishable due to the randomness of r , \mathbf{o} and $\mathbf{d}_{\mathcal{C}}$ in Eqs. (12) and (13). Thus the

softmax calculation is secure against a semi-honest server.

In back propagation as shown in Figure 15, **sim** conducts as follows: 1) abstracts the randomness of server and forms the HSS based share set χ_1 ; 2) sends χ_1 to \mathcal{F} and receives the HSS based share set χ_2 ; 3) constructs another HSS based share set $\tilde{\chi}_1$ which has the same structure as χ_1 ; and 4) sends $\tilde{\chi}_1$ to \mathcal{C} and receives another HSS based share set $\tilde{\chi}_2$ according to Eq. (22). Here $\tilde{\chi}_2$ and χ_2 are indistinguishable due to the randomness of Δ^c in Eq. (22).

Similar to the case under semi-honest client, the calculation of $[\delta]_c$ for the current layer is piggybacked in the weight/bias update for the previous layer to enable the next round of weight/bias update. In such case, **sim** conducts as follows: 1) abstracts the randomness of the server and forms a ciphertext $[g_3]_s$; 2) sends $[g_3]_s$ to \mathcal{F} and receives two ciphertext based on Eq. 32, and five ciphertext along with a plaintext tuple according to Eq. (34); 3) constructs another ciphertext $[\tilde{g}_3]_s$; 4) sends $[\tilde{g}_3]_s$ to \mathcal{C} and receives two ciphertext based on Eq. (32); and 5) sends a ciphertext (see Eq. (33)) and a plaintext tuple $(r_3\delta_1^s, r_4\delta_2^s)$ to \mathcal{C} and receives another five ciphertexts and one plaintext tuple according to Eq. (34). Here the six ciphertexts from step 2) are indistinguishable from these from step 5) due to the security of CKKS. And the plaintext in step 2) is also indistinguishable from the one in step 5) due to the randomness of p^c , δ_1^c , r_5 and r_6 . Thus, the back propagation is secure against a semi-honest server.

VITA

Qiao Zhang

Department of Electrical & Computer Engineering
Old Dominion University, Norfolk, VA 23529

Education

- Ph.D. Electrical and Computer Engineering, Dec. 2021, Old Dominion University
- M.Sc. Electrical and Computer Engineering, June 2017, Chongqing University of Posts and Telecommunications
- B.Sc. Electrical and Computer Engineering, June 2014, Chongqing University of Posts and Telecommunications

Publications

- Zhang, Qiao and Xin, Chunsheng and Wu, Hongyi, “*GALA: Greedy ComputAtion for Linear Algebra in Privacy-Preserved Neural Networks*,” in Proc. NDSS, Virtual, February 2021.
- Zhang, Qiao and Wang, Cong and Wu, Hongyi and Xin, Chunsheng and Phuong, Tran V., “*GELU-Net: A Globally Encrypted, Locally Unencrypted Deep Neural Network for Privacy-Preserved Learning*,” in Proc. IJCAI, Stockholm, Sweden, July 2018.
- Zhang, Qiao and Xin, Chunsheng and Wu, Hongyi, “*SecureTrain: An Approximation-Free and Computationally Efficient Framework for Privacy-Preserved Neural Network Training*,” in IEEE Trans. Network Science and Engineering, November 2020.
- Zhang, Qiao and Xin, Chunsheng and Wu, Hongyi, “*Privacy Preserving Deep Learning based on Multi-Party Secure Computation: A Survey*,” in IEEE Internet of Things Journal, February 2021.
- Fathalla, Efat and Wu, Hongyi and Azab, Mohamed M and Xin, Chunsheng and Zhang, Qiao, “*DT-SSIM: A Decentralized Trustworthy Self-Sovereign Identity Management Framework*,” in IEEE Internet of Things Journal, September 2021.

Typeset using L^AT_EX.