Thèse n° 7096

# Private and Secure Distributed Learning

Présentée le 18 septembre 2020

à la Faculté informatique et communications
Laboratoire de calcul distribué
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

**Georgios DAMASKINOS**

Acceptée sur proposition du jury

Prof. A. Ailamaki, présidente du jury
Prof. R. Guerraoui, directeur de thèse
Prof. F. Taiani, rapporteur
Dr T. Parnell, rapporteur
Prof. B. Faltings, rapporteur

École
polytechnique
fédérale
de Lausanne

2020

Ἓν οἶδα ὅτι οὐδὲν οἶδα.

— Socrates

To the very few that love me unconditionally . . .

# Acknowledgements

This work would not have been possible without all the people who supported me research-wise and/or life-wise. Therefore, I employ “we” instead of “I” throughout this thesis.

The contribution of my lab (DCL) was crucial. First of all I would like to thank my advisor, Prof. Rachid Guerraoui, for the freedom, trust and support he gave me. I would like to thank all my co-authors and especially Rhicheek Patra for mentoring me in the beginning when I had no idea of what research is. Besides research collaborations, DCL has been an amazing work environment. With many of the lab members we had (educational) lunch times or chats, and with some of these members much more than that. Many thanks to the senior guys: George Chatzopoulos, Vasileios Trigonakis, Matej Pavlovic, Dragos-Adrian Seredinschi, David Kozhaya and Tudor David; to the guys that we started together: Igor Zablotchi, Karolos Antoniadis, El Mahdi El Mhamdi and Oana Balmau; and to the guys that joined afterwards: Vlad Nitu, Arsany Guirguis, Athanasios Xygkis, Jovan Komatovic and Matteo Monti. I would also like to thank France Faille for being supportive for more than just the administrative challenges.

I am grateful to the committee members of my Ph.D. defense, namely, Francois Taiani, Thomas Parnell, Boi Faltings and Anastasia Ailamaki. I want to thank EPFL and the European Research Council for financially supporting this work (via the EDIC Ph.D. fellowship and the European ERC Grant 339539 - AOC). Additionally, I would like to thank all the students that trusted me as a supervisor.

During my Ph.D. studies I did three internships, the second of which played an important role for this work. Therefore, I am grateful to Thomas Parnell, Celestine Mendler-Dünner and Nikolaos Papandreou. I wish to also thank Dimitrios, with whom we “did our life” during this internship in Zurich.

Lausanne was an amazing place for me during these five years, largely due to the people I was lucky enough to meet. I would like to thank the Greeks: Thanos, Vaggelis, Vaggos, Eirini, Tolis, Melivia, Prodromos, Panagiotis and Stella; the same-year EPFL students: Irene, Lana, Kirill, Helena, Hermina, Patricia and Dusan; the members of “Swingtime Lausanne” and especially Aurore Ferrage for also translating the abstract; and of course big thanks to Marios and Lefteris (with whom we were friends and collaborations before joining EPFL) for making life in Lausanne significantly more pleasant.

**Acknowledgements**

Greece has been and will always be the place I crave to go back for “recharging”. A special thanks to my close friends since school: Stavros, Fanis, Michalis, Giannis, Stefanos, Arianna, Tony, Dimitris and to the “destroyed” crew of Sikinos.

Finally and most importantly I would like to thank my family (mainly my mother Natassa, my father Antonis and my brother Alexandros) for being with me every single day, regardless of the distance, and believing in me. Last but not least, I would like to sincerely thank Athina for her support and for believing in us throughout these 5 years.

*Lausanne, June 19, 2020* Georgios Damaskinos

# Abstract

The ever-growing number of edge devices (e.g., smartphones) and the exploding volume of sensitive data they produce, call for *distributed machine learning* techniques that are *privacy-preserving*. Given the increasing computing capabilities of modern edge devices, these techniques can be realized by pushing the sensitive-data-dependent tasks of machine learning to the edge devices and thus avoid disclosing sensitive data.

In spite of the privacy benefits, existing techniques following this new computing paradigm are limited in addressing three important challenges. First, for many applications, such as news recommenders, data needs to be processed *fast*, before it becomes obsolete. Second, given the large amount of uncontrolled edge devices, some of them may undergo arbitrary (*Byzantine*) failures and deviate from the distributed learning protocol with potentially negative consequences such as learning divergence or even biased predictions. Third, privacy-preserving learning protocols call for formal *privacy guarantees* that imply additional tuning cost for the learning protocols.

To address the fast data challenge, we introduce FLEET, the first system for *online* learning at the edge. FLEET employs two core components, namely I-PROF and ADASGD. I-PROF is a lightweight regression-based profiler that adjusts the size of the sensitive-data-dependent tasks on highly heterogeneous mobile devices. ADASGD is a staleness-aware learning algorithm that is robust to asynchronous updates.

To make learning *secure* against Byzantine failures, we present AGGREGATHOR and KARDAM. AGGREGATHOR is a scalable framework that facilitates *synchronous* learning. AGGREGATHOR tolerates Byzantine failures mainly based on a filtering component that compares the updates sent from every device at each synchronous round. Given the tolerance to failures, we boost the network layer of AGGREGATHOR by introducing a communication protocol based on unreliable links. KARDAM operates in an *asynchronous* setup and employs two components: (a) a filtering component that, based on statistical properties of the learning procedure, tolerates Byzantine failures and (b) a dampening component that adjusts to stale information and enables fast convergence.

To address the privacy guarantees challenge, we present DP-SCD, a differentially private version of an algorithm with relatively low *tuning cost*, namely stochastic coordinate descent.

**Abstract**

DP-SCD is based on the insight that under independent noise addition (necessary for the privacy guarantees), the consistency of the auxiliary information that stochastic coordinate descent employs, holds in expectation. We give convergence guarantees for DP-SCD and demonstrate its superiority in terms of tuning without any significant impact on the privacy-utility trade-off.

**Keywords:** machine learning, distributed algorithms, edge computing, mobile Android devices, differential privacy, stochastic gradient descent, stochastic coordinate descent, Byzantine failures, online learning, profiling, synchronous learning, asynchronous learning.

# Résumé

Le nombre sans cesse croissant de dispositifs mobiles (par ex., les smartphones) et le volume explosif de données sensibles qu'ils produisent, demande des techniques d *apprentissage automatique distribué* permettant de *préserver la vie privée de leurs utilisateurs*. Grâce aux capacités croissantes de calcul des dispositifs mobiles, ces techniques peuvent être réalisées en poussant les tâches de l'apprentissage automatique qui gèrent les données sensibles vers les dispositifs mobiles et ainsi éviter de divulguer des données sensibles.

Malgré les avantages que les techniques actuelles qui suivent ce nouveau paradigme informatique apportent en termes de confidentialité, celles-ci restent limitées pour répondre à trois défis majeurs. Premièrement, pour de nombreuses applications, telles que les recommandeurs de news, les données doivent être traitées *rapidement*, avant qu'elles ne deviennent obsolètes. Deuxièmement, étant donné le grand nombre de dispositifs mobiles qui ne sont pas contrôlés, certains d'entre eux peuvent subir des fautes arbitraires (*Byzantines*) et s'écarter du protocole d'apprentissage distribué et avoir des conséquences potentiellement négatives telles que des divergences d'apprentissage ou même des prédictions biaisées. Troisièmement, les protocoles d'apprentissage préservant la confidentialité exigent des *garanties de confidentialité* formelles qui impliquent un coût de réglage supplémentaire pour les protocoles d'apprentissage.

Pour relever le défi du traitement rapide des données, nous présentons FLEET, le premier système d'apprentissage *en ligne* embarqué. FLEET utilise deux composants principaux, à savoir I-PROF et ADASGD. I-PROF est un profileur léger basé sur la régression qui ajuste la taille des tâches nécessitant des données sensibles sur des appareils mobiles très hétérogènes. ADASGD est un algorithme d'apprentissage sensible à la validité des mises à jour asynchrones.

Pour rendre l'apprentissage *sécurisé* contre les fautes Byzantines, nous présentons AGGREGATHOR et KARDAM. AGGREGATHOR est un cadre évolutif qui facilite l'apprentissage *synchrone*. AGGREGATHOR tolère les fautes Byzantines principalement sur la base d'un composant de filtrage qui compare les mises à jour envoyées depuis chaque appareil à chaque cycle synchrone. Compte tenu de la tolérance aux pannes, nous renforçons la couche réseau de AGGREGATHOR en introduisant un protocole de communication basé sur des liens peu fiables. KARDAM fonctionne dans une configuration *asynchrone* et utilise deux composants : (a) un composant de filtrage qui, sur la base des propriétés statistiques de la procédure

d'apprentissage, tolère les fautes Byzantines et (b) un composant d'amortissement qui s'adapte aux informations périmées et permet une convergence rapide.

Pour relever le défi des garanties de confidentialité, nous présentons DP-SCD, une version différentiellement privée d'un algorithme avec un *coût de réglage* relativement faible, à savoir la descente de coordonnées stochastiques. DP-SCD est basé sur la démonstration qu'en cas d'ajout de bruit indépendant (nécessaire pour garantir la confidentialité), la cohérence des informations auxiliaires que la descente en coordonnées stochastiques utilise, est conservée. Nous donnons des garanties de convergence pour DP-SCD et démontrons sa supériorité en termes de réglage sans faire de compromis entre confidentialité et utilité.

**Mots-clés :** apprentissage automatique, algorithmes distribués, informatique de bord, appareils mobiles Android, confidentialité différentielle, descente du gradient stochastique, descente de coordonnées stochastiques, fautes Byzantines, apprentissage en ligne, profilage, apprentissage synchrone, apprentissage asynchrone.

# Contents

**Acknowledgements** **i**

**Abstract (English/Français)** **iii**

**List of Figures** **xi**

**List of Tables** **xiii**

**I Introduction and Background** **1**

**1 Introduction** **3**
1.1 Motivation . . . . . 3
1.2 Contributions and Publications . . . . . 5
1.2.1 Fast Data . . . . . 5
1.2.2 Byzantine Failures . . . . . 6
1.2.3 Privacy Guarantees . . . . . 7
1.3 Organization of the Thesis . . . . . 7

**2 Preliminaries** **9**
2.1 Notation . . . . . 9
2.2 Function Properties . . . . . 10
2.3 Supervised Learning . . . . . 11
2.3.1 Stochastic Gradient Descent . . . . . 11
2.3.2 Stochastic Coordinate Descent . . . . . 12
2.4 Byzantine-resilient Learning . . . . . 13
2.5 Differentially Private Learning . . . . . 14

**II Fast Data** **17**

**3 Online Stochastic Gradient Descent** **19**
3.1 Introduction . . . . . 19
3.2 FLEET . . . . . 21
3.2.1 Architectural Overview . . . . . 21
3.2.2 Workload Bound via Profiling . . . . . 22

3.2.3 Adaptive Stochastic Gradient Descent . . . 25
3.2.4 Implementation . . . 28
3.3 Evaluation . . . 29
3.3.1 Online VS Standard Federated Learning . . . 29
3.3.2 AdaSGD Performance . . . 32
3.3.3 I-Prof Performance . . . 35
3.3.4 Resource Allocation . . . 38
3.3.5 Learning Task Assignment Control . . . 39
3.4 Related Work . . . 39
3.5 Concluding Remarks . . . 41

**III Byzantine Failures** **43**

**4 Synchronous Stochastic Gradient Descent** **45**
4.1 Introduction . . . 45
4.2 Design of AggregaThor . . . 47
4.2.1 Threat Model . . . 47
4.2.2 Architecture . . . 47
4.2.3 Byzantine Resilience . . . 48
4.2.4 Communication Layer . . . 50
4.3 Evaluation of AggregaThor . . . 50
4.3.1 Evaluation Setup . . . 50
4.3.2 Non-Byzantine Environment . . . 52
4.3.3 Byzantine Environment . . . 56
4.4 Related Work . . . 58
4.5 Concluding Remarks . . . 59

**5 Asynchronous Stochastic Gradient Descent** **61**
5.1 Introduction . . . 61
5.2 Setup . . . 63
5.3 Kardam . . . 64
5.3.1 Byzantine-resilient Filtering Component . . . 64
5.3.2 Staleness-aware Dampening Component . . . 68
5.4 Experiments . . . 70
5.5 Related Work and Concluding Remarks . . . 73

**IV Privacy Guarantees** **75**

**6 Differentially Private Stochastic Coordinate Descent** **77**
6.1 Introduction . . . 77
6.2 Setup . . . 78
6.3 Differentially Private Stochastic Coordinate Descent . . . 78

6.3.1 Privacy Analysis . . . 79
6.3.2 Cost Analysis . . . 81
6.3.3 Primal Version . . . 81
6.3.4 Sequential version . . . 82
6.4 Convergence Analysis . . . 83
6.4.1 Update Step . . . 85
6.4.2 Perturbation Step . . . 86
6.5 Experiments . . . 87
6.5.1 Setup . . . 88
6.5.2 Results . . . 89
6.6 Related Work . . . 91
6.7 Concluding Remarks . . . 92

**V Concluding Remarks** **93**

**7 Conclusions** **95**
7.1 Summary and Implications . . . 95
7.2 Future Directions . . . 96

**VI Appendices** **99**

**A Supplementary Material** **101**
A.1 Staleness Controller . . . 101

**B Detailed Proofs** **103**
B.1 Asynchronous Byzantine Resilience . . . 103
B.1.1 Correct Cone . . . 103
B.1.2 Convergence Analysis . . . 107
B.2 Differentially Private Stochastic Coordinate Descent . . . 115
B.2.1 Objective Decrease Lower Bound . . . 115

**Bibliography** **119**

**Curriculum Vitae** **133**

# List of Figures


3.1 Online FL enables frequent updates without requiring idle-charging-WiFi connected mobile devices. . . . . 20

3.2 The architecture of FLEET. . . . . 22

3.3 Motivation for lower bounding the mini-batch size. The noise introduced by weak workers (i.e., with small mini-batch sizes) may be detrimental to learning. 23

3.4 The linear relation between computation time and mini-batch size depends on the specific device, and may even vary for the same device, depending on operation conditions such temperature. . . . . 24

3.5 Gradient scaling schemes of SGD algorithms. ADASGD, proposed in this paper, dampens stale gradients with an exponentially decreasing function ($\Lambda(\tau)$) based on the expected percentage of non-stragglers ($\tau_{\text{thres}} := s$-th percentile of staleness values), and boosts the gradient of the straggler ($\tau = 48$) due to its low similarity ($sim(\boldsymbol{x}_i)$). . . . . 26

3.6 Online FL boosts Twitter hashtag recommendations by an average of 2.3× comparing to Standard FL. . . . . 30

3.7 Staleness distribution of collected tweets follows a Gaussian distribution ($\tau < 65$) with a long tail ($\tau > 65$). . . . . 31

3.8 Impact of staleness on learning. . . . . 33

3.9 Impact of long tail staleness on learning. . . . . 34

3.10 Staleness awareness with IID data. . . . . 35

3.11 Staleness awareness with differential privacy. . . . . 35

3.12 I-PROF outperforms MAUI and drives the computation time closer to the SLO. 36

3.13 I-PROF outperforms MAUI and drives the energy closer to the SLO. . . . . 37

3.14 Resource allocation of FLEET vs. CALOREE. . . . . 39

3.15 Threshold-based pruning. . . . . 40

4.1 The components of AGGREGATHOR, and their layered relations with existing components. New components have a gray background. AGGREGATHOR acts as a light framework based on TensorFlow, that manages the deployment and execution of a *model training session* over a cluster of machines. . . . . 48

4.2 High-level components and execution graph. Each gray rectangle represents a group of `tf.Operation`, and each plain arrow represents a `tf.Tensor`. The sub-graph between the gradients to the variables corresponds to Equation 2.7. For readability purpose, the tensors from the variables to each "Inference" and "Gradient" groups of operations have not been represented. . . . . 49
4.3 Overhead of AggregaThor in a non-Byzantine environment. . . . . 52
4.4 Latency breakdown. . . . . 53
4.5 Throughput comparison. . . . . 54
4.6 Impact of $f$ on convergence. . . . . 55
4.7 Impact of malformed input on convergence. . . . . 56
4.8 Impact of dropped packets on convergence. . . . . 57

5.1 The gradients computed by non-stale honest workers (black dashed arrows) are distributed around (and are on average equal to) the actual gradient (solid blue arrow) of the cost function (thin black curve). A Byzantine worker can propose an arbitrary poisoning vector (red dotted arrow). A honest but stale worker computes the correct gradient but for a stale version of the model (long green dotted arrow). . . . . 61
5.2 Illustration of correct cone. If $\left\|\mathbb{E}\boldsymbol{Kar}_{t_p} - \nabla\mathcal{F}(\boldsymbol{\theta}_t)\right\| \leq (1+\epsilon)\sqrt{d}\sigma + \epsilon'$ then $\langle\mathbb{E}\boldsymbol{Kar}_{t_p}, \nabla\mathcal{F}(\boldsymbol{\theta}_t)\rangle$ is upper bounded by $(1-\sin\alpha)\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\|^2$ where $\sin\alpha = \frac{(1+\epsilon)\sqrt{d}\sigma+\epsilon'}{\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\|}$. . . . . 67
5.3 Staleness-aware learning for CIFAR-100. FedAvg is equivalent (given the setup) to Kardam without the dampening component and SSGD denotes the ideal (synchronous) SGD execution. The staleness follows a Gaussian distribution ($mean = 12$, $\sigma = 4$) and the dampening functions are $\Lambda_1 = \frac{1}{\tau+1}$, $\Lambda_2 = e^{0.5\tau}$, $\Lambda_3 = e^{0.2\tau}$. . . . . 71
5.4 Impact of staleness for CIFAR-100. . . . . 72
5.5 Impact of staleness for EMNIST. . . . . 72

6.1 Privacy-utility trade-off. Better utility means lower MSE or larger accuracy. DP-SCD outperforms DP-SGD for the applications that enable exact update steps (namely ridge regression and SVMs) despite DP-SCD having less hyperparameters. . . . . 89
6.2 Impact of noise on convergence. Differential privacy does not prevent convergence but increases the noise in reducing the objective and the distance to the optimum (aligned with the result of Theorem 7). . . . . 90
6.3 Impact of mini-batch size on utility for DP-SCD. Utility increases with increasing mini-batch size, till a saturation point that depends on the level of privacy. . . . . 90

B.1 If $\left\|\mathbb{E}\boldsymbol{Kar}_{t_p} - \nabla\mathcal{F}(\boldsymbol{\theta}_t)\right\| \leq (1+\epsilon)\sqrt{d}\sigma + \epsilon'$ then $\langle\mathbb{E}\boldsymbol{Kar}_{t_p}, \nabla\mathcal{F}(\boldsymbol{\theta}_t)\rangle$ is upper bounded by $(1-\sin\alpha)\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\|^2$ where $\sin\alpha = \frac{(1+\epsilon)\sqrt{d}\sigma+\epsilon'}{\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\|}$. . . . . 107

# List of Tables

3.1 CNN parameters. . . . . 32
3.2 Performance of CALOREE [124] on new devices. . . . . 38

4.1 CNN Model parameters. . . . . 51

6.1 Comparison of utility bounds of $(\boldsymbol{\epsilon},\boldsymbol{\delta})$-DP algorithms for empirical risk minimization. . . . . 92

# Introduction and Background Part I

# 1 Introduction

## 1.1 Motivation

The number of edge devices and the data produced by them have grown tremendously over the last 10 years. While in 2009, mobile devices only generated 0.7% of the worldwide data traffic, in 2018 this number exceeded 50% [141]. On the one hand, there is an evident boost on the data generation capabilities of modern mobile devices mainly attributed to the plethora of sensors and to cheap storage capabilities. On the other hand, an increasing number of users (46% increase from 2015 till 2019 [157]) spend an increasing amount of time (30% increase in time spent on social media from 2015 till 2019 [52]) on their mobile devices.

As the volume of data produced by mobile devices explodes, users expose increasingly detailed and sensitive information. The challenge of protecting the privacy of this sensitive information is pervasive in machine learning (ML) applications such as recommenders, image-recognition applications, and personal assistants. These ML-based services are often applied to highly personal and possibly sensitive content, including conversations, geolocation, or physical traits (faces, fingerprints), and typically require tremendous volumes of data for their training. For example, people in the USA of age 18-24, type on average around 900 words per day (128 messages per day [121] with an average of 7 words per message [120]). The Android next-word prediction service is trained with sequences of 4.1 words [86] which means that each user generates around 220 training samples daily. Tens of millions or even billions of user devices employ next-word prediction services [21].

The computing capabilities of mobile devices are also rising rapidly. Modern mobile devices are able to perform heavy computation tasks with energy efficiency thanks to their CPUs, GPUs and recently also dedicated AI accelerators (e.g., Huawei Kirin 980 CPU, Intel Myriad X, Qualcomm Snapdragon 845, Apple A12 Bionic CPU). In parallel, network transfer latency for mobile devices is decreasing; 5G networks enable bandwidth of up to 10 Gbps.

Large industrial players are now seeking to exploit the rising power of mobile devices to protect the privacy of their users' data, while also reduce the demand on their server infrastructures.

Termed *Federated Learning* (FL), and spearheaded among others by Google [35, 96, 158], this new computing paradigm consists in offloading the storage and computation costs of ML applications onto mobile devices by training a global model on decentralized data stored locally. By design, FL is compatible with frameworks such as secure aggregation and differential privacy for end-to-end private learning [22].

The standard use of FL has so far been limited to a few lightweight and extremely privacy-sensitive services, such as next-word prediction [178]. Its appeal is however bound to grow as privacy-related scandals continue to unfold [133, 166], and new data protection regulations come into force [30, 77]. This trend is clearly visible in two of the most popular machine learning frameworks (namely TensorFlow and PyTorch) [149, 164], and also in the rise of startups such as S20.ai [150] or SNIPS (now part of Sonos) [160], which are betting on private decentralized learning. Recent research results show great potential for FL across various applications such as artificial image generation [168], hospitalizations for cardiac events prediction [26], augmented reality [34], or traffic flow prediction [113]. These are encouraging signs, but we argue in this thesis that existing FL works are unfortunately not effective in addressing the following challenges that arise for a large segment of ML-based applications.

**The challenge of fast data.** Many popular applications such as news applications or interactive social networks (e.g., Facebook, Twitter, Linkedin) involve large amount of data where a piece of information may become obsolete in a matter of hours or even minutes [123]. The faster this data is processed, the better the quality of the application. For example, consider two users of a news recommender, namely Alice and Bob. Bob clicks on some news articles (e.g., subway is shutdown due to an accident) before Alice opens the application as he wakes up earlier than her. The news recommender should consider the fresh clicks of Bob to deliver better (more relevant) recommendations for Alice when she also wakes up and uses the application. In existing FL schemes, the device of Bob computes updates much later (when idle and charging) and thus Alice is likely to miss the *highly temporal* information (e.g., regarding the subway shutdown) from the clicks of Bob. Addressing this challenge is non-trivial given the heterogeneity, connectivity, latency and energy constraints of mobile devices.

**The challenge of Byzantine failures.** The large amount of mobile devices increases the possibility of failures. Each device contributes to the global model training while susceptible to software bugs and hardware faults [83]. The local datasets of these devices may be corrupt or even the devices themselves may be hijacked by an adversary. Such failures can be fatal to most modern ML schemes, even if only a single device is faulty. Ideally, *secure* ML applications should tolerate *Byzantine* failures, encapsulating all possible malfunctions. These failures include poisoning attacks [18], in the parlance of adversarial machine learning [18, 19, 137]. Existing works prior to this thesis lack (a) system support for *scalable* Byzantine learning

and (b) algorithmic support for Byzantine learning with fast (*asynchronous*) updates in the presence of heterogeneous learners with dynamic connectivity.

**The challenge of privacy guarantees.** Differential privacy (DP) [68] is a particularly well-suited formal privacy guarantee for iterative ML algorithms in the presence of a strong adversary. The composability property of DP enables a fine-grained way of measuring privacy along with modularity for the DP mechanisms. Nevertheless, increasing the level of privacy comes at a cost: a reduction in the predictive power of the resulting model (known as the *privacy-utility trade-off*). ML algorithms with DP guarantees typically offer hyperparameters to *tune* this trade-off, such as the lot size [2]. While such a knob is desirable, the tuning cost (in terms of time) grows exponentially with the number of additional hyperparameters (e.g., when tuning via grid search). In standard FL, this tuning is based on simulation data [21] to avoid a potential privacy leakage or a performance impact (in terms of energy or latency) on the edge device.

## 1.2 Contributions and Publications

This thesis addresses the aforementioned three challenges by introducing system support and new algorithms with formal convergence guarantees, under different setups corresponding to each of these challenges. The grayed-out contribution is presented in another thesis [140] although is also relevant to this thesis and a result of research performed during the same doctoral studies. The publications [45, 56] are the result of research performed during internships in the same doctoral studies. The contributions of this thesis alongside their corresponding publications are the following.

### 1.2.1 Fast Data

- We introduce FLEET, the first FL system that specifically targets fast data by performing *online* model updates. FLEET employs a new *regression-based profiler* that determines the ML workload that each mobile device can perform within predefined energy and computation time thresholds. FLEET also makes use of a new *staleness-aware learning* algorithm suitable for incorporating frequent updates to the global (shared) model.

  Publication: [55] *Georgios Damaskinos, Rachid Guerraoui, Anne-Marie Kermarrec, Vlad Nitu, Rhicheek Patra, Francois Taiani. FLeet: Online Federated Learning via Staleness Awareness and Performance Prediction. (under submission)*

- We introduce I-SIM, a new similarity metric that targets fast data by performing *online*, incremental updates. I-SIM is essentially a time-aware version of the adjusted cosine similarity [151]. We depict the efficacy of our new metric by building (a) a scalable

recommender on top of Apache Spark[1] and Apache Cassandra[2] and (b) a trust predictor.

Publication: [57] *Georgios Damaskinos, Rachid Guerraoui, Rhicheek Patra.* Capturing the Moment: Lightweight Similarity Computations. *IEEE International Conference on Data Engineering (ICDE), 2017.*

### 1.2.2 Byzantine Failures

- We introduce AGGREGATHOR, the first scalable and Byzantine-resilient framework, based on one of the most popular existing ML frameworks, namely TensorFlow [1]. AGGREGATHOR performs synchronous learning without adding any constraints to the application development, and ensures Byzantine resilience by filtering out updates from potentially Byzantine workers. The filtering scheme is based on a majority of distance measurements between each pair of updates. The overhead of AGGREGATHOR over TensorFlow is moderate when there are no Byzantine failures. In fact, we have also shown that AGGREGATHOR can be viewed as a performance booster for TensorFlow, as it enables the use of an unreliable (and faster) underlying communication protocol, i.e., UDP-based instead of TCP-based.

  Publication: [53] *Georgios Damaskinos, El Mahdi El Mhamdi, Rachid Guerraoui, Arsany Guirguis, Sebastien Rouault. AggregaThor: Byzantine Machine Learning via Robust Gradient Aggregation. Conference on Machine Learning and Systems (SysML / MLSys), 2019.*

- We present the first asynchronous and Byzantine-resilient learning algorithm, that we call KARDAM. Asynchrony enables frequent (online updates), also suitable for addressing the challenge of fast data. KARDAM leverages global and local statistical properties of the learning procedure to filter out updates from potentially Byzantine workers, while prohibiting Byzantine workers from flooding the parameter server (which in turn would prevent honest workers from updating the model). KARDAM also uses a dampening scheme that scales each update based on the device delay. The computation overhead for each update is negligible as the filtering component of KARDAM is mostly scalar-based.

  Publication: [54] *Georgios Damaskinos, El Mahdi El Mhamdi, Rachid Guerraoui, Rhicheek Patra, Mahsa Taziki. Asynchronous Byzantine Machine Learning (the case of SGD). International Conference on Machine Learning (ICML), 2018.*

[1] http://spark.apache.org/
[2] http://cassandra.apache.org/

### 1.2.3 Privacy Guarantees

- We introduce the first differentially private stochastic coordinate descent algorithm (DP-SCD). Stochastic coordinate descent is particularly suitable for DP learning at the edge due to its relatively low number of hyperparameters and its remarkable performance for training a global model with local learners [115]. We formally derive the privacy bounds and study the convergence of DP-SCD. We empirically show that despite the reduced hyperparameter tuning, DP-SCD has a comparable privacy-utility trade-off compared to the popular alternative (DP-SGD).

  Publication: [58] *Georgios Damaskinos, Celestine Duenner, Rachid Guerraoui, Nikolaos Papandreou, Thomas Parnell. Differentially Private Stochastic Coordinate Descent. (under submission)*

## 1.3 Organization of the Thesis

The rest of this thesis is organized into six parts that include seven chapters and two appendices.

Part I

- Chapter 2 presents the main notation along with background on the main algorithmic tools used in this thesis. This background includes the main function properties used for the formal guarantees of the algorithms, description of the two workhorse optimization algorithms, namely stochastic gradient and coordinate descent, description of Byzantine-resilient learning and description of the differential privacy algorithmic framework in ML.

Part II

- Chapter 3 describes FLEET, the first FL system that performs online stochastic gradient descent updates on commercial Android devices with fast data.

Part III

- Chapter 4 introduces AGGREGATHOR the first scalable solution for Byzantine learning based on synchronous stochastic gradient descent.
- In Chapter 5, motivated by the challenge of fast data, we present KARDAM, the first asynchronous Byzantine-resilient scheme based on stochastic gradient descent.

Part IV

- Chapter 6 presents the first differentially private version of a workhorse optimization algorithm (namely stochastic coordinate descent) that is appealing due to its relatively low needs for hyperparameter tuning.

Part V

- Chapter 7 concludes the thesis with a summary of the contributions and their impact, along with future research directions.

Part VI

- Appendix A contains supplementary material for better understanding the experimental setup used in the thesis.
- Appendix B contains formal proofs for certain theoretical results of this thesis.

# 2 Preliminaries

In this chapter we provide the main notation, function properties and the necessary background for the rest of this thesis. The background involves the two main optimization algorithms, namely stochastic coordinate descent and stochastic gradient descent, the notion of Byzantine-resilient learning, and the notion of differentially private learning.

## 2.1 Notation

The following includes the main symbols used in this thesis alongside their description. Throughout the thesis, for the sake of clarity, the description may be overridden locally. We denote vectors and matrices in bold.

| | |
|---|---|
| $t$ | Step at the parameter server, incremented after each update |
| $T$ | Total number of of steps |
| $t_p$ | Step (given by the parameter server) of the model currently used by worker $p$ |
| $\boldsymbol{X}$ | Training examples $\in \mathbb{R}^{M \times N}$, i.e., $N$ training examples of dimensionality $M$ |
| $\boldsymbol{\theta}_t$ | Model (parameter vector) at step $t$ with dimensionality $d$ |
| $\boldsymbol{\alpha}_t$ | Dual model at step $t$ |
| $\boldsymbol{v}_t$ | Auxiliary or shared vector at step $t$ |
| $\mathcal{F}(\boldsymbol{X}, \boldsymbol{\theta})$ | Cost function for a model $\boldsymbol{\theta}$ |
| $\mathcal{F}^*$ | Dual cost function |
| $S(\boldsymbol{\alpha})$ | Dual suboptimality for model $\boldsymbol{\alpha}$ |
| $l_i(\boldsymbol{\theta})$ | Cost function for example $i$ |
| $l_i^*$ | Convex conjugate of cost function $l_i$ |
| $\mu$ | Smoothness parameter for $l_i$ |
| $\nu$ | Smoothness parameter for $l_i^*$ |
| $\xi$ | Mini-batch of training examples |
| $L$ | Mini-batch size, i.e., $\lvert\xi\rvert := L$ |
| $\boldsymbol{\nabla}\mathcal{F}(\boldsymbol{X}, \boldsymbol{\theta})$ | Gradient of the cost function |
| $\boldsymbol{G}(\boldsymbol{\theta}, \xi)$ | Gradient estimator on a mini-batch |

| | |
|---|---|
| $\boldsymbol{g}_p$ | Each gradient is a tuple $[\boldsymbol{g}_p, l]$ denoting that a worker $p$ computed the gradient $\boldsymbol{g}_p$ w.r.t $\boldsymbol{\theta}_l$ ($\boldsymbol{g}_p := \boldsymbol{G}(\boldsymbol{\theta}, \xi)$) |
| $R$ | Number of gradients that the server waits for before updating the model parameters ($R = 1$ in asynchrony) |
| $\mathcal{G}_t$ | Set of gradients that the server receives in step $t$. Note that $\lvert\mathcal{G}_t\rvert = R$ |
| $\tau_{tl}$ | Staleness value for a gradient $[\boldsymbol{g}, l]$ at step $t$ ($\tau_{tl} := t - l$) |
| $\Lambda(\tau)$ | Dampening value for staleness $\tau$ |
| $\chi$ | Staleness dampening bound: $\tau_{tl} \cdot \Lambda(\tau_{tl}) \leq \chi$ |
| $\gamma_t$ | Learning rate at step $t$ s.t. $\sum_{t=1}^{\infty} \gamma_t = \infty$ and $\sum_{t=1}^{\infty} \gamma_t^2 < \infty$ |
| $K$ | Global Lipschitz coefficient of $\boldsymbol{\nabla}\mathcal{F}$, i.e $K = \sup_{\boldsymbol{x},\boldsymbol{y}\in\mathbb{R}^d} (\frac{\lVert\boldsymbol{\nabla}\mathcal{F}(\boldsymbol{x})-\boldsymbol{\nabla}\mathcal{F}(\boldsymbol{y})\rVert}{\lVert\boldsymbol{x}-\boldsymbol{y}\rVert})$ |
| $\lambda$ | Regularization parameter |
| $\epsilon, \delta$ | Differential privacy parameters |
| $C$ | Update scaling factor |
| $\kappa$ | Suboptimality scaling factor |
| $D$ | Convexity horizon |
| $n$ | Total number of workers |
| $f$ | Number of Byzantine workers |
| $\lVert.\rVert$ | L2 norm |

## 2.2 Function Properties

We present the main properties of real-valued functions that are useful for the assumptions and formal derivations of the algorithms of this thesis.

**Definition 1** (Convexity)**.** *A function $h : \mathbb{R}^d \to \mathbb{R}$ is called* convex*, if*

$$h(t\boldsymbol{u} + (1-t)\boldsymbol{w}) \leq th(\boldsymbol{u}) + (1-t)h(\boldsymbol{w}) \quad \forall \boldsymbol{u}, \boldsymbol{w} \in \mathbb{R}^d \quad \forall t \in [0,1]$$

**Definition 2** ($\mu$-Strong Convexity)**.** *A function $h : \mathbb{R}^d \to \mathbb{R}$ is called* $\mu$-strongly convex *w.r.t. a norm $\lVert.\rVert$, for $\mu \geq 0$ if*

$$h(\boldsymbol{u}) \geq h(\boldsymbol{w}) + \langle \nabla h(\boldsymbol{w}), \boldsymbol{u} - \boldsymbol{w} \rangle + \frac{\mu}{2}\lVert \boldsymbol{u} - \boldsymbol{w} \rVert^2 \quad \forall \boldsymbol{u}, \boldsymbol{w} \in \mathbb{R}^d$$

**Definition 3** ($\nu$-Smoothness)**.** *A function $h : \mathbb{R}^d \to \mathbb{R}$ is called* $\mu$-smooth *w.r.t. a norm $\lVert.\rVert$, for $\nu \geq 0$ if*

$$h(\boldsymbol{u}) \leq h(\boldsymbol{w}) + \langle \nabla h(\boldsymbol{w}), \boldsymbol{u} - \boldsymbol{w} \rangle + \frac{\nu}{2}\lVert \boldsymbol{u} - \boldsymbol{w} \rVert^2 \quad \forall \boldsymbol{u}, \boldsymbol{w} \in \mathbb{R}^d$$

**Definition 4** ($K$-Lipschitz Continuity)**.** *A function $h : \mathbb{R}^d \to \mathbb{R}$ is called* $K$-Lipschitz continuous *w.r.t. a norm $\lVert.\rVert$, for $K \geq 0$ if*

$$\lvert h(\boldsymbol{u}) - h(\boldsymbol{w}) \rvert \leq K\lVert \boldsymbol{u} - \boldsymbol{w} \rVert \quad \forall \boldsymbol{u}, \boldsymbol{w} \in \mathbb{R}^d$$

## 2.3 Supervised Learning

This thesis targets *supervised* learning, i.e., the dataset contains the ground truth for the corresponding learning task. The learning task consists in making accurate predictions for the labels of each data instance. For the example of image classification, each data instance has a set of features (image pixels), and a set of labels (e.g., {cat, person}).

Supervised learning includes both convex and non-convex optimization problems with the following form.

$$\min_{\boldsymbol{\theta}} \mathcal{F}(\boldsymbol{X},\boldsymbol{\theta}) \text{ where } \mathcal{F}(\boldsymbol{X},\boldsymbol{\theta}) := \frac{1}{N}\sum_{i=1}^{N} \ell_i(\boldsymbol{\theta}^\top \boldsymbol{x}_i) + \frac{\lambda}{2}\|\boldsymbol{\theta}\|^2 \tag{2.1}$$

where the model vector $\boldsymbol{\theta}$ is learnt from the training data $\boldsymbol{X} \in \mathbb{R}^{M\times N}$ with $N$ training examples $\boldsymbol{x}_i \in \mathbb{R}^M$ as columns, $\lambda$ denotes the regularization parameter, and $\ell_i$ the loss function for the example $\boldsymbol{x}_i$. The norm $\|\cdot\|$ refers to the $L_2$-norm.

### 2.3.1 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a very popular ML technique for minimizing the objective shown in Problem 2.1. SGD computes the gradient ($\boldsymbol{G}(\boldsymbol{\theta},\xi)$) and then updates the model parameters ($\boldsymbol{\theta}$) in a direction opposite to that of the gradient (descent). The vanilla SGD update rule given a sequence of learning rates $\{\gamma_t\}$ at any given step[1] is the following:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \gamma_t \cdot \boldsymbol{G}(\boldsymbol{\theta}^{(t)},\xi)) \tag{2.2}$$

The popularity (and the stochasticity) of SGD stem from its ability to employ noisy approximations of the actual gradient. In a distributed setup, SGD employs a random subset, namely a *mini-batch* ($\xi$), of $L < N$ training instances for the gradient computation:

$$\boldsymbol{G}(\boldsymbol{\theta},\xi) := \nabla\mathcal{F}(\boldsymbol{X}_\xi,\boldsymbol{\theta}) := \sum_{i=1}^{L} \nabla\mathcal{F}(\xi_i,\boldsymbol{\theta}) \tag{2.3}$$

The addends of the summation shown in Equation 2.3 can be computed in parallel. The size of the mini-batch ($L$) affects the amount of parallelism that modern computing clusters (e.g., multi-GPU) largely benefit from. Scaling the mini-batch size to exploit additional parallelism requires however a non-trivial selection of the sequence of learning rates [80]. A very important assumption for the convergence properties of SGD is that each gradient is an unbiased estimation of the actual gradient, i.e., gradients that are on expectation equal to the actual gradient. This assumption is typically ensured through uniform random sampling and *synchronous* training.

[1] A step denotes an update in the model parameters.

### 2.3.2 Stochastic Coordinate Descent

Stochastic Coordinate Descent (SCD) is another popular ML technique for minimizing the objective shown in Problem 2.1, especially for the case of Generalized Linear Models (GLMs) [118]. GLMs are popular due to their interpretability and relatively (comparing to, for example, neural networks) low number of model parameters that need training. These models employ convex loss functions $l_i$ that apply on data through a linear map ($\boldsymbol{\theta}^\top \boldsymbol{x}_i$). For this thesis and regarding SCD, we focus on GLMs.

SCD proceeds iteratively and repeatedly selects a coordinate $j \in [M]$ at random, solves a one dimensional auxiliary problem, and updates the parameters $\boldsymbol{\theta}$ as follows.

$$\boldsymbol{\theta}^+ \leftarrow \boldsymbol{\theta} + \boldsymbol{e}_j \delta^\star \quad \text{where} \quad \delta^\star = \arg\min_{\delta} \mathcal{F}(\boldsymbol{X}, \boldsymbol{\theta} + \boldsymbol{e}_j \delta) \tag{2.4}$$

where $\boldsymbol{e}_j$ denotes the unit vector with value 1 at position $j$. This problem often has a closed form solution; otherwise $\mathcal{F}$ is generally replaced by its second order Taylor approximation.

A crucial approach for improving the time complexity of each SCD update is to keep an auxiliary vector $\boldsymbol{v} := \boldsymbol{\theta}^\top \boldsymbol{X}$ (also called *shared vector*) in memory in order to avoid recurring computations. This auxiliary vector is updated in each iteration as $\boldsymbol{v}^+ \leftarrow \boldsymbol{v} + \delta^\star \boldsymbol{x}_i$.

**Dual SCD.** SCD can be equivalently applied to the dual formulation of Problem 2.1, often referred to as SDCA [155]. The dual optimization problem has the following objective.

$$\min_{\boldsymbol{\alpha}} \mathcal{F}^*(\boldsymbol{X}, \boldsymbol{\alpha}) \text{ where } \mathcal{F}^*(\boldsymbol{X}, \boldsymbol{\alpha}) := \frac{1}{N} \sum_{i=1}^{N} \ell_i^*(-\alpha_i) + \frac{1}{2\lambda N^2} \|X\boldsymbol{\alpha}\|^2 \tag{2.5}$$

where $\boldsymbol{\alpha} \in \mathbb{R}^N$ denotes the dual model vector and $\ell_i^*$ the convex conjugate of the loss function $\ell_i$. For the dual problem, the auxiliary vector is $\boldsymbol{v} := \boldsymbol{X}\boldsymbol{\alpha}$.

We use the first order optimality conditions to relate the primal and the dual model vectors as $\boldsymbol{\theta}(\boldsymbol{\alpha}) = \frac{1}{\lambda N} \boldsymbol{X}\boldsymbol{\alpha}$, which leads to the important definition of the duality gap [65]:

$$\text{Gap}(\boldsymbol{\alpha}) := \mathcal{F}^*(\boldsymbol{X}, \boldsymbol{\alpha}) + \mathcal{F}(\boldsymbol{X}, \boldsymbol{\theta}(\boldsymbol{\alpha})) = \langle \boldsymbol{X}\boldsymbol{\alpha}, \boldsymbol{\theta}(\boldsymbol{\alpha}) \rangle + \frac{\lambda}{2} \|\boldsymbol{\theta}(\boldsymbol{\alpha})\|^2 + \frac{\lambda}{2} \|\boldsymbol{\theta}\|^2 \tag{2.6}$$

By the construction of the two problems, the optimal values for the objectives match in the convex setting and the duality gap attains zero [155]. Therefore, the model $\boldsymbol{\theta}$ can be learnt from either of the two objectives shown in Problems 2.1 and 2.5, i.e., the primal and the dual problems are identical from an algorithmic perspective.

However, the primal and dual problems are quite different with respect to their data access pattern. When applied to the dual, SCD computes each update by processing one example,

whereas the primal SCD processes one coordinate across all the examples. This fact is crucial for the privacy guarantees as we show in Part IV.

## 2.4 Byzantine-resilient Learning

SGD has been both theoretically and empirically proven to not be resilient against Byzantine worker behavior [20]. A Byzantine worker can propose a gradient that can completely ruin the training procedure.

**Weak Byzantine resilience.** A very recent line of theoretical research has addressed the problem of Byzantine-resilient SGD [20, 72, 162, 176, 179]. These SGD variants all aggregate the gradients obtained from the workers before deriving the final gradient. Essentially, they compute statistics (e.g. median, quantiles, principal component analysis) over the set of aggregated gradients to derive the final gradient. Moreover, the update rule (Equation 2.2) for $n$ workers becomes:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \gamma_t \, \mathrm{GAR}\big(\boldsymbol{G}_1(\boldsymbol{\theta}^{(t)}, \xi_1), \ldots, \boldsymbol{G}_n(\boldsymbol{\theta}^{(t)}, \xi_n)\big) \tag{2.7}$$

where GAR denotes the *gradient aggregation rule*, and $\boldsymbol{G}_i(\boldsymbol{\theta}^{(t)}, \xi_i)$ denotes the gradient estimate of worker $i$, using its own randomly drawn mini-batch $\xi_i$ and the global model $\boldsymbol{\theta}$ at update step $t$.

In the context of non-convex optimization, it is generally hopeless to try to find a global minimum for $\mathcal{F}(\boldsymbol{X}, \boldsymbol{\theta})$. Instead, what can be proven is that the sequence of parameter vectors converges to a region around some $\boldsymbol{\theta}^*$ where $\nabla \mathcal{F}(\boldsymbol{X}, \boldsymbol{\theta}^*) = 0$, i.e, a flat region of the loss function [23]. Any gradient aggregation rule that satisfies this convergence property despite the presence of $f$ Byzantine workers, among the total of $n$ workers, is called *weakly Byzantine-resilient*.

**Strong Byzantine resilience.** In high dimensional spaces (i.e., $d \gg 1$), and with a highly non-convex loss function (which is the case in modern machine learning [87, 116]), weak Byzantine resilience may lead to models with poor performance in terms of prediction accuracy, as a Byzantine worker can fool a provably converging SGD rule by leveraging a *dimensional leeway* [72]. More precisely, this worker can make the system converge, as guaranteed by its designers, but to a state with poor (as compared to the maximum possible one in a non-Byzantine environment) prediction accuracy.[2]

[2] The intuition behind this issue relates to the so-called *curse of dimensionality*, a fundamental problem in learning: a square of unit 1 on each side has a diagonal of length $\sqrt{2}$. In dimension 3, the cube of unit 1 has a diagonal of length $\sqrt{3}$. In dimension $d$, the diagonal is of length $\sqrt{d}$. Given $d \gg 1$, points that differ by a distance of at most 1 in each direction end up being in a huge distance from each other.

We define *strong Byzantine resilience* as the ability for a GAR, in addition to being weakly Byzantine-resilient, to select gradients that are (in each coordinate) in a distance of at most $\frac{1}{\sqrt{d}}$ from some correct gradient, despite the presence of $f$ Byzantine workers among the total $n$ workers. A more detailed analysis of strong Byzantine resilience is available [71].

Attacking a non-Byzantine resilient GARs such as averaging is easy [20]. Attacking a GAR that ensures weak Byzantine resilience requires a powerful adversary, i.e., at least able to carry out the attack presented in [72].

## 2.5 Differentially Private Learning

Differential privacy (DP) is a guarantee for a function $f$ applied on a database of sensitive data [68]. In the context of ML, the objective function is the update function of the algorithm (Equations 2.2, 2.4 and 2.7). Two input datasets are *adjacent* if they differ only in a single input-label pair. Querying the model translates into making predictions for the label of some new input.

**Definition 5** (Differential privacy)**.** *A randomized mechanism $\mathcal{M} : D \rightarrow \mathbb{R}$ satisfies ($\epsilon$, $\delta$)-differential privacy if for any two adjacent inputs $d, d' \in D$ and for any subset of outputs $S \subseteq \mathbb{R}$ it holds that:*

$$Pr[\mathcal{M}(d) \in S] \leq e^{\epsilon} Pr[\mathcal{M}(d') \in S] + \delta \tag{2.8}$$

The *Gaussian mechanism* is a popular method for making a deterministic function $f : D \rightarrow \mathbb{R}$ differentially private (according to Definition 5). By adding Gaussian noise to the output of the function we can hide particularities of individual input values. The resulting mechanism is defined as: $\mathcal{M}(d) := f(d) + \mathcal{N}(0, S_f^2 \sigma^2)$ where $S_f$ denotes the sensitivity of the function $f$. This definition can be readily extended to the multi-dimensional case in order to fit the ML setting: An iterative machine learning algorithm can be viewed as a function $\boldsymbol{f} : \mathbb{R}^{M \times N} \rightarrow \mathbb{R}^M$ that repeatedly computes model updates from the data and thus requires a multi-dimensional noise addition at each iteration $t$:

$$\mathcal{M}_t(\boldsymbol{d}) = \boldsymbol{f}(\boldsymbol{d}) + \mathcal{N}(0, S_f^2 \sigma^2 \boldsymbol{I}),\ \boldsymbol{I} \in \mathbb{R}^{M \times M} \tag{2.9}$$

The sensitivity is defined as:

$$S_f := \max_{\text{adjacent } \boldsymbol{d}, \boldsymbol{d}'} \|\boldsymbol{f}(\boldsymbol{d}) - \boldsymbol{f}(\boldsymbol{d}')\| \tag{2.10}$$

**DP-SGD.** As an example of how the Gaussian mechanism can be applied to make ML differentially private, we consider SGD (§2.3.1). SGD can be made differentially private by using the Gaussian mechanism: For each update, DP-SGD adds noise with the variance given by the sensitivity of the update function, i.e., $S_f = \|G(\boldsymbol{\theta}, \xi)\|$. In practice, an additional gradient clipping step enforces a desired bound on the sensitivity $S_f$ [2].

**Privacy accounting.** Measuring the privacy leakage of a randomized mechanism $\mathcal{M}$ boils down to computing $(\epsilon, \delta)$, i.e., computing a bound for the privacy loss $(\epsilon)$ that holds with certain probability $(\delta)$. In the context of ML, $\mathcal{M}$ often consists of a sequence of mechanisms $\mathcal{M}_i$ that, for example, denote the model update at each iteration $i$. All these mechanisms have the same pattern in terms of sensitive (training) data access for most typical iterative ML algorithms, including SCD. Computing $(\epsilon, \delta)$ given the individual pairs $(\epsilon_i, \delta_i)$ is a problem known as composability. The *moments accountant* [2] is an important method that computes tighter bounds for the privacy loss (i.e., tighter $(\epsilon, \delta)$) compared to the standard composition theorems. This method is tailored to the Gaussian mechanism and employs the log moment of each $\mathcal{M}_i$ to derive the bound of the total privacy loss. The moments accountant can be viewed as a function that returns the privacy loss bound:

$$\epsilon = \mathrm{MA}(\delta, \sigma, q, T) \tag{2.11}$$

where $\sigma$ is the noise magnitude, $q$ is the sampling ratio (i.e., the ratio of the data that each $\mathcal{M}_i$ uses over the total data), and $T$ is the number of individual mechanisms $\mathcal{M}_i$ [2].

# Fast Data Part II

# 3 Online Stochastic Gradient Descent

## 3.1 Introduction

The starting point of this work is that Standard FL [21] is not suitable for ML-based applications that leverage *fast data* and thus could greatly benefit from frequent *online* model updates, mainly due to its constraint for *high device availability*: the selected mobile devices need to be idle, charging and connected to an unmetered network. This constraint removes any impact perceived by users, but also limits the availability of devices for learning tasks. Google observed lower prediction accuracy during the day as few devices fulfill this policy and these generally represent a skewed population [178]. With most devices available at night the model is generally updated every 24 hours.

This constraint may be acceptable for some ML-based services but is problematic to what we call *online learning* systems, which underlie many popular applications such as news recommenders or interactive social networks (e.g., Facebook, Twitter, Linkedin). These systems involve large amounts of data with high *temporality*, that generally become obsolete in a matter of hours or even minutes [123]. To illustrate the limitation of Standard FL, consider two users, Alice and Bob, who belong to a population that trains the ML model underlying a news recommendation system (Figure 3.1). Bob wakes up earlier than Alice and clicks on some news articles. To deliver fresh and relevant recommendations, these clicks should be used to compute recommendations for Alice when she uses the app, slightly after Bob. In Standard FL (upper half Figure 3.1), the device of Bob would wait until much later (when idle, charging and connected to WiFi) to perform the learning task thus negating the value of the task results for Alice. In an online learning setup (lower half of Figure 3.1), the activity of Bob is rapidly incorporated into the model, thereby improving the experience of Alice.

**Challenges and contributions.** In this paper we address the aforementioned limitation and enable *Online* FL. We introduce FLEET, the first FL system that specifically targets online learning, acting as a middleware between the operating system of the mobile device and the

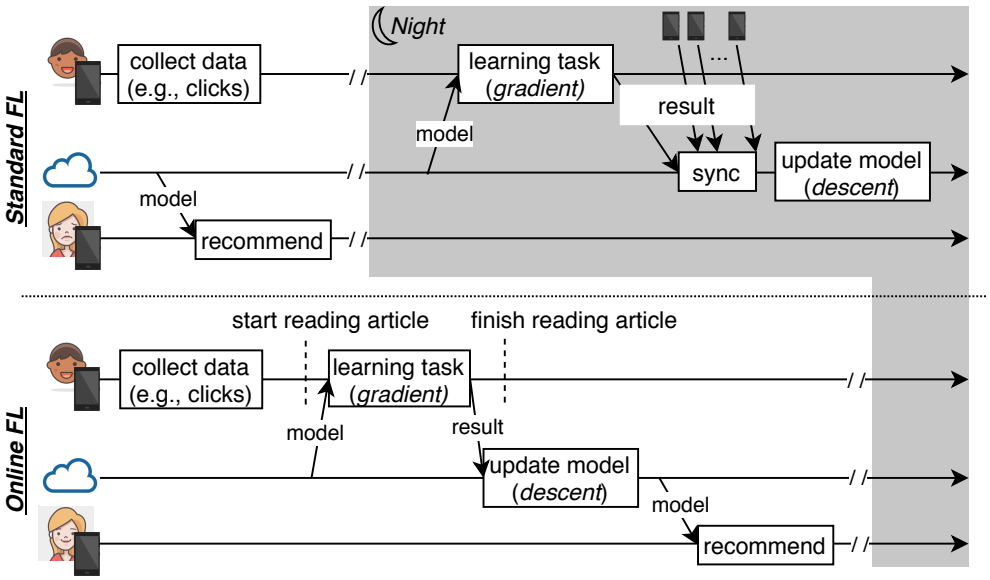



**Figure 3.1** – Online FL enables frequent updates without requiring idle-charging-WiFi connected mobile devices.

ML-based application. FLEET addresses two major problems that arise after forfeiting the high device availability constraint.

First, learning tasks may have an energy impact on mobile devices now powered on a battery. Given that learning tasks are generally compute intensive, they can quickly discharge the device battery and thereby degrade user experience. To this end, FLEET includes I-PROF (§3.2.2), our new profiling tool which predicts and controls the computation time and the energy consumption of each learning task on mobile devices. The goal of I-PROF is not trivial given the high heterogeneity of the devices and the performance variability even for the same device over time [132] (as we show in §3.3).

Second, as mentioned above, synchronous training discards all late results arriving after the model is updated thus wasting the battery of the corresponding devices and their potentially useful data. Frequent model updates call for small synchronization windows that given the high performance variability, amplify this waste. We therefore replace the synchronous scheme of Standard FL with asynchronous updates. However, asynchronous updates introduce the challenge of *staleness* as multiple users are now free to perform learning tasks at arbitrary times. A stale result occurs when the learning task was computed on an outdated model version; meanwhile the global model has progressed to a new version. Stale results add noise to the training procedure, slow down or even prevent its convergence [93, 185]. Therefore, FLEET includes ADASGD (§3.2.3), our new Stochastic Gradient Descent (SGD) algorithm that tolerates staleness by dampening the impact of outdated results. This dampening depends on (a) the past observed staleness values and (b) the similarity with past learning tasks.

We fully implemented the server side and the Android client of FLEET and our code is available[1]. We evaluate the potential of FLEET and show that it can increase the accuracy of a recommendation system (that employs Standard FL) by 2.3× on average, by performing the same number of updates but in a more timely (online) manner. Even though the learning tasks drain energy directly from the battery of the phone, they consume on average only 0.036% of the battery capacity of a modern smartphone per user per day. We also evaluate the components of FLEET on 40 commercial Android devices, by using popular benchmarks for image classification. Regarding I-PROF, we show that 90% of the learning tasks deviate from a fixed Service Level Objective (SLO) of 3 seconds by at most 0.75 seconds in comparison to 2.7 seconds for the competitor (the profiler of MAUI [48]). The energy deviation from an SLO of 0.075% battery drop is 0.01% for I-PROF and 0.19% for the competitor. We also show that our staleness-aware learning algorithm (ADASGD) learns 18.4% faster than its competitor (DYNSGD [93]) on heterogeneous data.

## 3.2 FLEET

FLEET incorporates two components we consider necessary in any system that has the ambition to provide both, the (a) privacy of FL and (b) the precision of online learning systems. The first component is I-PROF, a *lightweight ML-based profiling* mechanism that controls the computation time and energy of the learning task by using ML-based estimators. The second component of FLEET is ADASGD, a new adaptive learning algorithm that tolerates stale updates by automatically adjusting their weight.

### 3.2.1 Architectural Overview

Similar to the implementation of Standard FL [21], FLEET follows a client-server architecture (Figure 3.2) where each user hosts a *worker* and the service provider hosts the *server* (typically in the cloud). In FLEET, the worker is a library that can be used by any mobile ML-based application (e.g., a news articles application). The model training protocol of FLEET is the following (the numbers refer to Figure 3.2):

(1) The worker requests a learning task and sends information regarding the labels of the local data along with information about the state of the mobile device. We introduce the purpose of this information in Steps 2 and 3.
(2) I-PROF employs the device information to bound the workload size (i.e., set a *mini-batch size bound*) that will be allocated to this worker such that the computation time and energy consumption approximate an SLO set by the service provider or negotiated with the user (details in §3.2.2).
(3) ADASGD computes a *similarity* for the requested learning task with past learning tasks in order to adapt to updates with new data (details in §3.2.3).

[1] https://github.com/gdamaskinos/fleet

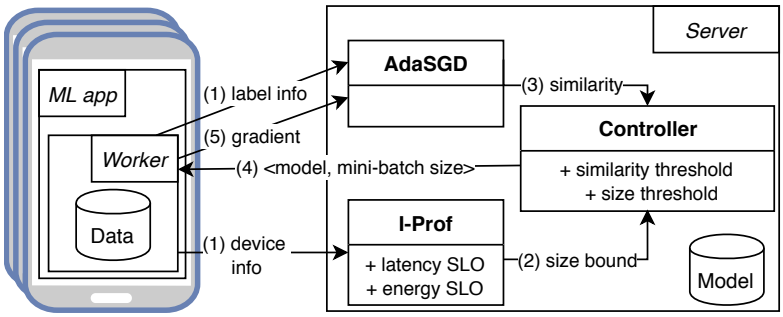



**Figure 3.2 –** The architecture of FLEET.

(4) In order to prevent the computation of learning tasks with low or no utility for the learning procedure, the controller checks if both the mini-batch size and the similarity value pass certain *thresholds* set by the service provider. If the check fails, the request of the worker is rejected, otherwise the controller sends the model parameters and the mini-batch size to the worker and the learning task execution begins (details about setting these thresholds in §3.2.4).
(5) Based on the mini-batch size returned by the server, the worker samples from its locally collected data, performs the learning task, i.e., computes the model *gradient* and sends it back to the server. On the server side, ADASGD updates the model after dynamically adapting this gradient based on its *staleness* and on its similarity value (details in §3.2.3).

The above protocol maintains the key "privacy-readiness" of Standard FL: the user data never leave the device during the learning procedure.

### 3.2.2 Workload Bound via Profiling

In Online FL, a mobile device should be able to compute model updates at any time, not only during the night, when the mobile device is idle, charging and connected to WiFi. Therefore, FLEET drops the constraint of Standard FL for high device availability. Hence, the learning task now drains energy directly from the battery of the device. Controlling the impact of a learning task on the user application in terms of energy consumption and computation time becomes crucial. To this end, FLEET incorporates a profiling mechanism that determines the workload size (i.e., the mini-batch size) appropriate for each mobile device.

**Best-effort solution.** To highlight the need for a specific profiling tool, we first consider a naive solution in which users process data points until they reach the SLO either in terms of computation time or energy. At this point, a worker sends back the resulting "best-effort" gradient. The service provider cannot decide beforehand whether for a given device, the cost (in terms of energy, time and bandwidth) to download the model, compute and upload the gradient is worth the benefit to the model. Updates computed on very small mini-batch sizes

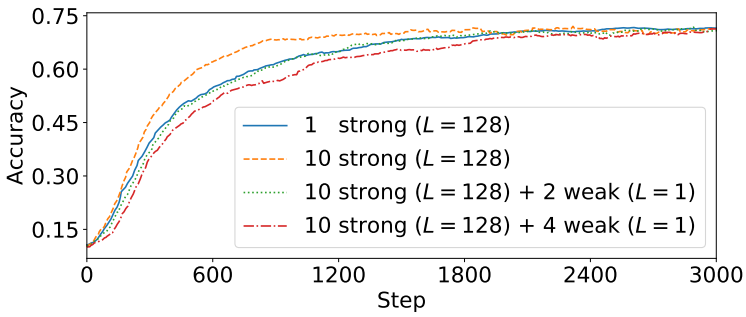



**Figure 3.3 –** Motivation for lower bounding the mini-batch size. The noise introduced by weak workers (i.e., with small mini-batch sizes) may be detrimental to learning.

(by weak devices) will perturb the convergence of the overall model, and might even negate the benefit of other workers.

To illustrate this point, consider the experiment of Figure 3.3. The figure charts the result of training a Convolutional Neural Network on CIFAR10 [42] under different combinations of "strong" and "weak" workers. The strong workers compute on a mini-batch size of 128 while the weak workers compute on a mini-batch size of 1. We observe that even 2 weak workers are enough to cancel the benefit of distributed learning, i.e., the performance with 10 strong + 2 weak workers is the same as training with a single strong worker.

One way to avoid this issue could be to drop all the gradients computed on a mini-batch size lower than a given bound or weigh them with a tiny factor according to the size of their underlying mini-batch. This way would however waste the energy required to obtain these gradients. A profiler tool that can estimate the maximum mini-batch size (workload bound) that a worker can compute is necessary for the controller to decide whether to reject the computation request of this worker, before the gradient computation. Unfortunately, existing profiling approaches [28, 40, 41, 48, 85, 100, 180] are not suitable because they are either relatively inaccurate (see §3.3.3) or they require privileged access (e.g., rooted Android devices) to low-level system performance counters.

**I-PROF.** Mobile devices have a significantly lower level of parallelism in comparison with cloud servers. For example, the graphical accelerators of mobile devices generally have 10-20 cores [10, 146] while the GPUs on a server have thousands of cores [134]. Given this low level of parallelism, even a relatively small mini-batch size can fill the processing pipelines. Hence, any additional workload will linearly increase the computation time and the energy consumption. Based on this observation, we built I-PROF, a lightweight profiler specifically designed for Online FL systems. We design I-PROF with three goals in mind: (a) operate effectively with data from a wide range of device types, (b) do so in a lightweight manner, i.e., introduce only a negligible latency to the learning task and (c) rely only on the data available on a stock (non-rooted) Android device.

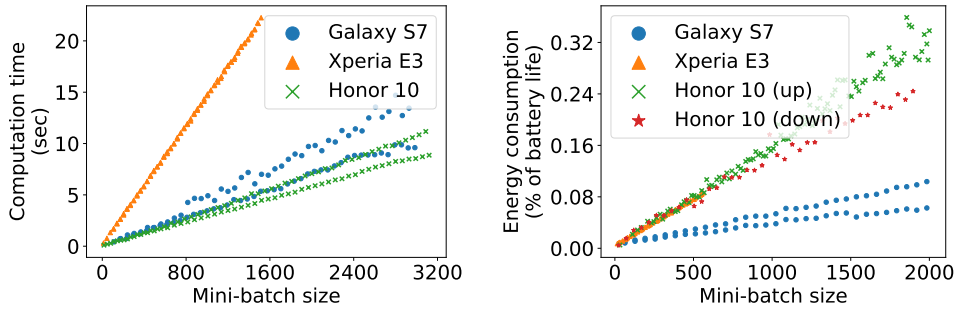



**Figure 3.4 –** The linear relation between computation time and mini-batch size depends on the specific device, and may even vary for the same device, depending on operation conditions such temperature.

I-PROF employs an ML-based scheme to capture how the device features affect the computation time and energy consumption of the learning task. I-PROF predicts the largest mini-batch size a device can process while respecting both the time and the energy limits set by the SLO. To this aim, I-PROF uses two predictors, one for computation time and one for energy. Each predictor updates its state with data from the device information sent by the workers.

Designing such predictors is however tricky, as modern mobile phones exhibit a wide range of capabilities. For example, in a matrix multiplication benchmark, Galaxy S6 performs 7.11 Gflops whereas Galaxy S10 performs 51.4 Gflops [117]. Figure 3.4 illustrates this heterogeneity on three different mobile devices by executing successive learning tasks of increasing mini-batch size ("up"). After reaching the maximum value, we let the devices cool down and execute subsequent learning tasks with decreasing mini-batch size ("down"). We present the results for the up-down part with the same color-pattern, except for Honor 10 in Figure 3.4(b) that we split for highlighting the difference. Figure 3.4 illustrates that the linear relation changes for each device and for certain devices (Honor 10, Galaxy S7) also changes with the temperature. Note that Honor 10 shows an increased variance at the end of the "up" part (Figure 3.4(b)) that is attributed to the high temperature of the device. The variance is significantly smaller for the "down" part.

In the following, we describe how I-PROF predicts the mini-batch size ($L$) given a computation time SLO[2] ($t_{SLO}$). The computation time linearly increases with the workload size, i.e., $t_{comp} = \alpha \cdot L$, where $\alpha$ depends on the device and its state. Considering the goal (i.e., $t_{comp} \rightarrow t_{SLO}$), the optimal mini-batch size is predicted as:

$$\hat{L} = \max\left(1, \frac{t_{SLO}}{\hat{\alpha}}\right) \tag{3.1}$$

[2]The prediction method given an energy SLO is the same.

I-PROF estimates the slope $\hat{\alpha}$ from the device characteristics and operational conditions using a method that combines linear regression and online passive-aggressive learning [47].

The input to this method is a set of device features based on measurements available through the Android API, namely available memory, total memory, temperature and sum of the maximum frequency over all the CPU cores. However, these features only encode the computing power of a device. For the prediction based on the energy SLO, I-PROF also needs a feature that encodes the energy efficiency of each device. We choose this additional feature as the energy consumption per non-idle CPU time[3]. We show in our evaluation (§3.3.3) that these features achieve our three design goals. Given a vector of device features ($\boldsymbol{x}$), and a vector of model parameters ($\boldsymbol{\theta}_{\text{prof}}$), the slope $\hat{\alpha}$ is estimated as $\hat{\alpha} = \boldsymbol{x}^T \boldsymbol{\theta}_{\text{prof}}$.

I-PROF uses a cold-start linear regression model for the first request of each user device. We pre-train the cold-start model using ordinary least squares with an offline dataset. This dataset consists of data collected by executing requests from a set of training devices with a mini-batch size increasing from 1 till a value such that the computation time reaches twice the SLO. I-PROF periodically re-trains the cold-start model after appending new data (device features).

Furthermore, I-PROF creates a personalized model for every new device model (e.g., Galaxy S7) and employs it for every following request coming from this particular model. I-PROF bootstraps the new model with the first request (for which the cold-start model is used to estimate the computation time). For all the following learning tasks that result in pairs of $(\boldsymbol{x}^{(k)}, \alpha^{(k)})$, I-PROF incrementally updates a Passive-Aggressive (PA) model [47] as: $\boldsymbol{\theta}_{\text{prof}}^{(k+1)} = \boldsymbol{\theta}_{\text{prof}}^{(k)} + \frac{f^{(k)}}{\|\boldsymbol{x}^{(k)}\|^2} \boldsymbol{v}^{(k)}$ where $\boldsymbol{v}^{(k)} = sign(\alpha^{(k)} - \boldsymbol{x}^{(k)\,T} \boldsymbol{\theta}_{\text{prof}}^{(k)}) \boldsymbol{x}^{(k)}$ denotes the update direction, and $f$ the loss function:

$$f(\boldsymbol{\theta}_{\text{prof}}, \boldsymbol{x}, \alpha) = \begin{cases} 0 & \text{if } |\boldsymbol{x}^T \boldsymbol{\theta}_{\text{prof}} - \alpha| \leq \epsilon \\ |\boldsymbol{x}^T \boldsymbol{\theta}_{\text{prof}} - \alpha| - \epsilon & \text{otherwise.} \end{cases} \tag{3.2}$$

The parameter $\epsilon$ controls the sensitivity to prediction error and thereby the aggressiveness of the regression algorithm, i.e., the smaller the value of $\epsilon$ the larger the update for each new data instance (more aggressive).

I-PROF focuses solely on the time and energy spent during an SGD computation. Despite network costs (in particular when transferring models) having also an important impact, they fall outside the scope of this work as one can rely on prior work [8, 112, 144] to estimate the time and energy of network transfers within FLEET.

### 3.2.3 Adaptive Stochastic Gradient Descent

The server-driven synchronous training of Standard FL is not suitable for Online FL, as the latter requires frequent updates and needs to exploit contributions from all workers, including

[3] CPU time spent by processes executing in user or kernel mode.

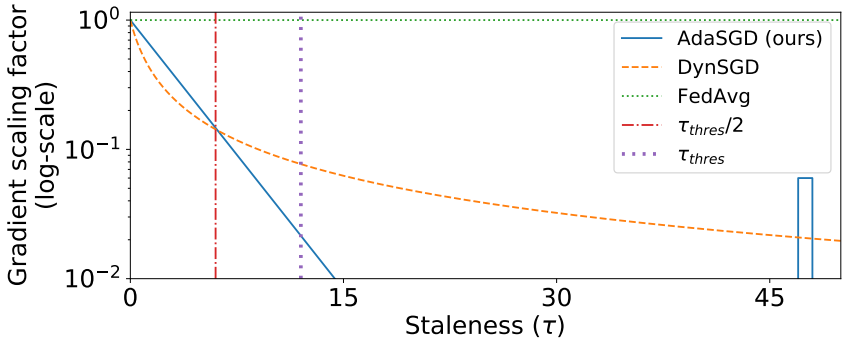



**Figure 3.5 –** Gradient scaling schemes of SGD algorithms. AdaSGD, proposed in this paper, dampens stale gradients with an exponentially decreasing function ($\Lambda(\tau)$) based on the expected percentage of non-stragglers ($\tau_{\text{thres}} :=$ $s$-th percentile of staleness values), and boosts the gradient of the straggler ($\tau = 48$) due to its low similarity ($sim(\boldsymbol{x}_i)$).

slow ones (§5.1). Therefore, we introduce AdaSGD, an asynchronous learning algorithm that is robust to stale updates. AdaSGD is responsible for aggregating the gradients sent by the workers and updating the application model ($\boldsymbol{\theta}_{\text{app}}$)[4]. Each update takes place after AdaSGD receives $K$ gradients. The aggregation parameter $K$ can be either fixed or based on a time window (e.g., update the model every 1 hour). The model update is:

$$\boldsymbol{\theta}_{\text{app}}^{(t+1)} = \boldsymbol{\theta}_{\text{app}}^{(t)} - \gamma_t \sum_{i=1}^{R} \min\left(1, \Lambda(\tau_i) \cdot \frac{1}{sim(\boldsymbol{x}_i)}\right) \cdot \boldsymbol{G}(\boldsymbol{\theta}_{\text{app}}^{(t_i)}, \xi_i) \tag{3.3}$$

where $\gamma_t$ is the learning rate, $t \in \mathbb{N}$ denotes the global logical clock (or step) of the model at the server (i.e., the number of past model updates) and $t_i \leq t$ denotes the logical clock of the model that the worker receives from the server. $\boldsymbol{G}(\boldsymbol{\theta}_{\text{app}}^{(t_i)}, \xi_i)$ is the gradient computed by the client w.r.t the model parameters $\boldsymbol{\theta}_{\text{app}}^{(t_i)}$ on the mini-batch $\xi_i$ drawn uniformly from the local dataset $\boldsymbol{x}_i$.

The workers send gradients asynchronously that can result in *stale* updates. The *staleness* of the gradient ($\tau_i := t - t_i$) shows the number of model updates between the model pull and gradient push of worker $i$. One option is to directly apply this gradient, at the risk of slowing down or even completely preventing convergence [93, 185]. The Standard FL algorithm (FedAvg [119]) simply drops stale gradients. However, even if computed on a stale model, the gradient may incorporate potentially valuable information. Moreover, in FLeet, the gradient computation may drain energy directly from the battery of the phone, thus making the result even more valuable. Therefore, AdaSGD utilizes even stale gradients without jeopardizing the learning process, by multiplying each gradient with an additional weight to the learning rate. This weight consists of (a) a dampening factor based on the staleness ($\Lambda(\tau_i)$) and (b) a boosting factor based on the user's data novelty ($\frac{1}{sim(\boldsymbol{x}_i)}$), that we describe in the following.

[4]Not to be confused with the model of the profiler ($\theta_{\text{prof}}$).

**Staleness-based dampening.** ADASGD builds on prior work on staleness-aware learning that has shown promising results [93, 185]. In order to accelerate learning, ADASGD relies on a system parameter: *the expected percentage of non-stragglers* (denoted by $s$%). We highlight that this value is not a hyperparameter that needs tuning for each ML application but a system parameter that solely depends on the computing and networking characteristics of the workers, while it can be adapted dynamically [136, 142]. We define the staleness-aware dampening factor $\Lambda(\tau) = e^{-\beta\tau}$, with $\beta$ chosen s.t. $\frac{1}{\frac{\tau_{\text{thres}}}{2}+1} = e^{-\beta\frac{\tau_{\text{thres}}}{2}}$ (i.e., the inverse dampening function [93] intersects with our exponential dampening function in $\frac{\tau_{\text{thres}}}{2}$), where $\tau_{\text{thres}}$ is the $s$-th percentile of past staleness values. Figure 3.5 shows the dampening factor of ADASGD compared to the inverse dampening function (employed by DYNSGD [93]). Our hypothesis is that the perturbation to the learning process introduced by stale gradients, increases exponentially and not linearly with the staleness. We empirically verify the superior performance of our exponential dampening function compared to the inverse in §3.3.2.

As a quantile, $\tau_{\text{thres}}$ is estimated from the staleness distribution. In practice, for the past staleness values to be representative of the actual distribution, an initial bootstrapping phase can employ the dampening factor of DYNSGD. After this phase, the service provider can set $s$% and deploy ADASGD. An underestimate of $s$% will slow down convergence, whereas an overestimate may lead to divergence. As we empirically observe (§3.3.1), the staleness distribution often has a long tail. In such cases, the best choice of $s$% is the one that sets $\tau_{\text{thres}}$ at the beginning of the tail.

**Similarity-based boosting.** In the presence of stragglers with large delays (comparing to the mean latency), staleness can grow and drive $\Lambda(\tau)$ close to 0, i.e., almost neglect the gradients of these stragglers. Nevertheless, these gradients may contain valuable information. In particular, they may be computed on data that are not similar to the data used by past gradients. Hence, ADASGD boosts these gradients by using the following similarity value:

$$sim(\boldsymbol{x}_i) = BC(\boldsymbol{LD}(\boldsymbol{x}_i), \boldsymbol{LD}_{global}) \tag{3.4}$$

where $BC$ denotes the Bhattacharyya coefficient [17], and $\boldsymbol{LD}$ the label distribution, that captures the importance of each gradient. We choose this coefficient given our constraints ($sim(\boldsymbol{x}_i) \in [0,1]$). For instance, given an application with 4 distinct labels and a local dataset ($\boldsymbol{x}_i$) that has 1 example with label 0, and 2 examples with label 1: $\boldsymbol{LD}(\boldsymbol{x}_i) = [\frac{1}{3}, \frac{2}{3}, 0, 0]$. The global label distribution ($\boldsymbol{LD}_{global}$) is computed on the aggregate number of previously used samples for each label. We highlight that $\boldsymbol{LD}$ is not specific to classification ML tasks; for regression tasks, $\boldsymbol{LD}$ would involve a histogram, with the length of the $\boldsymbol{LD}$ vector being equal to the number of bins instead of the number of classes.

The similarity value essentially captures how valuable the information of the gradient is. For instance, if a gradient is computed on examples of an unseen label (e.g., a very rare animal), then its similarity value is less than 1 (i.e., has information not similar to the current knowledge

of the model). For the similarity computation, the server needs only the indices of the labels of the local datasets without any semantic information (e.g., label 3 corresponds to "dogs").

### 3.2.4 Implementation

The server of FLEET is implemented as a web application (deployed on an HTTP server) and the worker as an Android library. The server transfers data with the workers via Java streams by using Kryo [98] and Gzip. In total, FLEET accounts 26913 Java LoC, 3247 C/C++ LoC and 1222 Python LoC.

**Worker runtime.** We design the worker of our middleware (FLEET) as a library and execute it only when the overlying ML application (Figure 3.2) is running in the foreground. Since Android is a UI-interactive operating system, background applications have low priority so their access to system resources is heavily restricted and they are likely to be killed by the operating system to free resources for the foreground running app. Therefore, allowing the worker to run in the background would make its performance very unpredictable and thus impact the predictions of I-PROF.

We build our main library for Convolutional Neural Networks in C++ on top of FLEET. We employ (i) the Java Native Interface (JNI) for the server, (ii) the Android NDK for the worker, (iii) an encoding scheme for transferring C++ objects through the java streams, and (iv) a thread-based parallelization scheme for the independent gradient computations of the worker. On recent mobile devices that support NEON [156], FLEET accelerates the gradient computations by using SIMD instructions. We also port a popular deep learning library (DL4J [63]) to FLEET, to benefit from its rich ecosystem of ML algorithms. However, as DL4J is implemented in Java, we do not have full control over the resource allocation.

FLEET relies on the developer of the overlying ML application to ensure the performance isolation between the running application and the worker runtime. The worker can execute in a window of low user activity (e.g., while the user is reading an article) to minimize the impact of the overlying ML application on the predictive power of I-PROF.

**Resource allocation.** Allocating system resources is a very challenging task given the latency and energy constraints of mobile devices [62, 124]. Our choice of employing only stock Android without root access means we can only control which cores execute the workload on the worker, with no access, for instance, to low-level advanced tuning. Given this limited control and the inherent mobile device heterogeneity, we opt for a simple yet effective scheme for allocating resources.

This scheme schedules the execution only on the "big" cores for ARM big.LITTLE architectures and on all the cores otherwise. In the case of computationally intensive tasks (such as the learning tasks of FLEET), big cores are more energy efficient than LITTLE cores because they

finish the computation much faster [81]. Regarding ARMv7 symmetric architectures with 2 and 4 cores that equip older mobile devices, the energy consumption per workload is constant regardless of the number of cores: a higher level of parallelism will consume more energy but the workload will execute faster. For this reason, our allocation policy relies on all the available cores so that we can take advantage of the embarrassingly parallel nature of the gradient computation tasks. For such tasks, we empirically show (§3.3.4) that this scheme outperforms more complex alternatives [124].

**Controller thresholds.** In practice, the service provider can adopt various approaches to define the size and similarity thresholds of the controller (Figure 3.2). One option is A/B testing along with the gradual increase of the thresholds. In particular, the system initializes the thresholds to zero and divides the users into two groups. The first group tests the impact of the mini-batch size and the second the impact of the label similarity. Both groups gradually increase the thresholds until the impact on the service quality is considered acceptable. The server can execute this A/B testing procedure periodically, i.e., reset the thresholds after a time interval. We empirically evaluate the impact of these thresholds on prediction quality in §3.3.5.

## 3.3 Evaluation

Our evaluation consists of two main parts. First, in §3.3.1, we evaluate the claim that Online FL holds the potential to deliver better ML performance than Standard FL [21] for applications that employ data with high temporality (§5.1). Second, we evaluate in more detail the internal mechanisms of FLEET, namely ADASGD (§3.3.2), I-PROF (§3.3.3), the resource allocation scheme (§3.3.4) and the controller (§3.3.5).

We deploy the server of FLEET on a machine with an Intel Xeon X3440 with four CPU cores, 16 GiB RAM and 1 Gb Ethernet, on Grid5000 [82]. The workers are deployed on a total of 40 different mobile phones that we either personally own or belong to the AWS Device Farm [11] (Oregon, USA). In §3.3.1, we deploy the worker on a Raspberry Pi 4 as our hashtag recommender is implemented on TensorFlow that does not yet support training on Android devices.

### 3.3.1 Online VS Standard Federated Learning

We compare Online with Standard FL on a Twitter hashtag recommender. Tweepy [169] enables us to collect around 2.6 million tweets spanning across 13 successive days and located in the west coast of the USA. We preprocess these tweets (e.g., remove automatically generated tweets, remove special symbols) based on [60]. We then divide the data into shards, each spanning 2 days, and divide each shard into chunks of 1 hour. We finally group the data into mini-batches based on the user id.

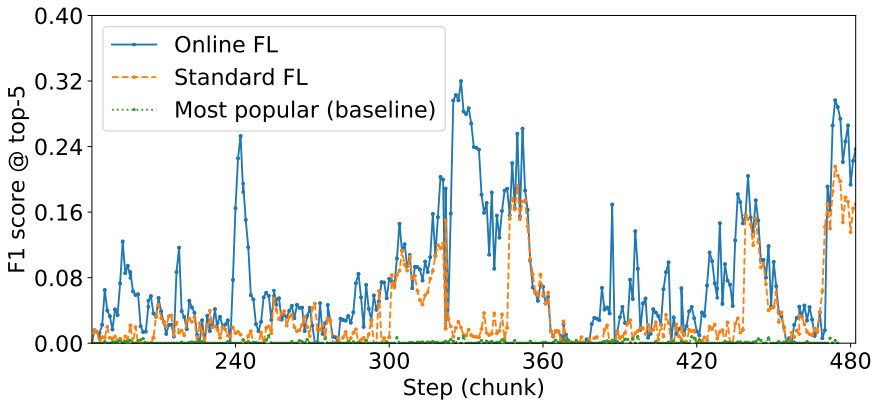
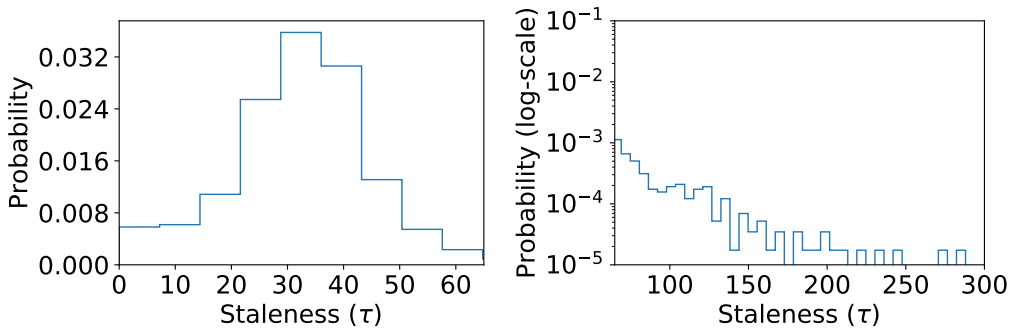



**Figure 3.6 –** Online FL boosts Twitter hashtag recommendations by an average of 2.3× comparing to Standard FL.

Our training and evaluation procedure follows an Online FL setup. Our model is a basic Recurrent Neural Network implemented on TensorFlow with 123,330 parameters [79], that predicts the 5 hashtags with the largest values on the output layer. The model training consists of successive gradient-descent operations, with each gradient derived from a single mini-batch (i.e., sent by a single user). For the Online FL setup, the model is updated every 1 hour. Training uses the data of the previous hour and testing uses the data of the next hour. For the Standard FL setup, the model is updated every day. Training uses the data of the previous day and testing uses the data of the next day. We highlight that under this setup, the two approaches employ the same number of gradient computations and the difference lies only in the time they perform the model updates. We also compare against a baseline model that always predicts the most popular hashtags [97, 135]. We evaluate the model on the data of each chunk and reset the model at the end of each shard.

**Quality boost.** For assessing the quality of the hashtag recommender, we employ the F1-score @ top-5 [78, 97] to capture how many recommendations were used as hashtags (precision) and how many of the used hashtags were recommended (recall). In particular, for each tweet in the evaluation set, we compare the output of the recommender (top-5 hashtags) with the actual hashtags of the tweet, and derive the F1-score. Figure 3.6 shows that Online FL outperforms Standard FL in terms of F1-score, with an average boost of 2.3×. Online FL updates the model in a more timely manner, i.e., soon after the data generation time, and can thus better predict (higher F1-score) the new hashtags than Standard FL. The performance of the baseline model is quite low as the nature of the data is highly temporal [101].

**Energy impact.** We measure the energy impact of the gradient computation on the Raspberry Pi worker. The Raspberry Pi has no screen; nevertheless recent trends in mobile/embedded processor design show that the processor is dominating the energy consumption, especially for compute intensive workloads such as the gradient




**(a)** **(b)**

**Figure 3.7 –** Staleness distribution of collected tweets follows a Gaussian distribution ($\tau < 65$) with a long tail ($\tau > 65$).

computation [84]. We measure the power consumption of every update of Online FL by executing the corresponding gradient computation 10 times and by taking the median energy consumption. We observe that the power depends on the batch size and increases from 1.9 Watts (idle) to 2.1 Watts (batch size of 1) and to 2.3 Watts (batch size of 100). The computation latency is 5.6 seconds for batch size of 1 and 8.4 for batch size of 100. Across all the updates of Online FL (that employ various batch sizes and result in the quality boost shown in Figure 3.6), we measure the average, median, 99$^{\text{th}}$ percentile and maximum values of the daily energy consumption as 4, 3.3, 13.4 and 44 mWh respectively. Given that most modern smartphones have battery capacities over 11000 mWh, we argue that Online FL imposes a minor energy consumption overhead for boosting the prediction quality.

**Staleness distribution.** We study the staleness distribution of the updates on our collected tweets, in order to set our experimental setup for evaluating AdaSGD (§3.3.2). We assume that the round-trip latency per model update (gradient computation time plus network latency) follows an exponential distribution (as commonly done in the distributed learning literature [4, 67, 104, 126]). The network latency for downloading the model (123,330 parameters) and uploading the gradients is estimated to 1.1 second for 4G LTE and 3.8 seconds for 3G HSPA+ [89]. We then estimate the average computation latency to be 6 seconds, based on our latency measurements on the Raspberry Pi. Therefore, we choose the exponential distribution with a minimum of $6 + 1.1 = 7.1$ seconds and a mean of $\frac{(6+1.1)+(6+3.8)}{2} = 8.45$ seconds. Given the exponential distribution for the round-trip latency and the timestamps of the tweets, we observe (in Figure 3.7) that the staleness follows a Gaussian distribution with a long tail (as assumed in [185]). The long tail is due the presence of certain peak times with hundreds of tweets per second.

**Table 3.1 –** CNN parameters.

| Dataset | Parameters | Input | Conv1 | Pool1 | Conv2 | Pool2 | FC1 | FC2 | FC3 |
|---|---|---|---|---|---|---|---|---|---|
| MNIST | Kernel size<br>Strides | 28×28×1 | 5×5×8<br>1×1 | 3×3<br>3×3 | 5×5×48<br>1×1 | 2×2<br>2×2 | 10 | – | – |
| E-MNIST | Kernel size<br>Strides | 28×28×1 | 5×5×10<br>1×1 | 2×2<br>2×2 | 5×5×10<br>1×1 | 2×2<br>2×2 | 15 | 62 | – |
| CIFAR-100 | Kernel size<br>Strides | 32×32×3 | 3×3×16<br>1×1 | 3×3<br>2×2 | 3×3×64<br>1×1 | 4×4<br>4×4 | 384 | 192 | 100 |

### 3.3.2 ADASGD Performance

We now dissect the performance of ADASGD via an image classification application that involves Convolutional Neural Networks (CNNs). We choose this benchmark due to its popularity for the evaluation of SGD-based approaches [2, 39, 94, 119, 185, 186]. We employ multiple scenarios involving various staleness distributions, data distributions, and a noise-based differentially private mechanism.

**Image classification setup.** We implement the models shown in Table 3.1 in FLEET[5] to classify handwritten characters and colored images. We use three publicly available datasets: MNIST [128], E-MNIST [44] and CIFAR-100 [42]. MNIST consists of 70,000 examples of handwritten digits (10 classes) while E-MNIST consists of 814,255 examples of handwritten characters and digits (62 classes). CIFAR-100 consists of 60,000 colour images in 100 classes, with 600 images per class. We perform min-max scaling as a pre-processing step for the input features.

We split each dataset into *training* / *test sets*: 60,000 / 10,000 for MNIST, 697,932 / 116,323 for E-MNIST and 50,000 / 10,000 for CIFAR-100. Unless stated otherwise, we set the aggregation parameter $R$ (§3.2.3) to 1 (for maximum update frequency), the mini-batch size to 100 examples [130], the $\epsilon$ (the Passive-Aggressive parameter) to 0.1 and the learning rate to $15*10^{-4}$ for CIFAR-100, $8*10^{-4}$ for E-MNIST, and $5*10^{-4}$ for MNIST.

Since the training data present on mobile devices are typically collected by the users based on their local environment and usage, both the size and the distribution of the training data will typically heavily vary among users. Given the terminology of statistics, this means that the data are not Independent and Identically Distributed (*non-IID*). Following recent work on FL [172, 173, 181, 186], we employ a non-IID version of MNIST. Based on the standard data decentralization scheme [119], we sort the data by the label, divide them into shards of size equal to $\frac{60000}{2*\text{number of users}}$, and assign 2 shards to each user. Therefore, each user will contain examples for only a few labels.

[5]We implement the CNN for E-MNIST on DL4J and the rest on our default CNN library.

**Staleness awareness setup.** To be able to precisely compare AdaSGD with its competitors, we control the staleness of the updates produced by the workers of FLeet. Based on [185] and the shape of the staleness distribution shown in Figure 3.7, we employ Gaussian distributions for the staleness with two setups: $D1 := \mathcal{N}(\mu = 6, \sigma = 2)$ and $D2 := \mathcal{N}(\mu = 12, \sigma = 4)$, to measure the impact of increasing the staleness. We set the expected percentage of non-stragglers ($s$%) to 99.7%, i.e., $\tau_{\text{thres}} = \mu + 3\sigma$. Appendix A.1 contains more information regarding the mechanism that controls the staleness. We evaluate the SGD algorithms on FLeet by using commercial Android devices from AWS.

We evaluate the performance of AdaSGD against three learning algorithms: (i) DynSGD [93], a staleness-aware SGD algorithm employing an inverse dampening function ($\Lambda(\tau) = \frac{1}{\tau+1}$), that AdaSGD builds upon (§3.2.3), (ii) the standard SGD algorithm with synchronous updates (SSGD) that represents the ideal (*staleness-free*) convergence behaviour, and (iii) FedAvg [119], the standard *staleness-unaware* SGD algorithm that is based on gradient averaging.

**Staleness-based dampening.** Figure 3.8 depicts that AdaSGD outperforms the alternative learning schemes for the non-IID version of MNIST. As expected, the staleness-free scenario (SSGD) delivers the fastest (ideal) convergence, whereas the *staleness-unaware* FedAvg diverges. The comparison between the two staleness-aware algorithms (DynSGD and AdaSGD) shows that our solution (AdaSGD) better adapts the dampening factor to the noise introduced by stale gradients (§3.2.3). AdaSGD reaches 80% accuracy 14.4% faster than DynSGD for $D1$ and 18.4% for $D2$. Figure 3.8 also depicts the impact of staleness on DynSGD and AdaSGD. We observe that the larger the staleness, the slower the convergence of both algorithms. The advantage of AdaSGD over DynSGD grows with the amount of staleness as the larger amount of noise gives more leeway to AdaSGD to benefit from its superior dampening scheme.

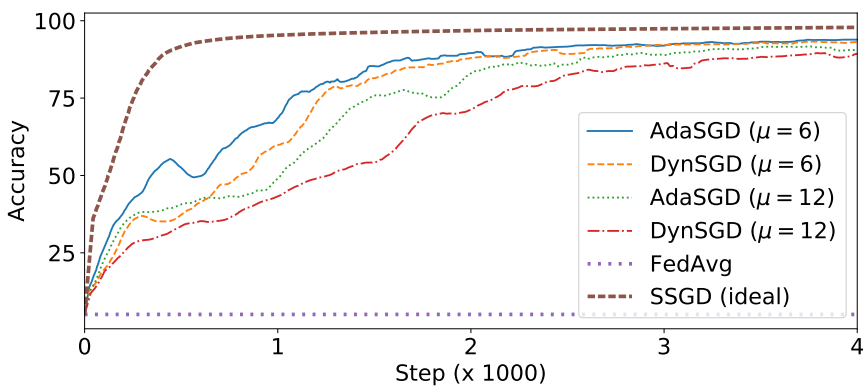
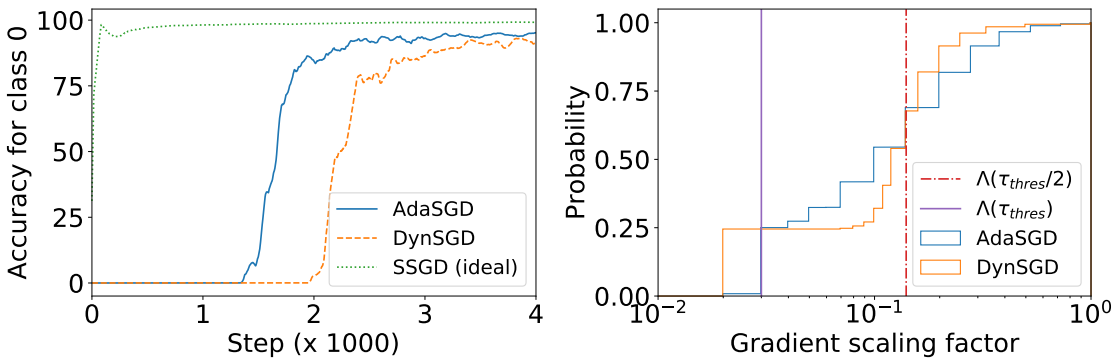



**Figure 3.8 –** Impact of staleness on learning.

**Similarity-based boosting.** We evaluate the effectiveness of the similarity-based boosting property of AdaSGD (§3.2.3) in the case of long tail staleness (Figure 3.7). We employ the non-IID MNIST dataset, $D1$ (thus $\tau_{\text{thres}}$ is 12) and set the staleness to $4 \cdot \tau_{\text{thres}} = 48$ for all the gradients computed on data with class 0. This setup essentially captures the case where a particular label is only present in stragglers. Figure 3.9(a) shows that AdaSGD incorporates the knowledge from class 0 much faster than DynSGD.

Figure 3.9(b) shows the CDF for the dampening values used to weight the gradients of Figure 3.9(a). We mark the two points of interest regarding the $\tau_{\text{thres}}$ by vertical lines (as also shown in Figure 3.5). If AdaSGD had no similarity-based boosting, all updates related to class 0 would almost not be taken into account, as they would be nullified by the exponential dampening function, therefore leading to a model with poor predictions for this class. Given the low class similarity of the learning tasks involving class 0, AdaSGD boosts their dampening value. The second vertical line denotes the staleness value ($\frac{\tau_{\text{thres}}}{2} = 6$) for which AdaSGD and DynSGD give the same dampening value (0.14). The slope of each curve at this point indicates that the dampening values for DynSGD are more concentrated whereas the ones for AdaSGD are more spread around this value.

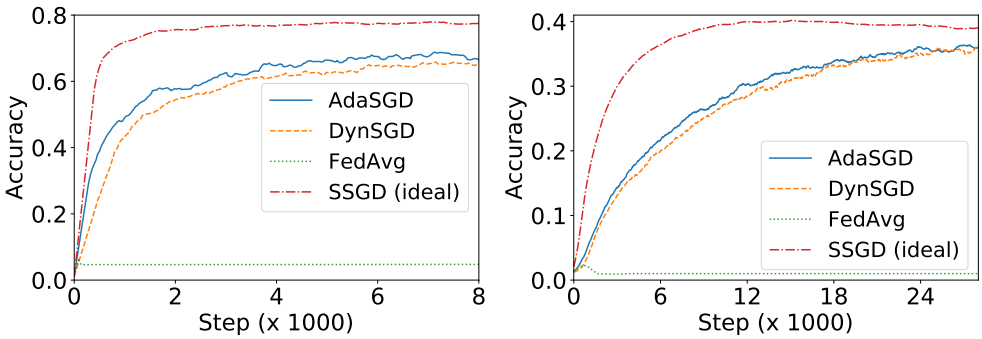



**(a)** **(b)**

**Figure 3.9 –** Impact of long tail staleness on learning.

**IID data.** Although data are more likely to be non-IID in an FL environment, the data collected on mobile devices might in some cases be IID. We thus benchmark AdaSGD under two additional datasets (E-MNIST and CIFAR-100) with the staleness following $D2$. Figure 3.10 shows that our observations from Figure 3.8 hold also with IID data. As with non-IID data, FedAvg diverges also in the IID setting, and AdaSGD performs better than DynSGD on both datasets.

**Differential privacy.** Differential privacy [68] is a popular technique for privacy-preserving FL with formal guarantees [22]. We thus compare AdaSGD against DynSGD in a differentially




**(a)** E-MNIST **(b)** CIFAR-100

**Figure 3.10 –** Staleness awareness with IID data.

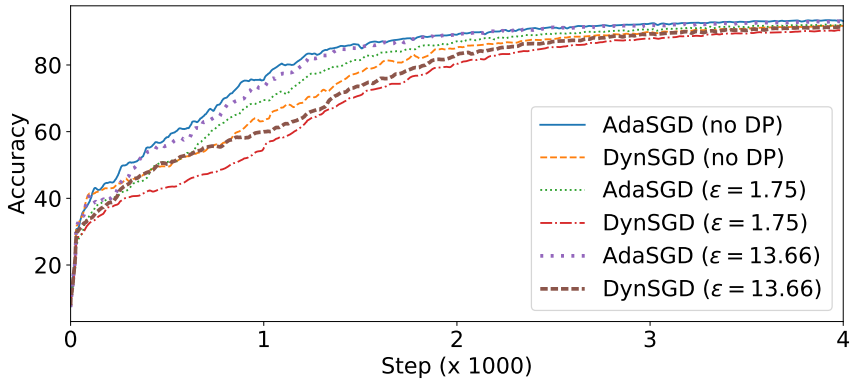


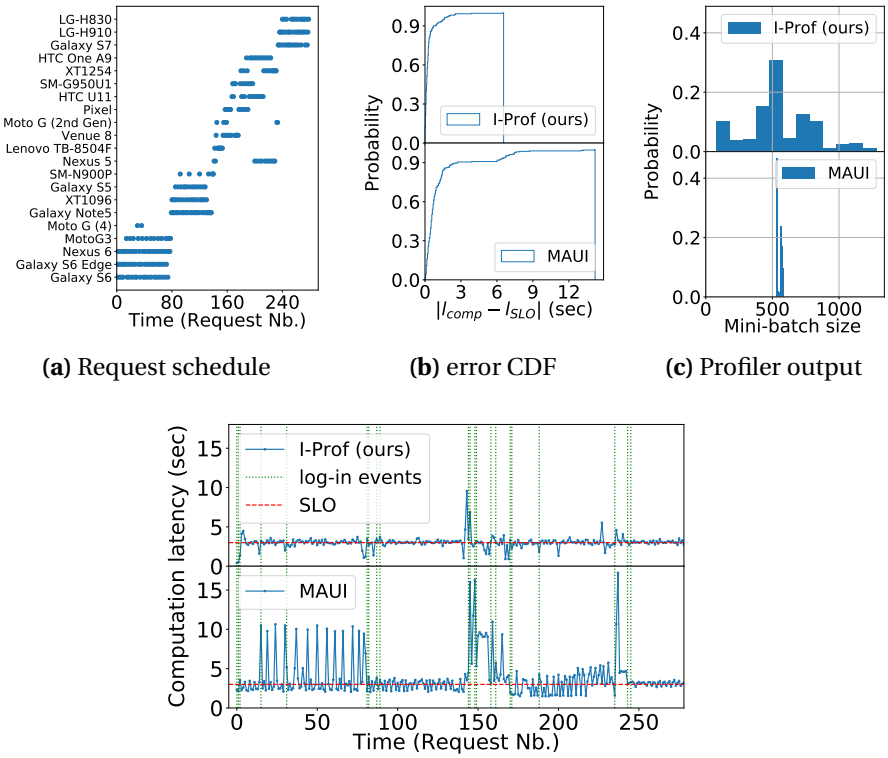

**Figure 3.11 –** Staleness awareness with differential privacy.

private setup by perturbing the gradients as in [2]. We keep the previous setup (IID data with $D2$) and employ the MNIST dataset. Based on [175], we fix the probability $\delta = 1/N^2 = \frac{1}{60000^2}$ and measure the privacy loss ($\epsilon$) with the *moments accountant* approach [2] given the sampling ratio ($\frac{\text{mini-batch size}}{N} = \frac{100}{60000}$), the noise amplitude, and the total number of iterations.

Figure 3.11 demonstrates that the advantage of AdaSGD over DynSGD also holds in the differentially private setup. A better privacy guarantee (i.e., smaller $\epsilon$) slows down the convergence for both staleness-aware learning schemes.

### 3.3.3 I-Prof Performance

We compare I-Prof against the profiler of MAUI [48], a mobile device profiler aiming to identify the most energy-consuming parts of the code and offload them to the cloud. MAUI predicts the energy by using a linear regression model (similar to the global model of I-Prof) on the number of CPU cycles ($\hat{E} = \theta_0 \cdot L$), to essentially capture how the size of the workload affects the energy (as in [127]). We adapt the profiler of MAUI to our setup by replacing the




**(a)** Request schedule **(b)** error CDF **(c)** Profiler output

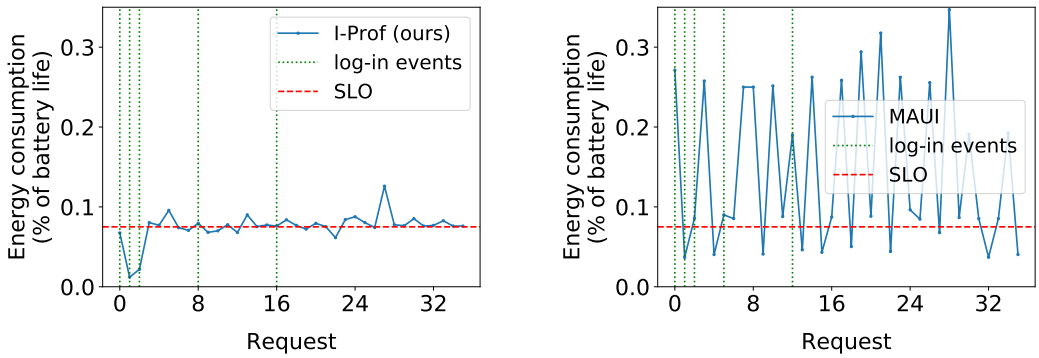
**(d)** Request computation time

**Figure 3.12 –** I-PROF outperforms MAUI and drives the computation time closer to the SLO.

CPU cycles with the mini-batch size for two main reasons. First, our workload has a static code path so the number of CPU cycles on a particular mobile device is directly proportional to the mini-batch size. Second, measuring the number of executed CPU cycles requires root access that is not available on AWS.

We bootstrap the global model of I-PROF and the model of MAUI by pre-training on a training dataset. To this end, we use 15 mobile devices in AWS (that are different from the ones used for the rest of the experiments), assign them learning tasks with increasing mini-batch size until the computation time becomes 2 times the SLO, and collect their device information for each task. We rely on the same methodology to evaluate energy consumption but use only 3 mobile devices in our lab, as AWS prohibits energy measurements.

For testing, we use a different set of 20 commercial mobile devices in AWS, each performing requests for the image classification application (on MNIST), starting at different timestamps (log-in events) as shown in Figure 3.12(a). In order to ensure a precise comparison with MAUI, we add a *round-robin dispatcher* to the profiler component which alternates the requests from a given device between I-PROF and MAUI.




**(a)** I-PROF (ours) **(b)** MAUI

**Figure 3.13 –** I-PROF outperforms MAUI and drives the energy closer to the SLO.

**Computation time SLO.** Figure 3.12(b) shows that I-PROF largely outperforms MAUI in terms of deviation from the computation time SLO. 90% of approximately 280 learning tasks deviate from an SLO of 3 seconds by at most 0.75 seconds with I-PROF and 2.7 seconds with MAUI. This is the direct outcome of our design decisions. First, I-PROF adds dynamic features (e.g., the temperature of the device) to train its global model (§3.2.2). As a result, the predictions are more accurate for the first request of each user. Second, I-PROF uses a personalized model for each device that reduces the error (deviation from the SLO) with every subsequent request (Figure 3.12(d)). Figure 3.12(c) shows that the personalized models of I-PROF are able to output a wider range of mini-batch sizes that better match the capabilities of individual devices. On the contrary, MAUI relies on a simple linear regression model which has acceptable accuracy for its use-case but is inefficient when profiling heterogeneous mobile devices.

**Energy SLO.** To assess the ability of I-PROF to also target the energy SLO, we use the same setup as for the computation time, except on 5 mobile devices[6]. We configure I-PROF with a significantly smaller error margin, $\epsilon = 6 * 10^{-5}$ (Equation 3.2), because the linear relation (capture by $\alpha$ as defined in §3.2.2) is significantly smaller for the energy than for the computation time (as shown in Figure 3.4).

Figure 3.13 shows that I-PROF significantly outperforms MAUI in terms of deviation from the energy SLO. 90% of 36 learning tasks deviate from an SLO of 0.075% battery drop by at most 0.01% for I-PROF and 0.19% for MAUI. The observation that I-PROF is able to closely match the latency SLO, while MAUI suffers from huge deviations, holds for the energy too. The PA personalized models are able to quickly adapt to the state of the device as opposed to the linear model of MAUI that provides biased predictions.

[6]AWS prohibits energy measurements so we only rely on devices available in our lab, listed in their log-in order: Honor 10, Galaxy S8, Galaxy S7, Galaxy S4 mini, Xperia E3.

**Table 3.2 –** Performance of CALOREE [124] on new devices.

| Running device | Deadline error (%) |
|---|---|
| Galaxy S7 | 1.4 |
| Galaxy S8 | 9 |
| Honor 9 | 46 |
| Honor 10 | 255 |

### 3.3.4 Resource Allocation

We evaluate our resource allocation scheme (§3.2.4) and compare it against CALOREE [124] which is a state of the art resource manager for mobile devices. The goal of CALOREE is to optimize resource allocation in order for the workload execution to meet its predefined deadline while minimizing the energy consumption. To this end, CALOREE profiles the target device by running the workload with different resource configurations (i.e., number of cores, core frequency). Since FLEET executes on non-rooted mobile devices, we can only adapt the number of big/little cores (but not their frequencies). By varying the number of cores allocated to our workload (i.e., gradient computation), we obtain the energy consumption of each possible configuration. From these configurations, CALOREE only selects those with the optimal energy consumption (the lower convex hull) which are packed in the so called *performance hash table* (PHT).

**CALOREE on new devices.** In their thorough evaluation, the authors of CALOREE used the same device for training and running the workloads. Therefore, we first benchmark the performance of CALOREE when running on new devices. We employ Galaxy S7 to collect the PHT and set the mini-batch size that I-PROF gives for a latency SLO of 3 seconds (§3.3.3). We then run this workload with CALOREE on different mobile devices, as shown in Table 3.2.

The performance of CALOREE degrades significantly when running on a different device than the one used for training. The first line of Table 3.2 shows the baseline error when running on the same device. The error increases more than 6× for a device with similar architecture and the same vendor (Galaxy S8) and more than 32× for a device of similar architecture but different vendor (Honor 9 and 10). This significant increase for the error is due to the heterogeneity of the mobile devices which make PHTs not applicable across different device models.

**CALOREE vs FLEET.** We evaluate the resource allocation scheme of FLEET by comparing it to the ideal environment for CALOREE, i.e., training and running on the same device (a setup nevertheless difficult to achieve in FL with millions of devices). Following the setup used for the energy SLO evaluation (§3.3.3), we employ 5 devices and fix the size of the workload (mini-batch size) based on the output of I-PROF. In particular we set the mini-batch size

to 280, 4320, 6720, 5280, 1200 for the devices shown in Figure 3.14 respectively. We set the deadline of CALOREE either equal or double than the computation latency of FLEET. We take 10 measurements and report on the median, 10th and 90th percentile.

Figure 3.14 shows the fact that in the ideal environment for CALOREE and even with double the time budget (giving more flexibility to CALOREE), FLEET has comparable energy consumption. Since gradient computation is a compute intensive task with high temporal and spacial cache locality, the configuration changes performed by CALOREE negatively impact the execution time and cancel any energy saved by its advanced resource allocation scheme. Additionally, the fewer configuration knobs available on non-rooted Android devices limit the full potential of CALOREE.

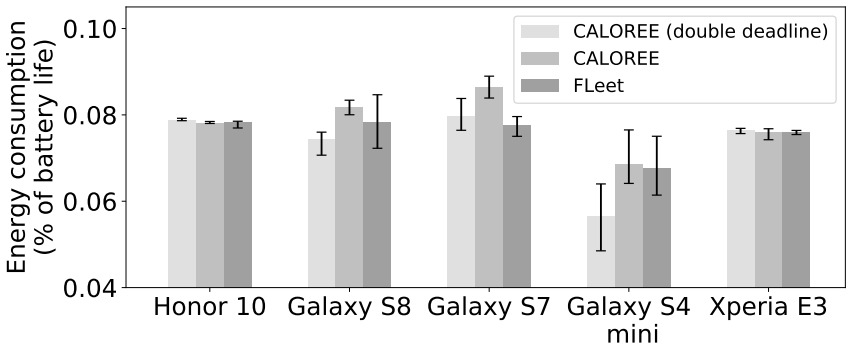


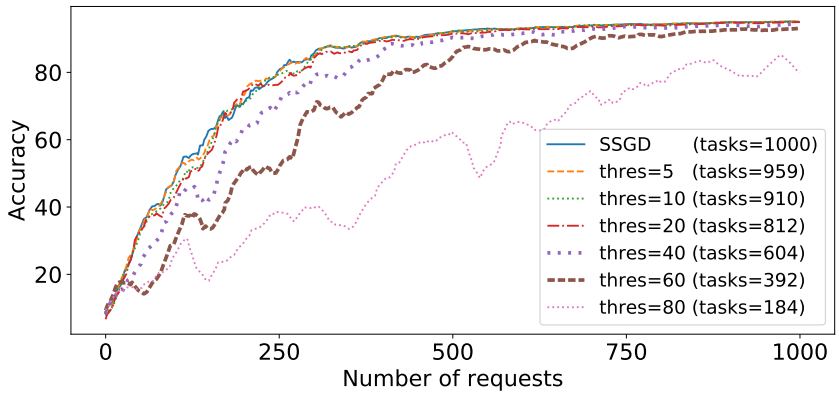

**Figure 3.14 –** Resource allocation of FLEET vs. CALOREE.

### 3.3.5 Learning Task Assignment Control

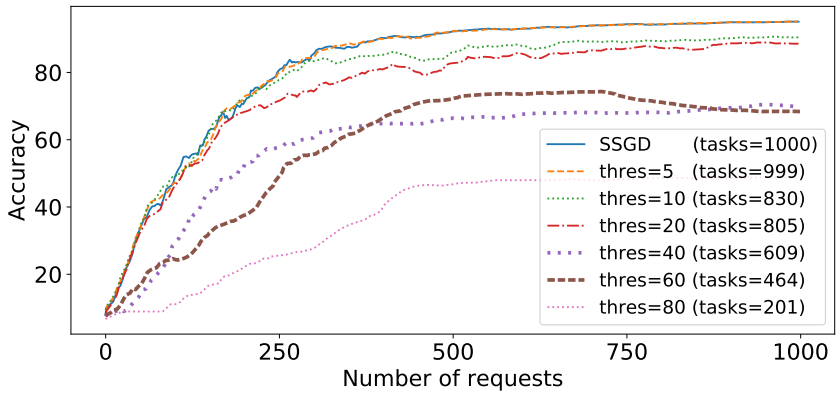
The controller of FLEET employs a threshold to prune learning tasks and thus control the trade-off between the cost of the gradient computations and the model prediction quality. This threshold can be based either on the mini-batch size or on the similarity values (Figure 3.2). To evaluate this trade-off, we employ non-IID MNIST with the mini-batch size following a Gaussian distribution $\mathcal{N}(\mu = 100, \sigma = 33)$ (based on the distribution of the output of I-PROF shown in Figure 3.12(c)), and set the threshold to the $n-$th percentile of the past values. Figure 3.15 illustrates that a threshold on the mini-batch size is more effective in pruning the less useful gradient computations than a threshold on the similarity. Figure 3.15(a) shows that even dropping up to 39.2% of the gradients (with the smallest mini-batch size) has a negligible impact on the accuracy (less than 2.2%). Figure 3.15(b) shows that one can drop 17% of the most similar gradients with an accuracy impact of 4.8%.

## 3.4 Related Work

**Distributed ML.** Adam [39] and TensorFlow [1] adopt the parameter server architecture [105] for scaling out on high-end machines, and typically require cross-worker communication.




**(a)** Based on mini-batch size




**(b)** Based on similarity

**Figure 3.15 –** Threshold-based pruning.

FLEET also follows the parameter server architecture, by maintaining a global model on the server. However, FLEET avoids cross-worker communication, which is impractical for mobile workers due to the device churn.

A common approach for large-scale ML is to control the amount of staleness for boosting convergence [49, 145]. In Online FL, staleness cannot be controlled as this would impact the model update frequency. The workers perform learning tasks asynchronously with end-to-end latencies that can differ significantly (due to device heterogeneity and network variability) or even become infinite (user disconnects).

Petuum [177] and TensorFlow handle faults (worker crashes) by checkpointing and repartitioning the model across the workers whenever failures are detected. In a setting with mobile devices, such failures may appear very often, thus increasing the overhead for checkpointing and repartitioning. FLEET does not require any fault-tolerance mechanism for its workers, as from a global learning perspective, they can be viewed as stateless.

**Federated learning.** In order to minimize the impact on mobile devices, Standard FL algorithms [21, 92, 119, 158] require the learning task to be executed only when the devices are idle, plugged in, and on a free wireless connection. However, in §3.3.1, we have shown that these requirements may drastically impact the performance of some applications. Noteworthy, FL techniques such as co-distillation [92] are orthogonal to the online characteristic, so they can be adapted for ADASGD, and plugged into FLEET.

**Performance prediction for mobile devices.** Estimating the computation time or energy consumption of an application running on a mobile device is a very broad area of research. Existing approaches [28, 40, 41, 85, 100, 180] target multiple applications generally executing on a single device. They typically benchmark the device or monitor hardware and OS-level counters that require root access. In contrast, FLEET targets a single application executing in the same way across a large range of devices. I-PROF poses a negligible overhead, as it employs features only from the standard Android API to enable Online FL, and requires no benchmarking of new devices. I-PROF is designed to make predictions for *unknown* devices.

Neurosurgeon [94] is a scheduler that minimizes the end-to-end computation time of inference tasks (whereas FLEET focuses on training tasks), by choosing the optimal partition for a neural network and offloading computations to the cloud. The profiler of Neurosurgeon only uses workload-specific features (e.g., number of filters or neurons) to estimate computation time and energy, and ignores device-specific features. By contrast, mobile phones, as targeted by I-PROF[7], exhibit a wide range of device-specific characteristics that significantly impact their latency and energy consumption (Figure 3.4).

Systems such as CALOREE [124] and LEO [125] profile mobile devices under different system configurations and train an ML model to determine the ones that minimize the energy consumption. They rely on a control loop to switch between these configurations such that the application does not miss the preset deadline. Due to the restrictions of the standard Android API, the available knobs are limited in our setup. For our application (i.e., gradient computation), we show that a simple resource allocation scheme (§3.2.4) is preferable even in comparison with an ideal execution model.

## 3.5 Concluding Remarks

This paper presented FLEET, the first system that enables *online* ML at the edge. FLEET employs I-PROF, a new *ML-based profiler* which determines the ML workload that each device can perform within predefined energy and computation time SLOs. FLEET also makes use of ADASGD, a new *staleness-aware learning* algorithm that is optimized for Online FL. We showed the performance of I-PROF and ADASGD on commercial Android devices with popular benchmarks. In our performance evaluation we do not focus on network and scalability

[7] In their in-depth experimental evaluation the authors of [94] consider a single hardware platform and not Android mobile devices.

aspects that are orthogonal to our work and addressed in existing literature. We also highlight that transferring the label and device information (Figure 3.2) poses a negligible network overhead compared to transferring the relatively large FL learning models.

Although we believe FLEET to represent a significant advance for online learning at the edge, there is still room for improvement. First, for the energy prediction, I-PROF requires access to the CPU usage that is considered as a security flaw on some Android builds and thus not exposed to all applications. In this case, I-PROF requires a set of additional permissions that belong to services from Android Runtime. Second, the transfer of the label distribution from the worker to the server introduces a potential privacy leakage. However, we highlight that the server has access only to the indices of the labels and not their values. In this paper, we focus on the protection of the input features and mention the possibility to deactivate the similarity-based boosting feature of ADASGD in the case that this leakage is detrimental. We plan to investigate noise addition techniques for bounding this leakage [68] in our future work. Finally, theoretically proving the convergence of ADASGD is non-trivial due to the unbounded staleness and the non Independent and Identically Distributed (non-IID) datasets among the workers. In this respect, a dissimilarity assumption similar to [108] may facilitate the derivation of the proof.

# Byzantine Failures Part III

# 4 Synchronous Stochastic Gradient Descent

## 4.1 Introduction

Traditionally, Byzantine resilience of a distributed service is achieved by using *Byzantine-resilient state machine replication* techniques [29, 33, 46, 152, 153]. Whereas this approach looks feasible for the single and deterministic server, applying it to workers (edge devices) appears less realistic. Indeed, the workers *parallelize*, i.e., share the computational-heavy and sensitive-data-dependent part of SGD: the gradient estimation. Having them compute different gradients (inherently non-deterministic) to agree on only one of their estimations, would imply losing a significant amount of the computations made, i.e., losing the very purpose of the distribution of the estimation.

Some theoretical approaches have been recently proposed to address Byzantine resilience without replicating the workers [20, 61]. In short, the idea is to use more sophisticated forms of aggregation[1] (e.g. *median*) than simple averaging. Despite their provable guarantees, most of these algorithms only ensure a *weak* form of resilience against Byzantine failures. These algorithms indeed ensure convergence to some state, but this final state could be heavily influenced by the Byzantine workers [72]. For most critical distributed ML applications, a *stronger* form of Byzantine resilience is desirable, where SGD would converge to a state that could have been attained in a non-Byzantine environment. Draco [37] and BULYAN [72] are the only proposals that guarantee strong Byzantine resilience. On the one hand, Draco requires (a) computing several gradients per worker and step (instead of one), and (b) strong assumptions as we discuss in §4.4. On the other hand, BULYAN internally *iterates over* a weak Byzantine GAR (e.g. Krum [20]), which is experimentally shown by its authors to be sub-optimal. Apart from Draco, none of the Byzantine-resilient approaches has been implemented and tested in a realistic distributed ML environment to assess the scalability.

We present AGGREGATHOR, a light and fast framework that brings Byzantine resilience to distributed machine learning. Due to its popularity and wide adoption, we build

[1] We refer to the various forms of gradient aggregation as *Gradient Aggregation Rules (GAR)*.

AGGREGATHOR around TensorFlow[2]. Our framework can thus be used to distribute, in a secure way, the training of any ML model developed for TensorFlow. AGGREGATHOR simplifies the experimentation on large and possibly heterogeneous server farms by providing automatic, policy-based device selection and cluster-wide allocation in TensorFlow. Following the TensorFlow design, any worker (including Byzantine ones) can alter the graph and execute code on any other node. We provide a code patch for TensorFlow that prohibits such a behavior to ensure Byzantine resilience.

AGGREGATHOR allows for both levels of robustness: *weak* and *strong* Byzantine resilience (§2.4) through MULTI-KRUM and BULYAN respectively. MULTI-KRUM assigns a *score* (based on a sum of distances with the closest neighbors) to each gradient a worker submits to the server, and then returns the average of the smallest scoring gradients set. BULYAN robustly aggregates $n$ vectors by iterating several times over a second (underlying) Byzantine-resilient GAR. In each loop, BULYAN extracts the gradient(s) selected by the underlying GAR, computes the closest values to the coordinate-wise median of the extracted gradient(s) and finally returns the coordinate-wise average of these values. We further discuss these two components in §4.2.3.

We optimize our implementation of BULYAN given MULTI-KRUM as the underlying GAR. We accelerate the execution by removing all the redundant computations: MULTI-KRUM performs the distance computations only on the first iteration of BULYAN; the next iterations only update the scores. We also parallelize the loops over the gradient coordinates (e.g. median coordinate-wise). We reduce the memory cost by allocating space only for one iteration of MULTI-KRUM along with the intermediate selected gradients. Both of our implementations support non-finite (i.e., $\pm$*Infinity* and *NaN*) coordinates, which is a crucial feature when facing actual malicious workers.

We evaluate and compare the performance of AGGREGATHOR against vanilla TensorFlow when no attacks occur. We first deploy both systems on top of the default, reliable communication protocol and quantify the respective overhead of MULTI-KRUM and BULYAN, i.e., the cost of weak and strong Byzantine resilience, to 19% and 43% respectively. We then consider an unreliable UDP-based communication channel leading to packet losses, to which TensorFlow is intolerant. We provide the necessary TensorFlow modifications to accommodate this lossy scheme. We show that AGGREGATHOR can also tolerate unreliable communication with a speedup gain of six times against vanilla TensorFlow in saturated networks. We also compare AGGREGATHOR with Draco and show a performance gain in terms of throughput and convergence rate.

The rest of this chapter is structured as follows. We describe the design of AGGREGATHOR in §4.2 and empirically evaluate its effectiveness in §4.3. We review the related work in §4.4 and conclude in §4.5.

[2]TensorFlow officially supports FL in a simulation environment [164].

The code of AGGREGATHOR is available at: https://github.com/LPD-EPFL/AggregaThor.

## 4.2 Design of AGGREGATHOR

Our goal is two-fold: First we target faster development and testing of robust, distributed training in TensorFlow; that essentially boils down to providing ease-of-use and modularity. Second we want to enable the deployment of Byzantine-resilient learning algorithms outside the academic environment.

### 4.2.1 Threat Model

We assume the standard synchronous parameter server model [105], with the only dissimilarity being that $f < n$ of the $n$ *workers* are controlled by an *adversary*. We refer to these workers as *Byzantine*. The goal of the adversary is to impair the learning, by making it converge to a state different from the one that would have been obtained if no adversary had stymied the learning process.

We assume the adversary, in the instance of the $f$ cooperating Byzantine workers, has unbounded computational power, and arbitrarily fast communication channels between its $f$ workers and with the parameter server. We assume an asynchronous network (see §4.2.4) and we assume that even Byzantine workers send gradients at each step[3]. We assume the adversary has access to the full training dataset $B$, and the gradients computed by each correct worker.

Finally, we assume, as in [20, 72, 176, 179], that the parameter server is correct. This server could be reliable and implemented on a trustworthy machine, unlike workers that could be remote user machines in the wild. This server could also be implemented by using standard Byzantine-resilient state machine replication [29, 33, 152].

### 4.2.2 Architecture

AGGREGATHOR is a light framework (as shown in Figure 4.1) that handles the distribution of the training of a TensorFlow neural network *graph* over a cluster of machines. One of our main contributions is that this distribution is robust to Byzantine cluster nodes, in a proportion that depends on the GAR used.

In TensorFlow, Byzantine resilience cannot be achieved solely through the use of a Byzantine-resilient GAR. Indeed, TensorFlow allows any node in the cluster to execute arbitrary operations anywhere in the cluster. A single Byzantine worker could then continually overwrite

[3]The default behavior of TensorFlow is to wait indefinitely for non-responding remote nodes, which is incompatible with asynchrony and Byzantine workers (not responding on purpose).

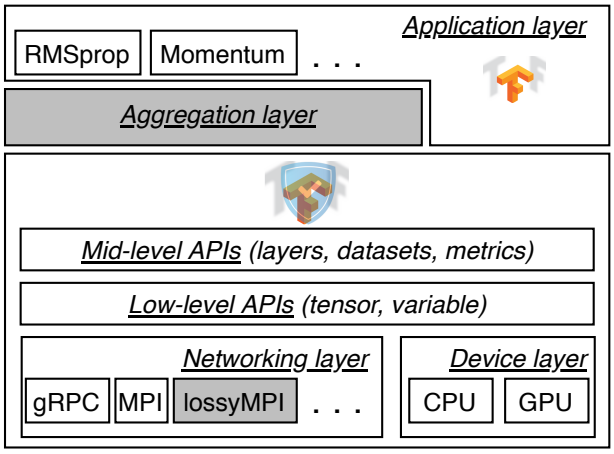



**Figure 4.1 –** The components of AGGREGATHOR, and their layered relations with existing components. New components have a gray background. AGGREGATHOR acts as a light framework based on TensorFlow, that manages the deployment and execution of a *model training session* over a cluster of machines.

the shared parameters[4] with arbitrary values. We overcome this issue in two steps: (a) by patching TensorFlow to make *tf.train.Server* instances belonging to the job named "ps" to discard remote graph definitions and executions, and (b) by using *in-graph replication*, where only the parameter server ("ps") builds the graph (Figure 4.2).

Pursuing our goal of ease-of-use and modularity, we provide the user of our framework with two tools; a *deploy* tool to deploy a cluster through SSH, and a *run* tool to launch a training session on the deployed cluster. Adding a new GAR or a new experiment boils down to (1) adding a python script to a directory, and (2) testing this new component; this consists in changing one or two command line parameters when calling the *run* tool. Cluster-wide device allocation, specifying which operations should run on which devices, is managed by our framework.

### 4.2.3 Byzantine Resilience

AGGREGATHOR ensures Byzantine Resilience by employing a GAR (§2.4). We quantify the cost of using a complex GAR in §4.3.

AGGREGATHOR relies, by default, on two GARs: MULTI-KRUM [20] and BULYAN [72]. The former rule requires that $n \geq 2f+3$ and the second requires that $n \geq 4f+3$. Intuitively, the goal of MULTI-KRUM is to select the gradients that deviate less from the "majority" based on their relative distances. Given gradients $\boldsymbol{G}_1 \ldots \boldsymbol{G}_n$ proposed by workers 1 to $n$ respectively,

[4]This is actually how the distributed example of TensorFlow given on https://www.tensorflow.org/deploy/distributed works: each worker node keeps overwriting the shared parameters.

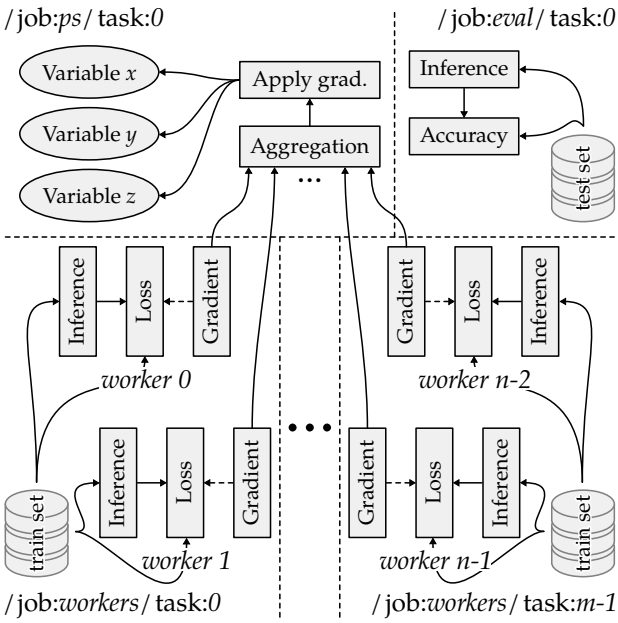



**Figure 4.2 –** High-level components and execution graph. Each gray rectangle represents a group of `tf.Operation`, and each plain arrow represents a `tf.Tensor`. The sub-graph between the gradients to the variables corresponds to Equation 2.7. For readability purpose, the tensors from the variables to each "Inference" and "Gradient" groups of operations have not been represented.

MULTI-KRUM selects the $m$ gradients with the smallest sum of scores (i.e., L2 norm from the other gradients) as follows:

$$(m)\underset{i\in\{1,\ldots,n\}}{\operatorname{argmin}} \sum_{i\to j} \|\boldsymbol{G}_i - \boldsymbol{G}_j\|^2 \tag{4.1}$$

where given a function $X(i)$, $(m)\operatorname{argmin}(X(i))$ denotes the indexes $i$ with the $m$ smallest $X(i)$ values, and $i \to j$ means that $\boldsymbol{G}_j$ is among the $n-f-2$ closest gradients to $\boldsymbol{G}_i$. BULYAN in turn takes the aforementioned $m$ vectors, computes their coordinate-wise median and produces a gradient which coordinates are the average of the $m-2f$ closest values to the median.

In [20], it was proven that Krum (i.e., MULTI-KRUM for $m = 1$) is weakly Byzantine-resilient. Yet, choosing $m = 1$ hampers the speed of convergence [6, 37]. MULTI-KRUM becomes practically interesting when we can chose the highest possible value for $m$ to leverage all the workers (in the limit of no Byzantine workers, this value should be $n$).

An extensive analysis for the Byzantine resilience of MULTI-KRUM and BULYAN is available in [71]. In particular, for $n \geq 2f+3$ and any integer $m$ s.t. $m \leq n-f-2$ MULTI-KRUM ensures weak Byzantine resilience against $f$ failures, and for $n \geq 4f+3$ and any integer $m$ s.t. $m \leq n-2f-2$ BULYAN ensures strong Byzantine resilience against $f$ failures. As a consequence, AGGREGATHOR can safely be used with any value between $1 \leq m \leq n-f-2$.

### 4.2.4 Communication Layer

A Byzantine-resilient GAR at a high-level layer enables the usage of a fast but unreliable communication protocol at the low-level one. Using the vanilla TensorFlow averaging does not work while employing unreliable communication, because lost or shuffled coordinates/packets (of even one gradient) can lead to learning divergence. The most straightforward solution to guarantee convergence in this case, is to drop the whole gradient if at least one coordinate was lost (i.e., the packet containing the coordinate was lost). We expect that such a solution will delay convergence especially in networks with high loss ratios. To avoid dropping the whole gradient in such a case, one can implement a variant of averaging which we call *selective averaging*. In this GAR, the lower layer replaces the lost coordinates with a special value (e.g. *NaN*) while the GAR layer ignores these coordinates while averaging. We expect this method to be faster than the first one. A third solution would be simply to use AGGREGATHOR on top, and put random values at the lost coordinates. Not caring about what happens at the low-level layer would not be harmful, as the Byzantine-resilient GAR on top guarantees convergence (as long as the unreliable communication is deployed only at (up to) $f$ links). Comparing the last two proposed solutions, it is worth noting that using the *selective averaging* model requires a special care for out-of-order packets. A sequence number should define the correct position of each packet so that the received packets are correctly put in their positions (in the gradient). Otherwise, learning convergence is not guaranteed. However, using AGGREGATHOR does not require sending the sequence number because this GAR does not have any assumptions on what is delivered at the lower-level layers.

TensorFlow does not support UDP and hence, we modify its underlying networking layer to support a fast but unreliable communication protocol, which we call *lossyMPI*, alongside those already supported, e.g., gRPC, RDMA, MPI. LossyMPI is devised by modifying the MPI communication endpoints to employ UDP sockets. At the receiving endpoint, we implement the aforementioned GARs to recoup the lost coordinates and guarantee convergence. To use UDP, we also implement a reliability scheme for metadata (accompanying gradients) and packets ordering.

## 4.3 Evaluation of AGGREGATHOR

We evaluate the performance of AGGREGATHOR following a rather standard methodology in the distributed ML literature. In particular, we consider the image classification task due to its wide adoption as a benchmark for distributed ML literature [1, 39, 182].

### 4.3.1 Evaluation Setup

We present the details of the configuration, benchmarks, and methods we employ for our evaluation. For clarity and for the rest of this section, we will refer with MULTI-KRUM to

the deployment of AggregaThor with the GAR being Multi-Krum and with Bulyan to the deployment of AggregaThor with the GAR being Bulyan.

**Platform.** Our experimental platform is Grid5000 [82]. Unless stated otherwise, we employ 20 nodes from the same cluster, each having 2 CPU (Intel Xeon E5-2630) with 8 cores, 128 GiB RAM and 10 Gbps Ethernet.

**Dataset.** We use the CIFAR-10 dataset [42], a widely used dataset in image classification [161, 182], which consists of 60,000 colour images in 10 classes. We perform min-max scaling as a pre-processing step for the input features of the dataset. We employ a convolutional neural network with a total of 1.75M parameters as shown in Table 4.1. We have implemented the same model with PyTorch to be compatible with Draco.

**Table 4.1 –** CNN Model parameters.

| | Input | Conv1 | Pool1 | Conv2 | Pool2 | FC1 | FC2 | FC3 |
|---|---|---|---|---|---|---|---|---|
| Kernel size<br>Strides | 32×32×3 | 5×5×64<br>1×1 | 3×3<br>2×2 | 5×5×64<br>1×1 | 3×3<br>2×2 | 384 | 192 | 10 |

**Evaluation metrics.** We evaluate the performance of AggregaThor using the following standard metrics.

*Throughput.* This metric measures the total number of gradients that the aggregator receives per second. The factors that affect the throughput is the time to compute a gradient, the communication delays (worker receives the model and sends the gradient) and the idle time of each worker. The idle time is determined by the overhead of the aggregation at the server. While the server performs the aggregation and the descent, the workers wait (synchronous training).

*Accuracy.* This metric measures the top-1 cross-accuracy: the fraction of correct predictions among all the predictions, using the *testing* dataset (see below). We measure accuracy both with respect to time and model updates.

**Evaluation scheme.** To cross-validate the performance, we split the dataset into *training* and *test sets*. The dataset includes 50,000 training examples and 10,000 test examples. Note that, if not stated otherwise, we employ an RMSprop optimizer [167] with a fixed initial learning rate of $10^{-3}$ and a mini-batch size of 100.

We split our 20 nodes into $n = 19$ workers and 1 parameter server. If not stated otherwise, we set $f = 4$ given that Bulyan requires $n \geq 4f + 3$.

We employ the best (in terms of convergence rate) combination of other hyper-parameters for the deployment of Draco. For example, we use the repetition method because it gives better results than the cyclic one. Also, we use the reversed gradient adversary model with the same parameters recommended by the authors and a momentum of 0.9.

### 4.3.2 Non-Byzantine Environment

In this section, we report on the performance of our framework in a non-Byzantine distributed setup. Our baseline is *vanilla* TensorFlow (TF) deployed with the built-in averaging GAR: *tf.train.SyncReplicasOptimizer*. We compare TF against AGGREGATHOR using (a) MULTI-KRUM, (b) BULYAN, (c) an alternative Byzantine-resilient median-based algorithm [176] (*Median*) implemented as a new GAR in our framework, and (d) the basic gradient averaging GAR (*Average*). We also report on the performance of (e) Draco.

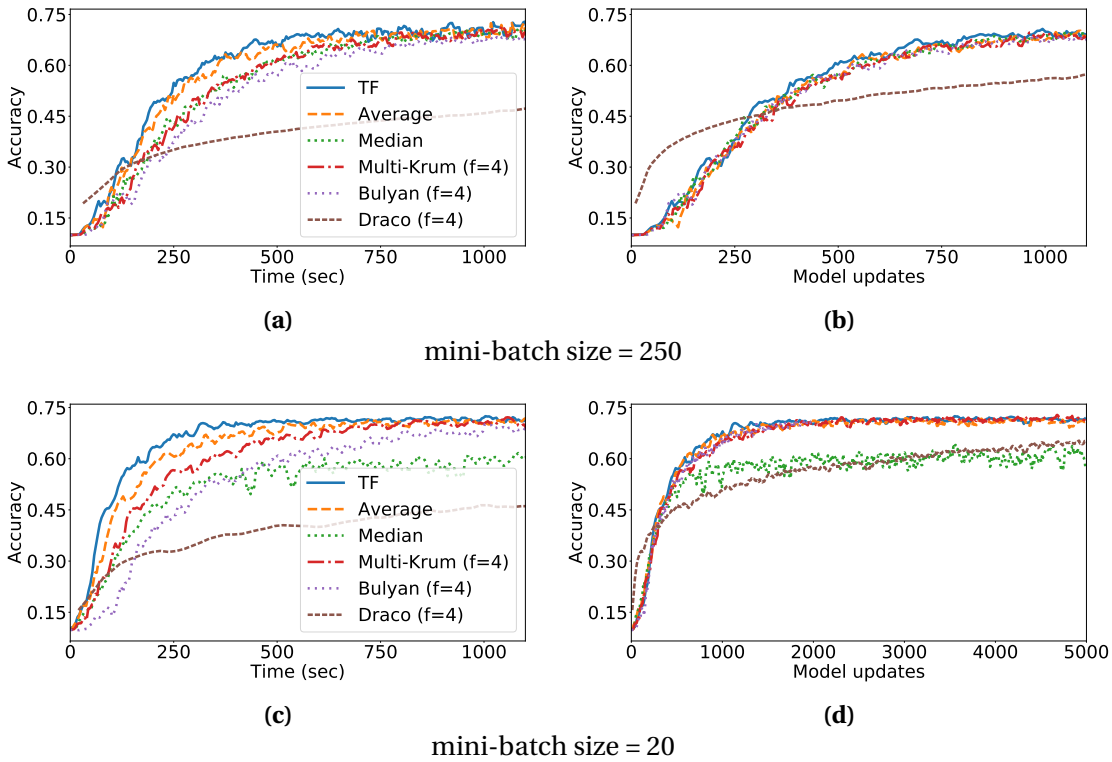



**Figure 4.3 –** Overhead of AGGREGATHOR in a non-Byzantine environment.

**Overhead in terms of convergence time.** In Figure 4.3(a), TensorFlow reaches 50% of its final accuracy in 3 minutes and 9 seconds, whereas MULTI-KRUM and BULYAN are respectively 19% and 43% slower for reaching the same accuracy. Our framework with *Average* leads to a 7% slowdown compared to the baseline. The *Median* GAR, with a mini-batch size of $b = 250$,

converges as fast as the baseline (model update-wise), while with $b = 20$, *Median* prevents convergence to a model achieving baseline accuracy.

We identify two separate causes for the overhead of AggregaThor. The first is the *computational overhead* of carrying out the Byzantine-resilient aggregation rules. The second cause is the inherent *variance increase* that Byzantine-resilient rules introduce compared to *Average* and the baseline. This is attributed to the fact that Multi-Krum, Bulyan and *Median* only use a fraction of the computed gradients; in particular *Median* uses only one gradient. Increasing the variance of the gradient estimation is a cause of convergence slowdown [23]. Since even *Median* converges as fast as the baseline with $b = 250$, the respective slowdowns of 19% and 43% for Multi-Krum and Bulyan correspond *only* to the computational overhead. The practitioner using AggregaThor does not need to increase the mini-batch size to achieve baseline final accuracy (Figure 4.3(d)).

Although Draco reaches the same final accuracy, the time to reach the model's maximal accuracy is slower than with our TensorFlow-based system. We attribute this mainly to the fact that Draco requires $2f + 1$ times more gradients to be computed than our system before performing a step.

We decompose the average latency per update step to assess the effect of the aggregation time on the overhead of AggregaThor against TensorFlow. We employ the same setup as in Figure 4.3(a).

Figure 4.4 shows that the aggregation time accounts for 35%, 27% and 52% of run times of Median, Multi-Krum, and Bulyan respectively. These ratios do not depend on the variance of the aggregated gradients, but solely on the gradient computation time: a larger/more complex model would naturally make these ratios decrease (i.e., the *relative cost* of Byzantine resilience would decrease). See Figure 4.5.

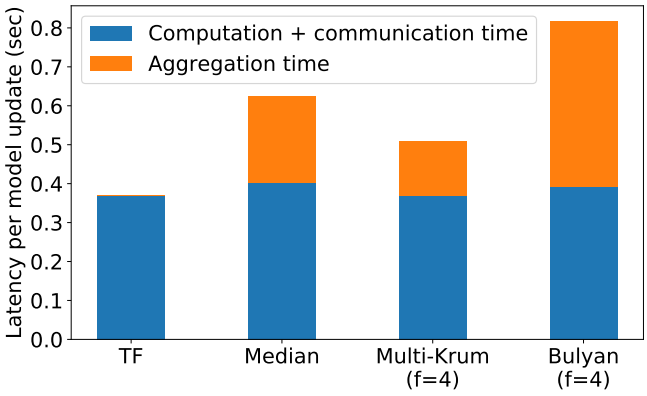
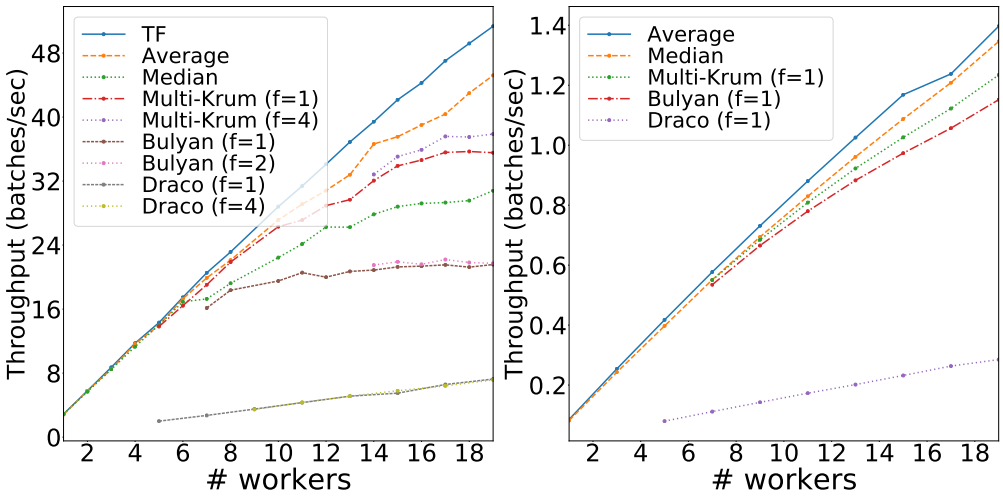



**Figure 4.4 –** Latency breakdown.

**Impact of $f$ on scalability.** We measure the scalability of AggregaThor with respect to TensorFlow for two models: the CNN that we use throughout the evaluation and a significantly larger one, ResNet50. Figure 4.5(a) shows that the throughput of all TensorFlow-based systems with up to 6 workers is the same. From this point on, the larger the number of workers, the larger the deviation between the Byzantine-resilient algorithms and TensorFlow. The reason behind this behavior is the fact that an increase in the number of workers introduces a larger overhead to the aggregation of a Byzantine-resilient algorithm (logic to ensure Byzantine resilience) than simple averaging. The more expensive the logic, the bigger this difference. For example, Bulyan scales poorly for this setup. This is confirmed in Figure 4.5(b) where the gradient computation is significantly more costly than gradient aggregation. This allows Multi-Krum and Bulyan to have better scalability.



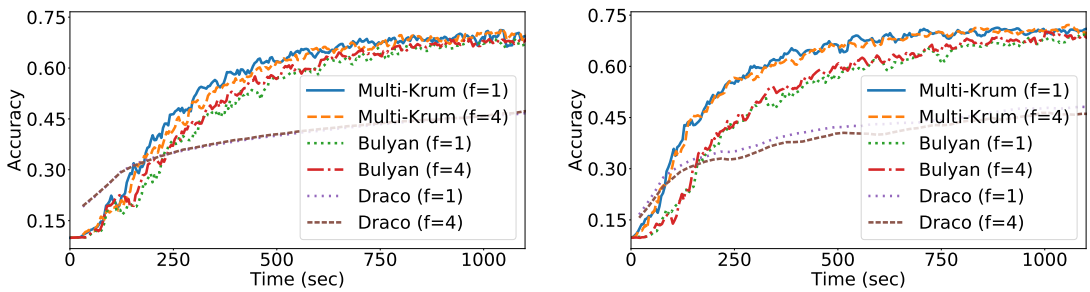

**(a)** CNN **(b)** ResNet50

**Figure 4.5 –** Throughput comparison.

Figure 4.5(a) confirms that the higher the declared $f$ the higher the throughput. This may appear counter-intuitive (resilience against more failures provides a performance benefit) but is the direct outcome of the design of the algorithmic components of AggregaThor. Since $m = n - f - 2$, the higher $f$ the fewer *iterations* for Bulyan [72] and the fewer the neighbors for Multi-Krum [20]. Moreover, for a larger value of $f$, these algorithms become more selective for the gradients that will be averaged. It is however very important to highlight that non-convex optimization is a notably complex problem that might take advantage of more variance to converge faster [71]. Therefore, anticipating faster convergence for a larger value of $f$ does not always hold, i.e., larger throughput does not always lead to faster convergence [23]. In conclusion, there exists a trade-off between the update throughput and the quality of each update that is partially controlled by the choice of $f$.

Draco is always at least one order of magnitude slower than the TensorFlow-based systems. This low throughput limits its scalability. An interesting observation here is that changing the number of Byzantine workers does not have a remarkable effect on the throughput. This is attributed to the method Draco uses to tolerate Byzantine behavior which is linear in $n$ [37], thus the performance is not affected by changing $f$.

**Impact of $f$ on convergence.** We show the effect of the choice of $f$ in a non-Byzantine environment. Figure 4.6(a) shows that the larger value of $f$ triggers a slightly slower convergence for Multi-Krum and slightly faster convergence for Bulyan. This is the direct consequence of the aforementioned trade-off. The throughput of Multi-Krum is boosted more than the throughput of Bulyan for the same increase on $f$ (from 1 to 4). Therefore, in the case of Bulyan, the faster model updates compensate for the additional noise whereas in the case of Multi-Krum the throughput boost is not enough. For a smaller mini-batch size (Figure 4.6(b)) the behaviour is similar but the impact of $f$ is smaller. This is because the mini-batch size is the second (the first is $f$) important parameter that affects the trade-off between update throughput and quality of each update (§2.3.1).




**(a)** mini-batch size = 250 **(b)** mini-batch size = 20

**Figure 4.6 –** Impact of $f$ on convergence.

**Cost analysis.** Our empirical results are consistent with the complexity of the algorithmic components of AggregaThor. The model update time complexity[5] of both Multi-Krum and Bulyan is $\mathcal{O}(n^2 d)$. This is essentially the same as a baseline GAR-based SGD algorithm, i.e., averaging[6] when $d \gg n$ (valid assumption for modern ML). Bulyan induces an additional overhead of $\mathcal{O}(nd)$ (coordinate-wise median) on top of $n-2f$ executions of Multi-Krum, leading to a total model update complexity of $\mathcal{O}(nd + f \cdot nd) = \mathcal{O}(n^2 d)$. We note that $\mathcal{O}(n^2 d)$ is a common bound on the complexity per round for weakly Byzantine-resilient SGD algorithms [20, 72, 162]. AggregaThor is strongly Byzantine-resilient with the same complexity.

[5] We refer to the worst-case time complexity.
[6] Averaging is not Byzantine-resilient.

Baseline averaging SGD requires $\mathcal{O}(\frac{1}{\sqrt{nb}})$ steps to converge. In other words SGD goes as fast as permitted by the square root of the total number of samples used per step. The more samples, the lower the variance and the better the gradient estimation.

Everything else being equal (mini-batch sizes, smoothness of the cost function etc), and in the absence of Byzantine workers, the number of steps required for AggregaThor to converge is $\mathcal{O}(\frac{1}{\sqrt{m}})$. The higher the value of $m$ the fewer steps required for convergence. Therefore, the best choice for $m$ is the largest safe one, i.e., $n-f-2$ for weak and $n-2f-2$ for strong Byzantine resilience (§4.2.3).

### 4.3.3 Byzantine Environment

We now report on our evaluation of AggregaThor in a distributed setting with Byzantine workers. We first report on two forms of *weak* Byzantine behavior, namely corrupted data and dropped packets. Then we discuss the effect of a *stronger* form of Byzantine behavior, drawing the line between Multi-Krum and Bulyan.

**Corrupted data.** Figure 4.7 shows that for a mini-batch size of 250, the convergence behavior of AggregaThor is similar to the ideal one (TensorFlow in a non-Byzantine environment). We thus highlight the importance of Byzantine resilience even for this "mild" form of Byzantine behavior (only one worker sends corrupted data) to which TensorFlow is intolerant (TensorFlow diverges).

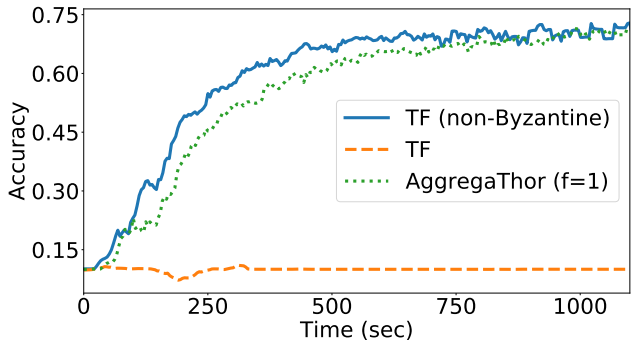



**Figure 4.7 –** Impact of malformed input on convergence.

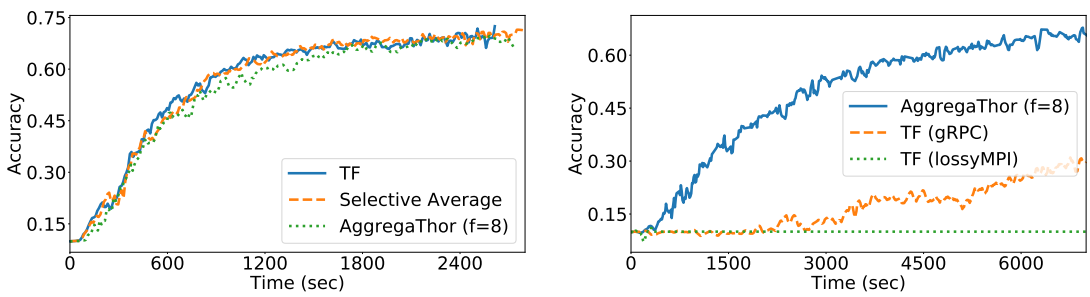
**Dropped packets.** We evaluate the impact of using unreliable communication between the parameter server and $f$ (Byzantine) workers. We also evaluate the performance of alternatives we proposed in §4.2.4 to tolerate the lost, malformed, and out-of-order packets. For this experiment we set $f$ to the maximum possible value given our 19 workers, i.e., we set $f$ to 8

(assuming Multi-Krum). For simplicity, we employ unreliable communication only for the gradient transfer[7].

We assess the effect of unreliable communication in a lossy environment by introducing additional (to the existing ones by the network) network packet drops via the Linux tc tool. We evaluate the performance of AggregaThor in the absence of additional packet drops (0% loss) and in the presence of a drop rate of 10%. This order of magnitude for the drop ratio, although high for a data-center environment, can be realistic in a WAN environment [99] for distributed machine learning [91].

Figure 4.8(a)[8] shows the performance of the three solutions proposed to tolerate unreliability of the communication layer (§4.2.4). The three solutions achieve almost the same performance. This highlights the advantage of using UDP as it mitigates the performance lost by Byzantine resilience. In this environment where no packet loss exists, dropping the whole gradient (while using vanilla TensorFlow) does not have remarkable effect on the learning convergence. We expect to see a delayed convergence for such an algorithm in an environment with a higher loss ratio.

Figure 4.8(b) shows the advantage of using UDP in a lossy environment. It depicts that AggregaThor over lossyMPI converges to 30% accuracy more than 6 times faster than TensorFlow over gRPC, under an artificial 10% drop rate. The main reason behind this big difference in the convergence speed is that the links that employ lossyMPI (between the server and the Byzantine workers) use a rapid mechanism to address the packet drops, i.e., they deliver corrupted messages to which AggregaThor is tolerant. The convergence time for both systems is one order of magnitude larger compared to the environment with no artificial drops. We believe this performance drop is induced by TCP reducing (halving) its transmission rate following packet losses. Finally, Figure 4.8(b) confirms the divergence of TensorFlow, which is non Byzantine-resilient, over lossyMPI.




**(a)** 0% drop rate **(b)** 10% drop rate

**Figure 4.8 –** Impact of dropped packets on convergence.

[7] Our setup can be easily extended to support an unreliable communication for the model transfer without any impact on the conclusions of our evaluation.

[8] TF here drops corrupted gradients as described in §4.2.4.

**Byzantine gradients.** The cost of attacking a non-Byzantine resilient GARs (such as averaging) is the cost for the computation of an estimate of the gradient, i.e., can be done in $O(nd)$ operations per round by a Byzantine worker. This cost is the same as the aggregation cost of the server per update step.

To attack weakly Byzantine-resilient GARs however, such as MULTI-KRUM, one needs to find a legitimate but harmful vector. A harmful vector is a vector that will (i) be selected by a weakly Byzantine-resilient GAR, and (ii) triggers a poor convergence, i.e., a convergence to an optimum that makes the performance of the model (e.g., in terms of accuracy) low in comparison with the one achieved when learning with no Byzantine workers. An attacker must thus first collect the vector of every correct worker (before they reach the server), and solve an optimization problem (with linear regression) to approximate this harmful but legitimate (selected by weakly Byzantine-resilient GAR) vector. If the desired quality of the approximation is $\epsilon$, the Byzantine worker would need at least $\Omega(\frac{nd}{\epsilon})$ operation to reach it with regression. This is a tight lower bound for a regression problem in $d$ dimensions with $n$ vectors [87]. In practice, if the required precision is in the order of $10^{-9}$, a total of 100 workers and a model of dimension $10^9$ would require a prohibitive cost for the attack ($\approx 10^{20}$ operations to be done in each step by the attacker).

To summarize, weak Byzantine resilience can be enough as a practical solution against attackers whose resources are comparable to the ones of the server. If that is the expected setting, we could switch-off BULYAN and use only MULTI-KRUM. However, strong Byzantine resilience, i.e., AGGREGATHOR combining both MULTI-KRUM and BULYAN, remains the provable solution against attackers with significant resources [72].

## 4.4 Related Work

Several weakly Byzantine-resilient algorithms have been proposed as an improvement of the workhorse SGD component in the synchronous non-convex setting. Krum [20] employed a median-like aggregation rule. [179] proposed a median-based and a mean-based aggregation rules (Equation 2.7). [176] evaluated three other median-based aggregation rules under different practical attack scenarios. [162] presented a combination of a median-based over a mean-based aggregation rule. A quorum-based aggregation approach was recently proposed in [6], achieving optimal convergence rates but suitable only for convex machine learning.

Following a different direction, Draco [37] has been the first framework to address the scalability of SGD through redundant gradient computations combined with a specific encoding scheme. In fact Draco is also strongly Byzantine-resilient following our definition, and has the advantage of requiring only $2f+1$ workers (instead of $4f+3$ for BULYAN). It has however a serious practical concern: it can only be used in settings where workers need (at least) an agreement on the ordering of the dataset so that the coding scheme can be agreed on. This violates critical privacy concerns in distributed learning and does not allow learning on private data. For instance, their algorithmic redundancy scheme requires a

comparison between gradients sum provided by different workers, for this comparison to be meaningful, the server needs the incoming gradients to be computed on similar datapoints, therefore hampering any possible use in private and local datasets. AGGREGATHOR in turn only requires the workers to be drawing data independently and identically distributed (but not the same datapoints). Additionally, as pointed out by the authors [37], the encoding and decoding time of Draco can be several times larger than the computation time of ordinary SGD. AGGREGATHOR avoids this overhead along with the redundant computations.

## 4.5 Concluding Remarks

We built AGGREGATHOR, a Byzantine-resilient framework, on top of TensorFlow without adding any constraints to the application development, i.e., AGGREGATHOR exposes the same APIs as TensorFlow. We also corrected an inherent vulnerability of TensorFlow in the Byzantine setting. The overhead of AGGREGATHOR over TensorFlow is moderate when there are no Byzantine failures. In fact, we have also shown that AGGREGATHOR could be viewed as a performance booster for TensorFlow, as it enables the use of an unreliable (and faster) underlying communication protocol, namely UDP instead of TCP, when running through a saturated network.

While designing AGGREGATHOR, we followed the parameter server model [105] and assumed the server is reliable while workers are not. This model has been considered for most theoretical analysis of Byzantine-resilient SGD [6, 20, 37, 54, 72, 176]. An orthogonal problem that should be investigated is the setting where the owner of the system does not trust their servers. In this case, a server could be made Byzantine-resilient using some state machine replication scheme [29, 33, 46, 152, 153]. Essentially, each worker could communicate with the replicas of the server and use the model that has been sent by 2/3 of the replicas. Since the computation in the server (GAR and model update) is deterministic, the correct servers will propose identical models to the workers. Although at first glance simple, we believe that the interplay between the specificity of gradient descent and the state machine replication approach might end up challenging to achieve efficiently.

# 5 Asynchronous Stochastic Gradient Descent

## 5.1 Introduction

Existing Byzantine distributed ML solutions all assume a synchronous model [20, 73, 162], that is highly restrictive for applications with *fast data* (Chapter 1). For each model update, (a) all (honest) workers are supposed to use the exact same model to compute the gradient, and (b) the parameter server waits for a quorum of workers before aggregating their gradients. When networks are asynchronous, exhibiting heterogeneous (sometimes arbitrary) communication delays, synchronous solutions inevitably lead to slow convergence.

On the other hand, Asynchronous SGD algorithms, such as AdaSGD (§3.2.3), enable huge performance benefits despite heterogeneous communication delays [88, 105, 110, 111]. In short, such algorithms (a) allow workers to make use of a stale model and (b) update the model as soon as a new gradient is delivered (instead of waiting for a quorum). Nevertheless, none of these asynchronous algorithms tolerate any Byzantine behavior. In fact, all provably convergent asynchronous SGD algorithms assume that all the workers are permanently honest about their gradient, i.e., provide unbiased estimations of the actual gradient (Figure 5.1).

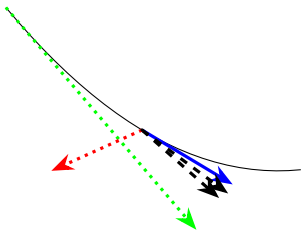


**Figure 5.1 –** The gradients computed by non-stale honest workers (black dashed arrows) are distributed around (and are on average equal to) the actual gradient (solid blue arrow) of the cost function (thin black curve). A Byzantine worker can propose an arbitrary poisoning vector (red dotted arrow). A honest but stale worker computes the correct gradient but for a stale version of the model (long green dotted arrow).

Combining asynchrony with Byzantine resilience is challenging. In particular, aggregating gradients that were computed on different models requires the knowledge of how the curvature of the cost function evolves with staleness. This curvature determines the window of synchrony within which a synchronous method can be transposed into an asynchronous context. Roughly speaking, the more locally curved the cost function is, the narrower this window and vice versa. Estimating the curvature requires heavy computations of the Hessian matrix ($\mathcal{O}(d^2)$), not to mention the fact that this would deprive the parameter server from the most prominent advantage of asynchrony, namely updating the model as soon as a *single* gradient is delivered (i.e., the parameter server would need to aggregate a quorum).

In this chapter, we consider for the first time the situation where a significant fraction of workers ($\frac{f}{n}$) can be Byzantine (arbitrarily adversarial) and consider unbounded communication delays. Such situation corresponds to that of many realistic distributed platforms today. We present the first asynchronous Byzantine gradient descent algorithm, we call KARDAM. KARDAM leverages the Lipschitzness of the cost function to filter out gradients from potentially Byzantine workers, while prohibiting Byzantine workers from flooding the parameter server (which in turn would prevent honest workers from updating the model). KARDAM also uses a dampening scheme that scales each gradient based on its staleness. The computation overhead for each update is negligible as the filtering component of KARDAM is mostly scalar-based. The time complexity for each update computed in terms of the dimension $d$ of a gradient is $\mathcal{O}(d + fn)$. This complexity is the same as the standard complexity of an asynchronous SGD update ($\mathcal{O}(d)$) for the very high-dimensional learning models of today ($d \gg (f, n)$). We prove the convergence of KARDAM and precisely determine its convergence rate. In particular, we prove its self-stabilizing property using a refined version of the global confinement argument [23].

We implemented and deployed KARDAM in a distributed setting and we report in this chapter on its in-depth empirical evaluation on the CIFAR-100 and EMNIST datasets. In particular, we evaluate the overhead of KARDAM with respect to non Byzantine-resilient solutions. KARDAM does not tamper with the learning procedure (i.e., include additional noise), yet it does induce a slowdown that we empirically show to be less than $\frac{f}{n}$, where $f$ is the number of Byzantine failures tolerated and $n$ the total number of workers (we also prove that theoretically). Finally, we show that the dampening component (when plugged onto an asynchronous non Byzantine-resilient SGD solution) outperforms alternative staleness-aware asynchronous competitors in environments with honest workers.

The code to reproduce our experiments as well as a few additional results (varying $f$) will be found at https://github.com/gdamaskinos/kardam.

## 5.2 Setup

We consider the general distributed model for machine learning, namely the parameter server [1, 50, 59, 88, 105, 106, 107][1]. We assume that $f$ of the $n$ workers are Byzantine (behave arbitrarily). Following the traditional assumption in distributed computing, we assume that the identities of the Byzantine workers are unknown whereas $f$ (in practice, an upper bound) is known. Computation is divided into (infinitely many) asynchronous model updates (steps).

**Definition 6** (Time)**.** *The global step (denoted by $t$) represents the global logical clock of the parameter server (or equivalently the number of model updates). The local timestamp (denoted by $t_p$) for a given worker $p$, represents the version (step) of the model that the worker receives from the server and computes the gradient upon. The difference $t - t_p$ can be arbitrarily large due to the asynchrony of the network.*

During each step $t$, the parameter server broadcasts the model $\boldsymbol{\theta}_t \in \mathbb{R}^d$ to all the workers. A cost function $\mathcal{F}$ reflects the quality of the model for the learning task. Each non-Byzantine worker $p$ computes an estimate $\boldsymbol{g}_p = \boldsymbol{G}(\boldsymbol{\theta}_{t_p}, \xi_p)$ of the actual gradient $\boldsymbol{\nabla}\mathcal{F}(\boldsymbol{X}, \boldsymbol{\theta}_{t_p})$ (§2.3.1). Each worker $p$ sends the timestamp $t_p$ (to declare which version of the model it used) and the gradient $\boldsymbol{g}_p$. See §2.1 for notational details.

A Byzantine worker $b$ proposes a gradient $\boldsymbol{g}_b$ which can deviate arbitrarily from $\boldsymbol{G}(\boldsymbol{\theta}_{t_b}, \xi_b)$ (see Figure 5.1). A Byzantine worker may have full knowledge of the system, including the gradients proposed by other workers. Byzantine workers can furthermore collude, as typically assumed in the distributed computing literature [27, 103, 114]. Since the communication is assumed to be asynchronous, the parameter server takes into account the first gradient received at time $t$. The parameter server then either suspects the gradient and ignores it, or employs it to update the model and move to step $t+1$. We make the following assumptions about any honest worker $p$.

**Assumption 1** (Unbiased gradient estimator)**.**

$$\mathbb{E}_{\xi_p}[\boldsymbol{G}(\boldsymbol{\theta}_{t_p}, \xi_p)] = \boldsymbol{\nabla}\mathcal{F}(\boldsymbol{\theta}_{t_p})$$

**Assumption 2** (Bounded variance)**.**

$$\mathbb{E}_{\xi_p}[\|\boldsymbol{G}(\boldsymbol{\theta}_{t_p}, \xi_p) - \boldsymbol{\nabla}\mathcal{F}(\boldsymbol{\theta}_{t_p})\|^2] \leq d\sigma^2$$

Assumptions 1 and 2 are common in the literature [23] and hold if the data used for computing the gradients is drawn uniformly and independently.

**Assumption 3** (Linear growth of $r$-th moment)**.**

$$\mathbb{E}_{\xi_p}[\|\boldsymbol{G}(\boldsymbol{\theta}, \xi_p)\|^r] \leq A_r + B_r\|\boldsymbol{\theta}\|^r \quad \forall \boldsymbol{\theta} \in \mathbb{R}, \quad r = 2, 3, 4$$

[1] Classical techniques of state-machine replication [114] can be used to ensure that the parameter server is reliable.

Assumption 3 translates into "the $r$-th moment of the gradient estimator grows linearly with the $r$-th power of the norm of the model" as assumed in [23].

**Assumption 4** (Lipschitz gradient)**.**

$$||\nabla\mathcal{F}(\boldsymbol{\theta}_1) - \nabla\mathcal{F}(\boldsymbol{\theta}_2)|| \leq K||\boldsymbol{\theta}_1 - \boldsymbol{\theta}_2||$$

**Assumption 5** (Convexity in the horizon)**.** *We require that beyond a certain horizon, $\|\boldsymbol{\theta}\| \geq D$, there exist $\epsilon > 0$ and $0 \leq \beta < \pi/2$ such that $\|\nabla\mathcal{F}(\boldsymbol{\theta})\| \geq \epsilon > 0$ and $\frac{\langle\boldsymbol{\theta},\nabla\mathcal{F}(\boldsymbol{\theta})\rangle}{\|\boldsymbol{\theta}\|\cdot\|\nabla\mathcal{F}(\boldsymbol{\theta})\|} \geq \cos\beta$.*

Assumptions 4 and 5 are the same as in [20], the first is classic, the second is a slight refinement of a similar assumption in [23]. It essentially states that, beyond a certain horizon $D$ in the parameter space, the opposite of the gradient points towards the origin.

**Definition 7** (Byzantine resilience)**.** *Let $\mathcal{F}$ be any cost function satisfying the assumptions above. Let $A$ be any distributed SGD scheme. We say that $A$ is Byzantine-resilient if the sequence $\nabla\mathcal{F}(\boldsymbol{\theta}_t) = 0$ converges almost surely to zero, despite the presence of up to $f$ Byzantine workers.*

## 5.3 KARDAM

In this section, we present the two main components of our algorithm, KARDAM[2], namely the filtering and the dampening components. We also establish the theoretical guarantees of each component. The full corresponding proofs are given in Appendix B.1.

### 5.3.1 Byzantine-resilient Filtering Component

The parameter server accepts a gradient $\boldsymbol{g}_p$ from worker $p$ (i.e., updates the model with $\boldsymbol{g}_p$) if $\boldsymbol{g}_p$ is accepted by the Byzantine-resilient filtering component of KARDAM. This component itself consists of a *Lipschitz filter* followed by a *frequency filter* that we describe in the following.

**Lipschitz filter.** This filter can be viewed as a kinetic validation at the parameter server based on the empirical Lipschitzness.

**Definition 8** (Empirical Lipschitz coefficient)**.** *The empirical Lipschitz coefficient at worker $p$ is defined as $\hat{K}_p = \frac{\|\boldsymbol{g}_p - \boldsymbol{g}_p^{prev}\|}{\|\boldsymbol{\theta}_{t_p} - \boldsymbol{\theta}_{t_p}^{prev}\|}$. The empirical Lipschitz coefficient at the parameter server is defined with respect to a received gradient from worker $p$ and an updated gradient from worker $q$ at the previous step $(t-1)$ as $\hat{K}_t^p = \frac{\|\boldsymbol{g}_p - \boldsymbol{g}_q\|}{\|\boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1}\|}$.*

The empirical Lipschitz coefficient ($\hat{K}_p$) reflects the local empirical observation of the gradient evolution, normalized by the model evolution. Each worker $p$ derives this coefficient between the current and the previous models used to compute the current and previous gradients of $p$ respectively.

[2] KARDAM was a Bulgarian khan who preempted the invasion of the Byzantine empire.

The Lipschitz filter accepts the candidate gradient $\boldsymbol{g}_p$ if the empirical Lipschitzness for $\boldsymbol{g}_p$ (Definition 8) is not suspicious, i.e., if it is smaller than a median empirical Lipschitzness of all the workers as follows.

$$\hat{K}_t^p \leq \hat{K}_t := quantile_{\frac{n-f}{n}}\{\hat{K}_p\}_{p \in P}$$

where $quantile_{\frac{n-f}{n}}$ represents the element that separates the $\frac{n-f}{n}$ fraction of workers with the smallest empirical Lipschitz values from the remaining $\frac{f}{n}$ fraction with the highest values (i.e., the $(100 \cdot \frac{n-f}{n})^{th}$ percentile). We highlight that there exist two honest workers $p_1$ and $p_2$ such that $\hat{K}_{p_1} \leq \hat{K}_t \leq \hat{K}_{p_2}$ since our single dimensional median is guaranteed to be bounded by values from any group of size $n-f$ (i.e., group of honest workers).

The complexity of the Lipschitz filter is $\mathcal{O}(d+n)$ (computing distances on 2 $d$-dimensional vectors, then getting the median of $n$ scalars, in $\mathcal{O}(n)$ with quick-select).

Obviously, the Lipschitz filter will end up filtering fast workers (that reach the more curved regions of the cost functions before the others) or slow workers (that are delayed in a curved region while everyone else is already in a less curved region). We note that this filter, roughly speaking, suspects $f$ workers to be Byzantine and thus a pessimistic choice for $f$ would increase the overhead of KARDAM (filters more gradients due to a pessimistic choice for $f$).

**Theorem 1** (Optimal Slowdown)**.** *We define the slowdown SL as the ratio between the number of updates from honest workers that pass the Lipschitz filter and the total number of updates delivered at the parameter server. We derive the upper and lower bounds of SL in the following.*

$$\frac{n-2f}{n-f} \leq SL \leq \frac{n-f}{n}$$

*The upper and lower bounds are tight and hold when there are $f$ Byzantine workers and no Byzantine workers respectively. Therefore* KARDAM *achieves the optimal bounds with respect to any Byzantine-resilient SGD scheme and $n \approx 3f$ workers.*

*Proof.* Any Byzantine-resilient SGD scheme assuming $f$ Byzantine workers will at most use $\frac{n-f}{n}$ of the total available workers (upper bound). By definition, the Lipschitz filter accepts the gradients computed by $\frac{n-f}{n}$ of the total workers with empirical Lipschitzness below $\hat{K}_t$. If every worker is honest, then the filter accepts gradients from $\frac{n-f}{n}$ of the workers. We thus get the tightness of the upper bound for the slowdown of KARDAM. For the lower bound, the Byzantine workers can know that putting a gradient proposition above $\hat{K}_t$ will get them filtered out and the parameter server will end up using only the honest workers available. The optimal attack would therefore be to slow down the server by getting tiny-Lipschitz gradients accepted while preventing the model from actually changing. This way, the Byzantine workers will make the server filter gradients from a total of $f$ out of the $n-f$ honest workers, leaving only $n-2f$ useful workers for the server. □

**Theorem 2** (Byzantine resilience in asynchrony)**.** *Let $A$ be any distributed SGD scheme. If the maximum successive gradients that $A$ accepts from a single worker and the maximum delay are both unbounded, then $A$ cannot be Byzantine-resilient when $f \geq 1$.*

*Proof.* Without any restrictions, the parameter server could only accept successive gradients from the same Byzantine worker (without getting any update from any honest worker), for example, if the Byzantine worker is faster than any other worker (which is true by the definition of a Byzantine worker and by the fact that delays on (honest) workers are unbounded). This way, the Byzantine worker can force the parameter server to follow arbitrarily bad directions and never converge. Hence, without any restriction on the number of gradients from the workers, we prove the impossibility of asynchronous Byzantine resilience. Readers familiar with distributed computing literature might note that if asynchrony was possible for Byzantine SGD without restricting the number of successive gradients from a single worker, this could be used as an abstraction to solve asynchronous Byzantine consensus (that is impossible to solve [76]). This provides another proof (by contradiction) for our theorem. □

Given Theorem 2 and the objective of making KARDAM Byzantine-resilient in an asynchronous environment (i.e., while letting workers be arbitrarily delayed), we introduce the frequency filter.

**Frequency filter.** The goal of this filter is to limit the number of successive gradients [3] from a single worker to a value of $f$, thus not allowing the Byzantine workers to prevent honest workers from updating the model. Consider $A$ as the list of workers who computed the last $2f$ accepted gradients. Assume that the candidate gradient $\boldsymbol{g}_q$ passes the Lipschitz filter. The frequency filter adds worker $q$ at the end of $A$ (i.e., at position $A[2f+1] = q$). If adding this candidate gradient $\boldsymbol{g}_q$ makes any set of workers of size $f$ appear more than $f$ times in $A$, then $q$ is rejected, otherwise, $q$ is accepted. For each worker $p$, the number of times $p$ appeared in $A$ is denoted by $n_p$. The frequency filter accepts a gradient from worker $p$ if the following holds: $\sum_{p \in \mathcal{S}} n_p \leq f$, where $\mathcal{S}$ denotes the set of $f$ workers with the $f$ maxima of $\{n_p\}_{p=1}^n$. The time complexity of the frequency filter is $\mathcal{O}(2f+1)$ to compute $\{n_p\}_{i=1}^n$ (going through the list $A$ of size $2f+1$), in addition to $\mathcal{O}(fn)$ to find the $f$ maxima among $\{n_p\}_{i=1}^n$.

**Lemma 1** (Limit of successive gradients)**.** *The frequency filter ensures that any sequence of length $2f+1$ consequently accepted gradients contains at least $f+1$ gradients computed by honest workers.*

*Proof.* Given any sequence of $2f+1$ consequently accepted gradients ($A$), we denote by $S$ the set of workers that computed these gradients. The frequency filter guarantees that any $f$ workers in $S$ computed at most $f$ gradients in $A$. At most $f$ workers in $S$ can be Byzantine, thus at least $f+1$ gradients in $A$ are from honest workers. □

[3] One open problem left in our work is the extent to which this filter is too harsh for asynchronous schemes. For instance, it can at least lead to a randomly shuffled round-robin schedule.

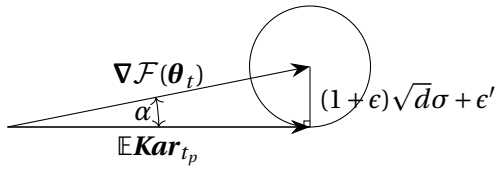



**Figure 5.2 –** Illustration of correct cone. If $\left\|\mathbb{E}\boldsymbol{Kar}_{t_p} - \nabla\mathcal{F}(\boldsymbol{\theta}_t)\right\| \leq (1+\epsilon)\sqrt{d}\sigma + \epsilon'$ then $\langle \mathbb{E}\boldsymbol{Kar}_{t_p}, \nabla\mathcal{F}(\boldsymbol{\theta}_t)\rangle$ is upper bounded by $(1-\sin\alpha)\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\|^2$ where $\sin\alpha = \frac{(1+\epsilon)\sqrt{d}\sigma+\epsilon'}{\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\|}$.

Given asynchrony (unbounded delays), we do not assume any upper bound on the norm of the model, the norm of the gradients or the values of the cost function (regularization schemes can make the loss grow arbitrarily and thereby the gradients norms). However, we assume (as in [23]) that the cost function is lower bounded by a positive scalar. This assumption holds for all the standard cost functions that are at least lower bounded by zero (e.g., square loss, cross-entropy or any norm-based cost). We denote $\boldsymbol{Kar}$ by the sequence of gradients accepted (i.e., not filtered) by KARDAM, and by $\boldsymbol{Kar}_t$ the gradient accepted by KARDAM in step $t$.

**Theorem 3** (Correct cone and bounded statistical moments)**.** *If $n > 3f+1$ then for any $t \geq t_r$ (we show that $t_r \in \mathcal{O}(\frac{1}{K\sqrt{|\xi|}})$ where $|\xi|$ is the batch-size of honest workers):*

$$\mathbb{E}[\|\boldsymbol{Kar}_t\|^r] \leq A'_r + B'_r\|\boldsymbol{\theta}_t\|^r$$

*for any $r = 2,3,4$, constants $A'_r, B'_r$ and*

$$\langle \mathbb{E}[\boldsymbol{Kar}_t], \nabla\mathcal{F}_t\rangle = \Omega(1 - \frac{\sqrt{d}\sigma}{\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\|})\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\|^2$$

*The expectation is on the random samples used for training.*

*Proof.* (Sketch - full proof in Appendix B.1.1) The frequency filter guarantees that there is always an update from a honest worker in any sequence of $f+1$ updates (Lemma 1), i.e., at any time $t$, there is an interval $t-i$ where $i < f+1$ such that the vector that passed the Lipschitz filter is a vector sent by an honest worker (therefore an unbiased estimation of the true gradient). With this in mind, and using triangle inequalities over a series of (at most $f$) previous updates, we prove inequalities on the $r$-th statistical moments of $\boldsymbol{Kar}$. Those inequalities are in turn plugged into the requirements for the (almost sure) global confinement argument of [23].

With the guarantees of almost sure global confinement, and using the Liptschitz properties, and (again) the existence of honest (unbiased) workers in the "recent past" as explained above, we find the lower-bound of the scalar product between the two desired vectors $\langle \mathbb{E}[\boldsymbol{Kar}_t], \nabla\mathcal{F}_t\rangle$ when their distance is small enough compared to their own norms (Figure 5.2). This finally shows that KARDAM remains in a cone of an angle $\alpha$ that is upper bounded by $\arcsin(\frac{(1+\epsilon)\sqrt{d}\sigma+\epsilon'}{\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\|})$ with appropriately chosen $\epsilon$ and $\epsilon'$. □

### 5.3.2 Staleness-aware Dampening Component

We now present the component of KARDAM that enables staleness-aware asynchronous updates for the ML model. For the sake of clarity, we denote the time by $t' := t - tr$ (Theorem 3). We introduce the *R-soft-async* protocol where the server updates the model only after receiving $R$ gradients. The update rule for KARDAM is the following.

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \gamma_t \boldsymbol{Kar}_t \\ &= \boldsymbol{\theta}_t - \gamma_t \sum_{[\boldsymbol{G}(\boldsymbol{\theta}_l,\xi_m),l]\in\mathcal{G}_t} \Lambda(\tau_{tl}) \cdot \boldsymbol{G}(\boldsymbol{\theta}_l,\xi_m)\end{aligned} \tag{5.1}$$

where $\boldsymbol{G}(\boldsymbol{\theta}_l,\xi_m)$ denotes the gradient w.r.t the model parameters $\boldsymbol{\theta}_l$ on the mini-batch $\xi_m$. We assume that every gradient passes the filtering scheme (§5.3.1) at the step $t$. KARDAM requires $|\mathcal{G}_t| = R$ gradients for each update.

The difference between the standard SGD update rule and our Equation 5.1 illustrates how KARDAM handles asynchronous updates. KARDAM dampens each update depending on its staleness value ($\tau_{tl}$). KARDAM employs a decay function $\Lambda(\tau_{tl})$ such that $0 \leq \Lambda \leq 1$ to derive the dampening factor for each distinct value of staleness.

**Definition 9** (Dampening function)**.** *We employ a bijective and strictly decreasing dampening function $\tau \mapsto \Lambda(\tau)$ with $\Lambda(0) = 1$.*[4] *Note that every bijective function is also invertible, i.e., $\Lambda^{-1}(v)$ exists for every $v$ in the range of the $\Lambda$ function.*

Let $\boldsymbol{\Lambda}_t$ be the set of $\Lambda$ values associated with the gradients at timestamp $t$.

$$\boldsymbol{\Lambda}_t = \{\Lambda(\tau_{tl}) \mid [g,l] \in \mathcal{G}_t\}$$

We partition the set $\mathcal{G}_t$ of gradients at timestamp $t$ according to their $\Lambda$-value as follows.

$$\begin{aligned}\mathcal{G}_t &= \bigsqcup_{\lambda\in\boldsymbol{\Lambda}_t} \mathcal{G}_{t\lambda} \\ \mathcal{G}_{t\lambda} &= \{[g,l] \in \mathcal{G}_t \mid \Lambda(\tau_{tl}) = \lambda\}\end{aligned}$$

Therefore, the update equation can be reformulated as follows.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \gamma_t \sum_{\lambda\in\boldsymbol{\Lambda}_t} \lambda \cdot \sum_{[\boldsymbol{G}(\boldsymbol{\theta}_l,\xi),l]\in\mathcal{G}_{t\lambda}} \boldsymbol{G}(\boldsymbol{\theta}_l,\xi)$$

**Definition 10** (Adaptive learning rate)**.** *Given the Lipschitz constant $K$, the total number of timestamps $T$, and the total number of gradients in each timestamp as $M$, we define $\gamma_t$ as follows.*

[4] If $\Lambda(0) = 1$, then there is no decay for gradients computed on the latest version of the model, i.e., $\tau_{tl} = 0$.

$$\gamma_t = \underbrace{\sqrt{\frac{\mathcal{F}(\boldsymbol{\theta}_1) - \mathcal{F}(\boldsymbol{\theta}^*)}{KTMd\sigma^2}}}_{\gamma} \cdot \underbrace{\frac{M}{\sum_{\lambda \in \boldsymbol{\Lambda}_t} \lambda |\mathcal{G}_{t\lambda}|}}_{\phi_t} \tag{5.2}$$

where $\gamma$ is the baseline component of the learning rate and $\phi_t$ is the adaptive component that depends on the amount of stale updates that the server receives at timestamp $t$. Moreover, $\phi_t$ incorporates the total staleness at any timestamp $t$ based on the staleness coefficients ($\lambda$) associated with all the gradients received in timestamp $t$.

$\boldsymbol{\theta}^*$ refers to the (not necessarily global) optimum we are heading to, and on which our adaptive learning rate depends. Assuming this value is known is made just for the sake of a proof, as is usually done in proofs for the speed of convergence of SGD. In practice, one does not need to know $\mathcal{F}(\boldsymbol{\theta}^*)$ and can assume it to be lower bounded [23]. This will produce overshooting (large steps) in the early phases of KARDAM, but will get to small enough step sizes: the baseline part of our adaptive learning rate contains a term $1/T$, where $T$ is the total number of iterations (also unknown before we run SGD). In practice, it is replaced by $1/t$ ($t$: step at the server). This part of our learning rate decreases with $t$ and will compensate for the overshooting described above (overcoming the overshooting in at most $O(1/\mathcal{F}(\boldsymbol{\theta}_1))$ steps).

**Remark 1** (Correct cone)**.** *As a consequence of passing the filter and of Theorem 3, $\boldsymbol{G}$ satisfies the following.*

$$\langle \mathbb{E}_\xi \boldsymbol{G}(\boldsymbol{\theta}, \xi), \boldsymbol{\nabla}\mathcal{F}(\boldsymbol{\theta}) \rangle > \Omega((\|\boldsymbol{\nabla}\mathcal{F}(\boldsymbol{\theta}_t)\| - \sqrt{d}\sigma)\|\boldsymbol{\nabla}\mathcal{F}(\boldsymbol{\theta}_t)\|)$$

The theoretical guarantee for the convergence rate of KARDAM depends on Assumptions 2 and 4 and Remark 1. These assumptions are weaker than the assumptions for the convergence guarantees in [93, 185]. In particular, due to unbounded delays and the potential presence of Byzantine workers, we only assume the unbiased gradient estimator $\boldsymbol{G}(\cdot)$ for honest workers (Assumption 1). We instead employ (Remark 1) the fact that $\boldsymbol{G}(\cdot)$ and $\boldsymbol{\nabla}\mathcal{F}(\boldsymbol{\theta})$ make a lower bounded angle together (and subsequently a lower bounded scalar product) for all the workers. The classical unbiased assumption is more restrictive as it requires this angle to be exactly equal to 0, and the scalar product to be equal to $\|\boldsymbol{\nabla}\mathcal{F}(\boldsymbol{\theta})\| \cdot \|\boldsymbol{G}(\boldsymbol{\theta})\|$. Most importantly, we highlight the fact that those assumptions are satisfied by KARDAM, since every gradient used in this section to compute the ***Kar*** update has passed the Lipschitz filter of the previous section.

**Theorem 4** (Convergence guarantee)**.** *We express the convergence guarantee in terms of the ergodic convergence, i.e., the weighted average of the $\mathcal{L}_2$ norm of all gradients ($||\nabla\mathcal{F}(\boldsymbol{\theta}_t)||^2$). Using the above-mentioned assumptions, and the maximum adaptive rate $\phi_{\max} = \max\{\phi_1, \ldots, \phi_t\}$, we get the following bound on the convergence rate.*

$$\frac{1}{T}\sum_{t=1}^{T} \mathbb{E}\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\|^2 \leq \left(2 + \phi_{\max} + \gamma KM\chi\phi_{\max}\right)\gamma Kd\sigma^2 + d\sigma^2 + 2DK\sigma\sqrt{d} + K^2D^2$$

*under the prerequisite that*

$$\sum_{\lambda \in \mathbf{\Lambda}_t} \lambda^2 |\mathbf{\Lambda}_t| \left\{ K\gamma_t^2 + \sum_{s=1}^{\infty} ( \sum_{\nu \in \mathbf{\Lambda}_{t+s}} \gamma_{t+s} K^2 \nu |\mathcal{G}_{t+s,\nu}| \Lambda^{-1}(\nu) \mathbb{I}_{(s \leq \Lambda^{-1}(\nu))} \gamma_t^2 ) \right\} \leq \sum_{\lambda \in \mathbf{\Lambda}_t} \frac{\gamma_t \lambda}{|\mathcal{G}_{t\lambda}|}$$

*where the Iverson indicator function is defined as follows*

$$\mathbb{I}_{(s \leq \Delta)} = \begin{cases} 1 & \text{if } s \leq \Delta \\ 0 & \text{otherwise.} \end{cases}$$

*and $\chi$ denotes a constant such that for all $\tau_{tl}$, the following inequality holds:*

$$\tau_{tl} \cdot \Lambda(\tau_{tl}) \leq \chi \tag{5.3}$$

It is important to note that the prerequisite for Theorem 4, holds for any decay function $\Lambda$ (since $\lambda < 1$ holds by definition) and for any standard learning rate schedule such that $\gamma_t < 1$. Various SGD approaches [93, 109, 184, 185] provide convergence guarantees with similar prerequisites.

**Theorem 5** (Convergence time complexity)**.** *Given any mini-batch size $|\xi|$, the number of gradients $M$ the server waits for before updating the model, and the total number of steps $T$, the time complexity for the convergence of* KARDAM *is:*

$$\mathcal{O}\left( \frac{\phi_{\max}}{\sqrt{T|\xi|M}} + \frac{\chi \phi_{\max}}{T} + d\sigma^2 + 2DK\sigma\sqrt{d} + K^2 D^2 \right)$$

Theorem 5 highlights the relation between the staleness and the convergence time complexity. This time complexity is linearly dependent on the decay bound ($\chi$) and the maximum adaptive rate ($\phi_{max}$).

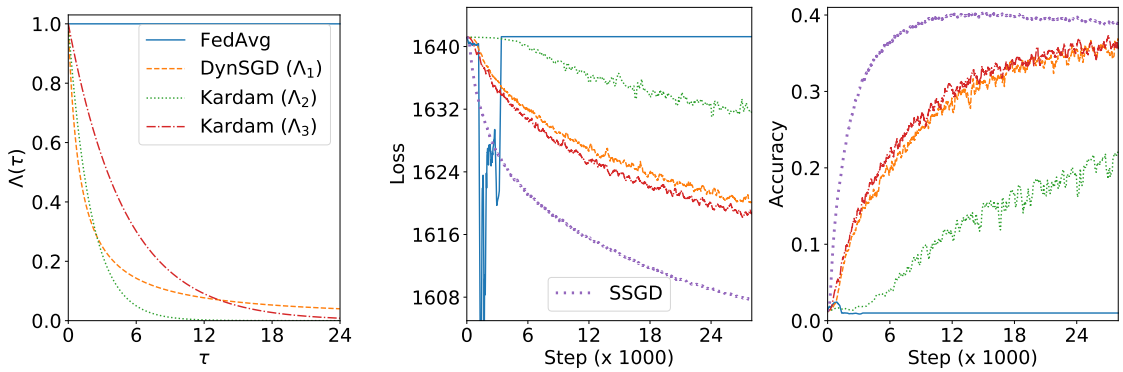
**Remark 2** (Dampening comparison)**.** *Given two dampening functions $\Lambda_1(\tau) = \frac{1}{1+\tau}$ and $\Lambda_2(\tau) = exp(-\alpha \sqrt[\beta]{\tau})$, and the convergence time complexity from Theorem 5, $\Lambda_2(\tau)$ converges faster than $\Lambda_1(\tau)$ when $\frac{\beta}{e} < \alpha \leq \frac{ln(\tau+1)}{\sqrt[\beta]{\tau}}$.*

We also empirically highlight Remark 2 by comparing these two functions in Figure 5.4 where DYNSGD [93] employs $\Lambda_1$ and KARDAM employs $\Lambda_2$.

The detailed proofs for Remark 2 and Theorems 4 and 5 are available in Appendix B.1.2.

## 5.4 Experiments

In this section, we report on our empirical evaluation of our distributed implementation of KARDAM. Experiments on Byzantine attacks are mostly illustrative for (a) the importance




**(a)** Dampening functions **(b)** Cost **(c)** Accuracy

**Figure 5.3 –** Staleness-aware learning for CIFAR-100. FedAvg is equivalent (given the setup) to KARDAM without the dampening component and SSGD denotes the ideal (synchronous) SGD execution. The staleness follows a Gaussian distribution ($mean = 12$, $\sigma = 4$) and the dampening functions are $\Lambda_1 = \frac{1}{\tau+1}$, $\Lambda_2 = e^{0.5\tau}$, $\Lambda_3 = e^{0.2\tau}$.

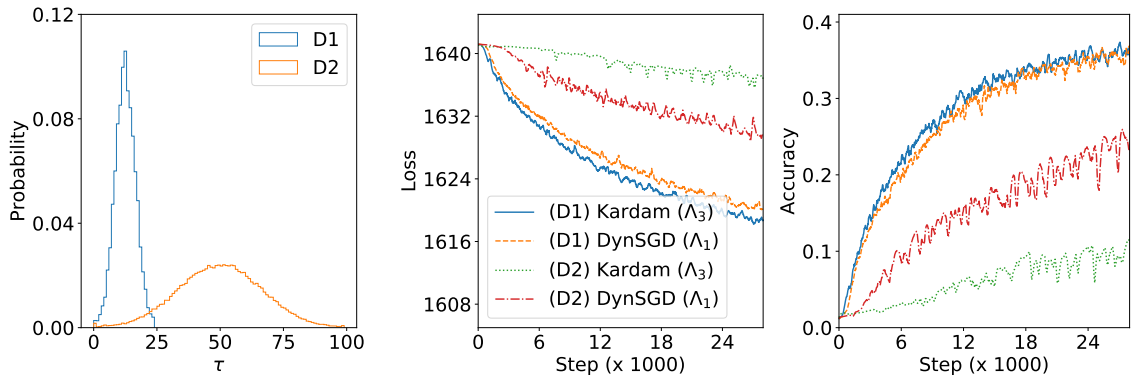
of the dampening component and (b) the overhead of the filtering component. Due to the intractability of testing all possible attacks, the only option is to prove Byzantine resilience mathematically and focus in the empirical part on the performance overhead of KARDAM.

We employ the image classification setup for E-MNIST and CIFAR-100 described in §3.3.2. If not stated otherwise, we employ a setup with no actual Byzantine behavior and deploy KARDAM with $f = 3$ on 10 workers.

**Staleness-aware learning.** We control the staleness (Appendix A.1) for approximating a Gaussian staleness distribution [185] (as shown in Figure 5.4(a)) and evaluate KARDAM with different dampening functions $\Lambda(\tau)$ (Definition 9) shown in Figure 5.3(a). We compare with the performance of KARDAM without the Byzantine resilience components (i.e., FEDAVG [119]) by using the constant function ($\Lambda_1 = 1$). Additionally, we compare against the ideal (synchronous) SGD execution (SSGD) and against a popular staleness-aware learning algorithm (DYNSGD [93]) that employs an inverse dampening function ($\Lambda_2(\tau) = \frac{1}{1+\tau}$). Finally, we use two exponential functions ($\Lambda = exp(-\alpha \cdot \tau)$).

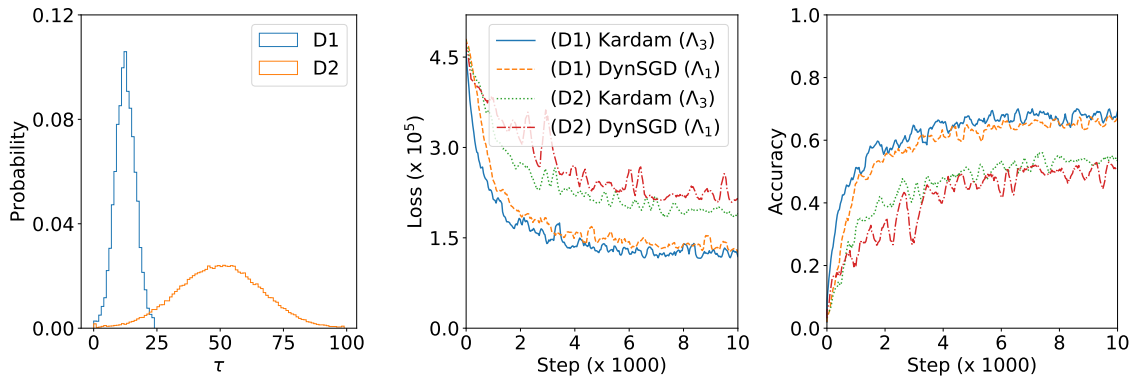
Figure 5.3 depicts the very fact that the staleness-aware component of KARDAM is crucial in asynchronous environments. We show that SSGD has the faster convergence whereas FEDAVG diverges (Figures 5.3(b) and 5.3(c)).

Figure 5.3 also highlights the need for an adjustable smoothness on the dampening function. A very steep function ($\Lambda_2$) almost ignores many of the updates (weighted by a very small value) and thus suffers a slower convergence. A tuned exponential function ($\Lambda_3$) accelerates the convergence in comparison with the inverse function of DYNSGD. Moreover, KARDAM ($\Lambda_3$) assigns larger weights to the less stale updates ($\tau < 13$) compared to DYNSGD and vice versa.




**(a)** Staleness distributions **(b)** Cost **(c)** Accuracy

**Figure 5.4 –** Impact of staleness for CIFAR-100.




**(a)** Staleness distributions **(b)** Cost **(c)** Accuracy

**Figure 5.5 –** Impact of staleness for EMNIST.

The dampening function selection is the outcome of adjusting the trade-off between the robustness and the magnitude of each update. We observe similar results for the E-MNIST dataset and thus highlight that the dampening function can be selected based on the expected staleness distribution, and not necessarily adjusted for each different application.

**Impact of staleness.** An increase in the amount of staleness leads to a slower convergence according to Theorem 5 (i.e., larger $\chi$ in Inequality 5.3). Figures 5.4 and 5.5 depict the impact of the amount of staleness on KARDAM and DYNSGD for two different staleness distributions ($D_1$ and $D_2$). We highlight that our experimental setup includes significantly higher staleness (D2) than the competitors [93, 185]. We observe that the smaller the mean of the distribution, the faster the convergence. We verify that KARDAM outperforms DYNSGD for $D_1$ given the better tailored exponential dampening. Regarding $D_2$, we observe that for CIFAR-100 DYNSGD performs better than KARDAM as the values of the exponential dampening function become too small and thus delay convergence. This depicts that the decay function $\Lambda(\tau)$ needs to be adjusted based on the staleness distribution. In §3.2.3, we provide one such algorithm that

can also be used as the dampening component of KARDAM given the generality of our formal guarantees.

**Byzantine resilience.** We observe that the overhead of the Byzantine resilience in the setup with no actual Byzantine behavior is only in terms of filtered (i.e., wasted) gradients and not in terms of convergence speed (in terms of update steps). Moreover, the drop ratio under the staleness distribution $D_1$ is 27.9% and 19.6% for KARDAM employing $\Lambda_1$ and $\Lambda_3$ respectively, thus aligned with our theoretical bound (Theorem 1). The slowdown would decrease accordingly by decreasing $f$, i.e., being more optimistic about the number of Byzantine workers.

We test KARDAM against a *baseline* Byzantine behavior (3 out of 10 workers send $g_p^{byz} = -10g_p$) and observe that KARDAM successfully filters 100% of the Byzantine gradients (an empirical confirmation of the theoretically proven Byzantine resilience of KARDAM).

## 5.5 Related Work and Concluding Remarks

KARDAM is, to the best of our knowledge, the first asynchronous distributed SGD algorithm that tolerates Byzantine behavior. In the following, we discuss papers that either address asynchrony or Byzantine behavior.

**Asynchronous stochastic gradient descent.** SGD is used widely in ML solutions due to its convergence guarantees with low time complexity per update, low memory cost, and robustness against noisy gradients. Several variants of SGD have been proposed to improve the convergence rate and the robustness against noise. Stale-synchronous parallel [49, 88] or bulk-synchronous [36, 188] variants typically target settings with limited staleness due to the limited performance variability among the computing devices. Other approaches consider variance minimization by importance sampling [5]. The theoretical guarantees underlying these approaches assume synchronous updates as well as a specific formula to compute a gradient norm on each sample, which is only valid for multilayer perceptrons. The scheduler in [184] assumes all workers to be constantly available, which makes the algorithm not applicable to our setting with Byzantine workers. [93] recently introduced a stale-synchronous parallel (SSP) heterogeneity-aware algorithm. SSP algorithms assume bounded staleness while KARDAM guarantees convergence without any such bound (i.e., asynchronous parallel). Additionally, KARDAM provides the flexibility of choosing the appropriate dampening function according to the expected staleness distribution while being Byzantine tolerant and asynchronous. We show both theoretically (Remark 2) and empirically (Figure 5.3) that an exponential dampening function leads to a faster convergence. [95] recently proposed an elegant optimizer to predict the optimal SGD variant based on the expected cost per iteration and the estimated number of iterations. This estimation does not however account for stale updates. Our convergence analysis for KARDAM could be employed to estimate the number of iterations for different dampening functions and hence to predict the optimal staleness-aware SGD variant. Finally, KARDAM put *before* lock-free solutions such as Hogwild [147] would

not break the convergence requirements (since the purpose of KARDAM is to preserve them despite Byzantine workers). However, KARDAM and Hogwild do not commute.

**Second order methods.** These methods rely on computing the Hessian matrix instead of the Lipschitz factor (KARDAM filtering component). They were not specifically designed for Byzantine resilience but can in fact be employed for that purpose. However, unlike our scalar-based Lipschitz filter ($\mathcal{O}(d)$ time complexity that is already within the usual cost of an SGD update), they suffer from the curse of dimensionality. Moreover, the parameter server does no less than $\Omega(d^2)$ verifications on the Hessian matrix or on the gradient covariance matrix. In the presence of a cheap (constant size $K$) heuristic, the parameter server will let the Byzantine worker with a margin of $d^2 - K$ open coordinates to use for an attack. Since $d \gg K$ the heuristic alternative clearly hampers Byzantine resilience.

**The differentiability lenses of Lipschitz.** A central piece of our work is to filter out suspected vectors based on their (lack of) similar Lipschitzness with the median behavior. We prove that this *filtering* idea is sound, given that a significant fraction ($\Omega(\frac{n-f}{n})$) of workers will almost surely pass it and that Byzantine workers passing it are not harmful. In fact, leveraging the Lipschitzness properties, in the differentiable context of gradient-based learning, is not an uncommon idea. It was used in different contexts, for example, to understand fine-grained robustness, i.e robustness of the model to internal errors at the level of neurons and/or weights, this was done in [69, 70] proving a tight upper bound on the Lipschitz coefficient of neural networks, and deriving an exponential dependency with the depth and a polynomial dependency with the Lipschitz coefficient of the activation function used in each layer. In the same time, Lipschitzness was leveraged to compute spectral bounds as in [12, 43] both of which observed the same exponential dependency on the depth. In fact, manipulating differentiable objects is what makes the world of learning fundamentally different from the usual world of distributed computing, where the focus is on combinatorial and discrete structures. The differentiability of learning algorithms acts as a source of relaxation to solve a distributed computing task (estimating a gradient, distributively) in asynchrony and in the presence of Byzantine workers. The shorter the time it takes for KARDAM to self-stabilize ($t_r$) the better in terms of the speed of convergence. As we prove in Theorem 3, $t_r$ is shorter with a larger global Lipschitz coefficient, i.e., steeper cost functions. Nevertheless, the cost function cannot be controlled. Yet, $t_r$ can be decreased by increasing the batch size per worker, which is no surprise in learning theory (increasing the batch size is one of the most unavoidable taxes [20, 24, 73] for increasing robustness). In practice, our experiments show no significant impact from $t_r$ in the absence of actual Byzantine workers. In their presence, KARDAM remains, to the best of our knowledge, the first provably Byzantine-resilient option to run SGD asynchronously.

An open problem now is how to tackle the Byzantine question in asynchronous machine learning *beyond* gradient-based algorithms. We argue that the core idea we present –filtering on quantiles from the recent past– could have applications to any approach where updates arrive with suspicions on either staleness or malicious behavior.

# Privacy Guarantees Part IV

# 6 Differentially Private Stochastic Coordinate Descent

## 6.1 Introduction

The availability of a huge amount of data has given rise to a plethora of machine learning (ML) applications. The main algorithmic challenges for these applications are two-fold: (i) minimize the learning time (including training and hyperparameter tuning) and (ii) maximize the output quality (e.g., the prediction accuracy made by the learnt model). Stochastic coordinate descent (SCD) [174] is a very popular optimization algorithm in both academia and industry due to its favorable convergence behavior and the absence of hyperparameters that need to be tuned [66, 74, 90, 115]. In particular, for training generalized linear models it is the algorithm of choice for many applications and has been implemented as a default solver in several popular packages such as Scikit-learn, TensorFlow and Liblinear [74].

**Challenge.** We study how existing applications, built on SCD, can be extended to guarantee differentially private model training. Making SCD private is however not trivial due to the noise addition that is vital for providing DP guarantees. An efficient implementation of SCD stores and updates not only the model vector $\boldsymbol{a}$ but also an auxiliary vector $\boldsymbol{v} := \boldsymbol{X}\boldsymbol{a}$ to avoid recurring computations. These vectors need to be *consistent* for standard convergence results to hold. However, independent noise addition to both vectors, necessary for DP, prohibits this consistency.

**Contribution.** We present DP-SCD, a differentially private version of the standard SCD algorithm [155] and formally prove the DP guarantees. We theoretically analyze SCD under noise addition and provide a bound on the maximum level of noise that can be tolerated to achieve a given level of utility. We empirically show (based on real datasets) that for applications for which DP-SCD performs exact minimization per update (e.g., ridge regression and support vector machines), DP-SCD achieves a better privacy-utility trade-off compared to the popular differentially private stochastic gradient descent algorithm (DP-SGD [2]) while,

at the same time, being free of a learning rate parameter that needs tuning. We also compare and discuss the primal and dual formulation of SCD in the differentially private setting.

## 6.2 Setup

Before we dive into the details of making SCD differentially private, we refer to Chapter 2 where we provide the notation details (§2.1), formally define the problem class and provide the necessary background on SCD (§2.3.2) and differential privacy (§2.5).

For the rest of this chapter we will use the common assumption that the data examples $\boldsymbol{x}_i$ are normalized, i.e., $\|\boldsymbol{x}_i\| = 1$ (as in [32]), and that the loss functions $\ell_i$ are $1/\mu$-smooth. A wide range of ML models fall into this setup including ridge regression and $L_2$-regularized SVM or logistic regression [155].

**Threat model.** We assume the threat model of [2]. We consider an adversary that has white-box access to the training procedure (algorithm, hyperparameters, and intermediate output) and can have access even to $[\cdots \boldsymbol{x}_{i-1}, \boldsymbol{x}_{i+1} \cdots]$, where $\boldsymbol{x}_i$ is the data instance the adversary is targeting. However, the adversary cannot have access to the intermediate results of any update computation. We make this assumption more explicit in §6.3.1.

## 6.3 Differentially Private Stochastic Coordinate Descent

We focus on the dual problem formulation (Problem 2.5) and summarize our main differentially private stochastic coordinate descent algorithm (DP-SCD) in Algorithm 1. The crucial extension in comparison with the standard dual SCD (SDCA [155]) is that we consider mini-batch based updates, that independently process a random sample of $L$ coordinates ($\mathcal{B}$) in each inner iteration (Steps 6-11). This is not only beneficial from a performance perspective, as the updates can be executed in parallel, but it also serves as a hyperparameter that steers the privacy-utility trade-off of our algorithm (similar to the lot size in [2]). We formalize our parallel updates based on [115]. In particular, we reuse the local subproblem formulation for the special case where each parallel process updates only a single example $j \in \mathcal{B}$.

$$\mathcal{F}_j^*(\alpha_j, \Delta, \boldsymbol{v}, \boldsymbol{x}_j) := \frac{1}{N}\ell_j^*(-\alpha_j - \Delta) + \frac{1}{2\lambda N^2}\left(\frac{1}{L}\|\boldsymbol{v}\|^2 + 2\boldsymbol{v}^\top \boldsymbol{x}_j \Delta + L\|\boldsymbol{x}_j\|^2 \Delta^2\right) \tag{6.1}$$

Note that the minimizer $\Delta_j$ in Step 7 can be often computed in closed form, e.g., for ridge regression, or SVMs. Exact minimization is however not necessary for our algorithm to converge. Approximate solutions are sufficient for convergence (e.g., [159, Assumption 1]). Hence, for logistic regression we use a single Newton step to efficiently update the coordinates.

**Algorithm 1:** DP-SCD (for Problem 2.5)

**Input:** $N$ examples $\boldsymbol{x}_i \in \mathbb{R}^M$, $y_i$: labels, $\lambda$: regularization, $T$: iterations, $L$: mini-batch size, $(\epsilon, \delta)$: DP parameters, $C$: scaling factor

1 **Init:** $\boldsymbol{\alpha} = \mathbf{0}$ ; $\boldsymbol{v} = \mathbf{0}$ ; shuffle examples $\boldsymbol{x}_i$
2 $\sigma \leftarrow$ *smallest noise magnitude, s.t.,* $\mathrm{MA}\left(\delta, \sigma, \frac{L}{N}, T\right) = \epsilon$
3 **for** $t = 1, 2, \ldots T$ **do**
4 $\quad \Delta\boldsymbol{v} = 0$
5 $\quad$ Randomly sample $L$ examples $\mathcal{B} \subset [N]$
6 $\quad$ **for** $j \in \mathcal{B}$ **do**
7 $\quad\quad \Delta_j = \arg\min_\Delta \mathcal{F}_j^*(\alpha_j, \Delta, \boldsymbol{v}, \boldsymbol{x}_j)$
8 $\quad\quad \Delta_j \mathrel{/}= \max\left(1, \frac{|\Delta_j|}{C}\right)$ `// scale`
9 $\quad\quad \boldsymbol{\Delta v} \mathrel{+}= \Delta_j \boldsymbol{x}_j$
10 $\quad\quad \boldsymbol{\Delta\alpha} \mathrel{+}= \boldsymbol{e}_j \Delta_j$
11 $\quad$ **end**
$\quad$ `// update the model`
12 $\quad \boldsymbol{\alpha} \mathrel{+}= \boldsymbol{e}_{\mathcal{B}}(\boldsymbol{\Delta\alpha} + \mathcal{N}(0, \sigma^2 2C^2 \boldsymbol{I_1}))$ `//` $\boldsymbol{I_1} \in \mathbb{R}^{L \times L}$
13 $\quad \boldsymbol{v} \mathrel{+}= \boldsymbol{\Delta v} + \mathcal{N}(0, \sigma^2 2C^2 \boldsymbol{I_2})$ `//` $\boldsymbol{I_2} \in \mathbb{R}^{M \times M}$
14 **end**
15 **return** $\boldsymbol{\theta} = \frac{1}{\lambda N} \boldsymbol{v}$ `// retrieve primal model`

Finally, to guarantee differential privacy, we bound the sensitivity of each coordinate update to be $C$ by scaling $\Delta_j$ (Step 8). We then use the Gaussian mechanism to make $\boldsymbol{\alpha}$ and $\boldsymbol{v}$ differentially private. We address two main questions regarding DP-SCD:

1. How much noise do we need to add to guarantee $(\epsilon, \delta)$-differential privacy? (§6.3.1)
2. Can we still give convergence guarantees for this new algorithm under noise addition? (§6.4)

We answer the first question in §6.3.1 by analyzing the sensitivity of our update function. For the second question (that we answer in §6.4), the main challenge is that independent noise addition destroys the consistency between $\boldsymbol{\alpha}$ and $\boldsymbol{v}$, i.e., $\boldsymbol{v} \neq \boldsymbol{X\alpha}$. We show how to address this challenge and prove convergence for our method.

### 6.3.1 Privacy Analysis

We view the training procedure as a sequence of mechanisms $\mathcal{M}_t$ where each mechanism corresponds to one outer iteration and computes an update on $L$ examples. We assume these mechanisms to be atomic from an adversary point of view, i.e., we assume no access to the individual coordinate updates. For determining the sensitivity of this mechanism it is important to note that all updates within mechanism $\mathcal{M}_t$ touch different data points and are computed independently. The output of each mechanism $\mathcal{M}_t$ is the concatenation $[\boldsymbol{\alpha}^\top, \boldsymbol{v}^\top]$ of the dual and the auxiliary vectors. The sensitivity of the output is given as:

**Lemma 2** (Sensitivity of DP-SCD)**.** *The sensitivity of each mechanism $\mathcal{M}_t$ in Algorithm 1 is bounded: $S_f \leq \sqrt{2}C$.*

*Proof.* Each mini-batch processing accesses sensitive information and thus needs to have DP guarantees. The input of each mini-batch processing consists of $\boldsymbol{\alpha}$ and $\boldsymbol{v}$ and the output consists of the updates, i.e., $\boldsymbol{f} = [\Delta_1, \cdots, \Delta_L, \sum_{\mathcal{B}} \boldsymbol{\Delta v}] \in \mathbb{R}^{L+M}$.

The sensitivity with respect to a single example is bounded by Step 8 of Algorithm 1. Given that $|\Delta_i| \leq C$ (per-example scaling) and $\|\boldsymbol{x}_i\| \leq 1$ (normalized data), it holds that $\|\boldsymbol{\Delta v}\| \leq \|\boldsymbol{x}_i\| * |\Delta_i| \leq C$. Hence, the sensitivity of $\boldsymbol{f}$ is as follows. Assume $\boldsymbol{X} \setminus \boldsymbol{X}' = \boldsymbol{x}_k$:

$$\begin{aligned}
\boldsymbol{f}(\boldsymbol{X}) &= [\Delta_1, \cdots, \Delta_k, \cdots, \Delta_L, \boldsymbol{\Delta v}_1 + \cdots + \boldsymbol{\Delta v}_k + \cdots + \boldsymbol{\Delta v}_L] \in \mathbb{R}^{L+M} \\
\boldsymbol{f}(\boldsymbol{X}') &= [\Delta_1, \cdots, 0, \cdots, \Delta_L, \boldsymbol{\Delta v}_1 + \cdots + 0 + \cdots + \boldsymbol{\Delta v}_L] \in \mathbb{R}^{L+M} \\
S_f^2 &:= \max_{\boldsymbol{X} \setminus \boldsymbol{X}' = \boldsymbol{x}_k} \|\boldsymbol{f}(\boldsymbol{X}) - \boldsymbol{f}(\boldsymbol{X}')\|^2 = \|[0, \cdots, \Delta_k, \cdots, 0, \boldsymbol{\Delta v}_k]\|^2 = \Delta_k^2 + \|\boldsymbol{\Delta v}_k\|^2 \leq C^2 + C^2 = 2C^2
\end{aligned}$$

□

**Theorem 6** (Privacy bound for DP-SCD)**.** *Algorithm 1 is $(\epsilon, \delta)$ differentially private for any $\epsilon = \mathcal{O}\left(q^2 T\right)$ and $\delta > 0$ if we choose $\sigma = \Omega\left(\frac{q\sqrt{T\, ln(1/\delta)}}{\epsilon}\right)$.*

*Proof.* Each mini-batch processing (Steps 6-11) accesses sensitive information and thus needs to have DP guarantees. We make the output of each mini-batch processing differentially private by using the Gaussian mechanism (Equation 2.9) with $M$ replaced by $M + L$ in our case. The moments of each mechanism $\mathcal{M}_{\boldsymbol{i}}$ are bounded (given Lemma 2 and [2, Lemma 3]). Hence, based on [2, Theorem 1], we can derive the lower bound for $\sigma$ that guarantees $(\epsilon, \delta)$-DP for the output model. □

In practice, we choose the smallest $\sigma$ that provides the given privacy guarantee, i.e.,

$$\mathrm{MA}\left(\delta, \sigma, \frac{L}{N}, T\right) \leq \epsilon \tag{6.2}$$

where $T$ denotes the iterations of Algorithm 1. Given that $\epsilon$ decreases monotonically with increasing $\sigma$, we perform binary search until the variance of the output of the MA gets smaller than 1% of the given $\epsilon$.

**Data-dependent constraints.** The update computation involves certain dataset-dependent constraints for applications such as logistic regression or SVMs. For example, logistic regression employs the labels to ensure that the logarithms in the delta computation are properly defined [155]. The noise addition breaks these constraints. An approach that enforces these constraints after the noise addition would leak privacy and break the DP guarantee.

We thus enforce these constraints in the beginning of the delta computation. As a result, the output model does not respect these constraints (as opposed to SCD). Nevertheless, there are no negative implications as the purpose of these constraints is to enable valid delta computations.

### 6.3.2 Cost Analysis

The performance overhead of DP-SCD with respect to SCD boils down to the cost of sampling the Gaussian distribution. This cost is proportional to the mini-batch size, i.e., larger size means less frequent noise additions and less frequent sampling. The noise addition also prohibits any performance optimizations that accelerate training for sparse datasets.

The time complexity for Algorithm 1 is $\mathcal{O}(T \cdot M)$. The updates for the coordinates with a given mini-batch can be parallelized. We discuss parallelizable variants of SCD in §6.6.

### 6.3.3 Primal Version

The primal formulation of DP-SCD (shown in Algorithm 2) computes the updates in a coordinate-wise manner, thus making differentially private learning more challenging than in the dual formulation. At each iteration of the inner loop (similar to Steps 6-11 of Algorithm 1) the primal version updates a given coordinate $j$ for all the examples. Therefore, the sampling ratio ($q$) is 1 as each $\Delta$-computation touches one coordinate of the entire dataset. This invalidates the important property of the mini-batch size ($L$) to regulate the privacy-utility trade-off. Additionally, the noise addition to make the primal version DP is significantly larger than the dual version.

**Algorithm 2:** PRIMALDP-SCD (for Problem 2.1)

**Input:** $S$: sample size, same input as Algorithm 1
1 **Init:** $\boldsymbol{\theta} = \mathbf{0}$ ; $\boldsymbol{v} = \mathbf{0}$ ; shuffle examples $\boldsymbol{x}_i$
2 $\sigma \leftarrow$ *smallest noise magnitude, s.t.,* $\mathrm{MA}(\delta, \sigma, 1, T) = \epsilon$
3 **for** $t = 1, 2, \cdots T$ **do**
4 $\quad \boldsymbol{\Delta v} = \mathbf{0}$
5 $\quad$ Randomly a block of $L$ coordinates $\mathcal{B} \subset [M]$
6 $\quad$ **for** $j \in \mathcal{B}$ **do**
7 $\quad\quad \Delta_j = \arg\min_{\Delta} \mathcal{F}_j(\theta_j, \Delta, \boldsymbol{v}, \boldsymbol{X}[j,:])$ `// update`
8 $\quad\quad \Delta_j \mathrel{/}= \max\left(1, \frac{|\Delta_j|}{C}\right)$ `// scale`
9 $\quad\quad \boldsymbol{\Delta v} \mathrel{+}= \Delta_j \boldsymbol{X}[j,:]$
10 $\quad\quad \boldsymbol{\Delta\theta} \mathrel{+}= \boldsymbol{e}_j \Delta_j$
11 $\quad$ **end**
$\quad$ `// update the model`
12 $\quad \boldsymbol{\theta} \mathrel{+}= \boldsymbol{e}_{\mathcal{B}}(\boldsymbol{\Delta\theta} + \mathcal{N}(0, \sigma^2(4L^2-2)C^2 \boldsymbol{I_1}))$ // $\boldsymbol{I_1} \in \mathbb{R}^{L \times L}$
13 $\quad \boldsymbol{v} \mathrel{+}= \boldsymbol{\Delta v} + \mathcal{N}(0, \sigma^2(4L^2-2)C^2 \boldsymbol{I_2})$ // $\boldsymbol{I_2} \in \mathbb{R}^{N \times N}$
14 **end**
15 **return** $\boldsymbol{\theta}$ `// DP model`

Each mini-batch processing accesses sensitive information and thus needs to have DP guarantees. We device the PRIMALDP-SCD by assuming that the norm of each row of $\boldsymbol{X}$ (i.e., a given coordinate for all the examples) instead of each column, is bounded ($\|\boldsymbol{X}[j,:]\| = 1$).

**Lemma 3** (Sensitivity of PRIMALDP-SCD)**.** *The sensitivity of each mechanism $\mathcal{M}_t$ in Algorithm 2 is bounded: $S_f \leq \sqrt{(4L^2-2)}C$.*

*Proof.* The input of each mini-batch processing consists of $\boldsymbol{\theta}$ and $\boldsymbol{v}$ and the output consists of the updates, i.e., $\boldsymbol{f} = [\Delta_1, \cdots, \Delta_L, \sum_{\mathcal{B}} \boldsymbol{\Delta v}] \in \mathbb{R}^{L+N}$. The sensitivity with respect to a single example is bounded by Step 8 of Algorithm 2 which ensures $|\Delta_j| \leq C$ (per-example scaling). Moreover, given the normalized data, it holds that $\|\boldsymbol{\Delta v}\| \leq \|\boldsymbol{X}[j,\xi]\| * \Delta_j \leq C$. Hence the sensitivity of $\boldsymbol{f}$ can be computed by choosing $\boldsymbol{X}, \boldsymbol{X}'$ such that $\|\boldsymbol{f}(\boldsymbol{X}) - \boldsymbol{f}(\boldsymbol{X}')\|^2$ is maximized. The sensitivity is maximized when $\boldsymbol{X} \setminus \boldsymbol{X}' = \boldsymbol{x}_0$:

$$\boldsymbol{f}(\boldsymbol{X}) = [\Delta_1, \Delta_2, \cdots, \Delta_L, \boldsymbol{\Delta v}_1 + \boldsymbol{\Delta v}_2 + \cdots + \boldsymbol{\Delta v}_L] \in \mathbb{R}^{L+N}$$
$$\boldsymbol{f}(\boldsymbol{X}') = [0, \Delta'_2, \cdots, \Delta'_L, 0 + \boldsymbol{\Delta v}'_2 + \cdots + \boldsymbol{\Delta v}'_L] \in \mathbb{R}^{L+N}$$

The difference among $\boldsymbol{f}(\boldsymbol{X}), \boldsymbol{f}(\boldsymbol{X}')$ is the result of the difference due to the missing example for $\boldsymbol{X}'$ that affects all the updates as each update employs all the examples for coordinate $j$. Moreover, the subsequent values-vectors can, in the worst case, be opposite. Therefore, the sensitivity is as follows by using the triangle inequality.

$$\begin{aligned}
S_f^2 &:= \max_{\boldsymbol{X}\setminus\boldsymbol{X}'=\boldsymbol{x}_k} \|\boldsymbol{f}(\boldsymbol{X}) - \boldsymbol{f}(\boldsymbol{X}')\|^2 \\
&= \|[\Delta_1, \Delta_2 - \Delta'_2, \cdots, \Delta_L - \Delta'_L, \boldsymbol{\Delta v}_1 + \boldsymbol{\Delta v}_2 - \boldsymbol{\Delta v}'_2 + \cdots + \boldsymbol{\Delta v}_L - \boldsymbol{\Delta v}'_L]\|^2 \\
&= |\Delta_1|^2 + |\Delta_2 - \Delta'_2|^2 + \cdots + |\Delta_L - \Delta'_L|^2 + (\|\boldsymbol{\Delta v}_1 + \boldsymbol{\Delta v}_2 - \boldsymbol{\Delta v}'_2 + \cdots + \boldsymbol{\Delta v}_L - \boldsymbol{\Delta v}'_L\|)^2 \\
&\leq |\Delta_1|^2 + (|\Delta_2| + |\Delta'_2|)^2 + \cdots + (|\Delta_L| + |\Delta'_L|)^2 + (\|\boldsymbol{\Delta v}_1\| + \|\boldsymbol{\Delta v}_2\| + \|\boldsymbol{\Delta v}'_2\| + \cdots + \|\boldsymbol{\Delta v}_L\| + \|\boldsymbol{\Delta v}'_L\|)^2 \\
&\leq C^2 + 4C^2 + \cdots + 4C^2 + (C + 2C + \cdots + 2C)^2 \\
&= (4L-3)C^2 + (2L-1)^2C^2 = (4L^2-2)C^2
\end{aligned}$$

□

We therefore conclude that the dual version (Algorithm 1) is preferable over the primal in the DP setting.

### 6.3.4 Sequential version

We present a baseline algorithm namely SEQDP-SCD to depict the importance of independent updates (inside a given mini-batch) for DP-SCD, and focus on the dual problem. As shown in Algorithm 3, SEQDP-SCD adopts the natural (i.e., as in vanilla SCD) method of performing *sequential* and thus correlated updates. In particular, the updates for both $\boldsymbol{\alpha}$ and $\boldsymbol{v}$ at sample

$j$ (Steps 10 and 11 of Algorithm 3) depend on all the previous samples of the same mini-batch ($\mathcal{B}$). In contrast, in our main DP-SCD algorithm (Algorithm 1), there is no such correlation and the updates can be also executed in parallel. On the one hand this correlation is better for convergence in terms of iterations (not time) [90]. On the other hand, due to this correlation, Algorithm 3 requires significantly more noise than Algorithm 1 that makes the overall performance of our proposed algorithm (Algorithm 1) significantly better as we show in §6.5.

**Algorithm 3:** SEQDP-SCD (for Problem 2.5)

```
   // same as Steps 1-3 of Algorithm 1
3  for t = 1,2,...T do
4      Δv = 0
5      Randomly sample L examples B ⊂ [N]
6      for j ∈ B do
7          Δ_j = argmin_Δ F*_j(α_j, Δ, v, x_j)
8          Δ_j /= max(1, |Δ_j|/C)                          // scale
9          Δv += Δ_j x_j
10         α += e_j Δ_j                                    // update the model
11         v += Δv
12     end
       // add noise
13     α += e_B N(0, σ²(4L² − 2)C² I_1)                    // I_1 ∈ R^{L×L}
14     v += N(0, σ²(4L² − 2)C² I_2)                        // I_2 ∈ R^{M×M}
15 end
16 return θ = (1/λN) v                                     // retrieve primal model
```

**Lemma 4** (Sensitivity of SEQDP-SCD)**.** *The sensitivity of each mechanism $\mathcal{M}_t$ in Algorithm 3 is bounded: $S_f \leq \sqrt{(4L^2-2)}C$.*

*Proof.* The proof is similar to Lemma 3. The difference among $\boldsymbol{f}(\boldsymbol{X}), \boldsymbol{f}(\boldsymbol{X}')$ consists of (a) the difference due to the missing example for $\boldsymbol{X}'$ and (b) the difference due to all the subsequent values (correlated updates). Moreover, the subsequent values-vectors can, in the worst case, be opposite. Therefore, the sensitivity follows by using the triangle inequality. □

## 6.4 Convergence Analysis

We recall the main challenge for generalizing the convergence guarantees of SCD to DP-SCD, namely the need to handle potential inconsistencies between the auxiliary vector $\boldsymbol{v}$ and the model vector $\boldsymbol{\alpha}$, i.e., $\boldsymbol{v} \neq \boldsymbol{X}\boldsymbol{\alpha}$. Note that a variant of Algorithm 1 that only updates $\boldsymbol{\alpha}$ and recomputes $\boldsymbol{v}$ in every iteration would overcome this issue. However, such a variant involves two disadvantages that make it impractical: (i) significant computational overhead and (ii) on the final step this variant would need to employ the entire dataset to map the dual model to the primal model ($\boldsymbol{\theta} := \frac{1}{\lambda N}\boldsymbol{X}\boldsymbol{\alpha}$), which creates a massive privacy leakage and negates the effect of the mini-batch (i.e., q=1 for the moments accountant).

To analyze the convergence of Algorithm 1 we split each mini-batch iteration of Algorithm 1 in two steps: (i) the *update* step includes the computation of $L$ coordinate updates (Steps 6-11) and (ii) the *perturbation* step adds Gaussian noise to the two vectors $\boldsymbol{\alpha}$ and $\boldsymbol{v}$ independently (Steps 12 and 13).

We base our analysis on [115] and thus include the parameters $\gamma$ and $\sigma'$ (not related to the standard deviation of the noise $\sigma$) for ease of analysis. We specify our general result to the case of Algorithm 1 by setting $\gamma = 1$ and $\sigma' = L$. We therefore consider the following privacy preserving model sequence $\{\boldsymbol{\alpha}\}_i$ with intermediate, non-public models $\{\hat{\boldsymbol{\alpha}}\}_i$. The corresponding sequence for $\{\boldsymbol{v}\}_i$ can be derived in a similar way.

$$\begin{aligned}
\boldsymbol{\alpha}_0 \to \hat{\boldsymbol{\alpha}}_1 &:= \boldsymbol{\alpha}_0 + \gamma \sum_{j=1}^{L} \Delta\boldsymbol{\alpha}_{0,j} \to \boldsymbol{\alpha}_1 := \hat{\boldsymbol{\alpha}}_1 + \boldsymbol{\eta}_{\alpha,1} \\
\to \hat{\boldsymbol{\alpha}}_2 &:= \boldsymbol{\alpha}_1 + \gamma \sum_{j=1}^{L} \Delta\boldsymbol{\alpha}_{1,j} \to \dots \\
\to \boldsymbol{\alpha}_n &:= \boldsymbol{\alpha}_0 + \gamma \sum_{i=1}^{n-1} \Delta\boldsymbol{\alpha}_i + \sum_{i=1}^{n-1} \boldsymbol{\eta}_{\alpha,i}
\end{aligned} \tag{6.3}$$

Here, $\boldsymbol{\eta} \sim \mathcal{N}(0, \sigma^2)$ is Gaussian noise added to preserve privacy and $\Delta\boldsymbol{\alpha}_i := \sum_{j=1}^{L} \Delta\boldsymbol{\alpha}_{i,j}$ with $\Delta\boldsymbol{\alpha}_{i,j} = \operatorname{argmin}_{\Delta} \mathcal{F}^*(\boldsymbol{\alpha}_i + \boldsymbol{e}_j \Delta)$.

To analyze the convergence behavior of Algorithm 1, we consider the two intermediate steps $\boldsymbol{\alpha}_i \to \hat{\boldsymbol{\alpha}}_{i+1}$ (update step) and $\hat{\boldsymbol{\alpha}}_{i+1} \to \boldsymbol{\alpha}_{i+1}$ (perturbation step) separately. The main idea is to show that:

(i) $\boldsymbol{\alpha}_i \to \hat{\boldsymbol{\alpha}}_{i+1}$ decreases the objective even if the update is computed based on a noisy version of $\boldsymbol{\alpha}, \boldsymbol{v}$.

(ii) the damage on the objective when adding noise and going from $\hat{\boldsymbol{\alpha}}_{i+1} \to \boldsymbol{\alpha}_{i+1}$ is bounded.

The intuition is that as long as the damage is smaller than the gain achieved with the update, the objective decreases and thus the algorithm converges.

The key observation that allows us to derive convergence guarantees in this setting is the following.

**Remark 3** (Consistency in expectation)**.** *Algorithm 1 preserves the consistency between $\boldsymbol{\alpha}$ and $\boldsymbol{v}$ in expectation, i.e.,* $\mathbb{E}[\boldsymbol{v}] = \boldsymbol{X}\mathbb{E}[\boldsymbol{\alpha}]$.

*Proof.* This follows from the construction of the model updates and the independent noise with zero mean that is added to both sequences $\{\boldsymbol{\alpha}\}_i, \{\boldsymbol{v}\}_i$. □

### 6.4.1 Update Step

Each iteration of Algorithm 1 computes a mini-batch update $\Delta\boldsymbol{\alpha}$ that is applied to the model $\boldsymbol{\alpha}$ and indirectly to the auxiliary vector $\boldsymbol{v}$ in Steps 12 and 13 respectively. We denote by $\dot{\Delta\boldsymbol{\alpha}}$ the unscaled version of this update, i.e., the update computed excluding Step 8. We add this step back later in our analysis. Lemma 5 gives a lower bound for the decrease in the objective achieved by performing this update even if $\dot{\Delta\boldsymbol{\alpha}}$ is computed based on noisy versions of $\boldsymbol{\alpha}, \boldsymbol{v}$ where $\mathbb{E}[\boldsymbol{v}] = \mathbb{E}[\boldsymbol{X}\boldsymbol{\alpha}]$ but $\boldsymbol{v} \neq \boldsymbol{X}\boldsymbol{\alpha}$.

**Lemma 5** (Update step - objective decrease lower bound)**.** *Assuming $\ell_i$ are $1/\mu$-smooth, then the update step of Algorithm 1 decreases the objective, even if computed based on a noisy version of $\boldsymbol{\alpha}, \boldsymbol{v}$. The decrease is as follows:*

$$\mathbb{E}[\mathcal{S}(\boldsymbol{\alpha}) - \mathcal{S}(\boldsymbol{\alpha} + \dot{\Delta\boldsymbol{\alpha}})] \geq \frac{\mu\lambda L}{\mu\lambda N + L}\mathbb{E}[\mathcal{S}(\boldsymbol{\alpha})] \tag{6.4}$$

*where $\mathcal{S}$ denotes the dual suboptimality defined as: $\mathcal{S}(\boldsymbol{\alpha}) := \mathcal{F}^*(\boldsymbol{\alpha}) - \min_{\boldsymbol{\alpha}} \mathcal{F}^*(\boldsymbol{\alpha})$.*

*Proof.* (Sketch) The starting point of our proof is [115, Lemma 3], that relates the decrease of each parallel update (achieved on the subproblem shown in Equation 6.1) to the global function decrease. Then, we take expectation w.r.t the randomization of the noise and proceed along the lines of [115, Lemma 5] to involve the duality gap (Equation 2.6) in our analysis. Finally, based on an inequality for the duality gap and Remark 3 we arrive at the bound stated in Lemma 5. Note that we recover the classical result of SDCA[155] for the sequential case where $L = 1$. The full proof is in Appendix B.2.1. □

**Scaling.** When computing the update $\Delta\boldsymbol{\alpha}$ in Algorithm 1, each coordinate of $\dot{\Delta\boldsymbol{\alpha}}$ is *scaled* to a maximum magnitude of $C$ (Step 8) in order to bound the sensitivity of each update step. In strong contrast to SGD, where this scaling step destroys the unbiasedness of the gradients and thus classical convergence guarantees no longer hold, for DP-SCD the scaling only translates into a smaller function decrease. This is a remarkable property of SCD when analyzed in the DP setting.

To incorporate scaling into our analysis we use the following inequality which is guaranteed to hold for some $\kappa \in [0, 1)$ due to the convexity of the objective.

$$\begin{aligned} &\mathcal{S}(\boldsymbol{\alpha} + \Delta\boldsymbol{\alpha}) \leq (1-\kappa)\mathcal{S}(\boldsymbol{\alpha} + \dot{\Delta\boldsymbol{\alpha}}) + \kappa\mathcal{S}(\boldsymbol{\alpha}) \\ \Leftrightarrow &\mathbb{E}[\mathcal{S}(\boldsymbol{\alpha}) - \mathcal{S}(\boldsymbol{\alpha} + \Delta\boldsymbol{\alpha})] \geq (1-\kappa)\mathbb{E}[\mathcal{S}(\boldsymbol{\alpha}) - \mathcal{S}(\boldsymbol{\alpha} + \dot{\Delta\boldsymbol{\alpha}})] \end{aligned} \tag{6.5}$$

The scaling step (Step 8 of Algorithm 1) preserves the linear convergence of Lemma 5 and decreases the rate by a factor of $(1-\kappa)$. Note that for $\kappa = 0$ (i.e., no scaling) the solution is exact and the smaller the scaling factor $C$, the larger the $\kappa$.

### 6.4.2 Perturbation Step

To prove the convergence of DP-SCD (Algorithm 1), it remains to show that adding noise at the end of each mini-batch update is not increasing the objective more than the decrease achieved by the rescaled update $\boldsymbol{\Delta\alpha}$.

**Lemma 6** (Perturbation step - objective increase upper bound)**.** *Assume $\ell_i^*$ are $\nu$-smooth. Then, the perturbation step of Algorithm 1 increases the objective by at most:*

$$\mathbb{E}[|\mathcal{S}(\boldsymbol{\alpha}+\Delta\boldsymbol{\alpha}+\boldsymbol{\eta})-\mathcal{S}(\boldsymbol{\alpha}+\Delta\boldsymbol{\alpha})|] \leq \frac{1}{2}\left(\nu+\frac{1}{\lambda N^2}\right)L\sigma^2 \tag{6.6}$$

*Proof.* Given that $\ell_i^*$ is $\nu$-smooth and the regularization term of $\mathcal{F}^*$ (Problem 2.5) is $\frac{1}{\lambda N^2}$-smooth, $\mathcal{F}^*$ is $\nu' = \nu + \frac{1}{\lambda N^2}$ smooth. We thus have $\mathcal{F}^*(\boldsymbol{\alpha}+\Delta\boldsymbol{\alpha}+\boldsymbol{\eta}) \leq \mathcal{F}^*(\boldsymbol{\alpha}+\Delta\boldsymbol{\alpha}) + \boldsymbol{\eta}^\top\nabla\mathcal{F}^* + \frac{\nu'}{2}\|\boldsymbol{\eta}\|^2$. We then subtract $\min_a \mathcal{F}^*(\boldsymbol{\alpha})$ on both sides and take expectations w.r.t the randomness in the perturbation noise. The claim follows from $\mathbb{E}[\boldsymbol{\eta}]=0$ and $\mathbb{E}[\|\boldsymbol{\eta}\|^2]=L\sigma^2$. □

Combining Inequality 6.5, Lemmas 5 and 6 yields our main result stated in the following theorem.

**Theorem 7** (Utility guarantee for Algorithm 1)**.** *Suppose that $\ell_i$ is $1/\mu$-smooth and $\nu$-strongly convex. If we choose $L$, $C$ such that $L(2(1-\kappa)\mu\lambda-1) > \mu\lambda N$ for $\kappa \in (0,1)$, and $T$ such that $T = \mathcal{O}\left(log\left(\frac{N^2\epsilon^2}{(\nu+\frac{1}{\lambda N^2})L^3\, ln(1/\delta)}\right)\right)$, then the utility of Algorithm 1 is bounded:*

$$\mathbb{E}[\mathcal{S}(\boldsymbol{\alpha}^{(T)})] \leq \mathcal{O}\left(\left(\nu+\frac{1}{\lambda N^2}\right)L^3\, log\left(\frac{N\epsilon}{\left(\nu+\frac{1}{\lambda N^2}\right)L}\right)\, ln\,(1/\delta)\,/\,(N^2\epsilon^2)\right) \tag{6.7}$$

*Proof.* We reorder terms in Inequality 6.6 and subtract $\mathcal{S}(\boldsymbol{\alpha})$ on both sides. We then combine Inequalities 6.4 and 6.5 and get that the suboptimality decreases per round by:

$$\mathbb{E}[\mathcal{S}(\boldsymbol{\alpha})-\mathcal{S}(\boldsymbol{\alpha}+\Delta\boldsymbol{\alpha}+\boldsymbol{\eta})] \geq \frac{(1-\kappa)\mu\lambda L}{\mu\lambda N+L}\mathbb{E}[\mathcal{S}(\boldsymbol{\alpha})] - \frac{1}{2}\left(\nu+\frac{1}{\lambda N^2}\right)L\sigma^2$$

At iteration $t$ we thus have:

$$\begin{aligned}
\mathbb{E}[\mathcal{S}(\boldsymbol{\alpha}^{(t-1)})] - \mathbb{E}[\mathcal{S}(\boldsymbol{\alpha}^{(t)})] &\geq \frac{(1-\kappa)\mu\lambda L}{\mu\lambda N+L}\mathbb{E}[\mathcal{S}(\boldsymbol{\alpha}^{(t-1)}))] - \frac{1}{2}\left(\nu+\frac{1}{\lambda N^2}\right)L\sigma^2 \\
\Leftrightarrow \mathbb{E}[\mathcal{S}(\boldsymbol{\alpha}^{(t)})] &\leq \underbrace{\left(1-\frac{(1-\kappa)\mu\lambda L}{\mu\lambda N+L}\right)}_{A}\mathbb{E}[\mathcal{S}(\boldsymbol{\alpha}^{(t-1)}))] + \frac{1}{2}\underbrace{\left(\nu+\frac{1}{\lambda N^2}\right)}_{\nu'}L\sigma^2
\end{aligned}$$

We apply the previous inequality recursively and get:

$$\mathbb{E}[\mathcal{S}(\boldsymbol{\alpha}^{(T)})] \leq A^T\mathbb{E}[\mathcal{S}(\boldsymbol{\alpha}^{(0)}))] + \mathcal{O}\left(\nu' L\sigma^2\right)$$

$$\overset{\text{Theorem 6}}{\leq} \quad A^T \mathbb{E}[\mathcal{S}(\boldsymbol{\alpha}^{(0)}))] + \mathcal{O}\left(\frac{\nu' L^3 T\, ln(1/\delta)}{N^2\epsilon^2}\right)$$

If we choose $L$ and $C$ such that $A < \frac{1}{2} \Leftrightarrow L(2(1-\kappa)\mu\lambda - 1) > \mu\lambda N$ and $T$ such that $T = \mathcal{O}\left(log\left(\frac{N^2\epsilon^2}{\nu' L^3\, ln(1/\delta)}\right)\right)$, we get the bound on the utility:

$$\mathbb{E}[\mathcal{S}(\boldsymbol{\alpha}^{(T)})] \leq \mathcal{O}\left(\frac{\nu' L^3\, ln(1/\delta)}{N^2\epsilon^2}\right) + \mathcal{O}\left(\frac{\nu' L^3 T\, ln(1/\delta)}{N^2\epsilon^2}\right)$$

By omitting the $ln$ term the bound on T simplifies as: $T = \mathcal{O}\left(log\left(\frac{N\epsilon}{\nu' L}\right)\right)$. Hence the utility bound becomes:

$$\mathbb{E}[\mathcal{S}(\boldsymbol{\alpha}^{(T)})] \leq \mathcal{O}\left(\left(\nu + \frac{1}{\lambda N^2}\right) L^3\, log\left(\frac{N\epsilon}{\left(\nu + \frac{1}{\lambda N^2}\right) L}\right) ln\,(1/\delta)\,/(N^2\epsilon^2)\right)$$

□

The suboptimality is proportional to the magnitude of the noise and hence, finding the exact minimizer requires $\sigma \to 0$ (i.e., $\epsilon \to \infty$). The smaller the $\sigma$ the larger the $\epsilon$ and thus the less private the learning is. We empirically confirm that DP-SCD converges smoother with a smaller $\sigma$ in §6.5.

Theorem 7 constitutes the first analysis of coordinate descent in the differentially private setting and it can be a stepping stone for future theoretical results in this setting as we discuss in §6.7.

## 6.5 Experiments

Our empirical results compare our new DP-SCD algorithm (Algorithm 1) against SCD, SGD, DP-SGD. We include SEQDP-SCD (Algorithm 3) as a baseline, to depict the importance of independent updates (inside a given mini-batch) for DP-SCD. We test the performance on three popular applications that belong to GLMs (§6.2), namely ridge regression, logistic regression and SVMs. Our implementation is in Python and available[1].

[1] https://github.com/gdamaskinos/dpscd

### 6.5.1 Setup

**Datasets.** We employ public real datasets. In particular, we report on YearPredictionMSD[2] for ridge regression, Phishing[3] for logistic regression, and Adult[4] for SVMs. We preprocess each dataset by scaling each coordinate by its maximum absolute value, followed by scaling each example to unit norm (normalized data). For YearPredictionMSD we center the labels at the origin. Based on [15] and regarding Adult, we convert the categorical variables to dummy/indicator ones and replace the missing values with the most frequently occurring value of the corresponding feature. We employ a training/test split for our data, to train and test the performance of our algorithms. YearPredictionMSD and Adult include a separate training and test set file. Phishing consists of a single file that we split with 75%:25% ratio into a training and a test set. Finally, we hold-out a random 25% of the training set for tuning the hyperparameters (validation set). The resulting training/validation/test size is {347786/115929/51630, 24420/8141/16281, 6218/2073/2764} and the number of coordinates are {90, 81, 68} for {YearPredictionMSD, Adult, Phishing} respectively.

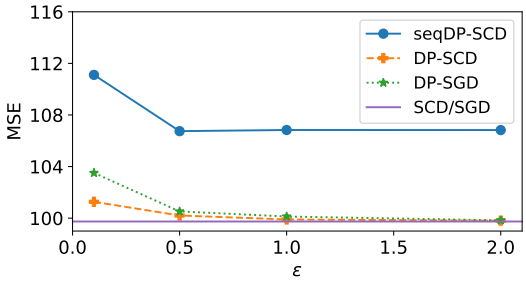
**Performance metrics.** *Accuracy* measures the classification performance as the fraction of correct predictions among all the predictions. The larger the accuracy, the better the utility. *Mean squared error (MSE)* measures the prediction error as: $MSE = \frac{1}{N}\sum_{i=1}^{N}(\hat{Y}_i - Y_i)^2$ where $\hat{Y}_i$ is the predicted value and $Y_i$ is the actual value. The lower the MSE, the better the utility. We quantify convergence by showing the decrease in the primal objective ($\mathcal{F}(\boldsymbol{X}^{(\text{training})}, \boldsymbol{\theta})$ from Problem 2.1) on the training set.

**Hyperparameters.** We fix $\lambda$ to $10^{-4}$ for YearPredictionMSD and Phishing and to $10^{-5}$ for the Adult dataset based on the best performance of SCD and SGD for a range of $\lambda \in \{10^{-8}, 10^{-7}, \cdots, 1, 10, \cdots, 10^{8}\}$. For a fair comparison of the DP algorithms, the iterations need to be fixed. Based on [175], we test the DP algorithms for $\{5, 10, 50\}$ epochs and fix the number of iterations to $T = 50N$ (i.e., 50 epochs) for YearPredictionMSD and $T = 10N$ for the other datasets. Based on [171, 183], we vary $\epsilon$ in $\{0.1, 0.5, 1, 2\}$ and fix $\delta = 0.001$. We choose the other hyperparameters by selecting the combination with the best performance (lowest MSE for ridge regression and largest accuracy for logistic regression and SVMs) on the validation set. The range of tested values is as follows.

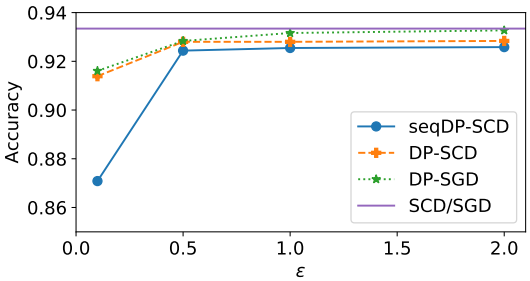
- $C, \eta \in \{10^{-8}, 10^{-7}, \cdots, 1, \cdots, 10^{4}\}$
- $|\xi|, L \in \{0, 5, 10, 50, 100, 200, 500, 1000, 1250, 1500, 1750, 2000\}$

**Deployment.** We run our experiments on commodity Linux machines. There are no special hardware requirements for our code other than enough RAM to load the datasets. We report

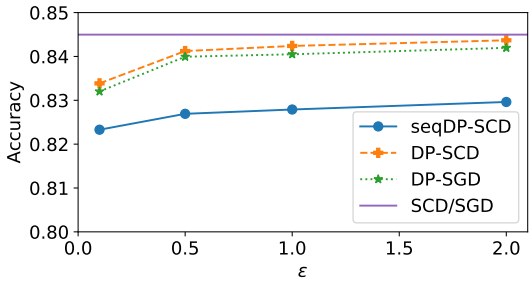
[2] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/regression.html
[3] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html
[4] https://archive.ics.uci.edu/ml/datasets/Adult




**(a)** Ridge regression (YearPredictionMSD)




**(b)** Logistic regression (Phishing)




**(c)** SVMs (Adult)

**Figure 6.1 –** Privacy-utility trade-off. Better utility means lower MSE or larger accuracy. DP-SCD outperforms DP-SGD for the applications that enable exact update steps (namely ridge regression and SVMs) despite DP-SCD having less hyperparameters.

the median result across 10 different runs by changing the seeding, i.e., the randomization due to initialization, sampling and Gaussian noise.

### 6.5.2 Results

**Tuning cost.** The hyperparameters of SEQDP-SCD, DP-SCD, SGD, DP-SGD are $(L, C)$, $(L, C)$, $(\eta, |\xi|)$, $(\eta, L, C)$ respectively; SCD requires no tuning. We tested a total of 156 configurations for DP-SCD as opposed to a total of 2028 for DP-SGD. For large-scale datasets that require significant amount of training time and resources, the difference in the number of hyperparameters constitutes an appealing property of DP-SCD. Noteworthy, the SGD tuning is not useful for the DP-SGD tuning as the best choice for $\eta$ depends on the choice of $C$ [165].

**Privacy-utility trade-off.** Figure 6.1 quantifies the trade-off between privacy and utility for different privacy levels (i.e., $\epsilon$ values). We observe that SEQDP-SCD has the worst performance due to the significantly larger noise compared to the other algorithms. DP-SCD performs better than DP-SGD for ridge regression and SVMs, and worse for logistic regression, that can be attributed to the following. On the one hand, DP-SCD requires $\sqrt{2}$ more noise than DP-SGD (for the same privacy guarantee) due to the need of a shared vector (§6.3). On the other hand, each update of DP-SCD finds an exact solution to the minimization problem for

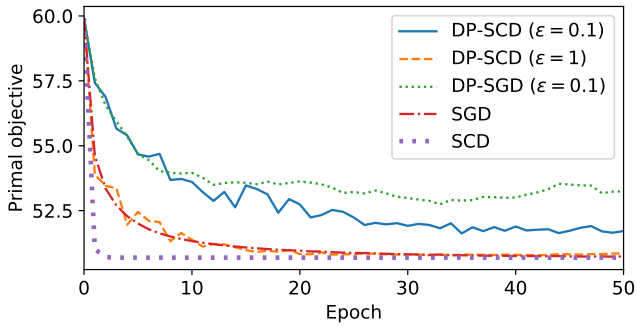



**Figure 6.2 –** Impact of noise on convergence. Differential privacy does not prevent convergence but increases the noise in reducing the objective and the distance to the optimum (aligned with the result of Theorem 7).

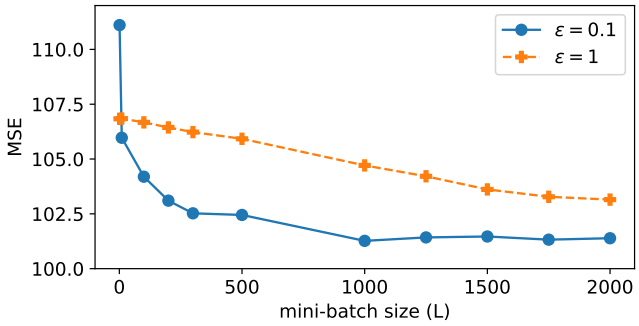



**Figure 6.3 –** Impact of mini-batch size on utility for DP-SCD. Utility increases with increasing mini-batch size, till a saturation point that depends on the level of privacy.

ridge regression and SVMs and an approximate one for logistic regression, whereas DP-SGD takes a direction opposite to the gradient. Apparently, the steps of DP-SCD in the case of ridge regression and SVMs are more precise despite suffering more noise than DP-SGD.

**Convergence.** Figure 6.2 shows the impact of noise on the convergence behavior for the DP algorithms on the YearPredictionMSD dataset. In particular, for a given $\epsilon$, we select the best (in terms of validation MSE) configuration (also used in Figure 6.1(a)), and measure the decrease in the objective with respect to epochs (not time as that would be implementation-dependent). We empirically verify the results of Theorem 7 by observing that the distance between the convergence point and the optimum depends on the level of privacy. Moreover, DP-SCD and DP-SGD converge with similar speed for $\epsilon = 0.1$. Decreasing the amount of noise ($\epsilon = 1$), makes DP-SCD converge almost as fast as SGD and with more stability comparing to $\epsilon = 0.1$. This is aligned with the results of §6.4, i.e., the fact that the larger amount of noise (decrease in $\epsilon$) makes the decrease in the suboptimality more noisy.

**Mini-batch size.** The mini-batch size is an important parameter for DP-SCD that affects the privacy-utility trade-off while also controls the level of parallelism (§6.3). Figure 6.3 shows that the utility improves (i.e., MSE drops) as the mini-batch size increases till a saturation point. The larger the noise, the smaller the value of this saturation point ($L = 1000$ for $\epsilon = 0.1$ and $L = 2000$ for $\epsilon = 1$). We observe that increasing the level of parallelism can only improve the utility of DP-SCD thus making our algorithm suitable for large-scale parallel processing.

## 6.6 Related Work

**Perturbation methods for DP.** Existing works achieve differentially private ML by perturbing the query output (i.e., model prediction). These works target both convex and non-convex optimization algorithms and focus on a specific application [31, 131], a subclass of optimization functions (properties of the loss function) [32] or a particular optimization algorithm [2, 163]. These approaches can be divided into three main classes. The first class involves *input perturbation* approaches that add noise to the input data [64]. These approaches are easy to implement but often prohibit the ML model from providing accurate predictions. The second class involves *output perturbation* approaches that add noise to the model after the training procedure finishes, i.e., without modifying the vanilla training algorithm. This noise addition can be model-specific [175] or model-agnostic [13, 138]. The third class involves *inner perturbation* approaches that modify the learning algorithm such that the noise is injected during learning. One method for inner perturbation is to modify the objective of the training procedure [32]. Another approach involves adding noise to the output of each update step of the training without modifying the objective [2]. Our new DP-SCD algorithm belongs to the third class.

**DP - Empirical Risk Optimization (ERM).** Various works address the problem of ERM (similar to our setup §6.2), through the lens of differential privacy. Table 6.1 compares the utility bounds between DP-SCD and representative works for each perturbation method for DP-ERM. We simplify the bounds following [171] for easier comparison. The assumptions of these methods, described in [171, Table 1] and §6.4, are similar[5]. We highlight that the bound for DP-SCD is independent of the dimensionality of the problem ($m$) due to the dual updates, while also incorporates the mini-batch size ($L$) for quantifying the impact of the varying degree of parallelism.

Existing DP-ERM methods based on SGD typically require the tuning of an additional hyperparameter (learning rate or step size) similar to DP-SGD [2]. The value of this hyperparameter for certain loss functions can be set based on properties of these functions [175]. Furthermore regarding [175], the authors build upon permutation-based SGD and employ output perturbation, but tolerate only a constant number of iterations.

[5] DP-SCD does not require the loss function to be Lipschitz.

**Table 6.1** – Comparison of utility bounds of $(\boldsymbol{\epsilon}, \boldsymbol{\delta})$-DP algorithms for empirical risk minimization.

| Method | Perturbation Method | Utility Bound |
|---|---|---|
| [183] | Output | $\mathcal{O}\left(\frac{m}{N^2\epsilon^2}\right)$ |
| [31, 32] | Inner (objective) | $\mathcal{O}\left(\frac{m}{N^2\epsilon^2}\right)$ |
| [171] | Inner (update) | $\mathcal{O}\left(\frac{m \cdot log(N)}{N^2\epsilon^2}\right)$ |
| DP-SCD | Inner (update) | $\mathcal{O}\left(\frac{L^3 \cdot log(N/L)}{N^2\epsilon^2}\right)$ |

**Coordinate descent.** SCD algorithms do not require parameter tuning if they update one coordinate (or block) at a time, by exact minimization (or Taylor approximation). One such algorithm is SDCA [155] that is similar to DP-SCD when setting $\epsilon \to \infty, L = 1$ and $C \to \infty$. Alternative SCD algorithms take a gradient step in the coordinate direction that requires a step size [14, 129].

Parallelizable variants of SCD (such as [25, 148]) have shown remarkable speedup when deployed on multiple CPUs/GPUs [38, 90, 139, 154, 187], or multiple machines [66, 115]. These works employ sampling to select the data to be updated in parallel. DP-SCD also employs sampling via the mini-batch size ($L$), similar to the lot size of DP-SGD [2], to (a) enable parallel updates and (b) steer the privacy-utility trade-off. DP-SCD is also similar to CoCoA [115] with $K = L$ and $T^{local} = 1$. A first differentiation is that each $\Delta_j$ is computed on a single example and not on a partition of the dataset ($\mathcal{P}_j$) when split among $K$ workers. A second one is the noise addition after each CoCoA update (Steps 12 and 13 of Algorithm 1) that is necessary for differential privacy.

## 6.7 Concluding Remarks

This paper introduces the first differentially private stochastic coordinate descent algorithm (DP-SCD). We formally derive its privacy bounds and study the convergence. We show that the dual formulation of DP-SCD is preferable over the primal due to the difference in the sensitive data access patterns. We empirically compare DP-SCD against a popular SGD alternative (DP-SGD) under the same privacy guarantees. DP-SCD has a better privacy-utility trade-off than DP-SGD for ridge regression and SVMs, while DP-SCD also requires less hyperparameter tuning than DP-SGD.

Our work is only a first step towards differentially private machine learning based on coordinate descent. We empirically confirm our theoretical result by showing that convergence is achieved despite the presence of noise. The convergence rate of DP-SCD is comparable to the one of DP-SGD and we plan to also study it theoretically.

# Concluding Remarks Part V

# 7 Conclusions

We conclude this thesis by summarizing the presented results alongside their implications. Finally, we discuss directions for future research.

## 7.1 Summary and Implications

This thesis addressed three challenges that arise when distributing the training tasks of machine learning on edge devices, in order to preserve the data privacy without the need to trust a central actor. A separate part of the thesis focused on each one of these challenges.

In Part II we addressed the challenge of fast data. Services that millions of people use daily, such as news recommendations, benefit from timely updates. However, prior work minimizes the impact on the edge device by upper-bounding the frequency of the learning tasks and thus prohibits timely updates.

FLEET provides the necessary abstractions to the application layer in order to tune the trade-off between the service quality boost (controlled via the frequency and size of the learning tasks) and the impact on the edge devices in terms of latency and energy consumption. FLEET employs an asynchronous learning algorithm suitable for timely updates and an ML-based profiler that estimates the impact of a learning task on an edge device given a high degree of heterogeneity for the computing capabilities. We showed that FLEET reduces the interval between model updates from days to hours or less.

In Part III we addressed the challenge of Byzantine failures. The learning procedure is susceptible to a single edge device failing to adhere to the "correct" execution. Such failures can be attributed to a wide range of causes from simple software bugs or data distortions to adversarial users compromising the edge devices. Such failures can nullify the predictive capabilities of the learning outcome or even worse, embed a latent yet dangerous bias (e.g., anti-vaccine health "advice" [9]).

AGGREGATHOR is a framework that tackles the problem of scalable Byzantine-resilient learning in the synchronous setup and KARDAM is an algorithm that targets asynchronous updates. Both of our solutions to the Byzantine challenge employ a filtering scheme for discarding (at the server) updates sent by edge devices with suspected failures. AGGREGATHOR utilizes pair-wise distances between updates whereas KARDAM utilizes statistical properties of the learning procedure.

In Part IV we addressed the challenge of privacy guarantees. We focused on differential privacy as the framework to provide strong formal privacy guarantees. Regulating the trade-off between the level of privacy and the predictive capabilities of the learning outcome demands additional hyperparameter tuning for the learning algorithm. Hyperparameter tuning is a very costly procedure given modern ML workloads [75].

DP-SCD is a differentially private learning algorithm that reduces the tuning cost by building on top of stochastic coordinate descent. We formally showed that despite the perturbation necessary for the privacy guarantees, DP-SCD converges with a similar privacy-utility trade-off compared to the popular stochastic gradient descent-based alternative.

At a high level, this thesis takes a step towards data privacy for modern machine learning applications running on unreliable edge devices. We have demonstrated how to achieve machine learning that is (a) fast, (b) secure (i.e., Byzantine-resilient), and (c) low-cost (in terms of tuning) yet differentially private, without significant overheads in terms of (a) energy consumption and latency impact on the edge device, (b) slowdown in the learning convergence, and (c) privacy-utility trade-off.

## 7.2 Future Directions

We discussed potential extensions of the works presented in this thesis in the corresponding concluding remarks sections. In the following we introduce additional related directions that have shown promising initial results during the same doctoral studies.

**Teacher-assisted inference on edge data.** While this thesis focuses on training FL models on decentralized data across edge devices, a promising direction is the one of employing pre-trained models for inference with private data on edge devices. The idea is to employ two sets of models for the inference: a relatively small *student* model on the edge device, and a relatively more accurate (and thus typically larger) *teacher* model on the server. A binary classifier running on the edge device and for a given input (e.g., an image), determines whether the student model is likely to make a poor prediction while the teacher is likely to make a much better one. In that case and by using a framework based on multi-party computation [51], the input (private) data is encrypted and sent to server, followed by the server sending an encrypted output of the teacher back to the edge device. The binary classifier essentially

regulates the trade-off between the predictive power boost and cost (in terms of latency and energy) of employing the server model (i.e., the teacher).

**Model update compression.** In this thesis the server and the edge devices exchange models and model updates with basic (§3.2.4) or no compression. A recent line of research has shown remarkable results (in terms of bandwidth-accuracy trade-off) for compressed training with techniques such as stochastic quantization [7], distillation [143], mixed precision training [122] or even binary networks [16]. Additionally, Agarwal et al. showed that compressed training can be also differentially private [3]. A promising idea is to adapt these algorithms and combine them with the algorithms presented in this thesis for addressing the FL challenges of fast data, Byzantine resilience and privacy guarantees with significant bandwidth savings.

**Model marketplace.** While this thesis targets centralized learning on decentralized data, existing work on fully decentralized learning [102, 170] opens promising research avenues. One promising setup to explore is a network of interconnected user devices that train local models on local private data, and wish to exchange knowledge (via exchanging the local models) without releasing their private data. We envisage a marketplace with user-user transactions exchanging ML models. Model confidence values (e.g., based on Vanhaesebrouck et al. [170]) and multiple model versions can facilitate these transactions.

# Appendices Part VI

# A Supplementary Material

## A.1 Staleness Controller

In order to have a precise comparison between ADASGD and the alternative SGD algorithms, we need to control the staleness of the updates. Staleness can be only indirectly affected through hand-tuning the heterogeneity of the devices (e.g., by adding artificial delays) given the performance variability even for the same device over time. Following the more practical solution, we enforce the staleness of each update by using $\tau$CONTROLLER, an *on-demand staleness* method, useful for the design and evaluation of the staleness-aware ML algorithm on any predefined staleness distribution (with arbitrary large values) without any dependence on the actual experimental setup (e.g., number of mobile devices, device type, network).

$\tau$CONTROLLER (shown in Algorithm 4) is implemented in the global updater component (§3.2) and consists of three phases. Algorithm 4 operates with a Gaussian distribution for the predefined staleness [185] which results in a value between $\tau_{min}$ and $\tau_{max}$ with a probability of 0.997[1]. In the cold-start phase, the server performs the descent operation and saves each model version for $\tau_{max} - \tau_{min}$ times before proceeding to the on-demand staleness phase. The purpose of the on-demand staleness phase is to accumulate $R$ gradients for each of the possible outputs of the distribution. This phase obtains each missing staleness value $\tau$ by sending the corresponding saved model to the client (set by using the *Priority* counter). In the operation phase, the server performs the descent operation by choosing $R$ accumulated gradients, the staleness of which is the output of the distribution. Inevitably, the on-demand staleness method discards updates that are too stale to be selected.

[1] $\tau$CONTROLLER can be easily modified for any distribution.

**Algorithm 4:** $\tau$CONTROLLER

**Input:** $n$: mini-batch size, $\gamma_t$: sequence of learning rates, $\tau_{min}, \tau_{max}$: max and min staleness, $R$: updates aggregation window

```
1  Θ^agr = []                                                    // List of models
2  G^agr = []                          // Dictionary: (key=step, value=list of gradients)
3  Priority = 0                                   // Model version to be sent to client
   Server
4  Function Descent(G^R):
5      θ^(recent) = Θ^agr[−1]                                    // Fetch most recent model
6      θ^(new) = θ^(recent) − γ_|Θ^agr| Σ_(G,s)∈G^R Λ(|Θ^agr| − s) · G
7      return θ^(new)
8  Function Pull():
9      lock(Θ^agr)
10     θ* = Θ^agr.get(Priority)
11     unlock(Θ^agr)
12     return θ*
13 Function Push(g, s):                                          // g: gradient, s: step
14     lock(Θ^agr)
15     G^agr = G^agr[s].add(g)
       // Cold-start
16     if |Θ^agr| < τ_max and |G^agr.values()| > R then
17         G^CS = G^aggr.getRandom(R)                                  // Get R gradients
18         θ^(new) = Descent(G^CS)
19         Θ^agr = Θ^agr ∪ θ^(new)
20         Priority++
21     end
       // On-demand staleness
22     for τ ∈ [τ_min, τ_max] do
23         if τ ∉ G^agr.keyset() or |G^agr.get(|Θ^agr| − τ)| < R then
24             Priority = |Θ^agr| − τ
25         end
26     end
       // Operation phase
27     for τ ∈ [τ_min, τ_max] do
28         if τ ∈ G^agr.keyset() and |G^agr.get(|Θ^agr| − τ)| > R then
29             G^R = []
30             for m ∈ [0, R] do
31                 τ̂ = Gaussian(μ = (τ_max − τ_min)/2, σ = (τ_max − τ_min)/6)
32                 G = (G^agr.get(|Θ^agr| − τ̂), |Θ^agr| − τ̂)
33                 G^R = G^R ∪ {G}
34                 G^agr[|Θ^agr| − τ̂].remove(G)
35             end
36             θ^(new) = Descent(G^R)
37             Θ^agr = Θ^agr ∪ {θ^(new)}
38             Θ^agr.remove(0)                                         // Remove oldest model
               // Drop unused gradients
39             for τ ∈ G^agr.keyset() do
40                 if |Θ^agr| − τ) > τ_max then
41                     G^agr.remove(|Θ^agr| − τ)
42                 end
43             end
44         end
45     end
46     unlock(Θ^agr)
47     return
```

# B Detailed Proofs

## B.1 Asynchronous Byzantine Resilience

### B.1.1 Correct Cone

**Lemma 7** (Bounded statistical moments)**.** *Let $r = 2,3,4$. There exist $A'_r \geq 0$ and $B'_r \geq 0$ such that:*

$$\mathbb{E}\|\boldsymbol{Kar}_t(\boldsymbol{\theta}_t,\xi)\|^r \leq A'_r + B'_r\|\boldsymbol{\theta}_t\|^r,\ \forall t \geq 0$$

*Proof.* Note that if $\boldsymbol{Kar}_t(\boldsymbol{\theta}_t)$ comes from a honest worker, we have $\boldsymbol{Kar}_t(\boldsymbol{\theta}_t,\xi) = \boldsymbol{G}(\boldsymbol{\theta}_t,\xi)$ therefore, $(\forall t \geq 0)\mathbb{E}\|\boldsymbol{Kar}_t(\boldsymbol{\theta}_t,\xi)\|^r \leq A_r + B_r\|\boldsymbol{\theta}_t\|^r$ since by assumption on the estimator $\boldsymbol{G}$ used by honest workers, we have
$(\forall \boldsymbol{\theta} \in \mathbb{R})\mathbb{E}\|\boldsymbol{G}(\boldsymbol{\theta},\xi)\|^r \leq A_r + B_r\|\boldsymbol{\theta}\|^r$.

Let $t > 2f+1$ be any step at the parameter server. Because of the Lipschitz filter (passed by $\boldsymbol{Kar}_t$), there exists $i \leq f$ such that $\boldsymbol{Kar}_{t-i}(\boldsymbol{\theta}_{t-i})$ comes from an honest worker. Therefore, $\|\boldsymbol{\theta}_{t-i}\| \leq \|\boldsymbol{\theta}_t\| + \sum_{l=1}^{i}\gamma'_{t-l}\boldsymbol{Kar}_{t-l}(\boldsymbol{\theta}_{t-l}) \leq \|\boldsymbol{\theta}_t\| + \sum_{l=1}^{i}\gamma_{t-l}\cdot\frac{\min(\boldsymbol{Kar}_0,\|\boldsymbol{Kar}_{t-l}(\boldsymbol{\theta}_{t-l})\|)}{\|\boldsymbol{Kar}_{t-l}(\boldsymbol{\theta}_{t-l})\|}\cdot\boldsymbol{Kar}_{t-l}(\boldsymbol{\theta}_{t-l}) \leq f\cdot\boldsymbol{Kar}_0 + \|\boldsymbol{\theta}_t\|$.

So, for $r = 2,3,4$ there exists $C_r$ such that $\|\boldsymbol{\theta}_{t-i}\|^r \leq (f\cdot\boldsymbol{Kar}_0)^r + C_r\|\boldsymbol{\theta}_t\|^r$.

According to the Lipschitz criteria:

$$\begin{aligned}
&\|\boldsymbol{Kar}_t(\boldsymbol{\theta}_t)\| \\
&\leq K_t(\|\boldsymbol{\theta}_t\| + \|\boldsymbol{\theta}_{t-1}\|) + \|\boldsymbol{Kar}_{t-1}(\boldsymbol{\theta}_{t-1})\| \\
&\leq \sum_{l=1}^{i} K_{t-l+1}(\|\boldsymbol{\theta}_{t-l+1}\| + \|\boldsymbol{\theta}_{t-l}\|) + \|\boldsymbol{Kar}_{t-i}(\boldsymbol{\theta}_{t-i})\| \\
&\leq 2K\sum_{l=0}^{i}\|\boldsymbol{\theta}_{t-l}\| + \|\boldsymbol{Kar}_{t-i}(\boldsymbol{\theta}_{t-i})\|
\end{aligned}$$

$$
\begin{aligned}
&\leq 2K \sum_{l=0}^{i} \sum_{s=l}^{i-1} [\gamma'_{t-s} \cdot \|\boldsymbol{Kar}_{t-s}(\boldsymbol{\theta}_{t-s})\| + \|\boldsymbol{\theta}_{t-i}\|] + \|\boldsymbol{Kar}_{t-i}(\boldsymbol{\theta}_{t-i})\| \\
&\leq 2K \sum_{l=0}^{i} \sum_{s=l}^{i-1} \gamma_{t-s} \|\boldsymbol{Kar}_{t-s}(\boldsymbol{\theta}_{t-s})\| \cdot \frac{\min(\boldsymbol{Kar}_0, \|\boldsymbol{Kar}_{t-s}(\boldsymbol{\theta}_{t-s})\|)}{\|\boldsymbol{Kar}_{t-s}(\boldsymbol{\theta}_{t-s})\|} \\
&\quad + 2fK\|\boldsymbol{\theta}_{t-i}\| + \|\boldsymbol{Kar}_{t-i}(\boldsymbol{\theta}_{t-i})\| \\
&\leq Kf(f-1)\boldsymbol{Kar}_0 + 2fK\|\boldsymbol{\theta}_{t-i}\| + \|\boldsymbol{Kar}_{t-i}(\boldsymbol{\theta}_{t-i})\| \\
&= D + E\|\boldsymbol{\theta}_{t-i}\| + F\|\boldsymbol{Kar}_{t-i}(\boldsymbol{\theta}_{t-i})\|
\end{aligned}
$$

Where $K$ is the global Lipschitz. (We do not need to know the value of $K$ to implement ***Kar*** but we use it for the proofs.) Taking both side of the above inequality to the power $r$, we have the following for $r = 2 \ldots 4$ for constants $D_r$, $E_r$ and $F_r$:

$$\|\boldsymbol{Kar}_t(\boldsymbol{\theta}_t)\|^r \leq D_r + E_r \cdot \|\boldsymbol{\theta}_{t-i}\|^r + F_r \cdot E\|\boldsymbol{Kar}_{t-i}(\boldsymbol{\theta}_{t-i})\|^r$$

As $\boldsymbol{Kar}_{t-i}(\boldsymbol{\theta}_{t-i})$ comes from an honest worker, using the Jensen inequality and the assumption on honest workers. We can take the expected value on $\xi$.

$$
\begin{aligned}
&\mathbb{E}\|\boldsymbol{Kar}_t(\boldsymbol{\theta}_t)\|^r \\
&\leq D_r + E_r \cdot \|\boldsymbol{\theta}_{t-i}\|^r + F_r[A_r + B_r\|\boldsymbol{\theta}_{t-i}\|^r] \\
&= D_r + F_r A_r + \|\boldsymbol{\theta}_{t-i}\|^r [E_r + F_r B_r] \\
&\leq D_r + F_r A_r + [(f \cdot \boldsymbol{Kar}_0)^r + C_r\|\boldsymbol{\theta}_t\|^r] \cdot [E_r + F_r B_r] \\
&= D_r + F_r A_r + f^r \boldsymbol{Kar}_0^r [E_r + F_r B_r] + [E_r + F_r B_r] \cdot \|\boldsymbol{\theta}_t\|^r
\end{aligned}
$$

We denote by $A'_r = D_r + F_r A_r + (f \cdot \boldsymbol{Kar}_0)^r \cdot [E_r + F_r B_r]$ and $B'_r = E_r + F_r B_r$, we obtain:

$$\mathbb{E}\|\boldsymbol{Kar}_t(\boldsymbol{\theta}_t)\|^r \leq A'_r + B'_r\|\boldsymbol{\theta}_t\|^r \qquad \square$$

**Lemma 8** (Global confinement)**.** *Let $\boldsymbol{\theta}_t$ the sequence of parameter models visited by **Kar**. There exist a constant $D > 0$ such that the sequence $\boldsymbol{\theta}_t$ almost surely verifies $\|\boldsymbol{\theta}_t\| \leq D$ when $t \mapsto \infty$.*

*Proof.* Lemma 7 shows that with ***Kar***, all the assumptions of Bottou [23] (Section 5.2) are holding even in the presence of Byzantine workers, and thus, the global confinement of $\boldsymbol{\theta}_t$. $\square$

**Theorem 3** (Correct cone and bounded statistical moments)**.** *If $n > 3f + 1$ then for any $t \geq t_r$ (we show that $t_r \in \mathcal{O}(\frac{1}{K\sqrt{|\xi|}})$ where $|\xi|$ is the batch-size of honest workers):*

$$\mathbb{E}[\|\boldsymbol{Kar}_t\|^r] \leq A'_r + B'_r\|\boldsymbol{\theta}_t\|^r$$

*for any $r = 2,3,4$, constants $A'_r, B'_r$ and*

$$\langle \mathbb{E}[\boldsymbol{Kar}_t], \nabla \mathcal{F}_t \rangle = \Omega(1 - \frac{\sqrt{d}\sigma}{\|\nabla \mathcal{F}(\boldsymbol{\theta}_t)\|}) \|\nabla \mathcal{F}(\boldsymbol{\theta}_t)\|^2$$

*The expectation is on the random samples used for training.*

*Proof.* First of all, it is important to note that a Byzantine worker can lie about its Lipschitz coefficient without being able to fool the parameter server. The median Lipschitz coefficient is always bounded between the Lipschitz coefficients of two correct worker, and it is against that the gradient of the Byzantine worker would be tested to be filtered out if harmful and accepted if useful.

From Lemma 8, KARDAM acts as self-stabilizing mechanism that guarantees the global confinement of the parameter vector.

Lemma 7 have proved the first part of Theorem 3. To continue the proof of this Theorem, the goal is to find a lower bound on the scalar product between KARDAM and the real gradient of the cost. This is achieved via an upper bound on: $\|\mathbb{E}\boldsymbol{Kar}_t - \nabla \mathcal{F}_t(\boldsymbol{\theta}_t)\|$. Let $p$ the worker whose gradient estimation $\boldsymbol{g_p}$ was selected by KARDAM to be the update for step $t$ at the parameter server. According to Lemma 1, considering the latest $2f+1$ timestamps, at least $f+1$ of updates came from honest workers. Hence, there exists $i < f$ such that, $\boldsymbol{Kar}_{t-i}$ came from an honest worker. Hence, $\mathbb{E}\boldsymbol{Kar}_{t-i} = \nabla \mathcal{F}_{t-i}$. By applying the triangle inequality twice, we have:

$$\begin{aligned} \|\boldsymbol{Kar}_t - \nabla \mathcal{F}(\boldsymbol{\theta}_t)\| \leq & \|\boldsymbol{Kar}_t - \boldsymbol{Kar}_{t-i}\| \\ & + \|\boldsymbol{Kar}_{t-i} - \nabla \mathcal{F}(\boldsymbol{\theta}_{t-i})\| \\ & + \|\nabla \mathcal{F}(\boldsymbol{\theta}_{t-i}) - \nabla \mathcal{F}(\boldsymbol{\theta}_t)\| \end{aligned} \tag{B.1}$$

We know:

$$\begin{aligned} \|\boldsymbol{Kar}_{t-i} - \boldsymbol{Kar}_t\| \leq & \sum_{k=i}^{1} \|\boldsymbol{Kar}_{t-k} - \boldsymbol{Kar}_{t-k+1}\| \leq K \sum_{k=1}^{i} \|\boldsymbol{\theta}_{t-k+1} - \boldsymbol{\theta}_{t-k}\| \\ \leq & K \sum_{k=1}^{i} \gamma_{t-k} \|\boldsymbol{Kar}_{t-k}\| \leq i \cdot K \cdot \gamma_{t-i} \cdot \|\boldsymbol{Kar}\|_{max(t,i)} \end{aligned}$$

where, $\|\boldsymbol{Kar}\|_{max(t,i)}$ is the upper-bound on the norm of $\boldsymbol{Kar}$ in the list from $t-i$ to $t-1$. Since $i < f$, we have $\|\boldsymbol{Kar}_{t-i} - \boldsymbol{Kar}_t\| \leq fK\gamma_{t-i}\|\boldsymbol{Kar}\|_{max(t,i)}$. Since $\boldsymbol{\theta}_t$ is globally confined (Lemma 2), by continuous differentiability of $\mathcal{F}$, so will be $\|\nabla \mathcal{F}(\boldsymbol{\theta}_{t,i})\|$, therefore $fK\|\boldsymbol{Kar}\|_{max(t,i)}$ is bounded, and multiplies $\gamma_{t-i}$ in the right hand side of the last inequality, and we know from the hypothesis on the learning rate that $\lim_{t\to\infty} \gamma_t = 0$ (sequence of summable squares, therefore goes to zero). Since $i < f$ (and obviously, $f$, as a global variable, is independent of $t$), then we

also have $\lim_{t\to\infty}\gamma_{t-i} = 0$. This means that for every $\epsilon > 0$, eventually, the left hand-side of the above inequality is bounded by $\epsilon\|\boldsymbol{Kar}_{t-i} - \nabla\mathcal{F}(\boldsymbol{\theta}_{t-i})\|$, more precisely, since $\gamma_t$ is typically $\mathcal{O}(\frac{1}{t})$, this will hold after $t_r$ such that $t_r = \Omega(\frac{1}{\epsilon K})$.

By replacing in Inequality B.1, we get:

$$\begin{aligned}
\|\boldsymbol{Kar}_t - \nabla\mathcal{F}(\boldsymbol{\theta}_t)\| &\leq (1+\epsilon)\|\boldsymbol{Kar}_{t-i} - \nabla\mathcal{F}(\boldsymbol{\theta}_{t-i})\| + \|\nabla\mathcal{F}(\boldsymbol{\theta}_{t-i}) - \nabla\mathcal{F}(\boldsymbol{\theta}_t)\| \\
&\leq (1+\epsilon)\|\boldsymbol{Kar}_{t-i} - \nabla\mathcal{F}(\boldsymbol{\theta}_{t-i})\| + \sum_{s=1}^{i} K_{t-s}\gamma_{t-s}\|\nabla\mathcal{F}(\boldsymbol{\theta}_{t-s})\| \\
&\leq (1+\epsilon)\|\boldsymbol{Kar}_{t-i} - \nabla\mathcal{F}(\boldsymbol{\theta}_{t-i})\| + f\cdot K\cdot\gamma_{t-i}\cdot\|\nabla\mathcal{F}\|_{max(t,i)}.
\end{aligned}$$

Where $K_{t-s}$ is the real local Lipschitz coefficient of the loss function at step $t-s$. Let $j = \min(\frac{\sqrt{d}\sigma}{2}, \|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\| - \sqrt{d}\sigma)$, $C = \frac{j}{2\epsilon\sqrt{d}\sigma}$, $\epsilon' = \frac{j}{2.C}$. As $\lim_{t\to\infty}\gamma_t = 0$ and $\|\nabla\mathcal{F}\|_{max(t,i)}$ is bounded, there exist a time after which, the above quantity can be made bounded as

$$\|\boldsymbol{Kar}_t - \nabla\mathcal{F}(\boldsymbol{\theta}_t)\| \leq (1+\epsilon)\|\boldsymbol{Kar}_{t-i} - \nabla\mathcal{F}(\boldsymbol{\theta}_{t-i})\| + \epsilon'.$$

And hence:

$$\begin{aligned}
\|\mathbb{E}(\boldsymbol{Kar}_t) - \nabla\mathcal{F}(\boldsymbol{\theta}_t)\| &\leq \mathbb{E}(\|\boldsymbol{Kar}_t - \nabla\mathcal{F}(\boldsymbol{\theta}_t)\|) \\
&\leq (1+\epsilon)\mathbb{E}(\|\boldsymbol{Kar}_{t-i} - \nabla\mathcal{F}(\boldsymbol{\theta}_{t-i})\|) + \epsilon'.
\end{aligned}$$

since $\boldsymbol{Kar}_{t-i}$ comes from a correct worker, we have:

$$\mathbb{E}(\|\boldsymbol{Kar}_{t-i} - \nabla\mathcal{F}(\boldsymbol{\theta}_{t-i})\|) \leq \sqrt{d}\sigma$$

Therefore, $\|\mathbb{E}(\boldsymbol{Kar}_t) - \nabla\mathcal{F}(\boldsymbol{\theta}_t)\| \leq (1+\epsilon)\sqrt{d}\sigma + \epsilon'$. Consequently, KARDAM only selects vectors that live on average in the cone of radius $\alpha$ around the true gradient, where $\alpha$ is given by:

$\sin(\alpha) = \frac{(1+\epsilon)\sqrt{d}\sigma+\epsilon'}{\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\|}$. (as long as $\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\| > (1+\epsilon)\sqrt{d}\sigma + \epsilon'$, this has a sense)

Note:

- The $\sqrt{d}$ in $\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\| > \sqrt{d}\sigma$ is not a harsh requirement, we are using the conventional notation where $\sqrt{d}\sigma$ is the upper bound on the variance, $\sigma$ should be seen as the "component-wise" standard deviation, therefore, the norm of a non-trivial gradient is naturally larger than the vector-wise standard deviation of its estimator, which is typically $\sqrt{d}\sigma$.

- As long as the true gradient has *a nontrivial meaning* (it is larger than the standard deviation of its correct estimators), $\alpha$ is strictly bounded between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$, which

means that as long as there is no convergence to null gradients, KARDAM is selecting vectors in the correct cone around the true gradient. Most importantly, this angle shrinks to zero when the variance is too small compared to the norm of the gradient, i.e., with large batch-sizes, KARDAM boils down to be an unbiased gradient estimator. However, we only require the "component-wise" condition.

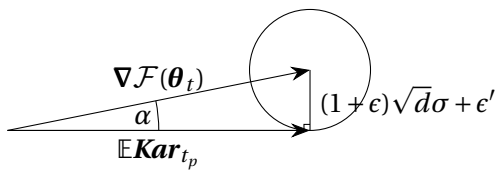



**Figure B.1 –** If $\left\|\mathbb{E}\boldsymbol{Kar}_{t_p} - \nabla\mathcal{F}(\boldsymbol{\theta}_t)\right\| \leq (1+\epsilon)\sqrt{d}\sigma + \epsilon'$ then $\langle \mathbb{E}\boldsymbol{Kar}_{t_p}, \nabla\mathcal{F}(\boldsymbol{\theta}_t)\rangle$ is upper bounded by $(1-\sin\alpha)\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\|^2$ where $\sin\alpha = \frac{(1+\epsilon)\sqrt{d}\sigma+\epsilon'}{\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\|}$.

In fact, as long as $\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\| > \sqrt{d}.\sigma$, we can consider small enough $\epsilon$ and $\epsilon'$ such that $D_1 = (1+\frac{3}{4C})\frac{\sqrt{d}\sigma}{\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\|}$, $D_2 = \frac{1}{C} + \frac{C-1}{C}\frac{\sqrt{d}\sigma}{\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\|}$ , and $\sin(\alpha) = \min(D_1, D_2) < 1$. This indeed guarantees that $\alpha < \frac{\pi}{2}$, moreover, it is enough to take $C >> \frac{\nabla\mathcal{F}(\boldsymbol{\theta}_t)\|}{\sqrt{d}\sigma}$ and $\alpha$ would satisfy $\sin(\alpha) \approx \frac{\sqrt{d}\sigma}{\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\|}$.

Actually, in a list of $L$ previous selected vectors, more than half of the vectors are from correct workers. (progress is made: liveness)

Consider a sublist of $L$ from $L_i$ to $L_j$. At the time of adding a worker in $L_j$, the frequency criteria was checked for the new addition to $L$. The active table at that time assure that in any new sublist of $L$, especially $L_i^j$), any $f$ workers appear at most $\frac{j-i}{2}$ times. As the number of Byzantine workers is maximum $f$. in sublist $L_i^j$, the Byzantine workers did less than half of the updates. In other words, at least half of the updates come from honest workers. This proves the safety of KARDAM.

The Byzantine workers may stop sending updates or send incorrect updates. In the case where the Byzantine workers stop sending updates, KARDAM still guarantees liveness. The reason is that there are at least $2f+1$ honest workers who update the model. □

### B.1.2 Convergence Analysis

We provide the convergence guarantee in terms of *ergodic convergence*, i.e., the weighted average of the $\mathcal{L}_2$ norm of all gradients ($||\nabla\mathcal{F}(\boldsymbol{\theta}_t)||^2$). For the sake of clarity in the proofs, if $X$ is a set, we also denote its cardinality by $X$.

**Remark 4** (Coordinate sum)**.** *Given a list of vectors $u_1, \dots, u_N$, we implicitly use the following inequality in our proof.*

$$\left\| \sum_{i=1}^{N} u_i \right\|^2 \leq N \cdot \sum_{i=1}^{N} \| u_i \|^2 \tag{B.2}$$

**Lemma 9** (Ergodic convergence rate)**.** *Assume that, for all steps $1 \leq t \leq T$*

$$\sum_{\lambda \in \mathbf{\Lambda}_t} \left\{ K\gamma_t^2 |\mathbf{\Lambda}_t| + \sum_{s=1}^{\infty} \sum_{\nu \in \mathbf{\Lambda}_{t+s}} \gamma_{t+s} K^2 \nu |\mathcal{G}_{t+s,\nu}| \Lambda^{-1}(\nu) \mathbb{I}_{(s \leq \Lambda^{-1}(\nu))} \gamma_t^2 |\mathbf{\Lambda}_t| \right\} \lambda^2 \leq \sum_{\lambda \in \mathbf{\Lambda}_t} \frac{\gamma_t \lambda}{|\mathcal{G}_{t\lambda}|}$$

*Then, the ergodic convergence rate is bounded as follows.*

$$\frac{\sum_{t=1}^{T} \left( \gamma_t \sum_{\lambda \in \mathbf{\Lambda}_t} \lambda \mathcal{G}_{t\lambda} \right) \mathbb{E} ||\nabla \mathcal{F}(\boldsymbol{\theta}_t)||^2}{\sum_{t=1}^{T} \gamma_t \sum_{\lambda \in \mathbf{\Lambda}_t} \lambda \mathcal{G}_{t\lambda}} \leq \frac{2(\mathcal{F}(\boldsymbol{\theta}_1) - \mathcal{F}(\boldsymbol{\theta}^*))}{\sum_{t=1}^{T} \gamma_t \sum_{\lambda \in \mathbf{\Lambda}_t} \lambda \mathcal{G}_{t\lambda}} + \frac{\left( \sum_{t=1}^{T} K\gamma_t^2 \sum_{\lambda \in \mathbf{\Lambda}_t} \lambda^2 \mathcal{G}_{t\lambda} + \gamma_t K^2 \sum_{\lambda \in \mathbf{\Lambda}_t} \lambda \mathcal{G}_{t\lambda} \sum_{j=t-\Lambda^{-1}(\lambda)}^{t-1} \gamma_j^2 \sum_{\lambda' \in \mathbf{\Lambda}_j} \lambda'^2 \mathcal{G}_{j\lambda'} \right) \cdot d \cdot \sigma^2}{\sum_{t=1}^{T} \gamma_t \sum_{\lambda \in \mathbf{\Lambda}_t} \lambda \mathcal{G}_{t\lambda}}$$

*Proof.* For the sake of concision, for every $m = [g, l] \in \mathcal{G}_{t\lambda}$, we denote by $\xi_{[t]}$ the set of $\xi$ values that the server sends during step $t$. Let $\xi_{[t,*\neq m]}$ denote the set $\xi_{[t]}$ minus the variable $\xi$ corresponding to message $m$. Additionally, $G[tm] := \boldsymbol{G}(\boldsymbol{\theta}_{t-\tau_{tl}}, \xi)$ and $\nabla \mathcal{F}[tm] := \nabla \mathcal{F}(\boldsymbol{\theta}_{t-\tau_{tl}})$.

A second order expansion of $\mathcal{F}$, followed by the application of the Lipschitz inequality to $\nabla \mathcal{F}$ yields the following.

$$\begin{aligned} \mathcal{F}(\boldsymbol{\theta}_{t+1}) - \mathcal{F}(\boldsymbol{\theta}_t) &\leq \langle \nabla \mathcal{F}(\boldsymbol{\theta}_t), \boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t \rangle + \frac{K}{2} \| \boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t \|^2 \\ &\leq -\gamma_t \sum_{\mathbf{\Lambda}_t} \lambda \mathcal{G}_{t\lambda} \langle \nabla \mathcal{F}(\boldsymbol{\theta}_t), \frac{1}{\mathcal{G}_{t\lambda}} \sum_{\mathcal{G}_{t\lambda}} G[tm] \rangle + \frac{K}{2} \gamma_t^2 \left\| \sum_{\mathbf{\Lambda}_t} \lambda \sum_{\mathcal{G}_{t\lambda}} G[tm] \right\|^2 \end{aligned}$$

Taking the expectation and using the correct cone property, we have:

$$\begin{aligned} \mathbb{E}_{\xi_{[t]}} \mathcal{F}(\boldsymbol{\theta}^{(t+1)}) - \mathcal{F}(\boldsymbol{\theta}_t) &\leq -\gamma_t \sum_{\mathbf{\Lambda}_t} \lambda \mathcal{G}_{t\lambda} \langle \nabla \mathcal{F}(\boldsymbol{\theta}_t), \frac{1}{\mathcal{G}_{t\lambda}} \sum_{\mathcal{G}_{t\lambda}} \nabla \mathcal{F}[tm] \rangle \\ &+ \frac{K}{2} \gamma_t^2 \mathbb{E}_{\xi_{[t]}} \left\| \sum_{\mathbf{\Lambda}_t} \lambda \sum_{\mathcal{G}_{t\lambda}} G[tm] \right\|^2 \end{aligned}$$

Using $\langle a, b\rangle = \frac{||a||^2+||b||^2-||a-b||^2}{2}$, we obtain the following inequality.

$$\begin{aligned}
\mathbb{E}_{\xi_{[t]}}\mathcal{F}(\boldsymbol{\theta}_{t+1}) - \mathcal{F}(\boldsymbol{\theta}_t) \leq & -\frac{\gamma_t}{2}\sum_{\boldsymbol{\Lambda}_t}\lambda\mathcal{G}_{t\lambda}\,\|\boldsymbol{\nabla}\mathcal{F}(\boldsymbol{\theta}_t)\|^2 \\
& -\frac{\gamma_t}{2}\sum_{\boldsymbol{\Lambda}_t}\lambda\mathcal{G}_{t\lambda}\left\|\frac{1}{\mathcal{G}_{t\lambda}}\sum_{\mathcal{G}_{t\lambda}}\boldsymbol{\nabla}\mathcal{F}[tm]\right\|^2 + \frac{K\gamma_t^2}{2}\underbrace{\mathbb{E}_{\xi_{[t]}}\left\|\sum_{\boldsymbol{\Lambda}_t}\lambda\sum_{\mathcal{G}_{t\lambda}}G[tm]\right\|^2}_{S_1} \\
& +\frac{\gamma_t}{2}\sum_{\boldsymbol{\Lambda}_t}\lambda\mathcal{G}_{t\lambda}\underbrace{\left\|\boldsymbol{\nabla}\mathcal{F}(\boldsymbol{\theta}_t) - \frac{1}{\mathcal{G}_{t\lambda}}\sum_{\mathcal{G}_{t\lambda}}\boldsymbol{\nabla}\mathcal{F}[tm]\right\|^2}_{S_2}
\end{aligned}$$

We now define two terms $S_1$ and $S_2$ as follows.

$$\begin{aligned}
S_1 &= \mathbb{E}_{\xi_{[t]}}\left\|\sum_{\boldsymbol{\Lambda}_t}\lambda\sum_{\mathcal{G}_{t\lambda}}(G[tm]-\boldsymbol{\nabla}\mathcal{F}[tm]) + \sum_{\boldsymbol{\Lambda}_t}\lambda\sum_{\mathcal{G}_{t\lambda}}\boldsymbol{\nabla}\mathcal{F}[tm]\right\|^2 \\
&= \mathbb{E}_{\xi_{[t]}}\left\|\sum_{\boldsymbol{\Lambda}_t}\lambda\sum_{\mathcal{G}_{t\lambda}}(G[tm]-\boldsymbol{\nabla}\mathcal{F}[tm])\right\|^2 + \mathbb{E}_{\xi_{[m]}}\left\|\sum_{\boldsymbol{\Lambda}_t}\lambda\sum_{\mathcal{G}_{t\lambda}}\boldsymbol{\nabla}\mathcal{F}[tm]\right\|^2 \\
&\quad + 2\mathbb{E}_{\xi_{[t]}}\langle\sum_{\boldsymbol{\Lambda}_t}\lambda\sum_{\mathcal{G}_{t\lambda}}(G[tm]-\boldsymbol{\nabla}\mathcal{F}[tm]), \sum_{\boldsymbol{\Lambda}_t}\lambda\sum_{\mathcal{G}_{t\lambda}}\boldsymbol{\nabla}\mathcal{F}[tm]\rangle \\
&= \mathbb{E}_{\xi_{[t]}}\left\|\sum_{\boldsymbol{\Lambda}_t}\lambda\sum_{\mathcal{G}_{t\lambda}}(G[tm]-\boldsymbol{\nabla}\mathcal{F}[tm])\right\|^2 + \mathbb{E}_{\xi_{[t]}}\left\|\sum_{\boldsymbol{\Lambda}_t}\lambda\sum_{\mathcal{G}_{t\lambda}}\boldsymbol{\nabla}\mathcal{F}[tm]\right\|^2 \\
&\quad + 2\langle\sum_{\boldsymbol{\Lambda}_t}\lambda\sum_{\mathcal{G}_{t\lambda}}(\boldsymbol{\nabla}\mathcal{F}[tm]-\boldsymbol{\nabla}\mathcal{F}[tm]), \sum_{\boldsymbol{\Lambda}_t}\lambda\sum_{\mathcal{G}_{t\lambda}}\boldsymbol{\nabla}\mathcal{F}[tm]\rangle \\
&= \underbrace{\mathbb{E}_{\xi_{[t]}}\left\|\sum_{\boldsymbol{\Lambda}_t}\lambda\sum_{\mathcal{G}_{t\lambda}}(G[tm]-\boldsymbol{\nabla}\mathcal{F}[tm])\right\|^2}_{A_1} + \underbrace{\mathbb{E}_{\xi_{[t]}}\left\|\sum_{\boldsymbol{\Lambda}_t}\lambda\sum_{\mathcal{G}_{t\lambda}}\boldsymbol{\nabla}\mathcal{F}[tm]\right\|^2}_{A_2}
\end{aligned}$$

Regarding $A_2$, applying Equation B.2 yields the following inequality.

$$A_2 \leq \mathbb{E}_{\xi_{[t]}}\boldsymbol{\Lambda}_t\cdot\sum_{\boldsymbol{\Lambda}_t}\lambda^2\|\sum_{\mathcal{G}_{t\lambda}}\boldsymbol{\nabla}\mathcal{F}[tm]\|^2 \leq \boldsymbol{\Lambda}_t\cdot\sum_{\boldsymbol{\Lambda}_t}\lambda^2\mathbb{E}_{\xi_{[t]}}\|\sum_{\mathcal{G}_{t\lambda}}\boldsymbol{\nabla}\mathcal{F}[tm]\|^2$$

Regarding $A_1$, the term $\|\ldots\|^2$ is expressed as a scalar product and expanded as follows.

$$\begin{aligned}
A_1 &= \mathbb{E}_{\xi_{[t]}}\sum_{\lambda,\lambda'\in\boldsymbol{\Lambda}_t}\Big(\sum_{\substack{m\in\mathcal{G}_{t\lambda},\\ m'\in\mathcal{G}_{t\lambda'}}}\lambda\lambda'\cdot\langle G[tm]-\boldsymbol{\nabla}\mathcal{F}[tm], G[tm']-\boldsymbol{\nabla}\mathcal{F}[tm']\rangle\Big) \\
&= \text{diagonal} + \text{off-diagonal}
\end{aligned}$$

$$= \sum_{\lambda \in \mathbf{\Lambda}_t} \sum_{m \in \mathcal{G}_{t\lambda}} \lambda^2 \cdot \mathbb{E}_{\xi_{[t]}} \|G[tm] - \nabla \mathcal{F}[tm]\|^2 + \mathbb{E}_{\xi_{[t,m' \neq m]}} \left( \mathbb{E}_{\xi} \langle G[tm] - \nabla \mathcal{F}[tm], G[tm'] - \nabla \mathcal{F}[tm'] \rangle \right)$$

$$\leq \sum_{\mathbf{\Lambda}_t} \lambda^2 \mathcal{G}_{t\lambda} \cdot d \cdot \sigma^2 + d \cdot \sigma^2 + 2DK\sigma\sqrt{d} + K^2 D^2$$

The sum over the off-diagonal terms (i.e., $(\lambda, m) \neq (\lambda', m')$) is bounded by $d \cdot \sigma^2 + 2DK\sigma\sqrt{d} + K^2 D^2$. Moreover, if $\lambda \neq \lambda'$, then $m \neq m'$ because $\mathcal{G}_{t\lambda}$ and $\mathcal{G}_{t\lambda'}$ are disjoint sets and thus for any off-diagonal pair $(\lambda, m), (\lambda, m')$ we have $m \neq m'$.

$$\begin{aligned}
&\mathbb{E}_{\xi_{[t]}} \langle G[tm] - \nabla \mathcal{F}[tm], G[tm'] - \nabla \mathcal{F}[tm'] \rangle \\
&\quad = \mathbb{E}_{\xi_{[t,m' \neq m]}} \left( \mathbb{E}_{\xi} \langle G[tm] - \nabla \mathcal{F}[tm], G[tm'] - \nabla \mathcal{F}[tm'] \rangle \right) \\
&\quad = \mathbb{E}_{\xi_{[t,m' \neq m]}} \langle \mathbb{E}_{\xi} G[tm] - \nabla \mathcal{F}[tm], G[tm'] - \nabla \mathcal{F}[tm'] \rangle \\
&\quad = \mathbb{E}_{\xi_{[t,m' \neq m]}} (\langle \mathbb{E}_{\xi} G[tm], G[tm'] \rangle - \langle \nabla \mathcal{F}[tm], G[tm'] \rangle - \langle \mathbb{E}_{\xi} G[tm], \nabla \mathcal{F}[tm'] \rangle + \langle \nabla \mathcal{F}[tm], \nabla \mathcal{F}[tm'] \rangle) \\
&\quad \leq \mathbb{E}_{\xi_{[t,m' \neq m]}} (\|\mathbb{E}_{\xi} G[tm]\| \cdot \|G[tm']\| + \|\nabla \mathcal{F}[tm]\| \cdot \|G[tm']\| \\
&\qquad + \|\mathbb{E}_{\xi} G[tm]\| \cdot \|\nabla \mathcal{F}[tm']\| + \|\nabla \mathcal{F}[tm]\| \cdot \|\nabla \mathcal{F}[tm']\|) \\
&\quad \leq d \cdot \sigma^2 + 2DK\sigma\sqrt{d} + K^2 D^2
\end{aligned}$$

Hence, we obtain the following inequalities for $S_1$ and $S_2$.

$$S_1 \leq \sum_{\mathbf{\Lambda}_t} \lambda^2 \mathcal{G}_{t\lambda} \cdot d \cdot \sigma^2 + \mathbf{\Lambda}_t \cdot \sum_{\mathbf{\Lambda}_t} \lambda^2 \mathbb{E}_{\xi_{[t]}} \|\sum_{\mathcal{G}_{t\lambda}} \nabla \mathcal{F}[tm]\|^2 + d \cdot \sigma^2 + 2DK\sigma\sqrt{d} + K^2 D^2$$

$$S_2 \leq \left\| \frac{1}{\mathcal{G}_{t\lambda}} \sum_{\mathcal{G}_{t\lambda}} \nabla \mathcal{F}(\boldsymbol{\theta}_t) - \nabla \mathcal{F}[tm] \right\|^2$$

Recall that, since $m = [g, l] \in \mathcal{G}_{t\lambda}$, we have $\nabla \mathcal{F}[tm] = \nabla \mathcal{F}(\boldsymbol{\theta}_{t-\tau_{tl}})$. By applying the Lipschitz inequality, we get:

$$\begin{aligned}
S_2 &\leq K^2 \|\boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-\Lambda^{-1}(\lambda)}\|^2 \\
&\leq K^2 \left\| \sum_{j=t-\Lambda^{-1}(\lambda)}^{t-1} \boldsymbol{\theta}_{j+1} - \boldsymbol{\theta}_j \right\|^2 \leq K^2 \left\| \sum_{j=t-\Lambda^{-1}(\lambda)}^{t-1} \gamma_j \sum_{\nu \in \mathbf{\Lambda}_j} \nu \sum_{\mathcal{G}_{j\nu}} G[jm] \right\|^2 \\
&\leq K^2 \underbrace{\left\| \sum_{j=t-\Lambda^{-1}(\lambda)}^{t-1} \gamma_j \sum_{\nu \in \mathbf{\Lambda}_j} \nu \sum_{\mathcal{G}_{j\nu}} (G[jm] - \nabla \mathcal{F}[jm]) \right\|^2}_{S_3 = \|a\|^2}
\end{aligned}$$

$$+ K^2 \underbrace{\left\| \sum_{j=t-\Lambda^{-1}(\lambda)}^{t-1} \gamma_j \sum_{\nu \in \mathbf{\Lambda}_j} \nu \sum_{\mathcal{G}_{j\nu}} \nabla \mathcal{F}[jm] \right\|^2}_{S_4 = \|b\|^2} + 2K^2 \langle a, b \rangle$$

Hence, we obtain the following inequalities for $S_3$ and $S_4$.

$$\mathbb{E}_{\xi_{[j]},\ldots} S_3 \leq \sum_{j=t-\Lambda^{-1}(\lambda)}^{t-1} \gamma_j^2 \sum_{\mathbf{\Lambda}_j} \nu^2 \mathcal{G}_{j\nu} \cdot d \cdot \sigma^2 \quad \text{(cross-products vanish)}$$

$$\mathbb{E}_{\xi_{[j]},\ldots} S_4 \leq \Lambda^{-1}(\lambda) \sum_{j=k-\Lambda^{-1}(\lambda)}^{t-1} \gamma_j^2 \mathbf{\Lambda}_j \sum_{\mathbf{\Lambda}_j} \nu^2 \mathbb{E} \left\| \sum_{\mathcal{G}_{j\nu}} \nabla \mathcal{F}[jm'] \right\|^2 \quad \text{(by Equation B.2).}$$

Moreover, we have $\mathbb{E}_* \langle a, b \rangle = \langle \mathbb{E}_* a, b \rangle = 0$.

$$\begin{aligned} \mathbb{E} S_2 \leq K^2 & \sum_{j=t-\Lambda^{-1}(\lambda)}^{t-1} \gamma_j^2 \sum_{\mathbf{\Lambda}_j} \nu^2 \mathcal{G}_{j\nu} \cdot d \cdot \sigma^2 \\ & + K^2 \Lambda^{-1}(\lambda) \sum_{j=t-\Lambda^{-1}(\lambda)}^{t-1} \gamma_j^2 \mathbf{\Lambda}_j \sum_{\mathbf{\Lambda}_j} \nu^2 \mathbb{E} \left\| \sum_{\mathcal{G}_{j\nu}} \nabla \mathcal{F}[jm'] \right\|^2 \end{aligned}$$

$$\begin{aligned} \mathbb{E}_{\xi_{[t]}} \mathcal{F}(\boldsymbol{\theta}_{t+1}) - \mathcal{F}(\boldsymbol{\theta}_t) \leq & -\frac{\gamma_t}{2} \sum_{\mathbf{\Lambda}_t} \lambda \mathcal{G}_{t\lambda} \|\nabla \mathcal{F}(\boldsymbol{\theta}_t)\|^2 \\ & + \sum_{\mathbf{\Lambda}_t} \left( \frac{K \gamma_t^2 \mathbf{\Lambda}_t \lambda^2}{2} - \frac{\gamma_t \lambda}{2 \mathcal{G}_{t\lambda}} \right) \mathbb{E} \left\| \sum_{\mathcal{G}_{t\lambda}} \nabla \mathcal{F}[tm] \right\|^2 \\ & + \left( \frac{K \gamma_t^2}{2} \sum_{\mathbf{\Lambda}_t} \lambda^2 \mathcal{G}_{t\lambda} + \frac{\gamma_t K^2}{2} \sum_{\mathbf{\Lambda}_t} \lambda \mathcal{G}_{t\lambda} \sum_{j=t-\Lambda^{-1}(\lambda)}^{t-1} \gamma_j^2 \sum_{\mathbf{\Lambda}_j} \nu^2 \mathcal{G}_{j\nu} \right) \cdot d \cdot \sigma^2 \\ & + \frac{\gamma_t K^2}{2} \sum_{\mathbf{\Lambda}_t} \lambda \mathcal{G}_{t\lambda} \Lambda^{-1}(\lambda) \sum_{j=t-\Lambda^{-1}(\lambda)}^{t-1} \gamma_j^2 \mathbf{\Lambda}_j \sum_{\mathbf{\Lambda}_j} \nu^2 \mathbb{E} \left\| \sum_{j\nu} \nabla \mathcal{F}[jm'] \right\|^2 \end{aligned}$$

Summing for $t = 1, \ldots, T$, we arrive at the following inequality.

$$\begin{aligned} \mathbb{E} \mathcal{F}(\boldsymbol{\theta}_{t+1}) - \mathcal{F}(\boldsymbol{\theta}_1) \leq & -\sum_t \frac{1}{2} \left( \gamma_t \sum_{\mathbf{\Lambda}_t} \lambda \mathcal{G}_{t\lambda} \right) \|\nabla \mathcal{F}(\boldsymbol{\theta}_t)\|^2 \\ & + \sum_t \left( \frac{K \gamma_t^2}{2} \sum_{\mathbf{\Lambda}_t} \lambda^2 \mathcal{G}_{t\lambda} + \frac{\gamma_t K^2}{2} \sum_{\mathbf{\Lambda}_t} \lambda \mathcal{G}_{t\lambda} \sum_{j=t-\Lambda^{-1}(\lambda)}^{t-1} \gamma_j^2 \sum_{\mathbf{\Lambda}_j} \nu^2 \mathcal{G}_{j\nu} \right) \cdot d \cdot \sigma^2 \\ & + \sum_t \sum_{\mathbf{\Lambda}_t} \left( \frac{K \gamma_t^2 \mathbf{\Lambda}_t \lambda^2}{2} - \frac{\gamma_t \lambda}{2 \mathcal{G}_{t\lambda}} \right) \mathbb{E} \left\| \sum_{\mathcal{G}_{t\lambda}} \nabla \mathcal{F}[tm] \right\|^2 \end{aligned}$$

$$+\sum_t\left(\sum_{s=1}^{\infty}\sum_{\mathbf{\Lambda}_{t+s}}\gamma_{t+s}K^2\nu\mathcal{G}_{t+s,\nu}\Lambda^{-1}(\nu)\mathbb{I}(s\le\Lambda^{-1}(\nu))\right)\frac{\gamma_t\mathbf{\Lambda}_t\lambda^2}{2}\mathbb{E}\left\|\sum_{\mathcal{G}_{t\lambda}}\nabla\mathcal{F}[tm]\right\|^2$$

The last term comes from the following observation.

$$\begin{aligned}
&\sum_{t=1}^{T}\sum_{\mathbf{\Lambda}_t}\sum_{s=1}^{\infty}\mathcal{F}_t^{\lambda}Z_{t-s}\mathbb{I}(s\le\Lambda^{-1}(\lambda))=\sum_{s=1}^{\infty}\sum_{t=1}^{T}\sum_{\mathbf{\Lambda}_t}\mathcal{F}_t^{\lambda}Z_{t-s}\mathbb{I}(s\le\Lambda^{-1}(\lambda))\\
&=\sum_{s=1}^{\infty}\sum_{l=1-s}^{T-s}\sum_{\mathbf{\Lambda}_{l+s}}\mathcal{F}_{l+s}^{\lambda}Z_l\mathbb{I}(s\le\Lambda^{-1}(\lambda))=\sum_{s=1}^{\infty}\sum_{t=1}^{T}\sum_{\mathbf{\Lambda}_{t+s}}\mathcal{F}_{t+s}^{\lambda}Z_t\mathbb{I}(s\le\Lambda^{-1}(\lambda))\\
&=\sum_{t=1}^{T}\left(\sum_{s=1}^{\infty}\sum_{\mathbf{\Lambda}_{t+s}}\mathcal{F}_{t+s}^{\lambda}\mathbb{I}(s\le\Lambda^{-1}(\lambda))\right)Z_t
\end{aligned}$$

Since the two last terms sum to a non-positive value, we arrive at the following inequality.

$$\begin{aligned}
&\sum_t\frac{1}{2}\left(\gamma_t\sum_{\mathbf{\Lambda}_t}\lambda\mathcal{G}_{t\lambda}\right)\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\|^2\le\mathcal{F}(\boldsymbol{\theta}_1)-\mathcal{F}(\boldsymbol{\theta}^*)\\
&\quad+\sum_t\left(\frac{K\gamma_t^2}{2}\sum_{\mathbf{\Lambda}_t}\lambda^2\mathcal{G}_{t\lambda}+\frac{\gamma_tK^2}{2}\sum_{\mathbf{\Lambda}_t}\lambda\mathcal{G}_{t\lambda}\sum_{j=t-\Lambda^{-1}(\lambda)}^{t-1}\gamma_j^2\sum_{\mathbf{\Lambda}_j}\nu^2\mathcal{G}_{j\nu}\right)\cdot d\cdot\sigma^2+\mathcal{O}(\frac{1}{K\cdot\sqrt{|\xi|}})
\end{aligned}$$

□

**Theorem 4** (Convergence guarantee)**.** *We express the convergence guarantee in terms of the ergodic convergence, i.e., the weighted average of the $\mathcal{L}_2$ norm of all gradients ($||\nabla\mathcal{F}(\boldsymbol{\theta}_t)||^2$). Using the above-mentioned assumptions, and the maximum adaptive rate $\phi_{\max}=\max\{\phi_1,\dots,\phi_t\}$, we get the following bound on the convergence rate.*

$$\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\|^2\le\left(2+\phi_{\max}+\gamma KM\chi\phi_{\max}\right)\gamma Kd\sigma^2+d\sigma^2+2DK\sigma\sqrt{d}+K^2D^2$$

*under the prerequisite that*

$$\sum_{\lambda\in\mathbf{\Lambda}_t}\lambda^2|\mathbf{\Lambda}_t|\left\{K\gamma_t^2+\sum_{s=1}^{\infty}(\sum_{\nu\in\mathbf{\Lambda}_{t+s}}\gamma_{t+s}K^2\nu|\mathcal{G}_{t+s,\nu}|\Lambda^{-1}(\nu)\mathbb{I}_{(s\le\Lambda^{-1}(\nu))}\gamma_t^2)\right\}\le\sum_{\lambda\in\mathbf{\Lambda}_t}\frac{\gamma_t\lambda}{|\mathcal{G}_{t\lambda}|}$$

*where the Iverson indicator function is defined as follows*

$$\mathbb{I}_{(s\le\Delta)}=\begin{cases}1 & \textit{if } s\le\Delta\\ 0 & \textit{otherwise.}\end{cases}$$

*and $\chi$ denotes a constant such that for all $\tau_{tl}$, the following inequality holds:*

$$\tau_{tl} \cdot \Lambda(\tau_{tl}) \leq \chi \tag{5.3}$$

*Proof.* We first recall Definition 10, which introduces the adaptive learning rate schedule, before we prove Theorem 4 via employing Lemma 9. Due to the choice of the learning rate (Definition 10), the inequality in Theorem 4 reduces to the following inequality.

$$\frac{1}{T}\sum_{t=1}^{T} \mathbb{E}\|\mathcal{F}(\boldsymbol{\theta}_t)\|^2 \leq S_5 + S_6 + S_7$$

First, we obtain the following equality for $S_5$.

$$S_5 = \frac{2(\mathcal{F}(\boldsymbol{\theta}_1) - \mathcal{F}(\boldsymbol{\theta}^*))}{\sum_{t=1}^{T} \gamma_t \sum_{\lambda \in \boldsymbol{\Lambda}_t} \lambda \mathcal{G}_{t\lambda}} = \frac{2\gamma^2 KTR \cdot d \cdot \sigma^2}{\gamma TR} = 2\gamma K \cdot d \cdot \sigma^2$$

Regarding $S_6$, we obtain the following inequality.

$$\begin{aligned}
S_6 &= \frac{\sum_{t=1}^{T} K\gamma_t^2 \sum_{\lambda \in \boldsymbol{\Lambda}_t} \lambda^2 \mathcal{G}_{t\lambda}}{\sum_{t=1}^{T} \gamma_t \sum_{\lambda \in \boldsymbol{\Lambda}_t} \lambda \mathcal{G}_{t\lambda}} \cdot d \cdot \sigma^2 = \frac{K\gamma^2 \sum_{t=1}^{T} \phi_t^2 \sum_{\lambda \in \boldsymbol{\Lambda}_t} \lambda^2 \mathcal{G}_{t\lambda}}{\gamma TR} \cdot d \cdot \sigma^2 \\
&\leq \frac{K\gamma^2 \sum_{t=1}^{T} \phi_t^2 \sum_{\lambda \in \boldsymbol{\Lambda}_t} \lambda \mathcal{G}_{t\lambda}}{\gamma TR} \cdot d \cdot \sigma^2 \text{ (since } \lambda^2 \leq \lambda \leq 1) \\
&\leq \frac{K\gamma^2 TR\phi_{\max}}{\gamma TR} \cdot d \cdot \sigma^2 = \phi_{\max}\gamma K \cdot d \cdot \sigma^2
\end{aligned}$$

Finally, we obtain the following inequality for $S_7$.

$$\begin{aligned}
S_7 &= \frac{\sum_{t=1}^{T} \gamma_t K^2 \sum_{\lambda \in \boldsymbol{\Lambda}_t} \lambda \mathcal{G}_{t\lambda} \sum_{j=t-\Lambda^{-1}(\lambda)}^{t-1} \gamma_j^2 \sum_{\lambda' \in \boldsymbol{\Lambda}_j} \lambda'^2 R_{j\lambda'}}{\gamma TR} \cdot d \cdot \sigma^2 \\
&\leq \frac{K^2\gamma^3 \sum_{t=1}^{T} \phi_t \sum_{\lambda \in \boldsymbol{\Lambda}_t} \lambda \mathcal{G}_{t\lambda} \sum_{j=t-\Lambda^{-1}(\lambda)}^{t-1} \phi_j^2 \sum_{\lambda' \in \boldsymbol{\Lambda}_j} \lambda'^2 R_{j\lambda'}}{\gamma TR} \cdot d \cdot \sigma^2 \\
&\leq \frac{K^2\gamma^3 \sum_{t=1}^{T} \sum_{\lambda \in \boldsymbol{\Lambda}_t} \lambda \mathcal{G}_{t\lambda} R\Lambda^{-1}(\lambda)\phi_{\max}}{\gamma TR} \cdot d \cdot \sigma^2 \\
&\leq \frac{K^2\gamma^3 \sum_{t=1}^{T} \sum_{\lambda \in \boldsymbol{\Lambda}_t} \mathcal{G}_{t\lambda} R\chi\phi_{\max}}{\gamma TR} \cdot d \cdot \sigma^2 \leq \frac{K^2\gamma^3 TR^2\chi\phi_{\max}}{\gamma TR} \cdot d \cdot \sigma^2 \\
&\leq \gamma^2 K^2 R\chi\phi_{\max} \cdot d \cdot \sigma^2
\end{aligned}$$

Hence, we prove the ergodic convergence rate.

$$\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}\|\nabla\mathcal{F}(\boldsymbol{\theta}_t)\|^2 \leq \left(2+\phi_{\max}+\gamma KR\chi\phi_{\max}\right)\cdot\gamma K\cdot d\cdot\sigma^2 + d\cdot\sigma^2 + 2DK\sigma\sqrt{d} + K^2D^2$$

□

**Theorem 5** (Convergence time complexity)**.** *Given any mini-batch size $|\xi|$, the number of gradients $M$ the server waits for before updating the model, and the total number of steps $T$, the time complexity for the convergence of* KARDAM *is:*

$$\mathcal{O}\left(\frac{\phi_{\max}}{\sqrt{T|\xi|M}} + \frac{\chi\phi_{\max}}{T} + d\sigma^2 + 2DK\sigma\sqrt{d} + K^2D^2\right)$$

*Proof.* Substituting the value of $\gamma$ from Definition 10 in RHS of Theorem 4, we get the following.

$$\begin{aligned}&\left(2+\phi_{\max}+\gamma KR\chi\phi_{\max}\right)\cdot\gamma K\cdot d\cdot\sigma^2 + d\cdot\sigma^2 + 2DK\sigma\sqrt{d} + K^2D^2\\ &= \mathcal{O}\left(\frac{\phi_{\max}}{\sqrt{T\cdot|\xi|\cdot R}} + \frac{\chi\cdot\phi_{\max}}{T} + d\cdot\sigma^2 + 2DK\sigma\sqrt{d} + K^2D^2\right)\end{aligned}$$

Note that $\sigma = \mathcal{O}(1/\sqrt{|\xi|})$ (Assumption 2) and therefore the bound is also dependent on $n$.

□

**Remark 2** (Dampening comparison)**.** *Given two dampening functions $\Lambda_1(\tau) = \frac{1}{1+\tau}$ and $\Lambda_2(\tau) = exp(-\alpha\sqrt[\beta]{\tau})$, and the convergence time complexity from Theorem 5, $\Lambda_2(\tau)$ converges faster than $\Lambda_1(\tau)$ when $\frac{\beta}{e} < \alpha \leq \frac{ln(\tau+1)}{\sqrt[\beta]{\tau}}$.*

*Proof.* From Inequality 5.3, we have the following for $\Lambda_1$ and $\Lambda_2$.

$$\begin{aligned}\chi_1 &= \max_{\tau}\left\{\frac{\tau}{\tau+1}\right\}\\ \chi_2 &= \max_{\tau}\left\{\tau\cdot exp(-\alpha\sqrt[\beta]{\tau})\right\}\end{aligned}$$

The maximum value of $\{\tau\cdot exp(-\alpha\sqrt[\beta]{\tau})\}$ is $\left(\frac{\beta}{e\alpha}\right)^{\beta}$ when $\tau = \left(\frac{\beta}{\alpha}\right)^{\beta}$. We get that $\chi_1 \geq \chi_2$ when the following holds.

$$\frac{\tau}{\tau+1} \geq \left(\frac{\beta}{e\alpha}\right)^{\beta}$$

Hence, from the above inequality, we get the following.

$$\tau \geq \frac{1}{\left(\frac{e\alpha}{\beta}\right)^{\beta} - 1}$$

Note that since $\tau > 0$, we get $\left(\frac{e\alpha}{\beta}\right)^{\beta} > 1$ which leads to the following lower bound on $\alpha$.

$$\alpha > \frac{\beta}{e} \tag{B.2}$$

Furthermore, for the $\phi_{max}$ terms, we compare the values between the two dampening functions.

$$\phi_1 = \max_{\tau}\left\{\frac{R}{\sum_{\mathbf{\Lambda}_t} \lambda \cdot |\mathcal{G}_{t\lambda}|}\right\} = \max_{\tau}\left\{\frac{R}{\sum_{\tau} \frac{1}{\tau+1} \cdot |\mathcal{G}_{t\lambda}|}\right\}$$
$$\phi_2 = \max_{\tau}\left\{\frac{R}{\sum_{\mathbf{\Lambda}_t} \lambda \cdot |\mathcal{G}_{t\lambda}|}\right\} = \max_{\tau}\left\{\frac{R}{\sum_{\tau} exp(-\alpha \sqrt[\beta]{\tau}) \cdot |\mathcal{G}_{t\lambda}|}\right\}$$

Hence, for $\phi_1 \geq \phi_2$, we need to show that $\frac{1}{\tau+1} \leq exp(-\alpha \sqrt[\beta]{\tau})$, i.e., $\tau + 1 \geq exp(\alpha \sqrt[\beta]{\tau})$. The relation holds for any $\alpha$ with the upper bound as follows.

$$\alpha \leq \frac{ln(\tau+1)}{\sqrt[\beta]{\tau}} \tag{B.3}$$

From Inequalities B.2 and B.3, we get the following.

$$\frac{\beta}{e} < \alpha \leq \frac{ln(\tau+1)}{\sqrt[\beta]{\tau}}$$

One possible setting is $\beta \approx 1.85$ when $1 \leq \tau \leq 10$, $\beta \approx 3.1$ when $11 \leq \tau \leq 33$, and $\beta \approx 4$ when $34 \leq \tau \leq 75$. Given these values of $\beta$ and $\tau$, $\Lambda_2(\tau)$ has a smaller convergence time complexity (Theorem 5) than $\Lambda_1(\tau)$. Hence, $\Lambda_2(\tau)$ converges faster than $\Lambda_1(\tau)$.

□

## B.2 Differentially Private Stochastic Coordinate Descent

### B.2.1 Objective Decrease Lower Bound

**Lemma 5** (Update step - objective decrease lower bound)**.** *Assuming $\ell_i$ are $1/\mu$-smooth, then the update step of Algorithm 1 decreases the objective, even if computed based on a noisy version*

*of $\boldsymbol{\alpha}, \boldsymbol{v}$. The decrease is as follows:*

$$\mathbb{E}[\mathcal{S}(\boldsymbol{\alpha}) - \mathcal{S}(\boldsymbol{\alpha} + \Delta\dot{\boldsymbol{\alpha}})] \geq \frac{\mu\lambda L}{\mu\lambda N + L}\mathbb{E}[\mathcal{S}(\boldsymbol{\alpha})] \tag{6.4}$$

*where $\mathcal{S}$ denotes the dual suboptimality defined as: $\mathcal{S}(\boldsymbol{\alpha}) := \mathcal{F}^*(\boldsymbol{\alpha}) - \min_{\boldsymbol{\alpha}} \mathcal{F}^*(\boldsymbol{\alpha})$.*

*Proof.* Consider the optimization step $\boldsymbol{\alpha}_i \to \hat{\boldsymbol{\alpha}}_{i+1}$ (Sequence 6.3) that employs a set of unscaled updates $(\Delta\dot{\boldsymbol{\alpha}}_j)$. Given the $\mu$-strong convexity of $\ell_i^*$ (follows from $1/\mu$-smoothness of $\ell_i$), the decrease in the dual objective is:

$$\begin{aligned}
\Delta_F &:= \mathcal{F}^*(\boldsymbol{\alpha}) - \min_{\Delta\dot{\boldsymbol{\alpha}}_j} \mathcal{F}^*(\boldsymbol{\alpha} + \gamma \sum_{j=1}^{L} \boldsymbol{e}_j \Delta\dot{\boldsymbol{\alpha}}_j) \\
&\geq \mathcal{F}^*(\boldsymbol{\alpha}) - \mathcal{F}^*(\boldsymbol{\alpha} + \gamma \sum_{j=1}^{L} \boldsymbol{e}_j \Delta\dot{\boldsymbol{\alpha}}_j) \;\; \forall \Delta\dot{\boldsymbol{\alpha}}_j \in \mathbb{R} \\
&\overset{[115,\text{ Lemma 3}]}{=} \mathcal{F}^*(\boldsymbol{\alpha}) - (1-\gamma)\mathcal{F}^*(\boldsymbol{\alpha}) - \gamma \sum_{i=1}^{L} \mathcal{G}_j^{\sigma'}(\Delta\dot{\boldsymbol{\alpha}}_j; \boldsymbol{v}, \alpha_j) \\
&= \gamma \left( \mathcal{F}^*(\boldsymbol{\alpha}) - \sum_{i=1}^{L} \mathcal{G}_j^{\sigma'}(\Delta\dot{\boldsymbol{\alpha}}_j; \boldsymbol{v}, \alpha_j) \right)
\end{aligned}$$

where $\mathcal{G}_j^{\sigma'}$ is the subproblem that each parallel loop is solving:

$$\mathcal{G}_j^{\sigma'} := \frac{1}{N}\ell_j^*(-\alpha_j - \Delta\dot{\boldsymbol{\alpha}}_j) + \frac{1}{L}\frac{1}{2\lambda N^2}\|\boldsymbol{v}\|^2 + \frac{\sigma'}{2\lambda N^2}\|\boldsymbol{x}_j \Delta\dot{\boldsymbol{\alpha}}_j\|^2 + \frac{1}{\lambda N^2}\boldsymbol{x}_j^\top \boldsymbol{v} \Delta\dot{\boldsymbol{\alpha}}_j$$

Therefore:

$$\frac{1}{\gamma}\Delta_F = \frac{1}{N}\sum_{j=1}^{L} \ell_j^*(-\alpha_j) + \frac{1}{2\lambda N^2}\|\boldsymbol{v}\|^2 - \frac{1}{N}\sum_{j=1}^{L} \ell_j^*(-(\alpha_j + \Delta\dot{\boldsymbol{\alpha}}_j)) - \frac{1}{2\lambda N^2}\|\boldsymbol{v}\|^2 - \frac{\sigma'}{2\lambda N^2}\sum_{j=1}^{L}\|\boldsymbol{x}_j \Delta\dot{\boldsymbol{\alpha}}_j\|^2 - \frac{1}{\lambda N^2}\sum_{j=1}^{L} \boldsymbol{x}_j^\top \boldsymbol{v} \Delta\dot{\boldsymbol{\alpha}}_j$$

Consider $\Delta\dot{\boldsymbol{\alpha}}_j := s(u_j - \alpha_j)$ for any $s \in (0, 1]$ and for $u_j = -\nabla_j \ell_j \left(\frac{1}{\lambda N}\boldsymbol{x}_j^\top \mathbb{E}[\boldsymbol{v}]\right)$. In some sense, $s$ denotes the deviation from the "optimal" $\dot{\Delta}$-value $(u_j - a_j)$. Hence we have:

$$\frac{N}{\gamma}\Delta_F \geq \sum_{j=1}^{L}\left(\ell_j^*(-\alpha_j) - \ell_j^*(-(\alpha_j + s(u_j - \alpha_j)))\right) - \frac{\sigma'}{2\lambda N}\sum_{j=1}^{L}\|\boldsymbol{x}_j s(u_j - a_j)\|^2 - \frac{1}{\lambda N}\sum_{j=1}^{L} \boldsymbol{x}_j^\top \boldsymbol{v} s(u_j - a_j)$$

By the $\mu$-strong convexity of $\ell_i^*$ (follows from $1/\mu$-smoothness of $\ell_i$) we have:

$$\ell_j^*(-(\alpha_j + s(u_j - \alpha_j))) = \ell_j^*(s(-u_j) + (1-s)(-a_j)) \leq s\ell_j^*(-u_j) + (1-s)\ell_j^*(-a_j) - \frac{\mu}{2}s(1-s)(u_j - a_j)^2$$

Hence we have:

$$\begin{aligned}
\frac{N}{\gamma}\Delta_F &\geq \sum_{j=1}^{L}\left(\ell_j^*(-\alpha_j) - s\ell_j^*(-u_j) - (1-s)\ell_j^*(-\alpha_j) + \frac{\mu}{2}s(1-s)(u_j-\alpha_j)^2\right) \\
&\quad - \sum_{j=1}^{L}\left(\frac{\sigma' s^2(u_j-\alpha_j)^2}{2\lambda N}\|\boldsymbol{x}_j\|^2 + \frac{1}{\lambda N}\boldsymbol{x}_j^\top \boldsymbol{v} s(u_j-\alpha_j)\right) \\
&= \sum_{j=1}^{L}\left(-s\ell_j^*(-u_j) + s\ell_j^*(-\alpha_j) + \frac{\mu}{2}s(1-s)(u_j-\alpha_j)^2 - \frac{\sigma' s^2(u_j-\alpha_j)^2}{2\lambda N}\|\boldsymbol{x}_j\|^2 - \frac{1}{\lambda N}\boldsymbol{x}_j^\top \boldsymbol{v} s(u_j-\alpha_j)\right)
\end{aligned}$$

By taking the expectation w.r.t. the randomization in the noise, using Jensen's inequality for convex functions ($\mathbb{E}[\ell_j^*(x)] \geq \ell_j^*(\mathbb{E}[x])$), the fact that $\sigma^2 = \mathbb{E}[\alpha_j^2] - \mathbb{E}[\alpha_j]^2$ and the fact that the noise on $\boldsymbol{\alpha}$ and $\boldsymbol{v}$ is independent we have:

$$\begin{aligned}
\frac{N}{\gamma}\mathbb{E}[\Delta_F] &\geq \sum_{j=1}^{L}\Big(-s\mathbb{E}[\ell_j^*(-u_j)] + s\mathbb{E}[\ell_j^*(-\alpha_j)] + \frac{\mu}{2}s(1-s)\mathbb{E}[(u_j-\alpha_j)^2] \\
&\quad - \frac{\sigma' s^2}{2\lambda N}\mathbb{E}[(u_j-\alpha_j)^2]\|\boldsymbol{x}_j\|^2 - \frac{s}{\lambda N}\boldsymbol{x}_j^\top\mathbb{E}[\boldsymbol{v}](u_j - \mathbb{E}[\alpha_j])\Big) \\
&\geq \sum_{j=1}^{L}\Big(-s\ell_j^*(-u_j) + s\mathbb{E}[\ell_j^*(-\alpha_j)] - \frac{s}{\lambda N}\boldsymbol{x}_j^\top\mathbb{E}[\boldsymbol{v}](u_j - \mathbb{E}[\alpha_j]) \\
&\quad + \left(\frac{\mu}{2}s(1-s) - \frac{\sigma' s^2}{2\lambda N}\|\boldsymbol{x}_j\|^2\right)(u_j^2 - 2u_j\mathbb{E}[\alpha_j] + \mathbb{E}[\alpha_j^2])\Big) \\
&\geq \sum_{j=1}^{L}\Big(-s\ell_j^*(-u_j) + s\mathbb{E}[\ell_j^*(-\alpha_j)] + \frac{s}{\lambda N}\boldsymbol{x}_j^\top\mathbb{E}[\boldsymbol{v}]\mathbb{E}[\alpha_j] \\
&\quad - \frac{s}{\lambda N}\boldsymbol{x}_j^\top\mathbb{E}[\boldsymbol{v}]u_j + \left(\frac{\mu}{2}s(1-s) - \frac{\sigma' s^2}{2\lambda N}\|\boldsymbol{x}_j\|^2\right)(u_j^2 - 2u_j\mathbb{E}[\alpha_j] + \mathbb{E}[\alpha_j]^2 + \sigma^2)\Big)
\end{aligned}$$

The following is the Fenchel-Young inequality that holds as equality given $u_j = -\nabla_j \ell_j\left(\frac{1}{\lambda N}\boldsymbol{x}_j^\top\mathbb{E}[\boldsymbol{v}]\right)$:

$$\ell_j\left(\frac{1}{\lambda N}\boldsymbol{x}_j^\top\mathbb{E}[\boldsymbol{v}]\right) + \ell_j^*(-u_j) = -\frac{1}{\lambda N}\boldsymbol{x}_j^\top\mathbb{E}[\boldsymbol{v}]u_j$$

Hence we have:

$$\begin{aligned}
\frac{N}{\gamma}\mathbb{E}[\Delta_F] \geq \sum_{j=1}^{L}\Big(&s\ell_j\left(\frac{1}{\lambda N}\boldsymbol{x}_j^\top\mathbb{E}[\boldsymbol{v}]\right) + s\mathbb{E}[\ell_j^*(-\alpha_j)] + \frac{s}{\lambda N}\boldsymbol{x}_j^\top\mathbb{E}[\boldsymbol{v}]\mathbb{E}[\alpha_j] \\
&+ \left(\frac{\mu}{2}s(1-s) - \frac{\sigma' s^2}{2\lambda N}\|\boldsymbol{x}_j\|^2\right)(u_j^2 - 2u_j\mathbb{E}[\alpha_j] + \mathbb{E}[\alpha_j]^2 + \sigma^2)\Big)
\end{aligned} \tag{B.4}$$

We then employ the definition of the duality gap ($Gap(\boldsymbol{\alpha}) := \mathcal{F}(\boldsymbol{\theta}(\boldsymbol{\alpha})) - (-\mathcal{F}^*(\boldsymbol{\alpha}))$) and take the expectation w.r.t. the randomization in the noise along with the Jensen inequality for convex functions.

$$\mathbb{E}[Gap(\boldsymbol{\alpha})] = \mathcal{F}(\boldsymbol{\theta}(\mathbb{E}[\boldsymbol{\alpha}])) - (-\mathbb{E}[\mathcal{F}^*(\boldsymbol{\alpha})]) = \frac{1}{N}\sum_{j=1}^{N}\left(\ell_j(\boldsymbol{x}_j^\top\boldsymbol{\theta}) + \mathbb{E}[\ell_j^*(-\alpha_j)]\right) + \frac{\lambda}{2}\|\boldsymbol{\theta}(\mathbb{E}[\boldsymbol{\alpha}])\|^2 + \frac{1}{2\lambda N^2}\|\boldsymbol{X}\mathbb{E}[\boldsymbol{\alpha}]\|^2$$

We then apply the Fenchel-Young inequality:

$$g(\boldsymbol{\theta}(\mathbb{E}[\boldsymbol{\alpha}])) + g^*(\boldsymbol{X}\mathbb{E}[(\boldsymbol{\alpha}]) \quad \geq \quad \boldsymbol{\theta}(\mathbb{E}[\boldsymbol{\alpha}])^\top\boldsymbol{X}\mathbb{E}[\boldsymbol{\alpha}] \Leftrightarrow \frac{\lambda}{2}\|\boldsymbol{\theta}(\mathbb{E}[\boldsymbol{\alpha}])\|^2 + \frac{1}{2\lambda N^2}\|\boldsymbol{X}\mathbb{E}[\boldsymbol{\alpha}]\|^2 \geq \frac{1}{\lambda N}\mathbb{E}[\boldsymbol{v}]^\top\boldsymbol{X}\mathbb{E}[\boldsymbol{\alpha}]$$

The above inequality holds as equality in light of Remark 3 and the primal-dual map. Therefore by using uniform mini-batch sampling the duality gap becomes:

$$\begin{aligned}
\mathbb{E}[Gap(\boldsymbol{\alpha})] &= \frac{1}{N}\sum_{j=1}^{N}\left(\ell_j(\boldsymbol{x}_j^\top\boldsymbol{\theta}) + \mathbb{E}[\ell_j^*(-\alpha_j)] + \frac{1}{\lambda N}\mathbb{E}[\alpha_j]\boldsymbol{x}_j^\top\mathbb{E}[\boldsymbol{v}]\right) \\
&= \frac{1}{N}\frac{N}{L}\sum_{j=1}^{L}\left(\ell_j(\boldsymbol{x}_j^\top\boldsymbol{\theta}) + \mathbb{E}[\ell_j^*(-\alpha_j)] + \frac{1}{\lambda N}\mathbb{E}[\alpha_j]\boldsymbol{x}_j^\top\mathbb{E}[\boldsymbol{v}]\right) \\
&= \frac{1}{L}\sum_{j=1}^{L}\left(\ell_j(\boldsymbol{x}_j^\top\boldsymbol{\theta}) + \mathbb{E}[\ell_j^*(-\alpha_j)] + \frac{1}{\lambda N}\mathbb{E}[\alpha_j]\boldsymbol{x}_j^\top\mathbb{E}[\boldsymbol{v}]\right)
\end{aligned}$$

By extracting these terms in Inequality B.4, the bound simplifies:

$$\frac{N}{\gamma}\mathbb{E}[\Delta_F] \quad \geq \quad sL\cdot\mathbb{E}[Gap(\boldsymbol{\alpha})] + \sum_{j=1}^{L}\left(\frac{\mu}{2}s(1-s) - \frac{\sigma' s^2}{2\lambda N}\|\boldsymbol{x}_j\|^2\right)(u_j^2 - 2u_j\mathbb{E}[\alpha_j] + \mathbb{E}[\alpha_j]^2 + \sigma^2)$$

Then by using $\|\boldsymbol{x}_j\|^2 = 1$ and $Gap(\boldsymbol{\alpha}) \geq \mathcal{S}(\boldsymbol{\alpha})$ we get:

$$\begin{aligned}
\mathbb{E}[\Delta_F] &= \mathbb{E}[\mathcal{S}(\boldsymbol{\alpha})] - \mathbb{E}[\mathcal{S}(\boldsymbol{\alpha} + \gamma\sum_{j=1}^{L}\Delta\dot{\boldsymbol{\alpha}}_j)] \\
&\geq \frac{sL\gamma}{N}\mathbb{E}[\mathcal{S}(\boldsymbol{\alpha})] + \frac{\sigma'\gamma s^2}{2\lambda N^2}\left(\frac{\mu\lambda(1-s)N}{s\sigma'} - 1\right)\sum_{j=1}^{L}\left((u_j - \mathbb{E}[\alpha_j])^2 + \sigma^2\right) \\
&\overset{\sigma'=L,\gamma=1}{=} \frac{sL}{N}\mathbb{E}[\mathcal{S}(\boldsymbol{\alpha})] + \frac{Ls^2}{2\lambda N^2}\left(\frac{\mu\lambda(1-s)N}{sL} - 1\right)\sum_{j=1}^{L}\left((u_j - \mathbb{E}[\alpha_j])^2 + \sigma^2\right) \\
&\overset{s=\frac{\mu\lambda N}{\mu\lambda N+L}}{=} \frac{\mu\lambda L}{\mu\lambda N + L}\mathbb{E}[\mathcal{S}(\boldsymbol{\alpha})]
\end{aligned}$$

Thus as long as $\mathbb{E}[\boldsymbol{\alpha}]$ is not equal $\boldsymbol{\alpha}^\star$ we can expect a decrease in the objective from computing an update $\Delta$ based on the noisy $\boldsymbol{\alpha}, \boldsymbol{v}$. □

# Bibliography


[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, pages 265–283, 2016.

[2] Martín Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *CCS*, pages 308–318, 2016.

[3] Naman Agarwal, Ananda Theertha Suresh, Felix Xinnan X Yu, Sanjiv Kumar, and Brendan McMahan. cpsgd: Communication-efficient and differentially-private distributed sgd. In *NIPS*, pages 7564–7575, 2018.

[4] Haider Al-Lawati and Stark C Draper. Gradient delay analysis in asynchronous distributed optimization. In *ICASSP*, pages 4207–4211. IEEE, 2020.

[5] Guillaume Alain, Alex Lamb, Chinnadhurai Sankar, Aaron Courville, and Yoshua Bengio. Variance reduction in sgd by distributed importance sampling. *arXiv preprint arXiv:1511.06481*, 2015.

[6] Dan Alistarh, Zeyuan Allen-Zhu, and Jerry Li. Byzantine stochastic gradient descent. In *Neural Information Processing Systems, to appear*, 2018.

[7] Dan Alistarh, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Randomized quantization for communication-optimal stochastic gradient descent. *arXiv preprint arXiv:1610.02132*, 2016.

[8] Majid Altamimi, Atef Abdrabou, Kshirasagar Naik, and Amiya Nayak. Energy cost models of smartphones for task offloading to the cloud. *TETC*, 3(3):384–398, 2015.

[9] How facebook and youtube help spread anti-vaxxer propaganda. https://www.theguardian.com/media/2019/feb/01/facebook-youtube-anti-vaccination-misinformation-social-media.

[10] https://developer.arm.com/ip-products/graphics-and-multimedia/mali-gpus, 2020.

[11] https://aws.amazon.com/device-farm/, 2020.

[12] Peter L Bartlett, Dylan J Foster, and Matus J Telgarsky. Spectrally-normalized margin bounds for neural networks. In *NIPS*, pages 6241–6250, 2017.

[13] Raef Bassily, Abhradeep Guha Thakurta, and Om Dipakbhai Thakkar. Model-agnostic private learning. In *NIPS*, pages 7102–7112, 2018.

[14] Amir Beck and Luba Tetruashvili. On the convergence of block coordinate descent type methods. *SIAM journal on Optimization*, 23(4):2037–2060, 2013.

[15] Richard Berk, Hoda Heidari, Shahin Jabbari, Matthew Joseph, Michael Kearns, Jamie Morgenstern, Seth Neel, and Aaron Roth. A convex framework for fair regression. *arXiv preprint arXiv:1706.02409*, 2017.

[16] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. signsgd: Compressed optimisation for non-convex problems. In *ICML*, pages 560–569, 2018.

[17] Bhattacharyya coefficient. https://en.wikipedia.org/wiki/Bhattacharyya_distance.

[18] Battista Biggio and Pavel Laskov. Poisoning attacks against support vector machines. In *ICML*, 2012.

[19] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *arXiv preprint arXiv:1712.03141*, 2017.

[20] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. Machine learning with adversaries: Byzantine tolerant gradient descent. In *NIPS*, pages 118–128, 2017.

[21] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konecny, Stefano Mazzocchi, H Brendan McMahan, et al. Towards federated learning at scale: System design. *Proceedings of the 2nd SysML Conference*, 2019.

[22] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *CCS*, pages 1175–1191. ACM, 2017.

[23] Léon Bottou. Online learning and stochastic approximations. *Online learning in neural networks*, 17(9):142, 1998.

[24] Olivier Bousquet and Léon Bottou. The tradeoffs of large scale learning. In *NIPS*, pages 161–168, 2008.

[25] Joseph K Bradley, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. Parallel coordinate descent for l1-regularized loss minimization. *arXiv preprint arXiv:1105.5379*, 2011.

[26] Theodora S Brisimi, Ruidi Chen, Theofanie Mela, Alex Olshevsky, Ioannis Ch Paschalidis, and Wei Shi. Federated learning of predictive models from federated electronic health records. *International journal of medical informatics*, 112:59–67, 2018.

[27] Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[28] Aaron Carroll, Gernot Heiser, et al. An analysis of power consumption in a smartphone. In *USENIX ATC*, volume 14, pages 21–21. Boston, MA, 2010.

[29] Miguel Castro, Barbara Liskov, et al. Practical Byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[30] California Consumer Privacy Act of 2018 (CCPA). https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180AB375.

[31] Kamalika Chaudhuri and Claire Monteleoni. Privacy-preserving logistic regression. In *NIPS*, pages 289–296, 2009.

[32] Kamalika Chaudhuri, Claire Monteleoni, and Anand D Sarwate. Differentially private empirical risk minimization. *JMLR*, 12(Mar):1069–1109, 2011.

[33] Ang Chen, Hanjun Xiao, Andreas Haeberlen, and Linh Thi Xuan Phan. Fault tolerance and the five-second rule. In *HotOS*, 2015.

[34] D. Chen, L. J. Xie, B. Kim, L. Wang, C. S. Hong, L. Wang, and Z. Han. Federated learning based mobile edge computing for augmented reality applications. In *ICNC*, pages 767–773, 2020.

[35] Fei Chen, Zhenhua Dong, Zhenguo Li, and Xiuqiang He. Federated meta-learning for recommendation. *arXiv preprint arXiv:1802.07876*, 2018.

[36] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

[37] Lingjiao Chen, Hongyi Wang, Zachary Charles, and Dimitris Papailiopoulos. Draco: Byzantine-resilient distributed training via redundant gradients. In *International Conference on Machine Learning*, pages 902–911, 2018.

[38] Wei-Lin Chiang, Mu-Chu Lee, and Chih-Jen Lin. Parallel dual coordinate descent method for large-scale linear classification in multi-core environments. In *KDD*, pages 1485–1494. ACM, 2016.

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

[40] Shaiful Alam Chowdhury, Luke N Kumar, Md Toukir Imam, Mohomed Shazan Mohomed Jabbar, Varun Sapra, Karan Aggarwal, Abram Hindle, and Russell Greiner. A system-call based model of software energy consumption without hardware instrumentation. In *IGSC*, pages 1–6, 2015.

[41] David Chu, Nicholas D Lane, Ted Tsung-Te Lai, Cong Pang, Xiangying Meng, Qing Guo, Fan Li, and Feng Zhao. Balancing energy, latency and accuracy for mobile sensor data classification. In *SenSys*, pages 54–67. ACM, 2011.

[42] Cifar dataset. https://www.cs.toronto.edu/~kriz/cifar.html.

[43] Moustapha Cisse, Piotr Bojanowski, Edouard Grave, Yann Dauphin, and Nicolas Usunier. Parseval networks: Improving robustness to adversarial examples. In *ICML*, pages 854–863, 2017.

[44] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. Emnist: an extension of mnist to handwritten letters. *arXiv preprint arXiv:1702.05373*, 2017.

[45] Katriel Cohn-Gordon, Georgios Damaskinos, Divido Neto, Joshi Cordova, Benoît Reitz, Benjamin Strahs, Daniel Obenshain, Paul Pearce, and Ioannis Papagiannis. Delf: Safeguarding deletion correctness in online social networks. In *USENIX Security*, 2020.

[46] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI*, pages 177–190. USENIX Association, 2006.

[47] Koby Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, and Yoram Singer. Online passive-aggressive algorithms. *JMLR*, 7(Mar):551–585, 2006.

[48] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *MobiSys*, pages 49–62. ACM, 2010.

[49] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R Ganger, Phillip B Gibbons, et al. Exploiting bounded staleness to speed up big data analytics. In *USENIX ATC*, pages 37–48, 2014.

[50] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *European Conference on Computer Systems*, page 4, 2016.

[51] Morten Dahl, Jason Mancuso, Yann Dupis, Ben Decoste, Morgan Giraud, Ian Livingstone, Justin Patriquin, and Gavin Uhma. Private machine learning in tensorflow using secure computation. *arXiv preprint arXiv:1810.08130*, 2018.

[52] Daily time spent on social networking by internet users worldwide from 2012 to 2019. https://www.statista.com/statistics/433871/daily-social-media-usage-worldwide/.

[53] Georgios Damaskinos, El Mahdi El Mhamdi, Rachid Guerraoui, Arsany Guirguis, and Sébastien Louis Alexandre Rouault. Aggregathor: Byzantine machine learning via robust gradient aggregation. In *Conference on Machine Learning and Systems (SysML / MLSys)*, 2019.

[54] Georgios Damaskinos, El Mahdi El Mhamdi, Rachid Guerraoui, Rhicheek Patra, Mahsa Taziki, et al. Asynchronous byzantine machine learning (the case of sgd). In *ICML*, pages 1153–1162, 2018.

[55] Georgios Damaskinos, Rachid Guerraoui, Anne-Marie Kermarrec, Vlad Nitu, Rhicheek Patra, and Francois Taiani. Fleet: Online federated learning via staleness awareness and performance prediction. *arXiv preprint arXiv:2006.07273*, 2020.

[56] Georgios Damaskinos, Rachid Guerraoui, Erwan Le Merrer, and Christoph Neumann. The imitation game: Algorithm selection by exploiting black-box recommenders. In *International Conference on Networked Systems*. Springer, 2020.

[57] Georgios Damaskinos, Rachid Guerraoui, and Rhicheek Patra. Capturing the moment: Lightweight similarity computations. In *ICDE*, pages 747–758, 2017.

[58] Georgios Damaskinos, Celestine Mendler-Dünner, Rachid Guerraoui, Nikolaos Papandreou, and Thomas Parnell. Differentially private stochastic coordinate descent. *arXiv preprint arXiv:2006.07272*, 2020.

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

[60] Bhuwan Dhingra, Zhong Zhou, Dylan Fitzpatrick, Michael Muehl, and William W Cohen. Tweet2vec: Character-based distributed representations for social media. *arXiv preprint arXiv:1605.03481*, 2016.

[61] Ilias Diakonikolas, Gautam Kamath, Daniel M. Kane, Jerry Li, Moitra. Ankur, and Stewart. Alistair. Robustly learning a gaussian: Getting optimal error, efficiently. *arXiv preprint arXiv:1704.03866*, 2017.

[62] Yi Ding, Nikita Mishra, and Henry Hoffmann. Generative and multi-phase learning for computer systems optimization. In *ISCA*, ISCA '19, page 39–52, New York, NY, USA, 2019. Association for Computing Machinery.

[63] Dl4j. https://deeplearning4j.org/.

[64] John C Duchi, Michael I Jordan, and Martin J Wainwright. Local privacy and statistical minimax rates. In *FOCS*, pages 429–438. IEEE, 2013.

[65] Celestine Dünner, Simone Forte, Martin Takáč, and Martin Jaggi. Primal-dual rates and certificates. In *ICML*, page 783–792. JMLR.org, 2016.

[66] Celestine Dünner, Thomas Parnell, Dimitrios Sarigiannis, Nikolas Ioannou, Andreea Anghel, Gummadi Ravi, Madhusudanan Kandasamy, and Haralampos Pozidis. Snap ml: A hierarchical framework for machine learning. In *NIPS*, pages 250–260, 2018.

[67] Sanghamitra Dutta, Viveck Cadambe, and Pulkit Grover. Short-dot: Computing large linear transforms distributedly using coded short dot products. In *NIPS*, pages 2100–2108, 2016.

[68] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4):211–407, 2014.

[69] E. M. El Mhamdi and R. Guerraoui. When neurons fail. In *IPDPS*, pages 1028–1037, May 2017.

[70] E. M. El Mhamdi, R. Guerraoui, and S. Rouault. On the robustness of a neural network. In *SRDS*, pages 84–93, Sept 2017.

[71] El-Mahdi El-Mhamdi and Rachid Guerraoui. Fast and secure distributed learning in high dimension. *arXiv preprint arXiv:*, 2019.

[72] El Mahdi El Mhamdi, Rachid Guerraoui, and Sébastien Rouault. The hidden vulnerability of distributed learning in Byzantium. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 3521–3530, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[73] El Mahdi El Mhamdi, Rachid Guerraoui, and Sébastien Rouault. The hidden vulnerability of distributed learning in byzantium. *arXiv preprint arXiv:1802.07927*, 2018.

[74] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *JMLR*, 9(Aug):1871–1874, 2008.

[75] Matthias Feurer and Frank Hutter. Hyperparameter optimization. In *Automated Machine Learning*, pages 3–33. Springer, 2019.

[76] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2):374–382, 1985.

[77] Regulation 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 (GDPR). https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679.

[78] Yuyun Gong and Qi Zhang. Hashtag recommendation using attention-based convolutional neural network. In *IJCAI*, pages 2782–2788, 2016.

[79] Google. Tensorflow text classification, 2020.

[80] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[81] P Greenhalgh. big. little technology: The future of mobile. *ARM, White paper*, 2013.

[82] Grid5000. https://www.grid5000.fr/.

[83] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *FAST*, pages 1–14, 2018.

[84] Matthew Halpern, Yuhao Zhu, and Vijay Janapa Reddi. Mobile cpu's rise to power: Quantifying the impact of generational mobile cpu design trends on performance, energy, and user satisfaction. In *HPCA*, pages 64–76. IEEE, 2016.

[85] Shuai Hao, Ding Li, William GJ Halfond, and Ramesh Govindan. Estimating mobile application energy consumption using program analysis. In *ICSE*, pages 92–101. IEEE Press, 2013.

[86] Andrew Hard, Kanishka Rao, Rajiv Mathews, Swaroop Ramaswamy, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. Federated learning for mobile keyboard prediction. *arXiv preprint arXiv:1811.03604*, 2018.

[87] Simon S Haykin. *Neural networks and learning machines*, volume 3. Pearson Upper Saddle River, NJ, USA:, 2009.

[88] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, pages 1223–1231, 2013.

[89] https://www.4g.co.uk/how-fast-is-4g/, 2020.

[90] Cho-Jui Hsieh, Hsiang-Fu Yu, and Inderjit S Dhillon. Passcode: Parallel asynchronous stochastic dual co-ordinate descent. In *ICML*, volume 15, pages 2370–2379, 2015.

[91] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R Ganger, Phillip B Gibbons, and Onur Mutlu. Gaia: Geo-distributed machine learning approaching {LAN} speeds. In *NSDI*, pages 629–647, 2017.

[92] Eunjeong Jeong, Seungeun Oh, Hyesung Kim, Jihong Park, Mehdi Bennis, and Seong-Lyun Kim. Communication-efficient on-device machine learning: Federated distillation and augmentation under non-iid private data. *arXiv preprint arXiv:1811.11479*, 2018.

[93] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. Heterogeneity-aware distributed parameter servers. In *SIGMOD*, pages 463–478, 2017.

[94] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *ASPLOS*, pages 615–629, 2017.

[95] Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, Saravanan Thirumuruganathan, Sanjay Chawla, and Divy Agrawal. A cost-based optimizer for gradient descent optimization. In *SIGMOD*, pages 977–992, 2017.

[96] Jakub Konečnỳ, H Brendan McMahan, Daniel Ramage, and Peter Richtárik. Federated optimization: distributed machine learning for on-device intelligence. *arXiv preprint arXiv:1610.02527*, 2016.

[97] Dominik Kowald, Subhash Chandra Pujari, and Elisabeth Lex. Temporal effects on hashtag reuse in twitter: A cognitive-inspired hashtag recommendation approach. In *WWW*, pages 1401–1410, 2017.

[98] Kryo. https://github.com/EsotericSoftware/kryo/.

[99] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, et al. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *SIGCOMM*, volume 45, pages 1–14. ACM, 2015.

[100] Yongin Kwon, Sangmin Lee, Hayoon Yi, Donghyun Kwon, Seungjun Yang, Byung-Gon Chun, Ling Huang, Petros Maniatis, Mayur Naik, and Yunheung Paek. Mantis: Automatic performance prediction for smartphone applications. In *USENIX ATC*, pages 297–308, 2013.

[101] Su Mon Kywe, Tuan-Anh Hoang, Ee-Peng Lim, and Feida Zhu. On recommending hashtags in twitter networks. In *International Conference on Social Informatics*, pages 337–350. Springer, 2012.

[102] Anusha Lalitha, Shubhanshu Shekhar, Tara Javidi, and Farinaz Koushanfar. Fully decentralized federated learning. In *Third workshop on Bayesian Deep Learning (NIPS)*, 2018.

[103] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *TOPLAS*, 4(3):382–401, 1982.

[104] Kangwook Lee, Maximilian Lam, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran. Speeding up distributed machine learning using codes. *IEEE Transactions on Information Theory*, 64(3):1514–1529, 2017.

[105] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, 2014.

[106] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In *NIPS*, pages 19–27, 2014.

[107] Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, volume 6, page 2, 2013.

[108] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. Federated optimization for heterogeneous networks. *arXiv preprint arXiv:1812.06127*, 2018.

[109] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *NIPS*, pages 2737–2745, 2015.

[110] Xiangru Lian, Huan Zhang, Cho-Jui Hsieh, Yijun Huang, and Ji Liu. A comprehensive linear speedup analysis for asynchronous stochastic parallel optimization from zeroth-order to first-order. In *NIPS*, 2016.

[111] Ji Liu, Stephen, and J. Wright. Asynchronous stochastic coordinate descent: Parallelism and convergence properties. Technical report, SIAM Journal on Optimization, 2015.

[112] Yan Liu and Jack YB Lee. An empirical study of throughput prediction in mobile data networks. In *GLOBECOM*, pages 1–6. IEEE, 2015.

[113] Yi Liu, James JQ Yu, Jiawen Kang, Dusit Niyato, and Shuyu Zhang. Privacy-preserving traffic flow prediction: A federated learning approach. *arXiv preprint arXiv:2003.08725*, 2020.

[114] Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.

[115] Chenxin Ma, Virginia Smith, Martin Jaggi, Michael I Jordan, Peter Richtárik, and Martin Takáč. Adding vs. averaging in distributed primal-dual optimization. In *ICML*, pages 1973–1982, 2015.

[116] Stéphane Mallat. Understanding deep convolutional networks. *Phil. Trans. R. Soc. A*, 374(2065):20150203, 2016.

[117] Matrix multiplication benchmark. https://browser.geekbench.com.

[118] Peter McCullagh. *Generalized linear models*. Routledge, 2018.

[119] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *AISTATS*, pages 1273–1282, 2017.

[120] Average text message length. https://crushhapp.com/blog/k-wrap-it-up-mom.

[121] Average text messages per day. https://www.textrequest.com/blog/how-many-texts-people-send-per-day/.

[122] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[123] Gilad Mishne, Jeff Dalton, Zhenghua Li, Aneesh Sharma, and Jimmy Lin. Fast data in the era of big data: Twitter's real-time related query suggestion architecture. In *SIGMOD*, pages 1147–1158. ACM, 2013.

[124] Nikita Mishra, Connor Imes, John D Lafferty, and Henry Hoffmann. Caloree: Learning control for predictable latency and low energy. *ASPLOS*, 53(2):184–198, 2018.

[125] Nikita Mishra, Huazhe Zhang, John D Lafferty, and Henry Hoffmann. A probabilistic graphical model-based approach for minimizing energy under performance constraints. *ASPLOS*, 50(4):267–281, 2015.

[126] Ioannis Mitliagkas, Ce Zhang, Stefan Hadjis, and Christopher Ré. Asynchrony begets momentum, with an application to deep learning. In *Annual Allerton Conference on Communication, Control, and Computing*, pages 997–1004. IEEE, 2016.

[127] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering developers to estimate app energy consumption. In *MobiCom*, pages 317–328. ACM, 2012.

[128] Mnist dataset. http://yann.lecun.com/exdb/mnist/.

[129] Yu Nesterov. Efficiency of coordinate descent methods on huge-scale optimization problems. *SIAM Journal on Optimization*, 22(2):341–362, 2012.

[130] Behnam Neyshabur, Ruslan R Salakhutdinov, and Nati Srebro. Path-sgd: Path-normalized optimization in deep neural networks. In *NIPS*, pages 2422–2430, 2015.

[131] Thông T Nguyên and Siu Cheung Hui. Differentially private regression for discrete-time survival analysis. In *CIKM*, pages 1199–1208, 2017.

[132] Takayuki Nishio and Ryo Yonetani. Client selection for federated learning with heterogeneous resources in mobile edge. In *ICC*, pages 1–7. IEEE, 2019.

[133] NSA's PRISM. https://www.theverge.com/2013/7/17/4517480/nsa-spying-prism-surveillance-cheat-sheet.

[134] NVIDIA. CUDA GPUs | NVIDIA Developer. https://developer.nvidia.com/cuda-gpus, 2020.

[135] Eriko Otsuka, Scott A Wallace, and David Chiu. Design and evaluation of a twitter hashtag recommendation system. In *IDEAS*, pages 330–333, 2014.

[136] Xue Ouyang, Peter Garraghan, David McKee, Paul Townend, and Jie Xu. Straggler detection in parallel computing systems through dynamic threshold calculation. In *AINA*, pages 414–421. IEEE, 2016.

[137] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Asia Conference on Computer and Communications Security*, pages 506–519, 2017.

[138] Nicolas Papernot, Shuang Song, Ilya Mironov, Ananth Raghunathan, Kunal Talwar, and Ulfar Erlingsson. Scalable private learning with PATE. In *ICLR*, 2018.

[139] Thomas Parnell, Celestine Dünner, Kubilay Atasu, Manolis Sifalakis, and Haris Pozidis. Large-scale stochastic learning using gpus. In *IPDPSW*, pages 419–428. IEEE, 2017.

[140] Rhicheek Patra. *Towards Scalable Personalization*. PhD thesis, EPFL, 2018.

[141] Percentage of all global web pages served to mobile phones from 2009 to 2018. https://www.statista.com/statistics/241462/global-mobile-phone-website-traffic-share/.

[142] Tien-Dat Phan, Guillaume Pallez, Shadi Ibrahim, and Padma Raghavan. A new framework for evaluating straggler detection mechanisms in mapreduce. *TOMPECS*, 4(3):1–23, 2019.

[143] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. In *ICLR*, 2018.

[144] Feng Qian, Zhaoguang Wang, Alexandre Gerber, Zhuoqing Mao, Subhabrata Sen, and Oliver Spatscheck. Profiling resource usage for mobile applications: a cross-layer approach. In *MobiSys*, pages 321–334. ACM, 2011.

[145] Aurick Qiao, Abutalib Aghayev, Weiren Yu, Haoyang Chen, Qirong Ho, Garth A Gibson, and Eric P Xing. Litz: Elastic framework for high-performance distributed machine learning. In *USENIX ATC*, pages 631–644, 2018.

[146] Qualcomm. Adreno™ Graphics Processing Units. https://developer.qualcomm.com/software/adreno-gpu-sdk/gpu, 2020.

[147] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.

[148] Peter Richtárik and Martin Takáč. Parallel coordinate descent methods for big data optimization. *Mathematical Programming*, 156(1-2):433–484, 2016.

[149] Theo Ryffel, Andrew Trask, Morten Dahl, Bobby Wagner, Jason Mancuso, Daniel Rueckert, and Jonathan Passerat-Palmbach. A generic framework for privacy preserving deep learning. *arXiv preprint arXiv:1811.04017*, 2018.

[150] S20.ai. https://s20.ai/home.

[151] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *WWW*, pages 285–295, 2001.

[152] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *CSUR*, 22(4):299–319, 1990.

[153] Siddhartha Sen, Wyatt Lloyd, and Michael J Freedman. Prophecy: Using history for high-throughput fault tolerance. In *NSDI*, pages 345–360, 2010.

[154] Shai Shalev-Shwartz and Tong Zhang. Accelerated mini-batch stochastic dual coordinate ascent. In *NIPS*, pages 378–385, 2013.

[155] Shai Shalev-Shwartz and Tong Zhang. Stochastic dual coordinate ascent methods for regularized loss minimization. *JMLR*, 14(Feb):567–599, 2013.

[156] SIMD ISAs | Neon. https://developer.arm.com/architectures/instruction-sets/simd-isas/neon.

[157] Smartphone users worldwide 2014-2020. https://statinvestor.com/data/33977/number-of-smartphone-users-worldwide/.

[158] Virginia Smith, Chao-Kai Chiang, Maziar Sanjabi, and Ameet S Talwalkar. Federated multi-task learning. In *NIPS*, pages 4424–4434, 2017.

[159] Virginia Smith, Simone Forte, Chenxin Ma, Martin Takáč, Michael I Jordan, and Martin Jaggi. Cocoa: A general framework for communication-efficient distributed optimization. *JMLR*, 18(1):8590–8638, 2017.

[160] Snips - using voice to make technology disappear. https://snips.ai/.

[161] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *JMLR*, 15(1):1929–1958, 2014.

[162] Lili Su. *Defending distributed systems against adversarial attacks: consensus, consensus-based learning, and statistical learning*. PhD thesis, University of Illinois at Urbana-Champaign, 2017.

[163] Kunal Talwar, Abhradeep Guha Thakurta, and Li Zhang. Nearly optimal private lasso. In *NIPS*, pages 3025–3033, 2015.

[164] Tensorflow - federated learning. https://www.tensorflow.org/federated/federated_learning.

[165] Om Thakkar, Galen Andrew, and H Brendan McMahan. Differentially private learning with adaptive clipping. *arXiv preprint arXiv:1905.03871*, 2019.

[166] The Facebook and Cambridge Analytica scandal. https://www.vox.com/policy-and-politics/2018/3/23/17151916/facebook-cambridge-analytica-trump-diagram.

[167] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5–rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[168] Aleksei Triastcyn and Boi Faltings. Federated generative privacy. In *IJCAI Workshop on Federated Machine Learning for User Privacy and Data Confidentiality (FML 2019)*, number CONF, 2019.

[169] Tweepy. https://tweepy.readthedocs.io/en/latest/.

[170] Paul Vanhaesebrouck, Aurélien Bellet, and Marc Tommasi. Decentralized collaborative learning of personalized models over networks. In *AISTATS*, 2017.

[171] Di Wang, Minwei Ye, and Jinhui Xu. Differentially private empirical risk minimization revisited: Faster and more general. In *NIPS*, pages 2722–2731, 2017.

[172] Shiqiang Wang, Tiffany Tuor, Theodoros Salonidis, Kin K Leung, Christian Makaya, Ting He, and Kevin Chan. Adaptive federated learning in resource constrained edge computing systems. *IEEE Journal on Selected Areas in Communications*, 37(6):1205–1221, 2019.

[173] Zhibo Wang, Mengkai Song, Zhifei Zhang, Yang Song, Qian Wang, and Hairong Qi. Beyond inferring class representatives: User-level privacy leakage from federated learning. In *INFOCOM*, pages 2512–2520. IEEE, 2019.

[174] Stephen J Wright. Coordinate descent algorithms. *Mathematical Programming*, 151(1):3–34, 2015.

[175] Xi Wu, Fengan Li, Arun Kumar, Kamalika Chaudhuri, Somesh Jha, and Jeffrey Naughton. Bolt-on differential privacy for scalable stochastic gradient descent-based analytics. In *SIGMOD*, pages 1307–1322, 2017.

[176] Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. Generalized Byzantine-tolerant sgd. *arXiv preprint arXiv:1802.10116*, 2018.

[177] Eric P Xing, Qirong Ho, Wei Dai, Jin-Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. In *KDD*, pages 1335–1344, 2015.

[178] Timothy Yang, Galen Andrew, Hubert Eichner, Haicheng Sun, Wei Li, Nicholas Kong, Daniel Ramage, and Françoise Beaufays. Applied federated learning: Improving google keyboard query suggestions. *arXiv preprint arXiv:1812.02903*, 2018.

[179] Dong Yin, Yudong Chen, Kannan Ramchandran, and Peter Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates. *arXiv preprint arXiv:1803.01498*, 2018.

[180] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. In *USENIX ATC*, volume 12, pages 1–14, 2012.

[181] Mikhail Yurochkin, Mayank Agarwal, Soumya Ghosh, Kristjan Greenewald, Nghia Hoang, and Yasaman Khazaeni. Bayesian nonparametric federated learning of neural networks. In *ICML*, pages 7252–7261, 2019.

[182] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *USENIX ATC*, pages 181–193, 2017.

[183] Jiaqi Zhang, Kai Zheng, Wenlong Mou, and Liwei Wang. Efficient private erm for smooth objectives. *IJCAI*, 2017.

[184] Ruiliang Zhang, Shuai Zheng, and James T Kwok. Asynchronous distributed semi-stochastic gradient optimization. In *AAAI*, pages 2323–2329, 2016.

[185] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. Staleness-aware async-sgd for distributed deep learning. In *IJCAI*, pages 2350–2356, 2016.

[186] Yue Zhao, Meng Li, Liangzhen Lai, Naveen Suda, Damon Civin, and Vikas Chandra. Federated learning with non-iid data. *arXiv preprint arXiv:1806.00582*, 2018.

[187] Yong Zhuang, Yuchin Juan, Guo-Xun Yuan, and Chih-Jen Lin. Naive parallelization of coordinate descent methods and an application on multi-core l1-regularized classification. In *CIKM*, pages 1103–1112, 2018.

[188] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *NIPS*, pages 2595–2603, 2010.

# Georgios Damaskinos

EPFL IC LPD, INR 313 (Bâtiment INR), Station 14, CH-1015 Lausanne
*Webpage:* gdamaskinos.com
*E-mail:* georgios.damaskinos@epfl.ch

**Research Interests** Machine learning, Mobile computing, Distributed systems, Recommender systems

**Education**

Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland
**Ph.D. Candidate, Computer Science** 09/2015 - Present
Distributed Computing Lab - Rachid Guerraoui

National Technical University of Athens (NTUA), Athens, Greece
**Diploma in Electrical and Computer Engineering** 10/2010 - 07/2015
(5-year academic program, equivalent to M.Sc. degree)
*GPA*: 9.46 / 10 (top 2%)
*Major*: Computer Systems, Computer Software, Computer Networks
*Thesis*: "Profiling and cost modeling of join algorithms for Big Data Analytics"
Built a machine learning tool that automatically chooses the optimal engine (Spark, Hive, PostgreSQL) and configuration for the deployment of various join algorithms.
Computing Systems Laboratory - Nectarios Koziris

**Publications**

[1] "DELF: Safeguarding deletion correctness in Online Social Networks" 08/2020
K. Cohn-Gordon, G. Damaskinos, D. Neto, J. Cordova, B. Reitz, B. Strahs, D. Obenshain, P. Pearce, I. Papagiannis
**USENIX Security Symposium 2020** (to appear)

[2] "FLeet: Online Federated Learning via Staleness Awareness and Performance Prediction" 06/2020
G. Damaskinos, R. Guerraoui, A.M. Kermarrec, V. Nitu, R. Patra, F. Taiani
*Links*: paper (under submission), code

[3] "Differentially Private Stochastic Coordinate Descent" 06/2020
G. Damaskinos, C. Duenner, R. Guerraoui, N. Papandreou, T. Parnell
*Links*: paper (under submission), code

[4] "The Imitation Game: Algorithm Selection by Exploiting Black-Box Recommenders" 06/2020
G. Damaskinos, R. Guerraoui, E. Merrer, C. Neumann
**NETYS 2020** (International Conference on Networked Systems)
*Links*: paper, code, video

[5] "AggregaThor: Byzantine Machine Learning via Robust Gradient Aggregation" 05/2019
G. Damaskinos, E.M. El Mhamdi, R. Guerraoui, A. Guirguis, S. Rouault
**MLSys 2019** (Conference on Machine Learning and Systems)
*Links*: paper, code, video

[6] "Asynchronous Byzantine Machine Learning (the case of SGD)" 07/2018
G. Damaskinos, E.M. El Mhamdi, R. Guerraoui, R. Patra, M. Taziki
**ICML 2018** (International Conference on Machine Learning)
*Links*: paper, code

[7] "Capturing the Moment: Lightweight Similarity Computations" 05/2017
G. Damaskinos, R. Guerraoui, R. Patra
**ICDE 2017** (IEEE International Conference on Data Engineering)
*Links*: paper, code

## Experience

Facebook, London, United Kingdom
**Software Engineering Intern** 06/2019 - 08/2019
Worked on the DELF paper [1] by mainly focusing on the performance evaluation. Implemented optimizations as a result of the visibility that this evaluation provided.
*Language*: Hack, SQL, Python

IBM, Zurich, Switzerland
**Research Intern** 06/2018 - 09/2018
Formally derived and implemented a differentially private optimization algorithm for generalized linear models, as part of the core algorithmic component of SnapML. The work continued at EPFL and led to [3].
*Language*: Python

Technicolor, Rennes, France
**Research Intern** 06/2017 - 08/2017
Designed and implemented a method to boost the evaluation of a new recommendation service by exploiting the output of a supposedly well-established online recommender, acting as a black-box that the new service can query. Patent filed with number PCT/EP19/052345. The work continued at EPFL and led to [4].
*Language*: Python

Athens Clue, Athens, Greece
**Web Developer** 12/2014
Designed and implemented a web application for an escape room.
*Language*: JavaScript, HTML, CSS

High School Students, Athens, Greece
**Tutor** 03/2012 - 06/2015
Provided complementary mathematics, physics, and programming lectures to 8 students who either needed additional help or wanted to go beyond their school curriculum and perfect their skills.

## Awards

**Best Teaching Assistant Award**, EPFL 12/2019
**Runner-up for Best Internship Award**, Technicolor 11/2017
**EDIC PhD Fellowship**, EPFL 04/2015
**Thomaideio Award** for the top graduating students, NTUA 2015
**Papakyriakopoulos Award** for excellence in Mathematics, NTUA 2011

## Professional Service

*Reviewer*: MLSys 2019 (external), ICML 2019, ICML 2020, NIPS 2020
*Project mentor during PhD*: **1** PhD semester, **1** MSc thesis, **5** MSc semester, **1** BSc semester, **1** intern
*EPFL teaching assistant*: Distributed algorithms CS-451 (2018, 2019),
Real-time systems CS-321 (2018, 2019), Analysis 1 MATH-101 (2016, 2017)

## Main Technical Skills

*Programming*: Python, Java, C, C++
*Frameworks/Tools*: TensorFlow, Apache Spark, Android NDK, PyTorch

## Languages

Greek: Native proficiency
English: Full professional proficiency (C2 - Michigan University ECPE)
German: Limited working proficiency (B1 - Goethe Zertifikat)
French: Elementary proficiency