DEGREE PROJECT IN Electrical Engineering,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2024

Safe Weighted Federated Learning: Accelerated Multi-Party Computation Architectures based Intel SGX

Jin Xin

KTH ROYAL INSTITUTE OF TECHNOLOGY
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Authors

Jin Xin <xiji@kth.se>
Information and Communication Technology Innovation, Electrical Engineering
KTH Royal Institute of Technology

Place for Thesis

Stockholm, Sweden

Examiner

Johan Håstad
KTH Royal Institute of Technology

Supervisor

Ming Xiao
KTH Royal Institute of Technology
Tianxiang Dai, Yong Li
Huawei Munich Research Center, Trustworthy Technology and Engineering Lab

Abstract

Federated Learning (FL) allows training Machine Learning (ML) models without sharing private data, which is crucial for domains like finance and healthcare. However, FL is vulnerable to inference and membership attacks. To enhance security, Multi-Party Computation (MPC) are used, but they face challenges such as high computational costs, memory requirements, and communication overhead, leading to prolonged training times.

This thesis introduces a weighted federated learning approach using secret-sharing scheme MPC, and accelerated by plain-text computation with Intel Software Guard Extensions (SGX). We propose and evaluate two secure weighted FL computation infrastructures, SGXDL and HybridSGXDL, for their efficiency, feasibility, and practicality in Convolutional Neural Networks (CNNs).

Because of utilizing Intel SGX for plain-text computation, these infrastructures achieve significant efficiency improvements, reducing training time by at least 5× compared to Piranha(GPU-accelerated MPC platform). For moderate tasks, improvements can reach 164×, and efficiency gains continue with larger models. Intel SGX limit available libraries to C standard libraries, consequently requiring MPC developers' expertise in both applied mathematics and C implementation. SGXDL and HybridSGXDL demonstrate the practicality of securely training CNNs such as ResNet18 and VGG16, allowing developers to access and use the models for secure training without extensive mathematics expertise.

Keywords

Federated Learning, Intel SGX, Multi-Party Computation

Abstract

Federated Learning (FL) möjliggör träning av maskininlärningsmodeller (ML) utan att dela privat data, vilket är avgörande för områden som finans och sjukvård. FL är dock sårbart för inferens- och medlemskapsattacker. För att öka säkerheten används Multi-Party Computation (MPC), men dessa står inför utmaningar som höga beräkningskostnader, minneskrav och kommunikationsbelastning, vilket leder till förlängda träningstider.

Denna avhandling introducerar en viktad federerad inlärningsmetod med användning av MPC baserat på en hemlighetsdelningsschema, accelererat genom beräkningar i klartext med Intel Software Guard Extensions (SGX). Vi föreslår och utvärderar två säkra viktade FL-beräkningsinfrastrukturer, SGXDL och HybridSGXDL, för deras effektivitet, genomförbarhet och praktiska tillämpning i konvolutionella neurala nätverk (CNN).

Genom att använda Intel SGX för klartextberäkningar uppnår dessa infrastrukturer betydande effektivitetsförbättringar, vilket minskar träningstiden med minst 5× jämfört med Piranha (GPU-accelererad MPC-plattform). För måttliga uppgifter kan förbättringarna nå 164×, och effektivitetsvinsterna fortsätter med större modeller. Intel SGX begränsar tillgängliga bibliotek till C-standardbibliotek, vilket därmed kräver MPC-utvecklares expertis inom både tillämpad matematik och C-implementering. SGXDL och HybridSGXDL demonstrerar praktikaliteten i att säkert träna CNNs som ResNet18 och VGG16, vilket möjliggör för utvecklare att använda modellerna för säker träning utan omfattande matematisk expertis.

Nyckelord

Federerad Inlärning, Intel SGX, Flerpartiberegningsprotokoll

Acknowledgements

I would like to express my deepest gratitude to my company supervisor, Dr. Tianxiang Dai for his academic guidance and support during my thesis study. I am also immensely grateful to my colleagues: Fei Mei, for assisting with my thesis plan; Jonas Fierlings, for helping with the code implementation; and Yufan Jiang, for swiftly helping me understand the MPC mechanism.

I would like to extend my sincere thanks to my examiner, Prof.Dr. Johan Håstad at KTH, for his insightful guidance and academic suggestions very important for the foundation of my work, which greatly contributed to the smooth progress of my thesis. My gratitude also goes to my supervisor, Prof.Dr. Ming Xiao, at KTH.

A special acknowledgment goes to Prof.Dr. Markus Flierl at KTH, whose invaluable guidance and support were crucial during my thesis internship. His assistance with my time plan and visa arrangements enabled me to start my internship on time. I deeply appreciate his kindness, enthusiasm, and unwavering commitment to illuminating the path for students and researchers.

Acronyms

CPU	Central Processing Unit
CNNs	Convolutional Neural Networks
ReLU	Rectified linear unit
FL	Federated Learning
MPC	Multi-Party Computation
TEEs	Trusted Execution Environments
ML	Machine Learning
DL	Deep Learning
SGX	Software Guard Extensions
RSA	Rivest–Shamir–Adleman
PKI	Public Key Infrastructure
DP	Differential Privacy
HE	Homomorphic Encryption
AES	Advanced Encryption Standard

Contents

Chapter 1

Introduction

1.1 Background

The success of Machine Learning (ML) in artificial intelligence applications, such as recommendation systems, image recognition, and natural language processing, has led to a significant increase in the volume of available datasets. In the current landscape, expansive dataset repositories and the pervasive influence of ML applications across various domains—such as healthcare [84], autonomous vehicle systems [54], and financial markets [57, 81]—pose considerable privacy threats when datasets are disclosed. FL facilitates collaborative model training by transmitting the gradients of local datasets, rather than raw data, to a central server for updating Deep Learning (DL) model weights. This approach not only preserves data privacy but also distributes computational load across decentralized devices.

Despite its promise, FL is vulnerable to gradient-related adversarial attacks, potentially exposing sensitive information[10, 40, 102]. To mitigate these vulnerabilities, several defense mechanisms have been proposed, including MPC, Differential Privacy (DP), and Trusted Execution Environments (TEEs). While MPC can mitigate threats through secret-sharing schemes without revealing the inputs of computations, it is burdened by significant communication overhead and computational complexity, particularly for non-linear operations such as activation functions in deep learning [48, 98]. The secure MPC-based training of large models can be up to $10^4\times$ slower than standard plain-text computation [98]. Other secure computation techniques, such as Homomorphic Encryption (HE), provides varying degrees of security and privacy but

also suffer from computational overhead, impeding their practical deployment for real-world deep learning applications' workloads.

Given the significant overhead associated with cryptographic computations, a crucial question emerges: Is it possible to perform these computations in plain-text to eliminate the overhead while still preserving security? This thesis seeks to address this question by utilizing Intel SGX to enable secure plain-text computation, thereby overcoming the efficiency and practicality challenges inherent in existing secure computation methods.

1.1.1 Piranha: a secure GPU computing platform

Secure MPC training of large-scale ML models often requires extensive periods to complete. To address this, Piranha [100] introduces a general-purpose solution to accelerate MPC computation, enabling the training of realistic neural networks within roughly one day with GPU assistance. However, the computation efficiency associated with secret sharing is still significantly lower than that of plain-text computation. Additionally, memory requirements increase rapidly as the number of computing clients grows. This escalating memory demand and computational burden of MPC pose significant challenges to its deployment in real-world applications.

Plain-text machine learning operates directly with floating-point arithmetic under reasonable memory constraints, whereas MPC operates over integer types. Although Piranha implements integer operations on GPUs for MPC, GPUs are inherently optimized for accelerating floating-point computations, which are more suitable for plain-text machine learning.

This research proposes and implements hardware security-based MPC protocols that utilize secure plain-text floating-point computation schemes. This approach aligns more naturally with plain-text machine learning computations and fully leverages GPUs' floating-point acceleration capabilities. The proposed hardware security-based computing infrastructures, specifically SGXDL, enable secure deep learning applications to be practically deployed in real-world scenarios without compromising efficiency, regardless of the number of computing participants. By leveraging Intel SGX for secure plain-text computation, this research aims to overcome the limitations of existing MPC methods, enhancing both the practicality and scalability of privacy-preserving machine learning.

1.2 Problems

Deploying and supporting real-world secure Multi-Party Computation (MPC) machine learning applications pose overhead challenges. Traditional Federated Learning (FL) is susceptible to security breaches, while MPC incurs substantial communication overhead, particularly for non-linear operations. Our proposed SGXDL and HybridSGXDL frameworks aim to address these issues by providing efficient and secure solutions, leveraging Intel SGX for plain-text computation.

1.2.1 Privacy Challenges in Federated Learning

Although Federated Learning promotes a privacy-aware framework, it remains vulnerable to inference attacks both from within and outside the system, compromising data privacy and system integrity [63, 65, 75]. The gradients sent to the server during FL process can unintentionally reveal sensitive information[36, 95, 101], allowing attackers to infer private data through modified inputs. This presents the privacy risk in scenarios where malicious actors intentionally misrepresent data. For instance, during adversarial CNNs training, if the attacker intend to declare an image belonging to Alice as belonging to Eve, the victim device may continuously contribute Alice's information to correct the model, leading to unintended data leakage. This underscores the critical need for advanced privacy-preserving techniques in FL to prevent such unauthorized access and protect individual user privacy. Techniques such as MPC, differential privacy(DP), and homomorphic encryption(HE) are vital to enhancing the privacy and integrity of the FL ecosystem. Effective privacy-preserving strategies are crucial to protect individual users' data and maintain the overall trustworthiness of the FL framework[10, 79, 113].

1.2.2 Overhead Challenges in Secure FL techniques

A lot of work have studied to integrate the encrypted privacy-preserving methods into FL to enhance the privacy. However, HE and MPC are not applicable for large-scale FL in real-world applications, because the encrypted computation without revealing the insights of data incur heavy communication and computation overhead. While DP provide a solution with reasonable overhead, it requires the aggregated value to contain noise up to a certain magnitude at aggregation-based tasks, which is not ideal for FL[63]. Additionally, DP is also facing the challenge of explainability,

trackability, accuracy drop and vulnerability to attacks, for example, complicating the ability to monitor specific information such as fraudulent activities and adjust models accordingly [19]. Thus, developing effective and secure FL solutions which address these limitations is imperative.

1.2.3 Expanded Memory Requirements and Workload Growth Challenges

Multi-Party Computation (MPC) protocols typically accommodate between one to four participants. However, in real-world scenarios, deep learning applications can involve more parties. As number of computing parties and the complexity of CNN model increase, the memory requirements and computational workloads of shares grow quickly. This escalation poses significant challenges to the feasibility of deploying MPC protocols in practical, large-scale applications. The Piranha platform serves as a notable example of these scalability challenges. During a 3-party setup for training AlexNet on the MNIST dataset with a batch size of 300, the platform experienced a memory overflow on a 24GB GPU. This scenario, despite being relatively modest tasks for deep learning applications, underscores the substantial scalability issues faced by current MPC protocols in managing large-scale operations.

To address the scalability challenges inherent in MPC, integrating Trusted Execution Environments (TEEs) is crucial. TEEs can replace segments of the encrypted computation workload with plain-text computations while preserving the security benefits of MPC. This integration can significantly reduce computational and memory overheads, making large-scale MPC deployments more feasible.

1.2.4 Non-linear Operation Overhead Challenge in MPC

Multi-Party Computation is an effective technique to bolster privacy and security in FL, enables secure computation between multiple parties without disclosing inputs and outputs, which inspire the research of applying MPC in FL[34]. However, MPC's encrypted computation methods incur significant communication overhead for non-linear operations because of more data share exchanges[59]. This overhead can further ruin processing speed due to network bandwidth constraints and synchronization, causing GPUs to remain underutilized as they await required data tokens from other devices. To improve the performance of MPC based FL in real-world scenarios

while maintaining security, optimizing non-linear operations in Convolutional Neural Networks (CNNs) is the most important module in this thesis to address.

1.3 Methodology

1.3.1 Multi-Party Computation based Federated Learning

Secure Multi-Party Computation Protocols(SMPC/MPC) have emerged as a promising technique to enhance security in FL[11, 31, 46, 47, 89, 110]. MPC enables participants to perform joint computations and gain insights from data based on secret shares without the decrypting process or revealing the raw data. Crucially, the information each party holds appear random and meaningless without useful information. To enhance this architecture, we aim to implement a MPC-basd FL framework to reinforce security guarantees while significantly reducing the computational overhead, by integrating Trusted Execution Environments (TEEs). This approach will streamline computations, ensuring efficient, secure processing without compromising the confidentiality integral to MPC.

1.3.2 Trusted Execution Environments(TEEs) - Intel SGX

Despite the benefits of MPC, it often suffers from significant communication overhead, particularly with large datasets and complex machine learning models. Trusted Execution Environments (TEEs), such as Intel SGX, offer a robust solution for ensuring privacy-preserving execution of plain-text computations [16]. TEEs provide a secure enclave for executing safety-critical operations and sensitive functions without exposure to external entities[28]. Intel SGX enables physical isolation on the Central Processing Unit (CPU) to securely process privacy data and code within an enclave[43]. The data and functions within the SGX enclave remain inaccessible to any external entity including the server that holds the TEEs, ensuring that computations can be performed securely and confidentially. By utilizing TEEs, it is feasible to replace the computationally intensive overhead operations in MPC with plain-text computations, maintaining accuracy and efficiency compared to other privacy-preserving techniques such as Differential Privacy(DP).

1.4 Solutions

To address the significant overhead issues associated with MPC in secure deep learning, we propose two secure computing infrastructures SGXDL and HybridSGXDL. These infrastrctures facilitate the deployment of secure and efficient deep learning applications among clients in real-world scenarios without compromising accuracy. By balancing the trade-off between efficiency and security, our approach aims to significantly reduce computational workload while maintaining the security of the system. The core idea is to replace secret-sharing computation with plain-text computation into Intel SGX enclaves, eliminating the need for encrypted computation and reducing computational workload.

While Intel SGX provides a secure environment for computation, it is not sufficient for a fully secure collaborative computing framework. Public key authentication is required at initialization to ensure data security outside of Intel SGX and during the communication process.

1.4.1 Intel-SGX Plain-text Secure Deep Learning(SGXDL)

The size of SGX enclaves has significantly increased from the previous 257MB to 64GB and beyond. This expansion allows current server enclaves to accommodate complete large deep learning models and calculations, effectively reducing communication overhead and operation complexity in Multi-Party Computation (MPC). The primary idea of SGXDL is to perform all computation operations within the enclaves. Intel-SGX Plain-text Secure Deep Learning (SGXDL) infrastructure conducts the entire plain-text training process inside the enclave. This approach eliminates the need for communication and Outside Calls (OCalls) during training, thereby mitigating the risk of side-channel attacks.

1.4.2 Hybrid Intel SGX-GPU Secure Deep Learning(HybridSGXDL)

While the proposed SGXDL infrastructure is effective and secure, performing the entire training process within the enclave on a CPU limits the use of GPU acceleration. Inspired by Piranha, which utilizes GPUs to accelerate MPC machine learning, we propose a heterogeneous Intel SGX-GPU computing infrastructure to further improve

the efficiency of secure computing by leveraging GPUs. This infrastructure distributes computation workloads between Intel SGX and local GPUs.

We introduce and implement the privacy-preserving infrastructure Hybrid Intel SGX-GPU Secure Deep Learning Framework (HybridSGXDL) to accelerate secure computing by using GPUs. This framework incorporates efficient secure deep learning schemes where data samples are distributed among parties using a secret-sharing scheme[5] and GPUs to accelerate the computations.

In HybridSGXDL, computing operations are divided into linear and non-linear modules. Linear operations, which are less computationally intensive for MPC (e.g., addition and multiplication), are executed locally on clients' GPUs. Intel SGX serves as a trusted third party to integrate the results of these secret-sharing linear computations from edge parties ,and compute non-linear operations such as Rectified linear unit (ReLU) and MaxPool in plain-text within Intel SGX.

The secret-sharing non-linear intermediate results are then returned from Intel SGX to the edge devices for the next set of linear operations in a recursive manner. The backward propagation process also follows the same non-linear and linear computation separation rules, the backward linear computation results would be integrated in the enclave. The updated gradients are then separated into random shares and distributed back to the corresponding devices, ensuring the integrity and privacy of the data throughout the process.

Table 1.4.1: Target Efficiency Benchmark Analysis

Computing Schemes	**Platforms**
Proposed secure computing infrastructures	SGXDL, HybridSGXDL
Cutting edge secure computation platform	Piranha
Plain-text computation	CPU, GPU

1.5 Purpose

The purposes of the research are to:

- Implement efficient and secure deep learning infrastructures that replace encrypted computation with secure plain-text computation in Intel SGX(HybridSGXDL and SGXDL).

- Explore the feasibility of conducting the entire training process of plain-text deep learning models within an enclave(SGXDL).

- Design a scheme that separates linear operations on local devices' GPUs using a secret-sharing approach, while performing non-linear operations in Trusted Execution Environments (TEEs).

- Implement secure and efficient deep learning platforms capable of running LeNet, AlexNet, ResNet18, and VGG16. These platforms will be implemented in pure C style, making them compatible with most types of servers.

1.6 Hypothesis, Environment and Threat Model

1.6.1 Hypothesis

- The number of computing parties $n \in \mathbf{N}$ is not theoretically limited and can be 1,2,3,4,5,6,7....etc in SGXDL. For HybridSGXDL, the number of participating parties implemented in the thesis is 3.

- All information is in secret-sharing format outside the Intel SGX enclave.

- There will be only one server holding the SGX, and other parties will send the secret shares to the SGX server. SGX is the only trusted entity, and all parties including the server are assumed to be semi-honest.

- There will be only communications between the SGX server and clients, with no communication between clients in SGXDL.

- In HybridSGXDL there are communications among servers and clients.

- Network setting: LAN 2GB/s with 2ms latency

- The federated learning (FL) in the thesis will apply the weighted federated learning format as described in [114].

1.6.2 Environment

Development Environment

Operating System: Ubuntu 22.04

Default Maximum Enclave Page Cache (EPC) Size for Intel SGX:64 GB

GPU: Nvidia RTX A5000, memory of 24GB

Architecture: x86_64

CPU op-mode(s): 32-bit, 64-bit

Address sizes: 46 bits physical, 57 bits virtual

Byte Order: Little Endian

CPU(s): 144

On-line CPU(s) list: 0-143

Vendor ID: GenuineIntel

Model name: Intel(R) Xeon(R) Platinum 8360Y CPU @ 2.40GHz

CPU family: 6

Model: 106

Thread(s) per core: 2

Core(s) per socket: 36

Socket(s): 2

Stepping: 6

CPU max MHz: 3500,0000

CPU min MHz: 800,0000

BogoMIPS: 4800.00

Build tools

Intel® SGX Linux 2.9 release

CUDA 11.7

GCC 13.1

C++ 17

1.6.3 Threat Model

The thesis considers a semi-honest threat model, where attackers may corrupt parties but still follow the protocol. In other words, the corrupted parties participate in Federated Learning (FL) honestly but attempt to extract as much information as possible from the messages they receive from other parties and the SGX (i.e., they are honest-but-curious). The semi-honest adversaries may try to learn information from the honest parties and exploit the established communication channels of the protocol.

In SGXDL, the communication is exclusively between the server's SGX and the clients . The key point is to establish a secure channel between the SGX and the clients, ensuring that the server party, which mainly handles the share messages, cannot learn or retrieve the information exchanged. And for HybridSGXDL, there are communications among clients and server, additional secure channels are required to protect the communication among clients. For more rigorous scenarios involving malicious environments, additional monitoring and detection mechanisms are required, but these are beyond the scope of this thesis.

SGXDL security targets

The purpose of SGXDL is to safeguard the model structures, parameters, and raw data, including intermediate products, during the training process. To extract useful information by integrating all shares, adversaries would need to simultaneously compromise all participating parties. Consequently, as the number of participating computing parties increases, the overall security is strengthened.

HybridSGXDL security targets

The primary security objective of HybridSGXDL is to ensure the confidentiality of raw data and model parameters across all participating parties, effectively preventing the server from accessing any plaintext information. This is in contrast to traditional federated learning models, where the server typically receives gradient values from all participants.

Even in the most adverse scenarios—such as when secure communication channels are entirely compromised and the server is capable of reconstructing the shares—only intermediate results might be exposed. The raw data remains protected, as these

intermediate results are composite products of both addition and multiplication during local convolution, making it infeasible to reconstruct the original raw data. This level of security is considered sufficient to meet the objectives of HybridSGXDL.

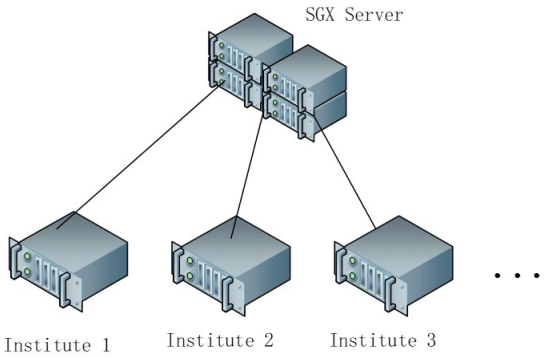

Figure 1.6.1: Communication Connection Model(SGXDL)

Chapter 2

Theoretical Background

This chapter provides a comprehensive overview of federated learning, focusing on weighted federated learning (wFL) based on secret-sharing schemes used in this thesis. It addresses the security challenges in traditional federated learning. Furthermore, it details the deep learning operations used in the thesis such as convolution and ReLU, essential for the deployment within Intel SGX using limited C standard libraries. Understanding these operations is crucial for grasping how secret-sharing computations for convolution, accelerated by GPUs on local devices, can be integrated into the HybridSGXDL framework.

Intel SGX functionalities are explored, including interface design and implementation. In conventional federated learning, the server can potentially perform inference attacks using received gradients. Despite the fact that data is protected through secret random shares in this thesis, there remains a risk of the server accessing all shares. To mitigate this risk, public key authentication is implemented to create a secure communication channel between SGX and client parties, ensuring data integrity and secure communication.

2.1 Secret-Sharing based MPC

Secret sharing scheme is a method to safely share data between numerous gatherings, provides superior performance for arithmetic operations such as matrix addition and multiplication over other cryptographic tools, and has been extensively used for privacy-preserving CNNs' inference and training[12, 13, 97].

2.1.1 Additive Secret Sharing

Additive secret sharing(ASS) and shamir secret sharing[83] are two widely used schemes. Shamir[83] proposed a (n,t)-threshold scheme based polynomials for the secret share that the raw data is divided into n shares, and (1) any t shares are able to retrieve the information (2) any t-1 shares reveal no useful information. And ASS is a (n,n)-threshold secret sharing scheme that raw data is divided into n shares, and can only be retrieved with n shares. In ASS, the raw data x will be randomly divided into $[x_0, x_1, x_2, ...x_{n-1}]$ on a finite field such that $\sum_{i=0}^{n-1} x_i = x$. Many ASS-based MPC protocols are widely used for privacy-preserving computation in recent days[7, 8, 20, 35, 58, 107]. This thesis defines ASS-based MPC learning frameworks over the real number field **R** for floating-point arithmetic operations.

2.1.2 Multi-Party Computation(MPC)

The security community has a long accepted common sense that any security systems are born with their corresponding attack surface, it is nearly impossible to design a fully secure complex system[25], adversaries will be able to penetrate and steathily take control over some of the network nodes. To enhance security, it is crucial to minimize the attack surface and maximize the cost of attacks.

MPC is introduced by Andrew Yao[108] in 1982. Yao gave a precise formulation for multi party computation problem(e.g. Two millinaires wish know who is richer without disclosing the information of wealth values). The problem can be modified as m parties wish to jointly compute a function $f(x_0, x_1, ...x_{m-1})$ where x_i is the i^{th} party's private input of integers with bounded range. Several well-known MPC protocols have been developed, including Yao's Garbled Circuits, GMW [33, 35], BGW [6], BMR [4], and GESS [50].

MPC enhances privacy in machine learning by allowing joint data processing without exposing individual inputs. However, the significant communication and computation overhead makes it impractical for large-scale data and models. Hybrid solutions combining MPC with homomorphic encryption (HE) have been proposed [30, 80]. To improve efficiency, trusted execution environments (TEEs) like Intel SGX are used to assist MPC, balancing encrypted computation and hardware security [21, 104]. Recognizing the high cost of nonlinear computations in MPC, plain-text arithmetic nonlinear operations are performed in SGX for HybridSGXDL to reduce

overhead.

2.2 Federated Learning

2.2.1 History of Secret-Sharing Federated Learning

The success of convolutional deep learning [55], advancements in attention mechanisms [96], and the emergence of large language models like ChatGPT [1] have highlighted the potential of deep learning in artificial intelligence. The availability of big data has accelerated the adoption of deep learning in various fields such as healthcare, finance, autonomous systems, and e-commerce. However, individual institutions often lack sufficient data, necessitating collaborative training with datasets from other parties. Directly sharing and integrating datasets poses significant privacy risks.

Federated Learning (FL), introduced by Google in 2016 [70], addresses these privacy concerns by allowing clients to train models locally and send model updates to a server for integration. However, FL frameworks are vulnerable to gradient-based attacks [64]. Consequently, secret-sharing-based Federated Learning (FL) emerged to enhance security by distributing data in secret shares, ensuring data remains encrypted and random, revealing very least information unless all shares are integrated [27]. This makes it significantly challenging for attackers, as they must compromise multiple parties simultaneously to obtain useful information.

There is a substantial body of work focusing on using secret-sharing schemes to assist federated learning. Bonawitz et al. [9] presented a privacy-preserving MPC protocol using Shamir's t-out-of-n Secret Sharing [83] for secure gradient aggregation in FL. Zhang et al. [111] combined threshold secret sharing with homomorphic encryption (HE) to address similar challenges. Y. Dong et al. [22] applied secret sharing for distributed learning. Jinhyun So et al. [88] proposed Turbo, which masks models at local clients with randomness using additive secret-sharing schemes, canceling out noise during aggregation. Secret-sharing is widely used in FL [42, 61, 76, 85, 87, 106].

Typically, centralized traditional FL trusts the server is honest to receive everything. However, for stronger security, this thesis assumes a semi-honest threat model where the server is also semi-honest and can be monitored by an attacker capable of observing

its behavior. In such cases, the Intel SGX hardware enclave ensures that all parties, including the server, cannot access the code and data within it. Even if an attacker alters the server's behavior and introduces disturbances, the FL performance may be affected, but the information accessible to the attacker remains unavailable due to the secure channel. Intel SGX provides strong protection against adversaries.

2.2.2 Secure techniques for Federated Learning

Homomorhpic Encryption (HE) [3, 61]allows FL to execute the gradients aggregation over cipher-text on server without decrypting or revealing information to the server in advance[15, 26, 38, 60, 109], which provides a strong guarantee for data privacy. However, HE supports a limited number of arithmetic operations and incurs substantial computational overhead due to complex cryptographic operations [77]. Encryption and decryption on local clients can consume up to 80% of training time in local plain-text machine learning model updates [109].

Local Differential Privacy (LDP) adds noise to local datasets, protecting individual data privacy [24, 49]. However, LDP introduces uncertainty, potentially reducing accuracy, thus necessitating a balance between privacy and accuracy [75].

Multi-Party Computation (MPC) protocols provide strong security for decentralized FL by distributing different random shares to clients, revealing information only when a threshold number of shares are integrated, thus requiring an attacker to compromise multiple parties simultaneously. However, MPC incurs significant communication and computation overhead, particularly for non-linear computations like Rectified Linear Unit (ReLU) and MaxPool, which involve comparisons and conversions between arithmetic and binary secret sharing [32, 97]. ReLU latency can be 10,000 times slower than convolution operations [32].

Intel SGX is an encrypted area on the CPU that protects the code and data within the enclave from external access [14, 17, 71, 72]. Introduced in 2015, Intel SGX allows running code and storing data securely on an untrusted device [17]. It provides the option of secure floating-point plain-text computation without accuracy loss, thereby eliminating the overhead associated with encrypted computation.

2.2.3 Data Split in Federated Learning

Classical Federated Learning (FL) frameworks are typically categorized as horizontal, vertical, and transfer learning, based on how datasets are split. In vertical FL, the dataset is split by features with the same objects, while in horizontal FL, the dataset is split by objects with the same feature space. In this thesis, data is distributed in secret shares, and the corresponding federated learning approach is weighted federated learning (wFL) [114], which will be introduced in the following section.

Wagh et al. [106] have demonstrated that secret-sharing schemes offer the highest efficiency in secure deep learning scenarios. When individual clients have stringent privacy requirements, they may be reluctant to expose personal data to any institution, complicating data generation and training processes. Via secret-sharing, access to private data for model training or inference can be certified and agreed by individual client to solve the concern. For instance, if Alice wants to take a genetic disease test and investigate the results herself, she can send her data in three secret-sharing values to three independent institutions that have databases for DNA diagnosis [23].

2.2.4 Weighted Federated Learning

Zhu et al. [114] proposed a weighted federated learning (wFL) scheme that utilizes secret sharing to split private data into random shares. Building on this approach, this thesis applies the wFL to private data to meet higher privacy demands. This secret-sharing MPC-based wFL method has the potential to unlock a wide range of machine learning applications that were previously inaccessible due to data privacy concerns.

Procedure of weighted federated learning

Consider a 3-party wFL scenarios with three MPC parties: P_0, P_1, P_2, two private data x and y. In Zhu's work x and y are defined in a finite integer field Z_p^*(where p is a suitable large prime number e.g. $|p| = 512$). State-of-art secret sharing schemes are also defined on finite fields and rings of integers to improve the security that attackers can not learn information from the magnitude of shares.

Since this thesis implements plain-text floating-point computation with the assistance of Intel SGX, it focuses on plain-text computation rather than cryptographic

computation. The secret-sharing scheme is employed purely for security purposes. Here, x and y are generated in a floating arithmetic field such that x is divided into 3 secret shares: $x_0, x_1, x_2 \in \mathbf{R}$, y is divided into 3 secret shares: $y_0, y_1, y_2 \in \mathbf{R}$. We do not set a bounded range in theory because such bounds can lead to information leakage. For example, if $x_0, x_1, x_2 \in (-100, 100), x_0 = -56, x_1 = -43$, it can be inferred that $x < 1$. Applying state-of-the-art cryptographic integer computation would involve overhead of conversions between integers and floating-point numbers. Therefore, a real number secret-sharing scheme is proposed for secure plain-text computation in this thesis.

K. Tjell et al. [94] introduced a real number secret-sharing scheme leveraging Gaussian distribution, bypassing the need for integer shares and modular arithmetic, thus facilitating direct arithmetic computation on shares. However, Tjell's work is based on (n,t)-threshold Shamir secret-sharing, where higher t implies more security but less reliability. In contrast, this thesis uses (n,n)-threshold additive secret sharing. In the following context, we address potential information leakage in real number additive secret-sharing.

Assume there is a secure channel between Alice and the MPC servers, implemented under the standard Public Key Infrastructure (PKI) assumption [2], ensuring that the secret shares are securely distributed during initialization.

Real number (n,n)-threshold additive secret sharing In Convolutional Neural Networks(CNNs), the fundamental arithmetic operations are addition and multiplication, with no division involved. Non-linear operations, such as activation functions, are executed in plain-text within the enclave. Thus, non-linear operations are not discussed in the context of secret sharing schemes in the following modules. The values of x_0, x_1, x_2, y_0, y_1, y_2 are set as:

- Select random numbers $x_0, x_1, y_0, y_1 \in \mathbf{R}$ where each is Gaussian distributed with mean value zero. Such that $x_0 \sim \mathcal{N}(0, \sigma_{x_0}^2), x_1 \sim \mathcal{N}(0, \sigma_{x_1}^2), y_0 \sim \mathcal{N}(0, \sigma_{y_0}^2), y_1 \sim \mathcal{N}(0, \sigma_{y_1}^2)$.
- $x_2 = x - x_0 - x_1$, $y_2 = y - y_0 - y_1$.

The shares of x are distributed as follows in HybridSGXDL:

- P_0 holds $(x_0, x_2), (y_0, y_2)$.

- P_1 holds $(x_0, x_1), (y_0, y_1)$.
- P_2 holds $(x_1, x_2), (y_1, y_2)$.

Here, we describe the shares distribution in HybridSGXDL(In SGXDL, each share is distributed to one of the MPC parties).

Addition of x and y

- P_0 make the computation $z_0 = x_0 + y_0$.
- P_1 make the computation $z_1 = x_1 + y_1$
- P_2 make the computation $z_2 = x_2 + y_2$
- Finally, z_0, z_1 and z_2 can be sent to SGX to get the result: $z_0 + z_1 + z_2 = x + y = (x_0 + x_1 + x_2) + (y_0 + y_1 + y_2)$.

Multiplication of x and y

- P_0 make the computation $z_0 = x_0 * y_0 + x_0 * y_2 + x_2 * y_0$.
- P_1 make the computation $z_1 = x_1 * y_1 + x_1 * y_0 + x_0 * y_1$
- P_2 make the computation $z_2 = x_2 * y_2 + x_2 * y_1 + x_1 * y_2$
- Finally, z_0, z_1, and z_2 can be sent to SGX to get the result: $z_0 + z_1 + z_2 = x * y = (x_0 + x_1 + x_2) * (y_0 + y_1 + y_2)$.

In the worst-case scenario where an attacker fully compromises the secure channels, they might be able to observe z_0, z_1 and z_2. And if $z_0 + z_1 + z_2 = 0 = x * y$ holds, then attacker could deduce x or y is 0. However, in the thesis, such leakage is considered to pose no significant risk because the intermediate products of HybridSGXDL exchanged during communication are not the result of single multiplication or addition operations. Instead, they are composite results of a convolution process, which inherently involves both multiplication and addition. Although there is a risk that information about these intermediate products could be exposed during training once the secure channels were compromised, it remains insufficient for reconstructing the original raw data involved in the computation. The primary goal of the secure computing infrastructure outlined in the thesis is to protect this raw data from being compromised by external parties or adversaries.s.

2.2.5 Privacy analysis for real secret-sharing in SGXDL

In SGXDL, data shares are generated in the same way in Section 2.2.4. However, the data distribution would be different that each party only holds one share of the one data, e.g. for x:

- P_0 holds x_0.
- P_1 holds x_1.
- P_2 holds x_2.

And send the shares secretly to the SGX directly which will handle all plain-text computation. In this section, we analyze the privacy of the proposed real number secret sharing when each party only holds one share. We will demonstrate later that the party P_i can learn very limited information leakage of data x from the holding share x_i.

Combining all three shares would be possible to reconstruct x, a set of fewer than 3 shares should reveal very little information, we formally state this information leakage by using the information theoretical measure called mutual information[93]. Mutual information is a measure that quantifies the amount of information one random variable provides about another, in terms of entropy. It essentially indicates how much the knowledge of one variable reduces the uncertainty of another. If the mutual information between two random variables is zero, it signifies that the variables are independent.

In computer system real numbers can only be represented by using a finite number of bits, typically according to the IEEE 754 standard. This means numbers are stored in a format that approximates real values, leading to rounding errors. Such floating-point errors would be discussed later.

$$I(X;Y) = h(X) - h(X|Y) \tag{2.1}$$

Mutual information $I(X|Y)$ measures the information that X and Y share, if $I(X|Y) = 0$ then X does not give any information about Y which means there is no information leakage in theory. $h(X)$ is the entropy of X and $h(X|Y)$ is the conditional entropy of X when Y is known.

Consider the mutual information $I(X;X_0)$ between X and X_0, where x is a certain value

of variable X, x_0 of X_0.

$$\begin{aligned} I(X; X_0) &= h(X_0) - h(X_0|X) \\ &= h(X_0) - h(X_0) = 0 \end{aligned} \tag{2.2}$$

X_0 is a Gaussian distribution independent from X such that $h(X_0|X) = h(X_0)$, we can infer that $I(X; X_1) = 0$ in the same way:

$$\begin{aligned} I(X; X_1) &= h(X_1) - h(X_1|X) \\ &= h(X_1) - h(X_1) = 0 \end{aligned} \tag{2.3}$$

Consider the mutual information $I(X; X_2)$ between $x \in X$ and $x_2 \in X_2$. Assume the variance of input data x is σ_x.

$$\begin{aligned} I(X; X_2) &= h(X_2) - h(X_2|X) \\ &= h(X - (X_0 + X_1)) - h(X - X_0 - X_1|X) \\ &= h(X - X_0 - X_1) - \int_{\mathcal{R}} p(x) h(X - X_0 - X_1|X = x) \\ &= h(X - X_0 - X_1) - \int_{\mathcal{R}} p(x) h(\mathcal{N}(x, \sigma_{x_0}^2 + \sigma_{x_1}^2)) \\ &\leq \frac{1}{2}\log(2\pi \mathrm{e}(\sigma_x^2 + \sigma_{x_0}^2 + \sigma_{x_1}^2)) - \frac{1}{2}\log(2\pi \mathrm{e}(\sigma_{x_0}^2 + \sigma_{x_1}^2)) \\ &= \frac{1}{2}\log(1 + \frac{\sigma_x^2}{\sigma_{x_0}^2 + \sigma_{x_1}^2}) \end{aligned} \tag{2.4}$$

Where X is independent of $-(X_0 + X_1)$, the variance of $X - (X_0 + X_1)$ is

$$\sigma_x^2 + (\sigma_{x_0}^2 + \sigma_{x_1}^2) \tag{2.5}$$

We can conclude that there is no information leakage for x_0 and x_1, and an upper bound(Equation2.4) for x_2's information leakage. Given that the variance of dataset σ_x^2 is a constant number, when the variance of x_0 and x_1 tends to infinity, the information leakage of x_2 tends to 0, we can select reasonable variance for Gaussian distribution that x_2 would not leak too much information.

In practical scenarios,before any information leakage occurs, the generation schedule of x_0, x_1, x_2 is unknown that x_0, x_2 could also be the Gaussian distribution random numbers, the attacker must identify which share deviates Gaussian distribution could only potentially leak information. The data share separation, determined individually

by each client, can vary randomly for each data sample. For example, for one data sample the non-Gaussian component can be x_2, while in another, it could be x_0, this make the cost of guessing the non-Gaussian component expensive for the adversary. Even if the adversary guess the non-Gaussian component correctly, the shares held by parties are protected under the secure channels, and there is a information leakage upper bound as shown in Equation 2.4, which can approach 0 with suitable variance selection($\sigma_{x_0}^2$ and $\sigma_{x_1}^2$). This randomness of the generation schedule significantly increases the difficulty and cost for the adversary in guessing the non-Gaussian component.

Floating point errors Computing systems are inherently discrete, with a finite number of floating-point numbers available. For instance, in a 32-bit floating-point data type, there are only 2^{32} available values, in contrast to the infinity of real numbers. Consequently, computers cannot generate a true Gaussian distribution; they can only approximate the Gaussian random number generation process. This discrepancy between theoretical models and practical implementation can lead to deviations in information leakage estimation. These deviations are measurable and can be considered reasonable. $I(X; X_0)$ and $I(X; X_1)$ accounting for the floating point error remain 0, because X,X_0 and X_1 are independent. For the mutual information for $I(X; X_2)$, where floating-point rounding and tailing errors might play a role, the impact can be more nuanced, one could generate samples and estimate the empirical entropy using a Monte Carlo simulation[74]. This approach would allow for a more detailed analysis of how these errors influence mutual information. However, this investigation is beyond the scope of the current thesis.

Most methods used for generating Gaussian distributions, such as the Box-Muller transform or the Ziggurat algorithm[67, 92], rely on fundamental properties like the Central Limit Theorem. While floating-point errors can slightly distort the output, the overall distribution still maintains its Gaussian characteristics (i.e., it is still symmetric, bell-shaped, and centered around a mean). Floating-point errors can approach zero with very high precision computer system.

In this thesis, the primary focus is on the efficiency of securing computing infrastructures using Intel SGX. Consequently, we do not delve into the effects of floating-point system errors on mutual information. Future work could explore this area, providing a more comprehensive understanding of how these errors impact

mutual information in contexts where precision is critical.

Table 2.2.1: Proposed SGXDL secure deep learning

Secure SGXDL weighted Federated Learning
1) Data Split: m-random secret shares of one sample distributed to m parties $P_0, P_1, ...P_{m-1}$. P_0 is the only server equipped Intel SGX.
2) Local Computation: The SGXDL CNNs model is securely stored within the Intel SGX enclave and remains within the enclave throughout the entire inference and training process.
3) SGX Aggregation: The shares owned by each party locally are sent to SGX secretly. SGX integrate the shares and complete the deep learning training in enclave.

2.2.6 Privacy analysis for real secret-sharing in HybridSGXDL

In HybridSGXDL, each party holds 2 shares that, P_i can learn very limited information leakage of x from its holding shares. Shares for each data sample x are generated in the same way as shown in Section 2.2.4

Assume the shares distribution condition is(the same in Section 2.2.4):

- P_0 holds (x_0, x_2).
- P_1 holds (x_0, x_1).
- P_2 holds (x_1, x_2).

The information leakage are defined in the same way of mutual information in Section 2.2.5.

$I(X; X_0, X_1)$ is information of random variable X that P_1 can learn from known X_0 and X_1. Since X_0 and X_1 are the Gaussian distributions independent from X, so the mutual information is zero:

$$I(X; X_0, X_1) = 0 \tag{2.6}$$

$I(X; X_0, X_2)$ is information of random variable X that P_0 can learn from known X_0 and

X_2. X_0 is independent from X, and due to Equation 2.4 such that:

$$\begin{aligned} I(X; X_0, X_2) &= I(X; X_2) \\ &\leq \frac{1}{2}\log(1 + \frac{\sigma_x^2}{\sigma_{x_0}^2 + \sigma_{x_1}^2}) \end{aligned} \tag{2.7}$$

$I(X; X_1, X_2)$ is information of random variable X that P_2 can learn from known X_1 and X_2. X_1 is independent from X, and due to Equation 2.4 such that:

$$\begin{aligned} I(X; X_1, X_2) &= I(X; X_2) \\ &\leq \frac{1}{2}\log(1 + \frac{\sigma_x^2}{\sigma_{x_0}^2 + \sigma_{x_1}^2}) \end{aligned} \tag{2.8}$$

There are information leakage upper bound for P_0 and P_2, with reasonable selection of large enough σ_{x_0} and σ_{x_1} such information leakage can approach 0. The entropy errors analysis of mutual information estimation considering the discrete fields and floating point errors in computer system, are the same with the analysis in Section 2.2.5 because

$$I(X; X_0, X_1) = 0 \tag{2.9}$$

$$I(X; X_0, X_2) = I(X; X_2) \tag{2.10}$$

$$I(X; X_1, X_2) = I(X; X_2) \tag{2.11}$$

2.3 Deep Learning

In this chapter, we introduce the history of deep learning, specifically focusing on Convolutional Neural Networks (CNNs). We then delve into the detailed implementation of deep learning models for the SGXDL and HybridSGXDL frameworks. Given the absence of existing CNNs C projects suitable for Intel SGX computation, we undertook the significant task of manually writing the code using standard C libraries, involving extensive manual work and mathematical induction. This chapter aims to highlight the implementation details essential for reproducing the results and understanding the code presented in the Appendix of this thesis.

Table 2.2.2: Proposed HybridSGXDL secure FL

Secure HybridSGXDL weighted Federated Learning
1) Data Split: m random secret shares of one sample are distributed to m Parties $P_0, P_1, ...P_{m-1}$. P_0 is the only server owning Intel SGX.
2) Local Computation: Each party generates random shares of the initialized parameters of Convolutional Neural Networks (CNNs) locally, following a Gaussian distribution. The parties then perform linear convolution operations using these weight shares and data shares, with computations accelerated by GPUs.
3) SGX Aggregation: Local computations, performed using a secret-sharing scheme, are securely transmitted to the SGX enclave. Within the SGX, these shares are integrated and converted back into plain text. The HybridSGXDL framework subsequently performs non-linear computations on this plaintext data and returns the results in the form of secret shares to the participating parties. This iterative process continues until a complete round of model forward and backward computation is completed.

2.3.1 Overview of Deep Learning

The idea of artificial neurons was introduced by Warren McCulloch and Walter Pitts in 1943, based on electrical circuits[68]. Since then, many researchers have advanced the field of neural networks, including Frank Rosenblatt, who developed practical applications like the Perceptron[82]. Research on theoretical aspects, including the backpropagation (invented in the $1970s$ and popularized in the $1980s$)[103], enabled the training of multi-layer neural networks[91].

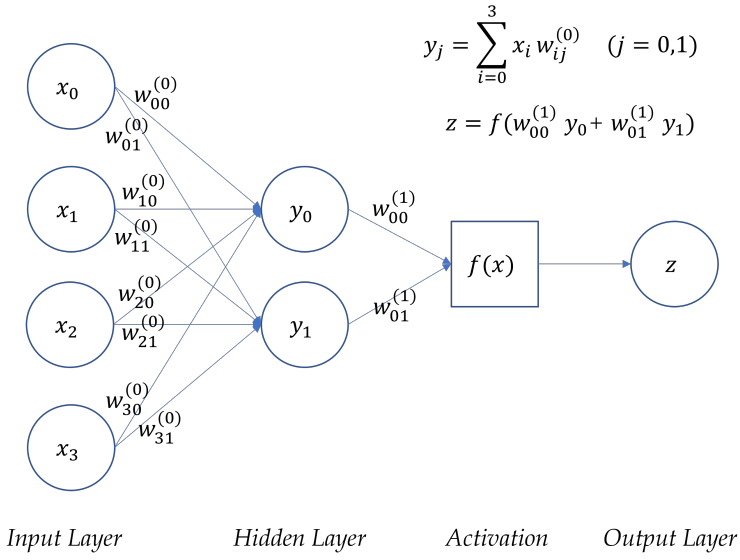

Figure 2.3.1: Simple Neural Network Structure

During the $1990s$ and early $2000s$, AI experienced a ”winter” period due to data scarcity, overfitting, gradient vanishing in deeper layers, and limited computational power. However, from 2015 onwards, the availability of GPUs[66] and large datasets facilitated more practical applications, leading to significant breakthroughs in deep learning structures.

One of the important structures that improved the efficiency and normalization ability of neural networks is the convolutional layer, proposed by LeCun et al.[55] and successfully applied in LeNet. The convolutional layer became a fundamental component of modern deep learning, contributing to major successes in image recognition competitions, such as AlexNet [51], which won the ImageNet Competition in 2012.

Deep learning has developed rapidly, and the appearances of advanced techniques like ResNet[39], DenseNet[41], and Transformer[96] models brought significant breakthroughs in DL performance. Especially Transformer advancements have enabled multi-modal applications and impressive applications in natural language processing(NLP) like ChatGPT[1]. Today, deep learning is recognized as one of the most powerful artificial intelligence paradigms, driving a wide range of applications in healthcare, finance, customer service, autonomous systems, and more.

2.3.2 Training phase of Deep Learning

Convolution Neural Networks (CNNs) involve a series of repeated operations, such as convolutions, followed by non-linear operations, including ReLU (Rectified Linear Unit) activations and MaxPooling. The training phase in deep learning is computationally intensive, requiring the model to process and learn all the samples in the large datasets comprehensively, identify the optimal optimization extreme for model parameters' values. We implemented the pure C-style deep learning models based on Intel SGX's standard C libraries, and the learning rate sets to fix the vanish gradients can see Appendix A.2.

2.3.3 Linear

The addition and multiplication computations are linear operations. In the context of Convolutional Neural Networks (CNNs), these operations are equivalent to the matrix multiplication and accumulation processes.

2D Convolution

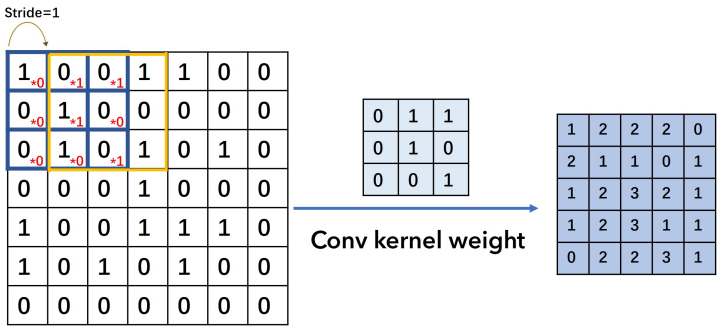

Figure 2.3.2: Convolution

Forward function, the convolution $Conv(w, x)$ is:

$$Conv[i_1, i_2, ..., i_D, c_{\text{out}}] = \sum_{c_{\text{in}}=1}^{C_{\text{in}}} \sum_{k_1=1}^{K_1} \sum_{k_2=1}^{K_2} ... \sum_{k_D=1}^{K_D} \mathbf{w}[k_1, k_2, ..., k_D, c_{\text{in}}, c_{\text{out}}] \cdot \mathbf{x}[i_1 - k_1, ..., i_D - k_D, c_{\text{in}}] \tag{2.12}$$

- $\mathbf{x}[i_1, i_2, \ldots, i_D, c_{\text{in}}]$ is the element of the input tensor $\mathbf{x}$ at channel c_{in}, located at position $(i_1, i_2, \ldots, i_D)$.
- $\mathbf{w}[k_1, k_2, \ldots, k_D, c_{\text{in}}, c_{\text{out}}]$ is the weight of the convolution kernel $\mathbf{w}$ at position $(k_1, k_2, \ldots, k_D)$, which connects the input channel c_{in} and the output channel c_{out}.
- D is the number of dimensions of the input tensor (e.g., 2D, 3D, 4D, etc.),
- $K_1, K_2, \ldots, K_D$ are the sizes of the convolution kernel along each dimension.
- C_{in} is the number of input channels (e.g., 3 channels for an RGB image), and C_{out} is the number of output channels.

Assume the input matrix $x^{C \times I \times I} \in \mathbf{R}$, the output $y^{O \times N \times N} \in \mathbf{R}$, and a weight $w^{C \times O \times M \times M} \in \mathbf{R}$. The Stride $S = 1$, Padding $P = 0$. C is the channels magnitude of input matrix x, and O is the channels magnitude of output matrix y. The weights w are the convolution kernels of the CNN. Padding is a technique that adds empty borders to the input image to prevent significant reduction in the output size. The stride is a parameter that specifies the number of pixels by which the filter matrix moves across the input matrix during the convolution operation.

$$N = \frac{I - M + 2 * P}{S} + 1 \tag{2.13}$$

N is the width and height of the output matrix y

$$\begin{aligned} y &= Conv(w, x) \\ y(o, i, j) &= \sum_{c=0}^{C-1} \sum_{p_1, p_2=0}^{M-1} w(c, o, p_1, p_2) * x(c, o, i + p_1, j + p_2) \end{aligned} \tag{2.14}$$

Backward function. Assume the loss function L. Loss functions in deep learning are used to measure how well a neural network model performs. L is the error between the final output of a CNNs model and given expected output value. The goal of training is to minimize the value of the loss function during the back propagation step in order to make the neural network better. While L in this thesis is the softmax function [44], which is effective for multi-class classification problems (details provided in Appendix B.1), the gradient update equations(Equation2.15, 2.16) apply universally to any loss function. Exploring alternative loss functions in future work may enhance performance. Then the gradients of L versus w for update is:

$$\begin{aligned} \frac{\partial L}{\partial w}(c, o, :, :) &= Conv(\frac{\partial L}{\partial y}(o, :, :), x(c, :, :)) \\ \frac{\partial L}{\partial w(c, o, p_1, p_2)} &= \sum_{i=0, j=0}^{N-1} \frac{\partial L}{\partial y(o, i, j)} \frac{\partial y(o, i, j)}{\partial w(c, o, p_1, p_2)} \\ &= \sum_{i=0, j=0}^{N-1} \frac{\partial L}{\partial y(o, i, j)} x(c, i + p_1, j + p_2) \end{aligned} \tag{2.15}$$

where $\frac{\partial y(o,i,j)}{\partial w(c,o,p_1,p_2)} = x(c, i + p_1, j + p_2)$ is the input matrix value, $\frac{\partial L}{\partial y(o,i,j)}$ is the loss gradients versus convolution's estimated output $y(o, i, j)$, which can be transmitted in the back propagation process from the final layer to the previous layers. $\frac{\partial L}{\partial w}(c, o, :, :)$ denote that the gradient is calculated for the weight associated with input channel c, output channel o. $\frac{\partial L}{\partial y}(o, :, :)$ represents the gradient of the loss with respect to the output y of the convolution for the $o - th$ channel. $x(c, :, :)$ is the input to the convolution layer for the $c - th$ channel.

We can also find that the backward of convolution for CNNs weight is also a linear convolution computation indeed. These gradients are propagated backward through the network, enabling the adjustment of weights from the final layer to the preceding layers. In a CNN, the output of one layer serves as the input to the next layer, making it essential to compute the output at each layer in order to effectively propagate the

gradients back and update the model parameters accordingly:

$$\begin{aligned}\frac{\partial L}{\partial x(c,i',j')} &= \sum_{o=0}^{O-1}\sum_{i=0,j=0}^{N-1}\frac{\partial L}{\partial y(o,i,j)}\frac{\partial y(o,i,j)}{\partial x(c,i',j')} \\ &= \sum_{o=0}^{O-1}\sum_{i=0,j=0}^{N-1}\frac{\partial L}{\partial y(o,i,j)}w(c,o,i'-i,j'-j)\end{aligned} \tag{2.16}$$

where $\frac{\partial y(o,i,j)}{\partial x(c,i',j')} = w(c,o,i'-i,j'-j)$ is the value of CNNs model's current weight w. For the input tensor x, the backpropagation of the convolution calculates the gradient of the loss function with respect to the input. The formula is similar to convolution forward, using the output gradient and the convolution kernel.

Algorithm 1 ConvolutionForward Appendix: A.1.1.
weight.width is the width of kernel, weight.height is the height of kernel, InputLayer.channels is the number of channels for input matrix. The same applies to other parameters.

Input: 4-dimensional **input** with dimensions of [batchsize, image.channels, image.width, image.height],
4-dimensional **weight** with dimensions of [InputLayer.channels, OutputLayer.channels, weight.width, weight.height],
1-dimensional **bias**,
activation $f(x)$.

Output: **output** with dimensions of [batchsize, OutputLayer.channels, output.Width, output.Height]

Conv Computation:

for $b = 1 : batchsize$ **do**
  for $i = 1 : InputLayer.channels$ **do**
    for $j = 1 : OutputLayer.channels$ **do**
      for $o_0 = 1 : output.width, o_1 = 1 : output.height$ **do**
        for $w_0 = 1 : weight.width, w_1 = 1 : weight.height$ **do**
          output$[b][j][o_0][o_1] += $ **input**$[b][i][o_0 + w_0][o_1 + w_1] *$ **weight**$[i][j][w_0][w_1]$
        end for
      end for
    end for
  end for
end for

Bias Computation:

for b=1:batchsize **do**
  for c=1:output.count **do**
    output[b][c]=$f(output[b][c]) + bias[c]$ /* output.count is the number of elements for each sample in a batch*/
  end for
end for

2.3.4 Non-Linear Function

ReLU

Forward Equation, where $x \in \mathbf{R}$

$$ReLU(x) = \begin{cases} x, x \geq 0 \\ 0, x < 0 \end{cases} \tag{2.17}$$

Backward Equation, where $x \in \mathbf{R}$ the gradient of ReLU versus the input x is:

$$\frac{\partial ReLU(x)}{\partial x} = \begin{cases} 1, x \geq 0 \\ 0, x < 0 \end{cases} \tag{2.18}$$

MaxPool

Forward Equation Assume the input matrix $x^{C \times I \times I}$ for maxpool, and the maxpool output of the input $y^{C \times N \times N}$, and the kernel size of maxpool $k_{maxpool} \in \mathbf{N}$, stride $S = k_{maxpool}$ set in the thesis, Padding is 0. the MaxPooling operation is applied to the spatial dimensions (height and width) for each channel. In the forward pass, the layer slides a fixed-size window over the input and selects the maximum value within each window. This reduces the spatial resolution of the feature maps while retaining the strongest activations. The maxpool forward can be described as:

$$y(c, i, j) = \max_{p_1, p_2} \{x(c, i \cdot S + p_1, j \cdot S + p_2)\} \tag{2.19}$$

where p_1 and p_2 are the dimensions of the pooling window.

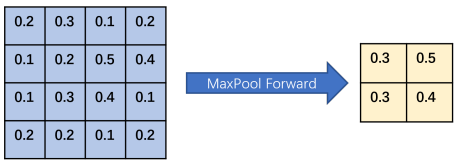

Figure 2.3.3: MaxPool Forward, Stride=2, Padding=0, $k_m = 2$

Backward Equation In the backward pass, the gradient is propagated only through the locations in the input matrix x where the maximum values were selected during the forward pass. The MaxPooling layer itself does not have any learnable parameters, but the backward operation ensures that only the relevant positions (those

with the maximum values in the forward pass) receive the gradient. The bcakward of maxpool can be described as:

$$\frac{\partial y(c,i,j)}{\partial x(c,i',j')} = \begin{cases} 1, & \text{if } x(c,i',j') \text{ was the maximum in its pooling window} \\ 0, & \text{otherwise} \end{cases} \tag{2.20}$$

- $\frac{\partial y(c,i,j)}{\partial x(c,i',j')}$ is the gradient of the MaxPooling output y with respect to the input x.
- i', j' represent the spatial location in the input matrix.
- The gradient is non-zero only if $x(c,i',j')$ corresponds to the maximum value in the pooling window during the forward pass.

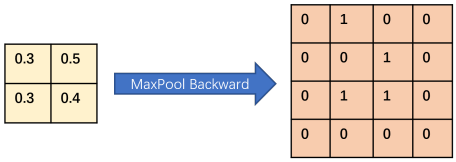

Figure 2.3.4: MaxPool Backward

MaxPool is a typical non-linear operation that is expensive in MPC, unlike average pooling. This thesis applies a "MaxPool-ReLU" block throughout the network to demonstrate efficiency improvements more clearly. Comparison operations in MPC require two rounds of communication, making MaxPool and ReLU quite costly[99].

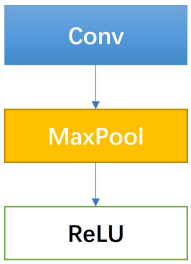

Figure 2.3.5: "MaxPool-ReLU" Block

2.4 Intel SGX

Intel Software Guard Extensions (SGX) is a set of security-related instruction codes built into modern Intel processors. Intel proposed SGX idea and published on HASP conference in 2013[69], introduced a physical CPU(Skylake) supporting SGX at 2016[112]. SGX provides an encrypted execution environment which is a physical

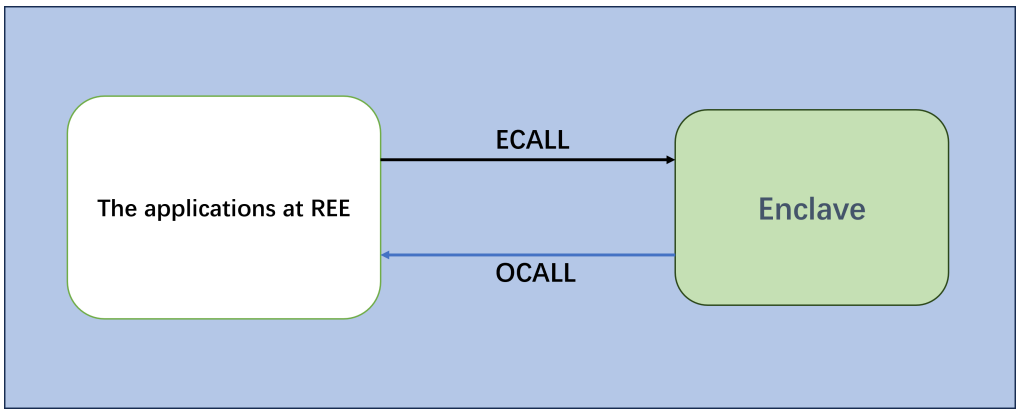

Figure 2.4.1: Intel SGX Interface Calls

area called enclave on CPU, protects and prevents the code and data from the outside environment. These enclaves are designed to protect sensitive data and computations from unauthorized access, even if the operating system or other system software is compromised.

- **ECall**: ECALLs are interface functions running within the enclave, defined in .edl files, and called by untrusted applications from the Rich Execution Environment (REE), which is untrusted. Data transfer is implemented by passing pointers and reading bytes; the enclave creates a copy of the data inside. When the execution of an ECALL is finished, the code continues to run the next REE ECALL functions, but the enclave remains unchanged until the next ECALL or the enclave is closed.

- **OCall**: OCALLs are untrusted interfaces used to call applications running in the REE from within the enclave. The usage of OCALLs requires interactions between SGX and the untrusted environment, exposing the system to potential side-channel attacks by observing the interactions It is crucial to minimize the usage of OCALLs as much as possible to enhance security.

In this thesis, we utilize Intel SGX hardware security to execute plain-text computations for deep learning instead of cryptographic or shared computations, reducing communication and computation overhead. A practical and important detail of Intel SGX libraries is that they forbid most C++ functions and some C libraries, including the timing library, for security controls. Consequently, all functions executed are basically in pure C style. Implementing the desired functionality from scratch using C standard libraries is crucial for using Intel SGX.

The deep learning models in this thesis are implemented based on C standard libraries. This requires knowledge of linear algebra, mathematical analysis, differential matrices, extreme points, derivative chain induction, and optimization theory, as well as the entire computational process of CNNs. These are complex topics for engineering programmers who want to use Intel SGX directly. The infrastructure provided in this thesis implements CNN models that programmers can use for secure training with Intel SGX without the requirement to understand the intricate details of CNN implementation, which involves sophisticated mathematical induction and implementation.

2.5 Hybrid AES-RSA Secure Channel

Asymmetric encryption algorithms, such as RSA, are inefficient for encrypting large data due to their low performance. Conversely, symmetric encryption algorithms like Advanced Encryption Standard (AES), while efficient, present significant key management challenges because the key must be securely shared between parties [37]. Research on hybrid AES-RSA algorithms validates their practicality and effectiveness in resolving these issues [45, 53, 62]. This thesis employs the AES-RSA hybrid encryption scheme to establish a secure communication channel. Future work can explore newer and more efficient methods to further enhance security of secure channel.

Advanced Encryption Standard (AES) AES is a symmetric encryption algorithm established by NIST in 2001, widely used for securing sensitive data due to its efficiency and robustness.

Rivest–Shamir–Adleman (RSA) RSA, developed by Rivest, Shamir, and Adleman in 1977, is an asymmetric encryption technique. While RSA is computationally intensive and impractical for large data encryption, it is ideal for securely transmitting symmetric keys.

The AES-RSA hybrid channel schedule is as follows:

1. Generate a random AES key.
2. Use AES key to encrypt data.

3. Sender encrypts the AES key using RSA public key
4. Receiver decrypts the AES key using RSA private key.
5. Decrypt the data using the AES key.

In this hybrid encryption scheme, AES encrypts the data, and RSA encrypts the AES symmetric key. This combination ensures efficient data encryption and secure key exchange, providing a robust solution for secure communication channels.

Chapter 3

SGXDL

3.1 Motivation for SGXDL

The core idea of SGXDL is to accommodate the entire CNN training process within the SGX enclave, ensuring that training completes inside the enclave without gradients being transferred outside. Consequently, CNN models remain protected and inaccessible to external entities. Initially, SGX's enclave capacity was limited to 128 MB, insufficient for the entire CNN model and training process. Today, the enclave capacity has increased to 512 GB, sufficient for even large language models(LLMs). Inspired by this progress, we propose executing all CNNs computations in the SGX, presenting a potential path for leveraging hardware security and GPU-TEEs to address privacy concerns.

While deep learning computations have seen significant efficiency improvements with GPUs, plain-text calculations within the enclave on the CPU offer three key advantages:

- Compared to other secure and cryptographic techniques, the SGX enclave's plain-text deep learning calculation efficiency outperforms GPU-assisted encrypted calculations, as examined with models like LeNet, AlexNet, ResNet18, and VGG16. The efficiency gap increases with larger model sizes. In this thesis, we benchmark the computing performance together with the known GPU assisted MPC platforms Piranha(efficiency winner among SecureML [73], Falcon [98], and FantasticFour [18]).

- Wu et al. [105] proposed GPU-TEEs, introducing GPU acceleration for Intel SGX. Our methodology embeds the entire CNN training within the SGX enclave, leveraging hardware security for privacy computations. SGX plain-text computation already outperforms other secure techniques, and its efficiency can further improve with GPU-TEEs, making plain-text deep learning execution within the enclave practical. SGXDL explores running all deep computations within SGX and prepares for the potential future use of GPU-enabled SGX.
- State-of-art MPC protocols, typically involving 2 to 4 parties, suffer from increased communication overhead, computation workload, and memory requirements as the number of parties increases, making them impractical on current servers. For instance, Piranha's 3-party memory requirement for ResNet18 exceeds the capacity of 24GB GPUs. SGXDL handles the complete process of training, with communication only at the beginning of each training batch when all parties simultaneously send shares securely to the SGX, then the enclave integrates these shares into plain-text within the enclave. Consequently, execution time and performance are minimally influenced by the number of participating parties, unlike state-of-art MPC protocols.

3.2 The structure of SGXDL

3.2.1 Data shares distribution

The data are stored in n parties distributedly in a secret-share scheme as follows(Figure 3.2.2). Assume the original data sample x. Generate the random numbers as the data shares $x_0, x_1, x_2, ..., x_{n-2}$(Section 2.2.4), the share x_{n-1} satisfies $x_{n-1} = x - \sum_{k=0}^{n-2} x_k$. The dataset and labels used for training would be sent in this way to the SGX enclave.

3.2.2 Secure Channel and Data Transfer initialization

For the server hosting the SGX enclave, although the method of share generation is unknown, it receives all the shares from the participating parties. This poses a risk because the server could potentially reconstruct the original data sample from these shares. To mitigate this risk, additional protection of the shares is necessary. Given the high computational cost of RSA asymmetric encryption, it is impractical to encrypt

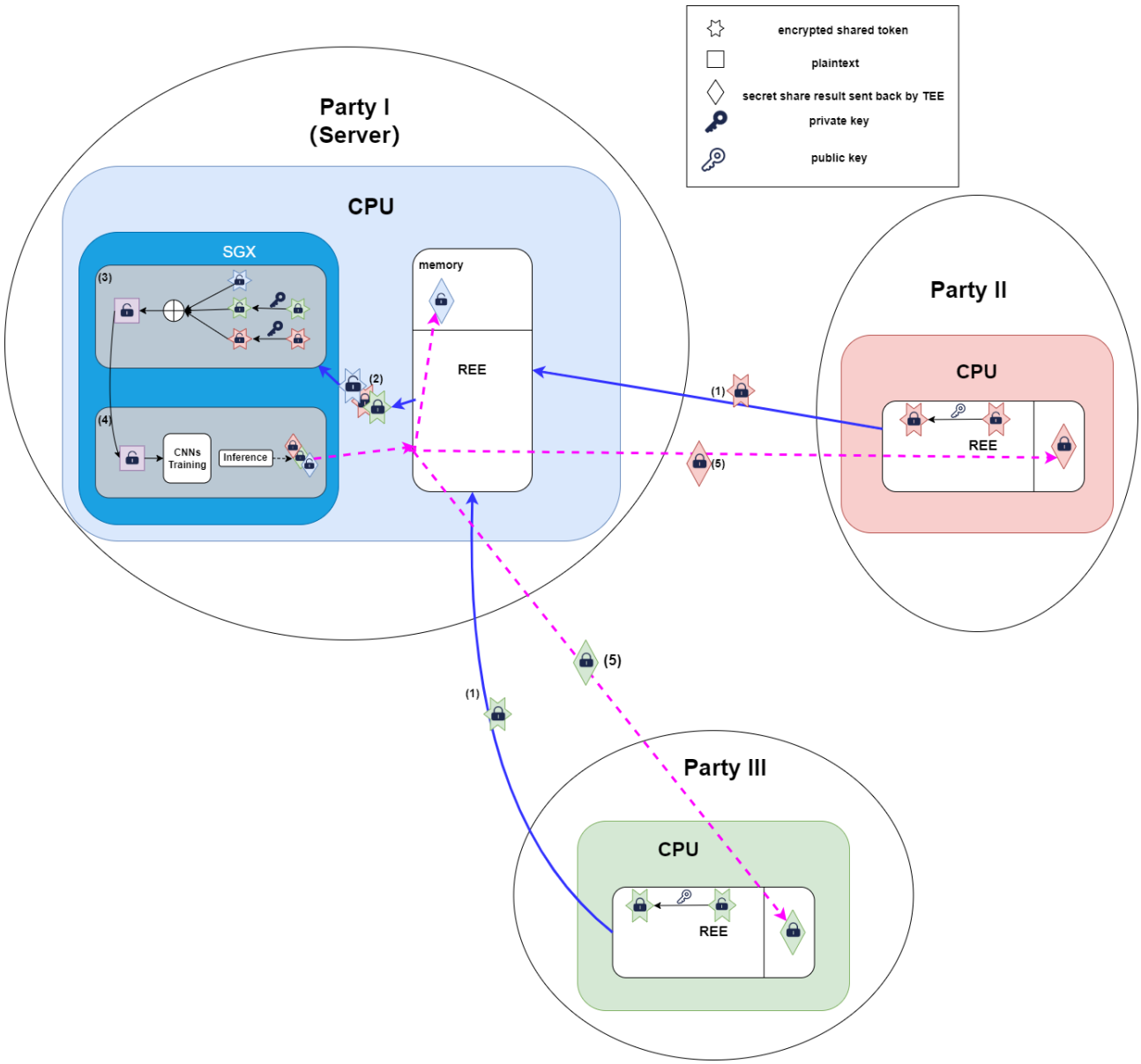

Figure 3.2.1: SGXDL secure computing structure. The blue connection indicates the shares transmission process from the parties to SGX for completing the training, and the hexgram symbol denotes the data shares transmitted from the parties to the SGX. The pink connection represents the computing results in secret-sharing format back to the parties. The shares would be retrieved to the plain-text format in the SGX enclave.

the dataset directly using RSA. Instead, RSA will be used to protect an AES symmetric key, as described in Section 2.5.

A secure transmission channel is established to achieve this. Locally, shares are encrypted using the AES key and this AES key is then encrypted using the RSA public key. Once the shares are transmitted to the enclave, the AES key is decrypted using the RSA private key within the enclave to recover the share values. This method ensures that the shares are securely integrated into plaintext values and remaining inaccessible

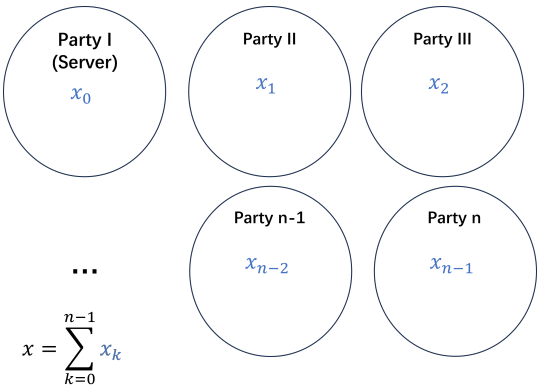

Figure 3.2.2: Data shares distribution

to the server. Most examples in the following modules will use a three-party scenario for illustration.

Public Key and Private Key distribution

At initialization, the enclave generates the RSA key pairs for the secure transmission channel. Since the server holds and knows the value of shares distributed to it, it is not necessary for the enclave to encrypt the share that held by the server. The enclave sends the public key to all the client parties and keeps the private key securely within the enclave.

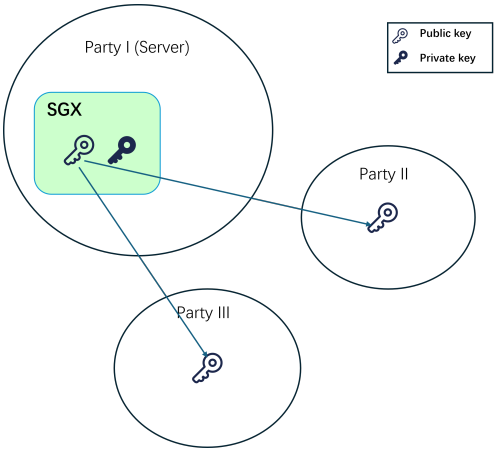

Figure 3.2.3: RSA Public Key Exchange at Initialization(In case of 3 party)

Secure Data Transmit and Integration

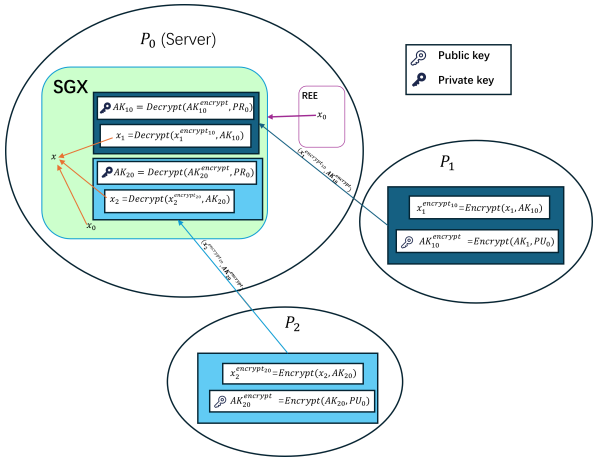

Figure 3.2.4: SGXDL secure data transmission channel

Assume the RSA public key PU, and the private key PR. $Encrypt(C, K)$ represents using the key K to encrypt the context C, and the $Decrypt(C, K)$ represnt using the key K to decrypt the context C. Assume the share x_k of data sample held by the party P_i, the random AES key $AK_{ij} \in \mathbf{R}$ are generated by party i and encrypts data sent to party j. Define the RSA key pairs PU_i and PR_i are generated by party P_i.

In the local party i

$$x_k^{encrypt_{ij}} = Encrypt(x_k, AK_{ij})$$

where $x_k^{encrypt_{ij}}$ is the encryption value for $k-th$ share of x with AES key AK_{ij}, and will be sent from party P_i to P_j.

$$AK_{ij}^{encrypt} = Encrypt(AK_{ij}, PU_j)$$

where $AK_{ij}^{encrypt}$ is the encryption of symmetric key AK_{ij} by public key P_i. And in the enclave, the symmetric key and the share would be decrypted that:

$$AK_{ij} = Decrypt(AK_{ij}^{encrypt}, PR_j)$$

$$x_k = Decrypt(x_k^{encrypt_{ij}}, AK_{ij})$$

The party we send the $(x_k^{encrypt_{ij}}, AK_{ij}^{encrypt})$ to the SGX, and in the sgx enclave, the encrypted symmetric key $AK_{ij}^{encrypt}$ is decrypted to AK_{ij} using the private key P_j. And

the encrypted shares would be decrypted to the original shares using the AK_{ij}.

$$x_k = Decrypt(x_k^{encrypt_{ij}}, Decrypt(AK_{ij}^{encrypt}, PR_j))$$

The the data used for training in enclave can be integrated in the enclave:

$$x = \sum_{k=0}^{n-1} x_k$$

Then the plain-text CNNs training can be executed with the plain-text dataset $\{x\}$ in the secure enclave.

3.3 Task order and Synchronization

Figure 3.3.1 illustrates the SGXDL workflow throughout the entire training process. Since the whole training process occurs within the enclave, the training can begin as soon as the SGX receives the shares of a batch of samples. The SGX has the flexibility to start training either after receiving a batch of data sample shares or after receiving all data shares.

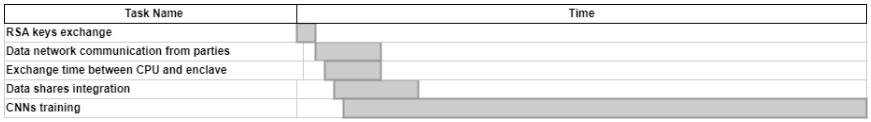

Figure 3.3.1: SGXDL task order and synchronization. The diagram describing SGXDL synchronization illustrates the workflow of SGXDL. Section 6.3.1 shows that during training, the exchange time between the CPU and the enclave, as well as the data sharing integration overhead, is almost zero. The rectangular task time blocks only represents a schedule of tasks' orders and does not reflect accurate time.

3.4 Summary

- The core idea of SGXDL is to perform plain-text CNN training entirely within the enclave, using a secret-sharing scheme to ensure that the attacker can not extract useful information without compromising all parties.
- The model and gradients remain within the enclave throughout the whole training process, protecting them from common federated learning attacks such

as membership inference and gradient attacks.

- During training, SGX communicates with the outside environment only at the beginning of training for dataset shares transmission, minimize the risk of side-channel attacks by limiting observable communication.
- Secure training in the enclave relies on CPU capability; hardware-secured plain-text computation has been shown to outperform other cryptographic techniques in efficiency, as demonstrated in Chapter 6. In the future, GPU-TEEs research could further enhance the efficiency of training complete plain-text models within the enclave.

Chapter 4

HybridSGXDL

To address privacy and efficiency challenges and leverage GPU advantages for secure deep learning in real-world scenarios, HybridSGXDL is proposed as a secure computing architecture. The operations would be separated to linear and non-linear operations with the assistance of Intel SGX. This architecture uses GPUs to accelerate the local linear computation which are in secret sharing format. Non-linear operations are executed in the enclave in plain-text. The communication between parties and the server would happen at the end of each linear and non-linear computation, to share the results for the next operations and complete the training together.

Unlike SGXDL, which keeps all computations within the enclave, HybridSGXDL expects devices to handle linear distributed computation with GPUs, secret-sharing the model weights and data among parties. CNNs training starts from random weights initialization, the initialization of CNNs model weight shares would be generated by all parties in a random manner, that no party knows the actual initial model weight values. The server can only receive the shares of intermediate computing results during the training.

CNN modules typically consist of linear operations followed by non-linear operations. Linear operations, such as addition and multiplication, are performed locally with GPU acceleration, while non-linear operations, such as ReLUs and MaxPool, are executed in plain-text within the Intel SGX enclave. The enclave acts as a trusted third party, integrating intermediate results from the linear operations. After completing the non-linear computations, the server returns the results in data shares format to the parties for the next linear convolution computation.

The HybridSGXDL implementation in this thesis is demonstrated under a 3-party computation scenario.

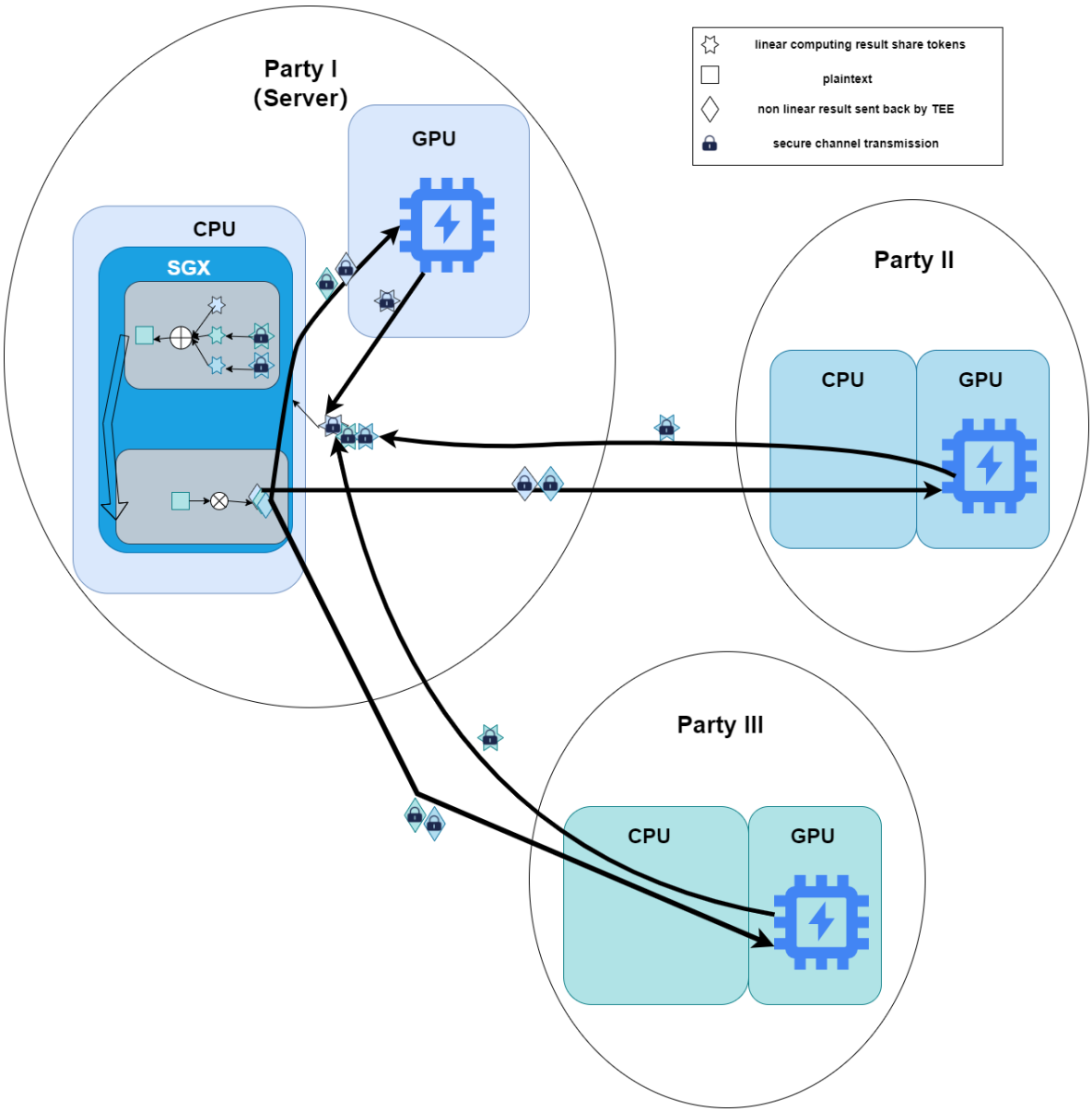

Figure 4.0.1: Hybrid safe computing structure

4.1 Model Weight Share Distribution

In HybridSGXDL, the model weights and dataset are distributed in a secret-sharing format. The weight updates are sent to the parties by the enclave, meaning the server has the potential to generate all the secret shares. The initialized weights are unknown to the parties, including the server, so even if in worst case an attacker hacks all weight update shares, the value of the weights remains unknown. Additionally,

adversaries must attack multiple participating parties to retrieve any potentially useful information, distributing the risks and significantly increasing the difficulty for attackers.

Since the server has the potential to generate all the secret shares, we apply the secure channel scheme described in Section 3.2.2 to protect these shares from the server.

4.1.1 Weight Share distribution

Assume the n shares are: $A = \{w_0, w_2, ..., w_{n-1}\}$, model's real value weight $w = \sum_{k=0}^{n-1} w_k$. The set A_i is the set of i-th party's holding shares, A_0 is the server party owing the SGX enclave, such that:

$$A_i = A \backslash \{w_{i+j \mod n}\}$$

At the initialization state, in the project set $j = 1$, then the random shares generation process would be:

- The server generate the n-1 shares randomly $\{w_i | w_i \in A_0\}$. At the same time the party P_1 generate the random share w_1 at the same time. (If $j \neq 1$, then it would be P_j creating w_j)
- The server send share set $\{A_0 \backslash \{w_{i+1 \mod n}\}\}$ to P_i, and P_1 sends share w_1 to all parties except P_0.
- The model initialization parameter weights have been created randomly, and one party have the information of the real values of model's weights.

The 3 party computation share distribution in the thesis

- $P_0 : (w_0, w_2)$
- $P_1 : (w_0, w_1)$
- $P_2 : (w_1, w_2)$

To randomly generate the model weight shares together by all the parties, and no party have useful information of the initialization weight real values. The 3-party condition would follow the steps:

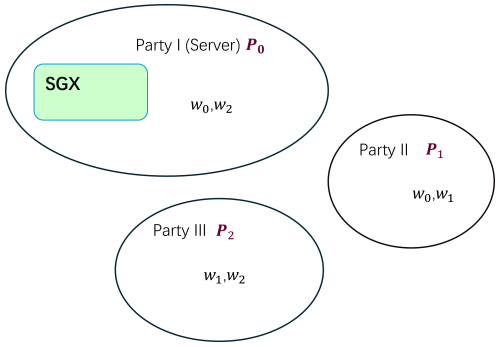

Figure 4.1.1: Weight share distribution for HybridSGXDL(3 Party)

- P_0 generate the random values w_0, w_2, P_1 generate the random value w_1 at the same time.
- P_0 send w_0 secret share to P_1, and w_2 to P_2. P_1 send w_1 to P_2 at the same time.

Then the real value of model initialization weights is: $w = w_0 + w_1 + w_2$, but no party knows this value, and in the following computation the model's weight would consent to be unknown for all the parties including the SGX enclave.

4.1.2 Public key and Private key Exchange for secure channel

Section 3.2.2 demonstrates the secure channel of clients sending shares to the SGX. Since the random initialization and transmission only requires the server P_0 to other parties P_1, P_2, and $P_1 \rightarrow P_2$. Only P_1 and P_2 needs create the RSA key pairs, the process is:

- P_1 and P_2 create the RSA public and private key pairs locally.
- P_1 and P_2 send their public keys to the server P_0, and P_2 sends out its public key to P_1 at the same time.

4.1.3 Secure Data Transmission and Integration

After defining the transmission flow in HybridSGXDL, the secure channel to prevent the server from potentially receiving all shares of intermediate products directly is the same as with SGXDL (Section 3.2.2). This involves using RSA key pairs to generate a symmetric AES key for efficient and secure transmission. The secure transmission workflows for HybridSGXDL are illustrated in Figure 4.1.3 and Figure 4.3.4.

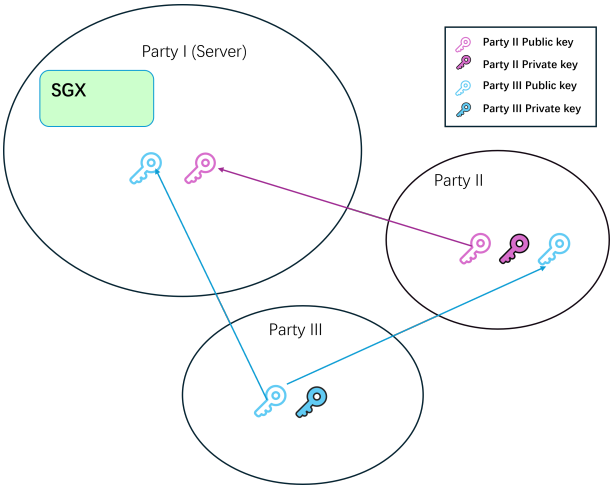

Figure 4.1.2: Public Key Exchange of Initialization for model weight shares(In case of 3 parties)

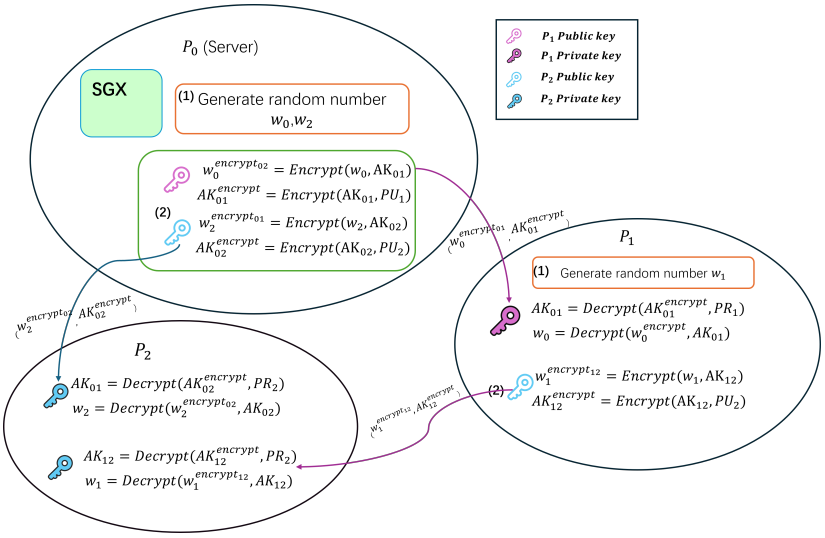

Figure 4.1.3: Model weight random initialization(In case of 3 parties)

4.2 Training Dataset Share Distribution

The training dataset X is distributed to the parties in the same secret-sharing scheme as the model weight w. Assume the dataset shares $X = \{x_0, x_1, ..., x_{n-1}\}(x = \sum_{k=0}^{n-1} x_k)$. The training dataset shares have the same share distribution as model weights, meaning P_i holds weight shares $A\backslash\{w_{i+1 \mod n}\}$and $X\backslash\{x_{i+1 \mod n}\}$.

For this thesis, we assume that the dataset shares are initially distributed to the parties by the individual data owners. Unlike institutions with predictable working hours and large datasets, individual data transmission is sporadic and unpredictable, making it difficult for attackers to gain useful information. Consequently, we do no not consider additional protection mechanism beyond this inherent unpredictability.

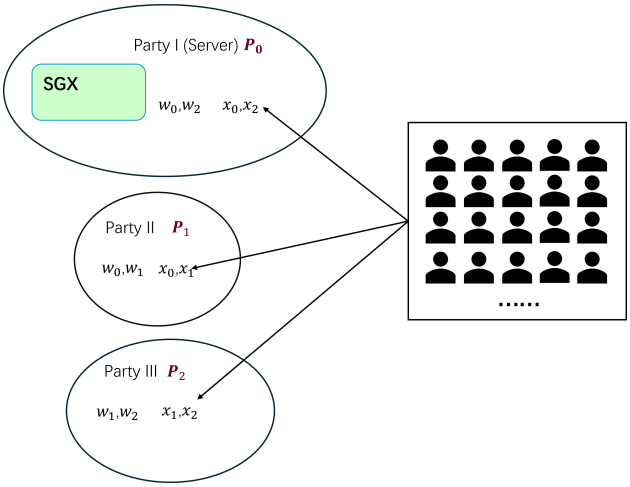

Figure 4.2.1: Dataset shares distribution

4.3 Linear and Non-Linear Computation for CNNs

CNNs typically consist of repeating modules, such as convolution followed by ReLUs and convolution followed by MaxPool. ReLUs and MaxPool are non-linear operations in MPC protocols that involve significant communication overhead, and common methods using polynomials to approximate these non-linear operations are also computationally intensive. HybridSGXDL separates the computation of CNNs into linear and non-linear operations.

Linear operations, such as addition and multiplication, are executed on local parties' GPUs, leveraging GPU acceleration. These operations follow the same flow as plain-text linear computations and maintain the same structure of arithmetic addition and multiplication operations. Non-linear operations, such as ReLUs and MaxPool, are executed in the SGX enclave, where plain-text computation is protected by SGX's hardware security.

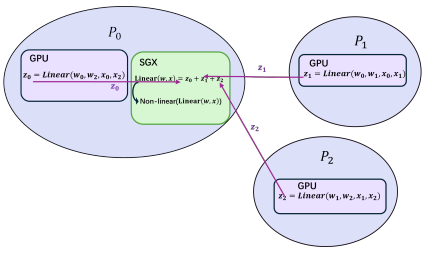

Figure 4.3.1: Non Linear and Linear operations Separation Computation

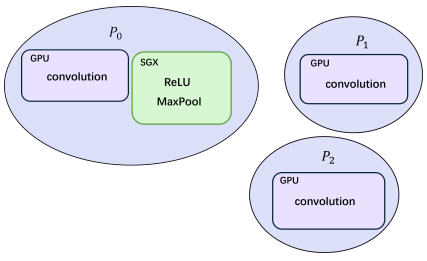

Figure 4.3.2: Non-Linear and Linear Operations

In the thesis, the repeating modules consist of convolution followed by MaxPool and

ReLU. In these modules, the linear operations are convolutions, and the non-linear operations are MaxPool and ReLU.

4.3.1 Linear Operations on local party's GPU

The linear computation method in HybridSGXDL would not ruin the results of linear operations(which only consists of multiplication and addition), in the following context we demonstrate the correctness of convolution forward result as an example. In the thesis we implemented and experimented the 3-party scenarios. All the following proofs are under the 3-party set-up. Assume the input image 3-dimensional matrix $x^{C\times I\times I}$, the convolution 3-dimensional output $y^{O\times N\times N}$(padding=0, stride=1), and the 4-dimensional convolution kernel $w^{C\times O\times M\times M}$. C is the channel number of the image, e.g. the number of channels for RGB image is 3. I is the width and height of the image matrix. O is the channels number of the output matrix. The magnitude of first dimension of w is the same with input image x's channels C. The convolution is:

$$Conv(w,x) : y(o,i,j) = \sum_{c=0}^{C-1} \sum_{p_1,p_2=0}^{M-1} w(c,o,p_1,p_2) * x(c,i+p_1,j+p_2) \tag{4.1}$$

Assume $w(c,o,p_1,p_2)_{u_0}$ is the weight share of party P_{u_0} for the matrix element $w(c,o,p_1,p_2)$. And the model weight shares $w = w_0 + w_1 + w_2$ that

$$w(c,o,p_1,p_2) = w(c,o,p_1,p_2)_0 + w(c,o,p_1,p_2)_1 + w(c,o,p_1,p_2)_2 \tag{4.2}$$

$$x(c,i+p_1,j+p_2) = x(c,o,p_1,p_2)_0 + x(c,i+p_1,j+p_2)_1 + x(c,i+p_1,j+p_2)_2 \tag{4.3}$$

Where $x(c,i+p_1,j+p_2)_{u_0}$ is the weight share of party P_{u_0} for input image's element $x(c,i+p_1,j+p_2)$. Then the convolution would be:

$$y(o,i,j) = \sum_{c=0}^{C-1} \sum_{p_1,p_2=0}^{M-1} (w(c,o,p_1,p_2)_0 + w(c,o,p_1,p_2)_1 + w(c,o,p_1,p_2)_2) * x(c,i+p_1,j+p_2) \tag{4.4}$$

$$
\begin{aligned}
y(o,i,j) &= \sum_{c=0}^{C-1} \sum_{p_1,p_2=0}^{M-1} w(c,o,p_1,p_2) * x(c,i+p_1,j+p_2) \\
&= \sum_{c=0}^{C-1} \sum_{p_1,p_2=0}^{M-1} ((\sum_{u_0=0}^{2} w(c,o,p_1,p_2)_{u_0}) * (\sum_{u_1=0}^{2} x(c,i+p_1,j+p_2)_{u_1})) \\
&= \sum_{c=0}^{C-1} (\sum_{\substack{p_1,p_2=0 \\ (u_0,u_1)\in\{(0,0),(0,2),(2,0)\}}}^{M-1} w(c,o,p_1,p_2)_{u_0} x(c,i+p_1,j+p_2)_{u_1} \\
&\quad + \sum_{\substack{p_1,p_2=0 \\ (u_0,u_1)\in\{(1,1),(1,0),(0,1)\}}}^{M-1} w(c,o,p_1,p_2)_{u_0} x(c,i+p_1,j+p_2)_{u_1} \\
&\quad + \sum_{\substack{p_1,p_2=0 \\ (u_0,u_1)\in\{(2,2),(2,1),(1,2)\}}}^{M-1} w(c,o,p_1,p_2)_{u_0} x(c,i+p_1,j+p_2)_{u_1}) \\
&= z_0 + z_1 + z_2
\end{aligned} \tag{4.5}
$$

such that

$$
\begin{aligned}
z_0 &= \sum_{c=0}^{C-1} \sum_{\substack{p_1,p_2=0 \\ (u_0,u_1)\in\{(0,0),(0,2),(2,0)\}}}^{M-1} w(c,o,p_1,p_2)_{u_0} x(c,i+p_1,j+p_2)_{u_1} \\
&= Conv(w_0,x_0) + Conv(w_0,x_2) + Conv(w_2,x_0)
\end{aligned} \tag{4.6}
$$

$$
\begin{aligned}
z_1 &= \sum_{c=0}^{C-1} \sum_{\substack{p_1,p_2=0 \\ (u_0,u_1)\in\{(1,1),(1,0),(0,1)\}}}^{M-1} w(c,o,p_1,p_2)_{u_0} x(c,i+p_1,j+p_2)_{u_1} \\
&= Conv(w_1,x_1) + Conv(w_1,x_0) + Conv(w_0,x_1)
\end{aligned} \tag{4.7}
$$

$$
\begin{aligned}
z_2 &= \sum_{c=0}^{C-1} \sum_{\substack{p_1,p_2=0 \\ (u_0,u_1)\in\{(2,2),(2,1),(1,2)\}}}^{M-1} w(c,o,p_1,p_2)_{u_0} x(c,i+p_1,j+p_2)_{u_1} \\
&= Conv(w_2,x_2) + Conv(w_2,x_1) + Conv(w_1,x_2)
\end{aligned} \tag{4.8}
$$

For local parties, P_0 computes z_0, P_1 computes z_1, P_2 computes z_2 on their GPUs, and send this linear results to the SGX enclave can calculate the output $y(o,i,j)$ as shown in Figure 4.3.1. w or x can not be retrieved from these intermediate products, and the w weights and x data do not leave the party during the entire training process. The above is an illustration of how an element in a convolution result matrix are calculated in HybridSGXDL, demonstrating that this linear operation ensures the accuracy of the

final result in comparison with plain-text computation. In actual situations, the matrix is, of course, calculated and transmitted in batches.

We have demonstrated that the correctness of the forward convolution operation remains intact in HybridSGXDL. The backward propagation process adheres to the same separation rules for linear and non-linear operations. Backward's linear operations are computed locally on clients' GPUs, while backward's non-linear operations are performed within the Intel SGX enclave. Because the backward computation of linear operations is also linear. The backward propagation of non-linear operations remains non-linear and is executed secretly in plain-text within the enclave. The backward propagation of convolution is essentially a forward convolution, except that it operates different operands in comparison with convolution forward. For example, the backward convolution compute the gradients for updating the weights, its operands are the input matrix of forward computation and the gradient of output matrix versus loss function(as shown in Equation 2.15). Therefore, the correctness of the convolution's backward pass also has been mathematically validated through the forward as shown in Equation 4.5,4.6,4.7,4.8.

Secure Transmission for Linear Results Since the RSA keys from clients to the server have been distributed at the initialization as shown in Figure 3.2.3, we want to send the linear results securely to the enclave using this secure channel to prevent the server from receiving all the shares for the linear intermediate computation results. We always use the secure channels for communications.

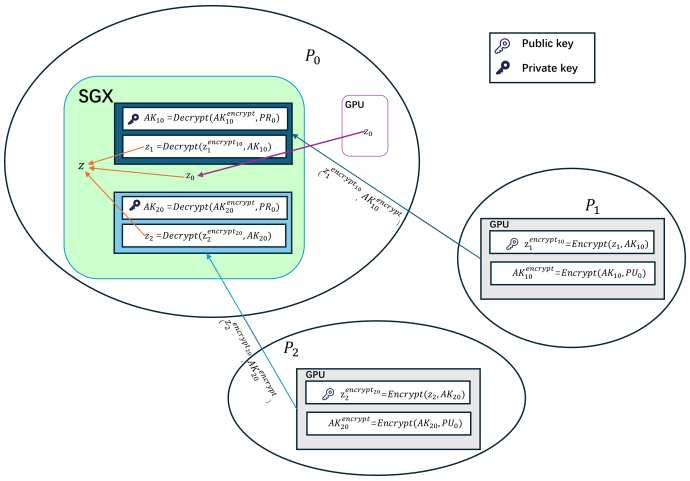

Figure 4.3.3: Secure Linear Results Computation

4.3.2 Non Linear Operations in enclave

In the enclave, the linear result shares are integrated into plain-text values. SGX hardware security prevents access from the outside environment. This keeps the computation process protected and secret within the SGX enclave. After completing the non-linear computation, the output would be separated into secret shares $\{o_0, o_1, o_2\}(ReLU(z) = o_0 + o_1 + o_2), zistheintegratedinput$, and sent back to parties' GPUs via the secure channel. These shares participate in the next convolution computation operations, continuing until the forward and backward computations of the CNNs are completed.

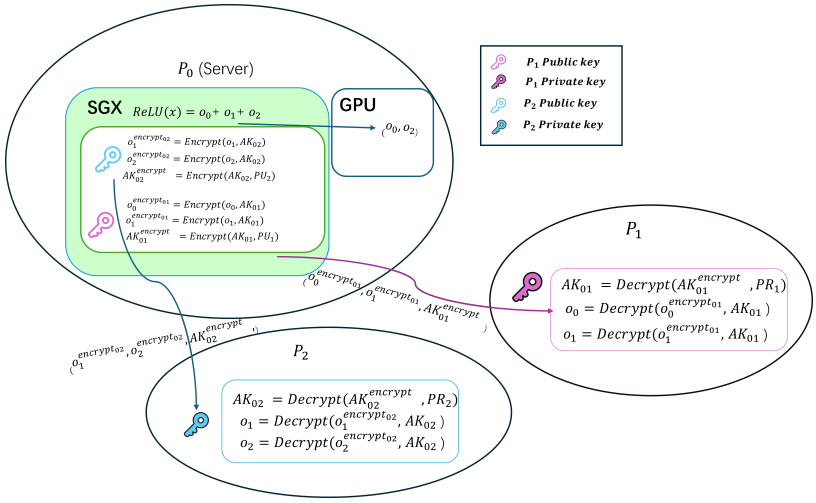

Figure 4.3.4: Secure Non Linear Results Transmission from Enclave

Though the server has known the real values of holding shares o_0, o_2, it is still necessary using secure channels for all communications to prevent the external adversaries.

4.4 Gradient Update

The model's weights w are updated at the end of each training. Assume the update is ∂w, separated into shares $\partial w = \partial w_0 + \partial w_1 + \partial w_2$ for updating the w_0, w_1, w_2, the secure transmission for the weight update workflow is the same with non-linear results transmission as shown in Figure 4.4.1.

The new model weight w' and the updated shares w_0', w_1', w_2' would be

$$w_{\prime} = w + \partial w \tag{4.9}$$

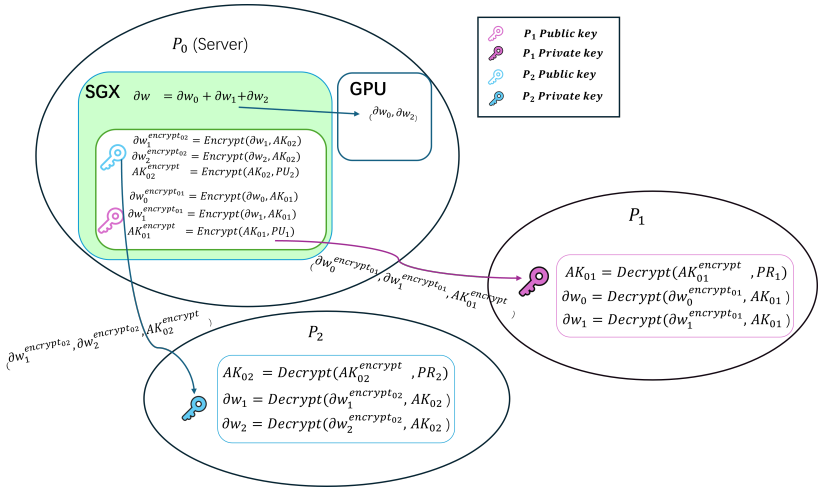

Figure 4.4.1: Gradient Shares Safe Transmission, AK_{ij} is the symmetric key used for encrypting the transmitted shares from party P_i to P_j.

$$w_0^{'} = w_0 + \partial w_0, w_1^{'} = w_1 + \partial w_1, w_2^{'} = w_2 + \partial w_2 \tag{4.10}$$

4.5 Task order and Synchronization

Figure 4.5.1 depicts the task order of a training batch for CNNs, we can notice that each computation layer in the CNNs waiting for the results from the last layer computation, that there is almost no synchronization for the further time optimization.

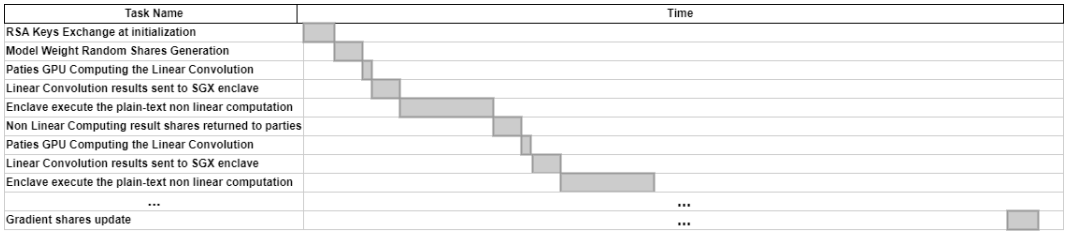

Figure 4.5.1: Task order and synchronization, the image depicts the task orders in a training batch and the synchronization condition. The rectangular task time blocks only represent a rough description for the task, and do not reflect the real time, in some cases, the plain-text non linear computation time can be very short and less than GPU linear computation time. Usually the network communication time is constant.

4.6 Summary

- HybridSGXDL separates CNN computations into linear and non-linear operations. Linear operations are computed distributedly in secret shares using the same logic as plain-text linear computations on local parties, enabling the

utilization of GPUs. Non-linear operations, which are computationally intensive in MPC cryptography protocols, are executed and protected in plain-text within the SGX enclave, ensuring secure and efficient computation.

- Unlike state-of-art federated learning (FL), HybridSGXDL does not require sharing model weights. Initialized weight shares are generated randomly by each party locally, ensuring that the integrated model weights are unknown to any party. Data shares and weights remain local during training convolution, and only the random multiplication intermediate product shares are sent out, protecting the data and weights from being revealed. Without access to model weights, common gradient attacks in federated learning, such as membership inference, are not applicable.
- The shares exchanges occur through a secure transmission channel, using hybrid AES-RSA secure channel for fast encryption and decryption.
- Data sample shares are sent directly by individuals to the computing parties, ensuring that no party knows the data sample values. This increases security for individuals and enables training on private datasets that they are unwilling to share with institutions. Additionally, the discrete and unpredictable communication from individuals makes the condition more complex for attackers and reduces the risk of data leakage, as discrete intercepted message contains minimal information.

Chapter 5

The Deep Learning Models in the Implementation

This section provides details of the deep learning models enabled by the secure computing infrastructures, essential for understanding the execution efficiency of these models. The linear operations in the thesis are convolutions and fully-connected neural layers, while non-linear operations are maxpool and ReLU (Eq 2.19, 2.17). The data type used in the implementation is double (8 bytes).

Though other practical CNN models typically use the float data type (4 bytes), the double data type VGG16 model in this thesis, with tens of millions of parameters and a size of 297 MB, offers practical reference for the secure computation of large language models, which are often hundreds of MBs in size (e.g., BERT is 110 MB [56]).

The following tables demonstrate the models' details for MNIST dataset implemented in the thesis. The structures for cifar10 is the same with MNIST except the input layers at the beginning to make them applicable for cifar10 images' sizes.

Network	Dataset	Model parameter number(double)	Model memory size(MB)
LeNet	MNIST	51,902	0.396
	Cifar10	52,202	0.398
AlexNet	MNIST	3,868,170	29.51
	Cifar10	3,868,746	29.51
ResNet18	MNIST	11,167,114	85.2
	Cifar10	11,168,266	85.21
VGG16	MNIST	38,946,762	297.14
	Cifar10	38,947,914	297.15

Table 5.0.1: The number and size of model parameters. Note that this reflects the parameters of plain-text CNN models; in cryptographic computations, the parameter count may be much higher due to the specific protocol's cryptographic format. In SGXDL, the enclave stores the plain-text CNN model and performs plain-text computations. This table demonstrates the computation workload and CNN model size in SGXDL. Since the CNN models share the same structure, the computation workload for MNIST and CIFAR-10 for the same number of batches is almost identical, so the time required to execute a batch for MNIST and CIFAR-10 should be similar.

layer name	**kernel size**	**output size**
conv1	1×6×5×5, stride=1, padding=2	6×28×28
relu	-	6×28× 28
maxpool	2×2, stride=2	6×14×14
conv2	6×16×5×5, stride=1	16×10×10
relu	-	16×10×10
maxpool	2×2, stride=2	16×5× 5
conv3	16×120× 5×5, stride=1	120×1× 1
relu	-	120×1× 1
fc	120×10	10
relu	-	10

Table 5.0.2: LeNet Structure(MNIST) [55], fc is the fully-connected layer, conv is the convolution layer. The C code defined the structure in implementation see Appendix A.2.1

layer name	kernel size	output size
conv1	1×32× 3×3, stride=1, padding=1	32×28×28
relu	-	32×28×28
maxpool	2×2, stride=2	32×14×14
conv2	32×64×3×3, stride=1, padding=1	64×14×14
relu	-	64×14×14
maxpool	2×2, stride=2	64×7×7
conv3	64×128×3×3, stride=1, padding=1	128×7×7
relu	-	128×7×7
conv4	128×256×3×3, stride=1, padding=1	256×7×7
relu	-	256×7×7
conv5	256×256×3×3, stride=1, padding=1	256×7×7
relu	-	256×7×7
maxpool	3×3, stride=2	256×3×3
fc1	(256*3*3)×1024	1024
relu	-	1024
fc2	1024×512	512
relu	-	512
fc3	512×10	10
relu	-	10

Table 5.0.3: AlexNet Structure(MNIST)[52]. Appendix A.2.2

layer name		kernel size	output size
conv1		1×64×3×3, stride=1, padding=3	64×32×32
relu		-	64×32×32
maxpool		2×2, stride=2	64×16×16
Res Block1	conv1	64×64×3×3, stride=1, padding=1	64×16×16
	relu	-	64×16×16
	conv2	64×64×3×3, stride=1, padding=1	64×16×16
	relu	-	64×16×16
	conv3	64×64×3×3, stride=1, padding=1	64×16×16
	relu	-	64×16×16
	conv4	64×64×3×3, stride=1, padding=1	64×16×16
	relu	-	64×16×16
Res Block2	conv1	64×128×3×3, stride=2, padding=1	128×8×8
	relu	-	128×8×8
	conv2	128×128×3×3, stride=1, padding=1	128×8×8
	relu	-	128×8×8
	conv3	128×128×3×3, stride=1, padding=1	128×8×8
	relu	-	128×8×8
	conv4	128×128×3×3, stride=1, padding=1	128×8×8
	relu	-	128×8×8
Res Block3	conv1	128×256×3×3, stride=2, padding=1	256×4×4
	relu	-	256×4×4
	conv2	256×256×3×3, stride=1, padding=1	256×4×4
	relu	-	256×4×4
	conv3	256×256×3×3, stride=1, padding=1	256×4×4
	relu	-	256×4×4
	conv4	256×256×3×3, stride=1, padding=1	256×4×4
	relu	-	256× 4× 4
Res Block4	conv1	256×512×3×3, stride=2, padding=1	512×2×2
	relu	-	512×2×2
	conv2	512×512×3×3, stride=1, padding=1	512×2×2
	relu	-	512× 2× 2
	conv3	512×512×3×3, stride=1, padding=1	512×2×2
	relu	-	512×2×2
	conv4	512×512×3×3, stride=1, padding=1	512×2×2
	relu	-	512×2×2
maxpool		2×2, stride=2	256×1×1
fc		512×10	10
relu		-	10

Table 5.0.4: ResNet18 Structure(MNIST)[39]. ResBlock is the convolution blocks with res connections. Appendix A.2.3

layer name		kernel size	output size
Conv Block1	conv1	1×64×3×3, stride=1, padding=3	64×32×32
	relu	-	64×32×32
	conv2	64×64×3×3, stride=1, padding1	64×32×32
	relu	-	64×32×32
	maxpool	2×2, stride=2	64×16×16
Conv Block2	conv1	64×128×3×3, stride=1, padding=1	128×16×16
	relu	-	128×16×16
	conv2	128×128×3×3, stride=1, padding=1	128×16×16
	relu	-	128×16×16
	maxpool	2×2, stride=2	128×8×8
Conv Block3	conv1	128×256×3×3, stride=1. padding=1	256×8×8
	relu	-	256×8×8
	conv2	256×256×3×3, stride=1, padding=1	256×8×8
	relu	-	256×8×8
	conv3	256×256×3×3, stride=1, padding=1	256×8×8
	relu	-	256×8×8
	conv4	256×256×3×3, stride=1, padding=1	256×8×8
	relu	-	256×8×8
	maxpool	2×2, stride=2	256×4×4
Conv Block4	conv1	256×512×3×3, stride=1, padding=1	512×4×4
	relu	-	512×4×4
	conv2	512×512×3×3, stride=1, padding=1	512×4×4
	relu	-	512×4×4
	conv3	512×512×3×3, stride=1, padding=1	512×4×4
	relu	-	512×4×4
	conv4	512×512×3×3, stride=1, padding=1	512×4×4
	relu	-	512×4×4
	maxpool	2×2, stride=2	512×2×2
Conv Block5	conv1	512×512×3×3, stride=1, padding=1	512×2×2
	relu	-	512×2×2
	conv2	512×512×3×3, stride=1, padding=1	512×2×2
	relu	-	512×2×2
	conv3	512×512×3×3, stride=1, padding=1	512×2×2
	relu	-	512×2×2
	conv4	512×512×3×3, stride=1, padding=1	512×2×2
	relu	-	512×2×2
	maxpool	2×2, stride=2	512×1×1
fc1		512×4096	4096
relu		-	4096
fc2		4096×4096	4096
relu		-	4096
fc3		4096 ×10	10
relu		-	10

Table 5.0.5: VGG16 Structure(MNIST)[86]. ResBlock is the convolution blocks with res connections. Appendix A.2.4

Chapter 6

Results and Analysis

This chapter presents the performance evaluation of SGXDL and HybridSGXDL in real-world deep learning applications, examining both macro and micro-level details. The chapter analyzes SGXDL and HybridSGXDL's features and identifies the scenarios where each is most suitable. A comparison with Piranha is also provided to highlight the superior efficiency of SGXDL and HybridSGXDL in secure computing. All data in this thesis were generated under a LAN setup with 2GB/s bandwidth and 2ms latency.

6.1 Non-linear operation in Intel SGX

Non-linear operations are heavily consuming with a large communication overhead and memory requirement of cryptography computation including MPC protocols. Piranha applied the MPC protocols with GPU acceleration, but the ReLU execution still requires more time than SGX's plain-text computation without GPU acceleration. There is a further question for MPC utilizing GPUs, that the GPU acceleration algorithms are primarily designed for floating point arithmetic computation with reasonable memory constraints, while MPC are operating over integer types with significantly high available memory requirements. Though Piranha adds the integer type support enabling GPU integer type acceleration, there is the cost that using less efficient GPU integer cores and kernels, such solutions also require an extremely high memory requirement.

Indicated by Figure 6.4.1, the ReLU plain-text computation in SGX is far more efficient

and quickly than Piranha's secure computation of MPC protocols with the acceleration of GPUs, which provides the idea to put the computation especially the non-linear computation of plain-text computation format in SGX enclave with the protection of hardware security techniques. This observation provides the evidence and support to propose the non-linear operation into SGX enclave for plain-text computation.

It can also be observed that the execution time from the beginning of a small computation workload does not increase until the ReLU computation workload size become larger than 1 MB, which indicates that the beginning lowest of 2ms is mainly the time to call SGX interface and transform the data to enclave.

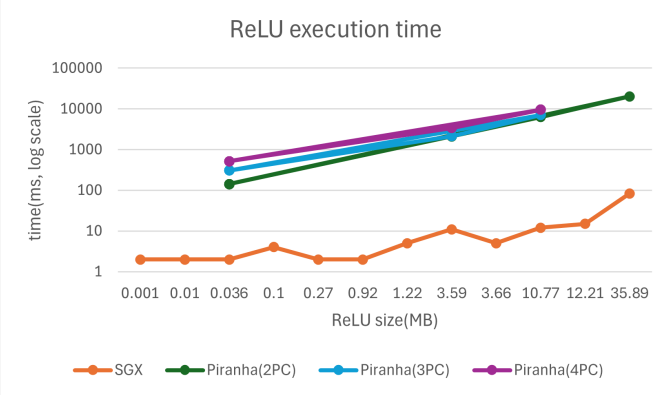

Figure 6.1.1: ReLU execution time in SGX compared with Piranha

6.2 Efficiency Comparison SGXDL vs. HybridSGXDL vs. Piranha

From Table 6.2.1, we can conclude that SGXDL is the most efficient secure computation infrastructure for moderate computation workloads, which have been useful for most CNN industry applications [29, 78, 90]. On the other hand, HybridSGXDL demonstrates superior performance for large models and high-batch training computations, making it a feasible solution for secure training of large language models (LLMs). Both SGXDL and HybridSGXDL outperform Piranha in terms of computation efficiency. Piranha faces limitations due to GPU memory overflow caused by high memory requirements, while SGXDL and HybridSGXDL, which use plain-text computation with reasonable memory requirements, do not suffer from this disadvantage.

			SGXDL	HybridSGXDL (3PC)	Piranha (2PC)	Piranha (3PC)	Piranha (4PC)
batchsize=1	LeNet	MNIST	**15.00**	12,272.41	588.35	1,618.52	2,462.94
		Cifar10	**29.50**	11,845.01	1,011.81	1,618.52	2,161.96
	AlexNet	MNIST	**1,415.70**	24,965.14	13,013.80	2,886.52	7,013.60
		Cifar10	**1,812.55**	23,977.07	3,431.93	2,765.13	6,996.19
	ResNet18	MNIST	**3,621.14**	63,932.61	73,486.00	22,878.86	57,304.80
		Cifar10	**3,642.14**	63,130.10	27,838.00	23,484.04	57,467.20
	VGG16	MNIST	**11,645.89**	76,775.55	16,905.30	13,959.71	29,298.70
		Cifar10	**13,000.70**	64,503.51	16,462.60	27,897.08	29,152.80
batchsize=100	LeNet	MNIST	**232.40**	12,298.98	7,805.07	1,337.70	13,530.10
		Cifar10	**451.40**	12,624.20	7,991.68	6,019.23	13,707.80
	AlexNet	MNIST	**17,032.80**	25,952.27	100,063.00	107,841.10	158,161.00
		Cifar10	**21,599.30**	28,661.03	99,370.00	63,969.34	159,431.00
	ResNet18	MNIST	**43,236.20**	67,048.59	567,553.00	N/A	N/A
		Cifar10	**43,933.70**	65,331.23	499,207.00	N/A	N/A
	VGG16	MNIST	168,929.90	**81,493.17**	N/A	N/A	N/A
		Cifar10	131,468.90	**67,897.09**	N/A	N/A	N/A
batchsize=300	LeNet	MNIST	**567.00**	13,065.24	22,753.20	17,391.69	38,193.00
		Cifar10	**1,083.60**	13,141.89	22,174.70	17,758.10	38,951.50
	AlexNet	MNIST	48,122.90	**27,641.74**	295,326.00	N/A	N/A
		Cifar10	62,005.30	**28,240.25**	294,758.00	N/A	N/A
	ResNet18	MNIST	122,788.40	**70,640.84**	N/A	N/A	N/A
		Cifar10	125,122.80	**66,859.13**	N/A	N/A	N/A
	VGG16	MNIST	519,591.67	**82,328.94**	N/A	N/A	N/A
		Cifar10	358,065.00	**75,625.55**	N/A	N/A	N/A
batchsize=1000	LeNet	MNIST	**1,916.33**	13,853.92	75,242.90	N/A	N/A
		Cifar10	**3,595.83**	16,421.78	75,242.90	N/A	N/A
	AlexNet	MNIST	160,078.83	**35,939.76**	N/A	N/A	N/A
		Cifar10	203,220.67	**28,807.49**	N/A	N/A	N/A
	ResNet18	MNIST	405,699.17	**91,033.73**	N/A	N/A	N/A
		Cifar10	407,061.33	**106,064.55**	N/A	N/A	N/A
	VGG16	MNIST	1,774,915.67	**137,688.56**	N/A	N/A	N/A
		Cifar10	1,136,187.33	**138,133.41**	N/A	N/A	N/A

Table 6.2.1: The secure computation execution(ms), 'N/A' is the memory overflow that not available to execute the computation. The bold font denote the efficiency winner of the same computation workload. The time is the time required to run a batch of training.

HybridSGXDL starts with a not-low 12,272.41 ms execution time for the smallest computation workloads due to the exchange overhead between the CPU and GPU. In contrast, SGXDL directly calculates the plain-text data with only one initial communication overhead in the enclave, and the exchange time between the enclave and CPU is nearly zero for normal data sizes. However, as the data computation size increases, HybridSGXDL shows its advantage with GPU acceleration for floating-point computations.

The datasets CIFAR-10 and MNIST use the same CNN structure, resulting in almost identical computation workloads for the same number of batches. SGXDL and HybridSGXDL exhibit similar execution times when the computation workload is consistent. However, Piranha's execution time can be highly unstable. For instance, Piranha (2PC) with batch size 1 and AlexNet shows significantly different execution times on MNIST and CIFAR-10, at 13,013.80 ms and 3,431.93 ms, respectively, despite their almost same computation workloads. Several other pairs with similar workloads also demonstrate this instability, as shown in Table 6.2.1.

This instability in Piranha using MPC protocols with GPU acceleration can be attributed to several factors. Firstly, Piranha is heavily impacted by communication overheads, causing the CPU to wait for communication and the GPU to operate below full capacity. Piranha needs more communication rounds that it has to handle the synchronization between communication and GPUs computation, in comparison with SGXDL and HybridSGXDL's single workflow. The frequent data exchanges between the CPU and GPU, along with the constant communication requirement, further exacerbate the issue. Although GPUs are capable of handling large workloads in parallel, their efficiency can be hindered by limited communication speed. This constraint forces GPUs to either wait for all data to arrive, incurring communication overhead, or proceed with computations despite incomplete data, leading to additional CPU-GPU exchanges to handle the remaining workloads. Both scenarios result in suboptimal operation. Even when all data is eventually received, the CPU may still need to wait for the GPU to complete its current tasks. In contrast, with HybridSGXDL, GPUs at local parties only begin processing after receiving all data shares for the current iteration, ensuring more efficient synchronization and operation. The complex synchronization of Piranha leads to unpredictable waiting periods during training, resulting in highly variable and unstable execution times, as illustrated in Table 6.2.1. In contrast, SGXDL and HybridSGXDL, as shown in Figure 3.3.1, follow a clear and

Number of images	100	300	1000	60000
MNIST	4	15	50	3,055
Cifar10	19	58	197	12,062

Table 6.3.1: The table presents the communication time required to transfer images of various sizes (ms) in a 3-party and LAN 2GB/s setup. It shows the time taken by the parties to send data shares for a specific number of samples to the SGX server. This communication time can remain consistent regardless of the number of parties, as all parties send their data shares simultaneously. However, when a large number of parties are sending samples concurrently, the server's receiving capacity need to be further evaluated for the communication time.

single workflow without requiring such complex synchronization, leading to stable execution efficiency.

6.3 SGXDL performance analysis

6.3.1 SGXDL overhead analysis

This section demonstrates the overhead involved in SGXDL training, which includes communication overhead, exchange time between the CPU and enclave, and data shares integration time. The analysis concludes that these overheads are nearly zero in practical training scenarios. This implies that SGXDL primarily spends time on plain-text computation, which is the main reason why SGXDL outperforms Piranha's cryptographic computation, which suffers heavily from communication overhead and CPU-GPU data exchange overhead.

Communication Time

Since each training session only requires a batch size of data samples, training can begin immediately after receiving the first batch of samples. During training, the server continues to receive additional data sample shares. As indicated in Figure 4.5.1, the communication overhead only involves the transmission of the first batch of data sample shares. Table 6.3.1 demonstrates the communication time for different sizes of shares. For example, the 50 ms communication overhead for a batch size of 1000 MNIST images in SGXDL constitutes only a small part of the overall batch training time.

Batchsize	CNNs model	Dataset	Communication overhead
batchsize=100	LeNet	MNIST	1.72%
		Cifar10	4.21%
	AlexNet	MNIST	0.02%
		Cifar10	0.09%
	ResNet18	MNIST	0.01%
		Cifar10	0.04%
	VGG16	MNIST	0.00%
		Cifar10	0.01%
batchsize=300	LeNet	MNIST	2.65%
		Cifar10	5.35%
	AlexNet	MNIST	0.03%
		Cifar10	0.09%
	ResNet18	MNIST	0.01%
		Cifar10	0.05%
	VGG16	MNIST	0.00%
		Cifar10	0.00%
batchsize=1000	LeNet	MNIST	2.61%
		Cifar10	5.48%
	AlexNet	MNIST	0.03%
		Cifar10	0.10%
	ResNet18	MNIST	0.01%
		Cifar10	0.05%
	VGG16	MNIST	0.00%
		Cifar10	0.02%

Table 6.3.2: The communication overhead for a batch of training of SGXDL(2PC). It shows that in some conditions of computation workloads, communication overhead for training a batch number of data samples have been nearly zero. As the number of data samples increases in practical scenarios, the communication overhead would reduce to zero.

Number of images(double)	1000	10000	20000	30000	40000	60000	800000	180000	240000
MNIST	~ 0	1	2	3	4	7	23	104	148
Cifar10	~ 0	7	9	14	56	100	150	N/A	N/A

Table 6.3.3: Exchange time between enclave and CPU(ms), the table shows the exchange time between CPU and enclave for different sizes of data exchanges. The data type of data is double.

Number of parties	2 PC	3 PC	4 PC	10 PC	15 PC	20 PC	40 PC	50 PC
Integration time(ms)	4	4	5	5	5	5	6	9

Table 6.3.4: The integration of data shares in the enclave for 60,000 MNIST images. The table demonstrates the integration time required to retrieve the original data under different party conditions. The data type used is double.

The exchange time between CPU and enclave

Table 6.3.3 demonstrates that in SGXDL and HybridSGXDL, the data exchange time between the enclave and CPU is nearly zero. This is because each training session processes a batch of images, with the maximum batch size being 1000. Therefore, in SGXDL and HybridSGXDL, the data size for each exchange is not larger than this 1000-image batch, resulting in minimal exchange time. The enclave, being a physical area on the CPU, allows for this minimal exchange time. This is in stark contrast to the exchange time between the CPU and GPU, where data exchange often constitutes a major overhead in GPU computations.

The integration time for data shares

As we know, data shares need to be integrated in the enclave before executing plain-text computation. Table 6.3.4 shows that the integration time would be nearly zero in applications because, in SGXDL, each session typically processes only a batch of data samples' shares, with a maximum batch size of 1000. This is much smaller than the data sizes in the Table 6.3.4. In extreme cases with many participating parties, there would be some integration time in account, but it would still represent a very small fraction of the total computation training time for SGXDL.

Since SGXDL executes the entire plain-text training process within the enclave, the number of parties only affects the data size for communication, exchange, and integration, with overheads being nearly zero in most practical cases. Therefore, we can conclude that the number of participating parties has a very limited influence on SGXDL's performance, making it a viable solution for scenarios with a large number of participating parties.

SGXDL computation ability analysis

As evidenced in Table 6.3.5, the computational capability of Intel SGX for CNNs is inferior even to that of the CPU. The Intel SGX environment supports only 9 threads

Batchsize	CNNs model	Dataset	CPU	Intel SGX
batchsize=1	LeNet	MNIST	7.33	15.00
		Cifar10	10.02	29.50
	AlexNet	MNIST	929.47	1,415.70
		Cifar10	3,833.64	1,812.55
	ResNet18	MNIST	2,815.83	3,621.13
		Cifar10	4,156.36	3,642.14
	VGG16	MNIST	4,665.35	11,645.89
		Cifar10	10,342.32	13,000.70
batchsize=100	LeNet	MNIST	31.51	232.40
		Cifar10	41.86	451.40
	AlexNet	MNIST	2,445.56	17,032.80
		Cifar10	7,061.73	21,599.30
	ResNet18	MNIST	10,977.70	43,236.20
		Cifar10	11,289.12	43,933.70
	VGG16	MNIST	20,770.98	168,929.90
		Cifar10	20,312.35	131,468.90
batchsize=300	LeNet	MNIST	88.43	567.00
		Cifar10	95.26	1,083.60
	AlexNet	MNIST	7,377.75	48,122.90
		Cifar10	8,695.30	62,005.30
	ResNet18	MNIST	35,786.55	122,788.40
		Cifar10	21,117.2	125,122.8
	VGG16	MNIST	68,704.3	519,591.6667
		Cifar10	94,676.65	358,065.00
batchsize=1000	LeNet	MNIST	244.93	1,916.33
		Cifar10	268.43	3,595.83
	AlexNet	MNIST	20,192.00	160,078.83
		Cifar10	25,138.83	203,220.67
	ResNet18	MNIST	97,536.67	405,699.17
		Cifar10	119,430.17	407,061.33
	VGG16	MNIST	238,136.00	1,774,915.67
		Cifar10	271,556.67	1,136,187.33

Table 6.3.5: The computation time comparison between CPU and Intel SGX(ms).

for parallel computing, in contrast to the CPU, which can utilize up to 144 threads. This disparity results in significantly slower computation times for SGX compared to the CPU. Additionally, the CPU itself underperforms compared to GPU acceleration for Convolutional Neural Networks (CNNs). Consequently, SGXDL is less efficient than HybridSGXDL for handling large data computation workloads.

6.4 HybridSGXDL performance analysis

6.4.1 Non-linear and Linear Computation

From Tables 6.4.1, 6.4.2, and 6.4.3, it is evident that non-linear operations, such as ReLU computation in Piranha, are significantly more expensive than secure plain-text ReLU computation in HybridSGXDL. This discrepancy primarily affects Piranha's performance, while HybridSGXDL benefits from it. For linear operations in HybridSGXDL, such as convolution and fully connected layers, the arithmetic computation time for small and moderate tasks is similar, overall time mainly

consisting of CPU-GPU exchange time. Although Piranha outperforms HybridSGXDL in linear computations for small and moderate workloads, its higher memory requirement leads to slower execution compared to HybridSGXDL and memory overflow for larger batch sizes and models, making it unfeasible.

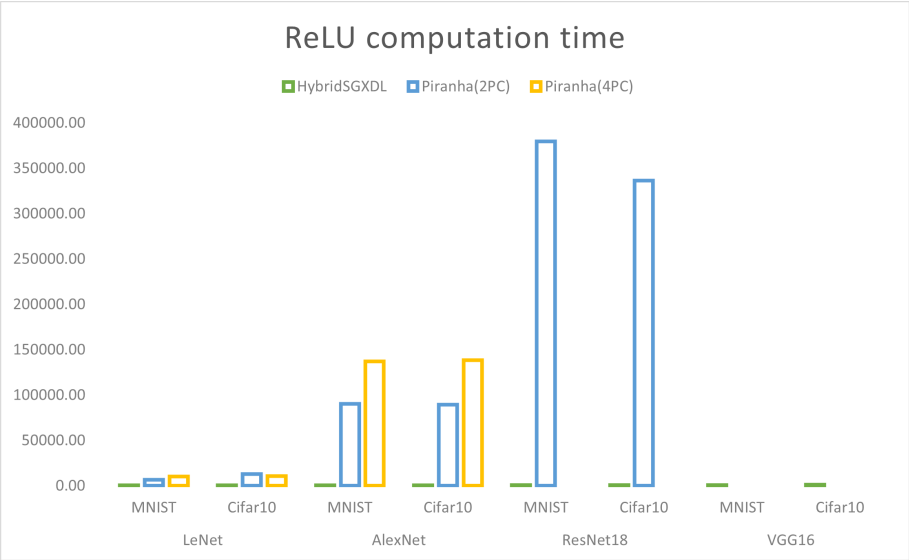

Figure 6.4.1: ReLU overhead for HybridSGXDL and Piranha (batch size = 100). The figure shows the ReLU computation time (ms) for the same workload in HybridSGXDL and Piranha, indicating that Piranha suffers from heavy ReLU computation overhead. There are no statistical data for Piranha executing VGG16 and ResNet18 due to its high memory requirements, making execution impossible.

6.4.2 Communication overhead

From Figure 6.4.2, we can infer several critical insights regarding the communication overhead of the evaluated protocols. Piranha with 2-party and 3-party protocols suffers significantly from communication overhead. Although the 4-party Piranha protocol appears to exhibit a lower communication overhead, it encounters numerous conditions it cannot execute due to large memory requirements, leading to memory overflow.

HybridSGXDL consistently maintains the lowest communication overhead compared to Piranha across all data points. Although its communication time increases with larger computational workloads, it is not significantly impacted by the workload size. This is due to the fact that the primary computation time is dominated by the efficient data exchange between the GPU and CPU, where GPU computation remains fast regardless of data size. In contrast, Piranha's communication time increases

CNNs model	Dataset	Secure computation	Relu	Conv	Fc
LeNet	MNIST	HybridSGXDL	20.42	9,788.42	2,454.76
		Piranha(2PC)	**449.01**	108.29	24.31
		Piranha(4PC)	**1,663.87**	672.95	97.75
	Cifar10	HybridSGXDL	22.70	10,104.54	6,437.34
		Piranha(2PC)	**872.90**	104.51	26.46
		Piranha(4PC)	**1,676.86**	348.28	136.82
AlexNet	MNIST	HybridSGXDL	40.11	17,294.70	7,613.41
		Piranha(2PC)	**6,123.99**	3,483.95	3,971.81
		Piranha(4PC)	**4,162.24**	1,432.48	1,390.19
	Cifar10	HybridSGXDL	1,013.14	17,663.05	8,572.06
		Piranha(2PC)	**1,692.40**	795.23	934.24
		Piranha(4PC)	**4,070.43**	1,475.19	1,450.55
ResNet18	MNIST	HybridSGXDL	148.27	60,316.00	3,427.16
		Piranha(2PC)	**20,307.40**	45,882.49	26.33
		Piranha(4PC)	**18,854.71**	25,714.94	142.07
	Cifar10	HybridSGXDL	85.01	59,677.30	3,347.17
		Piranha(2PC)	**6,720.78**	17,682.75	23.52
		Piranha(4PC)	**18,483.81**	26,096.50	114.66
VGG16	MNIST	HybridSGXDL	138.25	64,749.39	11,844.76
		Piranha(2PC)	**5,470.29**	11,251.64	176.01
		Piranha(4PC)	**12,715.48**	16,150.69	404.17
	Cifar10	HybridSGXDL	148.00	53,938.77	10,365.97
		Piranha(2PC)	**5,234.41**	11,062.23	157.72
		Piranha(4PC)	**12,705.31**	16,015.11	432.37

Table 6.4.1: Comparison of linear and non-linear operations computation time (ms) for batch size = 1 training. The table demonstrates the computation time for non-linear (ReLU) and linear operations (Conv: convolution, and Fc: fully connected neural layer) to train a batch of data samples with batch size = 1.

CNNs model	Dataset	Secure computation	Relu	Conv	Fc
LeNet	MNIST	HybridSGXDL	38.00	10,018.08	2,120.84
		Piranha(2PC)	**6,488.14**	807.35	24.03
		Piranha(4PC)	**10445.78**	1,986.38	115.88
	Cifar10	HybridSGXDL	34.75	9,991.44	2,436.28
		Piranha(2PC)	**13020.74**	794.57	24.09
		Piranha(4PC)	**10,634.79**	1,978.01	1,095.00
AlexNet	MNIST	HybridSGXDL	445.34	17,549.41	7,615.08
		Piranha(2PC)	90,160.70	8,169.61	1,240.08
		Piranha(4PC)	**136,924.35**	18,597.40	1,646.91
	Cifar10	HybridSGXDL	371.33	17,583.80	7,668.20
		Piranha(2PC)	**89612.17**	8,083.19	1,193.79
		Piranha(4PC)	**138,401.64**	18,334.15	2,694.90
ResNet18	MNIST	HybridSGXDL	581.09	62,805.01	3,298.77
		Piranha(2PC)	**379,778.39**	56,497.60	47.32
		Piranha(4PC)	N/A	N/A	N/A
	Cifar10	HybridSGXDL	648.20	60,432.96	3,729.67
		Piranha(2PC)	**336402.31**	50,944.13	48.21
		Piranha(4PC)	N/A	N/A	N/A
VGG16	MNIST	HybridSGXDL	982.80	67,135.34	12,960.23
		Piranha(2PC)	N/A	N/A	N/A
		Piranha(4PC)	N/A	N/A	N/A
	Cifar10	HybridSGXDL	1,061.00	55,810.51	10,145.53
		Piranha(2PC)	N/A	N/A	N/A
		Piranha(4PC)	N/A	N/A	N/A

Table 6.4.2: Comparison of linear and non-linear operations computation time (ms) for batch size = 100 training. ”N/A” indicates GPU memory overflow error, making execution unavailable.

CNNs model	Dataset	Secure computation	Relu	Conv	Fc
LeNet	MNIST	HybridSGXDL	57.28	10,601.25	2,003.70
		Piranha(2PC)	**18,863.01**	2,428.85	40.19
		Piranha(4PC)	**29,661.99**	5,508.42	121.90
	Cifar10	HybridSGXDL	46.43	10,118.56	2,489.78
		Piranha(2PC)	**18,748.62**	1,963.08	36.20
		Piranha(4PC)	**30,268.89**	5,702.77	2,979.77
AlexNet	MNIST	HybridSGXDL	1,056.59	18,145.94	7,518.71
		Piranha(2PC)	268,962.80	23,033.49	1,899.78
		Piranha(4PC)	N/A	N/A	N/A
	Cifar10	HybridSGXDL	1,013.14	17,663.05	8,572.35
		Piranha(2PC)	**268,029.46**	23,097.97	1,882.24
		Piranha(4PC)	N/A	N/A	N/A
ResNet18	MNIST	HybridSGXDL	1,704.54	64,257.77	3,279.16
		Piranha(2PC)	N/A	N/A	N/A
		Piranha(4PC)	N/A	N/A	N/A
	Cifar10	HybridSGXDL	581.50	62,191.56	3,379.08
		Piranha(2PC)	N/A	N/A	N/A
		Piranha(4PC)	N/A	N/A	N/A
VGG16	MNIST	HybridSGXDL	3,288.38	65,116.62	12,446.55
		Piranha(2PC)	N/A	N/A	N/A
		Piranha(4PC)	N/A	N/A	N/A
	Cifar10	HybridSGXDL	3,018.00	59,626.80	10,258.74
		Piranha(2PC)	N/A	N/A	N/A
		Piranha(4PC)	N/A	N/A	N/A

Table 6.4.3: Comparison of linear and non-linear operations computation time (ms) for batch size = 300 training. ”N/A” indicates GPU memory overflow error, making execution unavailable.

dramatically with larger model sizes, making it unsuitable for the secure training of large-scale models.

The substantial communication overhead in Piranha is primarily attributed to the frequent data share exchanges required for non-linear operations such as ReLU and Maxpool. Each of these operations necessitates multiple rounds of communication between parties, significantly increasing overhead.

In contrast, HybridSGXDL reduces communication overhead by performing secure plain-text floating-point computations with the assistance of SGX, requiring data share exchanges only as many times as the number of layers in CNNs. This single workflow communication strategy enables HybridSGXDL to maintain lower overhead and demonstrates its efficiency in handling larger computational workloads.

The significant communication overhead observed in Piranha protocols, particularly as the model size increases, suggests that these protocols may not be suitable for applications requiring efficient and scalable secure training of large models. On the other hand, HybridSGXDL’s ability to maintain low and stable communication overhead makes it a more practical solution for such applications, offering better performance and scalability.

Batchsize	CNNs model	Dataset	HybridSGXDL	Piranah(2PC)	Piranha(3PC)	Piranha(4PC)
batchsize=100	LeNet	MNIST	97.00	5,567.12	253.76	163.25
		Cifar10	131	5,734.14	4,492.22	460.76
	AlexNet	MNIST	1,040.00	76,375.15	88,994.24	2,072.67
		Cifar10	1,385	75,660.33	60,740.58	3,954.54
	ResNet18	MNIST	2,821	307,534.48	N/A	N/A
		Cifar10	2,851.00	305,505.69	N/A	N/A
	VGG16	MNIST	4,406.00	N/A	N/A	N/A
		Cifar10	4,425.00	N/A	N/A	N/A
batchsize=300	LeNet	MNIST	345.00	17,141.27	13,499.26	936.01
		Cifar10	440	17,724.98	13,816.36	1,382.27
	AlexNet	MNIST	3,201.00	229,091.92	N/A	N/A
		Cifar10	4,206.00	237,010.01	N/A	N/A
	ResNet18	MNIST	8,477.00	N/A	N/A	N/A
		Cifar10	8,552.00	N/A	N/A	N/A
	VGG16	MNIST	13,214.00	N/A	N/A	N/A
		Cifar10	13,293.00	N/A	N/A	N/A
batchsize=1000	LeNet	MNIST	1,149.00	59,986.84	N/A	N/A
		Cifar10	1,510.00	59,942.62	N/A	N/A
	AlexNet	MNIST	10,637.00	N/A	N/A	N/A
		Cifar10	14,093.00	N/A	N/A	N/A
	ResNet18	MNIST	28,311.00	N/A	N/A	N/A
		Cifar10	28,774.00	N/A	N/A	N/A
	VGG16	MNIST	44,552.00	N/A	N/A	N/A
		Cifar10	44,342.00	N/A	N/A	N/A

Table 6.4.4: Communication time(ms), the table demonstrates the communication time for a batch of training in secure computing infrastructures.

6.4.3 Exchange overhead between CPU and GPU

Figures 6.4.3, 6.4.4, and 6.4.6 provide a comprehensive analysis of the computation and exchange overheads in HybridSGXDL.

From Figure 6.4.6, we observe that the computation overhead in HybridSGXDL, which includes both non-linear operations on the CPU with SGX and linear operations on the GPU, decreases as the batch size increases. This trend is consistent across different neural network models such as LeNet, AlexNet, ResNet18, and VGG16. This indicates that the system becomes more efficient in handling larger batch sizes, leading to a reduction in the relative overhead.

Figure 6.4.4 further breaks down the exchange overhead between the CPU and GPU. It shows a potential downward trend in overhead as the computational workload increases with larger batch sizes. This suggests that the system's efficiency improves with increased workloads, likely due to better parallel processing and reduced relative communication time between the CPU and GPU. Additionally, Figure 6.4.3 provides a detailed comparison of the exchange overhead between different models and batch sizes. It highlights that, although the exchange overhead constitutes a significant portion of the total computation overhead, this overhead decreases as the batch size increases. For instance, the overhead for LeNet, AlexNet, ResNet18, and VGG16 is

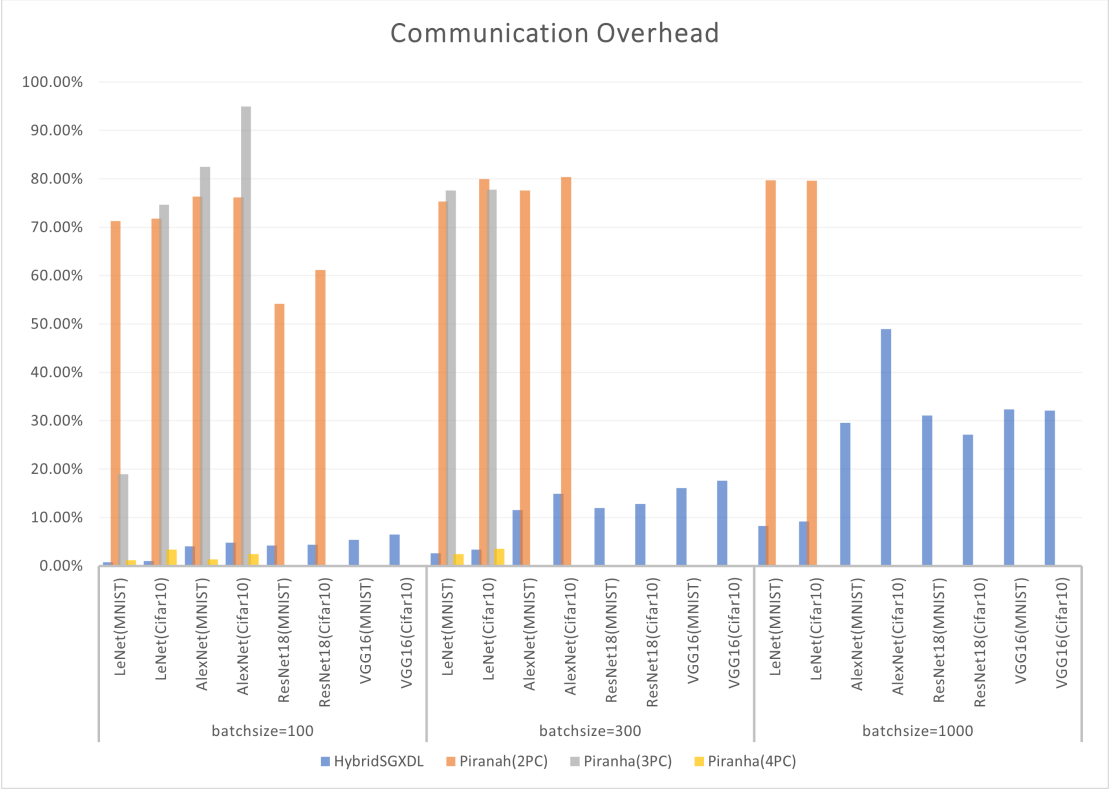

Figure 6.4.2: Communication Overhead Comparison. The figure compares the communication overhead for different neural network architectures (LeNet, AlexNet, ResNet18, and VGG16) across various batch sizes. Note that for batch sizes of 300 and 1000, the data points are missing for Piranha because of the large memory requirement unavailable to execute. These missing values represent very large overheads and are not zeros. This highlights the significant increase in overhead when the batch size reaches these values. It can be concluded that Piranha exhibits a very large overhead, which significantly impacts its performance.

substantially higher at smaller batch sizes but diminishes as the batch size grows, indicating improved efficiency and scalability.

In summary, these figures collectively demonstrate that while the exchange overhead between the CPU and GPU is a significant component of the total computation time, it exhibits a decreasing trend with increasing computational workloads and batch sizes. This also aligns with our analysis in Section6.3.1, where we concluded that the exchange overhead between the CPU and SGX is minimal and negligible. The efficient handling of larger computational workloads by HybridSGXDL highlights its potential for scalability and performance optimization in secure deep learning applications.

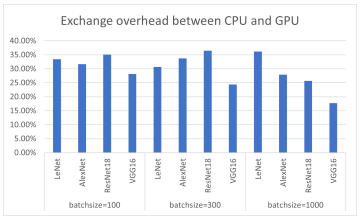

Figure 6.4.3: The image reveals apart from communication overhead, the remaining overhead for HybridSGXDL mainly consists of the exchange time between the CPU and GPU. This indicates that the time spent on non-linear operations on the SGX and linear operations on the GPU constitutes the main portion of the total computation time for HybridSGXDL.

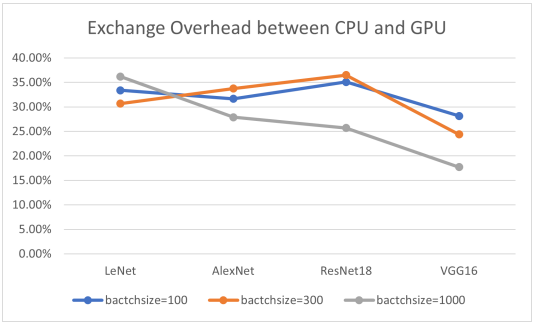

Figure 6.4.4: The figures indicate a potential downward trend in CPU and GPU exchange overhead as the computational workload increases with larger model sizes for each training batch.

Figure 6.4.5: Exchange overhead CPU and GPU in HybridSGXDL(MNIST), the figure describes the exchange overhead between CPU and GPU of HybridSGXDL among various CNNs models and batchsizes.

Batchsize	CNNs model	Exchange Time
batchsize=100	LeNet	4,108.15
	AlexNet	8,216.30
	ResNet18	23,528.87
	VGG16	22,939.80
batchsize=300	LeNet	4,008.90
	AlexNet	9,328.69
	ResNet18	25,774.83
	VGG16	20,071.21
batchsize=1000	LeNet	5,014.02
	AlexNet	10,028.05
	ResNet18	23,402.26
	VGG16	24,381.48

Table 6.4.5: Exchange time between CPU and GPU(MNIST,ms) in HybridSGXDL

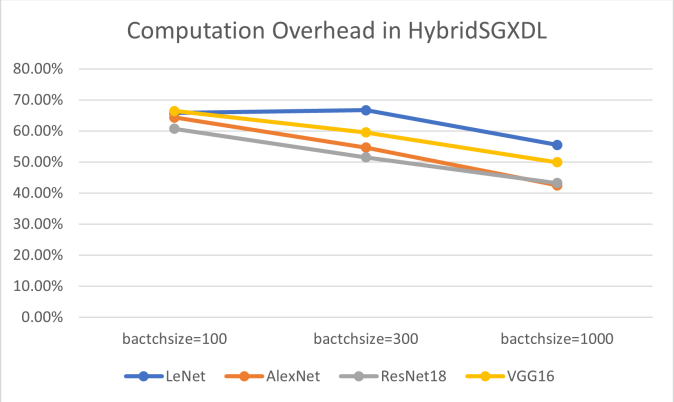

Figure 6.4.6: Computation Overhead of HybridSGXDL(MNIST)

6.4.4 Accuracy

Figure 6.4.7 illustrates that HybridSGXDL maintains accuracy without any drop during secure computation. This is achieved by executing floating-point plain-text calculations securely, ensuring precise and accurate results. Similarly, SGXDL also maintains plain-text computation without any accuracy drop, highlighting its effectiveness in secure environments.

In contrast, Piranha encounters accuracy concerns due to its use of integer cryptography calculations. By approximating non-linear operations like ReLU with polynomials, Piranha inevitably suffers from precision loss. This comparison underscores the advantage of HybridSGXDL and SGXDL in preserving computational accuracy while ensuring security.

The training loss curves for different models such as LeNet, VGG, and ResNet18 on the CIFAR-10 dataset show that HybridSGXDL effectively maintains low training loss across epochs. This consistent performance across various models demonstrates the robustness and reliability of HybridSGXDL in secure computation scenarios.

6.5 Security Analysis

This section provides the security analysis for the privacy-preserving deep learning infrastructures, SGXDL and HybridSGXDL, proposed in this thesis. The secure computing infrastructure utilizes Intel SGX hardware, involving a server with the SGX enclave and several clients. All participants are assumed to be honest but curious (semi-honest), meaning they may attempt to retrieve useful information but

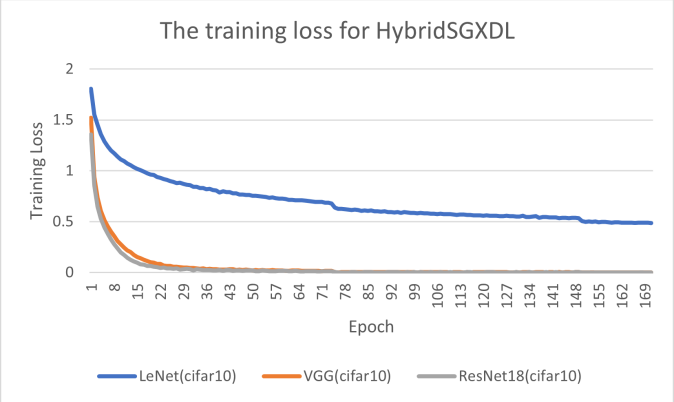

Figure 6.4.7: Training Loss in HybridSGXDL

will not maliciously attack the system by adding noise to the shares. The computing workflow follows the logic of weighted federated learning, where data is separated and distributed among participants in secret-sharing random data share formats. Secure channels are established to prevent the server from receiving all share values.

6.5.1 Side-Channel Attack for Intel SGX

A primary concern with Intel SGX is the potential for side-channel attacks, which can occur mainly through OCalls (calls from the enclave to execute functions outside the enclave) and interactions between the enclave and the external environment.

SGXDL

In SGXDL, the model is preserved within the enclave, and all computations are performed by SGX. The model remains within the enclave throughout the computing process. Interaction between the enclave and the outside world only occurs for dataset share exchanges, resulting in minimal interactions. There are no OCalls from the enclave during computing, making side-channel attacks like timing attacks, power attacks, and memory access pattern attacks unlikely. The entire training process is protected and executed within the enclave, with minimal observable interactions. Since the data shares exchange operations are uniform, no useful information about the model or data can be inferred from observable behaviors.

HybridSGXDL

HybridSGXDL separates training into linear and non-linear operations. This means the workflow and data process are not entirely oblivious to the participants, who

can identify which parts of the model involve convolution layers (linear operations) and which parts involve non-linear operations like maxpool, ReLU, and softmax. Participants can infer the structure of the CNN model by combining the data shares' structures and dimensions. However, the initial random parameters of the model are unknown to all parties outside the enclave, and only the updates in share format are transferred among participants, making the real values of model parameters unavailable to all parties.

In conclusion, HybridSGXDL may be susceptible to behavioral observation attacks to guess the structure of the CNN model, but the model parameter values remain secure. SGXDL provides stronger security protection than HybridSGXDL.

6.5.2 Gradients attack for Federated Learning

In traditional federated learning, attackers can obtain gradients transferred between the server and clients to retrieve information about data samples, such as membership and inference attacks. However, in SGXDL and HybridSGXDL, which use the weighted federated learning (wFL) pattern, the gradients and model in SGXDL do not leave the enclave because all computations are completed within the enclave. In HybridSGXDL, gradients are separated into random secret data shares, making the real values unavailable to all parties. Therefore, SGXDL and HybridSGXDL protect gradients within the enclave or encrypt them into random shares, mitigating the common threats in traditional federated learning arising from gradient transfer processes.

Chapter 7

Conclusion

In this thesis, we explored the applications of SGX for enabling floating-point plain-text computation within the enclave to advance privacy-preserving deep learning computations. We proposed and implemented two secure plain-text weighted federated learning computing infrastructures, SGXDL and HybridSGXDL, leveraging the hardware security capabilities of Intel SGX. These infrastructures were applied to practical deep learning models, demonstrating significant performance improvements. Both SGXDL and HybridSGXDL surpass the efficiency of Piranha, a multi-party computation (MPC) protocol enhanced with GPU acceleration, as reported in USENIX (2022). Our solutions achieve over 20× performance improvement and can train models that Piranha cannot due to memory overflow issues. SGXDL and HybridSGXDL make it feasible to securely train real-world neural networks with 100 million parameters of a floating data type (4 bytes), thus enabling secure training of large language models (LLMs).

Several factors contribute to the superior efficiency of SGXDL and HybridSGXDL over Piranha. These include the significant communication overhead for non-linear operations in Piranha, suboptimal GPU utilization due to complex communication synchronization, and the high computation workload for cryptographic and integer-type computations, whereas GPUs are optimized for floating-point operations. SGXDL excels in moderate computing sessions, prevalent in many industrial scenarios, while HybridSGXDL outperforms for larger models and datasets. This advantage becomes more apparent as the computation workload grows. SGXDL's superior performance in smaller and moderate sessions is attributed to lower communication overhead,

with data communication required only at the beginning of training. Conversely, HybridSGXDL's higher time-consuming data exchange overhead between CPU and GPU can take up to 2 seconds, compared to just 4 milliseconds of computing time. However, for larger computation workloads, the GPU's superior matrix computation capabilities make HybridSGXDL the best performer.

7.1 Future Work

While SGXDL offers stronger security protection than HybridSGXDL, its efficiency in training large models with extensive datasets is currently limited due to its reliance on CPU-based computation. Future improvements could integrate GPU support into SGX, significantly enhancing SGXDL's efficiency and making its performance comparable to standard plain-text deep learning training without security measures. Achieving privacy-preserving training without sacrificing efficiency would then become feasible.

Moreover, the secure infrastructures have been implemented on models such as ResNet18 and VGG16 using double data types, demonstrating feasibility for large language models (LLMs) with billions of floating-point parameters. Future work can extend SGXDL and HybridSGXDL to larger models such as GPT and BERT. Additionally, the secure channel designed in this thesis is at a preliminary level. Future research can focus on developing a more detailed and secure channel mechanism to further enhance the security of these computing infrastructures.

Bibliography

[1] Achiam, Josh, Adler, Steven, Agarwal, Sandhini, Ahmad, Lama, Akkaya, Ilge, Aleman, Florencia Leoni, Almeida, Diogo, Altenschmidt, Janko, Altman, Sam, Anadkat, Shyamal, et al. "Gpt-4 technical report". In: *arXiv preprint arXiv:2303.08774* (2023).

[2] Adams, Carlisle and Lloyd, Steve. *Understanding PKI: concepts, standards, and deployment considerations*. Addison-Wesley Professional, 2003.

[3] Aono, Yoshinori, Hayashi, Takuya, Wang, Lihua, Moriai, Shiho, et al. "Privacy-preserving deep learning via additively homomorphic encryption". In: *IEEE transactions on information forensics and security* 13.5 (2017), pp. 1333–1345.

[4] Beaver, D, Micali, S, and Rogaway, P. "The round complexity of secure protocols extended abstract". In: *22nd ACM STOC*.

[5] Beimel, Amos. "Secret-sharing schemes: A survey". In: *International conference on coding and cryptology*. Springer. 2011, pp. 11–46.

[6] Ben-Or, Michael, Goldwasser, Shafi, and Wigderson, Avi. "Completeness theorems for non-cryptographic fault-tolerant distributed computation". In: *Providing sound foundations for cryptography: on the work of Shafi Goldwasser and Silvio Micali*. 2019, pp. 351–371.

[7] Bogdanov, Dan, Laur, Sven, and Willemson, Jan. "Sharemind: A framework for fast privacy-preserving computations". In: *Computer Security-ESORICS 2008: 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings 13*. Springer. 2008, pp. 192–206.

[8] Bogdanov, Dan, Niitsoo, Margus, Toft, Tomas, and Willemson, Jan. “High-performance secure multi-party computation for data mining applications”. In: *International Journal of Information Security* 11 (2012), pp. 403–418.

[9] Bonawitz, Keith, Ivanov, Vladimir, Kreuter, Ben, Marcedone, Antonio, McMahan, H Brendan, Patel, Sarvar, Ramage, Daniel, Segal, Aaron, and Seth, Karn. “Practical secure aggregation for privacy-preserving machine learning”. In: *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 1175–1191.

[10] Bouacida, Nader and Mohapatra, Prasant. “Vulnerabilities in federated learning”. In: *IEEE Access* 9 (2021), pp. 63229–63249.

[11] Byrd, David and Polychroniadou, Antigoni. “Differentially private secure multi-party computation for federated learning in financial applications”. In: *Proceedings of the First ACM International Conference on AI in Finance*. 2020, pp. 1–9.

[12] Chaudhari, Harsh, Choudhury, Ashish, Patra, Arpita, and Suresh, Ajith. “ASTRA: high throughput 3pc over rings with application to secure prediction”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*. 2019, pp. 81–92.

[13] Chaudhari, Harsh, Rachuri, Rahul, and Suresh, Ajith. “Trident: Efficient 4pc framework for privacy preserving machine learning”. In: *arXiv preprint arXiv:1912.02631* (2019).

[14] Chen, Yu, Luo, Fang, Li, Tong, Xiang, Tao, Liu, Zheli, and Li, Jin. “A training-integrity privacy-preserving federated learning scheme with trusted execution environment”. In: *Information Sciences* 522 (2020), pp. 69–79.

[15] Cheng, Kewei, Fan, Tao, Jin, Yilun, Liu, Yang, Chen, Tianjian, Papadopoulos, Dimitrios, and Yang, Qiang. “Secureboost: A lossless federated learning framework”. In: *IEEE Intelligent Systems* 36.6 (2021), pp. 87–98.

[16] Choi, Joseph I. and Butler, Kevin R. B. “Secure Multiparty Computation and Trusted Hardware: Examining Adoption Challenges and Opportunities”. In: *Security and Communication Networks* 2019 (Apr. 2, 2019), pp. 1–28. ISSN: 1939-0114, 1939-0122. DOI: 10.1155/2019/1368905. URL: https://www.hindawi.com/journals/scn/2019/1368905/ (visited on 08/01/2023).

[17] Costan, Victor and Devadas, Srinivas. “Intel SGX explained”. In: *Cryptology ePrint Archive* (2016).

[18] Dalskov, Anders, Escudero, Daniel, and Keller, Marcel. “Fantastic four:{Honest-Majority}{Four-Party} secure computation with malicious security”. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 2183–2200.

[19] *Differential Privacy: How It Works, Benefits & Use Cases in 2024*. en. URL: `https://research.aimultiple.com/differential-privacy/` (visited on 05/04/2024).

[20] Doganay, Mahir Can, Pedersen, Thomas B, Saygin, Yücel, Savaş, Erkay, and Levi, Albert. “Distributed privacy preserving k-means clustering with additive secret sharing”. In: *Proceedings of the 2008 international workshop on Privacy and anonymity in information society*. 2008, pp. 3–11.

[21] Dong, Wentao and Wang, Cong. “Poster: Towards Lightweight TEE-Assisted MPC”. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, pp. 3609–3611.

[22] Dong, Ye, Chen, Xiaojun, Shen, Liyan, and Wang, Dakui. “Privacy-preserving distributed machine learning based on secret sharing”. In: *Information and Communications Security: 21st International Conference, ICICS 2019, Beijing, China, December 15–17, 2019, Revised Selected Papers 21*. Springer. 2020, pp. 684–702.

[23] Du, Wenliang and Atallah, Mikhail J. “Secure multi-party computation problems and their applications: a review and open problems”. In: *Proceedings of the 2001 workshop on New security paradigms*. 2001, pp. 13–22.

[24] El Ouadrhiri, Ahmed and Abdelhadi, Ahmed. “Differential privacy for deep and federated learning: A survey”. In: *IEEE access* 10 (2022), pp. 22359–22380.

[25] Evans, David, Kolesnikov, Vladimir, Rosulek, Mike, et al. “A pragmatic introduction to secure multi-party computation”. In: *Foundations and Trends® in Privacy and Security* 2.2-3 (2018), pp. 70–246.

[26] Fang, Haokun and Qian, Quan. “Privacy preserving machine learning with homomorphic encryption and federated learning”. In: *Future Internet* 13.4 (2021), p. 94.

[27] Fazli Khojir, Hamid, Alhadidi, Dima, Rouhani, Sara, and Mohammed, Noman. "FedShare: secure aggregation based on additive secret sharing in federated learning". In: *Proceedings of the 27th International Database Engineered Applications Symposium*. 2023, pp. 25–33.

[28] Felsen, Susanne, Kiss, Ágnes, Schneider, Thomas, and Weinert, Christian. "Secure and Private Function Evaluation with Intel SGX". In: *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*. CCS '19: 2019 ACM SIGSAC Conference on Computer and Communications Security. London United Kingdom: ACM, Nov. 11, 2019, pp. 165–181. ISBN: 978-1-4503-6826-1. DOI: 10.1145/3338466.3358919. URL: https://dl.acm.org/doi/10.1145/3338466.3358919 (visited on 07/20/2023).

[29] El-Feshawy, Somaya A, Saad, Waleed, Shokair, Mona, and Dessouky, Moawad. "IoT framework for brain tumor detection based on optimized modified ResNet 18 (OMRES)". In: *The Journal of Supercomputing* 79.1 (2023), pp. 1081–1110.

[30] Gascón, Adrià, Schoppmann, Phillipp, Balle, Borja, Raykova, Mariana, Doerner, Jack, Zahur, Samee, and Evans, David. "Privacy-preserving distributed linear regression on high-dimensional data". In: *Cryptology ePrint Archive* (2016).

[31] Gehlhar, Till, Marx, Felix, Schneider, Thomas, Suresh, Ajith, Wehrle, Tobias, and Yalame, Hossein. "Safefl: Mpc-friendly framework for private and robust federated learning". In: *2023 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2023, pp. 69–76.

[32] Ghodsi, Zahra, Veldanda, Akshaj Kumar, Reagen, Brandon, and Garg, Siddharth. "Cryptonas: Private inference on a relu budget". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 16961–16971.

[33] Goldreich, Oded. *Foundations of Cryptography, Volume 2*. Cambridge university press Cambridge, 2004.

[34] Goldreich, Oded. "Secure Multi-Party Computation". In: ().

[35] Goldreich, Oded, Micali, Silvio, and Wigderson, Avi. "How to play any mental game, or a completeness theorem for protocols with honest majority". In: *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*. 2019, pp. 307–328.

[36] Gupta, Prajjwal, Yadav, Krishna, Gupta, Brij B, Alazab, Mamoun, and Gadekallu, Thippa Reddy. “A novel data poisoning attack in federated learning based on inverted loss function”. In: *Computers & Security* 130 (2023), p. 103270.

[37] Hamza, Aljaafari and Kumar, Basant. “A review paper on DES, AES, RSA encryption standards”. In: *2020 9th International Conference System Modeling and Advancement in Research Trends (SMART)*. IEEE. 2020, pp. 333–338.

[38] Hardy, Stephen, Henecka, Wilko, Ivey-Law, Hamish, Nock, Richard, Patrini, Giorgio, Smith, Guillaume, and Thorne, Brian. “Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption”. In: *arXiv preprint arXiv:1711.10677* (2017).

[39] He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[40] Hitaj, Briland, Ateniese, Giuseppe, and Perez-Cruz, Fernando. *Deep Models Under the GAN: Information Leakage from Collaborative Deep Learning*. Sept. 14, 2017. arXiv: `1702.07464[cs,stat]`. URL: `http://arxiv.org/abs/1702.07464` (visited on 08/01/2023).

[41] Huang, Gao, Liu, Zhuang, Pleiss, Geoff, Van Der Maaten, Laurens, and Weinberger, Kilian Q. “Convolutional networks with dense connectivity”. In: *IEEE transactions on pattern analysis and machine intelligence* 44.12 (2019), pp. 8704–8716.

[42] Huang, Yimin, Wang, Wanwan, Zhao, Xingying, Wang, Yukun, Feng, Xinyu, He, Hao, and Yao, Ming. “EFMVFL: an efficient and flexible multi-party vertical federated learning without a third party”. In: *ACM Transactions on Knowledge Discovery from Data* 18.3 (2023), pp. 1–20.

[43] Hui, Siyuan, Zhang, Yuqiu, Hu, Albert, and Song, Edmund. “Horizontal Federated Learning and Secure Distributed Training for Recommendation System with Intel SGX”. In: *arXiv preprint arXiv:2207.05079* (2022).

[44] Jiang, Mingyang, Liang, Yanchun, Feng, Xiaoyue, Fan, Xiaojing, Pei, Zhili, Xue, Yu, and Guan, Renchu. “Text classification based on deep belief network and softmax regression”. In: *Neural Computing and Applications* 29 (2018), pp. 61–70.

[45] Kadam, Kalyani Ganesh and Khairnar, Vaishali. “Hybrid rsa-aes encryption for web services”. In: *International Journal of Technical Research and Applications”, Special* 31 (2015), pp. 51–56.

[46] Kaminaga, Hiroki, Awaysheh, Feras M, Alawadi, Sadi, and Kamm, Liina. “MPCFL: Towards Multi-party Computation for Secure Federated Learning Aggregation”. In: *Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing*. 2023, pp. 1–10.

[47] Kanagavelu, Renuga, Li, Zengxiang, Samsudin, Juniarto, Yang, Yechao, Yang, Feng, Goh, Rick Siow Mong, Cheah, Mervyn, Wiwatphonthana, Praewpiraya, Akkarajitsakul, Khajonpong, and Wang, Shangguang. “Two-phase multi-party computation enabled privacy-preserving federated learning”. In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE. 2020, pp. 410–419.

[48] Keller, Marcel and Scholl, Peter. “Efficient, oblivious data structures for MPC”. In: *Advances in Cryptology–ASIACRYPT 2014: 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, ROC, December 7-11, 2014, Proceedings, Part II 20*. Springer. 2014, pp. 506–525.

[49] Kim, Muah, Günlü, Onur, and Schaefer, Rafael F. “Federated learning with local differential privacy: Trade-offs between privacy, utility, and communication”. In: *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2021, pp. 2650–2654.

[50] Kolesnikov, Vladimir. *Secure two-party computation and communication*. Citeseer, 2006.

[51] Krizhevsky, Alex, Sutskever, Ilya, and Hinton, G. “ImageNet classification with deep convolutional neural networks (AlexNet) ImageNet classification with deep convolutional neural networks (AlexNet)”. In: *Actorsfit. Com* (2023).

[52] Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012).

[53] Kuswaha, Shashikant, Waghmare, Sachin, and Choudhary, P. “Data Transmission using AES-RSA Based Hybrid Security Algorithms”. In: *International Journal on Recent and Innovation Trends in Computing and Communication* 3.4 (2015), pp. 1964–1969.

[54] Kuutti, Sampo, Bowden, Richard, Jin, Yaochu, Barber, Phil, and Fallah, Saber. “A survey of deep learning applications to autonomous vehicle control”. In: *IEEE Transactions on Intelligent Transportation Systems* 22.2 (2020), pp. 712–733.

[55] LeCun, Yann et al. “LeNet-5, convolutional neural networks”. In: *URL: http://yann.lecun.com/exdb/lenet* 20.5 (2015), p. 14.

[56] Lee, JDMCK and Toutanova, K. “Pre-training of deep bidirectional transformers for language understanding”. In: *arXiv preprint arXiv:1810.04805* 3 (2018), p. 8.

[57] Li, Audeliano Wolian and Bastos, Guilherme Sousa. “Stock market forecasting using deep learning and technical analysis: a systematic review”. In: *IEEE access* 8 (2020), pp. 185232–185242.

[58] Li, Qiongxiu, Cascudo, Ignacio, and Christensen, Mads Græsbøll. “Privacy-preserving distributed average consensus based on additive secret sharing”. In: *2019 27th European Signal Processing Conference (EUSIPCO)*. IEEE. 2019, pp. 1–5.

[59] Lindell, Yehuda. “Secure multiparty computation”. In: *Communications of the ACM* 64.1 (2020), pp. 86–96.

[60] Liu, Changchang, Chakraborty, Supriyo, and Verma, Dinesh. “Secure model fusion for distributed learning using partial homomorphic encryption”. In: *Policy-Based Autonomic Data Governance* (2019), pp. 154–179.

[61] Liu, Yang, Kang, Yan, Xing, Chaoping, Chen, Tianjian, and Yang, Qiang. “A secure federated transfer learning framework”. In: *IEEE Intelligent Systems* 35.4 (2020), pp. 70–82.

[62] Liu, Ye, Gong, Wei, and Fan, Wenqing. “Application of AES and RSA Hybrid Algorithm in E-mail”. In: *2018 IEEE/ACIS 17th International Conference on Computer and Information Science (ICIS)*. IEEE. 2018, pp. 701–703.

[63] Lyu, Lingjuan, Yu, Han, Ma, Xingjun, Chen, Chen, Sun, Lichao, Zhao, Jun, Yang, Qiang, and Philip, S Yu. “Privacy and robustness in federated learning: Attacks and defenses”. In: *IEEE transactions on neural networks and learning systems* (2022).

[64] Lyu, Lingjuan, Yu, Han, and Yang, Qiang. “Threats to federated learning: A survey”. In: *arXiv preprint arXiv:2003.02133* (2020).

[65] Ma, Chuan, Li, Jun, Ding, Ming, Yang, Howard H, Shu, Feng, Quek, Tony QS, and Poor, H Vincent. “On safeguarding privacy and security in the framework of federated learning”. In: *IEEE network* 34.4 (2020), pp. 242–248.

[66] Macedonia, Michael. “The GPU enters computing’s mainstream”. In: *Computer* 36.10 (2003), pp. 106–108.

[67] Marsaglia, George and Tsang, Wai Wan. “The ziggurat method for generating random variables”. In: *Journal of statistical software* 5 (2000), pp. 1–7.

[68] McCulloch, Warren S and Pitts, Walter. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133.

[69] McKeen, Frank, Alexandrovich, Ilya, Berenzon, Alex, Rozas, Carlos V, Shafi, Hisham, Shanbhogue, Vedvyas, and Savagaonkar, Uday R. “Innovative instructions and software model for isolated execution.” In: *Hasp@ isca* 10.1 (2013).

[70] McMahan, Brendan and Ramage, Daniel. “Federated learning: Collaborative machine learning without centralized training data”. In: *Google Research Blog* (Mar. 2017).

[71] Mo, Fan and Haddadi, Hamed. “Efficient and private federated learning using tee”. In: *Proc. EuroSys Conf., Dresden, Germany*. 2019.

[72] Mo, Fan, Haddadi, Hamed, Katevas, Kleomenis, Marin, Eduard, Perino, Diego, and Kourtellis, Nicolas. “PPFL: privacy-preserving federated learning with trusted execution environments”. In: *Proceedings of the 19th annual international conference on mobile systems, applications, and services*. 2021, pp. 94–108.

[73] Mohassel, Payman and Zhang, Yupeng. “Secureml: A system for scalable privacy-preserving machine learning”. In: *2017 IEEE symposium on security and privacy (SP)*. IEEE. 2017, pp. 19–38.

[74] Mooney, Christopher Z. *Monte carlo simulation*. 116. Sage, 1997.

[75] Mothukuri, Viraaji, Parizi, Reza M, Pouriyeh, Seyedamin, Huang, Yan, Dehghantanha, Ali, and Srivastava, Gautam. “A survey on security and privacy of federated learning”. In: *Future Generation Computer Systems* 115 (2021), pp. 619–640.

[76] Muazu, Tasiu, Mao, Yingchi, Muhammad, Abdullahi Uwaisu, Ibrahim, Muhammad, Kumshe, Umar Muhammad Mustapha, and Samuel, Omaji. “A federated learning system with data fusion for healthcare using multi-party computation and additive secret sharing”. In: *Computer Communications* 216 (2024), pp. 168–182.

[77] Naehrig, Michael, Lauter, Kristin, and Vaikuntanathan, Vinod. “Can homomorphic encryption be practical?” In: *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. 2011, pp. 113–124.

[78] Naik, Ds Bhupal and Dondeti, Venkatesulu. “Pothole Detection and Classification in Vehicular Networks using ResNet-18”. In: *2023 IEEE 7th Conference on Information and Communication Technology (CICT)*. IEEE. 2023, pp. 1–6.

[79] Nair, Akarsh K, Raj, Ebin Deni, and Sahoo, Jayakrushna. “A robust analysis of adversarial attacks on federated learning environments”. In: *Computer Standards & Interfaces* 86 (2023), p. 103723.

[80] Nikolaenko, Valeria, Weinsberg, Udi, Ioannidis, Stratis, Joye, Marc, Boneh, Dan, and Taft, Nina. “Privacy-preserving ridge regression on hundreds of millions of records”. In: *2013 IEEE symposium on security and privacy*. IEEE. 2013, pp. 334–348.

[81] Ozbayoglu, Ahmet Murat, Gudelek, Mehmet Ugur, and Sezer, Omer Berat. “Deep learning for financial applications: A survey”. In: *Applied soft computing* 93 (2020), p. 106384.

[82] Rosenblatt, Frank. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6 (1958), p. 386.

[83] Shamir, Adi. “How to share a secret”. In: *Communications of the ACM* 22.11 (1979), pp. 612–613.

[84] Shamshirband, Shahab, Fathi, Mahdis, Dehzangi, Abdollah, Chronopoulos, Anthony Theodore, and Alinejad-Rokny, Hamid. “A review on deep learning approaches in healthcare systems: Taxonomies, challenges, and open issues”. In: *Journal of Biomedical Informatics* 113 (2021), p. 103627.

[85] Shi, Zhaosen, Yang, Zeyu, Hassan, Alzubair, Li, Fagen, and Ding, Xuyang. “A privacy preserving federated learning scheme using homomorphic encryption and secret sharing”. In: *Telecommunication Systems* 82.3 (2023), pp. 419–433.

[86] Simonyan, Karen and Zisserman, Andrew. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[87] So, Jinhyun, Güler, Başak, and Avestimehr, A Salman. “Byzantine-resilient secure federated learning”. In: *IEEE Journal on Selected Areas in Communications* 39.7 (2020), pp. 2168–2181.

[88] So, Jinhyun, Güler, Başak, and Avestimehr, A Salman. “Turbo-aggregate: Breaking the quadratic aggregation barrier in secure federated learning”. In: *IEEE Journal on Selected Areas in Information Theory* 2.1 (2021), pp. 479–489.

[89] Sotthiwat, Ekanut, Zhen, Liangli, Li, Zengxiang, and Zhang, Chi. “Partially encrypted multi-party computation for federated learning”. In: *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE. 2021, pp. 828–835.

[90] Stephen, Ancy, Punitha, A, and Chandrasekar, A. “Designing self attention-based ResNet architecture for rice leaf disease classification”. In: *Neural Computing and Applications* 35.9 (2023), pp. 6737–6751.

[91] Taud, Hind and Mas, Jean-Franccois. “Multilayer perceptron (MLP)”. In: *Geomatic approaches for modeling land change scenarios* (2018), pp. 451–455.

[92] Thistleton, William J, Marsh, John A, Nelson, Kenric, and Tsallis, Constantino. “Generalized Box–Müller method for generating q-gaussian random deviates”. In: *IEEE transactions on information theory* 53.12 (2007), pp. 4805–4810.

[93] Thomas, MTCAJ and Joy, A Thomas. *Elements of information theory*. Wiley-Interscience, 2006.

[94] Tjell, Katrine and Wisniewski, Rafael. “Privacy in distributed computations based on real number secret sharing”. In: *arXiv preprint arXiv:2107.00911* (2021).

[95] Tolpegin, Vale, Truex, Stacey, Gursoy, Mehmet Emre, and Liu, Ling. “Data poisoning attacks against federated learning systems”. In: *Computer Security–ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part I 25*. Springer. 2020, pp. 480–501.

[96] Vaswani, Ashish, Shazeer, Noam, Parmar, Niki, Uszkoreit, Jakob, Jones, Llion, Gomez, Aidan N, Kaiser, Łukasz, and Polosukhin, Illia. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

[97] Wagh, Sameer, Gupta, Divya, and Chandran, Nishanth. “SecureNN: 3-party secure computation for neural network training”. In: *Proceedings on Privacy Enhancing Technologies* (2019).

[98] Wagh, Sameer, Tople, Shruti, Benhamouda, Fabrice, Kushilevitz, Eyal, Mittal, Prateek, and Rabin, Tal. “Falcon: Honest-majority maliciously secure framework for private deep learning”. In: *arXiv preprint arXiv:2004.02229* (2020).

[99] Wang, Yongqin, Suh, G Edward, Xiong, Wenjie, Lefaudeux, Benjamin, Knott, Brian, Annavaram, Murali, and Lee, Hsien-Hsin S. “Characterization of mpc-based private inference for transformer-based models”. In: *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2022, pp. 187–197.

[100] Watson, Jean-Luc, Wagh, Sameer, and Popa, Raluca Ada. “Piranha: A {GPU} platform for secure computation”. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 827–844.

[101] Wei, Wenqi, Liu, Ling, Loper, Margaret, Chow, Ka-Ho, Gursoy, Mehmet Emre, Truex, Stacey, and Wu, Yanzhao. “A framework for evaluating gradient leakage attacks in federated learning”. In: *arXiv preprint arXiv:2004.10397* (2020).

[102] Wei, Wenqi, Liu, Ling, Wut, Yanzhao, Su, Gong, and Iyengar, Arun. “Gradient-leakage resilient federated learning”. In: *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2021, pp. 797–807.

[103] Werbos, Paul. “Beyond regression: New tools for prediction and analysis in the behavioral sciences”. In: *PhD thesis, Committee on Applied Mathematics, Harvard University, Cambridge, MA* (1974).

[104] Wu, Pengfei, Ning, Jianting, Shen, Jiamin, Wang, Hongbing, and Chang, Ee-Chien. “Hybrid Trust Multi-party Computation with Trusted Execution Environment.” In: *NDSS*. 2022.

[105] Wu, Xiaolong, Tian, Dave Jing, and Kim, Chung Hwan. “Building GPU TEEs using CPU Secure Enclaves with GEVisor”. In: *Proceedings of the 2023 ACM Symposium on Cloud Computing* (2023). DOI: 10.1145/3620678.3624659. URL: https://doi.org/10.1145/3620678.3624659.

[106] Wu, Yuncheng, Xing, Naili, Chen, Gang, Dinh, Tien Tuan Anh, Luo, Zhaojing, Ooi, Beng Chin, Xiao, Xiaokui, and Zhang, Meihui. “Falcon: A Privacy-Preserving and Interpretable Vertical Federated Learning System”. In: *Proceedings of the VLDB Endowment* 16.10 (2023), pp. 2471–2484.

[107] Xiong, Lizhi, Zhou, Wenhao, Xia, Zhihua, Gu, Qi, and Weng, Jian. “Efficient privacy-preserving computation based on additive secret sharing”. In: *arXiv preprint arXiv:2009.05356* (2020).

[108] Yao, Andrew C. “Protocols for secure computations”. In: *23rd annual symposium on foundations of computer science (sfcs 1982)*. IEEE. 1982, pp. 160–164.

[109] Zhang, Chengliang, Li, Suyi, Xia, Junzhe, Wang, Wei, Yan, Feng, and Liu, Yang. "{BatchCrypt}: Efficient homomorphic encryption for {Cross-Silo} federated learning". In: *2020 USENIX annual technical conference (USENIX ATC 20)*. 2020, pp. 493–506.

[110] Zhang, Chi, Ekanut, Sotthiwat, Zhen, Liangli, and Li, Zengxiang. "Augmented multi-party computation against gradient leakage in federated learning". In: *IEEE Transactions on Big Data* (2022).

[111] Zhang, Xinyang, Ji, Shouling, Wang, Hui, and Wang, Ting. "Private, yet practical, multiparty deep learning". In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2017, pp. 1442–1452.

[112] Zhang, Yahui, Zhao, Min, Li, Tingquan, and Han, Huan. "Survey of Attacks and Defenses against SGX". In: *2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC)*. 2020, pp. 1492–1496. DOI: 10.1109/ITOEC49072.2020.9141835.

[113] Zhao, Ying, Chen, Junjun, Zhang, Jiale, Wu, Di, Blumenstein, Michael, and Yu, Shui. "Detecting and mitigating poisoning attacks in federated learning using generative adversarial networks". In: *Concurrency and Computation: Practice and Experience* 34.7 (2022), e5906.

[114] Zhu, Huafei, Goh, Rick Siow Mong, and Ng, Wee-Keong. "Privacy-preserving weighted federated learning within the secret sharing framework". In: *IEEE Access* 8 (2020), pp. 198275–198284.

Appendix - Contents

Appendix A

First Appendix

A.1 Original Code for Pseudocode and model structure

A.1.1 Conv

```
#define CONVOLUTION_FORWARD(input,output,weight,bias,action)
{
for(int x=0;x<GETLENGTH(weight);++x)
 for(int y=0; y<GETLENGTH(*weight);++y)
  CONVOLUTE_VALID(input[x], output[y], weight[x][y]);
FOREACH(j, GETLENGTH(output))
 FOREACH(i, GETCOUNT(output[j]))
((double *)output[j])[i]=action(((double *)output[j])[i]
                        +bias[j]);
}

#define CONVOLUTE_VALID(input,output,weight)
{
  FOREACH(o0,GETLENGTH(output))
  FOREACH(o1,GETLENGTH(*(output)))
  FOREACH(w0,GETLENGTH(weight))
  FOREACH(w1,GETLENGTH(*(weight)))
    (output)[o0][o1]+=(input)[o0+w0][o1+w1](weight)[w0][w1];
}
```

A.2 Original Code for Pseudocode

A.2.1 LeNet5(MNIST)

```

#pragma once
#define LENGTH_KERNEL   5

#define LENGTH_FEATURE0 32
#define LENGTH_FEATURE1 (LENGTH_FEATURE0 - LENGTH_KERNEL + 1)
#define LENGTH_FEATURE2 (LENGTH_FEATURE1 >> 1)
```

```
#define LENGTH_FEATURE3 (LENGTH_FEATURE2 - LENGTH_KERNEL + 1)
#define LENGTH_FEATURE4 (LENGTH_FEATURE3 >> 1)
#define LENGTH_FEATURE5 (LENGTH_FEATURE4 - LENGTH_KERNEL + 1)

#define INPUT                   1
#define LAYER1                  6
#define LAYER2                  6
#define LAYER3                  16
#define LAYER4                  16
#define LAYER5                  120
#define OUTPUT          10

#define ALPHA 0.5
#define PADDING 2

typedef unsigned char uint8;
typedef uint8 image[28][28];


typedef struct LeNet5
{
double weight0_1[INPUT][LAYER1][LENGTH_KERNEL][LENGTH_KERNEL];
double weight2_3[LAYER2][LAYER3][LENGTH_KERNEL][LENGTH_KERNEL];
double weight4_5[LAYER4][LAYER5][LENGTH_KERNEL][LENGTH_KERNEL];
double weight5_6[LAYER5*LENGTH_FEATURE5*LENGTH_FEATURE5][OUTPUT];

double bias0_1[LAYER1];
double bias2_3[LAYER3];
double bias4_5[LAYER5];
double bias5_6[OUTPUT];

}LeNet5;

typedef struct Feature
{
double input[INPUT][LENGTH_FEATURE0][LENGTH_FEATURE0];
double layer1[LAYER1][LENGTH_FEATURE1][LENGTH_FEATURE1];
double layer2[LAYER2][LENGTH_FEATURE2][LENGTH_FEATURE2];
double layer3[LAYER3][LENGTH_FEATURE3][LENGTH_FEATURE3];
double layer4[LAYER4][LENGTH_FEATURE4][LENGTH_FEATURE4];
double layer5[LAYER5][LENGTH_FEATURE5][LENGTH_FEATURE5];
double output[OUTPUT];
}Feature;

void TrainBatch(LeNet5 *lenet,image *inputs,uint8 *labels,int batchSize);

void Train(LeNet5 *lenet, image input, uint8 label);

uint8 Predict(LeNet5 *lenet, image input, uint8 count);

void Initial(LeNet5 *lenet);
double relu(double x);
```

A.2.2 AlexNet(MNIST)

```

#pragma once

#define LENGTH_KERNEL   3
#define MAXPOOL_SIZE 2

#define PADDING 1
#define LENGTH_FEATURE0 28
#define LENGTH_FEATURE1_1       (LENGTH_FEATURE0 - LENGTH_KERNEL + 2*PADDING+1)
#define LENGTH_FEATURE1_2       (LENGTH_FEATURE1_1/2)

#define LENGTH_FEATURE2_1       (LENGTH_FEATURE1_2 - LENGTH_KERNEL +2*PADDING + 1)
#define LENGTH_FEATURE2_2       (LENGTH_FEATURE2_1/2)
```

```
14
15 #define LENGTH_FEATURE3_1       (LENGTH_FEATURE2_2 - LENGTH_KERNEL +2*PADDING+ 1)
16
17 #define LENGTH_FEATURE4_1       (LENGTH_FEATURE3_1 - LENGTH_KERNEL +2*PADDING+ 1)
18
19 #define MAXPOOL5_2_KERNEL 3
20 #define MAXPOOL5_2_STRIDE 2
21
22 #define LENGTH_FEATURE5_1       (LENGTH_FEATURE4_1 - LENGTH_KERNEL+2*PADDING+ 1)
23 #define LENGTH_FEATURE5_2       ((LENGTH_FEATURE5_1 - MAXPOOL5_2_KERNEL)/MAXPOOL5_2_STRIDE+ 1)
24
25 #define FC1_OUTPUT      1024
26 #define FC2_OUTPUT      512
27 #define FC3_OUTPUT      10
28
29
30 #define INPUT                   1
31 #define LAYER1                  32
32 #define LAYER2                  64
33 #define LAYER3                  128
34 #define LAYER4                  256
35 #define LAYER5                  256
36
37 #define OUTPUT          10
38
39 #define ALPHA 0.1
40
41
42 typedef unsigned char uint8;
43 typedef uint8 image[28][28];
44
45
46
47 typedef struct AlexNet
48 {
49         double weight1[INPUT][LAYER1][LENGTH_KERNEL][LENGTH_KERNEL];//Layer1 kernel
50         double weight2[LAYER1][LAYER2][LENGTH_KERNEL][LENGTH_KERNEL];//Layer2 Kernel
51         double weight3[LAYER2][LAYER3][LENGTH_KERNEL][LENGTH_KERNEL];//Layer3 Kernel
52         //double weight5_6[LAYER5 * LENGTH_FEATURE5 * LENGTH_FEATURE5][OUTPUT];
53         double weight4[LAYER3][LAYER4][LENGTH_KERNEL][LENGTH_KERNEL];//Layer4 Kernel
54         double weight5[LAYER4][LAYER5][LENGTH_KERNEL][LENGTH_KERNEL];//Layer5 Kernel
55
56
57         double fc1[LAYER5*LENGTH_FEATURE5_2*LENGTH_FEATURE5_2][FC1_OUTPUT];
58         double fc2[FC1_OUTPUT][FC2_OUTPUT];
59         double fc3[FC2_OUTPUT][FC3_OUTPUT];
60
61
62
63         double bias1[LAYER1];
64         double bias2[LAYER2];
65         double bias3[LAYER3];
66         double bias4[LAYER4];
67         double bias5[LAYER5];
68
69         double bias_fc1[FC1_OUTPUT];
70         double bias_fc2[FC2_OUTPUT];
71         double bias_fc3[FC3_OUTPUT];
72
73 }AlexNet;
74
75 typedef struct Feature
76 {
77         double input[INPUT][LENGTH_FEATURE0][LENGTH_FEATURE0];
78
79         double layer1_conv[LAYER1][LENGTH_FEATURE1_1][LENGTH_FEATURE1_1];
80         double layer1_pool[LAYER1][LENGTH_FEATURE1_2][LENGTH_FEATURE1_2];
81
82         double layer2_conv[LAYER2][LENGTH_FEATURE2_1][LENGTH_FEATURE2_1];
83         double layer2_pool[LAYER2][LENGTH_FEATURE2_2][LENGTH_FEATURE2_2];
84
```

```
85          double layer3_conv[LAYER3][LENGTH_FEATURE3_1][LENGTH_FEATURE3_1];
86
87          double layer4_conv[LAYER4][LENGTH_FEATURE4_1][LENGTH_FEATURE4_1];
88
89
90          double layer5_conv[LAYER5][LENGTH_FEATURE5_1][LENGTH_FEATURE5_1];//256*7*7
91          double layer5_pool[LAYER5][LENGTH_FEATURE5_2][LENGTH_FEATURE5_2];//256*3*3
92
93          double fc1[FC1_OUTPUT];
94          double fc2[FC2_OUTPUT];
95          double output[FC3_OUTPUT];
96
97  }Feature;
98
99
100 typedef struct Feature_Pad
101 {
102         double input[INPUT][LENGTH_FEATURE0+2*PADDING][LENGTH_FEATURE0+2*PADDING];
103
104         double layer1_conv[LAYER1][LENGTH_FEATURE1_1][LENGTH_FEATURE1_1];
105         double layer1_pool[LAYER1][LENGTH_FEATURE1_2+2*PADDING][LENGTH_FEATURE1_2+2*PADDING];
106
107         double layer2_conv[LAYER2][LENGTH_FEATURE2_1][LENGTH_FEATURE2_1];
108         double layer2_pool[LAYER2][LENGTH_FEATURE2_2+2*PADDING][LENGTH_FEATURE2_2+2*PADDING];
109
110         double layer3_conv[LAYER3][LENGTH_FEATURE3_1+2*PADDING][LENGTH_FEATURE3_1+2*PADDING];
111
112         double layer4_conv[LAYER4][LENGTH_FEATURE4_1+2*PADDING][LENGTH_FEATURE4_1+2*PADDING];
113
114
115         double layer5_conv[LAYER5][LENGTH_FEATURE5_1][LENGTH_FEATURE5_1];//256*7*7
116         double layer5_pool[LAYER5][LENGTH_FEATURE5_2][LENGTH_FEATURE5_2];//256*3*3
117
118         double fc1[FC1_OUTPUT];
119         double fc2[FC2_OUTPUT];
120         double output[FC3_OUTPUT];
121
122 }Feature_Pad;
123
124 void TrainBatch(AlexNet *alexnet, image *inputs, uint8 *labels, int batchSize);
125
126 void Train(AlexNet *alexnet, image input, uint8 label);
127
128 uint8 Predict(AlexNet *alexnet, image input, uint8 count);
129
130 void Initial(AlexNet *alexnet);
```

A.2.3 ResNet18(MNIST)

```
1
2
3   #pragma once
4
5   #define LENGTH_KERNEL   3
6
7   #define LENGTH_FEATURE0 32
8   #define LENGTH_FEATURE1 (LENGTH_FEATURE0 - LENGTH_KERNEL + 1)
9   #define LENGTH_FEATURE2 (LENGTH_FEATURE1 >> 1)
10  #define LENGTH_FEATURE3 (LENGTH_FEATURE2 - LENGTH_KERNEL + 1)
11  #define LENGTH_FEATURE4 (LENGTH_FEATURE3 >> 1)
12  #define LENGTH_FEATURE5 (LENGTH_FEATURE4 - LENGTH_KERNEL + 1)
13
14  #define INPUT                   1
15  #define LAYER0                  64
16  #define LAYER1                  64
17  #define LAYER2                  128
18  #define LAYER3                  256
19  #define LAYER4                  512
```

```
20 #define OUTPUT          10
21
22 #define Res1 16
23 #define Res2 8
24 #define Res3 4
25 #define Res4 2
26 #define pool2_size 1
27
28 #define ALPHA 0.5
29 #define PADDING 1
30
31 typedef unsigned char uint8;
32 typedef uint8 image[28][28];
33
34 typedef struct ResBlock1
35 {
36         double weight1_1[LAYER0][LAYER1][LENGTH_KERNEL][LENGTH_KERNEL];
37         double weight1_2[LAYER1][LAYER1][LENGTH_KERNEL][LENGTH_KERNEL];
38
39         double weight2_1[LAYER1][LAYER1][LENGTH_KERNEL][LENGTH_KERNEL];
40         double weight2_2[LAYER1][LAYER1][LENGTH_KERNEL][LENGTH_KERNEL];
41
42
43         double bias1_1[LAYER1];
44         double bias1_2[LAYER1];
45         double bias2_1[LAYER1];
46         double bias2_2[LAYER1];
47 }ResBlock1;
48 typedef struct ResBlock2
49 {
50         double weight1_1[LAYER1][LAYER2][LENGTH_KERNEL][LENGTH_KERNEL];
51         double weight1_2[LAYER2][LAYER2][LENGTH_KERNEL][LENGTH_KERNEL];
52         double conv1[LAYER1][LAYER2][1][1];
53         double weight2_1[LAYER2][LAYER2][LENGTH_KERNEL][LENGTH_KERNEL];
54         double weight2_2[LAYER2][LAYER2][LENGTH_KERNEL][LENGTH_KERNEL];
55
56         double bias1_1[LAYER2];
57         double bias1_2[LAYER2];
58         double bias2_1[LAYER2];
59         double bias2_2[LAYER2];
60 }ResBlock2;
61 typedef struct ResBlock3
62 {
63         double weight1_1[LAYER2][LAYER3][LENGTH_KERNEL][LENGTH_KERNEL];
64         double weight1_2[LAYER3][LAYER3][LENGTH_KERNEL][LENGTH_KERNEL];
65         double conv1[LAYER2][LAYER3][1][1];
66         double weight2_1[LAYER3][LAYER3][LENGTH_KERNEL][LENGTH_KERNEL];
67         double weight2_2[LAYER3][LAYER3][LENGTH_KERNEL][LENGTH_KERNEL];
68
69         double bias1_1[LAYER3];
70         double bias1_2[LAYER3];
71         double bias2_1[LAYER3];
72         double bias2_2[LAYER3];
73 }ResBlock3;
74 typedef struct ResBlock4
75 {
76         double weight1_1[LAYER3][LAYER4][LENGTH_KERNEL][LENGTH_KERNEL];
77         double weight1_2[LAYER4][LAYER4][LENGTH_KERNEL][LENGTH_KERNEL];
78         double conv1[LAYER3][LAYER4][1][1];
79         double weight2_1[LAYER4][LAYER4][LENGTH_KERNEL][LENGTH_KERNEL];
80         double weight2_2[LAYER4][LAYER4][LENGTH_KERNEL][LENGTH_KERNEL];
81
82         double bias1_1[LAYER4];
83         double bias1_2[LAYER4];
84         double bias2_1[LAYER4];
85         double bias2_2[LAYER4];
86 }ResBlock4;
87
88
89 typedef struct ResNet18
90 {
```

```
91          double weight1[INPUT][LAYER1][LENGTH_KERNEL][LENGTH_KERNEL];//Layer1 kernel
92          ResBlock1 res_block1;
93          ResBlock2 res_block2;
94          ResBlock3 res_block3;
95          ResBlock4 res_block4;
96
97          double fc[LAYER4*pool2_size*pool2_size][OUTPUT];
98          double bias1[LAYER1];
99          double bias_fc[OUTPUT];
100 }ResNet18;
101
102
103 typedef struct Res1_Feature
104 {
105         double conv1_1[LAYER1][Res1][Res1];
106         double conv1_2[LAYER1][Res1][Res1];
107
108         double conv2_1[LAYER1][Res1][Res1];
109         double conv2_2[LAYER1][Res1][Res1];
110 }Res1_Feature;
111
112
113 typedef struct Res2_Feature
114 {
115         double conv1_1[LAYER2][Res2][Res2];
116         double conv1_2[LAYER2][Res2][Res2];
117
118         double conv_res[LAYER2][Res2][Res2];
119
120         double conv2_1[LAYER2][Res2][Res2];
121         double conv2_2[LAYER2][Res2][Res2];
122 }Res2_Feature;
123
124 typedef struct Res3_Feature
125 {
126         double conv1_1[LAYER3][Res3][Res3];
127         double conv1_2[LAYER3][Res3][Res3];
128
129         double conv_res[LAYER3][Res3][Res3];
130
131         double conv2_1[LAYER3][Res3][Res3];
132         double conv2_2[LAYER3][Res3][Res3];
133 }Res3_Feature;
134
135 typedef struct Res4_Feature
136 {
137         double conv1_1[LAYER4][Res4][Res4];
138         double conv1_2[LAYER4][Res4][Res4];
139
140         double conv_res[LAYER4][Res4][Res4];
141
142         double conv2_1[LAYER4][Res4][Res4];
143         double conv2_2[LAYER4][Res4][Res4];
144 }Res4_Feature;
145
146
147
148
149
150 typedef struct Feature
151 {
152         double input[INPUT][LENGTH_FEATURE0][LENGTH_FEATURE0];
153         double conv1[LAYER0][LENGTH_FEATURE0][LENGTH_FEATURE0];
154         double pool1[LAYER0][16][16];
155         Res1_Feature res1f;
156         Res2_Feature res2f;
157         Res3_Feature res3f;
158         Res4_Feature res4f;
159
160         double pool2[LAYER4][pool2_size][pool2_size];
161         double output[OUTPUT];
```

```

}Feature;

void TrainBatch(ResNet18 *resnet, image *inputs, uint8 *labels, int batchSize);

void Train(ResNet18 *resnet, image input, uint8 label);

uint8 Predict(ResNet18 *resnet, image input, uint8 count);

void Initial(ResNet18 *resnet);
double relu(double x);
```

A.2.4 VGG16(MNIST)

```
#pragma once

// #include <charconv>
#define LENGTH_KERNEL   3

#define LENGTH_FEATURE0 32
#define LENGTH_FEATURE1 LENGTH_FEATURE0/2  //32
#define LENGTH_FEATURE2 LENGTH_FEATURE1/2  //16
#define LENGTH_FEATURE3 LENGTH_FEATURE2/2       //8
#define LENGTH_FEATURE4 LENGTH_FEATURE3/2       //4
#define LENGTH_FEATURE5 LENGTH_FEATURE4/2       //2



#define INPUT                   1
#define LAYER1                  64


#define LAYER2                  128

#define LAYER3                  256


#define LAYER4                  512

#define LAYER5                  512
#define FC1                             4096
#define FC2                             4096

#define OUTPUT          10

#define ALPHA 0.5
#define PADDING 1

typedef unsigned char uint8;
typedef uint8 image[28][28];


typedef struct VGG
{
        double weight1_1[INPUT][LAYER1][LENGTH_KERNEL][LENGTH_KERNEL];
        double weight1_2[LAYER1][LAYER1][LENGTH_KERNEL][LENGTH_KERNEL];
        double weight2_1[LAYER1][LAYER2][LENGTH_KERNEL][LENGTH_KERNEL];
        double weight2_2[LAYER2][LAYER2][LENGTH_KERNEL][LENGTH_KERNEL];


        double weight3_1[LAYER2][LAYER3][LENGTH_KERNEL][LENGTH_KERNEL];
        double weight3_2[LAYER3][LAYER3][LENGTH_KERNEL][LENGTH_KERNEL];
        double weight3_3[LAYER3][LAYER3][LENGTH_KERNEL][LENGTH_KERNEL];
        double weight3_4[LAYER3][LAYER3][LENGTH_KERNEL][LENGTH_KERNEL];


```

```
53          double weight4_1[LAYER3][LAYER4][LENGTH_KERNEL][LENGTH_KERNEL];
54          double weight4_2[LAYER4][LAYER4][LENGTH_KERNEL][LENGTH_KERNEL];
55          double weight4_3[LAYER4][LAYER4][LENGTH_KERNEL][LENGTH_KERNEL];
56          double weight4_4[LAYER4][LAYER4][LENGTH_KERNEL][LENGTH_KERNEL];
57
58          double weight5_1[LAYER4][LAYER5][LENGTH_KERNEL][LENGTH_KERNEL];
59          double weight5_2[LAYER5][LAYER5][LENGTH_KERNEL][LENGTH_KERNEL];
60          double weight5_3[LAYER5][LAYER5][LENGTH_KERNEL][LENGTH_KERNEL];
61          double weight5_4[LAYER5][LAYER5][LENGTH_KERNEL][LENGTH_KERNEL];
62
63          double fc1[LAYER5*LENGTH_FEATURE5*LENGTH_FEATURE5][FC1];
64          double fc2[FC1][FC2];
65          double fc3[FC2][OUTPUT];
66
67          double bias1_1[LAYER1];
68          double bias1_2[LAYER1];
69
70          double bias2_1[LAYER2];
71          double bias2_2[LAYER2];
72
73          double bias3_1[LAYER3];
74          double bias3_2[LAYER3];
75          double bias3_3[LAYER3];
76          double bias3_4[LAYER3];
77
78          double bias4_1[LAYER4];
79          double bias4_2[LAYER4];
80          double bias4_3[LAYER4];
81          double bias4_4[LAYER4];
82
83
84          double bias5_1[LAYER5];
85          double bias5_2[LAYER5];
86          double bias5_3[LAYER5];
87          double bias5_4[LAYER5];
88
89          double bias_fc1[FC1];
90          double bias_fc2[FC2];
91          double bias_fc3[OUTPUT];
92  }VGG;
93
94  typedef struct Feature
95  {
96          double input[INPUT][LENGTH_FEATURE0][LENGTH_FEATURE0];
97
98          double layer1_conv1[LAYER1][LENGTH_FEATURE0][LENGTH_FEATURE0];
99          double layer1_conv2[LAYER1][LENGTH_FEATURE0][LENGTH_FEATURE0];
100         double layer1_pool[LAYER1][LENGTH_FEATURE1][LENGTH_FEATURE1];
101
102         double layer2_conv1[LAYER2][LENGTH_FEATURE1][LENGTH_FEATURE1];
103         double layer2_conv2[LAYER2][LENGTH_FEATURE1][LENGTH_FEATURE1];
104         double layer2_pool[LAYER2][LENGTH_FEATURE2][LENGTH_FEATURE2];
105
106         double layer3_conv1[LAYER3][LENGTH_FEATURE2][LENGTH_FEATURE2];
107         double layer3_conv2[LAYER3][LENGTH_FEATURE2][LENGTH_FEATURE2];
108         double layer3_conv3[LAYER3][LENGTH_FEATURE2][LENGTH_FEATURE2];
109         double layer3_conv4[LAYER3][LENGTH_FEATURE2][LENGTH_FEATURE2];
110         double layer3_pool[LAYER3][LENGTH_FEATURE3][LENGTH_FEATURE3];
111
112         double layer4_conv1[LAYER4][LENGTH_FEATURE3][LENGTH_FEATURE3];
113         double layer4_conv2[LAYER4][LENGTH_FEATURE3][LENGTH_FEATURE3];
114         double layer4_conv3[LAYER4][LENGTH_FEATURE3][LENGTH_FEATURE3];
115         double layer4_conv4[LAYER4][LENGTH_FEATURE3][LENGTH_FEATURE3];
116         double layer4_pool[LAYER4][LENGTH_FEATURE4][LENGTH_FEATURE4];
117
118         double layer5_conv1[LAYER5][LENGTH_FEATURE4][LENGTH_FEATURE4];
119         double layer5_conv2[LAYER5][LENGTH_FEATURE4][LENGTH_FEATURE4];
120         double layer5_conv3[LAYER5][LENGTH_FEATURE4][LENGTH_FEATURE4];
121         double layer5_conv4[LAYER5][LENGTH_FEATURE4][LENGTH_FEATURE4];
122         double layer5_pool[LAYER5][LENGTH_FEATURE5][LENGTH_FEATURE5];
123
```

```
124         double fc1[FC1];
125         double fc2[FC2];
126         double output[OUTPUT];
127
128 }Feature;
129
130 void TrainBatch(VGG *vggnet, image *inputs, uint8 *labels, int batchSize);
131
132 void Train(VGG *vggnet, image input, uint8 label);
133
134 uint8 Predict(VGG *vggnet, image input, uint8 count);
135
136 void Initial(VGG *vggnet);
137 double relu(double x);
```

A.2.5 Transformer(MNIST)

```
1
2  #pragma once
3  #define PI 3.1415926
4  #define LENGTH_KERNEL   5
5  #define MAXPOOL_SIZE 2
6
7  #define PADDING 1
8  #define LENGTH_FEATURE0 28
9  #define LENGTH_FEATURE1_1       (LENGTH_FEATURE0 - LENGTH_KERNEL + 2*PADDING+1) //28-3+2+1 ->28*28
10 #define LENGTH_FEATURE1_2       (LENGTH_FEATURE1_1/2) //14*14
11
12 #define LENGTH_FEATURE2_1       (LENGTH_FEATURE1_2 - LENGTH_KERNEL +2*PADDING + 1) //14-3+2+1=14
13 #define LENGTH_FEATURE2_2       (LENGTH_FEATURE2_1/2)  //7*7
14
15 #define LENGTH_FEATURE3_1       (LENGTH_FEATURE2_2 - LENGTH_KERNEL +2*PADDING+ 1) //7*7
16
17 #define LENGTH_FEATURE4_1       (LENGTH_FEATURE3_1 - LENGTH_KERNEL +2*PADDING+ 1)//7*7
18
19 #define MAXPOOL5_2_KERNEL 3
20 #define MAXPOOL5_2_STRIDE 2
21
22 #define LENGTH_FEATURE5_1       (LENGTH_FEATURE4_1 - LENGTH_KERNEL+2*PADDING+ 1)        //7*7
23 #define LENGTH_FEATURE5_2       ((LENGTH_FEATURE5_1 - MAXPOOL5_2_KERNEL)/MAXPOOL5_2_STRIDE+ 1) //(7-3)/2 +1 = 3 no padding
24
25 #define FC      512
26
27
28 #define INPUT                   1
29 #define LAYER1                  16
30 #define LAYER2                  32
31
32 #define OUTPUT          10
33
34 #define ALPHA 0.01
35 #define RES1_CHANNEL 16
36 #define RES2_CHANNEL 32
37
38 #define RES_LENGTH_KERNEL 3
39
40 #define INPUTCHANNEL 1
41 #define PATCH_SIZE 7
42 #define PATCH_NUM  (28/PATCH_SIZE)*(28/PATCH_SIZE)
43 #define PATCH_DIM  INPUTCHANNEL*PATCH_SIZE*PATCH_SIZE
44 #define DIM 64
45 #define DEPTH 6
46 #define HEADER 8
47 #define MLP_DIM 128
48
49 typedef unsigned char uint8;
50 typedef uint8 image[28][28];
51
```

```
52  typedef struct ResBlock1
53  {
54          double weight1[RES1_CHANNEL][RES1_CHANNEL][RES_LENGTH_KERNEL][RES_LENGTH_KERNEL];
55          double weight2[RES1_CHANNEL][RES1_CHANNEL][RES_LENGTH_KERNEL][RES_LENGTH_KERNEL];
56          double bias1[RES1_CHANNEL];
57          double bias2[RES1_CHANNEL];
58  }ResBlock1;
59
60
61  typedef struct ResBlock2
62  {
63          double weight1[RES2_CHANNEL][RES2_CHANNEL][RES_LENGTH_KERNEL][RES_LENGTH_KERNEL];
64          double weight2[RES2_CHANNEL][RES2_CHANNEL][RES_LENGTH_KERNEL][RES_LENGTH_KERNEL];
65          double bias1[RES1_CHANNEL];
66          double bias2[RES1_CHANNEL];
67  }ResBlock2;
68
69
70  typedef struct Res1_Feature
71  {
72          double input_pad[RES1_CHANNEL][14][14];
73          double conv1[RES1_CHANNEL][12][12];
74
75          double conv1_pad[RES1_CHANNEL][14][14];
76          double conv2[RES1_CHANNEL][12][12];
77  }Res1_Feature;
78
79  typedef struct Res2_Feature
80  {
81          double input_pad[RES2_CHANNEL][6][6];
82          double conv1[RES2_CHANNEL][4][4];
83          double conv1_pad[RES2_CHANNEL][6][6];
84          double conv2[RES2_CHANNEL][4][4];
85  }Res2_Feature;
86
87
88
89  typedef struct ResNet
90  {
91          double weight1[INPUT][LAYER1][LENGTH_KERNEL][LENGTH_KERNEL];//Layer1 kernel
92          //maxpool
93          ResBlock1 res1;
94          double weight2[LAYER1][LAYER2][LENGTH_KERNEL][LENGTH_KERNEL];//Layer2 Kernel
95          //maxpool
96          ResBlock2 res2;
97
98          double fc[512][OUTPUT];
99
100
101         double bias1[LAYER1];
102         double bias2[LAYER2];
103
104
105         double bias_fc[OUTPUT];
106
107 }ResNet;
108
109
110
111
112 typedef struct Feature
113 {
114         double input[INPUT][LENGTH_FEATURE0][LENGTH_FEATURE0];
115         double conv1[LAYER1][24][24];
116         double max1[LAYER1][12][12];
117
118         Res1_Feature res1f;
119
120         double conv2[LAYER2][8][8];
121         double max2[LAYER2][4][4];
122         Res2_Feature res2f;
```

```
123
124         double output[10];
125
126 }Feature;
127
128
129 typedef struct ViT
130 {
131         double pos_embedding[PATCH_NUM+1][DIM];//    patchpos_embedding     ,
132         double patch_embedding_weight[INPUT][PATCH_DIM][DIM];//          patchlinear
133         double cls_token[1][DIM];//   patch_embedding
134         double patch_bias[DIM];
135
136         double q1x[HEADER][DIM][DIM];
137         double k1x[HEADER][DIM][DIM];
138         double v1x[HEADER][DIM][DIM];
139         double out_weight1[DIM][DIM];
140         double feed11[DIM][MLP_DIM];
141         double feed12[MLP_DIM][DIM];
142
143         double q2x[HEADER][DIM][DIM];
144         double k2x[HEADER][DIM][DIM];
145         double v2x[HEADER][DIM][DIM];
146         double out_weight2[DIM][DIM];
147         double feed21[DIM][MLP_DIM];
148         double feed22[MLP_DIM][DIM];
149
150
151         double q3x[HEADER][DIM][DIM];
152         double k3x[HEADER][DIM][DIM];
153         double v3x[HEADER][DIM][DIM];
154         double out_weight3[DIM][DIM];
155         double feed31[DIM][MLP_DIM];
156         double feed32[MLP_DIM][DIM];
157
158
159         double q4x[HEADER][DIM][DIM];
160         double k4x[HEADER][DIM][DIM];
161         double v4x[HEADER][DIM][DIM];
162         double out_weight4[DIM][DIM];
163         double feed41[DIM][MLP_DIM];
164         double feed42[MLP_DIM][DIM];
165
166         double q5x[HEADER][DIM][DIM];
167         double k5x[HEADER][DIM][DIM];
168         double v5x[HEADER][DIM][DIM];
169         double out_weight5[DIM][DIM];
170         double feed51[DIM][MLP_DIM];
171         double feed52[MLP_DIM][DIM];
172
173         double q6x[HEADER][DIM][DIM];
174         double k6x[HEADER][DIM][DIM];
175         double v6x[HEADER][DIM][DIM];
176         double out_weight6[DIM][DIM];
177         double feed61[DIM][MLP_DIM];
178         double feed62[MLP_DIM][DIM];
179
180         double mlp_w1[DIM][MLP_DIM];
181         double mlp_w2[MLP_DIM][OUTPUT];
182 }ViT;
183
184
185 typedef struct ViTFeature
186 {
187         double input[INPUT][28][28];
188         double input_patch[PATCH_NUM][INPUT][PATCH_SIZE][PATCH_SIZE];
189         double patch_embedding[PATCH_NUM][DIM];
190         double patch_embedding_cls[PATCH_NUM+1][DIM];
191         double q1[HEADER][PATCH_NUM+1][DIM];
192         double k1[HEADER][PATCH_NUM+1][DIM];
193         double v1[HEADER][PATCH_NUM+1][DIM];
```

```
194         double z1[HEADER][PATCH_NUM+1][DIM];
195         double attention_out1[PATCH_NUM+1][DIM];
196         double out1[PATCH_NUM+1][DIM];
197         double trans_feed11[PATCH_NUM+1][MLP_DIM];
198         double trans_feed12[PATCH_NUM+1][DIM];
199
200         double q2[HEADER][PATCH_NUM+1][DIM];
201         double k2[HEADER][PATCH_NUM+1][DIM];
202         double v2[HEADER][PATCH_NUM+1][DIM];
203         double z2[HEADER][PATCH_NUM+1][DIM];
204         double attention_out2[PATCH_NUM+1][DIM];
205         double out2[PATCH_NUM+1][DIM];
206         double trans_feed21[PATCH_NUM+1][MLP_DIM];
207         double trans_feed22[PATCH_NUM+1][DIM];
208
209
210         double q3[HEADER][PATCH_NUM+1][DIM];
211         double k3[HEADER][PATCH_NUM+1][DIM];
212         double v3[HEADER][PATCH_NUM+1][DIM];
213         double z3[HEADER][PATCH_NUM+1][DIM];
214         double attention_out3[PATCH_NUM+1][DIM];
215         double out3[PATCH_NUM+1][DIM];
216         double trans_feed31[PATCH_NUM+1][MLP_DIM];
217         double trans_feed32[PATCH_NUM+1][DIM];
218
219
220         double q4[HEADER][PATCH_NUM+1][DIM];
221         double k4[HEADER][PATCH_NUM+1][DIM];
222         double v4[HEADER][PATCH_NUM+1][DIM];
223         double z4[HEADER][PATCH_NUM+1][DIM];
224         double attention_out4[PATCH_NUM+1][DIM];
225         double out4[PATCH_NUM+1][DIM];
226         double trans_feed41[PATCH_NUM+1][MLP_DIM];
227         double trans_feed42[PATCH_NUM+1][DIM];
228
229         double q5[HEADER][PATCH_NUM+1][DIM];
230         double k5[HEADER][PATCH_NUM+1][DIM];
231         double v5[HEADER][PATCH_NUM+1][DIM];
232         double z5[HEADER][PATCH_NUM+1][DIM];
233         double attention_out5[PATCH_NUM+1][DIM];
234         double out5[PATCH_NUM+1][DIM];
235         double trans_feed51[PATCH_NUM+1][MLP_DIM];
236         double trans_feed52[PATCH_NUM+1][DIM];
237
238         double q6[HEADER][PATCH_NUM+1][DIM];
239         double k6[HEADER][PATCH_NUM+1][DIM];
240         double v6[HEADER][PATCH_NUM+1][DIM];
241         double z6[HEADER][PATCH_NUM+1][DIM];
242         double attention_out6[PATCH_NUM+1][DIM];
243         double out6[PATCH_NUM+1][DIM];
244         double trans_feed61[PATCH_NUM+1][MLP_DIM];
245         double trans_feed62[PATCH_NUM+1][DIM];
246
247         double mlp1[MLP_DIM];
248         double mlp2[OUTPUT];
249         double output[OUTPUT];
250
251 }ViTFeature;
252
253
254
255
256 void TrainBatch(ViT *vit, image *inputs, uint8 *labels, int batchSize);
257
258 void Train(ViT *vit, image input, uint8 label);
259
260 uint8 Predict(ViT *vit, image input, uint8 count);
261
262 void Initial(ViT *vit);
```

Appendix B

Appendix for additional theory

B.1 Softmax and Entropy Loss

Assume the n-output of CNNs: $\{o_0, o_1, ...o_{n-1}\}$, thesis's deep learning apply softmax and entropy loss as the loss function. Then the probabilities distribution of outputs are modified as:

$$p_i = \frac{\partial e^{o_i}}{\sum_{k=0}^{n-1} e^{o_k}} \tag{B.1}$$

Assume the one-hot label: $\{l_0, l_1, ...l_{n-1}\}$, where $l_j = 1, l_i = 0(i \neq j)$ The Entropy Loss L are:

$$L = -\sum_{i=0}^{n-1} l_i \ln p_i \tag{B.2}$$

And the gradients L versus output o are:

$$\frac{\partial L}{\partial o_i} = \begin{cases} p_i, i \neq j \\ p_j - 1, i = j \end{cases}$$