Towards Efficient and Secure Large-Scale Systems for Distributed Machine Learning Training

by

Chengliang Zhang

A Thesis Submitted to
The Hong Kong University of Science and Technology
in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy
in the Department of Computer Science and Engineering

March 2021, Hong Kong

Authorization

I hereby declare that I am the sole author of the thesis.

I authorize the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Signature redacted

Chengliang Zhang

March 2021

Towards Efficient and Secure Large-Scale Systems for Distributed Machine Learning Training

by

Chengliang Zhang

This is to certify that I have examined the above PhD thesis
and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by
the thesis examination committee have been made.

Signature redacted

Prof. Wei Wang, Thesis Supervisor

Signature redacted

Prof. Dit-Yan Yeung, Head of Department

Department of Computer Science and Engineering
March 2021

ACKNOWLEDGMENTS

First and foremost, I am profoundly grateful to my supervisor Professor Wei Wang, who has inspired and guided me continuously throughout my PhD study. Professor Wang's passion towards research and tremendous work ethic have never ceased to amaze me and inspire me to become a better researcher. I am beyond fortunate to work with such a brilliant researcher and benefit from his expertise and perspectives. I am indebted to his encouragement and support during the highs and lows I have been through. I shall forever cherish and benefit from every conversations we have had together in the past 4 years.

I also extend my sincere gratitude to Professor Feng Yan. Professor Yan has been a close collaborator and sharing insightful comments since the very first research project led by me. I am constantly impressed by the high standards he has kept for academic research. I owe a huge part of my PhD work to his selfless guidance. I would like to also thank his student Jun Yi for including me on the project MLCD.

My sincere appreciation goes to the mentors at Bell Labs Germany as well, including Dr. Ruichuan Chen, Dr. Istemi Ekin Akkus, and Dr. Paarijaat Aditya. They helped me shape the research idea of Citadel and navigate through it, the weekly online sessions were extremely enlightening and kept me motivated despite the ongoing pandemic.

I am fortunate enough to work with the most diligent and smart people at HKUST. I am grateful for Dr. Yinghao Yu for embracing me in his project and helping me get an early start. Special thanks to Huangshi Tian, Minchen Yu, Suyi Li, Junzhe Xia, Baichen Yang, Huancheng Puyang, for their respective contributions towards my projects SpecSync, MArk, BatchCrypt, and Citadel. I could not finish these projects without their generous assistance. I am thankful for other talented members of our fantastic research group SysAlg for their kind advice and pleasant company. The list is in the order of seniority: Chen Chen, Da Yan, Mingzhe Li, Qizen Weng, Zhifeng Jiang, and Yunchuan Zheng.

Last but not least, my deepest gratitude to my family. Any achievement I have would not be possible without their unconditional love and support. This dissertation is dedicated to them.

TABLE OF CONTENTS

Title Page ii

Authorization Page ii

Signature Page iii

Acknowledgments iv

Table of Contents v

List of Figures ix

List of Tables xi

Abstract xii

Chapter 1 Introduction 1

1.1 Why Efficiency and Privacy Matters in Distributed Machine Learning Training 1

1.2 Three Challenges for Large Scale ML Training 2

1.2.1 Large-Scale ML Training in Datacenters 2

1.2.2 Cross-silo FL with HE 3

1.2.3 Preserving Both Data and Model Privacy 4

1.3 Contributions 4

1.3.1 SpecSync: Speculative Synchronization for High Efficiency 4

1.3.2 BatchCrypt: Efficient HE for Cross-Silo Federated Learning 5

1.3.3 Citadel: Protecting Data Privacy and Model Confidentiality with SGX 5

1.4 Thesis Outline 6

1.5 List of Related Publications 6

1.6 List of Other Publications 6

Chapter 2 Speculative Synchronization for Fast Distributed Machine Learning **8**

2.1 Background and Motivation 8

2.1.1 ML Problems Solved by Risk Minimization 8

2.1.2 Parameter Server and Distributed SGD 8

2.1.3 Synchronization Schemes 9

2.2 Staying Fresh through Naïve Waiting 11

2.2.1 Pushes after a Pull: The Source of Staleness 11

2.2.2 Naïve Waiting 13

2.3 Speculative Synchronization 14

2.3.1 Overview 15

2.3.2 Adaptive Hyperparameter Tuning 16

2.4 Implementation 19

2.4.1 Architecture Overview 20

2.4.2 Workflow 21

2.5 Evaluation 23

2.5.1 Experiment Setup 23

2.5.2 Effectiveness of SpecSync 24

2.5.3 Robustness of SpecSync 25

2.5.4 Communication Overhead 27

2.5.5 SpecSync-Cherrypick vs. SpecSync-Adaptive 28

2.5.6 Discussion 29

2.6 Related Work 29

2.7 Summary 30

Chapter 3 Efficient Homomorphic Encryption for Cross-Silo Federated Learning **31**

3.1 Background and Related Work 31

3.1.1 Cross-Silo Federated Learning 31

3.1.2 Privacy Solutions in Federated Learning 32

3.1.3 Cross-Silo FL Platform with HE 34

3.2 Characterizing Performance Bottlenecks 34

3.2.1 Characterization Results 35

3.2.2 Potential Solutions and Their Inefficiency 38

3.3 BatchCrypt 39
3.3.1 Why is HE Batching for FL a Problem? 39
3.3.2 HE Batching for Gradients 41
3.3.3 dACIQ: Analytical Clipping for FL 43
3.3.4 BatchCrypt: Putting It All Together 46
3.4 Implementation 46
3.5 Evaluation 48
3.5.1 Methodology 48
3.5.2 Impact of BatchCrypt's Quantization 49
3.5.3 Effectiveness of BatchCrypt 51
3.5.4 Batching Efficiency 54
3.5.5 Cost Benefits 55
3.6 Discussion 56
3.7 Summary 56

Chapter 4 Protecting Data Privacy and Model Confidentiality for Collaborative Learning with SGX 58
4.1 Background and Related Work 58
4.1.1 Collaborative ML and Threat Model 58
4.1.2 Entities in Collaborative ML 58
4.1.3 Threat Model 59
4.2 Prior Arts and Their Insufficiency 60
4.2.1 Existing Solutions for Different Collaborative Learning Scenarios 61
4.2.2 Intel SGX 61
4.2.3 Private ML with a Single SGX Enclave 63
4.2.4 Private ML with Multiple SGX Enclaves 65
4.3 Citadel Design 65
4.3.1 Design Overview 66
4.3.2 Separating Data and Model Handling 68
4.4 Implementation 73
4.5 Evaluation 76
4.5.1 Methodology 76
4.5.2 Effectiveness of Zero-Sum Masking 77

4.5.3 Effectiveness of Hierarchical Aggregation 78
4.5.4 Citadel vs. Single Enclave 80
4.5.5 SGX Overhead in Citadel 81
4.6 Discussion 82
4.7 Summary 82

Chapter 5 Conclusions and Future Directions **83**
5.1 Future Directions 84

References **85**

LIST OF FIGURES

2.1 The architecture of Parameter Server (PS). 9

2.2 Asynchrony hides pushes after a pull (PAP). Worker-1 misses more PAP than anyone else and ends up with the most outdated parameters. 11

2.3 Distribution of the number of pushes received in a time interval after a pull is made (PAP). Each interval spans one second. 12

2.4 Naïve waiting defers each pull request by a short period of time, allowing worker-1 (and others) to uncover more updates than it could have had in fig. 2.2. 13

2.5 Convergence curves of naïve waiting with different delay times. Curves with zero delay correspond to the stock MXNet implementation. 13

2.6 Illustration of speculative synchronization. Worker-1 speculatively aborts computation after observing the two pushes made by two peers. It pulls parameters again and starts over. 14

2.7 The architecture of speculative synchronization. 20

2.8 Loss (left plot in each sub-figure) and runtime (right plot in each sub-figure) comparison of (a) MF, (b) CIFAR-10, and (c) ImageNet. For better visualization, we only show the results up to when one of the approaches is converged (i.e., loss is below the target value for 5 consecutive iterations). 22

2.9 Loss as a function of iteration number (left plot) and accumulated iteration number (right plot) for different synchronization schemes using CIFAR-10. 25

2.10 Loss comparison for different synchronization schemes running CIFAR-10 in both homogeneous and heterogeneous clusters. 26

2.11 Runtime speedup for achieving the same training accuracy target (left plot) and loss improvement with the same training time (right plot) for SpecSync-Adaptive over Original using CIFAR-10 as workload under different cluster size. 27

2.12 Accumulated data transfer over time for different workloads under different schemes. 27

2.13 Overall data transfer breakdown for SpecSync-Adaptive. 28

3.1 The architecture of cross-silo FL system, where HE is implemented as a pluggable module on the clients. 33

3.2 Iteration time breakdowns of **F**MNIST, **C**IFAR, and **L**STM for a client and the aggregator. 36

3.3 An illustration of a generic quantization scheme and BatchCrypt. The latter preserves additivity during batching, with the sign bits highlighted within values. 41

3.4 A typical layer gradient distribution. α is the clipping threshold. 43

3.5 The architecture of a client worker in BatchCrypt. 46

3.6 The quality of trained model with different quantization bit widths in BatchCrypt. 50

3.7 Breakdown of training iteration time under stock FATE and BatchCrypt, where "idle" measures the idle waiting time of a worker and "agg." measures the gradient aggregation time on the aggregator. Note that model computation is left out here as it contributes little to the iteration time. 51

3.8 Comparison of the network traffic incurred in one training iteration using the stock FATE implementation and BatchCrypt. 52

3.9 Time and communication comparisons of one iteration on workers between BatchCrypt and plain distributed learning without encryption. 53

3.10 Breakdown of iteration time and communication traffic of BatchCrypt with LSTM model with various quantization bit widths in one iteration. The corresponding batch sizes for bit width 8, 16, and 32 are 200, 100, and 50, respectively. 54

3.11 Total cost until convergence between FATE's stock implementation and BatchCrypt, **ins**tance and **net**work costs are highlighted separately. 55

4.1 An illustration of a single-enclave solution that protects the confidentiality of both data and training model. 63

4.2 The time needed to finish one epoch training running under SGX and native mode respectively. The slowdown shown in line represents the ratio between SGX time and native time. The memory figure depicts the amount of memory is used actively in SGX. 64

4.3 An architecture overview of Citadel. All codes (except the model update code) are open-sourced. 66

4.4 The workflow of Citadel with zero-sum masking, enclave TLS connections terminate within enclaves. 75

4.5 The iteration time breakdown w.r.t. training enclave numbers when the zero-sum masking is adopted. 78

4.6 The iteration time breakdown of different models w.r.t. aggregation children number when hierarchical aggregation is adopted. All experiments are run with 32 training enclaves. The zero-sum mask results with 32 training enclaves are shown as M bars for reference. 79

4.7 The total throughput normalized with the `single-enclave` solution throughput (labeled as S) w.r.t. training enclave number. 80

LIST OF TABLES

2.1 Models and datasets used for workloads. The iteration time is based on `m4.xlarge` instance. 22

2.2 Cost of hyperparameter exhaustive search using SpecSync-Cherrypick. 29

3.1 Benchmarking Paillier HE with various key sizes. 37

3.2 Network bandwidth (Mbit/sec) between aggregator and clients in different regions. 49

3.3 Summary of models used in characterizations. 49

3.4 Projected total training time and network traffic usage until convergence for the three models. The converged test accuracy for FMNIST, CIFAR as well as loss for LSTM and their corresponding epoch numbers are listed in the table. 53

4.1 The slowdowns of Citadel at different scale. The 32-R column shows the hierarchical aggregation implementation, while the rest show zero-sum masking. 81

Towards Efficient and Secure Large-Scale Systems for Distributed Machine Learning Training

by Chengliang Zhang

Department of Computer Science and Engineering

The Hong Kong University of Science and Technology

Abstract

Machine learning (ML) techniques have advanced in leaps and bounds in the past decade. Its success critically relies on the abundant computing power and the availability of big data, it is impractical to host ML training on a single machine, and a sole data source usually does not produce a general enough model. By distributing ML workload across multiple machines and utilizing data across multiple silos, we can substantially improve the quality of ML training. As large-scale ML training is increasingly deployed in production systems involving multiple entities, how to improve efficiency, and ensure the confidentiality of the participants become the pressing needs. First, how to efficiently train an ML model in a cluster with the presence of heterogeneity? Second, in the context of federated learning (FL) where multiple *data owners* collaboratively train a model together, how to mitigate the overhead introduced by the privacy-preserving techniques? Lastly, in the nuance case where many organizations who own data but not ML expertise would like to pool their data and collaborate with those who have expertise (*model owner*) to train generalizable models, how to protect the model owner's intellectual property (*model privacy*) while preserving the data privacy of data owners?

General ML training solutions find themselves inadequate under the efficiency and

privacy challenges posed by distributed ML. First, traditional distributed ML systems often conduct asynchronous training to mitigate the impact of stragglers. While it maximizes the training throughput, the price paid is degraded training quality as there are inconsistency across workers. Second, although techniques like Homomorphic Encryption (HE) can be conveniently adopted to preserve data privacy in FL, they induce prohibitively high computation and communication overheads. Third, there is yet to be a practical solution that can protect model owner's intellectual properties without compromising data owner's privacy.

To fill in the gaps mentioned above, we profile, analyze, and propose new strategies to improve training efficiency and privacy guarantees.

To improve the efficiency in distributed asynchronous training, we first propose a new distributed synchronization scheme, termed *speculative synchronization*. Our scheme allows workers to speculate about the recent parameter updates from others on the fly, and if necessary, the workers *abort* the ongoing computation, pull fresher parameters, and *start over* to improve the quality of training. We implement our scheme and demonstrate that speculative synchronization achieves substantial speedups over the *asynchronous parallel* scheme with minimal communication overhead.

Second, we present BatchCrypt, a system solution for cross-silo FL that significantly reduces the encryption and communication overhead caused by HE. Instead of encrypting individual gradients with full precision, we *encode a batch of quantized gradients* into a long integer and encrypt it in one go. To allow *gradient-wise aggregation* to be performed on *ciphertexts* of the encoded batches, we develop new quantization and encoding schemes along with a novel gradient clipping technique. Our evaluations confirm that BatchCrypt can effectively reduce the computation and communication overhead.

Lastly, to address the collaborative learning cases where model privacy is also concerned, we devise a scalable system Citadel. Citadel protects privacy for both data and model owner in untrusted infrastructures with the help of Intel SGX. Citadel performs training across multiple *training enclaves* running on behalf of data owners and an *aggregator enclave* on behalf of the model owner. Citadel further establishes a strong information barrier between these enclaves using *zero-sum masking* or *hierarchical aggregation* to prevent data/model leakage during collaborative training. We deploy Citadel on cloud to train various ML models, and prove it is scalable while providing strong privacy guarantees.

CHAPTER 1

INTRODUCTION

1.1 Why Efficiency and Privacy Matters in Distributed Machine Learning Training

Large-scale machine learning (ML) has demonstrated state-of-the-art performance in many practical applications, such as voice-driven personal assistants [1], photo search [2] and captioning [3], and autonomous vehicles [4]. Building premium ML models requires not only extensive ML expertise in feature selection, model design, hyperparameter tuning and testing, but also a large volume of high-quality training data from diverse sources. Distributed training is essential to accommodate such workloads. In distributed ML systems [5, 6, 7, 8, 9], the training data is dispersed across many *worker* nodes. All workers share access to the model parameters, sharded across multiple *servers*. Each worker iteratively refines the parameters based on its subset of training data, and communicates the refinement with parameter servers [6], in parallel with other workers. Naturally, distributed ML training faces the challenge of how to efficiently communicate among these workers and preserve high resource utilization.

Besides efficiency, privacy is another prominent concern for distributed ML. In many industries, data is dispersed and locked in multiple *data owners* (e.g., banks, hospitals, and institutes), where data sharing is strictly forbidden due to the growing concerns about data privacy as well as violating the government regulations [10, 11, 12]. Cross-silo federated learning (FL) [13, 14] offers an appealing solution to break "data silos" among organizations, where data owners *collaboratively learn* a global model without sharing privacy-sensitive data. To ensure that no client reveals its sensitive data, many approaches have been proposed [15, 16, 17, 18, 19]. Among them *additively homomorphic encryption* (HE) is particularly attractive in the cross-silo setting [18, 19, 13], as it provides a strong privacy guarantee at no expense of learning accuracy. However, HE causes huge overheads in

both communication and computation thanks to its cryptographic nature, which impedes its adoption in industrial applications.

Although FL offers *data privacy* for data owners, it does not address all privacy concerns in distributed ML. There are often cases where data owners have no sufficient ML expertise and have to collaborate with an ML solution provider (*model owner*) who devises ML model and training strategy. For example, hospitals collaborate with an IT firm to train a diagnostic imaging model over their patients' data [20]. For model owner here, the model is valuable intellectual property [21, 22, 23]. Revealing proprietary model details (e.g., architecture and weights) can potentially result in losing technological advances to its market competitors.

Efficiency is essential to deliver trained ML model swiftly, while privacy guarantees enable collaboration among various data owners and model owner otherwise impossible. We elaborate the efficiency and privacy challenges and research gaps that exist in distributed ML training in the following sections.

1.2 Three Challenges for Large Scale ML Training

To achieve efficient and secure large scale ML training, there are many challenges, we focus on the following three in this thesis. First, for distributed training, to maximize training efficiency, communication among workers must be judiciously studied. Second, for cross-silo FL with HE, whether we can alleviate the immense HE computation and communication overhead dictates the scale of model supported. Third, for scenarios where both data privacy and model confidentiality are required, how do we enable training without trust among the participants. In the rest of this section, we elaborate the unique challenges regarding each of the three missions and demonstrate why existing works fail in the context of large-scale deployment.

1.2.1 Large-Scale ML Training in Datacenters

Ideally in distributed training, to ensure high-quality updates from all workers, the computation should use the up-to-date model parameters. This can be achieved through a

Bulk Synchronous Parallel (BSP) implementation, where workers synchronize at the end of each iteration and will not proceed until the model parameters have been fully updated by *all* workers. However, BSP-style solution suffers from high synchronization overhead that makes it *impractical* to solve large ML problems [24, 5]: the presence of straggling workers inevitably slows down the entire learning progress. Therefore, prevalent ML systems [5, 9, 7, 8, 6] employ an Asynchronous Parallel (ASP) model, where workers eagerly start the next iteration without waiting for the others. In this way, the rate of update is maximized, leading to faster convergence than the BSP approach in many ML problems. However, without synchronization, updates produced on the inconsistent parameters may drive ASP's model away from the optimum [24, 25, 26].

Aiming at striking a balance between updates rate and updates quality, Stale Synchronous Parallel (SSP) model [24, 6, 27] is proposed as a middle ground between the ASP and BSP approach. In the SSP model, workers synchronize *only when* the *staleness* of parameters (measured by the number of missing updates from stragglers) exceeds a certain threshold. While this allows fast workers to use relatively fresher parameters, it provides little benefit for straggling machines. Consequently, updates generated by slowed machines may harm rather than benefit the training progress [26].

1.2.2 Cross-silo FL with HE

Although HE provides a strong privacy guarantee for cross-silo FL, it performs complex cryptographic operations (e.g., modular multiplications and exponentiations) that are extremely expensive to compute. Our testbed characterization shows that more than 80% of the training iteration time is spent on encryption/decryption. To make matters worse, encryption yields substantially larger ciphertexts, inflating the amount of data transfer by over 150× than plaintext learning. The significant overhead of HE in *encryption* and *communication* has become a major roadblock to facilitating cross-silo FL. According to our contacts at WeBank [28], most of their FL applications cannot afford to use the encrypted gradients and are limited to scenarios with less stringent privacy requirements (e.g., FL across departments or trustworthy partners). Existing work [29] proposes to use a dedicated hardware device (e.g., FPGA) for accelerated encryption/decryption, but the reported speedup remains insufficient given the dominance of the encryption overhead.

1.2.3 Preserving Both Data and Model Privacy

Model confidentiality is mostly unconsidered in prevalent solutions for collaborative ML, such as federated learning [13, 30, 14] and split learning [31, 32], where the training model needs to be shared fully or partially among participants.

Trusted hardware, such as Intel Software Guard Extensions (SGX) [33], has been used for collaborative ML with privacy guarantees for both data and model owners. A common approach is to perform ML training inside a single SGX enclave, where training data and the model are loaded first [34]. However, this solution does not scale to large models nor large training datasets, due to the restricted size of the enclave page cache (EPC) and excessive cryptographic overheads. Other approaches [35, 36, 37] manage to distribute training amongst multiple enclaves but operate under a weaker threat model, thus is not sufficient for protecting both data privacy and model confidentiality.

1.3 Contributions

In this dissertation, we strive to identify the fundamental system design and performance issues in the three important large scale ML system deployment scenarios, and come up with effective solutions. We summarize our key contributions as follows.

1.3.1 SpecSync: Speculative Synchronization for High Efficiency

To begin with, we explore a new aspect to improve the efficiency of distributed ML. We observe that in asynchronous training, a worker pulls fresh parameters from servers only before the start of each iteration, which hides all the updates from other workers during the iteration. Following this observation, we propose speculative synchronization (SpecSync), where each worker speculates about the parameter updates from others, and if necessary, it aborts the ongoing computation, pulls fresher parameters to start over, so as to opportunistically improve the quality of training. Our evaluations show that SpecSync can significantly accelerate the training speed by up to 3$\times$ without compromising on training accuracy.

1.3.2 BatchCrypt: Efficient HE for Cross-Silo Federated Learning

Next, we enable efficient cross-silo FL with HE using a simple *batch encryption* technique. Gradients are first *quantized*, *coded* into long integers so that the encryption overhead is significantly reduced. First, we design a customized quantization scheme just for FL, which is less prone to overflowing and more flexible to decrypt. Second, as gradients values are *unbounded*, they must be clipped before quantization. We propose an efficient analytical model dACIQ by extending ACIQ [38], a state-of-the-art clipping technique for ML over centralized data, to cross-silo FL over decentralized data. dACIQ allows us to choose optimal clipping thresholds with the minimum cumulative error. Compared with the stock implementation, BatchCrypt accelerates cross-silo FL training by up to 2 orders of magnitudes. In the meantime, the communication overhead is reduced by up to 100×. The significant benefits of BatchCrypt come at no cost of model quality, with a negligible accuracy loss less than 1%.

1.3.3 Citadel: Protecting Data Privacy and Model Confidentiality with SGX

Finally, we present a novel scalable system design Citadel to provide both data and model confidentiality for collaborative learning. First, to earn data owners' trust while protecting model confidentiality, we introduce two methods, *zero-sum masking* and *hierarchical aggregation*, to isolate codes handling data and model, and run the two parts in separate enclaves. Therefore, only codes that have direct access to data shall be shared with data owners to gain trust, while model handling codes remain private to model owner. Second, with data processing codes singled out, we can scale multiple such enclaves concurrently to process data in parallel, and aggregate the intermediate results together to update the model. Third, we employ techniques like hierarchical aggregation, multi-threading with a pre-compiled C library to make ML workloads adapt to SGX's memory constraints. We implement Citadel atop SCONE [39], and confirm that it can effectively speed up training by adding more enclaves. With 32 training enclaves running, we can boost the throughput to 4.7×–19.6× of those running in a single enclave.

1.4 Thesis Outline

The remainder of this dissertation is organized as follows. In §2, we present SpecSync, a novel distributed training synchronization scheme that improves efficiency. In §3, we propose BatchCrypt to achieve efficient and secure cross-silo FL. In §4, we showcase Citadel, a scalable collaborative training system that ensures both data privacy and model confidentiality with SGX. We make the final remarks in §5.

1.5 List of Related Publications

Conference Papers

1. **C. Zhang**,S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu, "BatchCrypt: Efficient Homomorphic Encryption for Cross-Silo Federated Learning," in *Proc. USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2020.

2. **C. Zhang**, H. Tian, W. Wang, and F. Yan, "Stay Fresh: Speculative Synchronization for Fast Distributed Machine Learning," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Vienna, Austria, July 2018.

3. **C. Zhang**, J. Xia, B. Yang, H. Puyang, W. Wang, R. Chen, I. Akkus, P. Aditya, and F. Yan, "Citadel: Protecting Data Privacy and Model Confidentiality for Collaborative Learning with SGX," submitted and under review.

1.6 List of Other Publications

Conference Papers

1. **C. Zhang**, M. Yu, W. Wang, and F. Yan, "MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving," in *Proc. USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

2. J. Yi, **C. Zhang**, W. Wang, C. Li, and F. Yan, "Not All Explorations Are Equal: Harnessing Heterogeneous Profiling Cost for Efficient MLaaS Training," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, New Orleans, LA, May 2020.

Journal Articles

1. **C. Zhang**, M. Yu, W. Wang, and F. Yan, "Enabling Cost-Effective, SLO-Aware Machine Learning Inference Serving on Public Cloud," in *IEEE Trans. Cloud Comput.*.

2. Y. Yu, **C. Zhang**, W. Wang, J. Zhang, and K. Letaief, "Towards Dependency-Aware Cache Management for Data Analytics Applications," in *IEEE Trans. Cloud Comput.*.

CHAPTER 2

SPECULATIVE SYNCHRONIZATION FOR FAST DISTRIBUTED MACHINE LEARNING

In this chapter, we first look at the inefficiency problem in distributed ML.

2.1 Background and Motivation

In this section, we briefly introduce the background of distributed machine learning (ML) and the means to facilitate it in large clusters through the Parameter Server (PS) architecture [5, 6, 7, 8, 9].

2.1.1 ML Problems Solved by Risk Minimization

In many learning problems, the input is a training dataset $\mathcal{D}$ consisting of n samples. A sample is a vector where each component characterizes a *learning feature*. Each sample x is associated with a label y. The objective of learning is to find a model with parameters w that correctly predicts label y given sample x. The learned model can then be used to predict y for any future x not seen in the training dataset.

To learn the model, the training algorithm solves a *risk minimization* problem. In particular, we define a loss function $l(x, y, w)$ that measures the prediction error (risk) if model w is used to predict label y given sample x. Our goal is to find the best model w that results in the minimum prediction errors over the entire training samples, i.e.,

$$\text{minimize}_w \sum_{x \in \mathcal{D}} l(x, y, w). \tag{2.1}$$

2.1.2 Parameter Server and Distributed SGD

To expedite training ML models on very large data sets, distributed ML systems have been proposed where the training is distributed over a cluster of commodity machines [5, 8, 6].

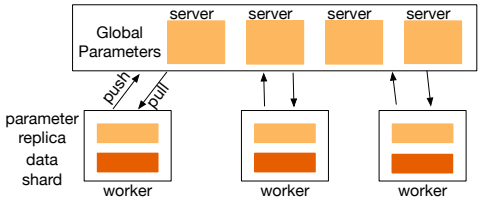


Figure 2.1: The architecture of Parameter Server (PS).

The recently proposed Parameter Server (PS) architecture can be employed to facilitate the system design [5, 6, 7, 8, 9].

As shown in fig. 2.1, in the PS-based systems, training samples $\mathcal{D}$ are partitioned into a number of subsets $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_m$, each maintained by a *worker* machine. The model parameters are sharded across multiple *servers*, and can be accessed by all workers. Each worker also maintains a local replica of model parameters and iteratively refines it based on its own training samples. Periodically, workers push their local updates to servers, jointly refining the global parameters. The worker then pulls the most up-to-date parameters from servers and proceeds to the next iteration of training.

More precisely, workers perform *distributed SGD* (stochastic gradient descent) with *data parallelism* [40] in the PS-based ML systems. Each worker i iteratively passes over its own training samples $\mathcal{D}_i$. In each pass, the worker calculates the subgradient $\nabla l(x, y, w)$ for each sample x and updates the model with

$$w \leftarrow w - \eta \sum_{x \in \mathcal{D}_i} \nabla l(x, y, w), \tag{2.2}$$

where η is the *learning rate*. Every time a worker pushes an update, we say it finishes one *iteration*; when *all* workers finish an iteration, we say the training has gone through one *epoch*.

2.1.3 Synchronization Schemes

Asynchronous Parallel (ASP). Most popular PS-based ML systems [5, 6, 7, 8, 9] adopt Asynchronous Parallelism when perform distributed SGD, where each worker eagerly proceeds to the next iteration without waiting for the parameter updates from other workers. ASP-style execution fully exploits the computing cycles, maximizing the rate of update. However, the price paid is the *compromised quality* of learned models. In cluster

environments, it is common to have workers process training samples at different speeds and make push and pull requests at different rates. This results in *inconsistent* model replicas among workers. By the time a fast worker finishes one iteration and pulls parameters from servers, some slowed machines may remain in the middle of iteration. The fast worker hence misses the updates from those machines and proceeds to the next iteration with stale version of parameters. Training with stale parameters may poison the algorithm throughput as pointed in the literature [25], compromising training quality.

Bulk Synchronous Parallel (BSP) comes as an alternative scheme that enforces a consistent, up-to-date view of global parameters across workers. With the BSP scheme, workers synchronize at the end of each iteration and cannot proceed until all workers have pushed updates to servers. The BSP scheme ensures quality updates from each iteration, and is widely adopted in parallel analytics frameworks such as MapReduce [41], Spark [42], MLlib [43] and GraphX [44]. However, the BSP scheme incurs high synchronization overhead: fast workers must wait for stragglers to complete, wasting their computing cycles in idle. For this reason, the BSP scheme often performs poorly on large ML problems [24, 27].

Stale Synchronous Parallel (SSP) [24, 6, 27, 45] is proposed recently as a middle ground between the ASP and BSP schemes. In the SSP scheme, workers can asynchronously start next iteration with stale parameters, provided that the *staleness* (measured by the worker's progress ahead of the straggler) is within a bounded amount. The SSP scheme allows fast workers to timely discover updates from slowed machines, and is shown to provide a convergence guarantee as opposed to the ASP approach [24]. However, slowed workers may still lag behind with a rather inconsistent view to the global parameters. Prior work [26] shows that updates from slowed workers are often harmful rather than beneficial, especially when the parameters are coming close to the optimum. In fact, popular ML systems like TensorFlow [9] and MXNet [7] choose *not* to implement SSP based on the claim that only in rare cases can the improvement on training speed or convergence be observed [46].

To summarize, relaxed synchronization schemes can effectively speed up distributed learning, but at the cost of compromised quality due to inconsistent model replicas among

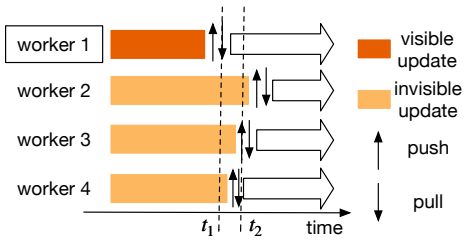


Figure 2.2: Asynchrony hides pushes after a pull (PAP). Worker-1 misses more PAP than anyone else and ends up with the most outdated parameters.

workers [45]. This motivates us to propose a new approach to accelerate asynchronous distributed learning by enabling more up-to-date views of global parameters across workers.

2.2 Staying Fresh through Naïve Waiting

In this section, we examine the behaviors of asynchronous learning through empirical studies and explore a new aspect to keep parameters fresh in computation.

2.2.1 Pushes after a Pull: The Source of Staleness

Without synchronization, each worker eagerly pulls parameters to start next iteration, and will miss the following pushes made by others before the next pull. That is, asynchrony hides *pushes after a pull* (PAP)—the main source where staleness derives. Fig.2.2 illustrates an example. The parameters worker-1 pulls at time t_1 have received only one update from itself, while the ones worker-2 pulls include four updates from all workers and are much fresher. In fact, worker-1 has the most outdated parameters, as it misses more PAP than anyone else.

Intuitively, to stay fresh, a worker should uncover more recent pushes made by others. A simple way to do so is to *defer the pull request by a small amount of time*, to capture more recent pushes that are otherwise invisible. In the previous example, delaying the pull request of worker-1 to t_2 exposes the two pushes from worker-3 and worker-4, enabling a more up-to-date view of the global parameters.

However, the deferral of a pull request inevitably delays the start of an iteration, lead-

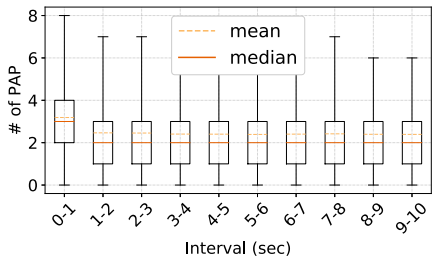


Figure 2.3: Distribution of the number of pushes received in a time interval after a pull is made (PAP). Each interval spans one second.

ing to longer completion time. Without a careful control, the harm caused by the delay may outweigh the benefits derived from fresher parameters. Therefore, whether deferring a pull request is beneficial critically depends on how many pushes are expected to be made shortly after the pull.

We investigate this issue through an empirical study using a real ML application. We trained a deep residual network with CIFAR-10 dataset [47] in MXNet [7], deployed in an Amazon EC2 cluster consisting of 40 `m4.xlarge` instances.[1] The training is performed in an ASP model. We collected the workload traces and analyzed how many pushes were made by others between two consecutive pulls of a worker (i.e., the number of missing updates in one iteration). In our experiment, an iteration typically spans around 14 seconds. We further divide an iteration into several 1-second intervals (i.e., 0-1s, 1-2s, etc.), and for each interval, we count the number of PAP received in it. We show the distribution of each interval as a box plot in fig. 2.3, where boxes depict 25th, 50th, and 75th percentiles, and whiskers depict 5th and 95th percentiles. We observe approximately uniform arrivals of PAP in each interval. In particular, if we focus on the number of pushes received within two seconds after a pull is made (the first two boxes in fig. 2.3), the median is over 6. That is to say, by delaying the pull request by 14% of the iteration time, we can allow 50% of workers to include at least 15% of recent updates. These promising numbers suggest that a slight delay of iteration is sufficient to discover more fresh parameters.

[1] In MXNet, each node is both a worker and a server.

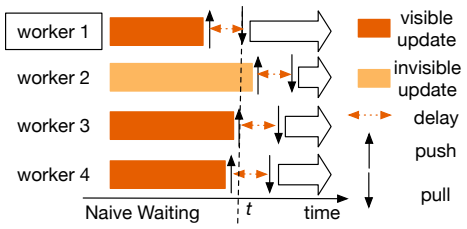
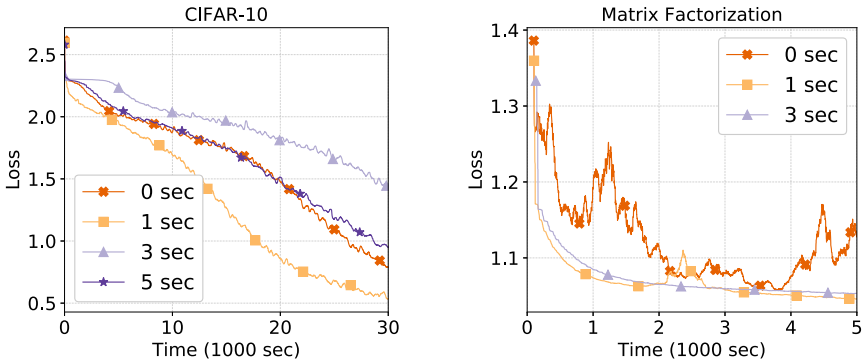


Figure 2.4: Naïve waiting defers each pull request by a short period of time, allowing worker-1 (and others) to uncover more updates than it could have had in fig. 2.2.



(a) CIFAR-10 (b) Matrix Factorization

Figure 2.5: Convergence curves of naïve waiting with different delay times. Curves with zero delay correspond to the stock MXNet implementation.

2.2.2 Naïve Waiting

Motivated by our empirical studies, we propose a simple strategy which we call *naïve waiting*. As the name suggests, each worker simply delays its pull requests by a short period of time, so as to uncover more pushes that are otherwise invisible. Fig. 2.4 illustrates naïve waiting applied to the example in fig. 2.2. We see that a slight delay of pulls allows *all* workers to include more updates in their parameters than they could in fig. 2.2: worker-1 now uncovers three updates (highlighted in fig. 2.4), while the other three workers get all four (not shown in fig. 2.4).

To quantify the benefits of naïve waiting in real-world systems, we implemented it in MXNet 0.7 [48] and evaluated its performance in an EC2 cluster composed of 40 `m4.xlarge` instances. We ran two benchmarking ML workloads: a deep residual network with CIFAR-10 dataset [47] and matrix factorization with MovieLens dataset [49]. For each workload, we configured naïve waiting with different delays and compared their impacts to the per-

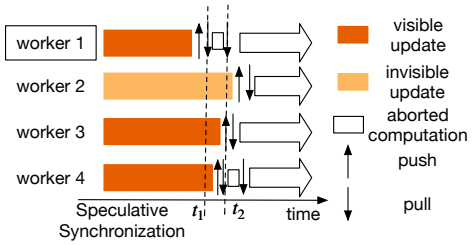


Figure 2.6: Illustration of speculative synchronization. Worker-1 speculatively aborts computation after observing the two pushes made by two peers. It pulls parameters again and starts over.

formance. Fig. 2.5 depicts the learning curves of the two ML workloads. By delaying each pull request by 1 second, both workloads achieve significant performance improvements. However, as the delay increases, more computing cycles get wasted, and the performance deteriorates. As illustrated by the CIFAR-10 workload in fig. 2.5a, delaying each pull request by 3 seconds yields little benefit over the original implementation; further increasing the delay to 5 seconds even does more harm than good.

To summarize, our experiments confirm that simply deferring each pull request to expose more fresh parameters can be beneficial. However, such a benefit is conditioned on finding the "right" delay time, which by itself is technically non-trivial. In fact, naively deferring each pull request may not always be justified—more often than not, the delay may expose only a few new updates. We therefore give up on naïve waiting and turn to a new approach to avoid unjustified delay.

2.3 Speculative Synchronization

In this section, we present a new scheme, called *speculative synchronization* (SpecSync), which allows workers to speculatively abort the ongoing computation and start over with fresher parameters. We model the gain and loss due to speculative re-execution, based on which we propose an effective heuristic algorithm to determine when to restart computation for workers. SpecSync can be implemented in both ASP and SSP models, *complementing* existing solutions with improved performance.

2.3.1 Overview

Key idea. Instead of imposing an arbitrary delay without justification, we let the worker asynchronously proceed to the next iteration immediately, while at the same time speculating about the updates made by others. Once the worker learns that the global parameters have been updated "enough" times, it will abort the ongoing iteration, pull the fresher parameters to start over—if that is not too late yet.

Continuing the example in fig. 2.2, we apply speculative synchronization and illustrate workers' behaviors in fig. 2.6. We start to focus on worker-1. After finishing the first iteration, it pulls parameters and starts the next iteration immediately. Shortly after it starts, it learns that two other peers have pushed updates to servers (highlighted in fig. 2.6), which it views as a significant-enough change made to the global parameters. Worker-1 hence aborts the ongoing iteration, re-synchronizes with servers to include those two recent updates, and starts over with much fresher parameters. The abort-and-restart decision is also made by worker-4 upon its notice of two updates pushed shortly after the second iteration starts. In contrast, workers 2 and 3 choose not to restart as they do not see enough updates (two in this example) pushed to servers since their last pulls.

Benefits. SpecSync offers two benefits.

1) It avoids unjustified delays, minimizing the cost of wasted computing cycles due to abortion. With SpecSync, an iteration gets restarted only when the global parameters have undergone *significant enough* updates within a short period of time after the iteration begins. When that happens, the improved quality of refinement brought by fresher parameters will surely outweigh the cost of slightly delayed computations.

2) SpecSync can be flexibly implemented in both ASP and SSP models, complementing *them with improved performance.* In fact, the only difference between ASP and SSP is that the latter enforces bounded staleness in that fast workers must wait for slowed ones to catch up. With SpecSync implemented in the SSP model, workers can *actively* seek opportunities to restart computation with fresher parameters, before the staleness bound is reached. This also gives slowed workers a chance to timely capture updates from fast peers by aborting computations, ensuring a more consistent view of global parameters. As a result, the quality of updates generated by straggling workers can be improved dramatically.

Challenges. However, to facilitate SpecSync, there are two major challenges we should address.

1) How can we efficiently notify each worker when the global parameters are updated by others, without incurring high communication overhead? Simply broadcasting each worker's push notification to others causes all-to-all communications, and is too expensive to implement. We shall address this challenge in §2.4 through a *centralized* system architecture, where a *scheduler* oversees pushes from all workers and notifies each when the global parameters have been updated sufficient times since its last pull.

2) As the global parameters are refined, how can we determine, for each worker, when to abort computation and start over? We employ a simple speculation strategy with two hyperparameters: `ABORT_TIME` and `ABORT_RATE`. In particular, after starting an iteration, a worker speculates about the parameter updates made in a time period of length `ABORT_TIME`. The worker counts the number of pushes made by others in that speculation period, and normalizes the count by the total number of workers to obtain the *push rate*. If the push rate exceeds `ABORT_RATE`, the worker is convinced that the global parameters have undergone significant updates since its last pull. The worker then aborts the ongoing computation, re-synchronizes with servers, and starts over.

The optimal setting of `ABORT_TIME` and `ABORT_RATE` is specific to the workload and cluster configuration. The choices of the two hyperparameters critically determine the performance of speculative synchronization. The question is: given a learning workload, how can we judiciously choose the two hyperparameters that lead to the optimal performance? Of course we can adopt the conventional ML hyperparameter tuning techniques like grid search and Cherrypick [50], However, such approach is time and money consuming thanks to the profiling steps, making it hard for agile adoption. Consequently, we try to find an efficient way to tune the two hyperparameters on the go efficiently and effectively in the next section.

2.3.2 Adaptive Hyperparameter Tuning

To achieve the optimal performance of SpecSync, we propose a heuristic algorithm that adaptively tunes the two hyperparameters. We start with a problem formulation.

Problem formulation. Suppose that worker i restarts an iteration after a speculation period. Worker i gains by uncovering more recent updates that are otherwise invisible, at the expense of a delayed computation. Such a delay may hide worker i's update from being captured by others. By the time worker i pushes an update, it would be too late for some other workers to capture that refinement, which in turn harms the parameter freshness of those workers.

In our model, we measure the *freshness gain* by the number of updates others pushed in the speculation period, and the *freshness loss* by the number of *missed peers*, which are workers that missed the (delayed) push of the speculator. More precisely, we divide time into *epochs*. Let Δ denote the `ABORT_TIME` used in epoch τ. For worker i, let $u_{i,\tau}(\Delta)$ be the number of updates it uncovers during its speculation period. The value of $u_{i,\tau}(\Delta)$ measures the freshness gain. Let $l_{i,\tau}(\Delta)$ be the number of missed peers, which quantifies the freshness loss. Worker i hence makes a *freshness contribution* $u_{i,\tau}(\Delta) - l_{i,\tau}(\Delta)$. To make a positive contribution, worker i aborts the computation only when the gains outweighs the loss, i.e., $u_{i,\tau}(\Delta) \geqslant l_{i,\tau}(\Delta)$. We shall discuss later how this can be achieved by correctly choosing an `ABORT_RATE`.

Summing up the contribution over all workers, we obtain the *overall freshness improvement* due to speculative synchronization, i.e.,

$$F_\tau(\Delta) = \sum_{i=1}^{m} \left(u_{i,\tau}(\Delta) - l_{i,\tau}(\Delta)\right). \tag{2.3}$$

Our goal is to find the optimal Δ at the beginning of each epoch that maximizes the overall freshness improvement:

$$\text{maximize}_\Delta F_\tau(\Delta). \tag{2.4}$$

Estimating gain and loss. Unfortunately, Problem (2.4) cannot be directly solved in practice, as the exact computation of $u_{i,\tau}(\cdot)$ and $l_{i,\tau}(\cdot)$ requires knowing the complete push/pull sequence in epoch τ before the epoch starts. We therefore turn to estimations of the gain and loss.

In particular, to estimate how many updates worker i will uncover after a speculation period Δ, we simply refer back to the previous epoch and count the number of updates

the worker would have uncovered if it had deferred its last iteration by Δ. More precisely, we estimate $u_{i,\tau}(\Delta)$ as

$$\tilde{u}_{i,\tau}(\Delta) = u_{i,\tau-1}(\Delta), \tag{2.5}$$

where $\tilde{u}_{i,\tau}(\cdot)$ is an estimation of $u_{i,\tau}(\cdot)$. Our insight is that the algorithmic behaviors and machine performance are usually stable in a short period of time. Therefore, the freshness gain computed based on the push history in the previous epoch can be used as a good approximation to that in the current epoch.

Ideally, we can use the same approach to estimate freshness loss, i.e., $\tilde{l}_{i,\tau}(\Delta) = l_{i,\tau-1}(\Delta)$. However, simply using the push history in epoch $\tau - 1$ may not be sufficient to compute $l_{i,\tau-1}(\Delta)$. In fact, assuming the worker deferred its last iteration by Δ, it is possible that the iteration would still be running now, and the freshness loss caused by that delay cannot be fully characterized until the iteration completes.

Unable to make use of historical traces, we simply estimate freshness loss $l_{i,\tau}(\Delta)$ as *the expected number of missed peers assuming uniform arrivals of pull requests*. While this technical assumption may not always hold in practice, it simplifies the analysis and makes the problem tractable. We shall show in §2.5 that despite this technical assumption, our solution can still achieve near-optimal performance in a number of ML applications.

Specifically, let $T_{i,\tau}$ be the iteration span of worker i in epoch τ, which can be accurately predicted from history. Deferring the iteration by Δ hides the worker's update from being captured by another with probability $\frac{\Delta}{T_{i,\tau}}$. Therefore, the expected number of missed peers, or the estimate of freshness loss, is

$$\tilde{l}_{i,\tau}(\Delta) = \frac{\Delta}{T_{i,\tau}}(m-1). \tag{2.6}$$

Substituting the gain and loss by their estimates, we obtain an estimate of the overall freshness improvement:

$$\tilde{F}_{\tau}(\Delta) = \sum_{i=1}^{m} \left(\tilde{u}_{i,\tau}(\Delta) - \frac{m-1}{T_{i,\tau}}\Delta \right). \tag{2.7}$$

Hyperparameter tuning. Directly searching the optimal Δ to maximize improvement estimate $\tilde{F}_{\tau}(\Delta)$ can be difficult, as Eq. (2.7) is non-convex. We present an efficient algorithm that optimally solves this problem. Our key observation is that, to maximize Eq. (2.7), it is

Algorithm 1 Adaptive Tuning

```
  – m: number of workers
  – T: averaged iteration span
1: function TUNEPARAM
2:     l ← sequence of all pushes made in the last epoch
3:     {Δ} ← time difference between all pairs of pushes in l
4:     for Δ ∈ {Δ} do
5:         F ← computation result of Eq. (2.7)
6:         if F is maximal then
7:             ABORT_TIME ← Δ
8:             ABORT_RATE ← Δ(m − 1)/Tm
```

sufficient to search *a finite number* of Δ. To see this, consider worker i, and let Δ gradually increase from 0. By definition, $\tilde{u}_{i,\tau}(\Delta)$ increments only when the increase of speculation period exposes one more push from another worker. Therefore, $\tilde{u}_{i,\tau}(\Delta)$ is a *step function*. Also note that the estimate of freshness loss (cf. Eq. (2.6)) linearly increases with Δ. Putting them together, the freshness contribution (gain minus loss) reaches the maximum *only if* the speculation interval *right aligns* with a push. It is easy to see that there are $O(m^2)$ such intervals in total, where m is the number of workers. We can therefore enumerate all candidate Δ and choose the one that maximizes Eq. (2.7). Algorithm 1 illustrates the details, with time complexity $O(m^3)$.

Once the optimal Δ^* is found (`ABORT_TIME`), we set `ABORT_RATE` so that workers abort computation only when the gain outweighs the loss. Specifically, let Γ denote the `ABORT_RATE` used by worker i. By our scheme, worker i aborts computation if the number of updates received during the speculation period exceeds Γm. We therefore set $\Gamma = \tilde{l}_{i,\tau}(\Delta^*)/m$ to ensure that re-execution always makes a positive contribution to the overall freshness improvement.

The experimental evaluation in §2.5 verifies that our heuristic approach achieves *near-optimal* speedup for many ML applications even compared with the scheme using the optimal hyperparameters cherry-picked via exhaustive search.

2.4 Implementation

In this section, we present our implementation of SpecSync atop MXNet [7, 48], a popular ML framework employing the parameter server (PS) architecture. While we base

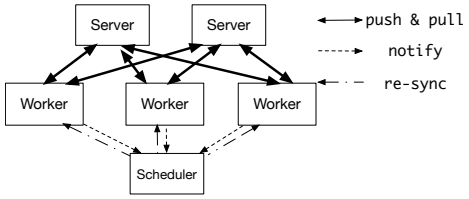


Figure 2.7: The architecture of speculative synchronization.

our implementation on MXNet, nothing precludes it from being ported to other PS-based systems such as TensorFlow [9].

2.4.1 Architecture Overview

Centralized implementation. We employ a *centralized implementation* for SpecSync. As shown in fig. 2.7, our implementation consists of three components: workers, servers, and a centralized scheduler. At the beginning of an iteration, a worker pulls model parameters from servers and starts computation. In the meantime, the worker *delegates* the speculation job to the centralized scheduler which oversees the parameter updates from others and notifies the worker when it is time to re-synchronize. Upon receiving an instruction from the scheduler, the worker pulls fresh parameters from servers and restarts the computation. After an iteration completes, the worker pushes the computed update to servers, and notifies the scheduler. This way, the scheduler can keep track of pushes made by all workers, enabling it a global view to perform speculation for each worker.

Benefits. The centralized implementation offers two benefits over a direct implementation where each worker performs speculation *individually*. First, it eliminates the need for all-to-all communication among workers, which is unavoidable in the direct implementation as each worker must *broadcast* a push notification to all others. In contrast, workers in the centralized architecture only report to the scheduler. We shall show in §2.5 that the communication overhead incurred among workers and the scheduler is negligible. Second, in the centralized architecture, only the scheduler needs to maintain a global view of the push history of workers. However, in the direct implementation, this information must be separately maintained by each worker, leading to unnecessary storage redundancy.

Algorithm 2 Speculative Synchronization

Workers: $i = 1, 2, \ldots, m$

function WORKERSTARTSEPOCH(e)
  Pull model parameters $w^{(e,0)}$ from servers
  for all training batch $t = 0, 1, 2, \cdots, T$ **do**
    Start computing gradient $g_i^{(e,t)}$ in a non-blocking manner
    if worker receives `re-sync` during computation **then**
      Abort computation and pull $w^{(e,t)}$ from servers
      go to 4 ▷ Restart compution
    Push $g_i^{(e,t)}$ to servers
    Pull $w^{(e,t+1)}$ from servers
    Send `notify` to scheduler

Scheduler:

– l: a list of timestamps of all pushes

function STARTSTRAINING
  for epoch $e = 0, 1, 2, ..., E$ **do**
    Issue WORKERSTARTSEPOCH(e) to all workers
function HANDLENOTIFICATION(`msg`)
  Append current timestamp to l
  Call CHECKRESYNC(`msg.sender`) after `ABORT_TIME`
function CHECKRESYNC(`senderId`)
  `cnt` $\leftarrow$ number of pushes received within `ABORT_TIME`
  if `cnt` $\geqslant m \times$ `ABORT_RATE` **then** ▷ Time to re-synchronize
    Issue `re-sync` to the worker with `senderId`

2.4.2 Workflow

We elaborate on the implementation of each component, following the workflow of SpecSync illustrated in algorithm 2.

Workers communicate with the scheduler through two dedicated messages: `notify` and `re-sync`. A `notify` message simply contains a `senderId` and is sent to the scheduler by a worker upon the completion of an iteration. The `notify` message triggers the speculation on the scheduler. At the same time, the worker pulls parameters from servers and proceeds to the next iteration, during which the worker expects a `re-sync` message from the scheduler to re-synchronize with servers and start over. Once the iteration completes, the worker pushes update to servers, sends a `notify` message to the scheduler, and repeats the entire process.

Servers are *agnostic* to speculative synchronization performed by workers and the scheduler. Their behaviors remain the same as in the stock MXNet, and is not shown in algorithm 2.

Table 2.1: Models and datasets used for workloads. The iteration time is based on `m4.xlarge` instance.

Workload	# parameters	Dataset	Dataset size	Iteration time
MF	4.2 million	Movielens	100,000	3s
CIFAR-10	2.5 million	CIFAR-10	50,000	14s
ImageNet	5.9 million	ImageNet	281,167	70s

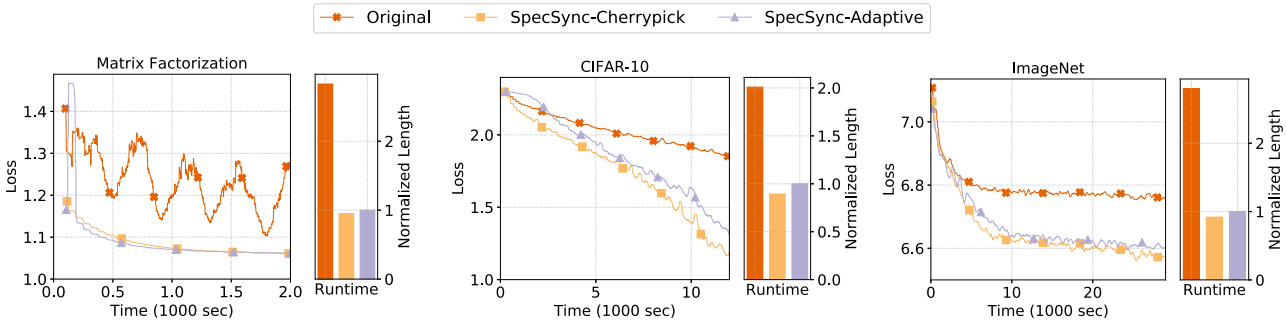


Figure 2.8: Loss (left plot in each sub-figure) and runtime (right plot in each sub-figure) comparison of (a) MF, (b) CIFAR-10, and (c) ImageNet. For better visualization, we only show the results up to when one of the approaches is converged (i.e., loss is below the target value for 5 consecutive iterations).

Scheduler keeps track of pushes made by workers through `notify` messages and performs two jobs. First, it computes the two hyperparameters `ABORT_TIME` and `ABORT_RATE` at the beginning of each epoch using algorithm 1. Second, it implements the logic of SpecSync *on behalf of each worker*. Specifically, it maintains a *push counter* and a *timer* for each worker. Upon receiving a `notify` message from a worker, the scheduler appends it to the push history and kicks start the speculation for the sender. To do so, it resets the push counter of the sender and starts the corresponding timer which will expire in `ABORT_TIME`. During the speculation period, the push counter increments whenever a `notify` message is received. Upon the timer expires, the scheduler checks whether the counter is more than $m \times$`ABORT_RATE`, where m is the number of workers. If so, the scheduler instructs the worker to re-synchronize through a `re-sync` message, as enough updates have been pushed since the worker's last pull.

2.5 Evaluation

We conduct extensive experimental evaluations to validate the effectiveness and robustness of the proposed SpecSync. We first compare the accuracy and runtime of SpecSync with the default synchronization scheme of MXNet. Then we demonstrate the robustness of SpecSync in terms of handling heterogeneity and scalability. We examine the communication overhead afterward. Finally, we discuss the advantage of hyperparameter tuning with respect to the algorithm efficiency and adaptivity.

2.5.1 Experiment Setup

Workloads. We use three different workloads to drive the experiments for evaluation, and we summarize them in table 2.1. For the first workload, we use MovieLens[49] as the dataset to train a recommendation system with matrix factorization (MF), the batch size is set to 100,000. We train a 110-layer deep residual network [51] with CIFAR-10 dataset as the second workload. To achieve the best performance, we set the batch size to 128 and let the learning rate decrease from an initial value 0.05 at epochs 200 and 250, respectively [52]. Finally, we train an 18-layer deep residual network with ImageNet dataset [53]. The batch size is set to 128 and the learning rate is set to 0.3. We choose the above workloads because they represent different characteristics of popular ML applications. For example, the input data of CIFAR-10 and ImageNet are pictures represented by dense vectors, while the input data of MF are user ratings represented by sparse vectors. The training data set size and model size of the above workloads are also representative from small to large applications with iteration time spanning from a few seconds to more than one minute.

Testbed. We build clusters on Amazon EC2 for running experiments. We use a 40-node cluster (**Cluster 1**) for effectiveness evaluation, which is composed of 40 `m4.xlarge` instances. This cluster is for the scenario of asynchronous distributed learning on homogeneous hardware. We build a heterogeneous cluster (**Cluster 2**) for evaluating SpecSync's ability to deal with heterogeneity, which consists of 10 `m3.xlarge`, 10 `m3.2xlarge`, 10 `m4.xlarge`, and 10 `m4.2xlarge` instances. To evaluate SpecSync's scalability, we use 3 clusters with 20, 30 and 40 `m4.xlarge` instances respectively. All the instances run

Ubuntu Server 16.04 LTS, and are configured to run our extended MXNet 0.7.

Schemes. We use three different synchronization schemes in our evaluation. (1) **Original**: default asynchronous synchronization scheme provided by MXNet. (2) **SpecSync-Cherrypick**: the cheerypick version (i.e., tune the `ABORT_TIME` and `ABORT_RATE` hyperparameters using grid search) of the proposed speculative synchronization, which is built on top of the asynchronous synchronization scheme provided by MXNet. (3) **SpecSync-Adaptive**: the adaptive version (i.e., tune the hyperparameters adaptively) of the proposed speculative synchronization, which is also implemented based on the asynchronous synchronization scheme provided by MXNet.

2.5.2 Effectiveness of SpecSync

We first evaluate the effectiveness of the proposed SpecSync on Cluster 1 with three different workloads outlined in table 2.1. We report two metrics: loss change over time (for accuracy evaluation) and runtime to convergence (for runtime performance evaluation), and present the results in fig. 2.8. Runtime is measured as the timespan from the beginning of training to convergence, where convergence is defined as the loss staying below the target value for 5 consecutive iterations. By comparing the results of different schemes, it is clear that the proposed SpecSync can significantly speed up the training process, i.e., up to 2.97× speedup for MF, up to 2.25× speedup for CIFAR-10, and up to 3× speedup for ImageNet. The results also demonstrate that even though SpecSync-Adaptive is not able to achieve the same performance as SpecSync-Cherrypick, the difference is very small, which verified the effectiveness of the proposed adaptive algorithm. We also notice for SpecSync-Adaptive, the learning curves (loss as a function of time) has more jitter at the beginning than SpecSync-Cherrypick, this is due to the adaptive nature of SpecSync-Adaptive.

With SpecSync, when re-synchronization occurs during an iteration, the length of the iteration becomes longer, but the training quality of the iteration is improved due to the fresher parameters used for training, so the overall training speed becomes faster. To demonstrate this, we plot the loss as a function of the iteration number and the accumulated iteration number for different schemes in fig. 2.9. The comparison results shows that

it takes up to 58% fewer iterations for SpecSync to converge compared with Original. The effectiveness of SpecSync varies across models, as models have different sesitivity levels towards staleness. Such an improvement suggests that with SpecSync, the quality of computing is improved while the efficiency of utilizing the distributed computing power by asynchronous learning is preserved.

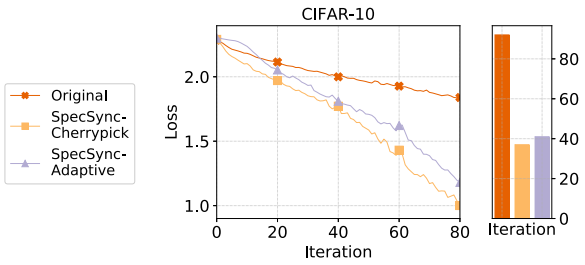


Figure 2.9: Loss as a function of iteration number (left plot) and accumulated iteration number (right plot) for different synchronization schemes using CIFAR-10.

2.5.3 Robustness of SpecSync

Heterogeneity. In distributed machine learning, heterogeneity (caused by various factors from hardware resources to software failures) can result in inconsistent training progress among workers and therefore slows down the training speed [26]. Here we conduct experiments using a heterogeneous cluster (Cluster 2), which consists of 4 different instance types, to evaluate SpecSync's robustness in the presence of heterogeneity. In the interest of space, we only show the results of CIFAR-10 in fig. 2.10. From the loss plot, it is clear that SpecSync-Adaptive outperforms Original in both homogeneous and heterogeneous clusters.[2] The results also verified that the heterogeneity could slow down the training speed. Heterogeneity slows down BSP because it increases synchronization overhead as fast workers need to wait for stragglers to complete in each iteration. ASP and SSP also suffer from heterogeneity because the staleness gap between workers is intensified due to the mismatch in training speed, which makes the convergence slower. SpecSync-Adaptive actively improves the freshness of parameter replica to alleviate inconsistency between workers compared to ASP and SSP, and does not have the synchronization issue of BSP,

[2]Given we have already verified the difference between SpecSync-Adaptive and SpecSync-Cherrypick is small, for clear presentation, we do not show the results of SpecSync-Cherrypick in all following plots.

thus being more robust to heterogeneity. Another interesting observation is the speedup of SpecSync-Adaptive over Original is smaller compared to the same experiments on homogeneous cluster as shown in fig. 2.8b. This is because our adaptive tuning approach assumes uniform arrivals of pull requests, but in heterogeneous environment, the arrival becomes less uniform, so the quality of the tuned hyperparameters deteriorates.

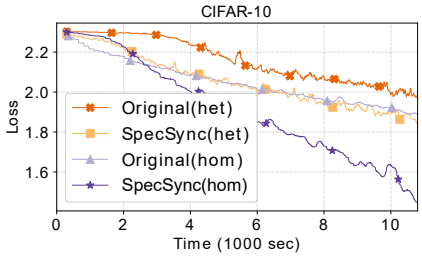


Figure 2.10: Loss comparison for different synchronization schemes running CIFAR-10 in both homogeneous and heterogeneous clusters.

Scalability. We perform sensitivity analysis for cluster size to evaluate the scalability of the proposed SpecSync-Adaptive. We show results for two scenarios that are commonly used in practice. The first scenario is when machine learning practitioners with clear training target in mind. So, we demonstrate the speedup of SpecSync-Adaptive over Original in runtime for achieving the same target training accuracy of CIFAR-10 using cluster size of 20, 30, and 40, respectively (the left plot of fig. 2.11). The second scenario is usually for the case with fixed budget and aims to achieve the best training accuracy under the given budget (e.g., rent cloud instances for a time period that can be fit into the given budget). Therefore, we compare the loss improvement of SpecSync-Adaptive over Original when training CIFAR-10 for the same amount of time using different cluster sizes as shown in the right plot of fig. 2.11. It is clear that in both scenarios, SpecSync-Adaptive consistently outperforms Original running with different cluster size. More importantly, when the cluster size grows, the improvement becomes even larger. This suggests that SpecSync-Adaptive has better scalability than Original.

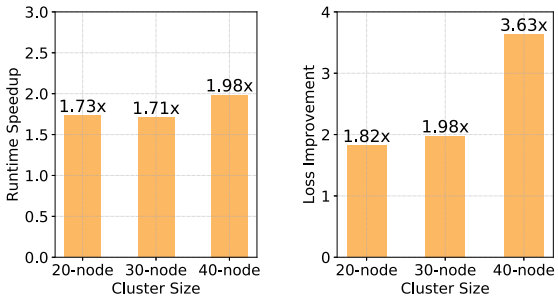


Figure 2.11: Runtime speedup for achieving the same training accuracy target (left plot) and loss improvement with the same training time (right plot) for SpecSync-Adaptive over Original using CIFAR-10 as workload under different cluster size.

2.5.4 Communication Overhead

SpecSync's centralized design utilizes information available at servers to make the resynchronization decisions. Since SpecSync and its adaptive hyperparameter tuning do not involve heavy computation, the main source of overhead is added communication due to exchanging additional information between workers and parameter servers. Fig. 2.12 reports the accumulated data transfer over time for different workloads using Original and SpecSync-Adaptive. It is clear that the accumulated data transfer is very close between SpecSync-Adaptive and Original at all times, meaning there is very little additional bandwidth consumed by SpecSync-Adaptive. In addition, the overall training time is shorter using SpecSync-Adaptive, so the total data transfer can be actually smaller compared to Original, i.e., compare the right most points in fig. 2.12. Take CIFAR-10 as an example, Original incurs a total data transfer of 3.17 TB while SpecSync-Adaptive only needs to transfer 2.00 TB in total, a saving of nearly 40% of communication overhead.

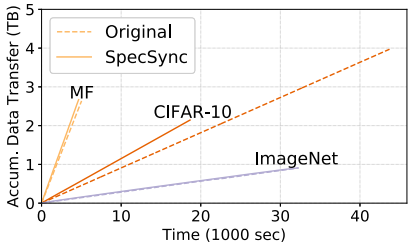


Figure 2.12: Accumulated data transfer over time for different workloads under different schemes.

We also refer to fig. 2.13 for the breakdown of overall data transfer incurred by SpecSync-Adaptive in three parts: push and pull, which are the regular communication traffic (i.e., also in Original); the additional pulls triggered by re-synchronization; notifications including `notify` and `re-sync` messages. The results indicate that the additional overhead introduced by notification messages is marginal given the message size is quite small. The main additional communication overhead is introduced by re-synchronization. However, such overhead is compensated by the improved quality of training, thanks to the fresher parameters. In general, for computation-bound workloads, SpecSync can achieve higher computation efficiency as the training quality improves with fresher parameters, and therefore alleviates the computation bottleneck impact. For communication-bound workloads, SpecSync automatically adjusts the speculative synchronization hyperparameters (e.g., `ABORT_TIME` and `ABORT_RATE`) so that the re-synchronization is performed more conservatively to balance the freshness and network performance.

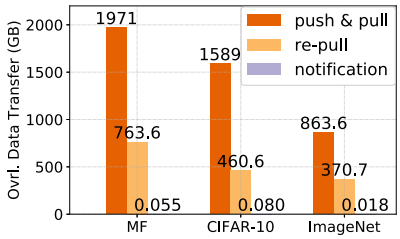


Figure 2.13: Overall data transfer breakdown for SpecSync-Adaptive.

2.5.5 SpecSync-Cherrypick vs. SpecSync-Adaptive

We evaluate SpecSync-Adaptive here against SpecSync-Cherrypick in terms of the overhead of tuning hyperparameters. For SpecSync-Adaptive, there is little overhead as it simply searches best hyperparameter through logged information (a short list of numbers) using a closed-form estimation function (Eq. (2.7)), no additional profiling experiment is needed. For SpecSync-Cherrypick, the hyperparameters are tuned through exhaustive search with profiling experiments. Examples of total search time for different workloads are shown in table 2.2. In the example, we search 10 different values of `ABORT_RATE` for each workload. For `ABORT_TIME`, we try to minimize the number of trials by restricting the search range and increasing search step (e.g., we use half of the batch time as

Table 2.2: Cost of hyperparameter exhaustive search using SpecSync-Cherrypick.

Workload	**# of trial for** ABORT_TIME	**# of trial for** ABORT_RATE	**Each trial time (hour)**	**Total search time (hour)**
MF	3	10	1.33	40
CIFAR-10	7	10	6	420
ImageNet	10	10	> 8	> 800

upper bound and make sure the search step is greater than communication time). It is clear that even with reasonable search range and step, the cost is still very high as each experiment takes long time, and the accumulated cost becomes even higher. Therefore, SpecSync-Adaptive has much lower search overhead for hyperparameter tuning. In addition, SpecSync-Adaptive adapts the hyperparameters for each iteration, which is more robust than the fixed value solution of SpecSync-Cherrypick.

2.5.6 Discussion

The experimental study in this section is mainly based on CPU cluster. For GPU or other architecture-based cluster, SpecSync is also compatible because the synchronization happens at node level, where the node can be a virtual concept, e.g., it can be a machine with CPU or GPU, and/or it can be a CPU or GPU within a multi-CPU/GPU machine. In the interest of space, we leave the detailed study for clusters with different hardware architecture as our future work. We also leave experimental study of other machine learning applications and other machine learning frameworks as our future work.

2.6 Related Work

Many recent works have been proposed to improve the performance of existing synchronization schemes employed in distributed ML systems. Notably, for the SSP-style execution, Wei et al. [54] proposed a system called Bösen to maximize the communication efficiency by prioritizing update messages that are the most significant to the final convergence. SSP-style solutions passively curb the influnce of stale parameters by limiting the inconsistency level, while SpecSync actively refreshes stale parameters based on speculation instead. SSP and SpecSync try to address the inconsistency issue from two perspectives, thus are orthogonal to each other. Harlap et al. [55] proposed FlexRR to mitigate the

straggler problem by dynamically offloading the training work of straggling workers to fast machines. For the ASP-style executions, Zhang et al. [56] proposed a staleness-aware asynchronous SGD algorithm that achieves guaranteed convergence by dynamically adjusting the learning rate based on the gradient staleness. Jiang et al. [26] applied the similar approach to heterogeneous clusters, where slowed workers are assigned lower learning rate to alleviate the negative impact of their updates. More recently, Chen et al. [57] revisited the BSP model and suggested synchronous training of deep models with *backup* workers, so as to mitigate the impact of stragglers. All those techniques are orthogonal to our proposal and can be combined with SpecSync.

SpecSync is inspired by *delay scheduling* [58] used in MapReduce clusters, where map tasks prefer to running on machines that can provide data locality. With delay scheduling, the assignment of a task is delayed for a short period of time if its data locality cannot be satisfied at the current moment. Unlike delay scheduling in MapReduce where the delay is almost surely beneficial [58], we have shown in §2.2 that naïvely delaying each pull request is not always justified (fig. 2.5). SpecSync is proposed to address this problem. In fact, aborting and restarting a task is considered expensive for MapReduce jobs, and is usually not an option to the scheduler.

2.7 Summary

In this chapter, we have investigated a new aspect to improve parameter freshness for asynchronous distributed learning. We have proposed SpecSync where each worker speculates about the parameter updates pushed by others after an iteration starts. In case that enough number of pushes have been made within a short period of time, the worker aborts the ongoing iteration, re-synchronizes with servers, and starts over with fresher parameters. We have designed an adaptive hyperparameter tuning algorithm to judiciously determine the span of speculation period and the threshold number of pushes beyond which a re-synchronization is triggered. We have implemented SpecSync atop MXNet in a centralized architecture with little extra communication overhead. Experimental evaluations through EC2 deployment demonstrate that SpecSync achieves up to 3× speedup in three benchmarking ML workloads.

CHAPTER 3

EFFICIENT HOMOMORPHIC ENCRYPTION FOR CROSS-SILO FEDERATED LEARNING

Besides efficiency, privacy is another hurdle that challenges large scale ML training. In this chapter, we focus on how to efficiently protect the data privacy for data owners in cross-silo federated learning (FL).

3.1 Background and Related Work

In this section, we highlight the stringent privacy requirements posed by cross-silo federated learning. We survey existing techniques for meeting these requirements.

3.1.1 Cross-Silo Federated Learning

According to a recent survey [14], federated learning (FL) is a scenario where multiple clients collaboratively train an ML model with the help of a central server; each client transfers local updates to the server for immediate aggregation, without having its raw data leaving the local storage. Depending on the application scenarios, federated learning can be broadly categorized into *cross-device* FL and *cross-silo* FL. In the cross-device setting, the clients are a large number of mobile or IoT devices with limited computing power and unreliable communications [59, 14, 60]. In contrast, the clients in the cross-silo setting are a small number of organizations (e.g., financial and medical) with reliable communications and abundant computing resources in datacenters [13, 14]. We focus on cross-silo FL in this chapter.

Compared with the cross-device setting, cross-silo FL has significantly more stringent requirements on privacy and learning performance [13, 14]. *First*, the final trained model should be *exclusively released* to those participating organizations—no external party, including the central server, can have access to the trained model. *Second*, the strong privacy

guarantee should not be achieved at a cost of learning accuracy. *Third*, as an emerging paradigm, cross-silo FL is undergoing fast innovations in both algorithms and systems. A desirable privacy solution should impose *minimum constraints* on the underlying system architecture, training mode (e.g., synchronous and asynchronous), and learning algorithms.

3.1.2 Privacy Solutions in Federated Learning

Many strategies have been proposed to protect the privacy of clients for federated learning. We briefly examine these solutions and comment on their suitability to cross-silo FL.

Secure Multi-Party Computation (MPC) allows multiple parties to collaboratively compute an agreed-upon function with private data in a way that each party knows nothing except its input and output (i.e., zero-knowledge guarantee). MPC utilizes carefully designed computation and synchronization protocols between clients. Such protocols have strong privacy guarantees, but are difficult to implement efficiently in a geo-distributed scenario like cross-silo FL [13]. Developers have to carefully engineer the ML algorithms and divide the computation among parties to fit the MPC paradigm, which may lower the privacy guarantees for better performance [61, 62, 63].

Differential Privacy (DP) is another common tool that can be combined with model averaging and SGD to facilitate secure FL [15, 16]. It ensures the privacy of each individual sample in the dataset by injecting noises. A recent work proposes to employ *selective parameter update* [16] atop differential privacy to navigate the tradeoff between data privacy and learning accuracy. Although DP can be efficiently implemented, it exposes plain gradients to the central server during aggregation. Later study shows that one can easily recover the information from gradients [18]. While such privacy breach and the potential accuracy drop might be tolerable for mobile users in cross-device FL, they raise significant concerns for participating organizations in cross-silo FL.

Secure Aggregation [17] is proposed recently to ensure that the server learns no individual updates from any clients but the *aggregated updates* only. While secure aggregation has been successfully deployed in cross-device FL, it falls short in cross-silo FL for two reasons. First, it allows the central server to see the aggregated gradients, based on which

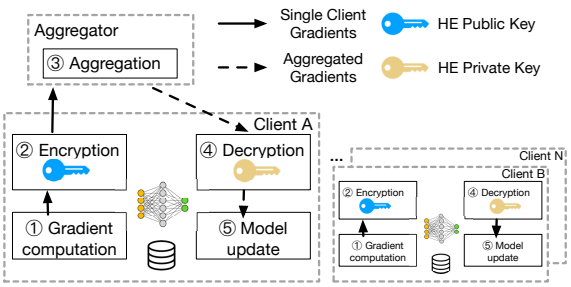


Figure 3.1: The architecture of cross-silo FL system, where HE is implemented as a pluggable module on the clients.

the information about the trained model can be learned by an external entity (e.g., public cloud running the central server). Second, in each iteration, clients must synchronize secret keys and *zero-sum masks*, imposing a strong requirement of *synchronous training*.

Homomorphic Encryption (HE) allows certain computation (e.g., addition) to be performed directly on ciphertexts, without decrypting them first. Many recent works [18, 19, 64, 65] advocate the use of additively HE schemes, notably Paillier [66], as the primary means of privacy guarantee in cross-silo FL: each client transfers the encrypted local updates to the server for direct aggregation; the result is then sent back to each client for local decryption. HE meets the three requirements of cross-silo FL. *First*, it protects the trained model from being learned by any external parties including the server as update aggregation is performed on ciphertexts. *Second*, it incurs no learning accuracy loss, as no noise is added to the model updates during the encryption/decryption process. *Third*, HE directly applies to the existing learning systems, requiring no modifications other than encrypting/decrypting updates. It hence imposes no constraints to the synchronization schemes and the learning algorithms. However, as we shall show in §3.2, HE introduces significant overhead to computation and communication.

To summarize, each of these privacy-preserving techniques has its pros and cons. MPC is able to provide strong privacy guarantees, but requires expert efforts to re-engineer existing ML algorithms. DP can be adopted easily and efficiently, but has the downside of weaker privacy guarantee and potential accuracy loss. Secure aggregation is an effective way to facilitate large-scale cross-device FL, but may not be suitable for cross-silo FL as it exposes the aggregated results to third parties and incurs high synchronization cost. HE can be easily adopted to provide strong privacy guarantees without algorithm modifica-

tions or accuracy loss. However, the high computation and communication overheads make it impractical for production deployment at the moment.

3.1.3 Cross-Silo FL Platform with HE

Fig. 3.1 depicts a typical cross-silo FL system [19, 13, 14], where HE is implemented as a pluggable module on the clients. The *aggregator* is the server which coordinates the *clients* and aggregates their encrypted gradients. Note that in this work, we assume the aggregator is *honest-but-curious*, a common threat model used in the existing FL literature [17, 64, 16]. The communications between all parties (the clients and the aggregator) are secured by cryptographic protocols such as SSL/TLS, so that no third party can learn the messages being transferred. Before the training starts, the aggregator randomly selects a client as the leader who generates an HE key-pair and synchronizes it to all the other clients. The leader also initializes the ML model and sends the model weights to all the other clients. Upon receiving the HE key-pair and the initial weights, the clients start training. In an iteration, each client computes the local gradient updates (①), encrypts them with the public key (②), and transfers the results to the aggregator. The aggregator waits until the updates from all the clients are received. It then adds them up and dispatches the results to all clients (③). A client then decrypts the aggregated gradients (④) and uses it to update the local model (⑤).

This architecture design follows the classic distributed SGD pattern. So, the existing theories and optimizations including flexible synchronization [24, 67, 68] and local update SGD [69, 70, 71] naturally apply. Moreover, as model updating is performed on the client's side using the plaintext gradient aggregation, we can adopt state-of-the-art adaptive optimizers such as Adam [72] for faster convergence—a huge advantage over the existing proposal [18] that applies encrypted gradients directly on the encrypted global model in the server.

3.2 Characterizing Performance Bottlenecks

In this section, we characterize the performance of cross-silo FL with three real applications driven by deep learning models in a geo-distributed setting. We show that encryp-

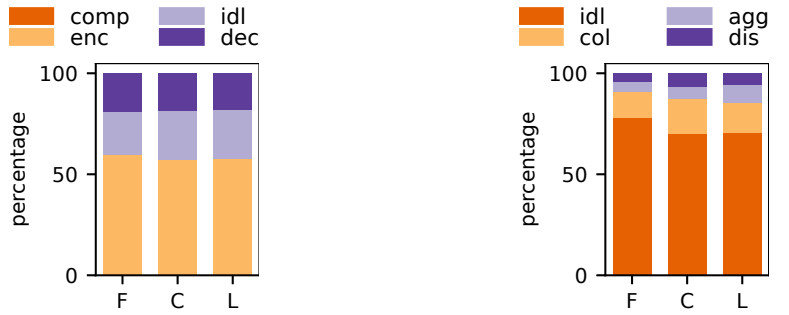
tion and communication come as two prohibitive bottlenecks that impede the adoption of FL among organizations. We survey possible solutions in the literature and discuss their inefficiency. To our knowledge, we are the first to present a comprehensive characterization for cross-silo FL in a realistic setting.

3.2.1 Characterization Results

Cross-silo FL is usually performed in multiple *geo-distributed* datacenters of participating organizations [14, 13]. Our characterization is carried out in a similar scenario where nine EC2 clients in five geo-distributed datacenters collaboratively training three ML models of various sizes, including FMNIST, CIFAR, and LSTM (table 3.3). Unless otherwise specified, we configure *synchronous training*, where no client can proceed to the next iteration until the (encrypted) updates from all clients have been aggregated. We defer the detailed description of the cluster setup and the ML models to §3.5.1.

We base our study in FATE (Federated AI Technology Enabler) [73], a secure compute framework developed by WeBank [28] to drive its FL applications with the other industry partners. To our knowledge, FATE is the only open-source cross-silo FL framework deployed in production environments. FATE has a built-in support to the Pailler cryptosystem [66] (key size set to 2048 bits by default), arguably the most popular additively HE scheme [29]. Our results also apply to the other partially HE cryptosystems.

Encryption and Communication Overhead We start our characterization by comparing two FL scenarios, with and without HE. We find that the use of HE results in *exceedingly long training time* with *dramatically increased data transfer*. More specifically, when HE is enabled, we measured the average training iteration time 211.9s, 2725.7s, and 8777.7s for FMNIST, CIFAR, and LSTM, respectively. Compared with directly transferring the plaintext updates, the iteration time is extended by 96×, 135×, and 154×, respectively. In the meantime, when HE is (not) in use, we measured 1.1GB (6.98MB), 13.1GB (85.89MB), and 44.1GB (275.93MB) data transfer between clients and aggregator in one iteration on average for FMNIST, CIFAR, and LSTM, respectively. To sum up, the use of HE increases both the training time and the network footprint by two orders of magnitude. Such performance overhead becomes even more significant for complex models with a large number



(a) Client: **comp**ute, **en**crypt, **idl**e, **dec**rypt

(b) Aggregator: **idl**e, **col**lect, **agg**regate, **dis**patch

Figure 3.2: Iteration time breakdowns of **F**MNIST, **C**IFAR, and **L**STM for a client and the aggregator.

of weights (e.g., LSTM).

Deep Dive. To understand the sources of the significant overhead caused by HE, we examine the training process of the three models in detail, where we sample an iteration and depict in fig. 3.2 the breakdown of the iteration time spent on different operations on the client's side (left) and on the aggregator's side (right), respectively.

As illustrated in fig. 3.2a, on the client's side, HE-related operations dominate the training time in all three applications. In particular, a client spent around 60% of the iteration time on gradient encryption (yellow), 20% on decryption (dark purple), and another 20% on data transfer and idle waiting for the gradient aggregation to be returned[1] (light purple). In comparison, the time spent on the actual work for computing the gradients becomes *negligible* ($< 0.5\%$).

When it comes to the aggregator (fig. 3.2b), most of the time ($> 70\%$) is wasted on idle waiting for a client to send in the encrypted gradients (orange). Collecting the gradients from all clients (yellow) and dispatching the aggregated results to each party (dark purple) also take a significant amount of time, as clients are geo-distributed and may not start transferring (or receiving) at the same time. The actual computation time for gradient aggregation (light purple) only accounts for less than 10% of the iteration span. Our deep-dive profiling identifies encryption and decryption as the two dominant sources of the

[1]Due to the synchronization barrier, a client needs to wait for all the other clients to finish transferring updates to the aggregator.

Table 3.1: Benchmarking Paillier HE with various key sizes.

Key size	Plaintext	Ciphertext	Encryption	Decryption
1024	6.87MB	287.64MB	216.87s	68.63s
2048	6.87MB	527.17MB	1152.98s	357.17s
3072	6.87MB	754.62MB	3111.14s	993.80s

exceedingly long training time.

Why is HE So Expensive? In additively HE cryptosystems such as Paillier [66], encryption and decryption both involve multiple modular multiplications and exponentiation operations with a large exponent and modulus (usually longer than 512 bits) [29], making them extremely expensive to compute. Encryption also yields significantly larger ciphertexts, which, in turn, causes a huge communication overhead for data transfer. In additively HE schemes such as Paillier, a ciphertext takes roughly the same number of bits as the key size, irrespective of the plaintext size. As of 2019, the minimum secure key size for Paillier is 2048 [74], whilst a gradient is typically a 32-bit floating point. This already translates to 64$\times$ size inflation after encryption.

We further benchmark the computation overhead and the inflated ciphertexts of Paillier with varying key sizes. We use `python-paillier` [75] to encrypt and then decrypt 900K 32-bit floating points. Table 3.1 reports the results on a `c5.4xlarge` instance. As the key size increases (higher security), both the computation overhead and the size of ciphertexts grow linearly. Since Paillier can only encrypt integers, floating point values must be scaled beforehand, and their exponents information contribute further to data inflation.

Summary. The prohibitive computation and communication overhead caused by HE, if not properly addressed, would lead to two serious economic consequences. First, given the dominance of HE operations, accelerating model computation using high-end hardware devices (e.g., GPUs and TPUs) is no longer relevant—a huge waste of the massive infrastructure investments in clients' datacenters. Second, the overwhelming network traffics across geo-distributed datacenters incurs skyrocketing Internet data charges, making cross-silo FL economically unviable. In fact, in WeBank, production FL applications may choose to turn off HE if the security requirement is not so strict.

3.2.2 Potential Solutions and Their Inefficiency

Hardware-Accelerated HE. HE process can be accelerated using software or hardware solutions. However, typical HE cryptosystems including Paillier have limited interleaving independent operations, thus the potential speedup of a single HE operation is quite limited. In fact, it is reported that a specialized FPGA can only accelerate Paillier encryption by 3× [29]. Moreover, simply accelerating the encryption itself does not help reduce the communication overhead.

Reducing Communication Overhead. As accelerating HE itself does not clear the barrier of adopting HE in FL, what if we reduce the amount of data to encrypt in the first place? Since data inflation is mainly caused by the mismatch between the lengths of plaintexts and ciphertexts, an intuitive idea would be *batching* as many gradients together as possible to form a *long* plaintext, so that the amount of encryption operations will reduce greatly. However, the challenge remains how to maintain HE's additive property after batching without modifying ML algorithms or hurting the learning accuracy.

While some prior works have explored the idea of joining multiple values together to reduce HE overhead, they give no viable implementation of batch encryption for cross-silo FL. [18] makes a false assumption that quantization is lossless, and uses adaptive optimizer Adam in its simulation even though its design does not support that. With only plain SGD available, [18] requires tedious learning rate scheduling tuning to achieve similar results of advanced optimizers [76]. The naive batching given in [19] cannot be correctly implemented as homomorphic additivity is not retained. In fact, none of these works have systematically studied the impact of batching. Gazelle [77] and SEAL [78] adopt the SIMD (single instruction multiple data) technique to speed up HE. However, such approach only applies to lattice-based HE schemes [79] and is restricted by their unique properties. For instance, it incurs dramatic computational complexity for lattice-based HE schemes to support more levels of multiplication [77]. Besides, these works only accelerate integer cryptographic operations. How to maintain the training accuracy in cross-silo FL context remains an open problem.

3.3 BatchCrypt

In this section, we describe our solution for gradient batching. We begin with the technical challenges. We first show that gradient quantization is required to enable batching. We then explain that generic quantization scheme lacks flexibility and efficiency to support general ML algorithms, which calls for an appropriately designed encoding and batching scheme; to prevent model quality degradation, an efficient clipping method is also needed. We name our solution BatchCrypt, a method that co-designs quantization, batch encoding, and analytical quantization modeling to boost computation speed and communication efficiency while preserving model quality in cross-silo FL with HE.

3.3.1 Why is HE Batching for FL a Problem?

On the surface, it seems straightforward to implement gradient batching. In fact, batching has been used to speed up queries over integers in a Paillier-secured database [80]. However, this technique only applies to *non-negative integers* [80]. In order to support floating numbers, the values have to be *reordered and grouped by their exponents* [80]. Such constraints are the key to preserving HE's additivity of batched ciphertexts—that is, the sum of two batched ciphertexts, once decrypted, should match the results of element-wise adding plaintext values in the two groups. Gazelle and SEAL [77, 78] employ SIMD technique to meet this requirement, but the approach is limited to lattice-based cryptosystems. We aspire to propose a universal batching method for all additively homomorphic cryptosystems.

Why is Quantization Needed? Gradients are *signed floating values* and must be ordered by their corresponding model weights, for which we cannot simply rearrange them by exponents. The only practical approach is to use integer representations of gradients in the batch, which requires quantization.

Existing Quantization Schemes. ML algorithms are resilient to update noise and able to converge with gradients of limited precision [81]. Fig. 3.3a illustrates how generic gradient quantization scheme can be used in HE batching. Notably, since there is no bit-wise

mapping between a ciphertext and its plaintext, permutation within ciphertexts is not allowed—only plain bit-by-bit addition between batched integers is available. Assume a gradient g in $[-1, 1]$ is quantized into an 8-bit unsigned integer. Let $[\cdot]$ denote the standard rounding function. The quantized value of g is

$$Q(g) = [255 * (g - min)/(max - min)],$$

where $max = 1$ and $min = -1$. Suppose n quantized gradients are summed up. The result, denoted by q_n, is dequantized as

$$Q^{-1}(q_n) = q_n * (max - min)/255 + n * min.$$

Referring to fig. 3.3a, gradients of a client (floating numbers in blue) are first quantized and then batch joined into a large integer. To aggregate the gradients of two clients, we simply sum up the two batched integers, locate the added gradients at the same bit positions as in the two batches (8-bit integers in red), and dequantize them to obtain the aggregated results.

Such a generic quantization scheme, though simple to implement, does not support aggregation well and has many ***limitations*** when applied to batched gradient aggregation.

(1) It is restrictive. To dequantize the results, it must know how many values are aggregated. This poses extra barriers to flexible synchronization, where the number of updates is constantly changing, sometimes even unavailable.

(2) It overflows easily in aggregation. As values are quantized into positive integers, aggregating them is bound to overflow quickly as the sum grows larger. To prevent overflow, batched ciphertexts have to be decrypted after a few additions and encrypted again in prior work [18].

(3) It does not differentiate *positive* overflows from *negative*. Once overflow occurs, the computation has to restart. Should we be able to tell them apart, a saturated value could have been used instead of discarding the results.

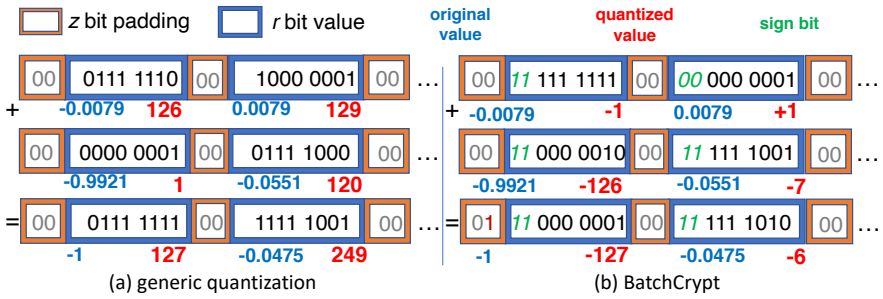


Figure 3.3: An illustration of a generic quantization scheme and BatchCrypt. The latter preserves additivity during batching, with the sign bits highlighted within values.

3.3.2 HE Batching for Gradients

Unsatisfied with the generic quantization technique, we aspire to devise a batching solution tailored to gradient aggregation. Our scheme should have the following desirable properties: (1) it preserves the additivity of HE; (2) it is more resilient to overflows and can distinguish positive overflows from negative ones; (3) it is generally applicable to existing ML algorithms and optimization techniques; (4) it is flexible enough that one can dequantize values directly without additional information, such as the number of values aggregated.

Gradient Quantization. Existing works use gradient compression techniques to reduce network traffic in distributed training [82, 83, 84, 85]. These quantization methods are mainly used to compress values for transmission [83] or accelerate inference where only multiplication is needed [38]. However, they are not designed for gradient aggregation, and we cannot perform computations over the compressed gradients efficiently, making them inadequate for FL. We scrutinize the constraints posed by our design objectives, and summarize the stemmed requirements for quantization as follows:

- **Signed Integers:** Gradients should be quantized into *signed* integers. In this way, positive and negative values can cancel each other out in gradient aggregation, making it less prone to overflowing than quantizing gradients into unsigned integers.
- **Symmetric Range:** To make values with opposite signs cancel each other out, the quantized range must be symmetrical. Violating this requirement may lead to an incorrect aggregation result. For example, if we map $[-1, 1]$ to $[-128, 127]$, then $-1 +$

1 would become $-128 + 127 = -1$ after quantization.

- **Uniform Quantization:** Literature shows that non-uniform quantization schemes have better compression rates as gradients have non-uniform distribution [84, 86]. However, we are unable to exploit the property as additions over quantized values are required.

BatchCrypt We now propose an efficient quantization scheme BatchCrypt that meets all the requirements above. Assume that we quantize a gradient in $[-\alpha, \alpha]$ into an r-bit integer. Instead of mapping the whole range all together, we *uniformly map* $[-\alpha, 0]$ and $[0, \alpha]$ to $[-(2^r - 1), 0]$ and $[0, 2^r - 1]$, respectively. Note that the value 0 ends up with two codes in our design. Prior work shows that 16-bit quantization ($r = 16$) is sufficient to achieve near lossless gradient quantization [87]. We will show in §3.5 that such a moderate quantization width is sufficient to enable efficient batching in FL setting.

With quantization figured out, the challenge remains how to encode the quantized values so that signed additively arithmetic is correctly enabled—once the batched long integer is encrypted, we cannot distinguish the sign bits from the value bits during aggregation. Inspired by how modern CPUs handle signed integer computations, we use *two's complement representation* in our encoding. By doing so, the sign bits can engage in the addition just like the value bits. We further use the *two sign bits* to differentiate between the positive and negative overflows. We illustrate an example of BatchCrypt in fig. 3.3b. By adding the two batched long integers, BatchCrypt gets the correct aggregation results for $-1 + (-126)$ and $+1 + (-7)$, respectively.

BatchCrypt achieves our requirements by co-designing quantization and encoding: no additional information is needed to dequantize the aggregated results besides the batch itself; positive and negative values are able to offset each other; the signs of overflow can be identified. Compared with the batching methods in [77, 78], BatchCrypt's batching scheme is generally applicable to all additively HE cryptosystems' and fully HE cryptosystems' additive operations.

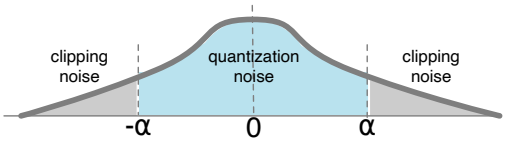


Figure 3.4: A typical layer gradient distribution. α is the clipping threshold.

3.3.3 dACIQ: Analytical Clipping for FL

Our previous discussion has assumed gradients in a bounded range (§3.3.2). In practice, however, gradients may go unbounded and need to be *clipped* before quantization. Also, gradients from different layers have different distributions [83]. We thus need to quantize layers individually [83, 84]. Moreover, prior works show that gradients from the same layer have a bell-shaped distribution which is near Gaussian [86, 88, 89]. Such property can be exploited for efficient gradient compression [83, 84]. Finally, gradients require *stochastic rounding* during quantization [83, 87, 82], as it stochastically preserves diminishing information compared to *round-to-nearest*.

Layer-wise quantization and stochastic rounding can be easily applied, yet it remains unclear how to find the optimal clipping thresholds in the FL setting. As shown in fig. 3.4, clipping is the process of saturating the outlaying gradients beyond a threshold α. If α is set too large, the quantization resolution becomes too low. On the other hand, if α gets too small, most of the range information from outlaying gradients has to be discarded.

In general, there are two ways to set the clipping threshold, *profiling-based* methods and *analytical modeling*. Profiling-based clipping selects a sample dataset to obtain a sample gradient distribution. Thresholds are then assessed with metrics such as KL divergence [90] and convergence rate [83]. However, such approach is impractical in FL for three reasons. First, finding a representative dataset in FL can be difficult, as clients usually have non-i.i.d. data, plus it breaks the data silo. Second, the gradient range narrows slowly as the training progresses [91], so clipping needs to be calibrated constantly, raising serious overhead concerns. Third, the profiling results are specific to the training models and datasets. Once the models or the datasets change, new profiling is needed. For both practicality and cost considerations, BatchCrypt instead adopts analytical modeling.

As shown in fig. 3.4, the accumulated noise comes from two sources. *Quantization*

noise refers to the error induced by rounding within the clipping range (the light blue area), while *clipping noise* refers to the saturated range beyond the clipping threshold (the gray area). To model the accumulated noise from both quantization and clipping, state-of-the-art clipping technique ACIQ [38] assumes that they follow a Gaussian distribution. However, ACIQ cannot be directly applied to BatchCrypt for two reasons. First, it employs a generic asymmetric quantization, which is not the case in BatchCrypt; second, in FL, gradients are not available at one place in plaintext to conduct distribution fitting.

We address these problems by extending ACIQ clipping to the distributed FL setting, which we call dACIQ. In particular, we adopt stochastic rounding with an r-bit quantization width. Assume that gradients follow Gaussian distribution $X \sim N(0, \sigma^2)$. Let q_i be the i-th quantization level. We compute the accumulated error in BatchCrypt as follows:

$$\begin{aligned} &E[(X - Q(X))^2] = \int_{-\infty}^{-\alpha} f(x) \cdot (x + \alpha)^2 dx + \int_{\alpha}^{\infty} f(x) \cdot (x - \alpha)^2 dx \\ &+ \sum_{i=0}^{2^r - 3} \int_{q_i}^{q_{i+1}} f(x) \ \cdot [\ (x - q_i)^2 \cdot (\frac{q_{i+1} - x}{\triangle}) \ + \ (x - q_{i+1})^2 \cdot (\frac{x - q_i}{\triangle})\ ] dx \\ &\approx \frac{\alpha^2 + \sigma^2}{2} \cdot [1 - erf(\frac{\alpha}{\sqrt{2}\sigma})] - \frac{\alpha \cdot \sigma \cdot e^{-\frac{\alpha^2}{2 \cdot \sigma^2}}}{\sqrt{2\pi}} + \frac{2\alpha^2 \cdot (2^r - 2)}{3 \cdot 2^{3r}}, \end{aligned} \tag{3.1}$$

where the first and the second terms account for the clipping noise, and the third the rounding noise. As long as we know σ, we can then derive the optimal threshold α from Eq. (3.1). We omit the detailed derivations in the interest of space.

Gaussian Fitting. Now that we have Eq. (3.1), we still need to figure out how to fit gradients into a Gaussian distribution in the FL setting. Traditionally, to fit Gaussian parameters μ and σ, Maximum Likelihood Estimation and Bayesian Inference can be used. They require information including the size of observation set, its sum, and its sum of squares. As an ML model may have up to millions of parameters, calculating these components as well as transferring them over Internet is prohibitively expensive. As a result, dACIQ adopts a simple, yet effective Gaussian fitting method proposed in [92]. The method only requires the size of observation set and its max and min, with the minimum computational and communication overhead. We later show that such light-weight fitting does not affect model accuracy in §3.5.

Algorithm 3 HE FL BatchCrypt

Aggregator:

function INITIALIZE
    Issue INITIALIZELEADER() to the randomly selected leader
    Issue INITIALIZEOTHER() to the other clients
function STARTSTRAINING
    for epoch $e = 0, 1, 2, ...,$ E **do**
        Issue WORKERSTARTSEPOCH(e) to all clients
        for all training batch $t = 0, 1, 2, \cdots,$ T **do**
            Collect gradients `range` and `size`
            Return clipping values α calculated by dACIQ
            Collect, sum up all $g_i^{(e,t)}$ into $g^{(e,t)}$, and dispatch it

Client Worker: $i = 1, 2, \ldots, m$

– r: quantization bit width, bs: BatchCrypt batch size

function INITIALIZELEADER
    Generate HE key-pair `pub_key` and `pri_key`
    Initialize the model to train `w`
    Send `pub_key`, `pri_key`, and `w` to other clients
function INITIALIZEOTHER
    Receive HE key-pair `pub_key` and `pri_key`
    Receive the initial model weights `w`
function WORKERSTARTSEPOCH(e)
    for all training batch $t = 0, 1, 2, \cdots,$ T **do**
        Compute gradients $g_i^{(e,t)}$ based on w
        Send per-layer `range` and `size` of $g_i^{(e,t)}$ to aggregator
        Receive the layer-wise clipping values α's
        Clip $g_i^{(e,t)}$ with corresponding α, quantize $g_i^{(e,t)}$ into r bits, with quantization range setting to $m\alpha$ ▷ Advance scaling
        Batch $g_i^{(e,t)}$ with bs layer by layer
        Encrypt batched $g_i^{(e,t)}$ with `pri_key`
        Send encrypted $g_i^{(e,t)}$ to aggregator
        Collect $g^{(e,t)}$ from aggregator, and decrypt with `pub_key`
        Apply decrypted $g^{(e,t)}$ to w

Advance Scaling. With multiple clients in FL, it is essential to prevent overflows from happening. Thanks to clipping, the gradient range is predetermined before encryption. Let m be the number of clients. If m is available, we could employ *advance scaling* by setting the quantization range to m times of the clipping range, so that the sum of gradients from all clients will not overflow.

In this work, we assume that gradients follow Gaussian distribution. However, one can extend this assumption to other distributions like Laplace by deriving a new Eq. (3.1) as well as finding efficient fitting methods.

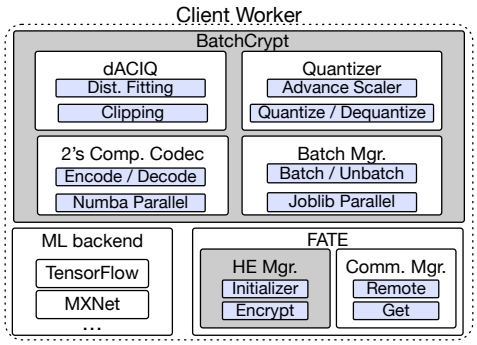


Figure 3.5: The architecture of a client worker in BatchCrypt.

3.3.4 BatchCrypt: Putting It All Together

Putting it all together, we summarize the workflow of BatchCrypt in algorithm 3.

Initialization. The aggregator randomly selects one client as the leader. The leader client generates the HE key-pair and initializes the model weights. The key-pair and model weights are then synchronized with the other client workers.

Training. After initialization, there is no differentiation between the leader and the other workers. Clients compute gradients and send the per-layer gradient range and size to the aggregator. The aggregator estimates the Gaussian parameters first and then calculates the layer-wise clipping thresholds as described in §3.3.3. Clients then quantize the gradients with range scaled by the number of clients, and encrypt the quantized values using BatchCrypt. Note that advanced scaling utilizing the number of clients is used to completely avoid overflowing. However, algorithm 3 is still viable even without that information, as BatchCrypt supports overflow detection. The encrypted gradients are gathered at the aggregator and summed up before returning to the clients.

3.4 Implementation

We have implemented BatchCrypt atop FATE (v1.1) [73]. While we base our implementation on FATE, nothing precludes it from being extended to the other frameworks such as TensorFlow Federated [93] and PySyft [94].

Overview. Our implementation follows the paradigm described in algorithm 3, as most of the efforts are made on the client side. Fig. 3.5 gives an overview of the client architecture.

BatchCrypt consists of dACIQ, Quantizer, two's Compliments Codec, and Batch Manager. dACIQ is responsible for Gaussian fitting and clipping threshold calculation. Quantizer takes the thresholds and scales them to quantize the clipped values into signed integers. Quantizer also performs dequantization. Two's Compliments Codec translates between a quantized value's true form and two's compliment form with two sign bits. Given the large volume of data to encode, we adopt `Numba` to enable faster machine codes and massive parallelism. Finally, Batch Manager oversees batching and unbatching gradients in their two's compliment form, it remembers data's original shape before batching and restores it during unbatching. Batch Manager utilizes `joblib` to exploit computing resources by multiprocessing. FATE is used as an infrastructure to conduct FL, in which all the underlying ML computations are written with TensorFlow v1.14 optimized for our machines shipped with AWS DLAMI [95]. FATE adopts the open-sourced `python-paillier` as the Paillier HE implementation. We again employ `joblib` to parallel the operations here. FATE's Communication Manager conducts the SSL/TLS secured communication with `gRPC`. During our characterizations and evaluations, the CPUs are always fully utilized during Paillier operations and BatchCrypt process.

Model Placement. In the typical parameter server architecture, model weights are placed on the server side, while we purposely place weights on the worker side in BatchCrypt. Prior work [18] employs the traditional setup: clients encrypt the initialized weights with HE and send them to the aggregator first; the aggregator applies the received encrypted gradients to the weights encrypted with the same HE key. Such placement has two major drawbacks. First, keeping weights on the aggregator requires *re-encryption*. Since new gradients are constantly applied to weights, the model must be sent back to the clients to decrypt and re-encrypt to avoid overflows from time to time, resulting in a huge overhead. Second, applying encrypted gradients prevents the use of sophisticated ML optimizers. State-of-the-art ML models are usually trained with adaptive optimizers [72] that scale the learning rates according to the gradient itself. By keeping the model weights

on the client side, BatchCrypt can examine the aggregated plaintext gradients, enabling the use of advanced optimizers like Adam, whereas on the aggregator side, one can only adopt plain SGD.

3.5 Evaluation

In this section, we evaluate the performance of BatchCrypt with real ML models trained in geo-distributed datacenters. We first examine the learning accuracy loss caused by our quantization scheme (§3.5.2). We then evaluate the computation and communication benefits BatchCrypt brings as well as how its performance compares to the ideal plaintext learning (§3.5.3). We then assess how BatchCrypt's speedup may change with various batch sizes (§3.5.4). Finally, we demonstrate the significant cost savings achieved by BatchCrypt (§3.5.5).

3.5.1 Methodology

Setting. We consider a geo-distributed FL scenario where nine clients collaboratively train an ML model in five AWS EC2 datacenters located in Tokyo, Hong Kong, London, N. Virginia, and Oregon, respectively. We launched two compute-optimized `c5.4xlarge` instances (16 vCPUs and 32 GB memory) as two clients in each datacenter except that in Oregon, where we ran only one client. Note that we opt to not use GPU instances because computation is not a bottleneck. We ran one aggregator in the Oregon datacenter using a memory-optimized `r5.4xlarge` instance (16 vCPUs and 128 GB memory) in view of the large memory footprint incurred during aggregation. To better outline the network heterogeneity caused by geo-locations, we profiled the network bandwidth between the aggregator and the client instances. Our profiling results are summarized in table 3.2. We adopt Pailler cryptosystem in our evaluation as it is widely adopted in FL [29], plus batching over it is not supported by Gazelle or SEAL [77, 78]. We expect our results also apply to other cryptosystems as BatchCrypt offers a generic solution.

Benchmarking Models. As there is no standard benchmarking suites for cross-silo FL, we implemented three representative ML applications in FATE v1.1. Our first application

Table 3.2: Network bandwidth (Mbit/sec) between aggregator and clients in different regions.

Region	Ore.	TYO.	N.VA.	LDN	HK
Uplink (Mbps)	9841	116	165	97	81
Downlink (Mbps)	9841	122	151	84	84

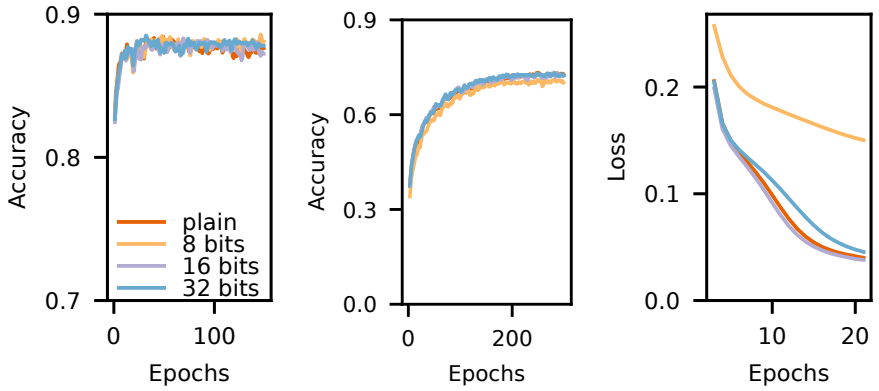
Table 3.3: Summary of models used in characterizations.

	FMNIST	**CIFAR**	**LSTM**
Network	3-layer FC	AlexNet [97]	LSTM [98]
Weights	101.77K	1.25M	4.02M
Dataset	FMNIST [96]	CIFAR10 [47]	Shakespeare [99]
Task	Image class.	Image class.	Text generation

is a 3-layer fully-connected neural network trained over FMNIST dataset [96], where we set the training batch size to 128 and adopt Adam optimizer. In the second application, we train AlexNet [97] using CIFAR10 dataset [47], with batch size 128 and RMSprop optimizer with 10^{-6} decay. The third application is an LSTM model [98] with Shakespeare dataset [99], where we set the batch size to 64 and adopt Adam optimizer. Other LSTM models that are easier to validate have significantly more weights. Training them to convergence is beyond our cloud budget. As summarized in table 3.3, all three applications are backed by deep learning models of various sizes and cover common learning tasks such as image classification and text generation. For each application, we randomly partition its training dataset across nine clients. We configure synchronous training unless otherwise specified.

3.5.2 Impact of BatchCrypt's Quantization

We first evaluate the impact of our quantization scheme, and see how quantization bit width could affect the model quality. We report the test accuracy for FMNIST and CIFAR workloads to see how BatchCrypt's quantization affects the classification top-1 accuracy. Training loss is used for LSTM as the dataset is unlabelled and has no test set. We simulated the training with nine clients using BatchCrypt's quantization scheme including dACIQ clipping. The simulation scripts are also open-sourced for public access. We set the quantization bit width to 8, 16, and 32, respectively, and compare the results against plain training (no encryption) as the baseline. We ran the experiments until convergence,



(a) FMNIST test acc. (b) CIFAR test acc. (c) LSTM train loss

Figure 3.6: The quality of trained model with different quantization bit widths in BatchCrypt.

which is achieved when the accuracy or loss does not reach a new record for three consecutive epochs.

Fig. 3.6 depicts the results. For FMNIST, plain baseline reaches peak accuracy 88.62% at the 40^{th} epoch, while the 8-bit, 16-bit, and 32-bit quantized training reach 88.67%, 88.37%, and 88.58% at the 122^{nd}, 68^{th}, and 32^{nd} epoch, respectively. For CIFAR, plain baseline reaches peak accuracy 73.97% at the 285^{th} epoch, while the 8-bit, 16-bit, and 32-bit quantized training reach 71.47%, 74.04%, and 73.91% at the 234^{th}, 279^{th}, and 280^{th} epoch, respectively. Finally, for LSTM, plain baseline reaches bottom loss 0.0357 at the 20^{th} epoch, while the 8-bit, 16-bit, and 32-bit quantized training reach 0.1359, 0.0335, and 0.0386 at the 29^{th}, 23^{rd}, and 22^{nd} epoch, respectively. We hence conclude that, with appropriate quantization bit width, BatchCrypt's quantization has negligible negative impact on the trained model quality. Even in the case where the quantized version requires more epochs to converge, we later show that such overhead can be more than compensated by the speedup from BatchCrypt.

Although 8-bit quantization performs poorly for CIFAR and LSTM, it is worth notice that, longer bit width does not necessarily lead to higher model quality. In fact, quantized training sometimes achieves better results. Prior quantization work has observed similar phenomenon [100], where the stochasticity introduced by quantization can work as a regularizer to reduce overfitting, similar to a dropout layer [101]. Just like the dropout rate, quantization bit width acts as a trade-off knob for how much information is retained and

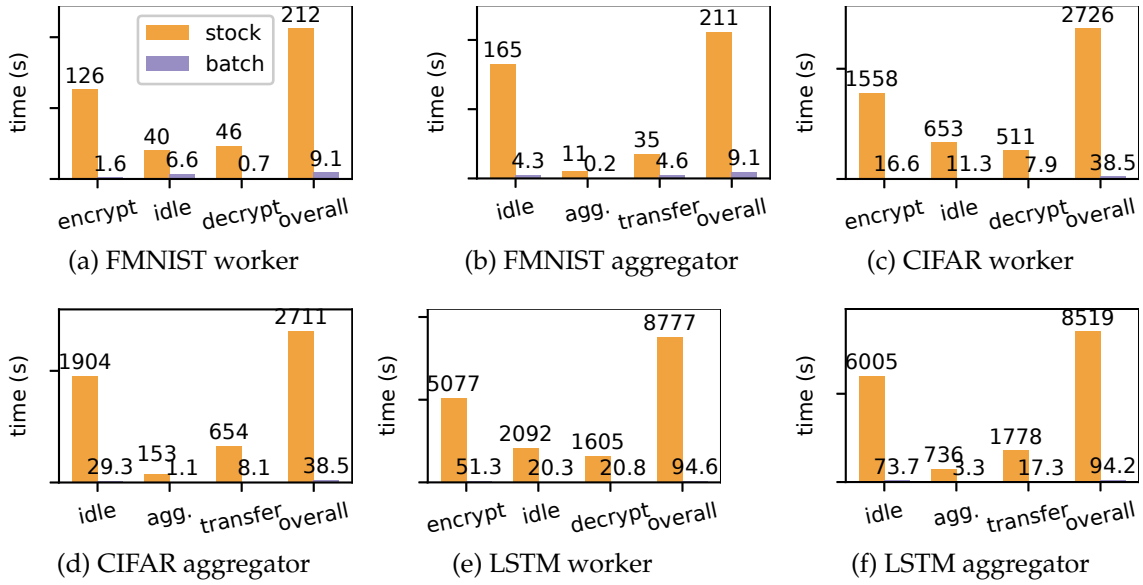


Figure 3.7: Breakdown of training iteration time under stock FATE and BatchCrypt, where "idle" measures the idle waiting time of a worker and "agg." measures the gradient aggregation time on the aggregator. Note that model computation is left out here as it contributes little to the iteration time.

how much stochasticity is introduced.

In summary, with apt bit width, our gradient quantization scheme does not adversely affect the trained model quality. In contrast, existing batching scheme introduces 5% of quality drop [19]. Thus, quantization-induced error is not a concern for the adoption of BatchCrypt.

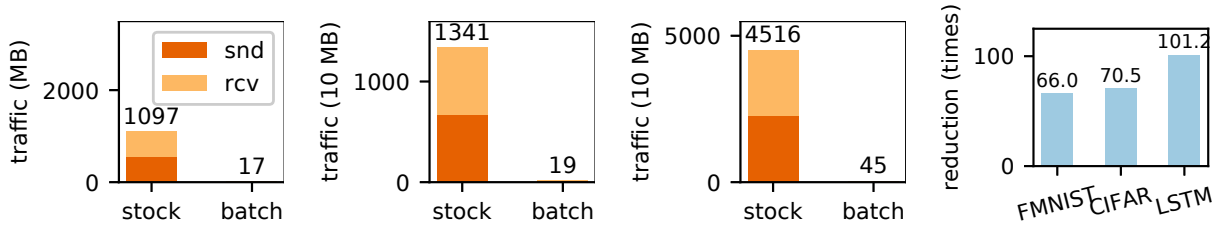
3.5.3 Effectiveness of BatchCrypt

BatchCrypt vs. FATE. We next evaluate the effectiveness of BatchCrypt in real deployment. We set the quantization bit width to 16 as it achieves a good performance (§3.5.2). The batch size is set to 100, in which we pad two zeros between the two adjacent values. We report two metrics: the iteration time breakdown together with the network traffic. We ran the experiments for 50 iterations, and present the averaged results against those measured with the stock FATE implementation in fig. 3.7 and fig. 3.8. We see in fig. 3.2 that BatchCrypt significantly speeds up a training iteration: 23.3× for FMNIST, 70.8× for CIFAR, and 92.8× for LSTM. Iteration time breakdown further shows that our implementa-

tion reduces the cost of HE related operations by close to 100×, while the communication time is substantially reduced as well ("idle" in worker and "transfer" in aggregator).

We next refer to fig. 3.8, where we see that BatchCrypt reduces the network footprint by up to 66×, 71×, and 101× for FMNIST, CIFAR, and LSTM, respectively. Note that FATE adopts `grpc` as the communication vehicle whose limit on payload forces segmenting encrypted weights into small chunks before transmission. By reducing the size of data to transfer, BatchCrypt alleviates the segmenting induced overhead (metadata, checksum, etc.), so it is possible to observe a reduction greater than the batch size.

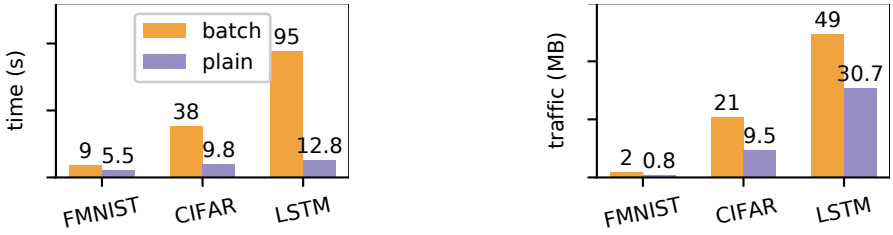
Our experiments also show that BatchCrypt achieves more salient improvements for larger models. First, encryption related operations take up more time in larger models, leaving more potential space for BatchCrypt. Second, since layers are batched separately, larger layers have higher chances forming long batches. BatchCrypt's speedup can be up to two orders of magnitude, which easily offset the extra epochs needed for convergence caused by quantization (§3.5.2).



(a) FMNIST (b) CIFAR (c) LSTM (d) Reduction

Figure 3.8: Comparison of the network traffic incurred in one training iteration using the stock FATE implementation and BatchCrypt.

BatchCrypt vs. Plaintext Learning. We next compare BatchCrypt with the plain distributed learning where no encryption is involved—an ideal baseline that offers the optimal performance. Fig. 3.9 depicts the iteration time and the network footprint under the two implementations. While encryption remains the major bottleneck, BatchCrypt successfully reduces the overhead by an order of magnitude, making it practical to achieve the same training results as the plain distributed setting. Note that encrypted numbers in FATE each carries redundant information such as public keys, thus causing the communication inflation compared with the plain version. Such inflation can be reduced if FATE



(a) Time (b) Communication

Figure 3.9: Time and communication comparisons of one iteration on workers between BatchCrypt and plain distributed learning without encryption.

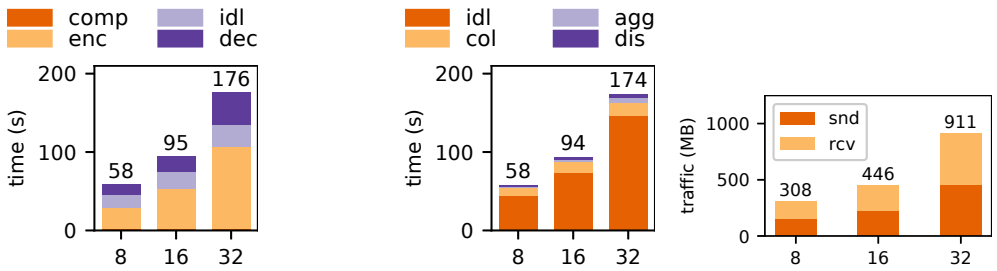
Table 3.4: Projected total training time and network traffic usage until convergence for the three models. The converged test accuracy for FMNIST, CIFAR as well as loss for LSTM and their corresponding epoch numbers are listed in the table.

Model	**Mode**	**Epochs**	**Acc./Loss**	**Time** (h)	**Traffic** (GB)
FMNIST	stock	40	88.62%	122.5	2228.3
	batch	68	88.37%	8.9	58.7
	plain	40	88.62%	3.2	11.17
CIFAR	stock	285	73.97%	9495.6	16422.0
	batch	279	74.04%	131.3	227.8
	plain	285	73.97%	34.2	11.39
LSTM	stock	20	0.0357	8484.4	15347.3
	batch	23	0.0335	105.2	175.9
	plain	20	0.0357	12.3	10.4

employs some optimized implementation.

Training to Convergence Our previous studies mainly focus on a single iteration. Compared with stock FATE and plain distributed learning, BatchCrypt requires a different number of iterations to converge. We hence evaluate their end-to-end performance by training ML models till convergence. As this would take exceedingly long time and high cost if performed in real deployment, we instead utilize our simulation in §3.5.2 and iteration profiling results to project the total time and network traffic needed for convergence.

Table 3.4 lists our projection results of the three solutions. Compared with the stock implementation in FATE, BatchCrypt dramatically reduces the training time towards convergence by 13.76×, 72.32×, and 80.65× for FMNIST, CIFAR, and LSTM, respectively. In the meantime, the network footprints shrink by 37.96×, 72.01×, 87.23×, respectively. We stress that these performance improvements are achieved without degrading the trained model quality. On the other hand, BatchCrypt only slows down the overall training time



(a) Worker: **comp**ute, **en**crypt, **idl**e, **dec**rypt

(b) Aggregator: **idl**e, **col**lect, **agg**regate, **dis**patch

(c) Aggregator traffic

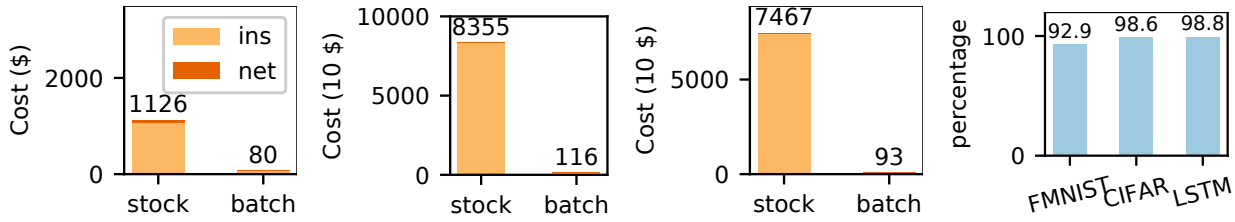
Figure 3.10: Breakdown of iteration time and communication traffic of BatchCrypt with LSTM model with various quantization bit widths in one iteration. The corresponding batch sizes for bit width 8, 16, and 32 are 200, 100, and 50, respectively.

by 1.78×, 2.84×, and 7.55× for the three models compared with plain learning—which requires no encryption and hence achieves the fastest possible training convergence. In summary, BatchCrypt significantly reduces both the computation and communication overhead caused by HE, enabling efficient HE for cross-silo FL in production environments.

3.5.4 Batching Efficiency

We have shown in §3.5.2 that ML applications have different levels of sensitivity towards gradient quantization. It is hence essential that BatchCrypt can efficiently batch quantized values irrespective of the chosen quantization bit width. Given an HE key, the longest plaintext it can encrypt is determined by the key size, so the shorter the quantization width is, the larger the batch size is, and the higher the potential speedup could be. We therefore look into how our BatchCrypt implementation can exploit such batching speedup.

We evaluate BatchCrypt by varying the batch size. In particular, we train the LSTM model on the geo-distributed clients with different quantization widths 8, 16, and 32. The corresponding batch sizes are set respectively to 200, 100, and 50. We ran the experiments for 50 iterations, and illustrate the average statistics in fig. 3.10. Figs. 3.10a and 3.10b show the time breakdown in the three experiments. It is clear that employing a shorter quantization bit width enables a larger batch size, thus leading to a shorter training time. Note



(a) FMNIST (b) CIFAR (c) LSTM (d) Savings

Figure 3.11: Total cost until convergence between FATE's stock implementation and BatchCrypt, **ins**tance and **net**work costs are highlighted separately.

that the speedup going from 8-bit to 16-bit is smaller compared with that from 16-bit to 32-bit, because HE operations become less of a bottleneck with larger batch size. Fig. 3.10c depicts the accumulated network traffic incurred in one iteration, which follows a similar trend as that of the iteration time. In conclusion, BatchCrypt can efficiently exploit batching thanks to its optimized quantization. Similar to [78, 77], BatchCrypt's batching scheme reduces both the computation and communication cost linearly as the batch size increases. In fact, if lattice-based HE algorithms are adopted, one can replace BatchCrypt's batching scheme with that of [78, 77], and still benefit from BatchCrypt's accuracy-preserving quantization.

3.5.5 Cost Benefits

The reduced computation and communication overheads enable significant cost savings: sustained high CPU usage leads to high power consumption, while ISPs charge for bulk data transfer over the Internet. As our evaluations were conducted in EC2, which provides a runtime environment similar to the organization's own datacenters, we perform cost analysis under the AWS pricing scheme. The hourly rate of our cluster is $8.758, while the network is charged based on outbound traffic for $0.042, $0.050, $0.042, $0.048, $0.055 per GB for the regions listed in table 3.2.

We calculate the total cost for training until convergence in table 3.4 and depict the results in fig. 3.11. As both computation and communication are reduced substantially, BatchCrypt achieves huge cost savings over FATE. While the instance cost reduction is the same as the overall speedup in table 3.4, BatchCrypt lowers the network cost by 97.4%, 98.6% and 98.8% for FMNIST, CIFAR, and LSTM, respectively.

3.6 Discussion

Local-update SGD & Model Averaging. Local-update SGD & model averaging is another common approach to reducing the communication overhead for FL [30, 71], where the aggregator collects and averages model weights before propagating them back to clients. Since there are only addition operations involved, BatchCrypt can be easily adopted.

Split Model Inference In many FL scenarios with restrictive privacy requirement, a trained model is split across clients, and model inference involves coordination of all those clients [13, 102]. BatchCrypt can be used to accelerate the encryption and transmission of the intermediate inference results.

Flexible Synchronization There have been many efforts in amortizing the communication overhead in distributed SGD by removing the synchronization barriers [24, 67, 68]. Although we only evaluate BatchCrypt's performance in synchronous SGD, our design allows it to take advantage of the flexible synchronization schemes proposed in the literature. This is not possible with Secure Aggregation [17].

Potential on Large Models Recent research and our evaluations show that more sophisticated ML models are more resilient to quantization noise. In fact, certain models are able to converge even with 1- or 2-bit quantization [103, 83]. The phenomenon promises remarkable improvement with BatchCrypt, which we will explore in our future work.

Applicability in Vertical FL Vertical FL requires complicated operations like multiplying ciphertext matrices [13, 64]. Batching over such computation is beyond BatchCrypt's current capability of only supporting additive operations. We will leave it as a future work.

3.7 Summary

In this chapter, we have systematically studied utilizing HE to implement secure cross-silo FL. We have shown that HE related operations create severe bottlenecks on computation and communication. To address this problem, we have presented BatchCrypt, a system solution that judiciously quantizes gradients, encodes a batch of them into long integers, and performs batch encryption to dramatically reduce the encryption overhead and the total volume of ciphertext. We have implemented BatchCrypt in FATE and evaluated its

performance with popular machine learning models across geo-distributed datacenters. Compared with the stock FATE, BatchCrypt accelerates the training convergence by up to 81× and reduces the overall traffic by 101×, saving up to 99% cost when deployed in cloud environments.

CHAPTER 4

PROTECTING DATA PRIVACY AND MODEL CONFIDENTIALITY FOR COLLABORATIVE LEARNING WITH SGX

After covering data privacy for data owners in FL, this chapter explores the scenarios in distributed ML training where model owner is introduced, and has its own privacy requirements.

4.1 Background and Related Work

4.1.1 Collaborative ML and Threat Model

In many application domains such as healthcare and finance, building a high-quality ML model requires the participation of both model owner and data owners. The model owner (e.g., a tech company or ML developer) has advanced ML expertise but may have no access to diverse training datasets. On the other hand, the data owners (e.g., hospitals and retailers) have quality labeled datasets, but any single one of them may not have enough data samples or ML expertise to build a quality model. An ideal solution is to have data owners collaborating with model owner in a way such that the model developed by the latter can be trained over the data owned by the former, while still preserving data privacy and model confidentiality.

4.1.2 Entities in Collaborative ML

Collaborative ML typically involves three entities, a model owner, many data owners, and a third-party infrastructure such as a public cloud for providing training resources. These entities have different goals in collaborative ML.

Data Owner. As explained in §3.1, due to privacy concerns and government regulations, each data owner wants to protect their data from being exposed to other entities, including the model owner, the cloud, and other data owners.

Model Owner. For the model owner, protecting the confidentiality of the training model (design and weights) is a top requirement. First, the model itself is a valuable intellectual property as its development demands tremendous research and engineering efforts [21, 22]. Protecting it helps maintain the model owner's technological advances and supports its business as data owners would otherwise train the model by themselves. Second, maintaining the model confidentiality is also a security requirement. Prior work shows that sharing the training model with (untrusted) participants poses new threats that are hard to defend such as membership inference, model inversion, and backdoor attack [104, 105, 106]. In security-critical applications such as fraud detection and spam filtering, exposing the details of the used model leads to a wider attack surface as adversaries can forge attacks to evade the model-provided defense mechanism by offline trial and error [107].

In addition to the confidentiality of the model itself, the model owner also wants to conceal the *ML training method* such as optimizer selection and configuration [108], gradient manipulation [109] and learning rate schedule [110]. These methods are critical to the training performance. Selecting, combining, and configuring them require extensive ML expertise, which are part of the model owner's intellectual property.

Third-Party Cloud. Training complex ML models over vast quantities of data requires a large amount of computational resources, which the model owner and the data owners may not have. Therefore, a common practice is to rent a large number of virtual instances on a third-party cloud (e.g., Azure and AWS) and perform distributed training across those instances.

4.1.3 Threat Model

We assume that the model owner and data owners are *honest but curious*. The data owners might act on their own or collude with each other to steal the training model and the

training method so that they can perform training on their own. Data owners also want to pry on each other's data to improve their competitiveness in the same business sector. The model owner, on the other hand, wants to access the training data for illicit use. Participants have no incentive to hinder the training and will follow the training protocols honestly.

Training is performed on a third-party cloud trusted by *neither the data owners nor the model owner*. The cloud instances, including privileged software like OS and hypervisor, are not trusted. Attacks can be performed by the cloud provider or anyone with access to the OS/hypervisor. However, data and model owners have to trust the implementation of Trusted Computing Base (TCB) (e.g., Intel SGX) and its attestation service. We also assume that the participants trust standard ML frameworks like TensorFlow [9] and PyTorch [111]. These frameworks are developed by reputable organizations and are under public scrutiny in open-source communities.

We do not address side-channel attacks [112] and denial-of-service attacks as they can be prevented [113, 114] and are out of the scope of the paper. We also leave out rollback attacks [115] on the data stored beyond enclaves, as they can be handled by existing approaches [116, 117]. Finally, we do not consider the model owner colluding with some data owners to steal others' data, as victim data owners can effectively guard against this attack with differential privacy [34].

4.2 Prior Arts and Their Insufficiency

In this section, we discuss why prior arts are insufficient to protect data privacy and model confidentiality for collaborative learning under the threat model introduced in §4.1.3. We start by introducing existing solutions designed for different collaborative learning scenarios and explaining why they cannot be adapted here. We then introduce the SGX-based solutions, which are the most related to our work, and discuss their problems for scalable collaborative learning.

4.2.1 Existing Solutions for Different Collaborative Learning Scenarios

Collaborative ML has been studied in the literature, including federated learning and split learning. However, existing solutions focus on quite different use scenarios, where only data privacy is considered but not model confidentiality.

Federated learning (FL) Like we have presented in §3.1, FL focuses primarily on data privacy, and is not meant to protect model confidentiality. Models have to be shared to all data owners in the clear, so that data owners can produce updates based on local data. Such practice is not sufficient for the model confidentiality requirement of our threat model, and cannot be intuitively extended to support it.

Split Learning (SL) offers an alternative approach to collaborative ML for training deep neural networks [32, 31]. In SL, a neural network is split into two parts from a certain layer, called a *cut layer*. The model owner releases the network up to the cut layer to data owners, while keeping the rest of the layers private to itself. The data owners train the network up to the cut layer with their private data and send the updates to a central server, based on which the model owner trains the remaining network. While this scheme preserves data privacy, the model confidentiality cannot be fully protected, as the network up to the cut layer still needs to be shared among data owners. Also, the parameters of that network are accessible to data owners only, meaning that the model owner has no access to the complete network of the trained model.

Given that these collaborative learning scenarios assume that the actual training task is distributed among data owners, they have fundamentally different system architectures than the scenario we focus on and thus these existing solutions are difficult to be adapted here.

4.2.2 Intel SGX

ML training requires access to both data and model. To protect their confidentiality, the training must be performed in a secure place trusted by both data and model owners. The trusted hardware offers a viable solution, which applications can use to create a trusted execution environment (TEE) even if the underlying platform is untrusted.

Intel SGX (Software Guarded Extensions [33]) is the most widely available hardware-assisted TEE compared with other implementations such as ARM TrustZone [118] and AMD Secure Memory Encryption (SME) [119]. It sets aside a protected memory region, called an *enclave*, within an application's address space. Code execution and memory access in an enclave are strongly isolated from external programs. The processor ensures that only code running in an enclave can access data loaded into it. External programs, including operating system (OS) and hypervisor, can invoke code inside an enclave only at statically-defined entry points. SGX also supports remote *attestation*, which allows a remote user to verify that the initial code and data loaded into an enclave match a given cryptographic hash, hence ensuring the enclave to perform the expected computation.

However, the enclave's hardware-protected confidentiality and integrity come with a steep price of performance. First, as the host platform is untrusted, copying between CPU and enclave memory must be protected to prevent memory bus snooping. SGX uses Memory Encryption Engine (MEE) to transparently encrypt and decrypt data exchanges through memory bus, incurring 2X-3X performance overhead than native execution [120]. Second, the performance of an enclave is usually bounded by the EPC (enclave page cache) size, a hardware-protected memory region used to host the enclave pages. The EPC size is usually small, e.g., only 168 MB in the most expensive Azure confidential computing instance [121]. Any memory usage beyond the EPC size will cause enclave pages to evict to the unprotected main memory. To ensure the confidentiality and integrity of the evicted EPC pages, SGX uses symmetric key cryptography which, unfortunately, compounds to a large overhead as the number of evictions increases. Such overhead can be mitigated by optimizing code to avoid paging as much as possible. Third, because system calls still need to be facilitated outside of enclaves, there is a substantial context switching overhead. State-of-the-art SGX systems tend to avoid system calls like IO and threading [39].

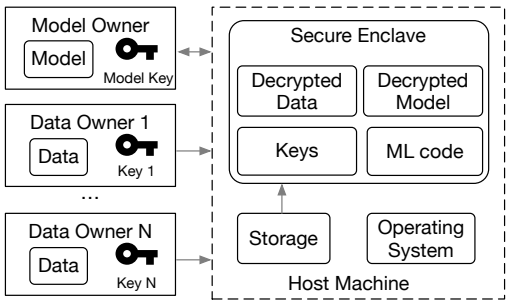


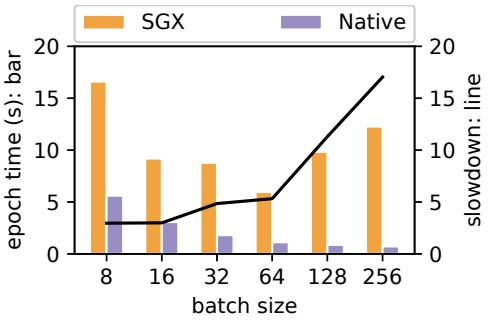
Figure 4.1: An illustration of a single-enclave solution that protects the confidentiality of both data and training model.

4.2.3 Private ML with a Single SGX Enclave

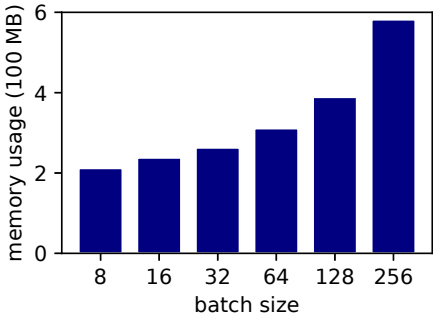
A simple solution for private collaborative ML is to use *a single SGX enclave* attested by both data owners and model owner. Fig. 4.1 illustrates such design.[1] The model owner and data owners hold their secrets – model and data – locally, while the training is performed in a remote enclave running on an untrusted host (e.g., a cloud server). Before the training begins, a data (or model) owner generates a private symmetric key and uses it to encrypt the data (or model). The encrypted data and model are then uploaded to an unprotected storage on the host. Note that, this is secure as the untrusted host has no access to the keys to decrypt the content. The host then creates an enclave containing the *agreed-upon ML code* by all secret (i.e., model and data) owners, and lets them initiate attestation to ensure the integrity and correctness of the initialized enclave. Once the attestation is passed successfully, each secret owner uploads its encryption key to the enclave over a TLS-protected channel, with which the enclave can retrieve the encrypted data and model from the storage and decrypt them. The training starts once the data, model, and ML code are all loaded into the enclave. When the training completes, the model owner downloads the trained model, and the enclave is destroyed along with the data it contains.

Poor Scalability. Such design, however, does not scale to a large training dataset. To illustrate this problem, we characterize its performance with Azure's latest confidential computing offering `DCsv2` [121]. We run experiments in a `Standard_DC8_v2` instance, the largest in `DCsv2` with 8 vCPUs and 32 GB memory, of which 168 MB is dedicated

[1] This design is an extension to [34], in which data owners also own the training model, similar to the FL setting.



(a) Training epoch time and slowdown w.r.t. batch size



(b) Memory usage w.r.t. batch size

Figure 4.2: The time needed to finish one epoch training running under SGX and native mode respectively. The slowdown shown in line represents the ratio between SGX time and native time. The memory figure depicts the amount of memory is used actively in SGX.

to an enclave's EPC. We train AlexNet [122] over images of size $32 \times 32 \times 3$ with TensorSCONE [123], an SGX-optimized version of TensorFlow v1.15. We then run the same training workload with the unmodified TensorFlow outside of the enclave. Note that, to speed up model training in a single machine without accelerators, a common technique is to configure a large batch size for increased parallelism and reduced iterations. We therefore evaluate the training epoch time (time needed to finish passing the entire dataset) with varying batch sizes in SGX and the native environment, respectively.

Fig 4.2a depicts the experimental results. When the batch size is small (8 or 16), running in SGX is only 2.9X slower than the native speed running outside of the enclave, meeting the expected performance of TensorSCONE [123]. Such slowdown is mainly due to the MEE encryption overhead but not EPC paging, as memory usage is barely over the EPC size (fig. 4.2b). Further enlarging the batch size leads to increased parallelism, which in turn reduces the epoch time in the native mode. This trend does not stand in SGX mode: as batch size increases, the epoch time first reduces but then surges rapidly, a consequence of frequent EPC paging due to excessive memory usage beyond the EPC size (fig. 4.2b). Therefore, one cannot expect to scale ML training by configuring a large batch size in an enclave. In fact, this can be 17X slower than running in the native mode (fig. 4.2a, batch size 256).

Exposing Training Logic. Note that in the single-enclave solution, the ML code must be shared and agreed by all data owners to ensure that it contains no malicious code that could harm their data (e.g., writing data to an external storage). However, this inevitably reveals the details of the model update logic, such as optimizer selection, learning rate scheduling, and gradient manipulation, which the model owner may not want to share as these are part of its intellectual property (§4.1.1).

4.2.4 Private ML with Multiple SGX Enclaves

As model training in a single enclave does not scale, recent work turns to a distributed solution with multiple enclaves. Notably, [35] augments FL with SGX enclaves hosted at the data owners' side for enhanced data privacy while taking advantage of data parallelism, but the data owners can still access the training model. Chiron [36], built atop Ryoan [120], ensures model confidentiality for ML-as-a-Service providers with SGX enclaves, and supports running multiple training enclaves in parallel. However, the design assumes that model owner (i.e., MLaaS provider) is not interested in harvesting data owners' data, which may not be the case in collaborative ML. SecureTF [37] presents a modified version of TensorFlow to support distributed training in multiple enclaves. However, it assumes that model and data belong to the same entity, and hence cannot be applied to collaborative ML. To our knowledge, a scalable solution for collaborative ML that can protect the privacy for both model and data owners is still missing.

4.3 Citadel Design

We aspire to devise a ML system that not only preserves data and model privacy simultaneously, but also enables distributed training across multiple SGX enclaves. To do so, we securely partition training workload, and make part of it replicable. Citadel achieves so by isolating *data handling codes* and *model handling ML codes*. The former can be shared with data owners to gain their trust, while the latter remains private to the model owner. After that, a *barrier* has to be inserted between the two parts to ensure the model handling codes cannot recover data owner's data with its private codes. With data handling codes isolated securely, Citadel can accelerate training through data parallelism.

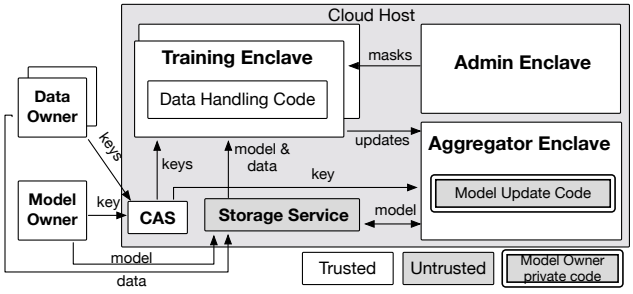


Figure 4.3: An architecture overview of Citadel. All codes (except the model update code) are open-sourced.

4.3.1 Design Overview

Fig. 4.3 illustrates an architecture overview of our system. Citadel facilitates collaborative ML in *multiple enclaves* hosted on untrusted infrastructure. These enclaves can run in a single or multiple cloud instances. A data (model) owner communicates with Citadel through a *client* running on a local machine. The client includes a *verifier* which the owner uses to attest Citadel. It also provides a *key manager* with which the owner generates a symmetric encryption key and uses it to encrypt data (model). The client uploads the encrypted data (model) to a general storage service in the cloud host. The storage itself does not need to be trusted since the secrets are encryption-protected. Citadel launches multiple enclaves on behalf of data and model owners, establishes trust between the enclaves and the secret owners via attestation (handled by CAS), and performs distributed training in those enclaves. Citadel runs three types of enclaves: *training enclaves*, *aggregator enclave*, and *admin enclave*.

Training Enclave. In Citadel, each data owner has a dedicated training enclave. Each training enclave needs to be attested by both the corresponding data owner(s) and the model owner to gain their trust. It takes private data as input and runs data handling codes (e.g., computing gradient updates), provided by the model owner, to generate model updates. As the code has direct access to the training data, it must be shared to and agreed by the data owner. The code should reveal no model information that the model owner wants to protect, such as hyper-parameter configurations and the training model. Instead, it loads such information as *environment variables* and *non-executable binary model files* from the storage service. Note that it is not possible to inject malicious code into the

model files, as they are non-executable in the standard ML toolchains. After the model updates are computed, the training enclave sends them to the aggregator enclave for a global aggregation. Citadel currently does not consider the scenario where models are too big for a single enclave. Such an issue could be addressed by either increasing EPC size with specialized SGX card [124], or applying existing model parallelism techniques to split large models [125, 126].

Aggregator Enclave. Citadel launches an aggregator enclave on behalf of the model owner to run the model handling code. Each training job has only one such enclave. It collects and aggregates updates from all training enclaves and utilizes them to update the training model. The updated model is encrypted and stored in storage service, so that the training enclaves can start the next training iteration by retrieving it. As the aggregator enclave has no access to data from data owners, the code running inside remains private to the model owner. This protects important training techniques developed by the model owner from being revealed such as learning rate schedule, optimizer selection, gradient selection and manipulation, which are all required in the update aggregation stage. The aggregator enclave is attested by the model owner only.

Admin Enclave. Citadel launches an admin enclave for a training job and uses it to schedule training workload and orchestrate the involved training and aggregator enclaves. Codes running inside an admin enclave (i.e., *mask generator* introduced in §4.3.2 and *enclave scheduler*) are open-sourced for public access. The enclave itself is attested by all model and data owners. To facilitate communication between an enclave and external entities, Citadel provides open-source utilities that run as *admin code* in a training or aggregator enclave. As the cloud host's network is untrusted, communications inside Citadel are secured by TLS connections with endpoints located inside the enclaves.

Attestation with CAS. In Citadel, a secret owner needs to attest multiple enclaves to ensure the integrity and confidentiality of the data and code. Using the default SGX attestation can be tedious as it is designed to attest one enclave at a time. CAS (configuration and attestation service) offers a simplified solution for secret management and attestation.

CAS itself is open-sourced and runs in an enclave, which the model and data owners can verify and attest. Once the secret owners have established trust over CAS, they can delegate their encryption keys to it, and instruct it on how to maintain their security. That is, which enclave can access what secrets, and what codes should run in which enclave. CAS honestly follows the specified security policy, attesting each enclave on behalf of the requested data or model owners and supplying it with secrets once it is trusted. With the help of CAS, model and data owners only have to initiate the attestation process once. Citadel employs PALÆMON [127], a trust management service built on top of SCONE [39], as its CAS system.

Fault Tolerance. Citadel's training enclaves are *stateless* by design, because model and data are all stored into and fetched from a storage system. In case of training enclave failures, Citadel can easily launch replacements and resume the training process via restarting the ongoing iteration. The training progress is always checkpointed since the updated model is stored into storage after each iteration. If admin or aggregator enclaves fail, we can also similarly restart the cluster and continue training.

4.3.2 Separating Data and Model Handling

A key design adopted by Citadel is to separate the model owner's ML code into two parts: *model update code* and *data handling code*. The model update code computes the global model updates based on the gradients received from the training enclaves. As such, it concerns with potentially confidential methods and values. Citadel runs the model update code in the aggregator enclave and ensures its confidentiality. In contrast, the data handling code is shared with the data owners (i.e., open-sourced) to gain their trust. It deals with standard forward and backward propagation and has direct access to the private data.

This separation provides the model owners with model confidentiality: Citadel ensures that the data owners only see *placeholders* for the model and hyperparameters, which are loaded dynamically into training enclaves after attestation (see the description of Training Enclave in §4.3.1). The secrets to load these values and replace the placeholders are only shared after the attestation, such that model and hyperparameters remain

unknown to data owners.

On the other hand, this separation alone does not fully provide data privacy for data owners. Although the data owners can verify the open-source data handling code and ensure it does not leak data purposely, prior work shows that a data owner's training data can still be inferred accurately from its computed gradients [18]. Citadel addresses this problem to protect data privacy from two aspects: First, data owners do not receive intermediate models from the model owner, such that they cannot pry into other data owners' data. Second, it establishes a *barrier* between the training enclaves and the aggregator enclave, so that the model owner only receives aggregated updates but not the raw updates from any *individual training enclave*. We propose two viable alternatives for such barrier: *zero-sum masking* and *hierarchical aggregation*.

Zero-Sum Masking. Zero-sum masking, originally proposed for federated learning (FL) as a way to implement Secure Aggregation [17], allows data owners to collectively generate masks and apply them to their individual updates before uploading them to the aggregator. The masks are generated in a way such that they are canceled out when summed up so that the aggregated updates are correctly recovered. Since the aggregator does not have access to individual masks, it cannot learn the raw gradients from any data owner.

Inspired by such solution, we propose a simpler zero-sum masking scheme for Citadel. Compared with the FL setting, where the masks have to be generated among distributed data owners, TEE enables us to execute codes that are verified and trusted by the concerning parties, so we can opt to a *centralized* mask generation approach. As shown in algorithm 4, N masks $m_0, m_1, ..., m_{N-1}$ are generated for N training enclaves by the admin enclave trusted by all secret owners. These masks have the same shape as the model gradients, while adding up to zero: $\sum_{i=0}^{N-1} m_i = 0$. The security of such approach is based on the fact that if data owners' values have uniformly random pairwise masks added to them, then the resulting values look uniformly random, conditioned on their sum being equal to the sum of data owner's values (see the security proof in [17]).

After the training starts, each training enclave first downloads and decrypts a fresh model from the storage service, and then computes gradients with the codes shared and verified by data owners. The training enclave i then requests admin enclave for a mask

m_i, applies it to its gradients, and finally sends them to the aggregator enclave. The aggregator enclave collects the masked gradients from all training enclaves, accumulates them and updates the model with them using a certain model update method. As individual update from each training enclave is obscured with a random mask, the model owner's private codes cannot infer any information about the training data from it. By accumulating all the updates together, the inlined masks cancel each other out, resulting in the same aggregated update as it would have been without masking. Our centralized zero-sum masking approach protects the model confidentiality, while guaranteeing the same level of privacy for data owners as Secure Aggregation [17] without the time-consuming synchronous distributed mask generation protocols.

Hierarchical Aggregation. The zero-sum masking solution requires *one-to-all* and *all-to-one* synchronizations in the mask distribution (between the admin enclave and training enclaves) and update aggregation (between training enclaves and the aggregator enclave) phases. As more training enclaves run in the system, such synchronization overheads become increasingly prominent. In fact, given SGX's memory limitations, neither generating and holding a large amount of masks within an admin enclave nor aggregating large updates within an aggregator enclave scales.

To avoid such all-to-one synchronization, we propose to establish a *tree-structured hierarchical aggregation* among training enclaves. Since our goal is to protect individual updates from being learned by the aggregator enclave, we can utilize training enclaves to aggregate the intermediate results, which are trusted by data owners. As described in algorithm 5, after processing a batch, each training enclave holds its own gradients and follows a tree-structured hierarchical aggregation scheme to accumulate gradients. Assume there are N training enclaves, and each *leader* in the aggregation tree has C children ($C-1$ neighbors have to transfer their updates to the leader in one round). It requires $\lceil \log_C N \rceil + 1$ rounds of aggregation (height of the aggregation tree), and on the l^{th} level, there are $\lceil ... \lceil \lceil N/C \rceil / C \rceil ... \rceil$ active nodes remaining. On the l^{th} level, we denote the i^{th} remaining active node by id_i^l, so that each of these active nodes has to send its aggregated results from last round to a leader node $id_{\lfloor N/C \rfloor}^l$. The recursion continues until the last leader accumulates the final results and sends it to the aggregator enclave. Hierarchical

Algorithm 4 Citadel with Zero-Sum Masking

Training Enclave i:

function STARTSTRAINING
  for epoch $e = 0, 1, 2, ..., E$ **do**
    for all training batch $t = 0, 1, 2, \cdots, T$ **do**
      Download and decrypt fresh model $model$
      Compute gradients $g_i^{(e,t)}$
      Request mask $m_i^{(e,t)}$ GETMASK(i) from admin enclave
      Apply mask to gradients $G_i^{(e,t)} = g_i^{(e,t)} + m_i^{(e,t)}$
      Send update $G_i^{(e,t)}$ to aggregator enclave by calling UPLOADUPDATE($G_i^{(e,t)}$)

Admin Enclave:

function GENERATEMASK(N)
  Initialize sum $sum = 0$
  Initialize $masks = []$
  for $i = 0, 1, 2, ..., N - 2$ **do**
    Generate random mask new_mask
    Append new_mask to $masks$
    $sum += new\_mask$
  Append $-sum$ to $masks$
function GETMASK(i)
  if $i == N - 1$ **then**
    Call GENERATEMASK(N) asynchronously for next iteration.
  return $masks[i]$

Aggregator Enclave:

function UPLOADUPDATE($G_i^{(e,t)}$)
  Record $updates[i] = G_i^{(e,t)}$
function STARTSTRAINING
  Download and decrypt fresh model $model$
  for epoch $e = 0, 1, 2, ..., E$ **do**
    for all training batch $t = 0, 1, 2, \cdots, T$ **do**
      Wait for updates from all trianing enclaves
      Summarize all updates $G^{(e,t)} = SUM(updates)$
      Apply aggregated gradients $G^{(e,t)}$ to model $model$
      Upload $model$ to storage service

aggregation avoids the expensive all-to-one synchronization, eliminating communication hotspots.

Comparison of Two Approaches. Both zero-sum masking and hierarchical aggregation effectively shield the raw updates of individual training enclaves from the aggregator enclave. Zero-sum masking requires an all-to-one communication from all training enclaves, and then all the updates have to be aggregated in the EPC-limited aggregator enclave, so the overall overhead grows as more training enclaves run in the system.

Algorithm 5 Citadel with Hierarchical Aggregation

Training Enclave i:

function STARTSTRAINING
  for epoch $e = 0, 1, 2, ..., E$ **do**
    for all training batch $t = 0, 1, 2, \cdots, T$ **do**
      Download fresh model $model$
      Compute gradients $g_i^{(e,t)}$
      Start hierarchical aggregation calling RECURSIVEAGGREGATE
function RECURSIVEAGGREGATE
  if I am leader in this round. **then**
    Collect and accumulate intermediate results.
    if I am the final leader. **then**
      Send aggregated result $G^{(e,t)}$ to aggregator.
    else
      Send the intermediate results to the next leader by calling its RECURSIVEAGGREGATE.
  else
    Send the intermediate results to the next leader by calling its RECURSIVEAGGREGATE.

Aggregator Enclave:

function STARTSTRAINING
  Download and decrypt fresh model $model$
  for epoch $e = 0, 1, 2, ..., E$ **do**
    for all training batch $t = 0, 1, 2, \cdots, T$ **do**
      Wait for final update $G^{(e,t)}$ from the last leading training enclave
      Apply aggregated gradients $G^{(e,t)}$ to model $model$
      Upload $model$ to storage service

Hierarchical aggregation, on the other hand, breaks the all-to-one communication pattern into a hierarchical aggregation tree. Although the potential network congestion is mitigated, extra cryptographic operations are needed to protect the communication connections on the aggregation tree. However, it is difficult, if not impossible, to quantitatively justify such trade-off in this scenario, as the time needed to finish a certain operation within an enclave depends on both the memory footprint and the memory access pattern. Assume there are N training enclaves. Let $t_{net}(x)$, $t_{enc}(x)$, and $t_{dec}(x)$ respectively denote the time needed to transfer, encrypt, and decrypt message x. Let t_{mask}, t_{train}, and t_{apply} be the computation time needed to apply a mask, generate gradients, and apply gradients to model, respectively, and $t_{agg}(k)$ the time spent on aggregating updates from k training enclaves. The iteration time for zero-sum masking $t_{mask}(N)$ is estimated as

$$\begin{aligned} t_{mask}(N) = t_{train} + t_{net}(m) + t_{dec}(m) + t_{mask} + t_{enc}(g) \\ + t_{net}(g) + t_{dec}(g) + t_{agg}(N) + t_{apply}, \end{aligned}$$

where m and g stand for a set of mask and gradients, respectively. Assuming each node

in the aggregation tree has C children, the iteration time for hierarchical aggregation $t_{tree}(N, C)$ is estimated as

$$t_{tree}(N, C) = (t_{enc}(g) + t_{dec}(g) + t_{agg}(C) + t_{net}(g)) \times (\lceil \log_C N \rceil + 1) + t_{train} + t_{apply}.$$

As a general guideline, zero-sum masking tends to work better on *smaller models* with *fewer* training enclaves, as the memory footprint within the aggregator is smaller and network congestion is less likely. When there is a large number of training enclaves, hierarchical aggregation becomes more favorable. We will evaluate the two approaches in §4.5.

4.4 Implementation

In this section, we describe the implementation details of Citadel. We base our implementation on SCONE [39], but it can also be extended to other SGX-enabling frameworks such as Graphene [128] and Ryoan [120]. We use MongoDB [129] as the storage service, which can be replaced by any generic object store or cloud storage system. We containerize all system components and orchestrate them in Kubernetes [130]. Our implementation consists of 5,000 lines of Python code and Linux Shell script, and is open-sourced.

Trusted Computing Base (TCB). For an easy support of SGX and multiple enclave orchestration, we adopt SCONE [39] in our system. SCONE provides SGX-protected Linux containers, so that we can utilize tools like Docker [131] and Kubernetes [130] to orchestrate enclaves.

Efficient Encryption & Decryption. As the host infrastructure is not trusted, encrypted data and models must be decrypted within the enclaves, and network connections between enclaves are also secured with TLS. This results in substantial cryptographic operations performed inside an enclave. Especially during the aggregation process, a single enclave has to decrypt results from multiple enclaves and add them up. Therefore, the efficiency of cryptographic inside an enclave plays an important role in overall performance of Citadel.

In a native setting without SGX, one way to increase performance is to increase the parallelism with multi-processing or multi-threading. However, inside an SGX enclave, each process runs inside its own enclave, so launching new processes is extremely slow as it requires to set up new enclaves and initializes EPC pages. Furthermore, the new sub-process enclaves contend with the parent enclave for EPC, resulting in performance degradation for all of them. Our experimental evaluations with the OpenSSL implementation of `AES-256-CBC` shows that, encrypting and decrypting 16 AlexNet [97] models with multi-processing enabled is at least 2X slower than processing them serially with SGX enabled. On the other hand, SCONE [39] provides efficient *user-level threading* to avoid costly system calls, so it is possible for us to improve cryptographic operations with multi-threading. However, due to Python's Global Interpreter Lock (GIL) [132], only one python thread can run at any given time even with multi-threading. To overcome such hurdle, we implement our cryptographic operations in C++ and compile it with CFFI [133]. This not only allows us to bypass the GIL limitation, but also enables the highly efficient performance of native codes.

Offline Mask Generation. In our zero-sum masking approach, protecting the mask confidentiality is the key to shielding individual updates from model owner and cloud provider. Therefore, the masks have to be generated within the admin enclave and encrypted before leaving it. However, when the number of training enclaves increases, the compute-intensive nature of mask generation and distribution would inevitably make admin enclave a performance bottleneck.

To address this problem, we choose to *generate masks offline* and *offload mask distribution to the untrusted storage service*. Before the training starts, the admin enclave generates sufficient sets of N masks and stores them in the storage service encrypted. During training, upon receiving a masking request from a training enclave, the admin enclave redirects the request to the storage service, and provides the training enclave with a corresponding decryption key. This design removes the heavy-lifting tasks off the critical path. In case of training enclave stragglers, Citadel may choose to employ relaxed consistency like SSP [24]: assuming the first k out of N training enclaves would participate in the aggregation, admin enclave can return the sum of remaining pre-generated masks $\sum_{i=k}^{N-1} m_i$ to

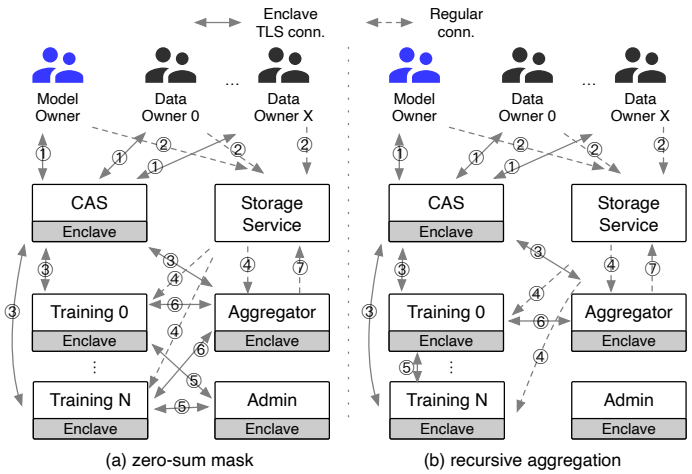


Figure 4.4: The workflow of Citadel with zero-sum masking, enclave TLS connections terminate within enclaves.

enclave $k-1$ as its mask, ensuring the overall sum remains zero.

Citadel Workflow. Putting it all together, we elaborate on the workflow of Citadel with the two aggregation approaches to protect model and data privacy simultaneously. The workflow of zero-sum masking is depicted in fig. 4.4a. ① The model and data owners attest the CAS, and share their encryption keys to CAS. ② The model and data owners upload their encrypted secrets to the storage service. ③ CAS attests training and admin enclaves on behalf of data owners, then shares the corresponding data encryption keys to training enclaves; CAS also attests training, admin, and aggregator enclaves on behalf of model owner, then shares the model encryption key to training and aggregator enclaves. ④ Training enclaves fetch corresponding data and model, decrypt them and compute gradients based on their own data; aggregator downloads and decrypts model. ⑤ Training enclaves ask admin enclave for masks and have the requests redirected to the storage service with mask decryption keys. ⑥ Training enclaves fetch masks from storage service, apply them to their updates, and send them over to aggregator. ⑦ Aggregator collects the masked updates, summarizes them, and updates the global model. The model is then encrypted and uploaded to the storage service. This completes one training iteration.

Similarly, fig. 4.4b depicts the workflow of hierarchical aggregation, where Steps ①-④ are the same as the masking approach. ⑤ Training enclaves recursively aggregate all the updates until the final sum of all updates is available at training enclave 0. ⑥ The

aggregator enclave receives the final update and applies it to the model. ⑦ The model is then encrypted and uploaded to the storage service, completing the current training iteration.

4.5 Evaluation

In this section, we evaluate the performance of Citadel with representative ML models trained on a public cloud. We first examine the scalability of Citadel with zero-sum masking in clusters of various sizes. We then evaluate hierarchical aggregation with different configurations to quantify how avoiding all-to-one communication can help improve system scalability. Finally, we assess the system overhead of our design by comparing Citadel with two baselines, i.e., the single-enclave baseline and native Citadel without SGX baseline.

4.5.1 Methodology

Settings. We consider a distributed ML setting where all instances are located within the same cluster. There is no need to perform geo-distributed training for privacy preservation, because all secrets uploaded to Citadel are encrypted and protected by Citadel. We conduct all experiments on Azure confidential computing instances with SGX support in `Canada Central` region. The instance type we chose is `Standard_DC4s_v2`, which has 4 vCPUs, 16 GB of memory, and 112 MB of EPC memory. We deploy exactly one enclave on each instance to avoid EPC contention. The scale of our evaluation is limited to 34 such instances (including all training, aggregator and admin enclaves), because Azure limits the total number of `DCsv2` family vCPUs that can be rented by a non-enterprise user. Nevertheless, we believe the trend demonstrated in our evaluation applies to a larger scale, and is sufficient to validate our implementation.

Benchmarking Models. We have implemented four ML models with their respective workloads and privacy requirements, using TensorFlow v1.15. The first two, AlexNetS and AlexNetL, belong to the same application where a certain number of hospitals collaborate with a medical tech company to train a diabetes diagnosis model based on Retinopa-

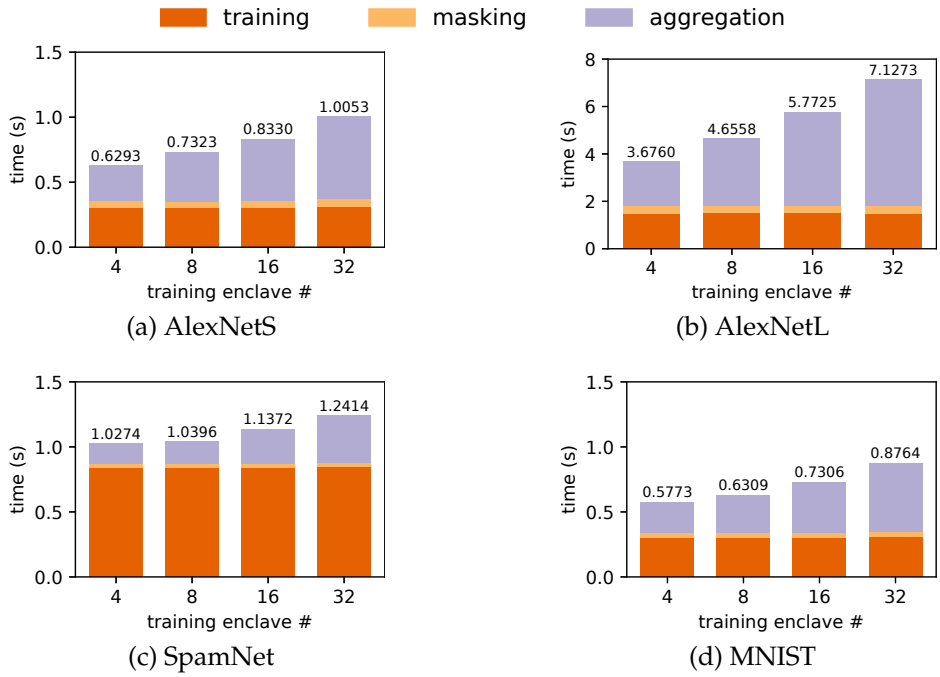
thy images [134]. The input images are scaled to $32 \times 32 \times 3$ for AlexNetS and $96 \times 96 \times 3$ for AlexNetL, to represent a smaller and a larger model, respectively. AlexNetS has 1.25M trainable parameters while AlexNetL has 15.9M trainable parameters. The third one SpamNet is a spam filtering model utilizing LSTM [98] network with 9.6K trainable parameters, where we use SMS messages [135] as input data. Here, the model is required to be private and the personal SMS messages are sensitive. The last one MNIST is a 12-layer CNN handwriting recognition model trained with MNIST dataset [136]. MNIST model has 887.5K trainable parameters. The model owner wants to protect its intellectual property, while data owners want to remain anonymous because the adversary may want to forge their handwriting. The aforementioned four workloads are backed by deep learning models of various sizes and can cover diverse types of tasks. To emulate multiple data owners, we randomly partition these datasets into multiple shards and encrypt them with different keys before uploading them into the storage service in Citadel.

Baselines. We use two baselines for comparison. The first one is the privacy-preserving `single-enclave` approach described in §4.2.3, and the second one is `native-distributed`, where we run Citadel natively without SGX. We will demonstrate how Citadel can provide strong privacy and confidentiality while still achieving high processing throughput.

4.5.2 Effectiveness of Zero-Sum Masking

We first evaluate the effectiveness of Citadel's zero-sum masking technique with the four workloads outlined before. We report the time breakdown of one training iteration, and present the results in fig. 4.5a-fig. 4.5d. The iteration time is measured as the timespan from downloading fresh models in training enclaves until the aggregator enclave uploads the updated model. Specifically, the `training` portion refers to the time spent inside training enclaves, but excludes mask-related operations and the time to transmit masked updates to aggregator. The `masking` portion includes the time spent on requesting, downloading, and applying the masks. The `aggregation` portion covers the time spent in the aggregator enclave, plus the time used to transmit all masked updates. All results are averaged across all enclaves over multiple iterations.

As we can see, Citadel with zero-sum masking scales well with the increase of training



(a) AlexNetS (b) AlexNetL

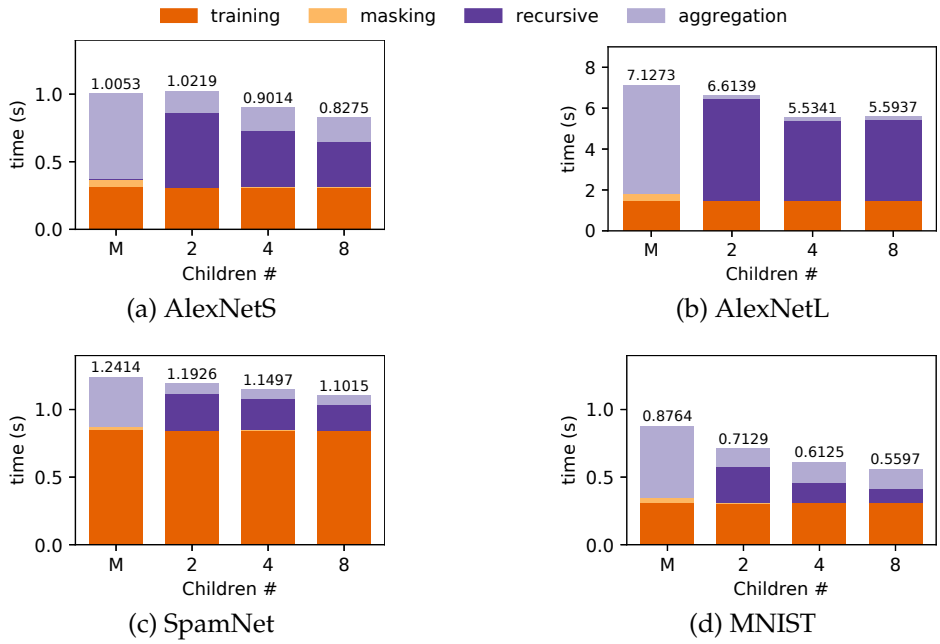
(c) SpamNet (d) MNIST

Figure 4.5: The iteration time breakdown w.r.t. training enclave numbers when the zero-sum masking is adopted.

enclaves. Octupling training enclaves from 4 to 32, the overall iteration time only increases by 59.7% for AlexNetS, 93.9% for AlexNetL, 20.8% for SpamNet, and 53.5% for MNIST. Looking into each portion separately, and we can see: 1) the training time stays constant when Citadel scales out as the training operations are irrelevant to cluster size, 2) the masking time also stays constant because of the offline mask generation described in §4.4, and 3) the aggregation time increases (inevitably) because aggregation involves the all-to-one communication and the summing-up of all masked gradient updates. Altogether, these results show only a modest increase of Citadel's iteration time with the increase of cluster size. This in turn indicates that Citadel can accommodate a large number of data owners and complex models with reasonable performance overhead.

4.5.3 Effectiveness of Hierarchical Aggregation

Although §4.5.2 exhibits that Citadel with zero-sum masking can effectively increase throughput by adding more training enclaves, we also notice the increasingly significant aggregation overhead with the increase of cluster size. In this subsection, we evaluate Citadel's hierarchical aggregation approach, and validate if it can further reduce the aggregation



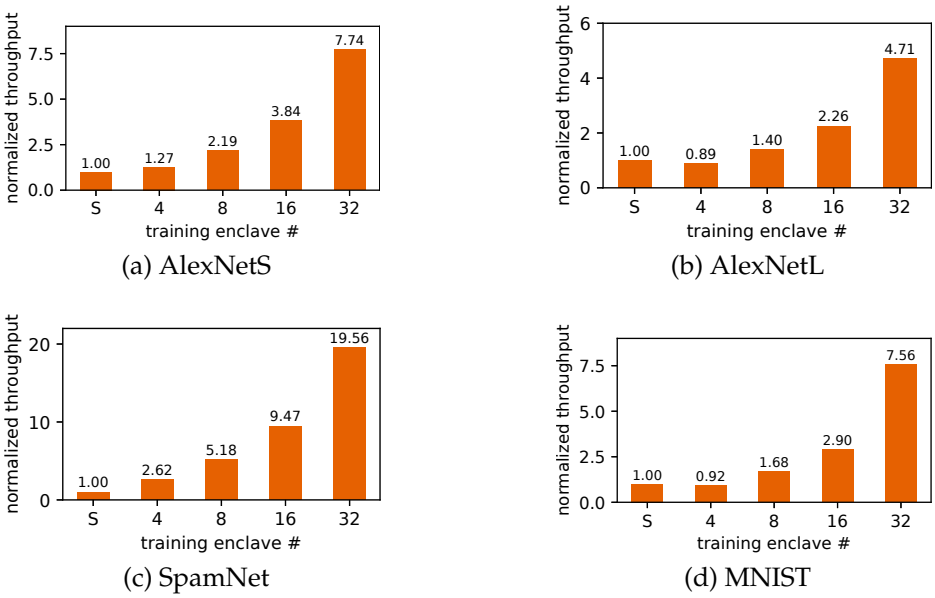
(a) AlexNetS (b) AlexNetL

(c) SpamNet (d) MNIST

Figure 4.6: The iteration time breakdown of different models w.r.t. aggregation children number when hierarchical aggregation is adopted. All experiments are run with 32 training enclaves. The zero-sum mask results with 32 training enclaves are shown as M bars for reference.

overhead. The results are shown in fig. 4.6a-fig. 4.6d. We target the scenario with 32 training enclaves which is the largest cluster we are able to run in Azure. We test the hierarchical aggregation approach with its aggregation tree children set to 2, 4 and 8, and use zero-sum masking approach for reference. The `recursive` portion in the breakdown refers to the timespan from when the first training enclave update is ready until the final aggregate result is ready.

As we can see in the results, hierarchical aggregation is indeed more efficient at scale compared with zero-sum masking. The overall iteration time is reduced by 17.7% for AlexNetS, 21.5% for AlexNetL, 11.3% for SpamNet, and 36.1% for MNIST. With AlexNetL, Citadel performs the best when the number of children is set to 4. When we reduce it to 2, although the computational overhead at each aggregation level decreases, the overall gain is offset by the increased aggregation depth; while if we increase it to 8, we face large EPC overhead at each aggregation step as 8 updates have to reside in the memory simultaneously. In conclusion, the children number is a tradeoff knob for aggregation performance and there is no one-size-fits-all optimal value across all models. We are unable to extend our evaluation to more training enclaves, but we believe hierarchical aggrega-



(a) AlexNetS (b) AlexNetL

(c) SpamNet (d) MNIST

Figure 4.7: The total throughput normalized with the `single-enclave` solution throughput (labeled as S) w.r.t. training enclave number.

tion can achieve better performance when Citadel scales out further, thus addressing the bottleneck in zero-sum masking approach.

4.5.4 Citadel vs. Single Enclave

The single-enclave solution described in §4.2.3 can achieve data privacy and a limited protection of model confidentiality. In this subsection, we compare Citadel with this single-enclave solution, and show that Citadel outperforms it in both privacy guarantees and performance. Specifically, we profile the single-enclave solution's throughput via training the same models on the same Azure instance. The results are shown in fig. 4.7a-fig. 4.7d. Note that, we show the better results between the zero-sum masking and hierarchical aggregation approaches.

Going from a single-enclave solution to a distributed system across multiple servers, Citadel introduces secured connections that require both network communication and cryptographic operations. As a result, we see marginal improvements compared with `single-enclave` when using only 4 training enclaves. However, with more training enclaves, Citadel is able to further improve its training throughput substantially. Note that, the benefit of distributed training becomes more prominent for ML models where

Table 4.1: The slowdowns of Citadel at different scale. The 32-R column shows the hierarchical aggregation implementation, while the rest show zero-sum masking.

# Train. Enclave	4	8	16	32	32-R
AlexNetS	1.22	1.18	1.23	1.24	1.40
AlexNetL	1.09	1.23	1.44	1.73	1.65
SpamNet	1.21	1.21	1.22	1.19	1.26
MNIST	1.15	1.15	1.14	1.15	1.17

training takes up more total time (e.g., SpamNet in fig. 4.5c). Also note that, in our experiment, we aggregate model updates after each iteration, and therefore, the result here demonstrates the lower bound of our throughput improvement. One can easily improve the training performance via communicating after every few iterations, a.k.a., local update SGD [69, 70, 71]. With the help of such techniques, Citadel's throughput could be further improved.

4.5.5 SGX Overhead in Citadel

Finally, we compare Citadel against running at the native speed. To do that, we repeat our evaluation on Citadel with the four workloads outside of SGX enclaves. All the experiments are conducted on the same Azure Kubernetes cluster but with native docker containers running the same code as in §4.5.2-§4.5.3. We seek to show how much of the total overhead is brought by SGX in the entire workflow. We run the experiments over multiple iterations and compile the results in table 4.1. Table 4.1 shows Citadel's slowdown with different numbers of training enclaves. The slowdown ranges from 1.18× to 1.40× for AlexNetS, 1.09× to 1.73× for AlexNetL, 1.19× to 1.26× for SpamNet, and 1.14× to 1.17× for MNIST, In particular, the slowdown is defined as the performance ratio between Citadel running with and without SGX under the same configuration. We can conclude that SGX results in 15%–73% performance slowdown, and the actual slowdown varies for different models and scales. With larger models like AlexNetL, memory consumption is higher, so the EPC paging happens more often, causing higher overhead. We also notice that, the more training enclaves there are, the more memory it needs to finish aggregation, thus a generally higher slowdown at larger scale.

4.6 Discussion

In this section, we discuss some future directions of Citadel moving forward.

Fault Tolerance. Citadel's training enclaves are stateless by nature, because model and data are all stored into and fetched from a storage system. In case of training enclave failures, Citadel can easily launch replacements and resume the training process via restarting the ongoing iteration. The training progress is always checkpointed since the updated model is stored into storage after each iteration. If admin or aggregator enclaves fail, we can also similarly restart the cluster and continue training.

Large Models. Citadel's current design does not consider the scenario where models are too big for a single enclave. Such an issue can be addressed by either increasing EPC size with specialized SGX card [124], or applying existing model parallelism techniques to split large models [125, 126].

4.7 Summary

In this paper, we have presented Citadel, the first scalable system for collaborative machine learning that protects both data privacy and model confidentiality with SGX. Citadel partitions the training workload into two parts, the open-sourced data handling codes running in training enclaves and the private model update codes running in the aggregator enclave. Citadel further imposes a barrier between the two parts by means of zero-sum masking and hierarchical aggregation to prevent data and model leakage. Experimental results show that Citadel scales to a large number of enclaves at the expense of a small performance overhead due to SGX.

CHAPTER 5

CONCLUSIONS AND FUTURE DIRECTIONS

In this dissertation, we comprehensively review existing large-scale ML training training, and seek to identify underlying performance and privacy issues, propose and validate novel solutions for efficient and privacy-preserving ML systems.

We first targeted to improve efficiency in asynchronous ML training by leveraging the trade-off between update rates and update quality. Unlike conventional asynchronous training that passively bounds inconsistency among workers, our proposed SpecSync makes each worker speculate about others' updates, and actively pull fresh parameters in case it is convinced that freshness gain out-weights the computation loss. Moreover, we designed an heuristic online hyperparameter tuning algorithm to judiciously determine the re-synchronization behavior. SpecSync effectively sped up distributed training without sacrificing accuracy.

Second, we scrutinized the current state of utilizing HE to conduct secure cross-silo FL. We showed that HE's computation overhead and large ciphertexts make it impractical for state-of-the-art model training. To address such concern, we proposed BatchCrypt, a system that performs quantization, encoding, and batching on gradients to drastically reduce the encryption overhead.

Third, we investigated collaborative ML training between model providing IT firms and data owners, and concluded that besides data privacy for data owners, model supplier also has the demand to protect its model design and weights from other participants. Driven by the lack of scalable solution for such scenario, we devised Citadel, a distributed system that utilizes multiple SGX enclaves to establish training environment trusted by all participating parties.

5.1 Future Directions

Our work on large-scale ML training systems is far from solving all problems. Large-scale ML training is undergoing fundamental shifts, we expect future ML systems to have 1) finer-grained communication management, 2) AI-powered resource planning.

First, with ML accelerator like TPUs [137], the bottleneck of distributed ML is moving from computation to communication. This move is more extreme in scenarios like cross-device FL, which provide only unstable connection with very little bandwidth. As a result, more and more efforts are put into dividing communication into smaller chunks and overlapping them with computation [70, 125]. While others have explored leveraging ML's error tolerance nature to reduce the communication size [82, 83, 84, 85]. With the growing demand for communication-efficient ML training systems, we can combine pipelining, compression, and fine-grained scheduling to improve ML training productivity.

Second, both ML training and serving require large amount of computing resource. On public clouds, the overwhelmingly large configuration space makes it challenging to estimate and choose the right cloud configuration. Recent publications explore the problem with AI-powered approaches including model fitting [138] and Bayesian optimization [138]. However, these systems are still far from being fully-automated, advanced profiling and domain knowledge is still required to facilitate resource configuration. With a combination of AI technologies, we will be able to provide an one-stop solution for resource estimation, resource allocation, and task scheduling.

REFERENCES

[1] D. Talbot, "How Microsoft Cortana improves upon Siri and Google Now," http://www.tomshardware.com/news/microsoft-cortana-unique-features,26506.html.

[2] C. Rosenberg, "Improving photo search: A step across the semantic gap," http://googleresearch.blogspot.com/2013/06/improving-photo-search-step-across.html, accessed: 2015-11-20.

[3] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "A picture is worth a thousand (coherent) words: building a natural description of images," http://googleresearch.blogspot.com/2014/11/a-picture-is-worth-thousand-coherent.html, accessed: 2015-11-20.

[4] F. Nelson, "Nvidia demos a car computer trained with deep learning," http://www.technologyreview.com/news/533936/, 2015.

[5] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *NIPS*, 2012.

[6] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter serve." in *USENIX OSDI*, 2014.

[7] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[8] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an efficient and scalable deep learning training system," in *USENIX OSDI*, 2014.

[9] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: A system for large-scale machine learning," in *USENIX OSDI*, 2016.

[10] "Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)," https://eur-lex.europa.eu/eli/reg/2016/679/oj, 2016.

[11] "California Consumer Privacy Act (CCPA)," https://oag.ca.gov/privacy/ccpa, 2018.

[12] "Cybersecurity Law of the People's Republic of China," http://www.lawinfochina.com/display.aspx?id=22826&lib=law, 2017.

[13] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, no. 2, p. 12, 2019.

[14] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. Bonawitz, Z. Charles, G. Cormode, R. Cummings *et al.*, "Advances and open problems in federated learning," *arXiv preprint arXiv:1912.04977*, 2019.

[15] M. Pathak, S. Rane, and B. Raj, "Multiparty differential privacy via aggregation of locally trained classifiers," in *NeurIPS*, 2010.

[16] R. Shokri and V. Shmatikov, "Privacy-preserving deep learning," in *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. ACM, 2015, pp. 1310–1321.

[17] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1175–1191.

[18] L. T. Phong, Y. Aono, T. Hayashi, L. Wang, and S. Moriai, "Privacy-preserving deep learning via additively homomorphic encryption," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 5, pp. 1333–1345, 2018.

[19] C. Liu, S. Chakraborty, and D. Verma, "Secure model fusion for distributed learning using partial homomorphic encryption," in *Policy-Based Autonomic Data Governance*. Springer, 2019, pp. 154–179.

[20] "IBM Watson Health: Diagnostic Imaging Solutions," https://www.ibm.com/watson-health/solutions/diagnostic-imaging, 2020.

[21] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction apis," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 601–618.

[22] T. Orekondy, B. Schiele, and M. Fritz, "Knockoff nets: Stealing functionality of black-box models," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4954–4963.

[23] Q. Jia, L. Guo, Z. Jin, and Y. Fang, "Preserving model privacy for machine learning in distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 8, pp. 1808–1822, 2018.

[24] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ML via a stale synchronous parallel parameter server," in *NIPS*, 2013.

[25] J. Langford, A. J. Smola, and M. Zinkevich, "Slow learners are fast," in *NIPS*, 2009.

[26] J. Jiang, B. Cui, C. Zhang, and L. Yu, "Heterogeneity-aware distributed parameter servers," in *ACM SIGMOD*, 2017.

[27] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons *et al.*, "Exploiting bounded staleness to speed up big data analytics." in *USENIX ATC*, 2014.

[28] "WeBank," https://www.webank.com/en/, 2019.

[29] I. San, N. At, I. Yakut, and H. Polat, "Efficient paillier cryptoprocessor for privacy-preserving data mining," *Security and communication networks*, vol. 9, no. 11, pp. 1535–1546, 2016.

[30] H. B. McMahan, E. Moore, D. Ramage, and B. A. y Arcas, "Federated learning of deep networks using model averaging," *ArXiv*, vol. abs/1602.05629, 2016.

[31] O. Gupta and R. Raskar, "Distributed learning of deep neural network over multiple agents," *Journal of Network and Computer Applications*, vol. 116, pp. 1–8, 2018.

[32] P. Vepakomma, O. Gupta, T. Swedish, and R. Raskar, "Split learning for health: Distributed deep learning without sharing raw patient data," *arXiv preprint arXiv:1812.00564*, 2018.

[33] "Intel SGX," https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html, 2020.

[34] N. Hynes, R. Cheng, and D. Song, "Efficient deep learning on multi-source private data," *arXiv preprint arXiv:1807.06689*, 2018.

[35] "Ping An: Security Technology Reduces Data Silos," https://www.intel.com/content/www/us/en/customer-spotlight/stories/ping-an-sgx-customer-story.html, 2020.

[36] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel, "Chiron: Privacy-preserving machine learning as a service," *arXiv preprint arXiv:1803.05961*, 2018.

[37] D. L. Quoc, F. Gregor, S. Arnautov, R. Kunkel, P. Bhatotia, and C. Fetzer, "securetf: A secure tensorflow framework," in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 44–59.

[38] R. Banner, Y. Nahshan, and D. Soudry, "Post training 4-bit quantization of convolutional networks for rapid-deployment," in *NeurIPS*, 2019.

[39] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell *et al.*, "{SCONE}: Secure linux containers with intel {SGX}," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 689–703.

[40] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *NIPS*, 2010.

[41] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *USENIX OSDI*, 2004.

[42] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *USENIX NSDI*, 2012.

[43] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "MLlib: Machine learning in apache spark," *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 1235–1241, 2016.

[44] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *ACM GRADES*, 2013, p. 2.

[45] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. P. Xing, "Solving the straggler problem with bounded staleness." in *ACM HotOS*, 2013.

[46] M. Li. Does MXNet support stale synchronous parallel (aka. SSP)? [Online]. Available: https://github.com/dmlc/mxnet/issues/841

[47] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[48] Apache MXNet, "https://mxnet.incubator.apache.org."

[49] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM Trans. Interactive Intelligent Sys.*, vol. 5, no. 4, p. 19, 2016.

[50] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *USENIX NSDI*, 2017.

[51] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE CVPR*, 2016.

[52] T. Schaul, S. Zhang, and Y. LeCun, "No more pesky learning rates," in *ICML*, 2013, pp. 343–351.

[53] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *IEEE CVPR*, 2009, pp. 248–255.

[54] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Managed communication and consistency for fast data-parallel iterative analytics," in *ACM SoCC*, 2015.

[55] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Addressing the straggler problem for iterative convergent parallel ml." in *ACM SoCC*, 2016.

[56] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-SGD for distributed deep learning," in *IJCAI*, 2016.

[57] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous sgd," *arXiv preprint arXiv:1604.00981*, 2017.

[58] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *ACM EuroSys*, 2010.

[59] H. B. McMahan, E. Moore, D. Ramage, S. Hampson *et al.*, "Communication-efficient learning of deep networks from decentralized data," *arXiv preprint arXiv:1602.05629*, 2016.

[60] J. Konečnỳ, H. B. McMahan, D. Ramage, and P. Richtárik, "Federated optimization: Distributed machine learning for on-device intelligence," *arXiv preprint arXiv:1610.02527*, 2016.

[61] W. Du, Y. S. Han, and S. Chen, "Privacy-preserving multivariate statistical analysis: Linear regression and classification," in *Proceedings of the 2004 SIAM international conference on data mining*. SIAM, 2004, pp. 222–233.

[62] P. Mohassel and Y. Zhang, "Secureml: A system for scalable privacy-preserving machine learning," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 19–38.

[63] P. Mohassel and P. Rindal, "Aby 3: a mixed protocol framework for machine learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 35–52.

[64] Y. Liu, T. Chen, and Q. Yang, "Secure federated transfer learning," *arXiv preprint arXiv:1812.03337*, 2018.

[65] K. Cheng, T. Fan, Y. Jin, Y. Liu, T. Chen, and Q. Yang, "Secureboost: A lossless federated learning framework," *arXiv preprint arXiv:1901.08755*, 2019.

[66] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1999, pp. 223–238.

[67] X. Lian, Y. Huang, Y. Li, and J. Liu, "Asynchronous parallel stochastic gradient for nonconvex optimization," in *NIPS*, 2015.

[68] C. Zhang, H. Tian, W. Wang, and F. Yan, "Stay fresh: Speculative synchronization for fast distributed machine learning," in *ICDCS*. IEEE, 2018.

[69] T. Lin, S. U. Stich, K. K. Patel, and M. Jaggi, "Don't use large mini-batches, use local sgd," *arXiv preprint arXiv:1808.07217*, 2018.

[70] J. Wang and G. Joshi, "Adaptive communication strategies to achieve the best error-runtime trade-off in local-update sgd," *arXiv preprint arXiv:1810.08313*, 2018.

[71] F. Haddadpour, M. M. Kamani, M. Mahdavi, and V. Cadambe, "Local sgd with periodic averaging: Tighter analysis and adaptive synchronization," in *NeurIPS*, 2019.

[72] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[73] "FATE (Federated AI Technology Enabler)," https://github.com/FederatedAI/FATE, 2019.

[74] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Recommendation for key management part 1: General (revision 3)," *NIST special publication*, vol. 800, no. 57, pp. 1–147, 2012.

[75] C. Data61, "Python paillier library," https://github.com/data61/python-paillier, 2013.

[76] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, "The marginal value of adaptive gradient methods in machine learning," in *NeurIPS*, 2017, pp. 4148–4158.

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

[78] "Microsoft SEAL (release 3.5)," https://github.com/Microsoft/SEAL, Apr. 2020, microsoft Research, Redmond, WA.

[79] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

[80] T. Ge and S. Zdonik, "Answering aggregation queries in a secure system model," in *VLDB*, 2007.

[81] L. Bottou and O. Bousquet, "The tradeoffs of large scale learning," in *NeurIPS*, 2008.

[82] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," *arXiv preprint arXiv:1712.01887*, 2017.

[83] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "Terngrad: Ternary gradients to reduce communication in distributed deep learning," in *NeurIPS*, 2017.

[84] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "Qsgd: Communication-efficient sgd via gradient quantization and encoding," in *NeurIPS*, 2017.

[85] A. Koloskova, S. U. Stich, and M. Jaggi, "Decentralized stochastic optimization and gossip algorithms with compressed communication," in *ICML*, 2019.

[86] C. Baskin, E. Schwartz, E. Zheltonozhskii, N. Liss, R. Giryes, A. M. Bronstein, and A. Mendelson, "Uniq: Uniform noise injection for non-uniform quantization of neural networks," *arXiv preprint arXiv:1804.10969*, 2018.

[87] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *ICML*, 2015.

[88] A. G. Anderson and C. P. Berg, "The high-dimensional geometry of binary neural networks," in *ICLR*, 2018.

[89] D. Soudry, I. Hubara, and R. Meir, "Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights," in *NeurIPS*, 2014.

[90] S. Migacz, "8-bit inference with tensorrt," in *GPU technology conference*, vol. 2, 2017, p. 7.

[91] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," *arXiv preprint arXiv:1412.7024*, 2014.

[92] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, "Scalable methods for 8-bit training of neural networks," in *NeurIPS*, 2018.

[93] "Tensorflow Federated," https://www.tensorflow.org/federated, 2019.

[94] T. Ryffel, A. Trask, M. Dahl, B. Wagner, J. Mancuso, D. Rueckert, and J. Passerat-Palmbach, "A generic framework for privacy preserving deep learning," *arXiv preprint arXiv:1811.04017*, 2018.

[95] "Aws deep learning ami," https://aws.amazon.com/machine-learning/amis/, 2019.

[96] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," 2017.

[97] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NeurIPS*, 2012.

[98] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[99] "Text generation with an rnn," https://www.tensorflow.org/tutorials/text/text_generation, 2019.

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

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

[102] S. Hardy, W. Henecka, H. Ivey-Law, R. Nock, G. Patrini, G. Smith, and B. Thorne, "Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption," *arXiv preprint arXiv:1711.10677*, 2017.

[103] J. Bernstein, Y.-X. Wang, K. Azizzadenesheli, and A. Anandkumar, "signsgd: Compressed optimisation for non-convex problems," *arXiv preprint arXiv:1802.04434*, 2018.

[104] B. Hitaj, G. Ateniese, and F. Perez-Cruz, "Deep models under the gan: information leakage from collaborative deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 603–618.

[105] K. Mandal and G. Gong, "Privfl: Practical privacy-preserving federated regressions on high-dimensional data over mobile networks," in *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*. ACM, 2019, pp. 57–68.

[106] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov, "How to backdoor federated learning," in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2020, pp. 2938–2948.

[107] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, 2017, pp. 506–519.

[108] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[109] J. Zhang, T. He, S. Sra, and A. Jadbabaie, "Why gradient clipping accelerates training: A theoretical justification for adaptivity," *arXiv preprint arXiv:1905.11881*, 2019.

[110] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[111] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[112] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 991–1008.

[113] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting {SGX} enclaves from practical side-channel attacks," in *2018 {Usenix} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 227–240.

[114] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, pp. 1–26, 2015.

[115] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune, "Memoir: Practical state continuity for protected modules," in *2011 IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 379–394.

[116] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer, "Pesos: Policy enhanced secure object store," in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–17.

[117] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh, "Shieldstore: Shielded in-memory key-value storage with sgx," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–15.

[118] "Arm TrustZone," https://developer.arm.com/ip-products/security-ip/trustzone, 2020.

[119] "AMD Memory Encryption," https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, 2016.

[120] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," *ACM Transactions on Computer Systems (TOCS)*, vol. 35, no. 4, pp. 1–32, 2018.

[121] "DCsv2-series," https://docs.microsoft.com/en-us/azure/virtual-machines/dcv2-series, 2020.

[122] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[123] R. Kunkel, D. L. Quoc, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer, "Tensorscone: A secure tensorflow framework using intel sgx," *arXiv preprint arXiv:1902.04413*, 2019.

[124] S. Chakrabarti, M. Hoekstra, D. Kuvaiskii, and M. Vij, "Scaling intel® software guard extensions applications with intel® sgx card," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2019, pp. 1–9.

[125] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *Advances in Neural Information Processing Systems*, 2019, pp. 103–112.

[126] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons, "Pipedream: Fast and efficient pipeline parallel dnn training," *arXiv preprint arXiv:1806.03377*, 2018.

[127] F. Gregor, W. Ozga, S. Vaucher, R. Pires, D. L. Quoc, S. Arnautov, A. Martin, V. Schiavoni, P. Felber, and C. Fetzer, "Trust management as a service: Enabling trusted execution in the face of byzantine stakeholders," *arXiv preprint arXiv:2003.14099*, 2020.

[128] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library {OS} for unmodified applications on {SGX}," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 645–658.

[129] "mongodb," https://www.mongodb.com/, 2020.

[130] "Kubernetes," https://kubernetes.io/, 2020.

[131] "Docker," https://www.docker.com/, 2020.

[132] "Python gil," https://realpython.com/python-gil/, 2020.

[133] "Cffi," https://cffi.readthedocs.io/en/latest/, 2020.

[134] "Diabetic retinopathy," https://www.kaggle.com/sovitrath/diabetic-retinopathy-224x224-gaussian-filtered, 2020.

[135] "Sms spam collection," https://www.kaggle.com/uciml/sms-spam-collection-dataset, 2020.

[136] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[137] Google, "Cloud TPU performance guide," https://cloud.google.com/tpu/docs/performance-guide, 2019.

[138] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of ACM EuroSys*, 2018.