Thèse n° 8995

System Support for Robust Distributed Learning

Présentée le 12 juillet 2022

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

pour l'obtention du grade de Docteur ès Sciences

par

Arsany Hany Abdelmessih GUIRGUIS

Acceptée sur proposition du jury

Prof. A. Ailamaki, présidente du jury
Prof. R. Guerraoui, directeur de thèse
Prof. F. Taiani, rapporteur
Prof. M. Canini, rapporteur
Prof. K. Argyraki, rapporteuse

École
polytechnique
fédérale
de Lausanne

2022

Do not forsake wisdom, and she will protect you;
love her, and she will watch over you
— The Bible

فَإِنْ كَانَ أَحَدٌ يَظُنُّ أَنَّهُ يَعْرِفُ شَيْئًا، فَإِنَّهُ لَمْ يَعْرِفْ شَيْئًا بَعْدُ كَمَا يَجِبُ أَنْ يَعْرِف

To two pure souls I lost during my PhD: Teta Demiana and Uncle Bassem …

Acknowledgements

PhD is hard. It is impossible to get through it alone, especially during a worldwide pandemic. I was lucky to be surrounded by amazing people who helped me on both (equally important) technical and non-technical sides. To reflect the contributions of those who helped me on sthe technical side, I use the pronoun *"we"* instead of *"I"* throughout this thesis.

Having a good advisor is a cornerstone to having an enjoyable PhD journey; mine was unforgettable, primarily because of Rachid. Rachid is not only a great researcher and advisor, but he is also an extremely nice person. Thank you for always being encouraging, patient, and friendly. Thank you also for giving me the autonomy to explore research areas I like and the proper guidance, without which I could not have done top-quality research.

Luckily, I had a unique chance to work with brilliant researchers and mentors from different institutes. These collaborations have led to research contributions, all included in this thesis (BINGO!). I am grateful to Anne-Marie Kermarrec and Erwan Le-Merrer for hosting me in their group at Inria. Thank you for allowing me to work on compelling problems in the beautiful city of Rennes. I would also like to thank Florin Dinu, Javier Picorel, and Do Le Quoc for my internship at Huawei Research Labs in Munich and the subsequent collaboration. Thank you for believing in me and pushing me to produce better research. I also want to thank CJ Zheng, Yitzchak Lockerman, David Eis, and Ari Silburt for my internship at Bloomberg AI. There, I had a special opportunity to see how to do research in industry settings.

I am grateful to my PhD oral exam jury: Prof. François Taïani, Prof. Marco Canini, and Prof. Katerina Argyraki. Thank you for your insightful questions and comments, which have helped me improve my thesis. I am also thankful to Prof. Anastasiia Ailamaki for presiding over my thesis committee. I had a distinctive chance to present my thesis to Prof. Dahlia Malkhi, whom I thank for the insightful discussion about my thesis and its scope. Such a discussion helped me prepare well for the oral exam and gain extra confidence in my work.

I could not have asked for a nicer lab than the distributed computing lab (DCL). First, a special shoutout to George D. for mentoring me during my first semester in the lab. George had also reviewed all my paper submissions (until his graduation), for which I am very grateful. I would also like to thank Mahdi for helping me write my first theory paper in the area of Byzantine ML. Many thanks to Pierre-Louis, Vasilis, Florin, Nirupam, Rafaël, Andrei, Thanasis,

Jovan, Gauthier, Sadegh (my awesome office-mate), and Antoine for their valuable comments on my thesis writing and presentation. I wish to thank Lê and Pierre-Louis for helping me write a proper french translation of the abstract. Special thanks to Pierre-Louis for helping me improve my thesis with passion as if it was his; I was impressed by that. Additionally, I would like to thank all the other members with whom I shared lots of cool chats, discussions, memories, and outings: the senior guys: George C., Matej, Adi (the lifting buddy), Igor, and Karolos; the guys of my batch: Sébastien and Matteo; and those who joined afterward: John, Anastasiia, Youssef, and Diana, the youngest member of the lab, whom I trust to continue the work on distributed ML systems. I am grateful to the admin staff, who made my PhD life easier: Fabien, who can solve any technical problem or question you could face, and France, for being always friendly and helpful with administrative (and not only) challenges.

During my PhD, I mentored a few students (as interns or while doing semester projects or Master thesis) whom I thank for their trust and great work: Jérémy, Anton, Sahar, and Adrian. I also wish to thank the Center for Digital Trust (C4DT) at EPFL for believing in my Garfield project and for spending time and effort to showcase it and create a demo on its functionality.

My work has received generous financial support from EPFL, the Swiss National Science Foundation, and Huawei via the EDIC PhD fellowship, the FNS Grant 200021_182542, and the Huawei grant N° 2021-0053, respectively. Bloomberg has also financially supported the work in Chapter 4, providing the computing resources for the experiments. Some of the experiments presented in this thesis use the Grid’5000 testbed, which is supported by a scientific interest group hosted by Inria and including CNRS, RENATER, and several universities as well as other organizations (see https://www.grid5000.fr). Thank you all for the support.

I want to thank those who made the transition from Alexandria to Lausanne smooth and made Lausanne feels like home. I am grateful to Abouna Mina for his help and support on many fronts throughout all these past years. I thank all the members of the Coptic Orthodox Church (and they are a lot!) in Geneva and Lausanne. Special thanks to Cyril Mikhail for the invaluable advice, discussions, and help from the first day of my PhD Big thanks to Emad Sabet and Giovanni Ekram for the great times we had together and for being great friends.

Family is everything. I would like to express my gratitude to my big family, scattered all over the world. Most importantly, I cannot stress enough how deeply I am grateful to my parents, Abouna Luka and Madlen Naguib, and my brother Abouna Bishoy for their endless love and support. I owe you a lot, and I wish to make you always proud of me. Last but not least, I am profoundly grateful to my life partner, Sandra, without whom I could not imagine how I would sail the PhD journey, especially during the hard times of COVID. Thank you for constantly believing in me and encouraging me no matter what. To more amazing moments together!

Lausanne, June 24, 2022 Arsany Guirguis.

Preface

The research presented in this dissertation was conducted in the Distributed Computing Laboratory at EPFL, under the supervision of Professor Rachid Guerraoui, between 2018 and 2022. The main results of this dissertation appeared originally in the following publications:

1. Rachid Guerraoui, Arsany Guirguis, Jérémy Plassmann, Anton Ragot, and Sébastien Rouault. Garfield: System support for Byzantine machine learning (regular paper). In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 39-51. IEEE, 2021.
2. Arsany Guirguis, Yitzchak Lockerman, Chengjian Zheng, David Eis, Rachid Guerraoui. Elastic resource allocation for multi-tenant deep learning clusters. *Under submission.*
3. Arsany Guirguis, Florin Dinu, Diana Petrescu, Do Le Quoc, Javier Picorel, Rachid Guerraoui. Accelerating transfer learning with cloud object stores. *Under submission.*
4. Rachid Guerraoui, Arsany Guirguis, Anne-Marie Kermarrec, and Erwan Le Merrer. FeGAN: Scaling distributed GANs. In *Proceedings of the 21st International Middleware Conference*, pp. 193-206. 2020.

Besides the above-mentioned publications, I have also contributed to other research projects that resulted in the following publications (author names are in alphabetical order):

1. Georgios Damaskinos, El-Mahdi El-Mhamdi, Rachid Guerraoui, Arsany Guirguis, and Sébastien Rouault. Aggregathor: Byzantine machine learning via robust gradient aggregation. In *Proceedings of Machine Learning and Systems 1*, pp. 81-106. 2019
2. El-Mahdi El-Mhamdi, Rachid Guerraoui, Arsany Guirguis, Lê Nguyên Hoang, and Sébastien Rouault. Genuinely distributed Byzantine machine learning. *In Proceedings of the 39th Symposium on Principles of Distributed Computing*, pp. 355-364. 2020.
3. El-Mahdi El-Mhamdi, Sadegh Farhadkhani, Rachid Guerraoui, Arsany Guirguis, Lê-Nguyên Hoang, and Sébastien Rouault. Collaborative learning in the jungle (decentralized, Byzantine, heterogeneous, asynchronous and nonconvex learning). *Advances in Neural Information Processing Systems 34.* 2021.
4. El-Mahdi El-Mhamdi, Rachid Guerraoui, Arsany Guirguis, Lê Nguyên Hoang, and Sébastien Rouault. Genuinely distributed Byzantine machine learning (extended version). Distributed Computing, pp. 1-27. 2022.

June 24, 2022 Arsany Guirguis.

Abstract

Machine learning (ML) applications are ubiquitous. They run in different environments such as datacenters, the cloud, and even on edge devices. Despite where they run, distributing ML training seems the only way to attain scalable, high-quality learning. But, distributing ML is challenging, essentially due to the unique nature of ML applications.

First, ML training needs to be *robust* against arbitrary (i.e., *Byzantine*) failures due to its usage in mission-critical applications. Second, training applications in datacenters run on *shared* clusters of computing resources, for which we need *resource allocation* solutions that meet the high computation demands of these applications while fully utilizing existing resources. Third, running distributed training in *the cloud faces a network bottleneck*, exacerbated by the fast-growing pace of computing power. Hence, we need solutions that reduce the communication load without impacting the training accuracy. Fourth, despite the scalability and privacy guarantees of training on edge devices via *federated learning*, the heterogeneity of devices' capabilities and their data distributions calls for robust solutions that cope with these challenges.

To achieve *robustness*, we introduce Garfield, a library to help practitioners make their ML applications Byzantine-resilient. Besides addressing the vulnerability of the shared-graph architecture followed by classical ML frameworks, Garfield supports various communication patterns, robust aggregation rules, and compute hardware (i.e., CPUs and GPUs). We show how to use Garfield in different architectures, network settings, and data distributions.

We explore *elastic* training (i.e., changing the training parameters mid-execution) to efficiently solve the *resource allocation* problem in datacenters' shared clusters. We present ERA, which provides *elasticity* in two dimensions: (1) it scales jobs horizontally, i.e., by adding or removing resources to or from the running jobs, and (2) it dynamically changes, at will, the per-GPU batch size to control the utilization of each GPU. We demonstrate that simultaneous scaling in both dimensions improves the training time without impacting the training accuracy.

We show how to use *cloud object stores (*COS*)* to alleviate the *network bottleneck* of training transfer learning (TL) applications in *the cloud.* We propose HAPI, a processing system for TL that spans the compute and the COS tiers, enabling significant improvements while remaining transparent to the user. HAPI mitigates the network bottleneck by carefully splitting the TL application such that feature extraction is, partially or entirely, executed next to storage.

Abstract

We show how to efficiently and robustly train *generative adversarial networks (GANs)* in the *federated learning* paradigm with FeGAN. Essentially, we co-locate both components of a GAN (i.e., a generator and a discriminator) on each device (addressing the scaling problem) and have a server aggregate the devices' models using *balanced sampling* and *Kullback-Leibler weighting*, mitigating training issues and boosting convergence.

Keywords: machine learning, distributed computing, robust machine learning, Byzantine failures, federated learning, elastic training, resource allocation, datacenters, cloud computing, cloud object stores, transfer learning, generative adversarial networks.

Résumé

Les applications d'apprentissage automatique sont omniprésentes. Elles sont exécutées dans des environnements aussi divers que les centres de données, le cloud et même sur les appareils des utilisateurs tels que les ordinateurs personnels et les téléphones portables. Qu'importe l'environnement d'exécution, distribuer l'entraînement de l'apprentissage automatique semble être le seul moyen de parvenir à de l'apprentissage de haute qualité et extensible ("scalable"). Cependant, distribuer l'apprentissage automatique est difficile, notamment en raison de la nature singulière de ses applications.

Tout d'abord, l'apprentissage automatique doit être *robuste* contre les pannes arbitraires, dites *Byzantines*, en raison de son utilisation dans des applications critiques. Deuxièmement, les applications d'apprentissage présentes dans les centres de données s'exécutent sur des grappes de ressources informatiques qui sont *partagées* entre plusieurs applications, grappes pour lesquelles nous avons besoin de solutions *d'allocation de ressources* qui répondent aux demandes de calcul intensif de ces applications, tout en utilisant pleinement les ressources existantes. Troisièmement, l'exécution répartie de l'apprentissage dans *le cloud est confrontée à un goulot d'étranglement du réseau*, exacerbé par la croissance rapide de la puissance de calcul. Nous avons donc besoin de solutions qui réduisent leur impact sur les infrastructures réseau sans pour autant affecter la précision de l'apprentissage. Quatrièmement, malgré l'extensibilité et les garanties de confidentialité de l'apprentissage effectué sur les appareils personnels via *l'apprentissage fédéré*, l'hétérogénéité des capacités des appareils et de la distribution de leurs données exige des solutions robustes qui permettent de relever ces défis.

Pour atteindre cette *robustesse*, nous introduisons GARFIELD, une bibliothèque pour aider les praticiens à rendre leures applications d'apprentissage automatique résistantes aux pannes Byzantines. En plus de combler la vulnérabilité de l'architecture à graphe partagé utilisée par les cadres d'applications classiques d'apprentissage automatique, GARFIELD prend en charge divers modèles de communication, règles d'agrégation robustes et matériels de calcul (i.e., les CPUs et les GPUs). Nous montrons comment utiliser GARFIELD avec différentes architectures, configurations réseau et distributions de données.

Nous explorons l'entraînement *élastique* (i.e., la modification des paramètres d'entraînement en cours d'exécution) pour résoudre efficacement le problème *d'allocation des ressources* dans les grappes d'ordinateurs partagées des centres de données. Nous présentons ERA, qui fournit de *l’élasticité* dans deux dimensions : (1) il adapte l’extensibilité horizontale des tâches, i.e., en ajoutant ou en retirant des ressources aux tâches en cours d’exécution, et (2) il modifie dynamiquement, à volonté, la taille du lot d’apprentissage alloué par GPU pour contrôler l’utilisation de chaque GPU. Nous démontrons qu’adapter l’extensibilité simultanément dans les deux dimensions améliore le temps d’apprentissage sans impacter la précision des modèles entraînés.

Nous montrons comment utiliser le *stockage d’objets dans le cloud (COS)* pour *réduire l’utilisation du réseau* par les applications d’apprentissage par transfert dans *le cloud*. Nous proposons HAPI, un système de traitement pour l’apprentissage par transfert qui couvre les niveaux de calcul et de COS, permettant des améliorations significatives tout en restant transparent pour l’utilisateur. HAPI désengorge le goulot d’étranglement du réseau en divisant soigneusement l’application d’apprentissage par transfert, de sorte que l’extraction de variable soit, partiellement ou entièrement, exécutée proche du stockage.

Nous montrons comment entraîner de manière efficace et robuste des *réseaux adverses génératifs (GANs)* dans le paradigme de *l’apprentissage fédéré* avec FeGAN. Pour ce faire, nous co-déployons les deux composants d’un GAN (i.e., un générateur et un discriminateur) sur chaque appareil (résolvant le problème d’extensibilité) et nous demandons à un serveur d’agréger les modèles des appareils en utilisant un *échantillonnage équilibré* et une *pondération de Kullback-Leibler*, ce qui atténue les problèmes d’entraînement et accélère la convergence.

Mots clés : apprentissage automatique, informatique distribuée, apprentissage automatique robuste, panne Byzantine, apprentissage fédéré, entraînement élastique, allocation des ressources, centres de données, informatique en nuage, stockage d’objets dans le cloud, apprentissage par transfert, réseaux adverses génératifs.

Contents

Acknowledgements **i**

Preface **iii**

Abstract (English/Français) **v**

List of Figures **xiii**

List of Tables **xvii**

I Introduction and Background **1**

1 Introduction **3**
- 1.1 Thesis Context 3
- 1.2 Motivation and Contributions 4
  - 1.2.1 Byzantine Failures 4
  - 1.2.2 The Datacenter Case 5
  - 1.2.3 The Cloud Case 6
  - 1.2.4 The Federated Learning Case 7
- 1.3 Thesis Roadmap 8

2 Background **9**
- 2.1 Neural Networks 9
- 2.2 Stochastic Gradient Descent 10
- 2.3 Distributed Machine Learning 11
  - 2.3.1 Parameter Server 12
  - 2.3.2 Decentralized Learning 13
  - 2.3.3 Federated Learning 13

II Fault-Tolerant Machine Learning **15**

3 System Support for Byzantine Learning **17**
- 3.1 Introduction 17
- 3.2 Background 19

3.2.1 Byzantine Resilience . . . 19
3.3 The GARFIELD Design . . . 20
3.3.1 Statistically–Robust GARs . . . 20
3.3.2 System Components . . . 23
3.4 The GARFIELD Implementation . . . 25
3.4.1 Communication in TensorFlow . . . 25
3.4.2 Communication in PyTorch . . . 26
3.4.3 SIMT *Median* Function . . . 26
3.4.4 Memory Management . . . 27
3.5 Applications . . . 28
3.5.1 Single Server, Multiple Workers (SSMW) . . . 28
3.5.2 Multiple Server, Multiple Workers (MSMW) . . . 29
3.5.3 Decentralized Learning . . . 30
3.6 Performance Evaluation . . . 31
3.6.1 Methodology . . . 31
3.6.2 Baselines . . . 32
3.6.3 GARs Microbenchmark . . . 33
3.6.4 Convergence Comparison . . . 34
3.6.5 Tolerating Byzantine Failures . . . 35
3.6.6 The Cost of Byzantine Resilience . . . 36
3.6.7 Scalability of GARFIELD . . . 38
3.7 Related Work . . . 40
3.8 Concluding Remarks . . . 41

III Practical Tips for Efficient Training **43**

4 Elasticity is All You Need **45**
4.1 Introduction . . . 45
4.2 Background . . . 48
4.2.1 Elastic Distributed Learning . . . 48
4.2.2 Deep Learning Scheduling . . . 48
4.2.3 PyTorch–elastic and ElasticJob . . . 49
4.3 The ERA Design . . . 49
4.3.1 Overview . . . 49
4.3.2 Architecture . . . 50
4.4 The ERA Implementation . . . 54
4.4.1 The Operator . . . 54
4.4.2 State Management . . . 55
4.4.3 Garbage Collection . . . 55
4.4.4 Batch Size Synchronization . . . 56
4.5 Performance Evaluation . . . 56
4.5.1 Methodology . . . 56

4.5.2 Baselines . . . 57
4.5.3 Microbenchmark . . . 59
4.5.4 End-to-end Large Experiment . . . 60
4.5.5 Scalability of ERA . . . 62
4.5.6 Fixed Job Schedule . . . 63
4.5.7 Convergence Comparison . . . 64
4.6 Related Work . . . 65
4.7 Concluding Remarks . . . 67

5 Cloud Object Stores for Faster Transfer Learning 69
5.1 Introduction . . . 69
5.2 Background . . . 72
5.2.1 Cloud Object Stores . . . 72
5.2.2 Hardware-accelerated pushdowns . . . 72
5.2.3 Transfer Learning . . . 73
5.3 Measurement Study . . . 73
5.3.1 Status Quo . . . 74
5.3.2 Per-layer Output Size . . . 76
5.3.3 Per-layer Computation Time . . . 76
5.3.4 Batch Size vs Per-layer Memory Usage . . . 77
5.4 The HAPI Design . . . 79
5.4.1 Insights . . . 79
5.4.2 System Description . . . 80
5.4.3 Efficient Client-side Profiling . . . 82
5.4.4 Splitting Algorithm . . . 82
5.4.5 Batch Adaptation Algorithm . . . 84
5.5 The HAPI Implementation . . . 85
5.6 Performance Evaluation . . . 86
5.6.1 Methodology . . . 86
5.6.2 End-to-end Experiment . . . 88
5.6.3 Impact of Network Bandwidth . . . 89
5.6.4 Scalability on the COS . . . 90
5.6.5 Reduction in Transferred Data . . . 91
5.6.6 Batch Adaptation . . . 92
5.6.7 Breakdown of Memory . . . 93
5.7 Related Work . . . 94
5.8 Concluding Remarks . . . 95

6 Federated Learning: The Case of Generative Adversarial Networks 97
6.1 Introduction . . . 97
6.2 Background . . . 99
6.2.1 Generative Adversarial Networks . . . 99
6.2.2 Distributed GANs . . . 100

6.3 The FEGAN Design and Implementation 100
    6.3.1 Overview 101
    6.3.2 The FEGAN Algorithm 102
    6.3.3 FEGAN Design Choices 103
    6.3.4 FEGAN System Aspects 105
6.4 Performance Evaluation Methodology 107
6.5 Convergence Comparison 110
    6.5.1 Convergence in *iid* Settings 110
    6.5.2 Convergence in *Non-iid* Settings 113
    6.5.3 FEGAN's Tolerance to GAN-specific Problems 115
6.6 FEGAN System Performance 117
6.7 Related Work 121
6.8 Concluding Remarks 122

IV Concluding Remarks **123**

7 Conclusions and Future Work **125**
7.1 Summary and Implications 125
7.2 Future Directions 126

Bibliography **129**

Curriculum Vitae **153**

List of Figures

2.1 A neural network architecture. 10

2.2 Two popular distributed ML architectures: the parameter server and decentralized. 12

3.1 GARFIELD components and API. 23

3.2 Examples of applications that can be built using GARFIELD. 28

3.3 Microbenchmark of different GARs deployed on a GPU. n denotes the number of inputs to the GAR and d denotes the length of one input/gradient. 33

3.4 Convergence of GARFIELD applications with respect to other baselines using two models. 34

3.5 GARFIELD tolerance to two Byzantine attacks. 35

3.6 Slowdown of fault–tolerant systems normalized to the vanilla baseline (i.e., TensorFlow) throughput. 36

3.7 GARFIELD's overhead breakdown. 37

3.8 The scalability of GARFIELD with increasing n_w. 38

3.9 Communication time of *decentralized learning* and vanilla baseline (deployed on GPUs) with n and d. 39

3.10 The scalability of GARFIELD with f_w and f_{ps}. 39

4.1 Cluster and resource utilization with a non–elastic solution. We submit 15 training tasks to a 64–GPU cluster with a mean arrival time of 4 minutes, following the Poisson distribution. 46

4.2 The resource allocation architecture. The ERA component, highlighted in green, is our main contribution. 50

4.3 Microbenchmark experiment: we submit a single job, training ResNet18 using ImageNet. 58

4.4 The effect of choosing a different number of starting workers, with only 1 job, training ResNet18 using ImageNet. Based on our configuration, no scaling can happen beyond 16 workers. 59

4.5 End-to-end real–life scenario with 100 submitted jobs. 60

4.6 The actual effect of improving raw performance metrics (e.g., throughput and GPU utilization) on the rate of finishing jobs. 61

4.7 Scalability with a different number of submitted jobs to the cluster. All jobs start at once at the beginning of the experiment. 62
4.8 Fixed schedule results: we submit 1 job every 2 minutes. 64
4.9 The convergence of ERA (with 2 maximum batch sizes) compared to Static while training ResNet18 with ImageNet. 65

5.1 Status quo architecture vs our splitting approach for TL computation. 74
5.2 Status quo: streaming images from the COS and doing ML training entirely in the compute tier. Dataset: ImageNet. Batch size = 500. Bars marked by 'X' crashed with OOM error. 74
5.3 Per–layer output sizes for 4 state-of-the-art DNNs. Dataset: Synthetic. Batch size = 1. 75
5.4 Computation time of the forward pass per layer for 4 state-of-the-art DNNs on both CPU and GPU. Dataset: Synthetic. Batch size = 200. 76
5.5 Maximum GPU memory used for the forward pass per layer with 4 state-of-the-art DNNs. For the backward pass, the maximum GPU memory is shown across all the layers starting from the indexes in Table 5.2 until the end of the network. Dataset: Synthetic. 77
5.6 GPU memory breakdown with splitting the forward pass at different layers of 4 DNNs (x-axis). Before splitting (the blue bars), we use a batch size of 100 while after (the orange bars), we use 1000. We compare this to a no-split case (i.e., status quo) in which we use batch size of 1000. With VGG11, we experiment with split indexes whose output size is smaller than the largest input shown in Figure 5.3. Dataset: ImageNet. 78
5.7 HAPI - System Overview. 79
5.8 HAPI request flow. A TL application is split at index 10. This shows a training iteration with training batch size 3000. This results in 3 requests being sent to the HAPI server. On the COS, for each of the 3 requests, feature extraction is performed with a COS batch size of 200. 81
5.9 Execution time of BASELINE compared to HAPI with different DNNs. Bars marked by 'X' crashed with OOM error. 88
5.10 The effect of varying bandwidth on HAPI. 89
5.11 Scalability of HAPI with multi-tenants (i.e., multiple concurrent requests) compared to an *ALL_IN_COS* solution that pushes down the entire computation next to storage. 90
5.12 Transferred data with HAPI, compared to BASELINE, with different batch sizes. The y-axis shows the average data size transferred in one iteration. 91
5.13 The benefits of batch adaptation (BA). 92
5.14 Total GPU memory consumption with HAPI compared to BASELINE. 93

6.1 (a) A Parameter server-based deployment [101] with one generator and distributed discriminators (on 21 machines) outperforms the centralized approach. Yet, (b) it does not scale due to communication bottlenecks. 98

6.2 A typical GAN architecture. The generator generates data (from a noisy signal) that looks like real data. The discriminator tries to differentiate between real and generated data. Both networks learn simultaneously from the feedback, typically using a backpropagation algorithm [107]. 100

6.3 FEGAN architecture: $\mathscr{D}$ and $\mathscr{G}$ are co-located on all devices. Dashed lines denote the communication of metadata about devices' datasets and specifications. Data across devices might be unbalanced and skewed. 101

6.4 A running example of how FEGAN works. Assume a dataset of 50 samples in 4 classes, distributed on 4 devices with $C = 0.5$ (where C is the fraction of devices sampled by the server in each round). Each device has an array of 4 entries depicting how many samples this device has from each class. In the first round, the server chooses device 1 (with the highest number of samples) and device 3 (with samples from a class not represented by device 1). When the devices reply, the server updates the model, applying *KL weighting* to their replies. In the second round, the server chooses the unsampled devices to maintain fairness among devices. 102

6.5 Convergence of FEGAN compared to the *Centralized* approach and MD-GAN. We distributed the Fashion-MNIST dataset identically and independently (*iid*) on 80 devices for the distributed deployments. *FID* metric: the lower, the better the generated samples. 110

6.6 Convergence of a few FEGAN configurations in a distributed setup with the Fashion-MNIST dataset distributed identically and independently (*iid*) on 80 devices. 112

6.7 Convergence of FEGAN in a distributed setup with MNIST; data is not distributed identically and independently (*non-iid*) on devices. 112

6.8 Convergence of FEGAN in a distributed setup with Fashion-MNIST; data is not distributed identically and independently (*non-iid*) on devices. 113

6.9 Convergence of FEGAN in a distributed setup with ImageNet; data is not distributed identically and independently (*non-iid*) on devices. 114

6.10 Convergence of FEGAN with *non-iid* Fashion-MNIST data distributed on 80 devices, showing all cases of enabling and disabling weighting and sampling schemes. 115

6.11 KL divergence of the real distribution from the distribution seen by the centralized server. The smaller the KL divergence, the lower the probability of mode collapse. 116

6.12 The co-location of both networks on all devices avoids yielding a powerful discriminator (as in MD-GAN), preventing the problem of vanishing gradients. 116

6.13 Scalability of FEGAN with the number of devices. 117

6.14 The impact of the values of E and C (60 devices). 118

6.15 The average number of samples per class seen by the server, through devices' updates, as a function of FL rounds. Here, the maximum number of samples at each device is 50. (ImageNet, 80 devices). 119

6.16 Bandwidth consumption at one random device for both MD-GAN and FEGAN. We observe the same distribution (as plotted) at different sampled devices. Using FEGAN, each device is picked (by the server), on average, less than 10 times in a 200-second interval. 119
6.17 Convergence of FEGAN in the presence of weak devices in terms of memory or processing power. FEGAN is adaptive to heterogeneous devices. (ImageNet, 60 devices) 120
6.18 FEGAN tolerates the server's crash. 121

List of Tables

3.1 Models used to evaluate GARFIELD. 32

5.1 The execution time (in seconds) of one request with 4 DNNs using both versions of HAPI's server implementation: (1) inside Swift's proxy and (2) decoupled from it. 85
5.2 Models used in our evaluation of HAPI. Note that we count only layers at which splitting would make sense. For example, ResNets are structured as a sequence of blocks (Figure 5.3b); we count each block as one layer. 86
5.3 The chosen split index by HAPI with different values for the bandwidth (see Figure 5.10). 89
5.4 The behavior of HAPI's batch adaptation in the experiment in Figure 5.13. . . . 91

6.1 Time to finish 200 epochs by FEGAN and FL-VANILLA. We use *balanced sampling* with the former and random sampling with the latter. The given two numbers in the first row represent *max_class* and *max_samples*, respectively. The gain in time to finish one epoch can be also viewed as a gain in consumed bandwidth over random sampling for not looking at diverse data batches. 119

Introduction and Background Part I

1 Introduction

1.1 Thesis Context

Machine learning (ML) is a revolutionary technology that allows machines to learn from data. We do not need to give a machine exact commands to follow, instead we can simply give it some data that it can use to learn some task through a *training* process. This radical idea is driving cutting–edge solutions in many domains. Nowadays, ML applications are ubiquitous. We use ML for day-to-day tasks such as face recognition on smartphones and social media [174, 264], next word prediction while typing on mobile devices [99], voice-controlled personal assistance (e.g., using Alexa or Siri) [167, 109], content recommendation [197] (e.g., for a YouTube video or a Netflix movie) and moderation [30, 80], as well as news feed generation on social media platforms [220, 234]. Besides, ML powers critical applications such as medical diagnosis and imaging [24], self-driving cars [208, 34], detecting hate speech [134, 171, 245], fake news [6, 8], and spams [269]. Clearly, ML has a significant and beneficial effect on our lives.

Practically, executing ML training applications faces several challenges. One issue is the noticeable computing power required to train an ML model. This issue is evidenced by the amount of computing resources needed to train state-of-the-art models [7, 78, 228]. On the solution track, the last decade has seen a considerable leap in specialized hardware such as GPUs, TPUs, and AI chips by big tech players including Intel [119], AMD [16], Tesla [233], Google [87], Apple [18], and Huawei [113] to cope with the computational requirements of ML training. Yet, that is not enough. To capture fine details in training data (and hence, achieve better overall performance), ML models are getting larger and more complicated [133, 132, 239]. In addition, the amount of data used for training is also constantly increasing, following the common wisdom: the more data you have, the better your ML model performs. For instance, the size of Google's ad impression log (to track the web pages on which an ad appears) to train an ad click predictor can reach trillions of examples [135], each representing a high-dimensional feature vector. Such a dataset expands daily with new instances [175].

Another critical issue that faces ML training is *data privacy* [165, 179]. Especially with the new regulations for data protection, such as the *General Data Protection Regulation (GDPR),* it is crucial to care about the privacy of training data (that might include sensitive information). Therefore, one open question is how we can train ML models without breaking data privacy.

All these challenges call for *distributed machine learning* solutions [177, 156, 51]. The general idea behind these solutions is to have multiple devices or machines perform computation independently and aggregate the results of their training periodically. Practical distributed system considerations such as the scale of the distributed deployment (how many machines contribute to training), how machines communicate, and how frequently they communicate depend on many factors such as the application requirements, the data sources, and the application scope. Each distribution scheme has its constraints and challenges [240]. Although some of these challenges are not entirely new to computer systems, the unique nature of ML opens the door for new solutions that are well-suited to ML and therefore perform better with ML workloads compared to non-ML, known solutions.

This thesis revisits some of these challenges, pervasive to computer systems, and gives practical ML-suited solutions as presented in the following. In other words, this thesis improves the state-of-the-art of distributed training by making it more efficient and robust.

1.2 Motivation and Contributions

We tackle four practical problems, all concerning distributed training. Indeed, this thesis looks at the same coin, i.e., *putting distributed ML to work*, from different angles. We first give a high-level overview of the problems addressed in this thesis, along with our solutions to them.

Distributed ML architectures. Architecturally, there are two main ways to connect machines for training. The first is following the now-classical *parameter server* architecture [158, 156]. In short, a central *server* holds the model parameters (e.g., weights of a neural network) while a set of *workers* performs gradient computation (e.g., using backpropagation [107]), typically following *stochastic gradient descent* (SGD) [212] (a standard optimization algorithm) on their local data, using the latest model they pull from the server. This server, in turn, gathers the updates from the workers in the form of *gradients* and aggregates them, usually through averaging [138]. The alternative and more decentralized approach does not distinguish servers and workers. Each node has a copy of the model and keeps its data locally [38, 195, 105], typically to protect this data and save bandwidth or devices' batteries. In this approach, all nodes apply SGD and communicate (in a peer-to-peer fashion) what they learned so far to refine their models, usually also through averaging.

1.2.1 Byzantine Failures

The fragility of distributed ML. Despite the performance benefits of the presented distributed ML architectures, these distribution schemes are fragile because the typical way to aggregate results from participating machines, i.e., *averaging*, does not tolerate a single corrupted input [32] as a single carefully-crafted input can jeopardize the training process. Notably, the multiplicity of machines increases the probability of misbehavior somewhere in the network. This fragility is problematic because ML plays a central role in safety-critical tasks such as driving and flying people, diagnosing their diseases, and recommending treatments to their doctors [151, 74]. We should leave as little room as possible for the routinely reported vulnerabilities [191, 29, 81] of today's ML solutions.

Byzantine ML in theory. Over the past few years, a growing body of work [32, 11, 47, 253, 27, 67, 229, 242] took up the challenge of *Byzantine-resilient ML*. The Byzantine failure model, as formerly introduced in the distributed computing literature [150, 42], encompasses crashes, software bugs, hardware defects, message omissions, corrupted data, and, even worse, hacked machines [28, 250]. This problem has been well-studied from a theoretical point of view. Different solutions were proposed for diverse architectures, including the setup with one trusted central server collecting the replies of Byzantine workers [32, 47, 60], the architecture of replicated servers communicating together to agree on a model [70, 71], and the fully decentralized setup with no central server [268, 69]. However, none of these works considered the practical costs of Byzantine resilience, and therefore the relevance of their results remains mostly theoretical.

Our *one-size-fits-all* solution. We present GARFIELD [92], a library that enables the development of Byzantine ML applications on top of popular frameworks such as TensorFlow [1] and PyTorch [193] while achieving *transparency*, i.e., applications developed with either framework do not need to change their interfaces to tolerate Byzantine failures. Essentially, GARFIELD addresses the Byzantine ML problem from a pragmatic point of view, unlike most of the previous works, which are mainly theoretical. The practical deployment of these solutions helps find insightful facts about the costs of Byzantine resilience in ML systems in real-world scenarios. GARFIELD guides the ML practitioners to develop their applications from a system's perspective rather than from an algorithmic perspective; this helps achieve both transparency and Byzantine resilience. GARFIELD allows for synchronous and asynchronous communication and includes several *statistically-robust gradient aggregation rules* (GARs). GARFIELD's design is generic: future GARs can be integrated with it with minimal coding effort. GARFIELD also supports full-stack computations on both CPUs and GPUs. We consider it a *one-size-fits-all* solution, as we show how to use it to achieve Byzantine resilience in various distinct scenarios.

1.2.2 The Datacenter Case

Training on shared clusters. In a datacenter, training tasks (also called jobs) usually run on shared clusters of powerful computing resources [103, 124], e.g., GPUs. Commonly, users submit their training jobs to a central controller that decides how to schedule them to run on the cluster. With the immense computing demand of today's ML applications, it is challenging to satisfy the resource demands of all the applications simultaneously [123]. Previous approaches [90, 251] proposed several *ML-aware* scheduling techniques, where the goal is to come up with a schedule that minimizes the average job completion time and maximizes resource utilization. Most of these proposals share one limitation: they give a *fixed* amount of resources to each job throughout the training. This design hinders achieving optimal performance. On the one hand, the cluster can be saturated quickly with many users' requests, leading to longer queuing delays. On the other hand, the case of not having so many requests leads to resource underutilization. Unfortunately, it is typically hard to predict the optimal number of devices and the best training batch size [26] required for training. Even with the most expert ML practitioners, the best parameters for resource utilization usually depend on the available hardware, which users do not always know in advance.

Elastic resource allocation. Our key to resolving this underutilization is to allow training jobs to be *horizontally elastic*, i.e., use a variable number of machines throughout training. In addition, we dynamically change the batch size (i.e., the computation granularity) during training to boost throughput and resource utilization. Our insight is that training can utilize different amounts of resources throughout its lifetime, allowing for better training throughput and resource utilization *without* impacting the final accuracy. We realize this with ERA, our *Elastic Resource Allocator* [95] that is designed specifically for *multi-tenant* ML clusters. ERA provides elasticity in two dimensions: the number of machines (or workers) contributing to training and the per-machine batch size. ERA scales up jobs that benefit from having more resources and scales down jobs to maximize the number of concurrently running jobs on the cluster while being fair to all jobs. Furthermore, ERA detects the underutilization of GPU memory and processing cycles and scales the batch size accordingly. Interestingly, this dynamic behavior of ERA allows users to run the same training application on different types of hardware since ERA takes care of adjusting training parameters seamlessly.

1.2.3 The Cloud Case

The communication overhead in cloud-based applications. Today's cloud architecture decouples computing resources from storage resources [43, 13], organized in separate tiers. This design allows each tier to scale independently from one another. To process data on the compute tier, one has to stream this data first from the storage devices. The network bandwidth between both tiers is typically a bottleneck [137, 198]. Different proposals suggested reducing the amount of data transferred by carefully filtering the data on the storage tier. For instance, there have been proposals to run SQL queries on the storage tier to reduce the amount of data transferred over the network [266, 275]. In the context of ML training, the same problem of limited network bandwidth exists yet is only exacerbated by the immense amount of data typically used for ML applications. The growing compute capabilities of the cloud object stores [117, 10] constitute an opportunity to alleviate the network bottleneck of training in the cloud.

Cloud object stores to the rescue. Our idea is to *partially offload* the ML computation to the *cloud object stores (*COS*)*, primarily to reduce the amount of data transferred from storage to compute. We split the training model into two parts, where each tier executes one part. The optimal splitting point depends on the properties of the training model and the network bandwidth. We identify the fine-tuning operation of transfer learning (TL) to be the perfect fit for this split setup, mainly due to its unique structure: a combination of feature extraction and training. We introduce HAPI [94], a processing system for TL that spans both the COS (representing the storage tier) and the compute tier. HAPI is transparent to the user, uses internally only very inexpensive profiling runs, and enables even low-end CPU-only users to contribute to training. We give practical tips on how the storage tier can serve multiple concurrent users despite the limited compute and memory resources it usually has.

1.2.4 The Federated Learning Case

The premise of federated learning. Federated learning (FL) [176] seeks to leverage a large crowd of edge devices to train a global model, using their local private data that never leaves the device. The training procedure is typically orchestrated by a central server that hosts the global model that needs to be trained. In that sense, FL achieves scalability and data privacy. Yet, training in the FL paradigm naturally happens on heterogeneous devices and uses skewed data. The inherent imbalance of data distribution on the contributing devices negatively impacts the accuracy of FL [4, 281]. The problem worsens when training more complicated ML techniques such as *generative adversarial networks (GANs)* [84]. In addition to the learning difficulty, scaling their training is also difficult since GANs are typically composed of two deep networks chained together (called the discriminator and the generator). Commonly, the generator network stays on a central server while the discriminator network is distributed among multiple workers (i.e., user devices). The scaling issues arise from the communication load between the server and the workers in this setup, given the repetitive forward and backward passes that span both networks.

Scalable GAN training with FL. We propose FEGAN [91], a scalable and robust GAN training system for the FL paradigm. For scalability, FEGAN co-locates both GAN's discriminator and generator networks on the server and all workers contributing to training. For robustness, we design mechanisms to make FEGAN resilient to GAN-specific issues: *mode collapse*, *vanishing gradients*, and *learning divergence*.[1] FEGAN prioritizes updates from specific workers over others, using *Kullback-Leibler (KL) weighting*, and it carefully schedules the application of devices' updates to the global trained model using *balanced sampling*. FEGAN is also cognizant of devices' heterogeneity regarding memory and computing power. Furthermore, FEGAN tolerates server and network failures through periodic checkpointing of the learning state.

---

[1] In short, *mode collapse* is the case when the generator learns to generate *only a few* classes of the training data, *vanishing gradient* is the case when the discriminator always beats the generator (i.e., the generator always fails to generate convincing data), and *learning divergence* is the case when the learning process fluctuates indefinitely between sub-optimal states.

1.3 Thesis Roadmap

This thesis is organized into four parts that include seven chapters.

Part I	• Chapter 2 gives the essential common background of all systems presented in this thesis; this covers an introduction to neural networks, stochastic gradient descent, and distributed machine learning architectures. Each subsequent chapter gives even further background that is specific to that chapter.
Part II	• Chapter 3 presents Garfield, the first library that offers *robustness, i.e., Byzantine resilience,* to machine learning applications (written for popular but fragile frameworks such as TensorFlow and PyTorch) at a low cost.
Part III	• Chapter 4 presents ERA, an *elastic resource allocator* for training tasks in *shared* datacenters' machine learning clusters. • Chapter 5 presents HAPI, the first processing system that *partially offloads* the machine learning computation to the cloud object store, *alleviating the typical network overhead* facing cloud-based training. • Chapter 6 presents FeGAN, a scalable system that trains generative adversarial networks in the *federated learning* setup while being robust to heterogeneous devices and skewed datasets.
Part IV	• Chapter 7 concludes the thesis, giving a summary of the presented work and directions for future work.

2 Background

In this chapter, we give a broad background of the topics discussed in this thesis. This thesis addresses practical problems facing distributed machine learning (ML) training. To explain our ideas, we use neural networks as a running example of an ML application throughout the thesis. We start this chapter by giving an overview of neural networks and how to train them (via *stochastic gradient descent*). We then discuss three popular architectures to distribute training on multiple devices. In each subsequent chapter, we give additional background specific to that chapter.

2.1 Neural Networks

Neural networks [143, 31] are one of the remarkable ML tools that contributed to significant successes in the last few years. Neural networks have been successful in various applications, including object recognition [160], image analysis [17], language modeling [132], and speech recognition [98]. The idea of neural networks is to mimic how the human brain works. In simple words, input signals travel among different neurons through synapses, and by activating specific neurons, the brain can learn new things.

From an architectural point of view, a neural network is only a graph of neurons connected by edges. Neurons are usually stacked in the form of layers, as shown in Figure 2.1. Each edge holds a weight that eventually defines how the network does its task. Each neuron encompasses a mathematical function that is applied to the inputs of that neuron. The output of each neuron is then passed to other neurons in the next layer through different edges, each holding a distinct weight. The last layer gives the network's output, which could be, for example, the label of an image (e.g., cat or dog) or an embedding vector of the input.

From a mathematical point of view, a neural network is a *parameterized* mathematical function, called *model*. A toy example: $f(i) = \alpha\, i + \beta$, where the factors $\left(\alpha, \beta\right) \triangleq \boldsymbol{x} \in \mathbb{R}^d$ are called the *parameters*. These parameters are essentially the weights on the edges of the neural network.

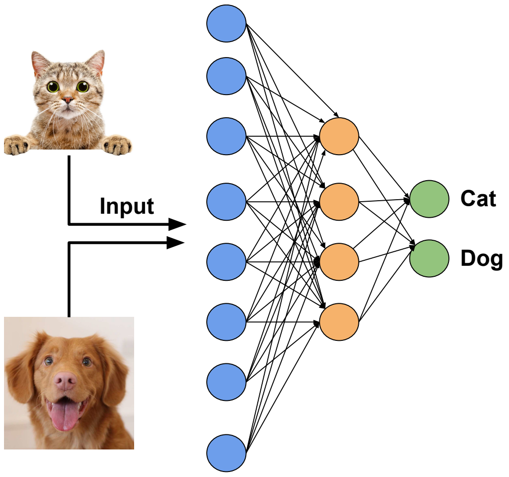


Figure 2.1: A neural network architecture.

2.2 Stochastic Gradient Descent

A natural question is: how can we find the *correct* weights of a neural network? The answer: via *stochastic gradient descent* (SGD) [212]. SGD is the most widely-used *optimization* algorithm in ML applications [51, 156]. It has become the de facto standard to optimize objective functions that can be used with ML techniques, including neural networks. Many impressive results of the last decade, e.g., in image classification challenges [142, 231], used SGD to *optimize a neural network*.

To explain how SGD works, assume the objective function (also called *loss* function) is $L(\boldsymbol{x}) \in \mathbb{R}$. This measures "how incorrect the model is, parameterized by $\boldsymbol{x}$, when labeling an input".[1] Indeed, the vector $\boldsymbol{x}$ constitutes the weights on the edges of a neural network, for example. For illustration purposes, but without loss of generality, we consider an image classification problem that uses a neural network to map an input image (e.g., the pixel colors of a picture of a cat) to a label (e.g., "cat"). The training phase then consists in tweaking the *parameters* (α and β in the toy example), noted $\boldsymbol{x} \in \mathbb{R}^d$ ($\mathbb{R}^2$ in the toy example), until the model correctly labels many images. The set of pairs of (image, correct label) used during this training phase is called the *training set*. In modern ML, the number of parameters d is large, usually above the million [248, 2, 142] and in some cases billions [55, 40]. The lower the output ($\boldsymbol{x}_{opt}$), the more accurately the neural network classifies its inputs. In short, SGD addresses the following optimization problem:

$$\boldsymbol{x}_{opt} \triangleq \underset{\boldsymbol{x} \in \mathbb{R}^d}{\arg\min}\, L(\boldsymbol{x}) \tag{2.1}$$

The procedure is iterative: in each step k,

1. Estimate the gradient $G\left(\boldsymbol{x}^{(k)}, \xi\right), \approx \nabla L\left(\boldsymbol{x}^{(k)}\right)$, with a subset ξ of size b of the training set,

---

[1] This wording is correct only when omitting any kind of regularization. This background section is only to give a broad introduction to SGD.

called *mini–batch.* This represents an approximation of the *uncomputable* real gradient, typically using the backpropagation algorithm [107]. This is a *stochastic* estimation of the uncomputable, real gradient $\nabla L\left(\boldsymbol{x}^{(\boldsymbol{k})}\right)$.

2. Then, using the estimated gradient, the model parameters ($\boldsymbol{x}$) are updated as follows:

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} - \gamma_k \cdot G\left(\boldsymbol{x}^{(k)}, \xi\right), \tag{2.2}$$

where $\{\gamma_k\}$ is called the *learning rate.*

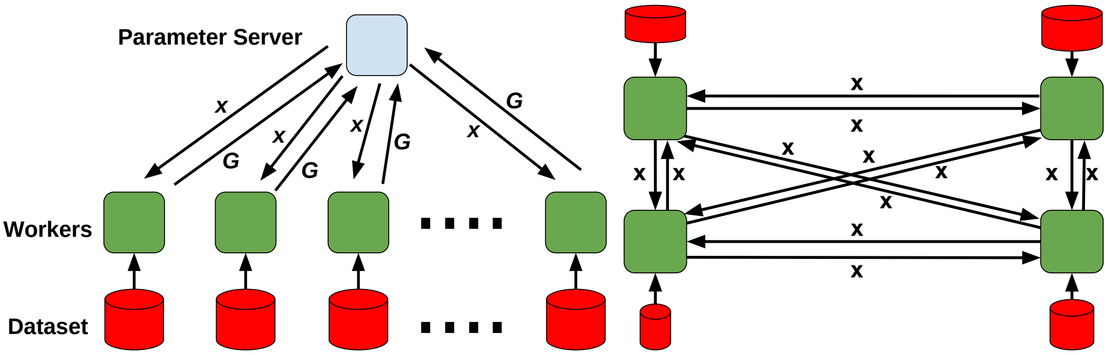
2.3 Distributed Machine Learning

Estimating the gradient at $\boldsymbol{x}$ is computationally expensive, given the big datasets and the complex high-dimensional models we have nowadays. It consists of *averaging* the b estimates of the gradient $G(x, \xi_i)$, where ξ_i is the i^{th} pair (input, correct label) from the mini–batch. Each $G(x, \xi_i)$ involves a *backpropagation* pass, so the number of arithmetic operations to carry out to estimate $G(\boldsymbol{x}, \xi) \approx \nabla L\left(\boldsymbol{x}^{(k)}\right)$ is $\mathcal{O}(b \cdot d)$.

Parallelism. Fortunately, this gradient estimation is easily parallelizable. For instance, using *data parallelism*, n machines can each partially estimate the gradient using a mini–batch of size $^b/_n$, which can be then aggregated together to restore the complete estimation. The computation of $G\left(\boldsymbol{x}^{(k)}, \xi\right)$ could be done entirely in parallel by distributing the b computations $G\left(\boldsymbol{x}^{(k)}, \xi_i\right)$ over n machines. Another approach to distributing training is *model parallelism.* In this approach, the model (e.g., a neural network) is partitioned onto multiple machines, each processing only a part of it. Both the forward and the backward passes require a synchronization step: while one machine is doing a computation, the other machines have to wait for it to finish processing and then pass the processed outputs to the next machine. Due to the synchronization overhead, this approach is less popular than data parallelism. We focus on the latter in this thesis.

Training synchrony. Training is synchronous if all machines proceed in synchronized iterations. For example, in synchronous training, no machine can move to iteration $t+1$ except when all other machines finish iteration t. On the contrary, asynchronous training does not restrict the difference between training iterations processed on different machines. While asynchronous training can yield *faster* updates to the model (due to the absence of synchronization among participating machines), synchronous training achieves *higher-quality* updates. In this thesis, we always consider synchronous training.

Network synchrony. The network is synchronous if there is an upper bound on the time it takes to transmit a message between two machines in the network. In asynchronous networks, this upper bound does not exist. Based on this definition, it is impossible to differentiate



(a) The parameter server architecture (b) The decentralized architecture

Figure 2.2: Two popular distributed ML architectures: the parameter server and decentralized.

between a delayed message and a crashed machine in asynchronous networks. In this thesis, we consider both types of networks. Note that network synchrony/asynchrony is orthogonal to training synchrony/asynchrony.

Different approaches for distributing machine learning computations have been proposed in the literature. We highlight three of them in the rest of this chapter.

2.3.1 Parameter Server

The parameter server architecture [158, 157], illustrated in Figure 2.2a, is a milestone for distributed ML, introduced to accelerate the computation over a single centralized process. This architecture accelerates training in controlled environments, e.g., in a datacenter, where communication is usually fast, and the model and the dataset exist within one building. In this model, a centralized server holds the parameters $\boldsymbol{x}$ and orchestrates the learning process, where the other machines (called workers) own data batches. For each training step, the server first broadcasts the parameters to workers, which share the heavy gradient estimation, i.e., each worker uses a mini–batch of size ${}^{b}/_{n}$. When a worker completes its gradient estimation $\boldsymbol{G}$, it sends it to the parameter server, which finally aggregates the received estimations (typically by *averaging* [36]) and updates the parameters $\boldsymbol{x}$, as in Equation (2.2).

Using the parameter server architecture, one can only employ the data parallelism scheme. Essentially, each worker holds a full copy of the model and a data shard. In a training loop, each worker trains the full model using the data shard it has. It then sends an update to the parameter server. Employing this scheme scales the quality of the update with the number of workers, as the *effective* (i.e., aggregate) batch size contributes to the update scales linearly with the number of workers.

2.3.2 Decentralized Learning

Another variant of distributing training is *decentralized learning* [237] (e.g., *AllReduce*), shown in Figure 2.2b. The main difference between this scheme and the parameter server architecture is that machines collaborate to train a model *without* a central entity. In decentralized learning, each machine owns a copy of the model and some local data that is never shared with the others [195, 105], typically to maintain the privacy of this data in addition to saving the bandwidth and the devices' batteries. Notably, the amount of data and its distribution differ from one machine to another. In each training step, each machine estimates a gradient (using SGD and its local data) and shares it with the others. Machines can connect in various ways, including using a ring structure, a tree structure, or in a peer-to-peer fashion. In Chapter 3, we use peer-to-peer communication because it gives better robustness guarantees, whereas, in Chapter 4, we use the ring structure due to its high performance. Regardless of the network topology, after machines communicate the gradient estimations, they aggregate the received estimations locally and use the aggregate estimate to update the parameters $\boldsymbol{x}$. Equivalently, machines can update their local models using their local gradient estimates and then share the updated model, as shown in Figure 2.2b.

2.3.3 Federated Learning

The federated learning (FL) [176] approach seeks to leverage a large crowd of *edge devices* (e.g., smartphones) that own private data without imposing any data movement (unlike the parameter server model, which requires moving data to the datacenter first). The devices collaboratively train a global model, which is hosted on a centralized server. The server decides which *subset of the devices* should participate in the learning round through random selection. Note that, in the parameter server architecture, all workers participate in *all* training iterations. Selected devices, in turn, send their updates to the server after *several local learning iterations* over their private data. This step constitutes a third difference compared to the parameter server architecture, wherein the latter workers do exactly one local training iteration in each step. Typically, the server aggregates the devices' updates using *federated averaging* [176]. The accuracy of models trained by FL is negatively impacted by the imbalanced data distribution on the devices [281].

Fault-Tolerant Machine Learning Part II

3 System Support for Byzantine Learning

In this chapter, we show how one can safely run distributed training in a Byzantine environment. We present GARFIELD, a library to transparently make machine learning (ML) applications, initially built with popular (but fragile) frameworks, such as TensorFlow and PyTorch, Byzantine–resilient. GARFIELD is a *one-size-fits-all* solution: it supports multiple architectures, network conditions, and resilience levels. The key idea to GARFIELD's generality is that it helps practitioners write their training algorithms from a system's perspective rather than an algorithmic perspective, reducing the coding effort and addressing the vulnerability of the shared–graph architecture followed by classical ML frameworks. GARFIELD encompasses various communication patterns, supports full-stack computations on both CPUs and GPUs, and gives multiple robust aggregation functions that are proven to be Byzantine-resilient.

3.1 Introduction

The starting point of this chapter is *distributed training*, whether the distribution architecture follows the parameter server model (see Section 2.3.1) or the decentralized model (see Section 2.3.2). As the number of participating machines in a distributed setup increases, so does the probability of failure of any of these machines. In distributed computing, the most general way to model such failures is to assume an adversary that can control a subset of the system and make it arbitrarily deviate from the normal execution: we talk about Byzantine failures [150]. This failure model includes bogus software, faulty hardware, as well as malicious attacks. With the increasing use of ML in mission-critical applications [74, 34, 208], building robust systems against these kinds of failures becomes a necessity. For instance, in a data-center environment, processing units fail [63], and training data (uploaded by the users of a service) could be malicious. Using vanilla *state machine replication* (SMR) to solve such a problem was shown impractical (and in some cases impossible) in the ML context [60].

Tolerating Byzantine machines without replicating them has been recently well-studied with various models and settings, e.g., in [11, 242, 70, 71]. The key idea is to replace the vulnerable averaging scheme (typically used to aggregate updates of participating machines) with a *statistically-robust gradient aggregation rule* (GAR), e.g., *Median* [252]. Most work on Byzantine-resilient ML (e.g., [70, 69, 32]) has, however, been theoretical, and it is not clear how to put the published algorithms to work, especially in the pragmatic form of library extensions to existing, and now classical, ML frameworks, namely TensorFlow [1] and PyTorch [193].

These frameworks share two specific characteristics that go against Byzantine resilience. First, and for performance reasons, they rely on a shared memory design. For instance, TensorFlow uses one shared computation graph among all machines to define the learning pipeline. Such a design is problematic as Byzantine nodes can corrupt the learning state at honest ones. Second, most of the high–level communication abstractions given by such frameworks assume trusted, highly–available machines. For instance, the distributed library of PyTorch allows for collective communication among processes, yet such calls block indefinitely (or fail after some timeout) in the case of a process crash or network failure. In this latter case, for example, one Byzantine machine can break all the communication rounds, disallowing the distributed system to learn.

To solve the aforementioned problems, we propose GARFIELD: a library that enables the development of Byzantine ML applications on top of popular frameworks such as TensorFlow and PyTorch while achieving *transparency*: applications developed with either framework do not need to change their interfaces to tolerate Byzantine failures. Essentially, GARFIELD addresses the Byzantine ML problem from a practical point of view, unlike most of the previous works, which are mainly theoretical. Such a library helps find insightful facts about the very practical costs of Byzantine resilience in ML systems, as we show later.

GARFIELD adopts an object-oriented vision for ML applications. The novelty of GARFIELD's design lies in (1) the way the *objects*, be they servers or workers, communicate, and (2) the way they aggregate the replies, be they models or gradients. Essentially, our insight is to write ML applications from the objects' perspective rather than from an algorithmic perspective; this enables applications written with GARFIELD to achieve both transparency and Byzantine resilience. In particular, GARFIELD allows for both synchronous and asynchronous communication and includes several *statistically-robust gradient aggregation rules* (GARs). GARFIELD also supports full-stack computations on both CPUs and GPUs.

Along our GARFIELD implementation journey, we took several design decisions to promote its practicality. We implemented specific schemes to parallelize Byzantine–resilient GARs, especially on GPUs. Moreover, we introduce the notion of *separate replicated graphs* for TensorFlow rather than relying on its *shared graph* design, as the latter would be a killer in a fully Byzantine environment, i.e., without any trusted machine. We use gRPC for point–to–point pull-based communication due to its speed and because it is currently the defacto standard communication method for popular ML frameworks. Our implementation parallelizes RPC calls to improve the scalability of algorithms implemented with GARFIELD. We also carefully manage the CPU memory to minimize memory copying and allow for faster algorithms.

We report on the usage of GARFIELD with three ML architectures: (1) tolerating Byzantine workers while assuming one trusted, central server, (2) replicating the parameter server to account for Byzantine servers as well as Byzantine workers, and (3) considering a peer-to-peer, decentralized setting with no distinction between servers and workers.

We report on our evaluation of GARFIELD, addressing the general question of the practical cost of Byzantine resilience in a distributed ML deployment. We consider various ML models and datasets, as well as different hardware, i.e., CPUs and GPUs. We also study the cost of different degrees of resilience.

Essentially, we show that Byzantine resilience introduces up to a 10% loss in accuracy compared to non–Byzantine deployments. In contrast, crash resilience does not introduce any such loss. In the latter case, the server is replicated, but Byzantine behavior is not tolerated. In terms of throughput, we quantify the overhead of various Byzantine resilience degrees compared to a vanilla deployment. We find that tolerating Byzantine servers induces much more overhead than tolerating Byzantine workers. For instance, we quantify the cost of adding Byzantine resilience to servers, compared to tolerating only Byzantine workers with a trusted server, to 53%, and the cost of Byzantine resilience, compared to the crash–tolerant baseline, to 22% (using GPUs). We attribute the resilience overhead mainly to communication. Our experiments show that communication accounts for more than 75% of the overhead, while robust aggregation contributes to only 11% of such overhead. We also highlight that Byzantine algorithms in a peer-to-peer setup do not scale, unlike those following the parameter server architecture. We also report on the fact that the overhead of Byzantine resilience depends more on the number of participating nodes, be they workers or servers, than on the model dimension. Notably, using GPUs achieves a performance improvement of at least one order of magnitude over CPUs.

GARFIELD is an open-source project and is available at https://github.com/LPD-EPFL/Garfield.

3.2 Background

3.2.1 Byzantine Resilience

In the parlance of classical distributed computing, a system tolerates a Byzantine fault when it copes with a machine that can deviate arbitrarily from the algorithm assigned to it [150]. Such a behavior abstracts all kinds of failures, including software bugs, hardware defects, corrupted data [79], communication omissions, or even adversarial attacks. This problem is well-studied in the literature on distributed systems with many proposed algorithms. We consider the ML context where any machine contributing to the learning process, i.e., a worker or a server, can behave arbitrarily. In fact, Byzantine resilience also captures the classical data poisoning problem in the ML literature [28], which addresses the issue of workers holding poisoned/corrupted data. Employing such data in a gradient computation leads to faulty gradients, corrupting the entire learning procedure [60]. In such a context, tolerating Byzantine behavior ensures following the same learning path that would have been achieved in the absence of Byzantine machines.

Is state machine replication (SMR) suitable for ML? A simple solution to the Byzantine ML problem is to replicate the computation on multiple machines (e.g., to replicate workers). While this idea would work abstractly, it faces a few practical challenges. Replicating workers in different locations might be problematic in the case of employing private data for model training [2, 221]. Even if possible, classical SMR might incur prohibitive communication costs, diminishing the benefit of distributing the ML application. Fortunately, however, the very nature of an ML computation requires an *eventual* form of consistency as only convergence to a *good* state is required. There is no notion of total ordering that should be preserved among workers' updates; this drastically reduces the difficulty of the Byzantine problem.

3.3 The GARFIELD Design

GARFIELD is a library to build Byzantine–resilient SGD–based ML applications that rely on (1) robust aggregation and (2) communication of plain vectors (without compression nor quantization). GARFIELD builds on the insight that distributed ML algorithms do not require strong consistency; only convergence to high accuracy is required [32, 73]. This insight has two implications: (1) the servers do not need to run *SMR*, and (2) the servers do not need to collect replies from all workers; only a subset of replies is sufficient. This latter observation is a cornerstone in tolerating Byzantine behavior in asynchronous networks [73, 69]. This weak consistency model has been shown sufficient for correctness and convergence in theory, even in asynchronous environments. Essentially, GARFIELD masks the faults at the application layer, abstracting all the faults of the lower layers like communication omissions [60].

We first introduce a few *statistically–robust gradient aggregation rules (GARs)* that are included in GARFIELD and then, we discuss the modular design of GARFIELD.

3.3.1 Statistically–Robust GARs

A GAR is merely a function of $\left(\mathbb{R}^d\right)^q \rightarrow \mathbb{R}^d$, with d, the dimension of the input vector space $\mathbb{R}^d$ (i.e., a gradient or a model), and q, the number of input vectors to be aggregated. All GARs are synchronous, i.e., wait for q vectors before applying an aggregation function on them. Hence, in synchronous, non–faulty settings, these GARs can be deployed with q machines in the system (so that the aggregator node can gather replies from *all* nodes in the system within some time limit). Yet, in asynchronous networks, one would require to deploy $q+f$ nodes to use these GARs to ensure the liveness of the protocol, where f denotes the maximum number of Byzantine inputs. In short, all GARs output a vector with special statistical properties that make them safe to use in the Byzantine setting.

1. *Median* [252] outputs a vector of coordinate-wise medians among the input vectors. We formally define the *median* function (applied per-coordinate) as follows:

$$\forall q \in \mathbb{N} - \{0\},\ \forall \left(x_1 \dots x_q\right) \in \mathbb{R}^q, \tag{3.1}$$

$$median\left(x_1 \dots x_q\right) \triangleq x_s \in \mathbb{R} \tag{3.2}$$

such that with $sorted\left(x_1 \dots x_q\right)$,

$$\begin{cases} x_s = x_{\lceil \frac{q}{2} \rceil} & \text{if } q \text{ is odd} \\ x_s = \frac{x_{\frac{q}{2}} + x_{\frac{q}{2}+1}}{2} & \text{if } q \text{ is even.} \end{cases} \tag{3.3}$$

We formally define the coordinate–wise median (i.e., *Median*) as follows:

$$\forall \left(d, q\right) \in (\mathbb{N} - \{0\})^2,\ \forall \left(x_1 \dots x_q\right) \in \left(\mathbb{R}^d\right)^q, \tag{3.4}$$

$$Median\left(x_1 \dots x_q\right) \triangleq x_s \in \mathbb{R}^d \tag{3.5}$$

such that:

$$\forall i \in [1 \, .. \, d],\ x_s[i] = median\left(x_1[i] \dots x_q[i]\right) \tag{3.6}$$

Median requires $q \geq 2f + 1$, and its asymptotic complexity is $\mathcal{O}(qd)$.

2. *Krum* [32] assigns a *score* to each vector (equals to the sum of ℓ_2 distances with the closest $q - f - 2$ neighbors) and then returns the smallest scoring vector. *Multi–Krum* (a variant of *Krum*) instead averages the m smallest scoring vectors, achieving a better convergence rate than *Median* [60]. Formally, given vectors $\left(x_1 \dots x_q\right) \in \left(\mathbb{R}^d\right)^q$, *Multi–Krum* averages the $m = q - f - 2$ vectors with the smallest scores (i.e., ℓ_2 norm from the other vectors). We define the score of each vector $s(i)$ as follows:

$$s(i) = \sum_{i \to j} \|x_i - x_j\|^2, \tag{3.7}$$

where $i \to j$ means that x_j is among the $q - f - 2$ closest vectors to x_i. *Multi–Krum* then chooses a set $\mathcal{S}$ of vectors as follows:

$$\mathcal{S} = (m) \operatorname*{arg\,min}_{i \in \{1,\dots,q\}} s(i), \tag{3.8}$$

where given a function $s(i)$, $(m) \operatorname{arg\,min}(s(i))$ denotes the indexes i with the m smallest $s(i)$ values. *Multi–Krum* then returns the average of vectors whose indexes $\in \mathcal{S}$. *Krum* sets $m = 1$. Both *Krum* and *Multi–Krum* require however $q \geq 2f + 3$, and their asymptotic complexity is $\mathcal{O}(q^2 d)$.

3. *MDA* [211] finds a subset group of vectors of size $q - f$ with the minimum diameter among all other subsets, where the diameter of a group is defined as the maximum distance between any two vectors of this subset. *MDA* then outputs the average of the chosen subset. We formally define the *MDA* function as follows:

Let $\left(x_1 \ldots x_q\right) \in \left(\mathbb{R}^d\right)^q$, and $\mathcal{X} \triangleq \left\{x_1 \ldots x_q\right\}$ the set containing all the input vectors.

Let $\mathcal{R} \triangleq \left\{\mathcal{M} \mid \mathcal{M} \subset \mathcal{X},\ |\mathcal{M}| = q - f\right\}$ the set of all the subsets of $\mathcal{X}$ with a cardinality of $q - f$, and let:

$$\mathcal{S} \triangleq \underset{\mathcal{M} \in \mathcal{R}}{\arg\min} \left(\max_{(x_i, x_j) \in \mathcal{M}^2} \left(\left\| x_i - x_j \right\| \right) \right). \tag{3.9}$$

Then, the aggregated gradient is given by:

$$MDA\left(x_1 \ldots x_q\right) \triangleq \frac{1}{q - f} \sum_{x \in \mathcal{S}} x. \tag{3.10}$$

Notably, *MDA* carries an exponential[1] asymptotic complexity of $\mathcal{O}\left(\binom{q}{f} + q^2 d\right)$. Yet, as we will discuss later, its assumptions about the variance between input vectors are weaker than for the previous two GARs. It requires $q \geq 2f + 1$.

Algorithm 1 The *Bulyan* algorithm.

```
1:  Bulyan ($f, [x_1, ..., x_q]$):
2:    $\theta = n - 2f$
3:    $\beta = \theta - 2f$
4:    $X^{ext} = Array[\theta][d]$                        ▷ set to store extracted vectors.
5:    $M = Array[d]$                                      ▷ set to store coordinate-wise medians.
6:    $C = Array[\beta][d]$                               ▷ $\beta$ set to store closest coordinates to the median.
7:
8:    For ($i \in [1, ..., \theta]$) do:
9:      $x_{sel} = f([x_1, ..., xq] \setminus X^{ext})$        ▷ $f$ is another GAR, e.g., Krum.
10:     $X^{ext} = X^{ext} + x_{sel}$                      ▷ '+' denotes append to list.
11:   $M = Median(X^{ext})$
12:   For ($j \in [1, ..., d]$) do:
13:     $C[:][j] = Argpartition(|X^{ext}[:][j] - M[j]|, \beta)$   ▷ the closest $\beta$ coordinates to M[j]
14:   return $Average(X^{ext}[C])$                      ▷ coordinate-wise average of selected coordinates
15: end function
```

4. *Bulyan* **[72]** (explained in Algorithm 1) robustly aggregates q vectors by iterating $\theta = q - 2f$ times over another Byzantine–resilient GAR, e.g., *Krum* (lines 8–10). In each iteration, *Bulyan* removes the vector selected by such a GAR from the input set in the subsequent iterations and puts that vector in a *selection* set. When the size of the selection set becomes $\theta = q - 2f$, *Bulyan* computes the coordinate-wise median of the vectors in that set (line 11). For each coordinate, *Bulyan* extracts the closest $\beta = q - 4f$ values (from the selection set) to the computed median for that coordinate and averages these values, i.e., it computes the coordinate-wise average of the $\beta = q - 4f$ closest values to the median (lines 12–13). Finally, *Bulyan* outputs the vector of

---

[1] Exponential when $f = \mathcal{O}(q)$, polynomial when $f = \mathcal{O}(1)$.

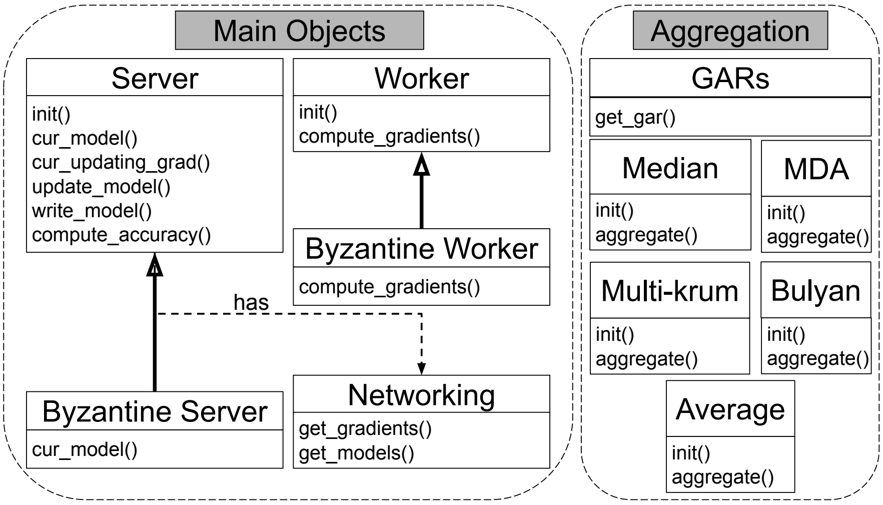


Figure 3.1: GARFIELD components and API.

all the coordinate-wise averages. Unlike previous GARs, *Bulyan* can sustain a model with a large dimension. Yet, it requires $q \geq 4f+3$, and its asymptotic complexity is $\mathcal{O}(q^2 d)$.

Tradeoffs. In addition to the differences in the ratio of tolerated Byzantine nodes (inequalities relating q with f) and the computational cost of each GAR, the model dimension is also crucial in deciding which GAR to use. For high dimensions (e.g., order of millions) and a strong adversary, one should use *Bulyan*. In low dimensions, the application setup should satisfy the *variance* assumption, as given below:

$$\exists \kappa \in \,]1, +\infty) \, , \; \forall \left(i, t, \theta\right) \in \left[1 \mathinner{..} q-f\right] \times \mathbb{N} \times \mathbb{R}^d, \tag{3.11}$$

$$\kappa \, \Delta \sqrt{\mathbb{E}\left(\left\| g_t^{(i)} - \mathbb{E}\, g_t^{(i)} \right\|^2\right)} \leq \left\| \nabla L\left(\theta\right) \right\|, \tag{3.12}$$

where,

$$\Delta = \begin{cases} \frac{2\sqrt{2}f}{q-f} & \text{if GAR} = \textit{MDA}\ [72] \\ \sqrt{2\left(q-f+\frac{f(q-f-2)+f^2(q-f-1)}{q-2f-2}\right)} & \text{if GAR} = \textit{Krum}\ [32] \\ \sqrt{q-f} & \text{if GAR} = \textit{Median}\ [252], \end{cases} \tag{3.13}$$

where $g_t^{(i)}$ is the estimated gradient by an *honest*[2] worker i at time t, and $L(\theta)$ is the loss function at the model state θ.

3.3.2 System Components

We have designed GARFIELD in a modular way, as shown in Figure 3.1. In this section, we describe each of these modules.

---

[2] Note that we cannot assume the same for Byzantine workers, which can act arbitrarily.

Main objects. GARFIELD defines two main objects that can be used for learning: Server and Worker. The server is responsible for storing and updating the model state while workers train this model using their local data. The server typically initiates a learning step by asking a few workers to compute a gradient estimate given the model state it owns. For this, the server object exposes the method *get_gradients()*, which we describe later in detail. Some algorithms require servers to exchange their model states, and hence the server object comes with the method *get_models()*, which is the second key method in our *Networking* interface, as we describe below. In addition to the *Networking* methods, the server object exposes methods to (1) update the model state, given some optimizer and a gradient estimate, (2) re-write the model, which is useful in the case of multiple server replicas, and (3) compute accuracy, given the model state and a test set.

The worker design, on the other hand, is much simpler. The worker object is passive in the sense that it only responds to requests of the server. Basically, the worker owns some data and defines some loss function. Its main job is to compute a gradient estimate (when asked by the server) using the data chunk it owns. The worker then replies to the server by the gradient estimate it computed.

To support experimenting with Byzantine behavior, GARFIELD defines two objects: *Byzantine Server* and *Byzantine Worker*, which inherit from our main objects, Server and Worker, respectively. Both objects implement the popular attacks used in the Byzantine ML literature including simple ones like reversing vectors, dropping vectors, or random vectors [60, 70] and the state–of–the–art attacks like *little is enough* [25] and *fall of empires* [256].

Networking. Existing networking abstractions in both frameworks, TensorFlow and PyTorch, are not enough to be used (1) in a Byzantine, asynchronous environment and (2) with replicated parameter servers.[3] For example, one can deploy distributed training on PyTorch using *DistributedDataParallel()* or on TensorFlow by choosing the *ParameterServerStrategy()*. However, both are high–level abstractions that assume trusted, always–available machines.

GARFIELD (as a part of the *Networking* API) supports two abstractions to handle communication: *get_gradients()* and *get_models()*. The first one is used to read the computed gradients by the workers. It accepts two parameters: t, the index of the current iteration, and q_w, the number of workers from which a server should receive replies with $q_w \leq n_w$ (n_w denotes the total number of workers); $q_w = n_w$ denotes synchronous communication with no faults in the system, i.e., a server is expecting to receive replies from *all* workers. This function then returns the fastest q_w gradients it receives. The second abstraction works in the same way, yet fetching models from servers instead of gradients from workers. Both abstractions then enable easy and natural communication among all machines in the network in both synchronous and

---

[3] Not to be confused with replicated graphs in TensorFlow. What we mean by *replicated parameter servers* is the case where the server needs to be replicated where these replicas (all have the same graph) are independent, rather than shared between machines, and they do exactly the same computation on the same data for fault tolerance rather than for performance. Both kinds of replication can be also combined.

asynchronous settings (as we describe in Section 3.5). We give the implementation details of these abstractions in Section 3.4.

Aggregation. GARFIELD implements the Byzantine–resilient GARs presented in Section 3.3.1 on both CPUs and GPUs. We create wrappers (including dependency management and automatic compilation and loading) to use them as *custom operations*[4] in both TensorFlow and PyTorch. Such wrappers make it possible to involve the GARs with the same interface for both frameworks, though the lower–level interfaces each framework provides differ substantially.

To use a GAR, the common interface consists in two functions: *init()* and *aggregate()*. The *init()* function takes the name of the required GAR (e.g., “median”), the value of n, the total number of inputs, and f, the maximum number of Byzantine inputs. The second function, *aggregate()*, takes n *tensors* (could represent gradients or models) and outputs the aggregated one. Whether this function will execute on a CPU or a GPU depends on the device on which the input vectors are stored. In this way, our design abstracts the device, CPU or GPU, and the framework, TensorFlow or PyTorch, away from the developer.

3.4 The GARFIELD Implementation

First, we present how we implement the communication abstractions, i.e., *get_models()* and *get_gradients()* in TensorFlow and PyTorch. Then, we show how we implement an efficient version of the *median* function (which is used in *Median* and *Bulyan* GARs) on GPUs.[5] Finally, we discuss some tricks we employ for better memory management. We have open-sourced our code of GARFIELD [96].

3.4.1 Communication in TensorFlow

TensorFlow adopts the notion of a shared dataflow graph in which all computations are defined in one graph, even if deployed in a distributed environment, where all participating nodes share this graph. This is a critical vulnerability in the Byzantine setting, as Byzantine nodes can write and execute code on the other honest nodes [60]. Also, such a shared graph abstraction hides the data communication among workers and servers, reducing the programming flexibility and disallowing having multiple communication rounds per learning step, which is crucial for Byzantine resilience in some cases [70].

We follow another route in which all nodes create an independent yet replicated graph. Though this design has high memory overhead, we believe it is necessary to tolerate adversarial behavior.[6] In addition to resolving the vulnerability, this design allows for more flexible

---

[4] For example, https://www.tensorflow.org/guide/create_op

[5] We also include GPU implementation of other GARs yet, we focus on *median* because its GPU implementation is challenging as we discuss.

[6] Such an overhead could be reduced if the environment is Byzantine–free.

communication patterns among the participating nodes. We use gRPC for communication and protocol buffers [238] for serializing and deserializing data. We use the pull model for transferring data: when a node, be it a worker or a server, needs some data, it pulls this data from the other nodes by initiating multiple remote procedure calls to such nodes. Each node implements a server that serves these requests. We define the protocol buffers which encode data exchanged between participants. We parallelize the replicated communication between workers and servers for requesting gradients and updated models so as to reduce the communication time as much as possible. However, abandoning the highly optimized TensorFlow distributed runtime and using independent graphs on each node require context switches between TensorFlow and Python runtimes (as protocol buffers currently cannot serialize tensors directly). Concretely, when a node is requested to send a gradient or a model, it serializes the requested data to a protocol buffer, exiting the TensorFlow graph/runtime. On the receiver side, a node deserializes the received bytes back to a tensor. Our experiments show that the overhead of these conversions (including memory copying) is non-negligible.

3.4.2 Communication in PyTorch

We implement the same abstractions in PyTorch yet with a slightly different design compared to the TensorFlow one. First, there is no context switch between PyTorch and Python since PyTorch gives communication abstractions that can be used directly on tensors. Second, we pipeline the communication with aggregation (whenever possible) as PyTorch gives access to gradients of each layer in the deep network separately; this allows for better utilization of both network and computation devices and hence, better scalability. Third, in addition to RPC support,[7] our networking library also supports the distributed *communication collectives* of PyTorch.[8] In the latter case, our implementation automatically chooses the best communication backend between *nccl* and *gloo* to allow GPU-to-GPU communication whenever possible. This is a plus compared to the RPC–based implementation as the latter does not allow communication over GPUs.

3.4.3 SIMT *Median* Function

Our implementation of the *median* function on CPU is straightforward: each of the $m \geq 1$ available cores processes a continuous share of $^n/_m$ coordinates. Then each core applies, for each coordinate of its share, *introselect* (or equivalent) by calling the standard C++ `std::nth_element`.

Nevertheless, even embarrassingly parallel algorithms like *median* would not necessarily benefit from running on GPGPUs (General-Purpose computing on Graphics Processing Unit). That is because modern GPGPUs, to achieve parallel execution on many threads while limiting instruction fetch costs, batch threads into groups of, e.g., 32 threads that execute the

---

[7] PyTorch fully supports gRPC communication only in *v1.6*.

[8] https://pytorch.org/docs/stable/distributed.html

same instruction.[9] Algorithms like *introselect* [183] are branch–intensive, with possibly many instructions executed in each branch, and so fail to scale on GPUs.

Reminiscent of [130], our implementation of *median* is built around a primitive that orders 3 elements without branching, by the use of the *selection instruction*, which converts a predicate to an integer value. Let `v` be the table of size 3 to reorder by increasing values. Thanks to the selection instruction, we can compute

```
int[] c = {v[0] > v[1], v[0] > v[2], v[1] > v[2]},
```

where `a > b` is 1 if $a > b$ else 0.
Then using the intermediate results (found by solving the "reordering truth tables"):

```
int[] i = {
   (1+c[0]+2*c[1]+c[2]-(c[1]⊕c[2]))/2,
   (4-c[0]-2*c[1]-c[2]+(c[0]⊕c[1]))/2 },
```

we can finally reorder the elements of `v` into `w` with:

```
int[] w = {v[i[0]], v[3-i[0]-i[1]], v[i[1]]}.
```

Using this reordering primitive, we manage to implement an efficient version of the *QuickSelect* algorithm (i.e., with minimal branching), which we use to compute the *median*.

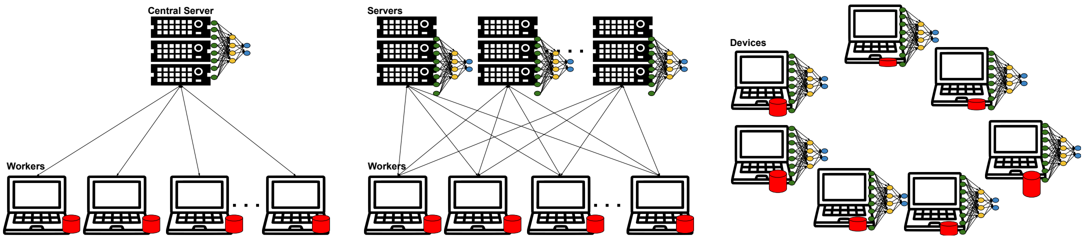
3.4.4 Memory Management

We describe here a few tricks that we use to minimize memory footprint and reduce copying data between the CPU and the GPU memory. First, whenever possible, we pin training data to memory. This pinning impacts the time it takes to copy data to the GPU memory for computing gradient estimates on workers. On the other hand, we do not pin the test set to memory as using it is usually much less frequent than the training data. Second, we pin model weights in the parameter server main memory as gRPC currently does not allow communicating values that reside on the GPU memory. Given that the model weights are communicated in each round, we never copy such weights to the GPU memory (except when testing the accuracy). Yet, we store the model on workers on the GPU (or multiple GPUs whenever possible) so as to accelerate gradient computation.

We carefully optimize the memory used by our GARs. For example, aggregating gradients may require multiple iterations, calculating some distance-based scores for each of them in each iteration, e.g., with *Multi-Krum* or *Bulyan*. We then cache the results of each of these iterations (in the CPU or GPU memory) and hence remove redundant computations. Besides, we reduce the memory cost by allocating space only for one iteration along with the intermediate selected gradients, further reducing the memory cost.

---

[9] In the case of branching, the threads execute in *lock-step*.



(a) Single server, multiple workers (b) Multiple server, multiple workers (c) Decentralized learning

Figure 3.2: Examples of applications that can be built using GARFIELD.

3.5 Applications

In this section, we show how one can use GARFIELD to build Byzantine ML applications. We give three examples (depicted in Figure 3.2) that span different architectures for Byzantine–resilient algorithms. We use the following notations in this section: nw is the number of workers, fw is the number of Byzantine workers, nps is the number of parameter servers, and fps is the number of Byzantine parameter servers. When there is no distrinction between servers and workers (e.g., in Section 3.5.3), we use n and f to denote the number of machines and Byzantine machines respectively.

3.5.1 Single Server, Multiple Workers (SSMW)

Our first application represents the standard setup that was vastly studied in the last few years, e.g., in [32, 11, 60]. Such a setup uses the vanilla parameter server architecture [158] (see Figure 3.2a), yet with one crucial difference: instead of aggregating the workers' updates by averaging them, the server uses a statistically–robust GAR (see Section 3.3.1) for aggregation. Such setup usually assumes a synchronous network, i.e., there is an upper bound on the time it takes the workers to compute gradients and reply to the server.

Listing 3.1 shows how to build such a setup, using GARFIELD, in a few lines of code.

Listing 3.1: The implementation of *SSMW* setup with GARFIELD.

```python
from garfield import Server, GARs
# parsing training arguments
ps = Server(...) #args omitted for brevity
for i in range(num_iter):
  gradients = ps.get_gradients(i, nw)
  aggr_grad = gar(gradients=gradients, f=fw)
  ps.update_model(aggr_grad)
  if i%comp_acc_freq == 0:
    acc = ps.compute_accuracy()
```

Listing 3.1 depicts the code on the parameter server side, given the passiveness of workers in our design. First, the server object is initiated (line 3) with the appropriate parameters. Then, the training loop (lines 4–7) runs as much as the user specifies. In each training iteration, the server first asks the workers to compute gradient estimates using the current model state. Note that the second argument in *get_gradients()* method specifies the number of replies the server should wait for before continuing the iteration. The server then applies some GAR on the received gradients and then updates the model using the aggregated gradient. Periodically, the server computes the accuracy of the current model state (lines 8–9).

Notably, AggregaThor's official code base [62] (which implements this setup) is in the order of thousands of lines of code.

3.5.2 Multiple Server, Multiple Workers (MSMW)

Our second application extends the first one by considering multiple servers and multiple workers. This setup was considered recently [70, 71] to tolerate Byzantine servers as well as Byzantine workers. Such a setup requires replicating the server on multiple machines (see Figure 3.2b). It can also accommodate asynchronous networks in the sense that it does not assume any upper bound on the computation nor communication delays.

Listing 3.2 shows how to build *MSMW* setup, allowing multiple server replicas and asynchronous communication.

Listing 3.2: The implementation of *MSMW* setup with Garfield.

```
1  from garfield import Server, GARs
2  # parsing training arguments
3  ps = Server(...) #args omitted for brevity
4  for i in range(num_iter):
5    gradients = ps.get_gradients(i, nw-fw)
6    aggr_grad = gar(gradients=gradients, f=fw)
7    ps.update_model(aggr_grad)
8    models = ps.get_models(nps-fps)
9    aggr_models = gar(gradients=models,f=fps)
10   ps.write_model(aggr_models)
11   if i%comp_acc_freq == 0:
12     acc = ps.compute_accuracy()
```

We focus here again on the server side. Lines 1–7 are very similar to those in Listing 3.1 (except for the number of expected gradients to collect in line 5). Lines 8–10 show the additional communication step required among the server replicas to ensure consistency among servers' parameters and hence, convergence. In such a step, each server first fetches the updated model from a few other servers (whose number is given as an argument to *get_models()* method). Then, it aggregates the collected models and stores the result as the new model.

3.5.3 Decentralized Learning

Our third application considers the Byzantine ML problem in decentralized settings as in e.g., [268, 267, 69]. Decentralized learning is different from the previous two use cases as it does not use the parameter server architecture (see Figure 3.2c), i.e., communication is done in a peer–to–peer fashion. Such a setup is mainly useful when data is sensitive and should be kept private. Notably, it addresses settings where data is not identically distributed on the contributing machines.

Listing 3.3 shows how to build such an application, allowing for a decentralized setup and multiple communication steps per training iteration, which are mainly required to keep weak consistency among models of the contributing nodes.

Listing 3.3: The implementation of decentralized learning with GARFIELD.

```
1  from garfield import Server, Worker, GARs
2  # parsing training arguments
3  wrk = Worker(...) #args omitted for brevity
4  ps = Server(...) #args omitted for brevity
5  for i in range(num_iter):
6    gradients = ps.get_gradients(i, n-f)
7    aggr_grad = gar(gradients=gradients, f=f)
8    if non_iid:
9      aggr_grad = contract(...)
10   ps.update_model(aggr_grad)
11   models = ps.get_models(n-f)
12   aggr_models = gar(gradients=models,f=f)
13   ps.write_model(aggr_models)
14   if i%comp_acc_freq == 0:
15     acc = ps.compute_accuracy()
16
17 def contract(...):
18   for _ in range(steps):
19     ps.latest_aggr_grad = aggr_grad
20     aggr_grads = ps.get_aggr_grads(n-f)
21     aggr_grad = gar(gradients=aggr_grads, f=f)
22   return aggr_grad
```

Note that the function *get_aggr_grads* (used in line 20) is used to fetch *latest_aggr_grad* (defined in line 19) from the other machines. *get_aggr_grads* then fetches the aggregated gradients from the previous step. We define this function solely to serve algorithms that require multiple aggregations of gradients in a single SGD step. We highlight here two main differences compared to the previous examples. First, each node creates both Server and Worker objects (lines 3–4). Second, there is a multi-round step (lines 17–22) in each iteration to *contract* models on correct machines, especially when the data is not identically distributed on them. The goal of this step is to force the model states on all machines to get closer to each other (i.e., to achieve *approximate* consensus [69]).

3.6 Performance Evaluation

We first describe the settings we use and the baselines we consider. We then show microbenchmarks for the GARs we implemented, followed by large–scale experiments, evaluating GARFIELD using the applications given in the previous section.

3.6.1 Methodology

Testbed. Our experimental platform is Grid5000 [89]. For the experiments deployed with CPUs, we employ nodes from the same cluster, each having 2 CPUs (Intel Xeon E5-2630 v4) with 10 cores, 256 GiB RAM and 2×10 Gbps Ethernet. For the GPU-based experiments, we employ nodes from two clusters (due to the limited number of nodes in one cluster); nodes in different clusters have different specifications. Each node has 2 identical GPUs.

Metrics. We use the following standard metrics:

Accuracy. This measures the top-1 cross-accuracy: the fraction of correct predictions among all the predictions using the *test* set; this shows the quality of the learned model over time.

Throughput. This quantifies the number of updates that the system processes per second. For deployments that employ multiple parameter servers, we report the highest throughput, which corresponds to the fastest correct machine.

Application. We consider three applications, all discussed in Section 3.5. In some experiments, we focus more on the setup with multiple servers and multiple workers (*MSMW*) as it gives the flexibility to test with a different number of Byzantine servers and workers. We consider two variants of *MSMW*: the first one uses Bulyan [72] to aggregate gradients and hence achieves Byzantine resilience in high dimensions while assuming network asynchrony. The second one uses *Multi–Krum* [60] to aggregate gradients while assuming network synchrony. Unless otherwise stated, we use our TensorFlow version with the first variant and PyTorch for the second one. We consider image classification due to its wide adoption as a benchmark for distributed ML systems [51, 1]. We use MNIST [152] and CIFAR-10 [141] datasets. MNIST is a dataset of handwritten digits with 70,000 28 × 28 images in 10 classes. CIFAR-10 consists of 60,000 32 × 32 colour images in 10 classes. Table 3.1 presents the models we use for evaluation.

Setup. Unless otherwise stated, we use the following *default* setup. For TensorFlow experiments, we employ 18 workers, including at most 3 faulty workers ($n_w = 18, f_w = 3$), and 6 servers, including at most 1 faulty server ($n_{ps} = 6, f_{ps} = 1$). Note that in *decentralized learning*

Table 3.1: Models used to evaluate GARFIELD.

Model	# parameters	Size (MBs)
MNIST_CNN	79510	0.3
CifarNet	1756426	6.7
Inception	5602874	21.4
ResNet50	23539850	89.8
ResNet200	62697610	239.2
VGG19	128807306	491.4

experiments, we do not use any servers (i.e., workers communicate in a peer–to–peer fashion). We employ a batch size of 32 for each worker, leading to an effective batch size of 480 in the normal case (i.e., counting only batches of honest workers). For PyTorch experiments, we use 10 workers, with 3 Byzantines, and 3 servers, with only 1 Byzantine. We use a batch size of 100 for each worker. We repeated all the experiments multiple times and found that the error bars are always very small and hence, omitted for better readability.

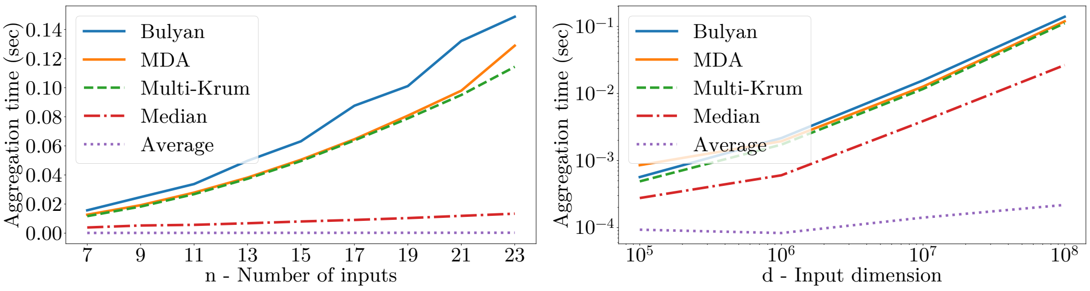
3.6.2 Baselines

To the best of our knowledge, GARFIELD is the first library to accommodate both Byzantine servers and workers. We chose the following baselines for our evaluation of GARFIELD.

Vanilla baseline. This is the vanilla deployment of TensorFlow or PyTorch. Such deployment fails to tolerate any Byzantine behavior whatsoever. We highlight two key differences between this deployment and GARFIELD–based deployments. First, this setup uses only one trusted parameter server, and hence, uses less number of communication links in the network and eliminates the overhead of synchronization required in the case of using multiple servers. Second, a lightweight GAR i.e., *Average* is used by such a baseline, where more computational intensive ones e.g., *Bulyan* are used by GARFIELD. Comparing GARFIELD to this baseline quantifies the overhead of Byzantine resilience.

AGGREGATHOR [60]. This is the only existing scalable ML system that achieves Byzantine resilience, yet only for Byzantine workers. It is built on top of TensorFlow and supports training only on CPUs. AGGREGATHOR uses one central, trusted server while tolerating Byzantine workers (i.e., uses *SSMW* setup), and it considers synchronous networks. For a fair comparison with our GARFIELD–based systems, we use the same GAR for both deployments. Thus, comparing with this baseline quantifies the overhead of using our object–oriented design and the communication layer we provide.

Crash–tolerant protocol. We implement a strawman approach to tolerate crash failures, assuming synchronous communication, using GARFIELD components. As worker crashes do



(a) Number of Inputs

(b) Input dimension

Figure 3.3: Microbenchmark of different GARs deployed on a GPU. n denotes the number of inputs to the GAR and d denotes the length of one input/gradient.

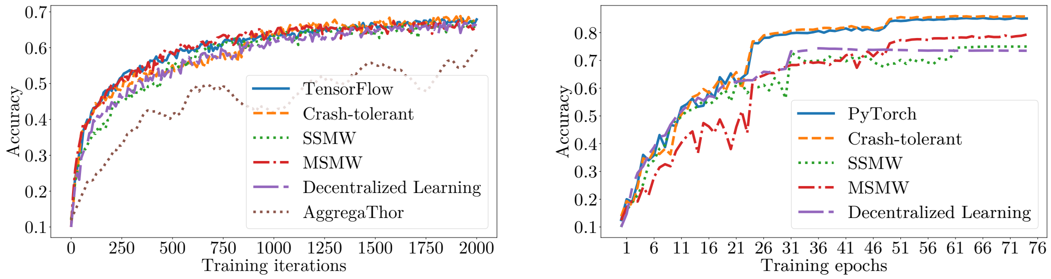
not affect the learning convergence eventually, we only tolerate server crashes by replicating the server. Server replicas get the updates from all workers and *average* them in each iteration, but workers contact only one of these replicas, i.e., the *primary*, to get the updated model. In the case of *primary* crash (signaled by a timeout), workers contact the *next* server, marking it as the new *primary*. The new primary sends its view of the model to all workers so as to inform them about the change. The model sent by the new primary could be outdated compared to the model of the crashed primary (due to missing some updates). We consider this as acceptable behavior, and learning can converge eventually [255], given that $n_{ps} \geq f_{ps} + 1$, where n_{ps} is the total number of replicas, and f_{ps} is the maximum number of crashing nodes, i.e., servers. Thus, this deployment guarantees eventual convergence without any guarantees on throughput or convergence rate. Some ML systems already use Paxos [149] for crash fault tolerance [51, 158]. However, our strawman algorithm, we believe, gives strictly weaker guarantees (in terms of consistency of model state among replicas) and hence has a higher throughput than Paxos.

3.6.3 GARs Microbenchmark

The Byzantine–resilient GARs are the basic enabling tools for Byzantine resilience. We provide a microbenchmark for their GPU–based implementation performance with respect to the number of inputs/gradients (i.e., n the number of workers or servers) and the gradient dimension (i.e., d).

Vanilla frameworks, e.g., TensorFlow, *average* gradients at the parameter server. So, we also include the evaluation of *Average*, which has been implemented as a part of the GARFIELD library. *Average* is our baseline in this experiment.

For a fair comparison, we set f, the number of Byzantine inputs, to $\lfloor \frac{n-3}{4} \rfloor$ (which is the bound for the strongest GAR we use, i.e., *Bulyan* [72]) for all Byzantine-resilient GARs and hence, the smallest possible n is 7. We set $d = 10^7$ in Figure 3.3a and $n = 17$ in Figure 3.3b. The metric for this microbenchmark is the *aggregation time*: it includes the aggregation of n input vectors



(a) Convergence with CifarNet

(b) Convergence with ResNet50 (1 epoch = 200 iterations)

Figure 3.4: Convergence of GARFIELD applications with respect to other baselines using two models.

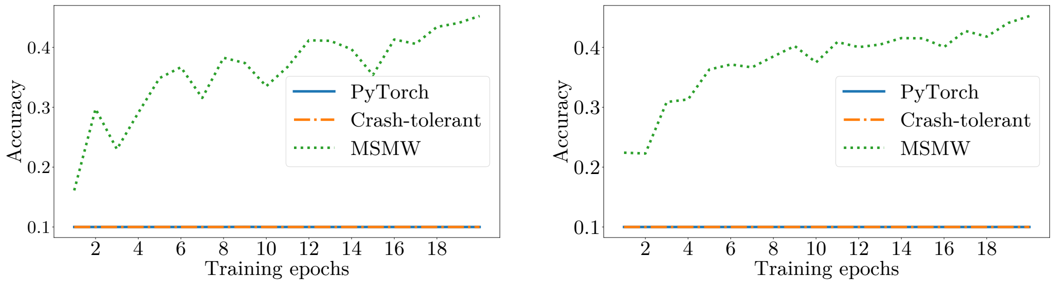
(all resident in GPU memory) and the transfer of the resulting vector back to main memory. Each point is the average of 21 runs, for which we observed a standard deviation two orders of magnitude below the observed average. We ran this microbenchmark on an *Intel Core i7-8700K* CPU and two *Nvidia GeForce 1080 Ti* GPUs.

Theoretically, the asymptotic complexities of *MDA*, *Multi–Krum*, *Bulyan*, and *Average* are respectively $\mathcal{O}\left(\binom{n}{f} + n^2 d\right)$ [72], $\mathcal{O}\left(n^2 d\right)$ [32], $\mathcal{O}\left(n^2 d\right)$ [72] and $\mathcal{O}(nd)$. Our implementation of *Median* has a best case complexity of $\mathcal{O}(nd)$ and worst case of $\mathcal{O}\left(n^2 d\right)$. In practice, for a fixed d (Figure 3.3a), we observe these asymptotic behaviors for *Multi–Krum* and *Bulyan*: quadratic in n. *Median* shows good scalability with n, maintaining a consistent performance that is very close to *Average*. Although the asymptotic complexity of *MDA* is exponential, our implementation achieves only a quadratic growth with n. The values of n and f used in these experiments are merely too low to expose such a behavior, i.e., the exponential growth with n. *Average* aggregation time remains roughly constant for a fixed d and $n < 15$, with an aggregation time of ~ 8 ms, and then grows linearly. For a fixed n (Figure 3.3b), we observe a linear time increases with respect to d for each one of the studied GARs.

3.6.4 Convergence Comparison

We inspect the convergence of GARFIELD and the baselines with training iterations. Figure 3.4 shows the results of two experiments: the first one (Figure 3.4a) trains CifarNet on TensorFlow–based systems (including AGGREGATHOR) using CPUs, where the second one (Figure 3.4b) trains ResNet50 on PyTorch–based systems using GPUs. Both experiments use CIFAR-10 as a dataset. The first experiment puts GARFIELD in a head–to–head comparison with the state–of–the–art Byzantine ML system, i.e., AGGREGATHOR. The second experiment is an instance of deployment of GARFIELD while training a bigger model using GPUs.

Figure 3.4a shows that all the systems achieve almost the same final accuracy (except AGGREGATHOR). Some of the Byzantine–resilient deployments converge a bit slower than those using *averaging* during training, yet reach the same accuracy eventually, i.e., after doing enough



(a) Random vectors

(b) Reversed vectors

Figure 3.5: GARFIELD tolerance to two Byzantine attacks.

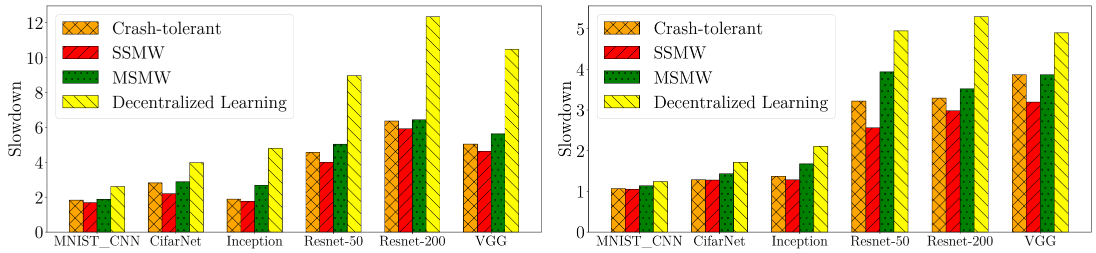
iterations. Interestingly, we can notice that Byzantine–resilient applications do not add much overhead compared to the crash–tolerant one in terms of the number of iterations till convergence (less than 1%). Surprisingly, GARFIELD applications achieve better final accuracy than AGGREGATHOR. We speculate that the reason is the fact that AGGREGATHOR relies on an old version of TensorFlow compared to GARFIELD (1.10 vs. 2.3)[10]. Another related reason is the fact that the latest version of TensorFlow is also integrated with the highly–optimized *Keras* library, which we also use. Figure 3.4b shows that Byzantine–resilient applications fail to reach the same final accuracy as vanilla PyTorch, with up to 10% final accuracy loss. This accuracy loss, although not clear in Figure 3.4a, is expected as a direct byproduct of using Byzantine–resilient GARs to aggregate workers' gradients and also due to having diverging servers (in some cases). Note that such GARs inevitably introduce accuracy loss as these robust aggregation protocols guarantee the convergence only to a ball around a local minimum [32]. The inability to optimize the model beyond this ball explains the observed accuracy loss. Notably, this behavior is seen after the stabilization of the accuracy (i.e., after convergence), and hence even if we let any of the Byzantine–resilient deployments run for more iterations, the accuracy will not likely jump up to the vanilla (i.e., non-Byzantine) deployment. Moreover, we notice that combining network asynchrony with decentralization leads to the biggest accuracy loss. Asynchrony essentially leads to the aggregation of outdated models and gradients, slowing down convergence and reducing the final accuracy. Interestingly, the crash–tolerant deployment does not experience such a loss compared to the vanilla case.

3.6.5 Tolerating Byzantine Failures

As a sanity check to our implementation, we conduct experiments with real adversarial behavior, where we apply attacks on the vanilla baseline (PyTorch in this experiment), a GARFIELD–based application (*MSMW* in this experiment), and the crash–tolerant protocol. Figure 3.5

---

[10] It might look unfair to use different versions of TensorFlow for comparison. Yet, note that almost all of the code base of AGGREGATHOR (which has thousands of LoC, written for V1.10) requires updating (to be compatible with V2.3). We chose to test with the latest publicly available code as-is while highlighting the different versions issue to make the context of our results clear to the reader.



(a) CPU (b) GPU

Figure 3.6: Slowdown of fault–tolerant systems normalized to the vanilla baseline (i.e., TensorFlow) throughput.

shows 2 attacks on both servers and workers. In the first attack (Figure 3.5a), the Byzantine node replaces its reply with random values, whereas, in the second attack (Figure 3.5b), such a reply is reversed and amplified (multiplied by -100). We train CifarNet with 11 workers and 3 servers (in the case of fault-tolerant algorithms) with 1 Byzantine node from each party. We run the training for 20 epochs. On the one hand, both the vanilla deployment and the crash–tolerant deployment fail to learn under both attacks. On the other hand, *MSMW* manages to train the model safely and converges to normal, high accuracy.

3.6.6 The Cost of Byzantine Resilience

We show here the computation and the communication costs of Byzantine resilience. We quantify the overhead of employing GARFIELD–based applications compared to the other baselines by measuring the throughput while training several models. Essentially, our goal is to understand the main factors that drive the cost of Byzantine resilience.[11] In this section and without loss of generality, we use ResNet50 as our model (unless otherwise stated). We do not employ any Byzantine behavior in these experiments as we want to quantify the overhead of Byzantine resilience in a normal, optimistic environment. Thus, we denote here the number of *declared* Byzantine nodes with the number of Byzantine nodes (i.e., f_w and f_{ps}).

Model dimension. Figure 3.6 depicts the cost of Byzantine resilience in terms of *slowdown*, which we define as the throughput of the fault-tolerant systems, *normalized to* that of the vanilla baseline in each case. The overhead of crash tolerance ranges from 83% to 537% on CPUs (7%–286% on GPUs), that of *SSMW* ranges from 69% to 492% on CPUs (5%–219% on GPUs), that of *MSMW* ranges from 88% to 544% on CPUs (14%–292% on GPUs), and that of *decentralized learning* ranges from 161% to 1135% on CPUs (24%–429% on GPUs) compared to the vanilla deployments. More interestingly, compared to the crash–tolerant deployment, *MSMW* overhead ranges from 1% to 42% on CPUs (0.1%–22% on GPUs) and

---

[11] Note that AGGREGATHOR's architecture is the same as *SSMW*. We have only included the results of the latter for better readability.

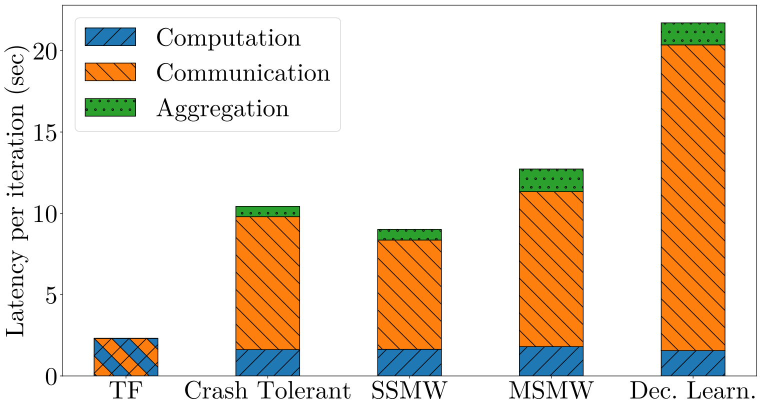


Figure 3.7: GARFIELD's overhead breakdown.

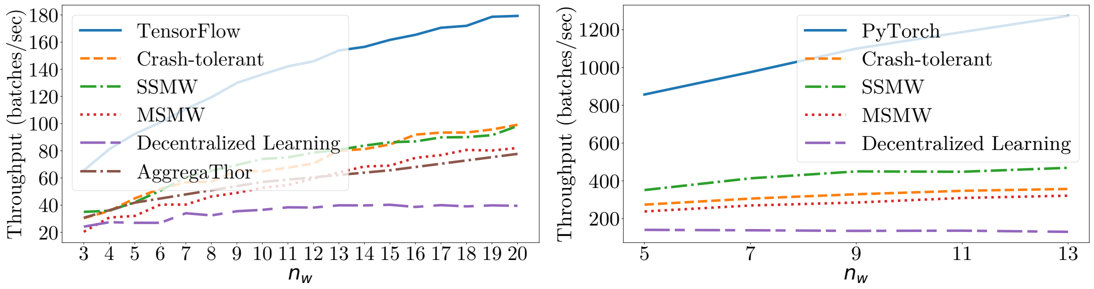
decentralized learning overhead ranges from 41% to 154% on CPUs (16%–61% on GPUs). It is evident that CPU–based deployments show higher slowdowns than that of the GPU–based ones. We attribute that behavior to two reasons: (1) we test with more machines in the first case, inducing higher communication overhead, and (2) GARs overhead is bigger with CPUs than with GPUs.

We extract several observations from Figure 3.6. First, the cost of Byzantine resilience, compared to vanilla baselines, is significant (reaches $\sim 12\times$ in the worst case). However, such a cost is expected (and might be even considered reasonable), noticing the cost of weaker alternatives (e.g., crash tolerance). Interestingly, the cost of workers' Byzantine resilience (using *SSMW*) is always less than that of crash tolerance (more clear with big models). Second, ML training, especially on GPUs, is network–bound: applications that require more communication have bigger slowdowns. Essentially, communication constitutes more than 75% of the overhead, whereas robust aggregation accounts for less than 11% (see Figure 3.7). Third, increasing the model dimension increases the overhead of Byzantine resilience, yet only until a certain point; after that point, the overhead remains roughly constant even with bigger models. The reason for that lies in the factors driving such an overhead. With small models, the cost of robust aggregation is visible (i.e., constitutes a non-negligible chunk of the overhead). Yet, with bigger models, the communication overhead prevails, which is in $\mathcal{O}(d)$ for all deployments.

Overhead breakdown. We pick one instance and take a closer look at all the deployments to understand the factors affecting their performance. Concretely, we run the same experiment while training ResNet50, breaking the average latency per iteration for each deployment. Figure 3.7 depicts the breakdown of the systems overhead when deployed on the CPU–based cluster. It is hard to decompose communication and the computation time for TensorFlow. Thus, the *blue-and-orange* bar denotes the time spent in both of them combined.

We can observe in the figure that the computation time is roughly the same for all applications ($\sim$ 1.6s). Yet, the communication cost dominates the overhead (ranges from 75% to 86%).

This makes (1) crash tolerance costly more than Byzantine workers' tolerance (22% extra communication), and (2) Byzantine servers' tolerance costly more than only workers' tolerance (42% more communication). Furthermore, we note that the aggregation time in *decentralized learning* is two times bigger than that of *SSMW* due to the extra model aggregation step done by the former application.



(a) CPU TF-version

(b) GPU PT-version

Figure 3.8: The scalability of GARFIELD with increasing n_w.

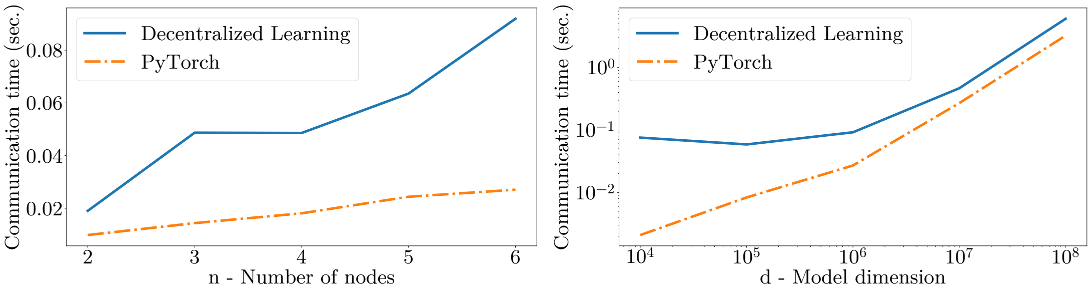
3.6.7 Scalability of GARFIELD

In this section, we analyze the scalability of multiple training applications with a different number of workers, Byzantine workers, and Byzantine servers.

Number of workers. Increasing the number of workers (n_w), and hence increasing the *effective* batch size, is crucial for scaling distributed ML applications. Figure 3.8 depicts the scalability of GARFIELD–based applications while training CifarNet on CPUs (Figure 3.8a) and ResNet50 on GPUs (Figure 3.8b). In this figure, throughput is measured in *batches/sec* rather than *updates/sec* since employing more workers allows for increasing the number of batches processed per iteration.

Figure 3.8a shows that *SSMW* outperforms AGGREGATHOR. This happens arguably due to the optimizations we include in GARFIELD in addition to using a newer version of TensorFlow. We draw three main observations from Figure 3.8. First, all systems scale by employing more workers (except the *decentralized learning* application), with around one order of magnitude higher throughput with GPUs compared to CPUs. Second, the throughput gap between the vanilla deployments and the fault-tolerant deployments increases with increasing n_w, keeping the slowdown introduced by *SSMW*, *MSMW*, and crash–tolerant almost constant. Third, the scalability of *MSMW* is almost as good as that of the crash–tolerant deployment and the difference in throughput with increasing n_w is almost constant. This shows that *complete* Byzantine resilience does not harm scalability compared to crash resilience.

To understand why *decentralized learning* does not scale, we focus on its communication overhead with different number of nodes (n) and model dimension (d). Figure 3.9 shows

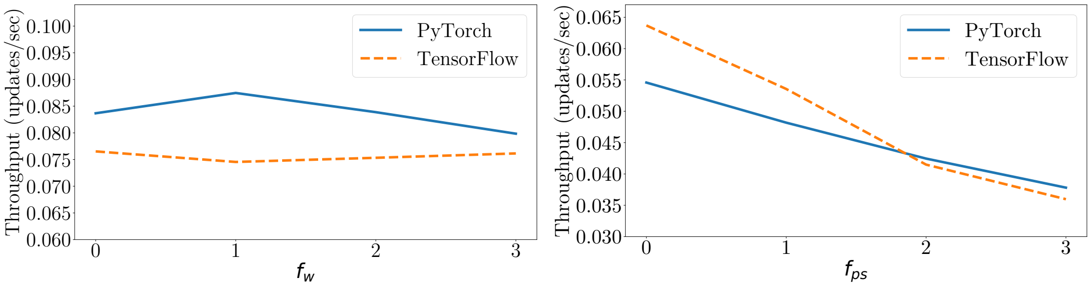


(a) Number of Inputs

(b) Input dimension

Figure 3.9: Communication time of *decentralized learning* and vanilla baseline (deployed on GPUs) with n and d.

the communication latency of *decentralized learning* and vanilla baseline (PyTorch in this experiment) with different values of n (with $d = 10^6$) and d (with $n = 6$). It is evident that increasing d saturates the bandwidth quickly and increase the communication time linearly for both systems (Figure 3.9b). Yet, the scalability issue of *decentralized learning* appears in Figure 3.9a, where the communication time increases drastically (i.e., quadratically) for *decentralized learning* while only linearly with the vanilla competitor. Basically, *decentralized learning* requires $\mathcal{O}(n^2)$ messages per round while the vanilla deployments require only $\mathcal{O}(n)$ messages per round. Note that the *all-to-all* communication in *decentralized learning* is crucial to achieve Byzantine resilience.



(a) Number of Byzantine workers

(b) Number of Byzantine servers

Figure 3.10: The scalability of GARFIELD with f_w and f_{ps}.

Number of Byzantine workers. In general, practitioners deploy multiple workers to distribute the heavy load of training to multiple machines (rather than for fault tolerance). In this sense, increasing the number of Byzantine workers (f_w) does not call for increasing the total number of workers (n_w), as long as the inequality relates f_w to n_w (according to the used Byzantine-resilient algorithm) holds. In this experiment (in Figure 3.10a), we fix n_w and hence, fixing the effective batch size in all cases. Fixing n_w results in a fixed communication cost in all cases, making the throughput almost the same even with increasing f_w. The same results are confirmed with both frameworks, with a slight superiority of PyTorch to TensorFlow.

Number of Byzantine servers. We replicate the server in the first place to tolerate Byzantine servers. Therefore, increasing the number of Byzantine servers (f_{ps}) calls for increasing the number of server replicas (n_{ps}) so as to satisfy the Byzantine resilience condition: $n_{ps} \geq 3f_{ps}+1$. Thus, increasing f_{ps} introduces new communication links, leading to a throughput drop as shown in Figure 3.10b. Such a drop is confirmed in the *state machine replication* (SMR) literature [114, 3], where the amount of drop (less than 50%) is not significant compared to what was reported before in the literature [57]. The assumption of 1 faulty parameter server introduces an overhead of 33% to achieve Byzantine resilience. An interesting fact here is that increasing f_{ps} does not increase the number of iterations required for convergence since the selected number of gradients with which GARFIELD does the update is constant anyway.

3.7 Related Work

To the best of our knowledge, AGGREGATHOR [60] is the only implementation of a Byzantine ML system that is prior to our work. AGGREGATHOR relies on two components: the aggregation layer, which uses *Multi–Krum* to robustly aggregate workers' gradients, and the communication layer, which enables experimenting with lossy networks. Though AGGREGATHOR follows the shared graph design, it disallows workers to change such a graph to combat any possible Byzantine behavior.

The design of GARFIELD is fundamentally different from that of AGGREGATHOR: while the latter is merely a layer on top of TensorFlow, GARFIELD is a standalone library that can be plugged into different frameworks. Indeed, AGGREGATHOR supports only one architecture, namely using a single, trusted server with multiple workers in synchronous environments. From those perspectives, it is not very robust. GARFIELD, on the other hand, can flexibly adapt to different scenarios, e.g., with multiple server replicas and asynchronous environments.

On the theoretical side, several Byzantine-resilient ML algorithms have been proposed; all try to mathematically bound the deviation of the aggregated gradient from the correct ones. Krum [32] employs a median-like aggregation rule. Multi-Krum [60] generalizes the idea by averaging more gradients to benefit from additional workers. Bulyan [72] addresses an attack that can trick some Byzantine-resilient algorithms by having them converge to a stable yet faulty state. Different variants of robust mean-based algorithms under different assumptions and scenarios were considered in [252, 270]. Kardam [61] uses filtering mechanisms to achieve Byzantine resilience in an asynchronous training setup. Zeno [253] and Zeno++ [258] achieve Byzantine resilience using a performance-based ranking approach in synchronous and asynchronous settings respectively. Draco [47] uses a coding scheme to restore correct gradients using redundant computations. Detox [207] extends this idea by combining coding schemes with robust aggregation to hit the sweet spot in the resilience–optimality spectrum. ByzSGD [70, 71] shows how to combine robust GARs to tolerate Byzantine servers too. In particular, it replicates the parameter server on multiple machines while letting them communicate to limit the divergence among their model states. ByRDiE [268] and BRIDGE [267] combine robust aggregation with performance-based ranking to achieve Byzantine resilience in the decentralized settings.

While such proposals shaped the literature of Byzantine-resilient ML, they only remain theoretical (i.e., without a deeper look at the very practical costs of such solutions). GARFIELD closes this gap by providing a tool to practically build these solutions. GARFIELD uses robust aggregation and already implements many of the mentioned GARs. GARFIELD can straightforwardly include the other ones with minimal coding effort.

The problem of tolerating benign (i.e., crash) failures of parameter vectors was also addressed in the literature. Qiao et al. [202] leverage the self-correcting behavior of SGD to tolerate such failures. Other proposals addressed the problem of making the parameter server crash-resilient [158, 51] using Paxos [149]. Others rely on checkpoints or live replication [1] of the parameter server. However, we believe that extending those tools to the Byzantine context would be prohibitive.

3.8 Concluding Remarks

This chapter presented GARFIELD, a library to build Byzantine machine learning (ML) applications on top of popular frameworks such as TensorFlow and PyTorch while achieving *transparency*: applications developed with either framework do not need to change their interfaces to be made Byzantine-resilient. GARFIELD supports multiple statistically-robust gradient aggregation rules (GARs), which can be combined in various ways for different resilience properties. In some situations, GARs fail to ensure Byzantine resilience when the underlying assumption on a bounded variance is not satisfied [25]. Yet, several techniques were proposed for variance reduction, e.g., [243, 12, 192], which help restore the resilience guarantees of such GARs. Such techniques can be added seamlessly to GARFIELD without affecting its throughput performance. In the same vein, we believe GARFIELD could be also used to implement applications that combine privacy and security properties with Byzantine resilience as in [106, 181]. Indeed, any protocol that relies on exchanging replies and aggregating them in some robust manner, e.g., [77], can be implemented with GARFIELD. Our code is open-sourced and available at [96]. Our evaluation of GARFIELD (using three Byzantine ML applications) showed that Byzantine resilience, unlike crash resilience, induces an inherent loss in the final accuracy and that the throughput overhead of Byzantine resilience is moderate compared to crash resilience. Furthermore, we showed that (1) the Byzantine resilience overhead comes more from communication than from aggregation and that (2) the overhead of tolerating Byzantine servers is much more than that of tolerating Byzantine workers.

Practical Tips for Efficient Training Part III

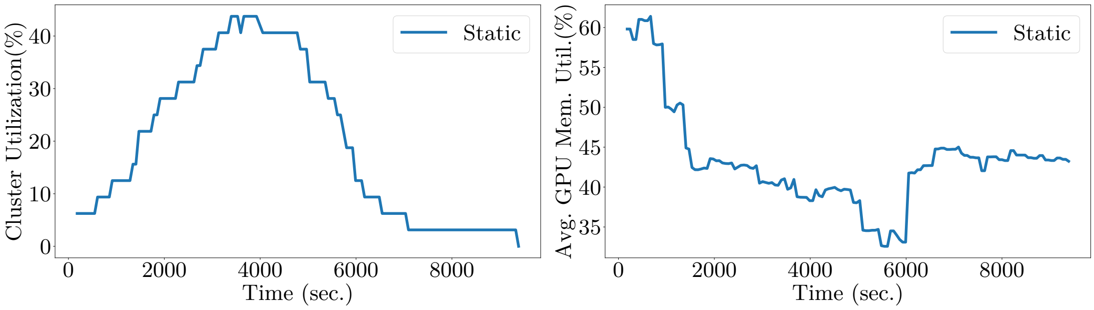
4 Elasticity is All You Need

In this chapter, we consider training in a datacenter, where typically, we have various training tasks (with heterogeneous workloads) that need to run on a shared cluster of limited resources. We resolve the fierce resource contention with *elasticity*: giving training tasks different amounts of resources throughout their lifetime. We present an Elastic Resource Allocator (ERA) for optimizing resource utilization on shared deep learning clusters. ERA is elastic in two dimensions: (1) it scales jobs up and down, i.e., by adding or removing resources to or from the running jobs, and (2) it dynamically changes the per–GPU batch size, at will, to control the GPU utilization. The goal of ERA is to (a) maximize the number of concurrently running jobs and (b) to achieve good job–level and cluster–level performance, such as maximizing the average throughput and minimizing the average job completion time.

4.1 Introduction

We consider the training of deep learning (DL) applications in a datacenter. Training DL tasks (also called jobs) demands a significant amount of computing, memory, storage, and network resources [88, 125]. Nowadays, datacenters serve these tasks, including long-running training tasks [123], on shared clusters of a limited number of resources.

In the past few years, the research community has explored different strategies for resource allocation, aiming for high cluster utilization [251], fairness [90], and overall users' happiness [196]. Such proposals assume that the requested resources are fixed throughout the training. This rigid approach is a road-blocker against optimal performance and resource utilization. On the one hand, having many users' requests leads to longer queuing delays. On the other hand, having a few of them leads to resource underutilization. For machine learning practitioners, it is not always easy to anticipate the optimal number of machines and the best batch size [26]. In addition, the best parameters for resource utilization typically depend on the allocated hardware in the cluster, which the users do not know in advance. To capture this problem, we conduct the following experiment (see Figure 4.1). We measure the cluster utilization (i.e., the percentage of active GPUs to the total number of available



(a) Cluster utilization

(b) GPU memory utilization

Figure 4.1: Cluster and resource utilization with a non–elastic solution. We submit 15 training tasks to a 64–GPU cluster with a mean arrival time of 4 minutes, following the Poisson distribution.

GPUs in the cluster) and the average GPU memory utilization (as reported by *nvidia-smi*[1]) while submitting 15 training tasks to a 64–GPU cluster. Tasks submission schedule follows the Poisson distribution with a mean arrival time of 4 minutes. Figure 4.1a shows that the cluster utilization is upper–bounded by 45%, and Figure 4.1b shows that the average GPU memory utilization peak is only 60%.

Our key to resolving this underutilization is to allow DL training jobs to be elastic, i.e., assign a different number of machines to training tasks throughout training time. Such elasticity of DL training has been recently examined, e.g., in [190, 115]. Though these approaches achieve some gains, they are limited in a few aspects. For example, Autoscaler [190] focuses only on scaling a single job without looking at cluster utilization or maximizing the number of concurrently running jobs. AFS-L and AFS-P (described in [115]) ignore the utilization level of the individual GPUs when making scaling decisions. Moreover, AFS-L assumes the knowledge of the remaining time for job completion, which might not be practical in some cases. Furthermore, all these approaches apply only horizontal scaling, ignoring scaling the *per–machine* batch size (also called *local* batch size), which we found crucial for optimal performance.

We present ERA: an *Elastic Resource Allocator* for multi-tenant DL clusters. ERA provides elasticity in two dimensions: the number of machines (also called workers) contributing to training and the per-machine batch size, hereafter, batch size. ERA scales up jobs that benefit from having more resources and scales down jobs to maximize the number of concurrently running jobs on the cluster while being fair to all jobs. Interestingly, ERA does not require any prior information on the workload or the remaining time for jobs to terminate. Furthermore, ERA detects the underutilization of GPU processing cycles and memory and scales the batch size accordingly. Thus, a user does not need to adjust their training script based on the target hardware as ERA takes care of that seamlessly. Our insight here is that scaling the batch size improves the computation efficiency (by maximizing the utilization of the GPUs) and

---

[1] https://developer.nvidia.com/nvidia-system-management-interface

hence, mitigates the communication overhead resulting from scaling; this opens the door for even more *rewarding* horizontal scaling and hence, improved performance. Notably, changing the batch size might alter the convergence behavior [271]. This fact forced some *elastic* solutions to keep the *global* batch size fixed while scaling [22]. To mitigate this, we take two design decisions. First, we allow users to specify a maximum *global* batch size that ERA can use for their jobs. Second, we scale the learning rate linearly should the *global batch size* (i.e., the aggregate batch size used by all workers) change to preserve the noise scale and the convergence behavior [226, 219]. ERA takes decisions dynamically in the middle of training, i.e., not at fixed events such as job arrival or departure.

ERA works as follows. ERA periodically polls the running jobs for the latest application–level and cluster–level measurements. Meanwhile, ERA monitors the incoming requests that need to be scheduled on the cluster. Given these inputs, ERA finds the optimal share for each job. Specifically, ERA solves an integer non-linear optimization problem (INLP), seeking the optimal resource allocation plan. ERA aims at maximizing (a) the number of parallel running jobs, (b) the average throughput of all jobs, and (c) the cluster utilization (i.e., minimizing the number of idle machines) while minimizing the overhead of frequent scaling (to avoid thrashing [251]). We address this *non-linear* problem using a substitution trick: we introduce an auxiliary decision variable and two additional constraints to address the non-linearity of the problem. In addition, ERA follows a thresholding technique to increase the batch size should it detect an individual GPU underutilization. ERA also detects out-of-memory (OOM) errors and adapts the batch size accordingly in this case.

We implemented ERA as a Kubernetes (K8s) operator [277] to be seamlessly compatible with K8s [41], a widely-used container orchestration platform. We use *elasticJob* [200] to encapsulate users' requests, which should be written with PyTorch [193]. We use PyTorch-elastic [199] and its K8s controller to apply the elasticity decisions of ERA. We use *ConfigMaps* [144] as a side-channel between ERA and the running jobs to communicate the batch size change events.

We evaluate ERA on a shared 64–GPU cluster. Our training tasks span two domains: image classification and language modeling. In the first domain, we train various models with ImageNet [66] while we fine-tune BERT [132] in the second one. We compare ERA to a vanilla baseline with no elasticity (denoted as *Static*) in addition to two state-of-the-art systems that use elasticity: Autoscaler and AFS-P.

We show that ERA reduces the average job completion time (JCT) by 2.5×, 2.1×, and 2× compared to *Static*, Autoscaler, and AFS-P, respectively, with up to 60 jobs. In the same vein, ERA reduces the makespan by 1.9×, 1.56×, and 1.76× compared to *Static*, Autoscaler, and AFS-P baselines, respectively. In terms of throughput, jobs managed by ERA can process, on average, around 3000 samples/second, whereas AFS-P, Autoscaler, and *Static* can process only 1500, 600, and 500 samples/second, respectively. ERA also boosts the average GPU compute and memory utilization by up to 25%. Finally, we show that the scaling decisions of ERA do not affect the convergence behavior compared to *Static*: jobs managed by ERA can converge, to the same accuracy, faster than Static.

Our experiments also give further interesting observations on elasticity. For example, we observe that elastic systems can converge to the same JCT regardless of the initial number of starting workers (i.e., if the user is not an expert, elastic systems can choose the best number of machines to use for them). Furthermore, we noticed that horizontal scaling alone might lead to *worse* GPU utilization (even compared to *Static*) because the communication overhead (results from scaling up) could outweigh the benefit of the scaling. Notably, our two-dimensional (2D) scaling does not fall into this trap.

To summarize, our contribution is three-fold. 1. We design ERA, a 2D scaling algorithm through which we show that scaling the batch size is crucial for optimal performance. 2. We integrate ERA with K8s, a state-of-the-art container orchestration platform. 3. We show, using real-world scenarios, that ERA outperforms the state-of-the-art elastic systems.

4.2 Background

4.2.1 Elastic Distributed Learning

As given in Section 2.3, data distribution, as well as training, is either orchestrated by a central server [156] or in a decentralized manner [195]. We use the latter approach, in which case the algorithm works as follows. Each machine contributing to training (also called a worker) possesses a local model copy and partial data. Each worker first estimates a gradient using a data batch, which is then used to update the model. As we consider synchronous training, workers synchronize their models after each iteration. Essentially, workers aggregate their updated models (from each other), typically by averaging them, to obtain the new model.

The summation of the sizes of the batches used by all workers is called the *global batch size*. Increasing the number of workers decreases the processing load for each of them. The question now is: should the global batch size change with changing the number of workers? On the one hand, fixing the global batch size (by changing the *local* batch size linearly) ensures the same convergence behavior regardless of the number of workers and thus, promotes reproducibility [22]. On the other hand, changing the global batch size (by fixing the *local* batch size) preserves resource utilization but might affect the convergence behavior [219]. To mitigate this probable change in convergence, some approaches suggested scaling the learning rate with the batch size [226, 129].

4.2.2 Deep Learning Scheduling

There has been increasing attention recently to scheduling strategies for DL workloads in shared clusters. The problem statement is as follows. Consider a fixed set of resources and multiple training requests. The goal is to assign the resources to the requests while minimizing the average job competition time (JCT). Previous approaches examined different strategies such as prioritizing short jobs [218], prioritizing efficient jobs [115], or prioritizing jobs with the lowest memory fragmentation [279]. Scheduling techniques also included time-slicing of resources [251], task migration [90], packing tasks on GPUs [279], and load balancing [196].

4.2.3 PyTorch–elastic and ElasticJob

PyTorch–elastic [199], an engine offered by PyTorch [193], is elastic in the sense that it allows distributed applications to use a varying number of workers throughout training. It also helps tolerate worker crashes and network omissions. PyTorch–elastic stores the state of the running jobs in a persistent key-value store, e.g., *etcd*. Before starting training, workers elect a master for the job. The master address is then stored in *etcd* so that it can be found later by other workers joining the job. Workers then try to connect to each other. If this step fails within a pre-defined time window, workers elect another master and start over. This process is called *rendezvous*. In the case of failure or scaling (up or down), workers restart and initiate a new *rendezvous*. It is then the responsibility of the training script to checkpoint the training progress regularly to avoid losing it in the event of failure or scaling.

ElasticJob [200] is a Kubernetes *custom resource*[2] for elasticity. Users can use elasticJob to describe their training task and then adjust the resources allocated to it. Precisely, elasticJob provides an interface for users to define the main elasticity characteristics of their job such as a range (min, max) on the number of workers. ElasticJob also exposes *replicaSpecs*, which describes the specifications of each worker encapsulated in a container. For instance, within *replicaSpecs*, the user can define limits on the number of CPU cores and the memory required for their job and can also pass custom environment variables to the training task.

4.3 The ERA Design

4.3.1 Overview

ERA dynamically allocates resources to DL tasks in shared clusters. Due to its wide adoption in DL training platforms, we integrate ERA with Kubernetes (K8s) [41], a container orchestration platform that can efficiently manage multiple applications in shared clusters. We envision that the cluster operator would run ERA with the goal of improving the cluster utilization and the users' experience (by allowing their workloads to run faster).

In this chapter, we only consider GPUs while solving the resource allocation problem. The main ideas of ERA can also be applied to other resources such as CPU cores and memory. For simplicity, we assume homogeneous clusters: all GPUs in the cluster have the same specifications, and the network latency is also homogeneous for all machines. In addition, we focus on synchronous training. ERA does not require any prior information about the workloads (i.e., datasets and models used), the convergence behavior, or the training hyperparameters. ERA

---

[2] https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

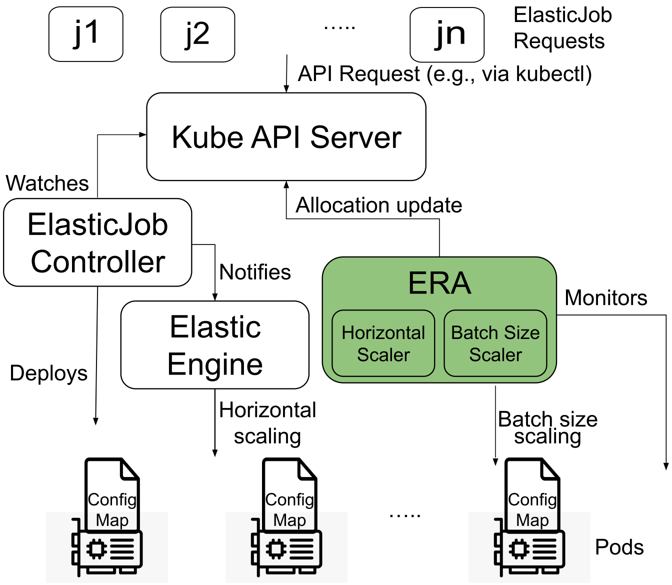


Figure 4.2: The resource allocation architecture. The ERA component, highlighted in green, is our main contribution.

also does not require an estimation of the remaining time for job completion.

ERA controls jobs by adjusting their specifications dynamically while respecting the minimum and maximum values provided by the users. ERA communicates with each job through a K8s *ConfigMap*, initially created by the user. Each ConfigMap includes the batch size, the maximum allowed batch size, and a timestamp of the latest modification to the batch size event. Each worker then has to mount a ConfigMap to its volume.

4.3.2 Architecture

Figure 4.2 depicts the architecture of our resource allocation system. At a high level, users submit their requests by creating *elasticJobs*, which eventually run on one or more pods.[3] In this section, we explain the main components shown in Figure 4.2.

Kube API server. The Kube API server is the entry point to any request submitted to the K8s cluster. In addition to serving users' requests, the API server also serves in-cluster requests, which we describe later. The server typically receives an elasticJob request, parses it, and stores a *job entry* in an *etcd* server. When requested, the API server sends the stored *job entry* to the *elasticJob controller* to be deployed. The API server also replies to ERA's requests about the cluster state, the number of waiting jobs to be scheduled, jobs' metadata, and jobs' states

---

[3] A pod is a unit of computation in Kubernetes. In our case, 1 pod is equivalent to 1 GPU. Furthermore, 1 worker is 1 GPU.

(i.e., running, succeeded, or failed). In addition, when ERA takes a scaling decision, it sends a request to the API server, which serves such a request, passing it to the appropriate entity.

ElasticJob controller. We use a K8s controller that is provided by PyTorch–elastic [200], which is responsible for managing the elasticJobs that are running in the cluster. This entails creating a K8s pod for each worker, which in turn hosts a container with the corresponding specifications requested by the user. In the case of a scaling request, the controller communicates the new setup to the elastic engine in order to apply the requested scaling decision. For instance, in the case of scaling up, the controller creates a new pod (with the requested specifications) and makes it available to the elastic engine to add the new worker. The controller also monitors the state of the running pods and changes such a state when applicable.

ERA. This component is the main contribution of our system. ERA is responsible for two main things: (1) monitoring the running jobs for application-level and cluster-level measurements, and (2) scaling jobs in two dimensions: the number of running workers (i.e., horizontal scaling) and the per–worker batch size. Scaling in both dimensions is orthogonal, i.e., one job can scale horizontally without scaling the batch size and vice versa. Note that scaling in either dimension scales the *global* batch size, unlike recent approaches, e.g., [22]. To keep the same convergence behavior expected by the user, ERA scales the learning rate with either scaling type. We show in Section 4.5 that this reaction practically preserves convergence, following the theoretical guarantees [226] given by that approach as well.

Regarding the first task, ERA periodically monitors (a) the cluster for cluster-specific measurements such as the number of *active* and *idle* GPUs and (b) the pods associated with each job for application-specific measurements such as training throughput or the number of executed training steps. Each application-specific measurement is averaged[4] over all the pods associated with the job. ERA uses these measurements to take scaling decisions, as we describe later. Notably, ERA does not run a synchronized clock for all jobs. Hence, the measurement time could differ from one job to another. For fairness, the period between measurements (i.e., the frequency of taking measurements) is fixed to a constant across all jobs.

For its second task (i.e., taking scaling decisions), ERA comprises two components, one for each dimension of scaling. The first one is the *horizontal scaler.* ERA scales the number of workers based on three inputs: throughput, cluster utilization, and scaling overhead. To take these factors into account and to find the optimal allocation plan, ERA solves the following optimization problem:

---

[4] Median would be more robust in heterogeneous clusters.

$$\begin{aligned}
\max \quad & \sum_{j\in\mathscr{J}} e_j(n_j)\times\Delta_j-\gamma_j\times\left|\Delta_j\right| \\
\text{s.t.} \quad & \forall_{j\in\mathscr{J}} \quad min_j \le n_j+\Delta_j \le max_j \\
& \sum_{j\in\mathscr{J}} \quad n_j+\Delta_j=\min\left(G_{total},\sum_{j\in\mathscr{J}} max_j\right) \\
& \forall_{j\in\mathscr{J}} \quad \Delta_j\in\mathbb{Z},
\end{aligned} \tag{4.1}$$

where $\mathscr{J}$ is the set of all running jobs, e_j is the scaling efficiency score (see Equation 4.2 below) of job j, Δ_j is the change in the number of pods (i.e., the decision variables of the optimization problem), γ_j is the penalty for scaling job j (up or down), typically to reduce the scaling overhead, min_j and max_j are the minimum and the maximum number of workers requested by the user for their job j, n_j is the current number of pods assigned to job j, and G_{total} is the total number of GPUs available in the cluster.

The scaling efficiency term ($e_j(n_j)$) describes how well a job would benefit from additional resources. It is defined as follows:

$$e_j(n_j)=\frac{t_j(n_j)}{n_j\times t_j(1)}, \tag{4.2}$$

where $t_j(n_j)$ is the expected throughput of job j with n_j workers. It is important to note that the function t_j differs from one job to another. In other words, the scaling efficiency in our design is a job–specific metric, where each job has its own function for scaling efficiency. To get a robust throughput measurement, ERA takes the previous measurements into account to compute the function t_j. Precisely, ERA builds a linear model using the history of throughput measurements and then uses that model to estimate throughput with any value for n_j. Notably, our model takes all previous throughput measurements, even without excluding the old measurements. Our experiments show that this linear scaling approach, though simple, is very accurate practically for throughput estimation. Note that ERA prioritizes efficient jobs, which might come at the expense of *fairness*. Typically, inefficient jobs are given fewer resources than the efficient ones, according to Equation 4.1. Yet, this decision promotes cluster utilization and overall performance in terms of throughput and job completion time. Our rationale is that training workloads are not similar in their characteristics, and hence giving a fair share to each job would lead to suboptimal performance.

Scaling jobs too often can lead to *thrashing* [251, 115], in which a job cannot readily progress due to frequent stopping for scaling. To reduce the probability of facing this issue, we penalize each job with the term γ_j in the objective function. Not surprisingly, γ_j is a function of the number of times of scaling j (up or down) so far, which we denote by s_j. To put this number in context, we compare it to the total number of scaling rounds that took place (denoted by s). Precisely, we have $\gamma_j=\frac{s_j}{s}$.

Our optimization problem (Equation 4.1) is non-linear due to the absolute value in the objective function. This non-linearity makes the problem intractable. We follow a substitution trick to make it *tractable*. Specifically, we replace $|\Delta_j|$ with a new variable y_j in the objective function, and we add the following constraints:

$$\begin{aligned} &\forall_{j \in \mathscr{J}} \quad y_j \geq \Delta_j, \\ &\forall_{j \in \mathscr{J}} \quad y_j \geq -\Delta_j. \end{aligned} \tag{4.3}$$

Both constraints ensure that $y_j = |\Delta_j|$. We use the *PuLP* [178] to solve this *linear* problem.

The solution to the optimization problem results in the *optimal* values for Δ_j, which denote the optimal changes in the number of workers for each job j. To apply the changes, ERA sends an *allocation update* to the API server, which in turn asks the elasticJob controller to update the cluster state. Note that ERA solves this optimization problem periodically. The operator determines the period between decisions. In our experiments, we solve this optimization problem (and hence take a scaling decision) every 4 minutes.

The second scaling component of ERA is the *batch size scaler*. Scaling in this dimension is performed independently for each job because there is no competition between jobs on resources.[5] To take a scheduling decision, ERA compares the individual resource utilization (i.e., the GPU utilization and the GPU memory utilization) to pre-defined thresholds. If the utilization is less than the threshold, ERA scales up the batch size (specifically by doubling it), passing the new value to the application through the corresponding *ConfigMap*. If the new batch size leads to an out-of-memory (OOM) error, the application reduces the allowed maximum batch size and notifies ERA to prevent that situation from happening again. Given our algorithm, the minimum batch size that can be used is the one requested by the user in the original configuration.

Notably, choosing the thresholds is a crucial step in our algorithm. Such choices should depend on the expected workloads and the hardware. Hence, we envision these thresholds to be chosen by the operator. In our experiments, we chose the threshold values based on some experiments to strike a tradeoff between resource utilization and scaling efficiency (given that there are some workloads that will never be able to hit 100% utilization even with the maximum possible batch size).

Using the *correct* ranges for the batch size and the number of workers is crucial. Users set these ranges based on the application's nature and behavior (like typical ML hyperparameters). If a user chooses excessively high or low values for the batch size, the model might get stuck into a bad local minimum or might take a very long time to converge. Similarly, if a user chooses a very high value for the minimum number of workers, the application might get stuck in the queue due to the scarcity of resources. Based on that, these parameters should not affect the

---

[5] Note that ERA currently does not support packing multiple jobs on the same GPU. In the future, if this option is allowed, scaling the batch size of one job can affect the performance of another job (if both jobs run on the same GPU).

benefits of ERA as long as the user chooses reasonable ranges (i.e., not adversarial ranges). In such a case, ERA can then dynamically set the optimal parameters for the user's job to achieve the best benefit.

The elastic engine (PyTorch–elastic). When ERA decides to scale a job, it notifies the elastic engine (passing by the Kube API server and the elasticJob controller) of the change to take appropriate steps to implement it. For simplicity, we discuss only a scale-up event; similar steps are taken in scale-down events. In a scale-up event, the engine first reads the current state of the job from the *etcd* server and then adds the additional pods (prepared previously by the controller) to the set of pods assigned to that job. The updated set of workers (hosted on the pods) then initiates a *rendezvous* to find each other (see Section 4.2.3). After concluding successfully, the engine commits the new state to *etcd*.

Applications. To benefit from ERA, users should include three key components in their training scripts. First, they have to periodically checkpoint their progress to avoid losing it during scaling events. The checkpointing frequency controls a resilience-overhead tradeoff: increasing it improves the tolerance to failures yet also increases the overhead. Second, applications have to take periodic measurements and make them available to ERA. The frequency of logging is left to the user to decide. Third, applications should periodically read the latest batch size from their ConfigMaps and change it accordingly. We allow users to set an upper bound for the batch size to ensure specific convergence behavior.

4.4 The ERA Implementation

In this section, we give some details about our implementation of ERA. As we use PyTorch–elastic and its K8s controller [200], we consider applications written in PyTorch.

4.4.1 The Operator

We implemented ERA as a K8s operator [277]. The operator comprises (a) two listeners for jobs creation and deletion, (b) a background daemon for jobs monitoring, and (c) two handlers for initialization and cleaning up. When the operator starts, it loads the cluster configuration and initiates a background thread to take scaling decisions.

When a job starts, ERA collects the necessary state of that job, as we discuss in Section 4.4.2. On deletion, ERA removes the job from its internal state so as not to be considered in the upcoming scaling events. In both events, ERA updates the state of the cluster, noting down the number of idle GPUs. Furthermore, ERA periodically checks for finished jobs, in which case it releases them from its state.

ERA monitors jobs via separate *asyncio events* in a daemon. We do this asynchronously for scalability. ERA periodically polls the pods assigned to each job to read the latest measurements. Two notes here. First, to read the pod logs, ERA sends a request to the API server, which serves the request and reads the logs. Second, ERA stores the time of the last measurement to avoid redundant readings in subsequent iterations, reducing the reading overhead significantly.

4.4.2 State Management

We describe here the state that ERA saves to help take scaling decisions. At a high level, ERA stores the total number of available GPUs, the number of idle GPUs, and some information about the running and the incoming not–yet–scheduled jobs. For each job, ERA stores its specifications, its measurements, and previous scaling decisions. This latter metric is crucial for computing the scaling overhead, which we consider in our optimization problem, i.e., Equation 4.1.

ERA optimizes the storage of measurements as follows. For each job, ERA averages[6] the measurements of pods assigned to it. In particular, ERA stores the throughput, training progress, GPU utilization, and GPU memory utilization. ERA keeps only the freshest measurement for each reported global batch size. This is crucial to get an up-to-date throughput expectation, as we discuss later.

In order to save memory and to be able to predict throughput (see Equation 4.2), ERA builds a model using the throughput measurement history. Essentially, if ERA detects that only the number of workers has changed so far, it builds a linear model[7] with one independent variable (i.e., the number of workers). If the batch size has also changed, ERA fits a linear model with two variables instead. Using either model, ERA can compute the scaling efficiency (Equation 4.2) and generally predict throughput with any number of workers.

4.4.3 Garbage Collection

Some jobs might end up in *Error* state or *partial Error* state, i.e., with some pods failing while the rest are running. Such jobs are considered *"Running"* by Kubernetes; this might trick ERA to give pods to these failing jobs while they are not running, wasting the cluster resources.

To mitigate this issue, ERA runs periodic garbage collection to look for such jobs. If a job is in an *Error* state, ERA deletes it from its internal state, reclaiming its resources. If the job partially fails, ERA checks if the number of active workers (up and running) is more than the minimum number of workers specified by the user. In this case, ERA scales down this job to that number, forcing the active workers to initiate a new *rendezvous* and continue training. Otherwise, ERA considers the job failing and reclaims its resources.

---

[6] Note that in the case of the presence of stragglers, averaging will be vulnerable to outliers. As we assume a homogeneous network and GPUs, we leave this issue to future extensions of our work.

[7] The relation between the global batch size and the throughput is sometimes not completely linear [219]. Yet, our experiments showed the linear model is sufficient for most cases.

4.4.4 Batch Size Synchronization

ERA communicates the scaled batch size to the applications through their ConfigMaps (see Section 4.3.2). Each worker (hosted on a pod) should then read the new batch size and apply the change accordingly. We found that K8s does not update the ConfigMaps of all pods synchronously, i.e., different workers might read different values for the batch size simultaneously; this might lead to an unexpected convergence behavior or long lags in other cases.

To resolve this issue, ERA annotates the batch size change event with a timestamp, which workers can then read while getting the new batch size. When workers read a new value for the batch size, they communicate the latest timestamp they read to each other. The worker with the latest timestamp then broadcasts the up-to-date value of the batch size, restoring the synchrony of the batch size among all workers.

4.5 Performance Evaluation

We discuss in this section the results of our empirical evaluation of ERA. First, we describe our testbed, our setup and metrics, and the baselines. Then, we explain the results in detail.

4.5.1 Methodology

Testbed. We use a Kubernetes–managed 64–GPU cluster. Each worker has a dedicated GPU, 5 CPU cores, 64 GiB RAM, and 2×40 Gbps Ethernet.

Metrics. We use the following metrics for evaluation.
1. *Average throughput.* The average number of processed samples per second over all running jobs in the cluster.
2. *GPU utilization & GPU memory utilization.* The average GPU–related metrics for all *active* GPUs in the cluster. We use *nvidia-smi* to measure these metrics, and we set it to read the average value over the last 5 seconds.
3. *Cluster utilization.* The ratio of the number of used GPUs to the total number of GPUs in the cluster.
4. *Makespan.* The time required to finish all jobs.
5. *Average job completion time (JCT).* The average time required for one job to finish.
6. *Average queuing delay.* The average time a job would wait in the queue before running on the cluster.

Applications. We use 2 DL applications for our evaluation: (a) image classification, training 12 different models, including AlexNet [142], ResNet18 [104], VGG19 [224], and DenseNet121 [111], using ImageNet [66], and (b) language modeling, fine-tuning BERT [132] using the Wikipedia dataset. We choose these applications for their wide use in academic and industry settings.

On each job scheduling event (see below our job schedulers), we randomly choose one application from this pool. By default, we use a batch size of 64 for the first application and 8 for the second one. Unless otherwise stated, we do not set an upper bound for the batch size, and we use a fixed learning rate throughout the training. We start training with 2 workers and set the minimum and the maximum number of workers to 1 and 16, respectively. We run training for a fixed number of epochs. Each worker logs a measurement every 10 iterations.

ERA configuration. The correct settings of ERA are cluster and workload dependent. Using trial and error, we chose the following setup. ERA monitors jobs every 2 minutes, takes a decision every 4 minutes, and collects garbage every 4 minutes. We set the GPU compute and memory utilization thresholds to 85% and 80%, respectively.

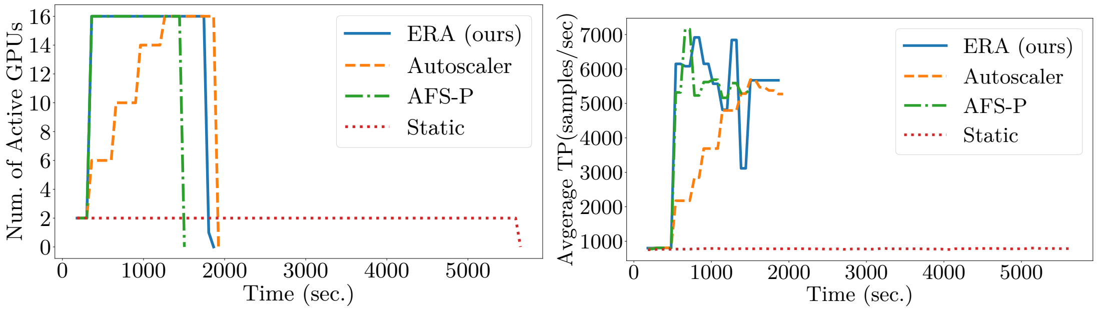
Job schedules. We experiment with 3 schedules for each we submit n jobs in total. The first, denoted by *Poisson*, submits a job every 4 minutes on average, according to the Poisson distribution. This job schedule is our proxy to a realistic setup where jobs from different business teams arrive regularly following a popular distribution. The second, denoted by *All_at_once*, submits n jobs once at the beginning. This schedule simulates the submission of a batch of jobs at once, for example, for hyperparameter tuning or similar tasks. The third, denoted by *Fixed*, submits 1 job every 2 minutes. This schedule simulates periodic job submissions for A/B testing, for instance.

4.5.2 Baselines

We compare ERA to the following baselines.

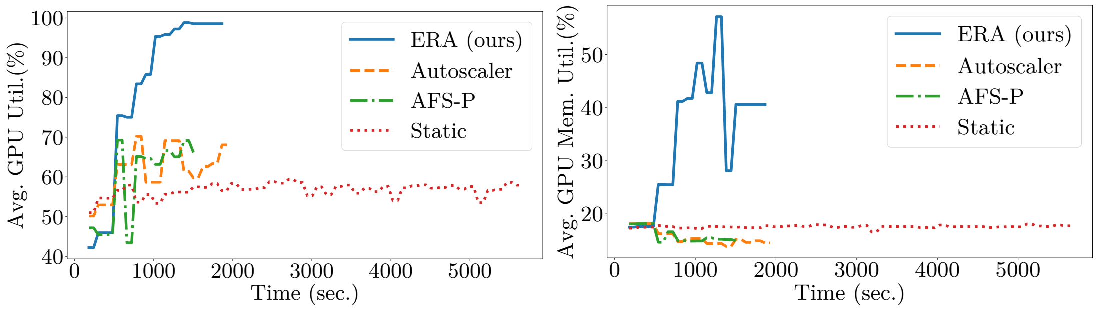
Static (no elasticity). This is status quo: resource scheduling follows a First-Come-First-Served policy, and the amount of resources allocated to a job remains fixed throughout training. If a job arrives and there are no available resources, the job queues until some resources are freed. Comparing ERA to this baseline highlights the benefits of both horizontal scaling and batch size scaling.

Autoscaler [190]. Autoscaler scales jobs horizontally based on the expected throughput gain. It uses a fixed increment per decision. After scaling up, Autoscaler measures throughput. If throughput has increased, it can further scale up this job in the next round. Autoscaler applies the same criteria in the case of scaling down: only throughput is taken into consideration. Note that the value of the fixed increment (or decrement) controls an efficiency–concurrency tradeoff. Using a large increment (though helps improve throughput) reduces concurrency (i.e., less number of concurrent tenants can use the cluster), leading to higher makespan and job completion time. The main influence behind this behavior is the fact that Autoscaler does not scale down existing jobs to accommodate for waiting jobs in the queue (i.e., it scales down



(a) Number of used GPUs

(b) Throughput



(c) GPU utilization

(d) GPU memory utilization

Figure 4.3: Microbenchmark experiment: we submit a single job, training ResNet18 using ImageNet.

a job only if its throughput is degraded). Using a small increment allows for more concurrent users to use the cluster but also reduces the average throughput of the training jobs. In our experiments, we set the increment to 4 workers.[8]

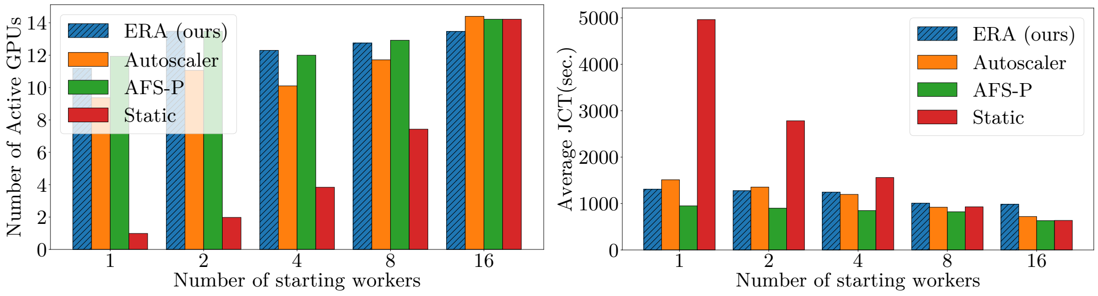
As Autoscaler is not open-sourced, we implemented our version of it. We use PyTorch–elastic as an elastic engine rather than the elasticity methods described in the original paper. Essentially, we opt for unifying the underlying elastic engine for a fair comparison. We apply scaling decisions using the same schedule we use for ERA (i.e., every 4 minutes).

AFS-P [115]. AFS-P also focuses on horizontal scaling, yet unlike Autoscaler, it addresses the multi-tenant case.[9] AFS-P first assigns 1 GPU to each job, which then competes for the rest of the GPUs (if any). For each additional GPU, AFS-P selects the job with the highest priority to get it. AFS-P gives priorities to jobs based on their throughput gain in the case of adding 1 GPU. Note that AFS-P does not scale the per-GPU batch size. We have also implemented AFS-P as it is not open-sourced.

---

[8] The original paper has also used an increment of 4 workers.

[9] [115] also proposed AFS-L, which assumes prior knowledge of the required time to finish a job. As we do not assume this in ERA, we exclude AFS-L from our baseline set.



(a) Average number of used GPUs

(b) Job completion time (JCT)

Figure 4.4: The effect of choosing a different number of starting workers, with only 1 job, training ResNet18 using ImageNet. Based on our configuration, no scaling can happen beyond 16 workers.

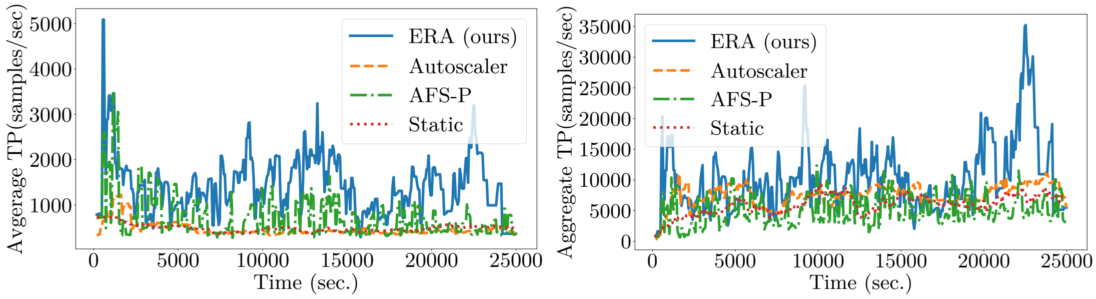
4.5.3 Microbenchmark

We start with a microbenchmark experiment, whose results are shown in Figure 4.3. We submit a single job, training ResNet18 using the ImageNet dataset to the cluster.

Figure 4.3a shows the number of GPUs used throughout the training. Static does the whole training using 2 GPUs, Autoscaler increases the number of GPUs by 4 in all scaling decisions (indicating that the job would benefit from more GPUs, up to 16 in total), where both ERA and AFS-P jump from 2 GPUs to 16 in the first scaling decision. Both systems decide to use 16 GPUs after 4 minutes, whereas Autoscaler reaches this optimal value after 16 minutes. This delayed decision is reflected in the average throughput of Autoscaler (see Figure 4.3b): while ERA and AFS-P can reach more than 6000 samples/second after only a few minutes, Autoscaler lags behind, reaching slightly less than this value after around 28 minutes. Figure 4.3b shows as well the poor performance of Static throughout the whole training, which stresses the need for elasticity during training.

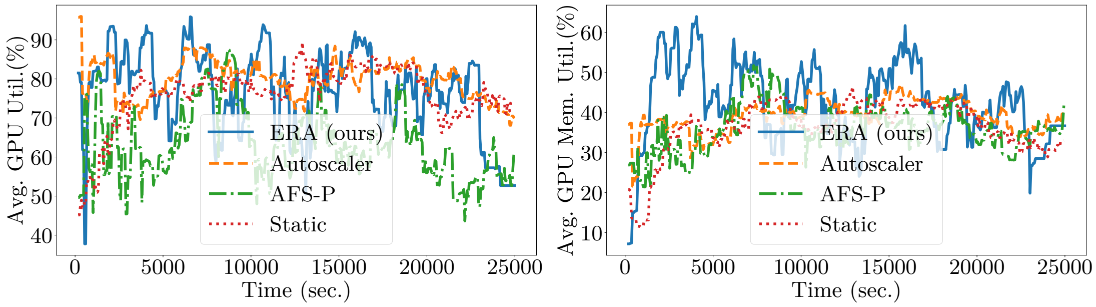
The main benefit of ERA manifests in Figures 4.3c and 4.3d, which show the GPU utilization and the GPU memory utilization. The main factor that controls both metrics is the batch size. Figure 4.3c shows that scaling the batch size leads to almost 100% GPU utilization while the other baselines cannot achieve more than 70%. Figure 4.3d shows up to 60% memory utilization for ERA, where the other baselines are upper bounded by 20% utilization. This shows that the two–dimensional scaling of ERA is beneficial in such a case.

Initial number of workers. Our second microbenchmark (Figure 4.4) shows the effect of choosing different numbers of *starting* workers. Figure 4.4a shows the average number of GPUs used throughout the training. Static uses a fixed number of GPUs (equal to the number of starting workers). The other systems scale this job up, using a larger number of GPUs. As ERA and AFS-P scale quickly, they report a higher average for the number of active GPUs than Autoscaler.



(a) Average throughput

(b) Aggregate throughput



(c) GPU utilization

(d) GPU memory utilization

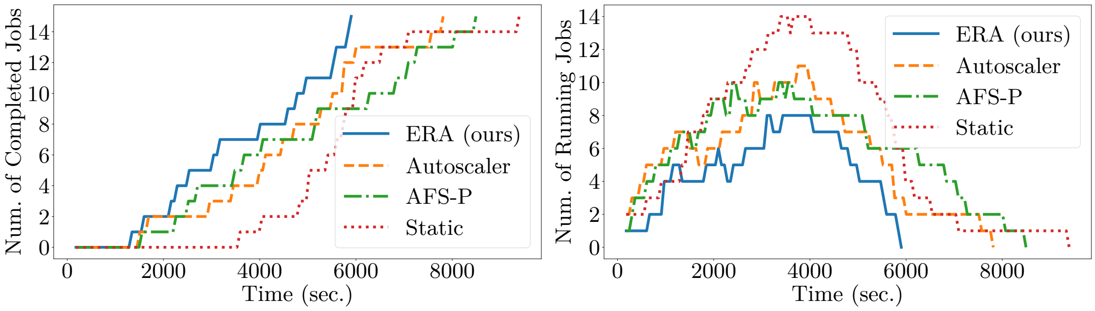
Figure 4.5: End-to-end real–life scenario with 100 submitted jobs.

The enhanced cluster utilization benefit is shown in Figure 4.4b, which shows the average JCT, the defacto standard metric of scheduling efficiency in literature, e.g., in [251, 274, 190]. All systems, except Static, manage to achieve comparable completion times *regardless of* the starting number of workers. This is an interesting result because it shows that elastic systems can converge to the best number of workers even if the starting number (specified by the user) is sub-optimal. Another takeaway from this figure is that ERA behaves slightly worse than the other elastic baselines in some cases (including achieving higher job completion time in Figure 4.3). The main reason is the overhead of batch size scaling. Although such an overhead is very little (as it only entails re-creating the in-memory data loader), it manifests clearly in such a small experiment. This overhead is negligible in large–scale experiments, as we show in Section 4.5.4.

4.5.4 End-to-end Large Experiment

Figure 4.5 shows the results of an end-to-end experiment with 100 jobs submitted to the cluster according to the *Poisson* schedule. For a fair comparison, we run each system for a fixed period: 7 hours. We pose this experiment as a proxy to real scenarios with many requests (of various workloads) to be served by one shared cluster.

Figure 4.5a shows the average throughput over time. Most of the time, ERA achieves the best average throughput, followed by AFS-P, then Autoscaler, and finally, Static. ERA's throughput



(a) Number of completed jobs

(b) Number of running jobs

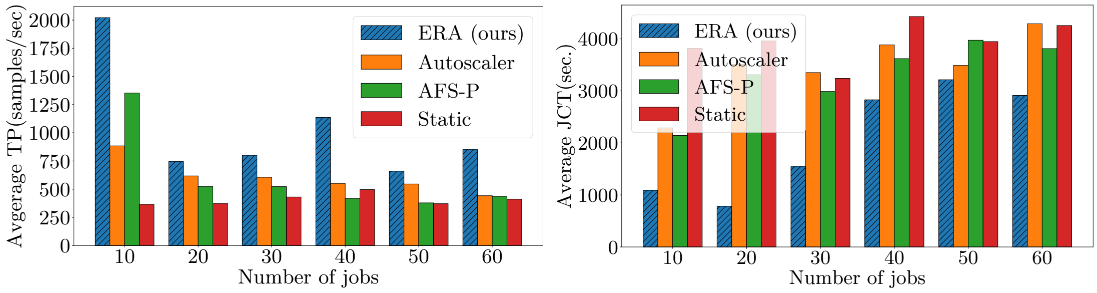
Figure 4.6: The actual effect of improving raw performance metrics (e.g., throughput and GPU utilization) on the rate of finishing jobs.

is the highest because it uses all the available resources, horizontally and vertically, where the other baselines fail to do so. AFS-P has an advantage over Autoscaler because the former is designed for multi-tenant clusters, unlike the latter, which looks only at scaling one job.

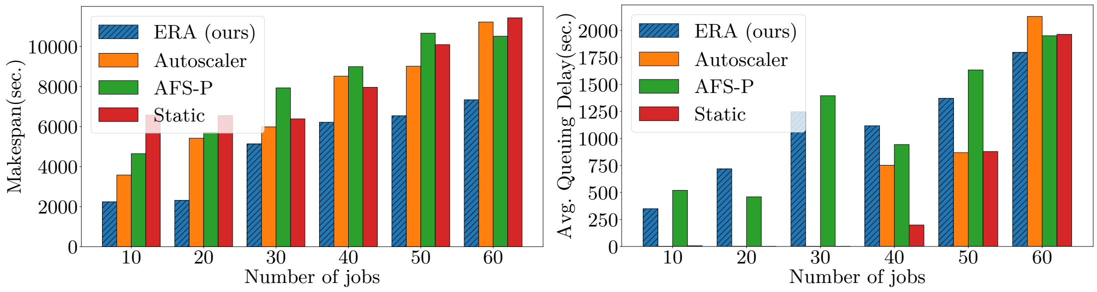
The average throughput could be misleading in some cases because it might favor a system that runs 1 job quite well over another system that can accommodate multiple jobs, each behaving slightly worse. Figure 4.5b depicts the *aggregate* throughput: the summation of all running jobs throughput. ERA remains on top of the other baselines. But, both Autoscaler and Static show more competitive performance than looking only at the average throughput. We infer then that both systems do a good job in accommodating multiple parallel jobs. The reason for this behavior is that (1) Static does not scale jobs, leaving the maximum room for incoming jobs, whereas (2) Autoscaler applies scaling slowly, leaving enough time for the other jobs to join the cluster without waiting in the queue.

Figure 4.5c shows the average GPU utilization throughout the training. ERA utilizes the GPUs slightly more (on average) than the other baselines. Notably, we run various jobs with different workloads, where some jobs utilize the GPU better than others; this leads to fluctuations in the reported results. Another interesting observation is that AFS-P behaves worse than the other baselines. When AFS-P increases the number of workers, it increases consequently the communication overhead. Without scaling up the batch size simultaneously, that overhead prevails, reducing the GPU utilization. Essentially, since training is synchronous, GPUs wait for each other to finish their execution before moving to the next training iteration. This result shows the necessity of scaling the batch size to reap all the benefits of horizontal scaling.

Figure 4.5d shows the average GPU memory occupancy throughout the training. This metric is largely controlled by the batch size. As ERA is the only system that scales the batch size, it outperforms the other baselines in almost all cases, maximizing the GPU memory utilization.



(a) Average throughput (b) Job completion time (JCT)



(c) Makespan (d) Queuing delay

Figure 4.7: Scalability with a different number of submitted jobs to the cluster. All jobs start at once at the beginning of the experiment.

A closer look. We take a closer look at the actual benefits of improving the raw performance metrics like throughput and GPU utilization. We consider the same experiment but with only 15 jobs (see Figure 4.6). We focus on two specific metrics: the number of completed jobs (Figure 4.6a) and the number of parallel running jobs (Figure 4.6b), both over time.

Figure 4.6a shows that ERA has a higher rate of finishing jobs, which shows, practically, the benefit of having high throughput and high GPU utilization. Essentially, jobs run faster and hence, finish first. Figure 4.6b confirms this behavior. Although ERA is the best system to parallelize jobs, it has the lowest peak in the number of *parallel* jobs; this happens because jobs finish faster, even before the other jobs start.

4.5.5 Scalability of ERA

Figure 4.7 conveys the scalability with different numbers of submitted jobs to the cluster. With 64 GPUs available in the cluster, at most 32 jobs can run in parallel (each starting with 2 GPUs); the remaining jobs should queue until some resources are freed.

Figure 4.7a shows the throughput, averaged over both time and jobs. In general, the elastic systems behave better than Static in almost all cases. Essentially, they use the idle resources once they are available, improving their throughput. In the very congested case (with 60 jobs), there is no clear benefit for elasticity, except with ERA, which achieves roughly double the throughput of the other systems. The main reason for this behavior is the batch size scaling, which enables ERA to improve its throughput further.

Figure 4.7b shows that ERA achieves the lowest average JCT in all cases. With less than 32 jobs, ERA behaves significantly better because more resources are available in this case. With many jobs, the benefits of ERA start to fade because resources are consumed even before scaling. Even in this case, scaling the batch size gives ERA an additional advantage compared to the other baselines.

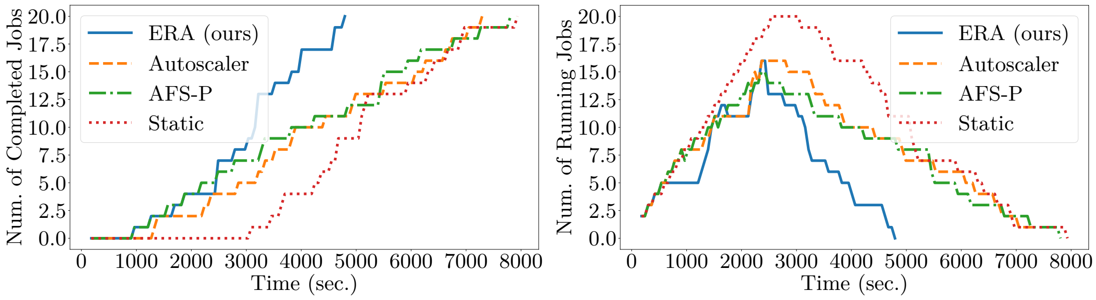
Similarly, ERA has the lowest makespan in all cases (Figure 4.7c). Increasing the number of jobs from 20 to 30 significantly increases the makespan of ERA as, in this case, there are not many free resources to be used. With more than 30 jobs, ERA still scales better than the other baselines (noting the rate of increase in makespan with the number of jobs).

Figure 4.7d shows an interesting result, which can be arguably viewed as one downside of elasticity. It shows that systems that apply aggressive scaling, namely ERA and AFS-P, suffer from higher queuing delays, especially with a few jobs to serve. As both systems try to use all the resources in the cluster, the upcoming not-yet-scheduled jobs do not usually find resources. Although this situation is typically resolved at the next scaling round, arriving jobs have to queue until that time comes. This effect diminishes with high contention on resources because jobs have to queue anyway in this case. Furthermore, since the scaling-aggressive systems can finish jobs faster (see Figure 4.7b), they can quickly free resources for the queued jobs.

4.5.6 Fixed Job Schedule

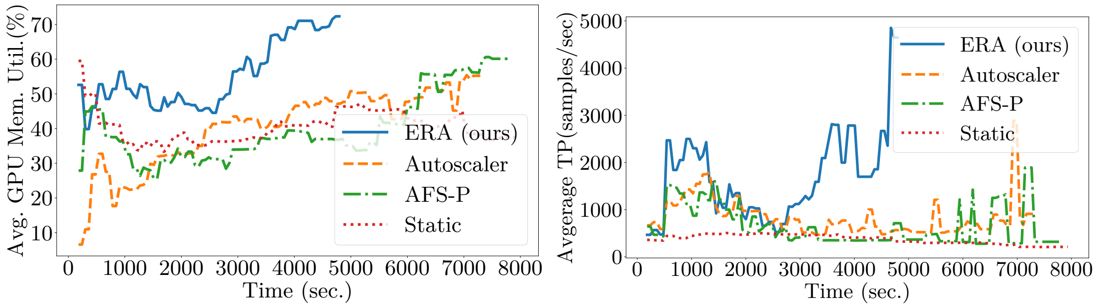
Figure 4.8 depicts the results of running 20 jobs following the *Fixed* schedule, with one job submitted every 2 minutes. This can be seen as a proxy to cases where the number of jobs submitted to the cluster spikes, e.g., as in [123]. Figure 4.8a shows that ERA has the highest rate of finishing jobs with a makespan of about 5000 seconds. The second best is Autoscaler, which finishes in around 7300 seconds.

Three factors are instrumental in achieving this result. First, ERA maximizes the number of parallel running jobs. Figure 4.8b highlights this fact: ERA allows the number of running jobs to jump faster than the other baselines. For instance, ERA increases the number of running jobs from 5 to 11 in one scaling decision. This parallelization ensures fairness and improves the JCT and the makespan. Second, the second constraint in its optimization function (Equation 4.1) guarantees that all the resources in the cluster are always utilized. Third, ERA maximizes the utilization of the individual GPUs memory, which typically reduces the time taken to finish jobs. Figure 4.8c depicts this fact, where it shows that by the time the number of parallel running jobs increases, the average GPU memory utilization also increases, exceeding 70% in the best case. AFS-P achieves the second-best utilization, which is upper bounded by 60%. This double scaling (i.e., in two dimensions) gives a significant boost in



(a) Number of completed jobs

(b) Number of running jobs



(c) GPU memory utilization

(d) Average Throughput

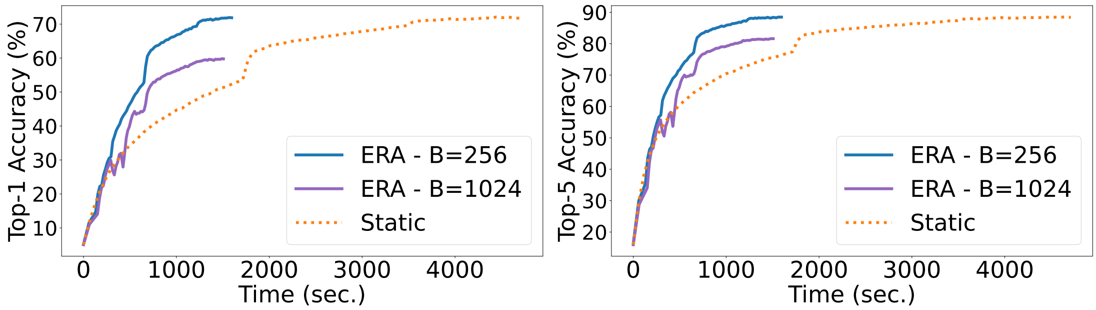
Figure 4.8: Fixed schedule results: we submit 1 job every 2 minutes.

throughput, as shown in Figure 4.8d.

An interesting observation is the simultaneous drop in throughput (Figure 4.8d) and the GPU memory utilization (Figure 4.8c) during scaling up (see Figure 4.8b). This happens because scaling up entails stopping some pods, creating new ones, and letting the pods assigned to the same job meet to restart training. This process is costly and has a high overhead, which is tied to PyTorch–elastic (our elastic engine). We speculate that using another elastic engine might reduce this overhead.

4.5.7 Convergence Comparison

We experiment the effectiveness of ERA's approach to preserve convergence while scaling. We submit 1 job, training ResNet18 using ImageNet. We start with 8 workers, a *local* batch size of 64, and a learning rate of 0.08. We experiment with 2 bounds for the *local* batch size: 256 and 1024. The first bound is known to be optimal for training ImageNet [88], whereas the second bound is intentionally high to show a case of sub-optimal settings. We scale the learning rate linearly with the batch size, and we use our batch size synchronization technique (Section 4.4.4). We run training for 80 epochs, and we show the Top-1 accuracy (Figure 4.9a) and Top-5 accuracy (Figure 4.9b). Note that AFS-P and Autoscaler do not employ specific techniques to preserve convergence, and they have not reported training accuracy results in their papers. For this reason, we have excluded them from the baseline set for this experiment.



(a) Top-1 Accuracy

(b) Top-5 Accuracy

Figure 4.9: The convergence of ERA (with 2 maximum batch sizes) compared to Static while training ResNet18 with ImageNet.

In terms of time, although all systems run for a fixed number of epochs, ERA finishes first as it can execute training epochs faster thanks to its elasticity. In terms of accuracy, both figures show that ERA (with B=256) and Static achieve similar convergence behavior, which confirms that using the correct settings, scaling the learning rate with the batch size can effectively preserve the target convergence behavior. However, ERA (with B=1024) fails to converge to the best accuracy, demonstrating that setting an ill-suited maximum batch size can degrade convergence. Indeed, with a maximum batch size of 1024, training got stuck in a bad local minimum, and no matter how long the model trains, it does not reach a better value, i.e., Figure 4.9 shows the final accuracy at which training converges.

4.6 Related Work

Elastic schedulers. A few papers explored elasticity in running deep learning (DL) workloads. Or et al. [190] proposed Autoscaler to scale *independent* jobs dynamically based on their throughput. The authors also presented an algorithm to detect and remove stragglers, which further improves throughput. Hwang et al. [115] proposed AFS-L and AFS-P to improve the overall throughput of all the jobs in a cluster. The main idea is to prioritize efficient jobs to short jobs during resource allocation. The authors demonstrated that AFS-L outperforms AFS-P, yet the former requires the knowledge of the remaining time for jobs to finish, unlike the latter. Saxena et al. [217] proposed an autoscaling system tailored for IBM Ffdl platform [122]. It considers multiple batch sizes (in a range given by the user) for optimal throughput. Autoscaling scaling decisions are taken only with job completion events (rather than periodically). Recently, Athlur et al. [22] proposed Varuna, an elastic system to train massive deep learning models on commodity GPUs and networking using a combination of data and model parallelism. Varuna's goal is different from the other schedulers as it aims at running huge workloads on inexpensive hardware. It does that by running training on *spot* VMs and adapting the *local* batch size to the available memory. Yet, Varuna keeps the *global* batch size constant despite the number of machines running training. In summary, ERA differs from these works in 3 aspects: (a) it improves the individual GPU utilization (by scaling the batch size), (b) it maximizes the number of parallel running jobs (by scaling down the running jobs and making room for the waiting no-yet-scheduled jobs), and (c) it does not require knowledge on the workload nor the remaining time for jobs' completion.

Non-elastic schedulers. DL scheduling was extensively studied in the past few years. Xiao et el. [251] proposed Gandiva, whose main idea is to utilize the intra-job predictability to (a) time-slice GPUs and (b) migrate jobs to better-fit GPUs for better cluster efficiency. Peng et al. [196] presented Optimus, which uses online resource-performance models to predict convergence during training. Using such predictions, Optimus allocates resources to DL tasks to minimize JCT. Gu et al. [90] proposed Tiresias, which relies on two algorithms for job scheduling and placement to reduce JCT. Tiresias also uses prior from previous jobs to predict the training time of running jobs. HiveD [279] focuses on a different aspect of the problem: sharing GPUs safely and efficiently. HiveD proposed a hierarchical logical structure of the cluster. The authors showed this minimizes fragmentation and queuing delays and maximizes resource utilization. Salus [274] presented two primitives, job switching and memory sharing, which achieve low memory fragmentation, high average throughput, and low JCT. The authors used such primitives to build different sharing policies. Themis [169] uses a two-level scheduling architecture in which training tasks contribute to an auction, run by a central arbiter, to compete for the resources. It explores the tradeoff between fairness and efficiency: it promises efficiency in the short term but fairness among all tasks in the long run. All these proposals are orthogonal to ERA, whose goal is to explore the benefits of elasticity in solving the resource allocation problem.

System support for elasticity. We now discuss papers that provide system support for elasticity, i.e., to apply actual scaling of jobs on the ground. Qiao et al. [201] proposed Litz to support elasticity in the parameter server architecture. Litz relies on three principles: stateful workers, model scheduling, and relaxed consistency. The authors showed that Litz scales jobs efficiently and with minimal overhead. Mai et al. [170] proposed KungFu, a general approach to scale learning in different aspects. KungFu provides APIs to monitor and adapt hyperparameters, including the batch size, the learning rate, and the number of workers. KungFu achieves this by defining adaptation policies (APs), which can be embedded inside the dataflow graph of the training task. Xie et al. [259] proposed Elan, which relies on multiple techniques that promote elasticity efficiency, such as concurrent IO-free replication, faster communication with RDMA, and topology-aware communication. Like ERA, Elan scales the learning rate linearly with the batch size for preserved convergence. Wu et al. [249] proposed EDL, a leader–based system that provides elasticity APIs for popular ML frameworks like TensorFlow [1] and PyTorch. EDL incorporates techniques to reduce scaling overhead like stop-free scaling and dynamic data pipeline. ERA uses PyTorch–elastic [199] as its underlying engine, and ERA can be integrated with any of these elastic engines.

Scaling the batch size. Few papers looked at the benefits of scaling up the global batch size (which could be seen as equivalent to scaling up the number of workers) to distributed training. Such papers looked at theoretical guarantees and practical benefits of increasing the batch size on convergence. In [226, 227], Smith et al. explained how the batch size affects noise scale, which in turn affects the trained model accuracy. The main conclusion of these works is that increasing the batch size is equivalent to (in terms of accuracy) decaying the learning rate, which is a successful common practice in many algorithms. In addition, the authors showed that increasing the batch size is more desirable as it achieves faster convergence (w.r.t. time) compared to decaying the learning rate. Shallue et al. systematically and empirically explored the benefits of increasing the batch size on convergence [219]. They showed there is no one-size-fits-all rule for such an issue and that such behavior is workload and hyper-parameters dependent. In general, they showed that increasing the batch size can lead to faster convergence, but only to some limit, after which the benefits of increasing the batch size diminish. We used this insight while designing ERA essentially by allowing the user to choose an upper bound on the batch size to be used for training.

Scaling the learning rate. Some papers studied how to change the learning rate by scaling the batch size. Lin et al. [164] proposed a simple linear scaling for the *effective* learning rate, which captures both the learning rate and the momentum coefficient. The authors showed how their scaling rule improves both training and validation accuracy, especially in the case of abruptly changing the batch size in the middle of training. Johnson et al. proposed AdaScale [129], which scales the learning rate based on the variance reduction that accompanies the increase in the batch size. The authors showed that in some cases, this scaling rule is more beneficial than the simplistic linear scaling. In a related line of research, Jin et al. proposed AutoLRS [128], an optimization method to automatically *tune* the learning rate in the middle of training to minimize the validation loss. AutoLRS periodically uses Bayesian optimization to predict the best learning rate for the next group of learning iterations. ERA can use this approach to achieve the best accuracy despite scaling the batch size. Although ERA uses linear scaling of the learning rate, its design allows for replacing that approach with a more complex one, incurring only a little engineering overhead.

4.7 Concluding Remarks

In this chapter, we presented ERA, an Elastic Resource Allocator for multi-tenant deep learning clusters, which are typically used in today's datacenters. Essentially, ERA scales in two dimensions: the number of workers contributing to training and the per-worker batch size. ERA is dynamic: it scales jobs in the middle of training while not requiring any prior information on the workloads nor their convergence behavior. We integrated ERA with Kubernetes and PyTorch–elastic, and we evaluated ERA with a 64–GPU cluster. We showed that ERA improves throughput and resource utilization while minimizing the average job completion time compared to two state-of-the-art baselines.

5 Cloud Object Stores for Faster Transfer Learning

In this chapter, we consider training in the cloud, where there are typically two independent tiers: cloud object store (COS) and compute. Traditionally, the COS's job is only to store data while all the computation happens on the compute tier. Nowadays, the computational capabilities of the COS are significantly increasing. Yet, fundamental constraints remain: the cloud network to the compute tier remains a bottleneck, and the COS computational resources remain at a premium, despite the upgrade because they only complement the compute tier and not replace it.

We identify transfer learning (TL) as an important class of applications that can naturally fit today's augmented COS. TL comprises two phases: pre-training and fine-tuning. We consider the latter phase in this chapter, and we refer to it as TL. We show how to leverage the unique structure of TL, a combination of feature extraction and training, to flexibly bypass the aforementioned constraints and improve both client and operator-centric metrics. The key to both is to mitigate the network bottleneck by carefully splitting the TL application such that feature extraction is, partially or entirely, executed next to storage. Such splitting additionally decouples the requirements of the two computational phases, enabling increased concurrency inside the COS while avoiding out-of-memory errors via storage-side batch size adaptation. Guided by these insights, we present HAPI, a processing system for TL that spans the compute and the COS tiers, enabling significant improvements while remaining transparent to the user.

5.1 Introduction

Disaggregated cloud object stores (COS) have become fundamental to the cloud's scalability and cost-effectiveness. All major cloud providers offer this popular service (e.g., Amazon S3 [13], Google Cloud Storage [86], Azure Blob Storage [43]). For several years, COS have also included limited computational capabilities. For example, since 2017, users can run a limited subset of SQL queries next to storage via Amazon S3 Select [266, 275]. Such compute pushdown can significantly improve performance by reducing the amount of data sent between the COS and the compute tier.

Current and future capabilities of the COS. Very recently, the COS's computational capabilities have seen considerable upgrades. Amazon S3 Lambda [15] can now run user-defined functions next to storage, significantly expanding the scope of the computations that can be pushed down. In addition, users can perform more complex tasks such as image processing inside the COS [56]. Augmenting the COS with GPUs is the natural next step because GPU bandwidth has long exceeded storage and network bandwidth [155]. Concerned about the storage/network bottleneck, Nvidia is actively expanding its AI stack to include high-performance storage. Nvidia acquired Excelero[1] (creators of NVMesh, which shares NVMe drives across networks) and SwiftStack[2] (OpenStack Swift [189] maintainer) and developed GPUDirect Storage [117, 187] (direct data path between local/remote storage and GPU). Furthermore, cloud providers are even starting to include GPUs inside the COS [116]. Nevertheless, the same fundamental constraints remain: the network between the COS and the compute tier is the bottleneck [137, 198], and the computational resources inside the COS remain limited, despite the upgrade because they are only expected to complement the compute tier rather than replace it. Notably, this new trend does not contrast with resource disaggregation because it does not remove *the barrier* between the storage and the compute, i.e., we expect to have independent computing resources to do the main computations.

In light of this trend and constraints, it is important to identify the applications that can most benefit from today's augmented COS to improve client-perceived metrics. Equally important is to identify opportunities for the cloud operator to navigate the compounded operational challenges to improve operator-level performance metrics like application throughput and resource utilization. Following the theme of this thesis, we tackle this challenge with a focus on machine learning (ML) applications. Larger and larger ML datasets are stored in the COS [21], making ML applications a logical target to try to run next to storage.

We identify transfer learning (TL)[3] as a natural fit for today's augmented COS. TL is an increasingly important area of ML that allows knowledge in one task to be reused in a different but related task [246]. The importance of TL is evidenced by the growing support from cloud giants [203, 14, 85]. McKinsey [9] ranks TL as the most used ML technique, surpassing deep learning and reinforcement learning. The key to this natural fit lies in the unique structure of TL: a combination of feature extraction and training phases using a deep neural network (DNN). We show how this structure can be leveraged to make the most of the augmented COS while flexibly bypassing the disaggregated cloud's constraints to ultimately enable improvements in both client and operator-level metrics of interest.

Our main idea is to carefully split the TL application such that the feature extraction phase is, partially or entirely, executed in the COS. The client in the compute tier executes the training

---

[1] https://www.excelero.com/press/excelero-launches-nvmesh-on-azure/

[2] https://www.anandtech.com/show/15605/nvidia-acquires-swiftstack-an-object-storage-company

[3] To be precise, we mean here the *fine-tuning* part of TL.

phase and potentially the later part of the feature extraction phase. Our approach for choosing the split point navigates two potentially opposing insights. On the one hand, it is best to minimize the amount of data sent over the bottleneck network. On the other hand, one should efficiently use the limited computational capabilities of the COS by avoiding unnecessary pushdowns. Both insights may conflict because, in general, the first layers of DNNs have the largest output sizes. Guided by our per-layer measurement study (Section 5.3.2), we use the insight that layer output sizes decrease non-monotonically, so it is often possible to split early at a layer with a comparatively smaller output size.

The key to improving operator-centric metrics, such as resource usage (we focus on GPU memory) and compute throughput, is noticing that splitting also enables decoupling the requirements of the feature extraction and the training phases. The main insight is that the batch size (i.e., the computation granularity) only affects the model accuracy during the training phase. This allows for flexibility during feature extraction, where a dynamic batch size adaptation approach can help increase concurrency inside the COS while avoiding out-of-memory (OOM) errors that plague ML practitioners.

We combine our splitting and batch adaptation algorithms into HAPI,[4] our processing system for TL that spans both the COS and the compute tiers. HAPI is completely transparent to the user, uses internally only very inexpensive profiling runs and enables even low-end CPU-only users to contribute.

Specifically, the contributions of this chapter are:

1. The identification of TL as a natural fit for today's augmented COS.
2. A per-layer measurement study of TL applications' runtime, per-layer output size, and GPU memory usage.
3. A splitting algorithm that divides the feature extraction phase between the COS and the compute tiers, improving application runtime by reducing data transfers over the cloud network.
4. A batch adaptation algorithm that decouples the requirements of the training and feature extraction phases, enabling increased computation throughput and GPU memory utilization via increased concurrency in the COS.

Our evaluation of HAPI with state-of-the-art models shows up to 11× improvement in application runtime and up to 8.3× reduction in data transferred between the COS and the compute tier compared to running TL entirely in the compute tier while enabling 100% use of GPU memory. Compared to running TL entirely in the COS, HAPI improves throughput by 4.9× while serving 10 requests. Interestingly, we show that HAPI enables low-end CPU-only users to run TL up to 7.83× faster compared to running computation solely in the compute tier.

---

[4] https://en.wikipedia.org/wiki/Hapi_(Nile_god)

5.2 Background

5.2.1 Cloud Object Stores

Cloud object stores (COS), such as Amazon S3 [13], Google Cloud Storage [86], and Azure Blob Storage [43], have become a common way to store large-scale unstructured data because they provide ease of use, high availability, high scalability, and durability at a low cost [161]. The COS is connected to the compute tier by a network that, unfortunately, has shown to be a bottleneck [140]. For example, studies [137, 198] report less than 100MBps read throughput per connection from Amazon S3. In light of this, there has been a trend to push down computation inside the COS, starting with a restricted subset of SQL (e.g., Amazon S3 Select) and recently expanding to more complex computations such as image processing [56].

Despite the recent COS augmentation to enable more complex processing next to storage, two challenges remain. First, the cloud network between the compute tier and the COS can easily become the bottleneck. Second, the computational resources inside the COS come at a premium because they are only meant to complement and not replace the compute tier. Therefore, we have to use these resources efficiently.

In this work, we use OpenStack Swift [189], an open-source, highly-scalable object store. We assume a Swift-like COS design with two components: *proxy servers* and *storage nodes*. A proxy server interfaces with the client, receiving client requests with a REST API and reading data from storage nodes. These storage nodes store data in the form of *objects*, which are typically replicated across multiple disks for fault tolerance. Inspired by recent trends [116], we assume that COS proxy servers are equipped with GPUs.

5.2.2 Hardware-accelerated pushdowns

Pushdowns were initially restricted to a subset of SQL, including filtering, projecting, and aggregation (e.g., Amazon S3 Select [33]). The current, natural trend is to offer the benefits of pushing down to a wider range of applications. Unfortunately, there are several reasons why restricting pushdowns to CPUs leads to wasted resources and performance. First, for more complex operations, CPUs can become a bottleneck. Studies show that even with 32 cores, an SGD optimizer can bottleneck the CPU when using a 100Gbps network [127]. Second, it is not sufficient for the CPU processing to be just faster than the network because the output of a pushdown may be smaller than its input. For a pushdown to generate an output at 12.5GBps ($\approx$ 100Gbps), assuming an input/output ratio of 2, it needs to process input at 25GBps. Finally, the aggregate storage bandwidth of a storage server tends to increase faster than the capabilities of CPUs [263].

As a result, the current trend is to allow pushdowns to use specialized hardware such as FPGAs and GPUs. PolarDB uses mid-range low-cost FPGAs inside storage drives [44], Aquoman [263] proposes to push down TPC-H queries on FPGAs, and the AWS Advanced Query Accelerator

(AQUA) pushes down analytical queries on FPGA-based AWS Nitro chips [19]. Regarding GPUs, several works [45, 97] have proposed to use them in storage systems to speed up erasure coding. IBM is offering high-performance storage with NVIDIA GPUs [117, 116]. Finally, there is a push to more-closely integrate storage with GPUs [187, 154], which further increases the appeal of next-to-storage GPUs.

5.2.3 Transfer Learning

Transfer learning (TL) [283] is an increasingly popular ML technique with a wide range of applications ranging from one-shot and multi-shot learning for language models [40, 244], self-driving cars [53], healthcare [206], personalization of models (in the context of federated learning) [172, 126] to transferring attacks and defenses from one domain to another [52, 48].

TL comprises two phases: *pre-training* and *fine-tuning*. The first phase is very similar to vanilla training, where all the parameters/weights of the model are updated. Pre-training typically targets a task with a large amount of training data. In the second phase, fine-tuning, TL adapts the pre-trained model (to solve an initial task) to a similar task by transferring common model features. We consider solely fine-tuning in this work. TL has been widely used in industry and academia since it provides the advantages of reducing training time for a deep neural network (DNN) and reducing the generalization error [83] while training with a new dataset.

In general, to solve a new task, TL first performs *(i) feature extraction*, to extract the features from new input data using (partially or entirely) the pre-trained model, and then *(ii) training*, to create a new classifier using the extracted features [83]. Typically, the first few layers of the pre-trained model are *frozen* during the training process, i.e., the weights of these layers are not updated with backpropagation. We call the last layer of the feature extraction phase: the *freeze index*.

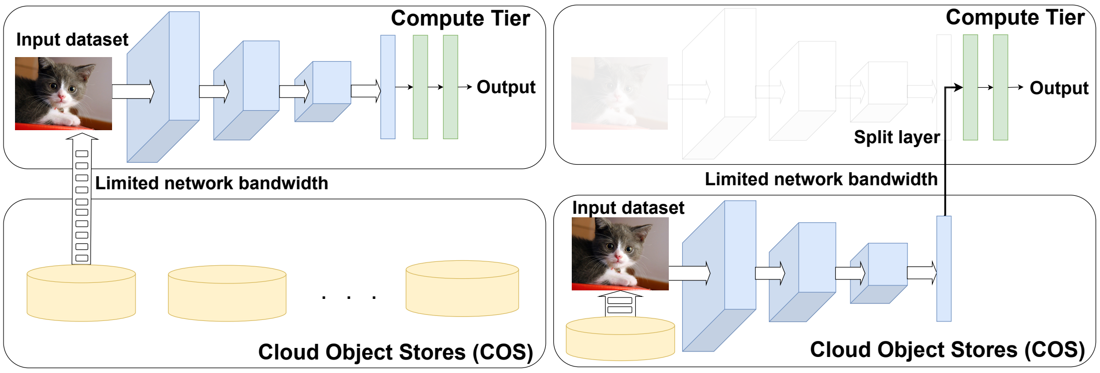
In this work, we show that the feature extraction phase can be pushed down to the COS tier to reduce the training latency and the network traffic between the compute tier and the COS tier while also maximizing the utilization of the COS computing resources such as the GPU memory.

5.3 Measurement Study

We present a detailed measurement study of several TL applications using 4 vision DNNs (AlexNet [142], ResNet18 [104], VGG11 [224], DenseNet121 [111]) and two datasets: synthetic and ImageNet [66]. The synthetic dataset is simply a collection of random tensors with specific shapes.[5] We use it to understand the resource consumption of specific DNNs processing without necessarily looking at how they perform on the optimization side. We characterize the per-layer properties across three different dimensions: output size, computation time, and

---

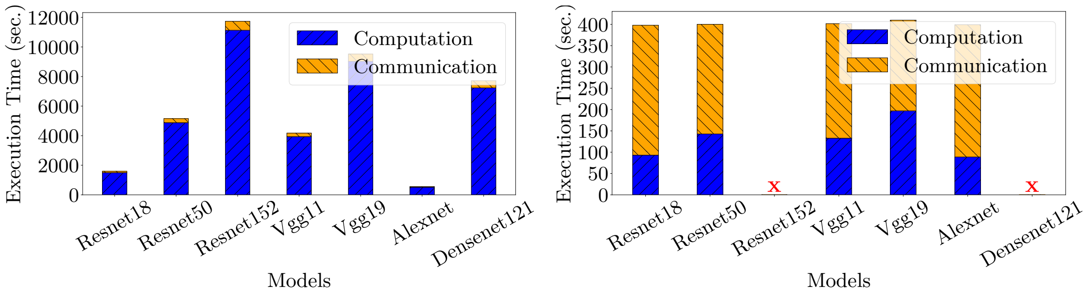
[5] Since we focus on vision applications for this section, we use the standard 224 × 224 × 3 tensor shape.


Input dataset
Compute Tier
Output
Limited network bandwidth
Cloud Object Stores (COS)
. . .
Compute Tier
Output
Split layer
Limited network bandwidth
Input dataset
Cloud Object Stores (COS)

(a) Status quo

(b) Our splitting approach

Figure 5.1: Status quo architecture vs our splitting approach for TL computation.



(a) CPU

(b) GPU

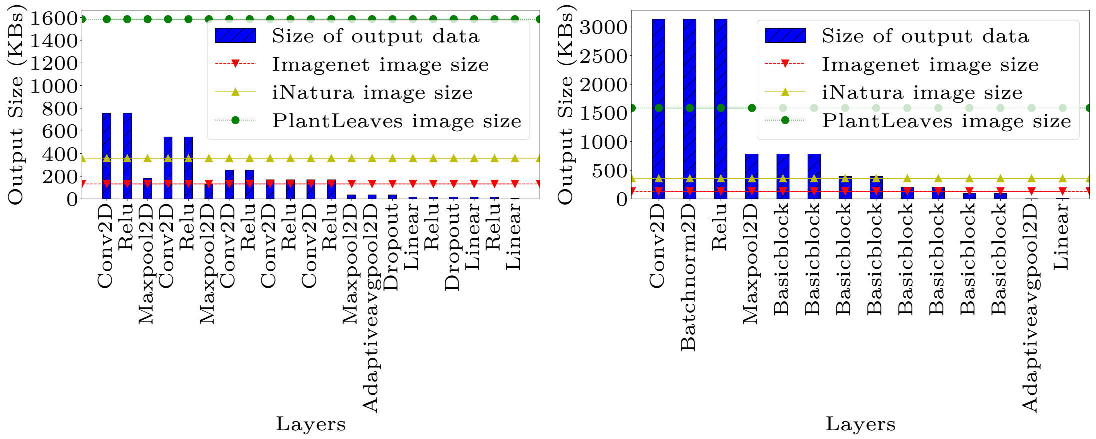
Bandwidth = 150 Mbps (rate limited)

Figure 5.2: Status quo: streaming images from the COS and doing ML training entirely in the compute tier. Dataset: ImageNet. Batch size = 500. Bars marked by 'X' crashed with OOM error.

maximum GPU memory used. Note that although we focus here on vision models only, our study is extensible to other domains as well including speech recognition [98] and language modelling [239]. The details of our methodology are in Section 5.6.1. The insights derived from this study guide our design in Section 5.4.

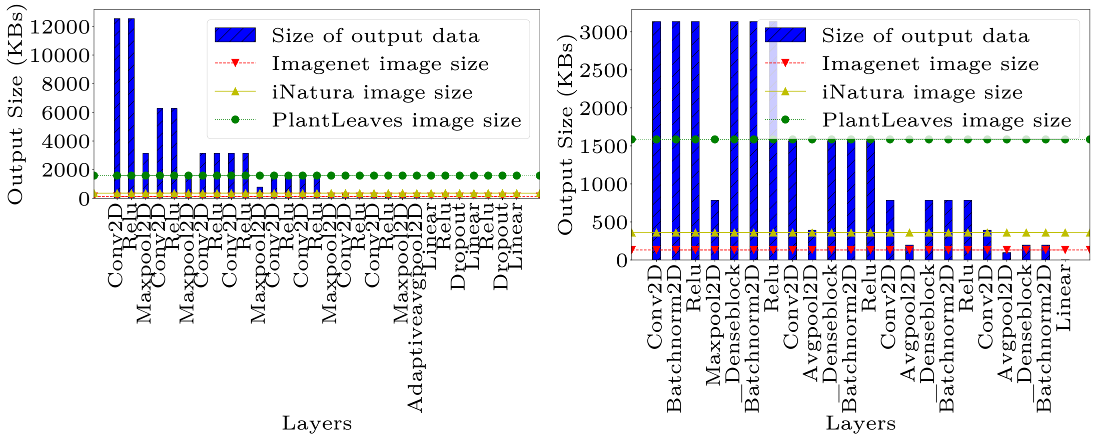
5.3.1 Status Quo

We start by illustrating the limitations of the standard approach of running the ML computation entirely in the compute tier while streaming the input images from the COS (see Figure 5.1a). We run an experiment where we perform training with a batch size of 500 images. For comparison, training is done on either CPU or GPU. To illustrate the impact of the limited network bandwidth between the COS and the compute tier, we show an example where we artificially rate-limit the network bandwidth to 150 Mbps. As it is standard, for performance reasons, the computation of one batch is overlapped with the data transfer for the next batch.



(a) AlexNet

(b) ResNet18



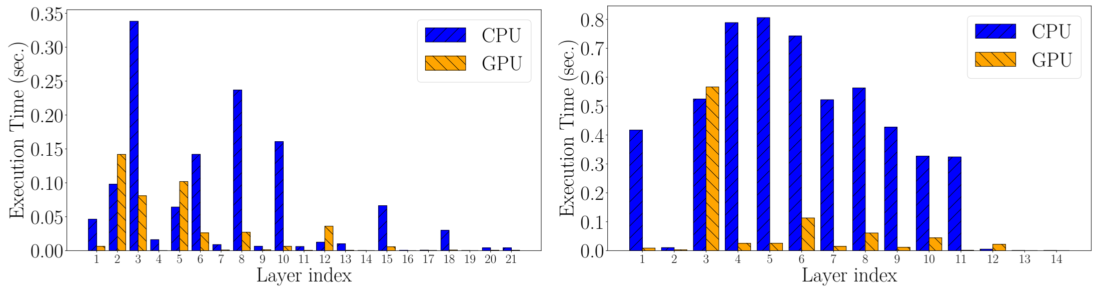
(c) VGG11

(d) DenseNet121

Figure 5.3: Per–layer output sizes for 4 state-of-the-art DNNs. Dataset: Synthetic. Batch size = 1.

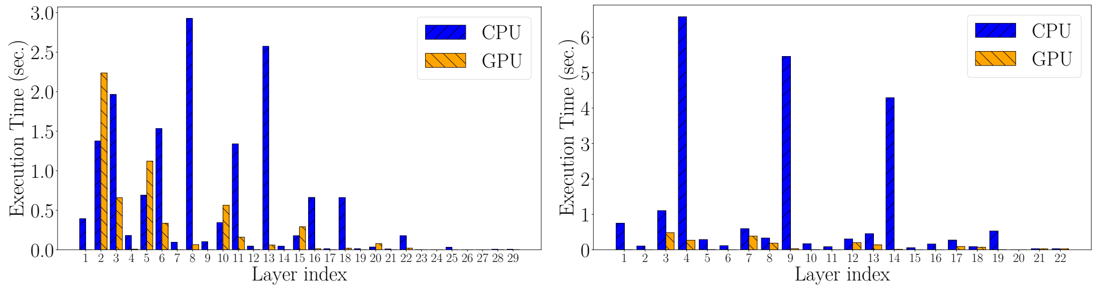
Figure 5.2 shows the communication-computation breakdown. There are several takeaways. First, as expected, training on the CPU is much more time-consuming than on the GPU. Second, the communication is the bottleneck when training with GPUs because the network transfer rate is far lower than the speed at which the GPU can train. A corollary of that is that the GPU is not efficiently utilized, remaining idle while waiting for data, despite having communication and computation overlapped as much as possible. Finally, some of the models crash with out-of-memory (OOM) errors on GPU. Unfortunately, to get around this, practitioners can only choose between a set of sub-optimal solutions: use a smaller batch size which could affect the accuracy, move to more expensive GPUs, or run with slow CPUs.

Based on these insights, we conclude that we need a solution that alleviates the network bottleneck, improves GPU utilization, and enables fast ML computations even on CPUs. Our main idea is to split the TL computation among the COS and the compute tiers, as depicted in



(a) AlexNet

(b) ResNet18



(c) VGG11

(d) DenseNet121

Figure 5.4: Computation time of the forward pass per layer for 4 state-of-the-art DNNs on both CPU and GPU. Dataset: Synthetic. Batch size = 200.

Figure 5.1b.

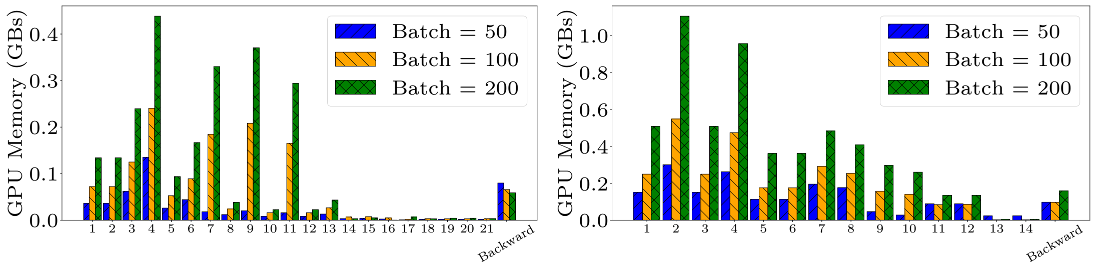
5.3.2 Per-layer Output Size

Figure 5.3 compares the application input size of 3 datasets [66, 236, 223] (the horizontal lines) with the output size of each layer (the bars). The figure shows a batch size of 1, essentially representing one input image as it gets transformed through the DNN. One can accurately extrapolate [148] from this result to any batch size by simply multiplying by the batch size.

Two main insights can be derived from the figure. First, the layer output size generally increases in the beginning (typically with convolution layers) and then decreases (typically with pooling layers). A corollary of that is, for all models, there exist several intermediate layers for which layer output size is smaller than the application input size. Second, the decrease in the intermediate layer output size is not always monotonic. Thus, we can find early on, in the DNN structure, layers that have an output size smaller than the application input size. These layers are good candidates for splitting the TL application between the COS and the compute tiers.

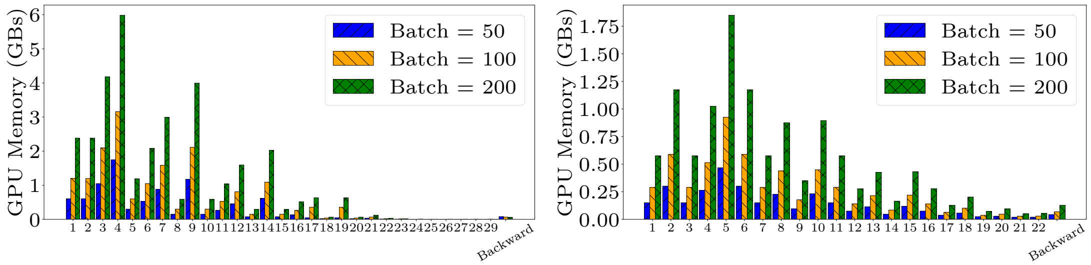
5.3.3 Per-layer Computation Time

Figure 5.4 shows the per-layer computation time for the forward pass on CPU and GPU. There are several insights. For all models, earlier layers are more computationally expensive. The



(a) AlexNet

(b) ResNet18



(c) VGG11

(d) DenseNet121

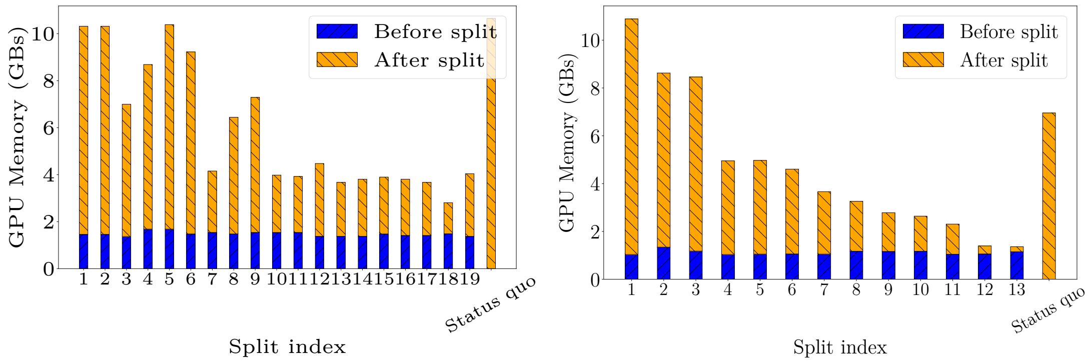
Figure 5.5: Maximum GPU memory used for the forward pass per layer with 4 state-of-the-art DNNs. For the backward pass, the maximum GPU memory is shown across all the layers starting from the indexes in Table 5.2 until the end of the network. Dataset: Synthetic.

pattern of the variation is model-dependent. Second, as expected, the computation on the CPU generally takes significantly more time. The important exception is the last few layers, where there is very little difference between runtime on CPU and GPU. Even more, for some of the later layers, executing on the CPU is more efficient. This suggests that even clients with a low-end hardware configuration can contribute by executing the last part of the DNN.

5.3.4 Batch Size vs Per-layer Memory Usage

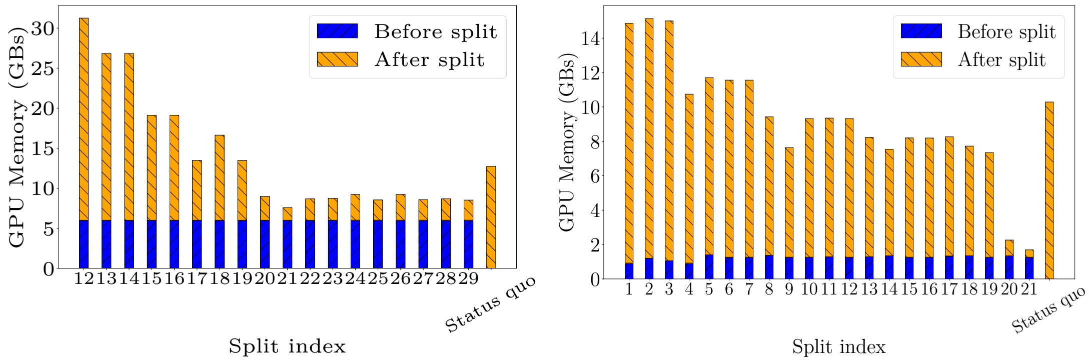
Figure 5.5 shows the maximum amount of utilized GPU memory per layer for the forward pass over all layers in the deep network. It also shows (right side of each sub-figure) the maximum amount of memory used across all the layers participating in the backward pass. For each model, the backward phase ends at the corresponding freeze index listed in Table 5.2. For the forward pass, the memory is mainly composed of the intermediate outputs. For the backward phase, the results are aggregated because intermediate outputs from all participating layers need to be kept in memory until the phase finishes. For the backward pass, the memory is composed of intermediate outputs and the computed gradients [107].

There are three main insights. First, given the same batch size, the first layers generally use more memory. Second and related, an increase in batch size causes a much larger increase in



(a) AlexNet

(b) ResNet18



(c) VGG11

(d) DenseNet121

Figure 5.6: GPU memory breakdown with splitting the forward pass at different layers of 4 DNNs (x-axis). Before splitting (the blue bars), we use a batch size of 100 while after (the orange bars), we use 1000. We compare this to a no-split case (i.e., status quo) in which we use batch size of 1000. With VGG11, we experiment with split indexes whose output size is smaller than the largest input shown in Figure 5.3. Dataset: ImageNet.

per-layer memory usage in the first layers suggesting these layers should be the focus if we want to reduce overall memory consumption. Third, given different batch sizes, the first layers can actually become cheaper (in terms of memory consumption) than the backward phase, illustrating the potential benefits of adapting the batch size for the first layers. Figure 5.6 shows an example. In this experiment, we split the forward pass into two parts: before the split index, we use a batch size of 100, and after, we use a batch size of 1000. We measure the GPU memory with different split indexes. The main takeaway from the figure is that combining a small batch size before the split with a later split index can greatly reduce the total GPU memory usage, sometimes below that of the status quo (Section 5.3.1), i.e., with no splitting.

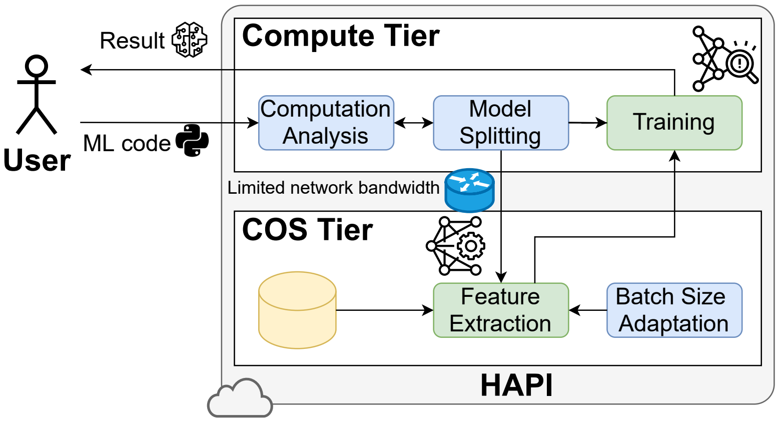


Figure 5.7: HAPI - System Overview.

5.4 The HAPI Design

5.4.1 Insights

Our measurement study (Section 5.3) shows that DNNs have a general structure that can be leveraged to make the TL computation faster. We next present HAPI, our processing system for TL. The design of HAPI relies on two main insights.

First, the feature extraction phase (i.e., the first few layers of the DNN) is typically more demanding – in terms of (a) execution time, (b) GPU memory, and (c) output size – than the training phase (the later layers of the DNN). Hence, pushing down, partially or entirely, feature extraction next to the COS (while running training on the compute tier) reduces the amount of data transferred over the network.

Second, splitting the TL computation enables decoupling the batch size of feature extraction from the training batch size. This decoupling reduces the memory requirement of the TL computation and helps manage more efficiently the scarce and expensive GPU memory of the COS, allowing for concurrent users to better share the COS's GPUs.

Limitations of pushing everything in the COS. One could envision pushing down both phases of TL, feature extraction and training, next to storage. This solution, which we call ALL_IN_COS, reduces the network traffic to the minimum possible, i.e., no data will be required to transfer during training; only at the end, the user might want to download the trained model from the COS. Yet, this solution fails to decouple the batch size of feature extraction from that of training, leading to a choice between unsatisfactory options: limiting concurrency on the COS, leading to poor throughput, or running the risk of OOM errors. We experiment with this approach in Section 5.6.4, showing that it does not scale with multiple concurrent requests.

5.4.2 System Description

We next describe the high-level architecture of our system, HAPI, which is also depicted in Figure 5.7.

Terminology. The DNN layer at which HAPI decides to split the DNN is called the *split index*. The *freeze index* is the DNN layer that separates feature extraction from training, and it is chosen by the user. The *training batch size* is the batch size specified by the user, and it is used in the training phase. HAPI might decide to use a different batch size for computation on the COS (Section 5.4.5); we call this the COS *batch size*. We use *HTTP POST request* to refer to a request sent from the HAPI client to the HAPI server. We use *storage request* to refer to a request sent by the HAPI server to the storage nodes.

At the high level. HAPI is composed of two main components: the HAPI client, which runs on the compute tier, and the HAPI server, which runs in the COS. We assume a COS design similar to that of open source systems like OpenStack Swift [189]. This design differentiates between COS proxy and COS storage nodes. The HAPI server runs on the COS proxy. There is a fast network between the COS proxy and the COS storage nodes, which host the DNNs and the dataset. However, the network between the compute tier and the COS has limited bandwidth.

Request flow. Figure 5.8 depicts an example of the flow of one request. To initiate a TL computation, users provide their application to the HAPI client, which extracts the application configuration (i.e., the model type, the dataset, the freeze index, etc.) and splits (Section 5.4.4) the DNN (Step (1)) into 2 parts: one to be executed on the COS and the other to be executed on the compute tier. This splitting decision happens once per TL computation before its start.

Subsequently, for each training iteration (i.e., the processing of a training batch size of data), the HAPI client sends to the HAPI server HTTP POST requests (Step (2)) containing the necessary information: split index, model type, and the name of the object that stores the corresponding data batch. Several such POST requests may be sent during one iteration because one POST request is needed for each object on storage. Therefore, the number of POST requests depends on the ratio between training batch size and the size of the objects on storage (we assume fixed-sized objects).

When the HAPI server receives a request, it first reads the object that holds the training data and the specified DNN by sending a *storage request* to the COS storage nodes (Step (3)) and then executes the feature extraction part up to the split index. Note that the HAPI server chooses, at will, the COS batch size to be used for feature extraction (Section 5.4.5) depending on the availability of its GPU memory and the concurrent (or to-be-shortly-served) POST requests (Step (4)).

After finishing the feature extraction portion assigned to it, the HAPI server sends back the outputs of the split index layer to the HAPI client, which then uses these outputs to continue the TL computation on the compute tier. Note that the client uses the training batch size for

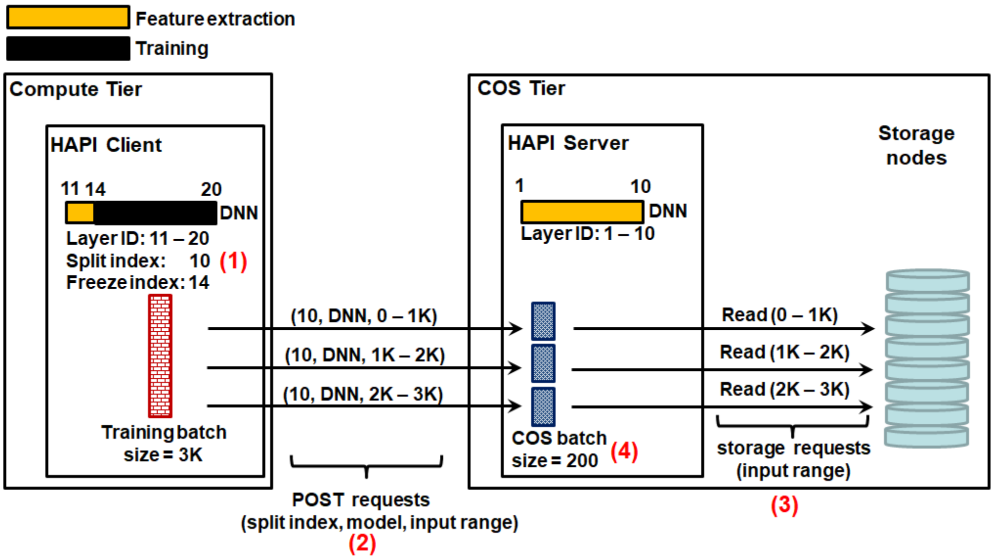


Figure 5.8: HAPI request flow. A TL application is split at index 10. This shows a training iteration with training batch size 3000. This results in 3 requests being sent to the HAPI server. On the COS, for each of the 3 requests, feature extraction is performed with a COS batch size of 200.

its entire computation even while executing the last part of feature extraction (if any).

Observations. First, HAPI only executes feature extraction next to storage in the COS. Consequently, the split index will be always smaller than or equal to the freeze index (i.e., the split is always before the training phase). Second, the COS batch size is upper bounded by the training batch size as the latter defines the value requested by the user, which we cannot exceed. Nonetheless, the client can pre-fetch more data (via concurrent POST requests) than the training batch size to improve training performance. Third, scaling down the COS batch size does not affect the convergence of TL as it is used only for feature extraction rather than training. Fourth, since we limit the size of the POST request, these requests are lightweight and can be executed fast on the COS's GPUs.

Reasoning behind the design. Our lightweight POST request design contributes to system scalability and avoids head-of-line blocking situations where lightweight requests are stuck behind expensive ones. The HAPI server is stateless: different POST requests, even those coming from the same TL computation, are treated independently. Effectively, the HAPI server does not keep a history of executed computations, and it does not keep DNNs, which were used in previous computations, in memory. Although this decision adds the overhead of loading DNNs multiple times to the GPU memory, it makes the server's fault tolerance easier and makes distributing requests to multiple proxy nodes straightforward. Scaling the system would merely require adding or removing independent HAPI servers on the COS proxy nodes. Finally, we note that feature extraction is a standard process that is typically deterministic and predictable in terms of memory consumption and computation time. These properties allow the operator to accurately predict the computation behavior, a desirable property for any computation pushed down in the COS [222].

5.4.3 Efficient Client-side Profiling

The HAPI server and client rely on a profile of per-layer output sizes. In addition, the server relies on a profile of the per-layer memory consumption. In HAPI, the profiling is performed at the client, once per application, before execution starts.

The memory profile is composed of: (1) model weights, (2) input data, and (3) intermediate layer outputs generated during the computation. The last two grow proportionally with the batch size, unlike (1), which remains constant.

For profiling, the HAPI client runs a forward pass with a synthesized data sample (i.e., with the same dimensions as input data), using the training model and keeping track of the per-layer memory consumption and the size of the intermediate output. Crucially, only one data sample (i.e., batch size of 1) is sufficient, and hence, this step is very lightweight both in terms of time (few ms) and memory (few MBs). The HAPI client sends the profiling results to the HAPI server in the specifications of the workload inside every POST request.

The HAPI server estimates the memory necessary for its partition of the computation by multiplying the profiled values by the COS batch size, which it will calculate (Section 5.4.5). Multiplication is an accurate strategy in most cases, as can be inferred from Figure 5.5. We also show that this approach effectively captures the actual required memory (and therefore prevents OOM errors) in Section 5.6.

The benefit of our strategy is three-fold. First, it is cheap and lightweight. Second, it is agnostic to the training model as well as to the DNN layers. Third, it does not rely on pre-computed models, which might be based on specific models of hardware or software versions and hence, inaccurate.

5.4.4 Splitting Algorithm

The HAPI client runs the splitting algorithm (shown in Algorithm 2) once per application. The algorithm comprises two phases. The first is the candidate selection, which is guided purely by model properties. The second phase, the winner selection, selects one of the candidate layers and is guided by properties of the environment, namely the network bandwidth to the COS.

The first phase works as follows. Based on the intermediate output sizes estimation step (Section 5.4.3), the HAPI client chooses layers whose output size is smaller than the input size (scaled by the batch size). The rationale is simple: the main goal is to reduce network traffic compared to sending the entire application input to the client.

Algorithm 2 Choose the split index

```
procedure PROFILE_MODEL(model, input_size)
    one_point=random_tensor(input_size)
    intermediate_sizes = forward(model, one_point)
    model_size = get_size(model)
    return intermediate_sizes, model_size
end procedure
procedure CHOOSE_SPLIT_IDX(model, input)
    input_size = size(input)/len(input)
    intermediate_sizes, model_size=profile_model(model, input_size)
    // candidate selection phase
    potential_layers = model.where(intermediate_sizes < input_size)
    // winner selection phase
    C = read_network_bandwidth()
    for each layer $l$ in potential_layers do
        if intermediate_sizes[$l$] < C then
            winner = intermediate_sizes[$l$]
            break
        end if
    end for
    if no winner then
        winner = freeze_idx
    end if
    return winner
end procedure
```

The second phase, winner selection, is a dynamic approach that navigates the trade-off between two potentially opposing needs: (1) to push down as few layers as possible to save COS resource while (2) reducing the time spent in network communication to improve user-perceived latency. The key to the success of the algorithm is our insight from Section 5.3.2, namely that the layer output size decreases in general but non-monotonically. Hence, it is possible to find layers early in the DNN with comparatively small output sizes. To best navigate this trade-off, the algorithm chooses the earliest candidate layer with an output size lower than C, where C is a function of the network bandwidth, essentially trading off an optimal splitting point, with respect to network transfers, for reduced pushdown to the COS. In our experiments, we found that a good value for C is network bandwidth times 1s.

To better understand the dynamicity of the algorithm, consider that if the network bandwidth is abundant, the algorithm tends to choose an earlier split point with a comparatively larger output. With limited bandwidth, the split index moves towards later layers. The training batch size has a similar influence. The larger the batch size, the larger the layer output sizes. In this case, the algorithm tends to choose a later split index to compensate since later layers tend to have comparatively smaller output sizes. In all cases, the split index layer is upper bounded by the freeze index layer as we do not push down any part of the training phase to the COS.

5.4.5 Batch Adaptation Algorithm

This algorithm repeatedly runs at the HAPI server. A new run of the algorithm is triggered when two conditions hold: (1) there is available GPU memory for new requests, and (2) there exists at least one queued request that has not yet been accounted for in the previous runs of the algorithm. Because of the HAPI client's design, the HAPI server likely receives several requests in quick succession. Thus, the HAPI server waits for new requests for a small amount of time, a small fraction of the time needed to serve one request. This approach navigates the following trade-off. If the server delays the start of the algorithm too long, this might unnecessarily delay requests. However, if the server does not wait enough, arriving requests might have to wait for the current batch to finish processing (when there is insufficient memory). The algorithm takes into account the already-running requests (i.e., at the time of applying the algorithm) but not the future requests. The goal of the algorithm is to maximize the GPU memory utilization over the existing requests while fitting as many of them inside the GPU memory. The output of the algorithm is the batch size to be used for each request, i.e., the COS batch size. Note that it is safe to change the batch size on the server side (i.e., on storage) since processing on this side includes feature extraction (partially or fully) only. This processing entails only a forward pass with no actual training, and hence this processing is deterministic. Based on that, changing the batch size at this stage does not affect the convergence of training.

To choose the batch size for each request, the HAPI server solves the following optimization problem:

$$\begin{aligned}
\max \quad & \sum_{r \in \mathscr{R}} b_r \times \mathscr{M}_r(\text{data}) + \mathscr{M}_r(\text{model}) \\
\text{s.t.} \quad & \forall_{r \in \mathscr{R}} \;\; b_{r_{min}} \leq b_r \leq b_{r_{max}}, \\
& \sum_{r \in \mathscr{R}} \;\; b_r \times \mathscr{M}_r(\text{data}) + \mathscr{M}_r(\text{model}) \\
& \leq \mathscr{M}_{\text{total}} - \mathscr{M}(\text{occupied}),
\end{aligned} \tag{5.1}$$

where $\mathscr{R}$ is the set of requests in the queue, b_r is the batch size to be used for request r (i.e., the decision variables of the optimization problem), $\mathscr{M}_r(\text{data})$ is the amount of memory occupied by both the input and the intermediate outputs of the DNN model for request r, $\mathscr{M}_r(\text{model})$ is the amount of memory occupied by the DNN model weights for request r, $\mathscr{M}_{\text{total}}$ is the total amount of the GPU memory, $\mathscr{M}(\text{occupied})$ is the amount of memory occupied by other already-running requests in addition to estimation for the reserved memory for Cuda and the ML framework,[6] and $b_{r_{min}}$ and $b_{r_{max}}$ are the minimum and maximum bounds allowed for the batch size. Note that $b_{r_{max}}$ is set by the HAPI client (typically, same as the training batch size) while sending the request, whereas $b_{r_{min}}$ is set by the operator. In our experiments, we set $b_{r_{min}}$ to 25 as we observe that using a smaller batch size would lead to unnecessary overhead.

We now make a few notes about our algorithm. First, the HAPI server distributes requests evenly on the existing GPUs. The batch adaptation algorithm runs separately for each GPU on the COS. Second, since all requests look independent to the server (even if they are coming

---

[6] Note that this amount of memory is estimated by the operator apriori, i.e., before starting the server

from the same user), requests from the same user might be assigned different COS batch sizes. Our experiments show that in such small-scale ML computation (i.e., only forward pass with a limited amount of data), the batch size does not have a visible effect on the total execution time. Finally, if the server cannot find an optimal solution for the optimization problem, it removes one request from the list and tries again. The server keeps doing the same until a solution is found. The removed requests contribute to the next batch assignment round, typically after some of the existing requests finish their computation.

5.5 The HAPI Implementation

We next give details on HAPI's implementation. We use OpenStack Swift [189] as our COS, and we use PyTorch [193] for ML computation on both client and server sides.

The HAPI server. We have built HAPI's server inside the Swift's proxy server. In particular, in addition to the batch adaptation algorithm, we implemented functions to do the ML computation. When a POST request is received, the server checks first if it is an ML request. In that case, it reads the object that holds the training data from the storage nodes and then passes it to the appropriate ML function. On the ML side, we created custom DNN models that can run the forward pass between arbitrary start and end layers. Using these models, we are able to split computation at any arbitrary layer.

Concurrency limitations in Swift. Our first implementation of the server works well with one request, but we observe that completion time suffers greatly with concurrent requests. The reason for this lies in a key design of Swift. Essentially, Swift uses a *green threading model*, which allows for concurrency only at the user level. Hence, from the OS's point of view, each Swift process runs in a single thread. Since the number of processes spawned by Swift to serve users' requests is very limited, this design limits the number of TL requests that can run simultaneously, limiting the system scalability.

To overcome this issue, our final implementation decouples the HAPI server from the Swift proxy server (yet both run on the same machine). The HAPI server still reads the training data from Swift's storage nodes. The main advantage of this implementation is that it enables higher concurrency levels. Table 5.1 shows the execution time of the TL computation with 4 DNNs with both implementations. We find that the decoupled implementation improves the application runtime. Lastly, the decoupled implementation has the advantage of not being bound to a specific COS design. In our evaluation (Section 5.6), we use the decoupled version.

Model	ResNet18	ResNet50	Alex	DN121
In Proxy	348	384	308	487
Decoupled	331	324	307	411

Table 5.1: The execution time (in seconds) of one request with 4 DNNs using both versions of HAPI's server implementation: (1) inside Swift's proxy and (2) decoupled from it.

The HAPI client. Apart from choosing the split index, the HAPI client runs the user's TL code. We identify two parts of the vanilla training code that needs to be changed to be compatible with HAPI. First, we use our custom models to enable starting computation from an arbitrary layer instead of the default first layer. Second, we stream the intermediate outputs (of the last layer executed on the COS) using HTTP POST requests instead of streaming the raw training data (typically with HTTP GET requests). Users provide the exact same training parameters in both cases, with status quo and HAPI, and hence, the whole process is transparent to them.

5.6 Performance Evaluation

We evaluate HAPI with multiple state-of-the-art DNNs using ImageNet as a dataset. We first give our methodology and then present the results in detail.

5.6.1 Methodology

Hardware setup. We use two identical GPU-accelerated machines from a public cloud: one runs the compute tier and the HAPI client, and the other hosts the COS and the HAPI server. Each machine has an Intel Xeon Gold 6278C CPU with 16 cores, 64 GBs RAM, 2 Tesla T4 GPUs (with 16 GB RAM each), and a 300GB SSD. Both machines run Ubuntu 20.04 server 64bit with Tesla Driver 460.73.01 and CUDA 11.2. The network bandwidth between both machines is around 12 Gbps, as measured by *iperf3*.

Model	Freeze Layer	Number of Layers
AlexNet	17	22
ResNet18	11	14
ResNet50	20	22
VGG11	25	28
VGG19	36	45
DenseNet121	20	22

Table 5.2: Models used in our evaluation of HAPI. Note that we count only layers at which splitting would make sense. For example, ResNets are structured as a sequence of blocks (Figure 5.3b); we count each block as one layer.

Applications. We use image classification as our DL application due to its wide use in academia and industry. We train various model families of different sizes, including AlexNet [142], ResNet [104], VGG [224], and DenseNet [111]. The complete list, along with the default freeze layer, is shown in Table 5.2. We use ImageNet [66] as our dataset. Unless otherwise stated, we use AlexNet as our default training model.

Metrics. We use the following metrics for evaluation.
1. *Execution Time.* The end-to-end latency of executing one epoch of training. Our computation shows that running training for multiple epochs only multiplies this value by a factor.

2. Transferred Data. The average amount of data transferred over the network (i.e., between the COS and the compute tier) per training iteration. Note that one iteration might comprise multiple POST requests to the COS. In addition, one epoch usually comprises multiple iterations.
3. GPU Memory Consumption. The maximum amount of GPU memory occupied during the execution. We measure this metric periodically using *nvidia-smi*.

COS configuration. We use OpenStack Swift as our COS. We configure it to run one proxy server with 3 replicated disks. The number of *workers* (i.e., the number of *green threads* accepting requests) is set to *auto*, so it maps to the number of cores in the system, which is 16.

HAPI configuration. We divide our dataset into equal-sized chunks of 1000 images, each chunk being stored as one object in the COS. We choose this object size following recommendations [137] that suggest avoiding small requests to maximize COS throughput. We set the POST request size to 1000. This has 2 implications: (1) the smallest training batch size we experiment with is 1000, and (2) the number of parallel requests sent by the HAPI client to the COS is equal to the training batch size divided by 1000. Unless otherwise stated, the default COS batch size is 200. Our batch adaptation algorithm may decide to change this value. We set the minimum COS batch size to 25 because on our hardware, smaller batch sizes add unnecessary overhead. Unless otherwise stated, we limit the bandwidth between the HAPI client and the COS to 1 Gbps. Such value reflects the average bandwidth one can get from today's COS [137, 198]. Nevertheless, we analyze the impact of changing the network bandwidth in Section 5.6.3.

Baseline. To the best of our knowledge, HAPI is the first system to split ML computation between the compute tier and the COS. We compare HAPI to a BASELINE that runs solely on the compute tier by streaming first as many images as the training batch size from the COS and then applying the whole training locally. We pipeline streaming data with computation.

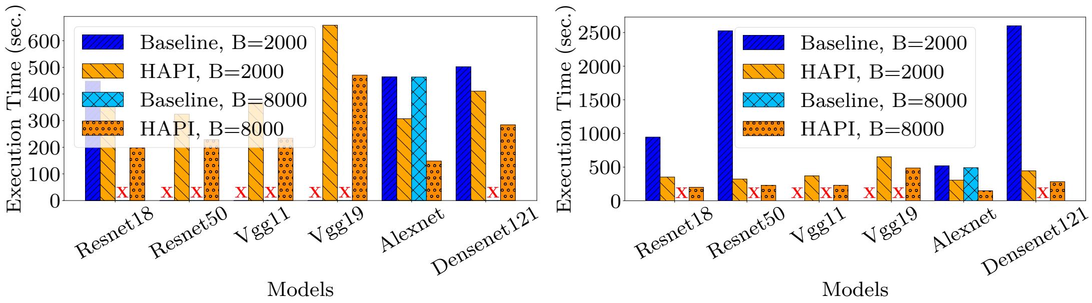
Summary of the findings:

1. HAPI improves the execution time compared to BASELINE on both GPUs and CPUs (Section 5.6.2).

2. HAPI shows benefits across a range of network bandwidths, alleviating the network bottleneck and allowing for faster computation (Section 5.6.3).

3. HAPI scales well with up to 10 requests, achieving considerable throughput gain compared to executing the computation entirely on the COS (Section 5.6.4).

4. HAPI reduces the data transferred over the network, upper bounding it even with excessively large training batch sizes (Section 5.6.5).



(a) GPU (b) CPU

Figure 5.9: Execution time of Baseline compared to HAPI with different DNNs. Bars marked by 'X' crashed with OOM error.

5. The batch adaptation algorithm of HAPI is able to utilize 100% of the GPU memory of the COS while preventing effectively OOM errors (Section 5.6.6).

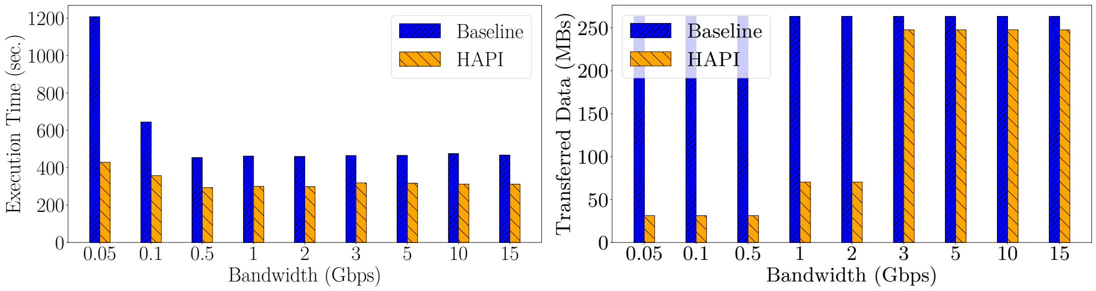
5.6.2 End-to-end Experiment

Figure 5.9 shows the end-to-end execution time of HAPI and Baseline with 2 training batch sizes and with different models. We experiment with a strong client (i.e., equipped with 2 GPUs) and a weak client that has access only to a CPU.

The first observation is that Baseline fails to complete the execution in many cases due to out-of-memory (OOM) errors. Even with 64 GBs of memory, Baseline fails to run even the smallest version of VGG networks with a batch size of 2000 (see Figure 5.9b). With a batch size of 8000 (which is the batch size required to train ImageNet for the best accuracy [88]), Baseline manages to run only AlexNet. This does not happen with HAPI for 2 reasons: (1) the batch adaptation algorithm on the COS reduces the batch size for feature extraction when it realizes the computation will not fit in the available memory. (2) The actual training part (that happens in the compute tier) requires very little memory, and hence no matter how big the training batch size is, the memory used never reaches the upper limit.

Using the same batch size, HAPI outperforms Baseline in all cases. Precisely, HAPI achieves an average execution time speedup of 2.24× (on GPU) and 3.91× (on CPU) and up to 3.13× (on GPU) and 7.83× (on CPU) over Baseline. Interestingly, in some cases, a weak client that runs HAPI can outperform a strong client that runs Baseline, a direct benefit of HAPI's splitting algorithm. For example, training ResNet18, with batch size 2000, on CPU with HAPI takes 352s, whereas on GPU with Baseline it takes 448s.

Since HAPI is compute-bound, an additional execution time speedup can be seen with increasing the batch size. For example, the execution time of HAPI drops from 307s to 148s with increasing the batch size from 2000 to 8000 while training AlexNet on GPU. Interestingly,



(a) Execution Time

(b) Transferred Data

Figure 5.10: The effect of varying bandwidth on HAPI.

Bandwidth (Gbps)	0.05	0.1	0.5	1	2	3	5	10	15
Split Index	17	17	16	13	13	5	5	5	5

Table 5.3: The chosen split index by HAPI with different values for the bandwidth (see Figure 5.10).

we cannot make the same observation for BASELINE. Indeed, increasing the batch size has seemingly no effect on the execution time. The reason for this lies in the fact that the bottleneck for BASELINE is the network, and hence speeding up the computation (by increasing the batch size) does not help improve the end-to-end execution time.

5.6.3 Impact of Network Bandwidth

Figure 5.10 depicts the execution time and the amount of data transferred with different values of network bandwidth between the compute tier and the COS. Figure 5.10a shows that HAPI outperforms BASELINE in all cases, even with abundant bandwidth (1.5× faster). HAPI performs even better (2.8× faster) in extreme cases where the bandwidth is as little as 50 Mbps. HAPI manages to have almost a flat curve along with different values for the bandwidth.

The key to this behavior lies in controlling the amount of data transferred in each case, which is shown in Figure 5.10b. Up to 2 Gbps, HAPI chooses to transfer less than 100 MBs per iteration, where BASELINE transfers more than 250 MBs per iteration in all cases. With enough bandwidth (3 Gbps or more), HAPI adapts and transfers almost the same amount of data as BASELINE. The main knob in such an adaptation is the split index. Note that, in the case of HAPI, the transferred data are the output of the last layer executed in the COS. The size of such outputs changes dramatically from one layer to another, as shown in Section 5.3.2. HAPI carefully chooses the split index depending on the available bandwidth as shown in Table 5.3. In fact, HAPI chooses the split index to be equal to the freeze layer index (i.e., layer 17) when the bandwidth is limited, whereas it chooses layer 5 (whose output is still smaller than the raw input images) with abundant bandwidth.

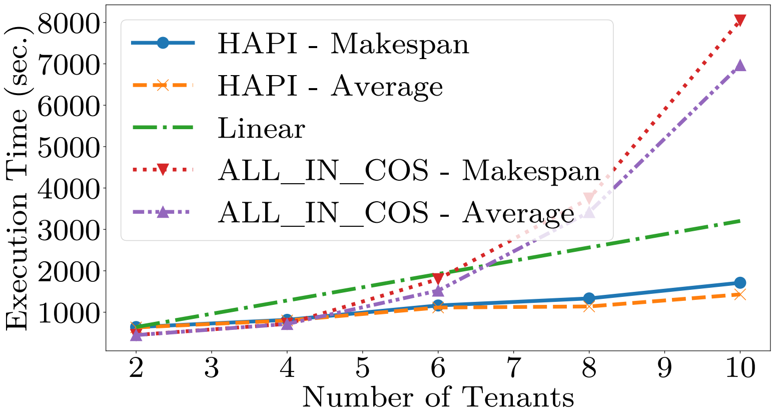


Figure 5.11: Scalability of HAPI with multi-tenants (i.e., multiple concurrent requests) compared to an *ALL_IN_COS* solution that pushes down the entire computation next to storage.

5.6.4 Scalability on the COS

Figure 5.11 shows how HAPI scales with multiple tenants (up to 10). We found that 10 concurrent tenants (i.e., requests) are sufficient to saturate both GPUs of the COS. In this experiment, each tenant submits one request to train a model from Table 5.2. The model is chosen in a round-robin fashion. All requests use a training batch size of 1000 and are submitted at once at the beginning of the experiment. Since the goal of this experiment is to overload the COS, we do not run an actual computation on the client side. Figure 5.11 shows both the makespan (the time it takes to finish all requests) and the average job completion time (JCT). In addition, we compare against the ALL_IN_COS solution discussed in Section 5.4.1, in which the entire TL computation is pushed to the COS.

The figure shows that the server can run up to 6 tenants concurrently, finishing all of them almost at the same time. This is noted from the very small difference between the makespan and average JCT. Yet, the scalability of HAPI is not perfect, as sharing the GPUs on the COS side has an inevitable overhead. Precisely, running concurrently 6 tenants increases the average JCT by 1.7× compared to the case of 2 tenants.

When the COS serves more than 6 tenants, the GPU memory becomes insufficient to fit all of the requests, and therefore, some requests wait in the queue for other requests to finish first, i.e., the optimization problem is not solved from the first trial. Consequently, some tenants finish their computation faster than others. For example, the difference between makespan and the average JCT with 10 tenants is 280s.

Although the ALL_IN_COS solution achieves comparable (if not better) performance to HAPI with a few tenants, it fails to scale with increasing the number of tenants. Compared to the ALL_IN_COS solution, we find HAPI to achieve 2.2× improvement on average in throughput (based on the average JCT) and up to 4.9× improvement with 10 tenants.

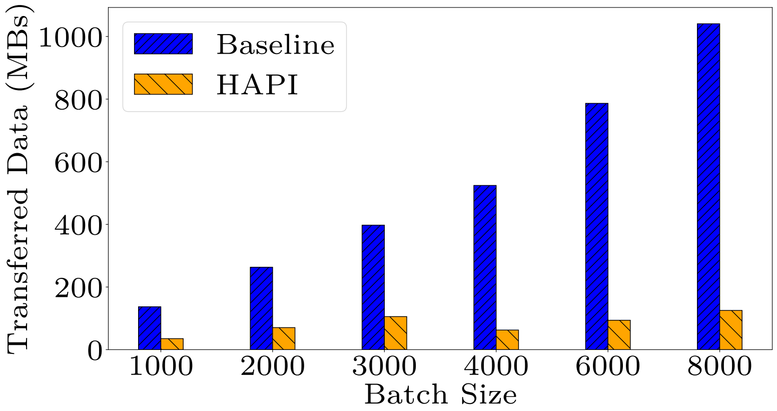


Figure 5.12: Transferred data with HAPI, compared to BASELINE, with different batch sizes. The y-axis shows the average data size transferred in one iteration.

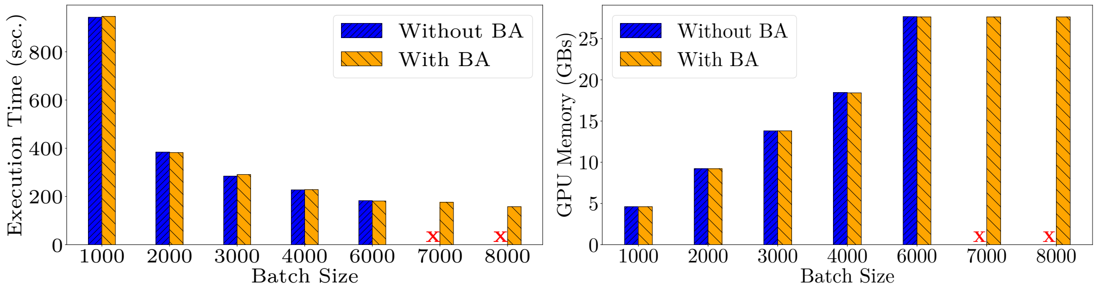
Training batch size (× 1000)	1	2	3	4	6	7	8
% of requests whose COS batch size is reduced	0	0	0	0	1.88	15.38	30
Average reduction in COS batch size	0	0	0	0	1.78	14.51	25.13

Table 5.4: The behavior of HAPI's batch adaptation in the experiment in Figure 5.13.

5.6.5 Reduction in Transferred Data

Figure 5.12 shows the average data size transferred from the COS to the client per training iteration with different training batch sizes. As the training batch size increases, BASELINE streams more data per iteration, and hence, we observe a linear increase in the data transferred with the batch size. This, in turn, increases the execution time linearly, especially when the bandwidth is limited.

On the other hand, HAPI manages to keep the amount of data transferred almost flat with any batch size. The key to this behavior is the adaptive choice of the split index. To give a concrete example, notice the decrease in the amount of data transferred despite increasing the batch size from 3000 to 4000. In fact, with a batch size of 3000, HAPI chooses 13 to be the split index, transferring on average 105.6 MBs per iteration. With a batch size of 4000, it chooses 16 to be the split index, transferring 62.6 MBs per iteration. The logic is that with a larger batch size, more data is expected to be transferred from the COS in each training iteration, assuming the same split index and network bandwidth. HAPI adapts by choosing a later split index for the increased batch size to try to keep the amount of time spent in network communication comparable. This adaptive behavior alleviates the network bottleneck and makes the whole operation compute-bound.



(a) Execution Time

(b) GPU Mem. Consumption

Figure 5.13: The benefits of batch adaptation (BA).

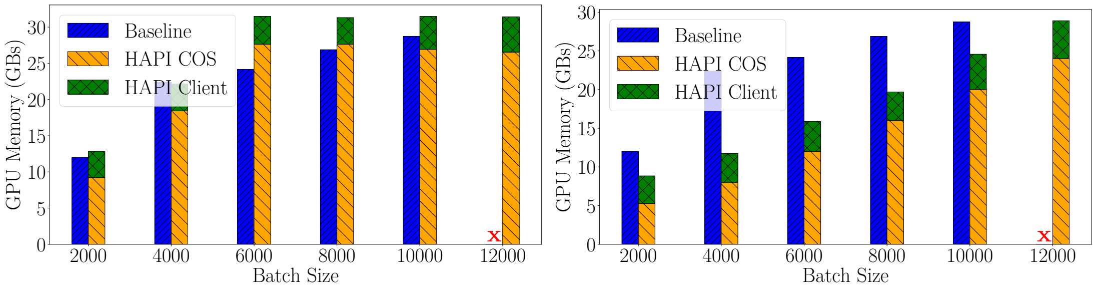
5.6.6 Batch Adaptation

We next discuss the benefits of our batch adaptation algorithm. We run two versions of HAPI: one with batch adaptation (BA) and one without BA. We experiment with several training batch sizes up to 8000. Note that increasing the batch size increases the number of parallel POST requests sent to the COS since we fix the size of each POST request to 1000. For example, using a training batch size of 6000 results in sending 6 parallel POST requests to the COS. In addition, to overload the GPU memory of the COS, we set the COS batch size to 1000. The results of this experiment are shown in Figure 5.13. Table 5.4 complements Figure 5.13 with: (1) the percentage of requests whose batch size is reduced by our algorithm and (2) the average reduction in the batch size.

Figure 5.13a shows the execution time, which generally decreases with increasing the batch size. The reason is that the number of training iterations decreases with increased batch size, enabling faster epochs. Up to 6 requests, HAPI achieves the same execution time in both cases, indicating that the overhead of our BA algorithm is insignificant in the case when there is sufficient memory. For more than 6 requests, the memory becomes insufficient, so our BA algorithm starts reducing the batch size, successfully preventing OOM crashes. The overhead of the BA algorithm remains very small in these cases, which is visible in the fact that the execution time remains small with big batch sizes. Precisely, the server takes around 25ms to execute the BA algorithm.

Figure 5.13b shows the memory usage in each case. On the one hand, when we do not use BA, the GPU memory consumption increases linearly with the batch size until the crashing point due to the lack of resources (OOM error). On the other hand, our fully-fledged HAPI adapts the batch size to fit into the existing memory showing a plateau in memory usage with increasing the batch size. Note that the best memory occupancy we could see is around 28 GBs for both GPUs combined. This constitutes a 100% utilization of the GPU memory, despite knowing that one GPU memory size is, theoretically, 16 GBs. The remaining *unutilized* part of the memory is reserved by CUDA and PyTorch.

Figure 5.13 and Table 5.4 show the importance of scaling down the COS batch size to avoid



(a) COS batch size = 1000

(b) COS batch size = 200

Figure 5.14: Total GPU memory consumption with HAPI compared to BASELINE.

OOM errors. For example, with a training batch size of 8000, HAPI decided to scale down 2-3 requests on average, where the average reduction of the COS batch size is around 25%. Put differently, if the COS batch sizes were 25% less on average (i.e., 750 instead of 1000), all requests would have fit (theoretically) with no need to adapt the COS batch size. Such information is usually obscure and hard to get for the users, leaving them susceptible to choosing a large batch size and crashing the training procedure.

5.6.7 Breakdown of Memory

Figure 5.14 shows the GPU memory consumption by HAPI (including both the compute tier and the COS tier), with 2 COS batch sizes, compared to BASELINE, and using different training batch sizes. Note that the memory consumption of BASELINE and the client side of HAPI is identical in Figures 5.14a and 5.14b; only the memory of the COS changes.

Figure 5.14a highlights the interesting fact that splitting computation could not only fully utilize the GPU memory of the COS but also give the abstraction of extra memory to the user. Precisely, the aggregate GPU memory can exceed 30 GBs in some cases, which is beyond the capabilities of the user in this experiment (as shown in the case of using a batch size of 12000). This allows the user to train their model with arbitrarily large batch sizes. This figure also highlights another source of overhead that appears with sending many parallel requests. In particular, there is a slight decrease in the GPU memory allocated on the COS with more than 8 requests (i.e., with batch size > 8000). The reason for this decrease is the overhead of having *independent* requests running on the same GPU. Since each request is served by a separate process, there is an independent chunk of memory reserved for each one. Hence, having many requests leaves less room for actual data to be allocated on the GPUs. Our memory modeling however manages to capture this overhead, making the best use of the available memory without crashing. Note that this result shows that HAPI is useful not only to improve the execution time, but also to improve the GPU memory utilization. Following this, we believe HAPI can be beneficial even in the cases where the network is not the bottleneck [180], i.e., to improve the memory utilization.

In Figure 5.14b, the COS batch size is small (only 200). Here, we can see that the aggregate memory consumption could be even less than BASELINE. This shows how much the COS batch size knob can control memory consumption.

5.7 Related Work

HAPI contains a unique combination of context (COS), workload (TL), and design decisions (splitting a DNN between cloud tiers, batch adaptation). While related work exists in these directions, HAPI is the first to combine them in a single cohesive system.

Batch adaptation in ML. Several projects [216, 225, 272] proposed to dynamically adapt the *training* batch size as training proceeds, leading to shorter training times without loss in accuracy. These ideas are complementary to HAPI's batch adaptation because they can be applied in HAPI's training phase. Instead, the key insight of batch adaptation in HAPI is that during the same iteration, it is useful to use different batch sizes for training and feature extraction.

Splitting compute between edge and cloud. Neurosurgeon [131] splits ML *inference* between the cloud and edge devices to achieve low latency, low energy consumption, and high data center throughput. The partitioning is automatic and adapts to dynamics in server load and network bandwidth. Compared to Neurosurgeon, besides considering more complex application (i.e., TL), HAPI dynamically manages the concurrency on the storage-side via batch adaptation. MAUI [58] performs fine-grained (function-level) energy-aware offload of mobile code to the infrastructure, via runtime decisions driven by the goal to save energy. MAUI's approach to splitting cannot be directly used in HAPI because it requires programmer annotations, and it transfers all program state including variables. Odessa [204] offloads stages in perception applications but without analyzing the global impact on the application performance.

Offloading compute to storage nodes. Rhea [82] uses static analysis of Java bytecode to generate stateless best-effort storage-side filters for Hadoop applications that read unstructured and semi-structured data. Rhea shares the goal of mitigating the network bottleneck to storage but targets a different workload, does not split an application and does not optimize the resource utilization in the storage layer. Pyxis [49] uses static analysis and dynamic runtime information to automatically partition a database application between application and storage nodes. Pyxis automatically generates database-side stored procedures to minimize overall latency subject to CPU constraints. For this, it minimizes the number of control transfers as well as amount of data sent during each control transfer. Pyxis differs in the application, context, and concerns (i.e., control transfers are specific to database applications).

Computation pushdown in COS. Work on pushing down computation in the COS mainly focuses on how to best leverage the limited subset of SQL supported by Amazon S3 Select-like systems since 2017. PushdownDB [275] analyzes which DBMS primitives can be cost-effectively moved into S3 and on how more complex operations (e.g., joins) can be re-implemented to take advantage of S3 Select. FlexPushdownDB [266] combines the benefits of caching and computation pushdown by introducing a separable operator.

Systems challenges for TL. Cartel [59] provides collaborative TL at the edge by transferring knowledge across geographically distributed edge clouds while minimizing the amount of data sent over the backhaul network. Cartel is different because it focuses on challenges in learning at the edge, e.g., detecting similarities between edge clouds and performing the knowledge transfer.

Model/pipeline parallelism. Splitting in HAPI is reminiscent of model and pipeline parallelism, which aim to distributedly train large models. Model parallelism [50] splits a DNN across workers. This is a form of intra-batch parallelism as all workers process the same batch. Pipeline parallelism [184, 112] further improves the efficiency of model parallelism. Pipedream [184] combines intra with inter-batch parallelism (multiple batches execute concurrently) and mitigates staleness and consistency issues by keeping several versions of the weights. GPipe [112] does not do inter-batch parallelism, but splits a single batch into micro-batches and pipelines those.

HAPI differs in several ways. The high-level goal is different as HAPI aims to mitigate the COS network bottleneck. The approach to splitting is different as the splitting point in HAPI is dynamic but fixed in model/pipeline parallelism. The concerns are different as HAPI's server handles multiple clients concurrently. The challenges are different as there is no backpropagation on HAPI's server or between HAPI's client and server.

Data ingestion pipelines. The feature extraction phase in HAPI is reminiscent of data ingestion/preparation pipelines [182, 280, 120] because of their goal (data preparation) and the critical focus on efficiently delivering data for training. However, these pipelines are far more general in nature (e.g., can even run user-defined functions [182]) and can be very large (e.g., serve data at TB/s [280]), which makes them unsuitable for running inside the COS. In HAPI, the main difference is that the feature extraction phase is part of the DNN which, while limiting the knobs available, allows us to focus on accelerating each separate application.

5.8 Concluding Remarks

In this chapter, we proposed a solution to alleviate the network bottleneck while training machine learning models in the cloud. Specifically, we identified transfer learning (TL) as a natural fit to benefit from the recent evolution in the computational capabilities of disaggregated cloud object stores. We show how to leverage the unique structure of TL to mitigate the cloud network bottleneck and maximize the use of the scarce computational resources inside the object store. The main insight is to carefully split the TL application during or right after feature extraction. This can reduce network transfers and decouples the training and feature extraction phases, facilitating the use of batch adaptation in the object store. We combine these insights into HAPI, a processing system for TL that spans the compute and the object store tiers while remaining transparent to the user. We show up to 11× improvement in application runtime and up to 8.3× reduction in data transfers while utilizing 100% of the GPU memory on the object store.

6 Federated Learning: The Case of Generative Adversarial Networks

In this chapter, we consider training in the federated learning (FL) setup. We tackle a challenging class of ML techniques: *generative adversarial networks (GANs)*. The goal of a GAN is to learn to generate new data with the same statistics as a training set. Existing approaches to distribute GANs either *(i)* fail to scale, for they typically put the two components of a GAN (the *generator* and the *discriminator*) on separate machines, inducing significant communication overhead, or *(ii)* they face GAN training specific issues, exacerbated by distribution.

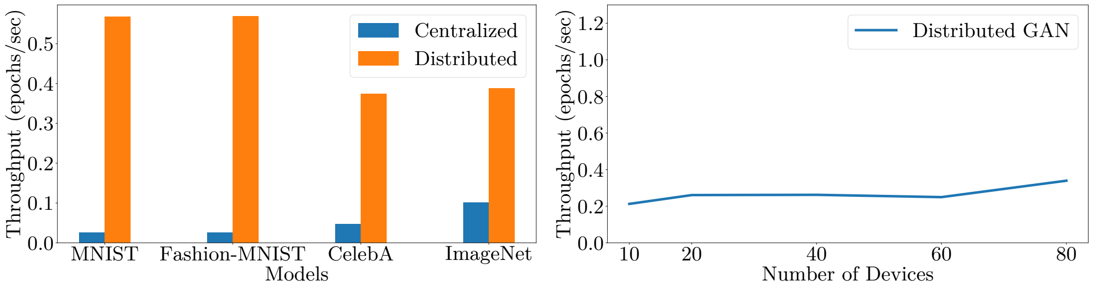
To address these issues, we present FeGAN, a system for distributing GANs over hundreds of devices, addressing three main issues that face training GANs: *mode collapse*, *vanishing gradients*, and *learning divergence*. Essentially, we co-locate a generator with a discriminator on each device (addressing the scaling problem) and have a server aggregate the devices' models using *balanced sampling* and *Kullback-Leibler (KL) weighting* mitigating training issues and boosting convergence.

6.1 Introduction

GANs enable learning the statistical distribution of a target dataset and generating new samples from that dataset on demand. This feature can be used in a wide range of applications, such as generating pictures from text descriptions [209], producing videos from still images [241], or increasing at will an image resolution [153]. Other applications, such as DeepFakes generation [159], are as impressive as critical for society.

At the core of a GAN lie two deep neural networks: the *generator* and the *discriminator*. They confront each other in a game: the generator aims at generating data that looks real (i.e., coming from the real data distribution) to feed the discriminator. The training stops when the latter can no longer distinguish real data from the generated one. The generator has then captured the data distribution and has reached the point where it can generate new samples.

Training GANs is resource and time consuming. For instance, it takes up to 48 hours to train a GAN to learn the distribution of a 512x512 image dataset on Google TPUs v3 devices [39].



(a) (b)

Figure 6.1: (a) A Parameter server-based deployment [101] with one generator and distributed discriminators (on 21 machines) outperforms the centralized approach. Yet, (b) it does not scale due to communication bottlenecks.

Considering the number of potential applications, it is then of paramount importance to improve this training time. One obvious way to scale a system is to distribute the computing load on multiple devices. MD-GAN [101] has been recently proposed as a way to distribute GANs. MD-GAN leverages a single generator (at a central location) and distributes the discriminator across multiple devices. A similar architecture is also adopted by [273]. Such an architecture follows the celebrated *parameter server* model [157], where the server is the generator, and the workers are discriminators. Figure 6.1a illustrates the benefit of such an architecture with 21 machines on four datasets (MNIST [152] and Fashion-MNIST [276], often used as baselines, and CelebA [166] and ImageNet [65], two state-of-the-art datasets for vision applications). Clearly, distributing the training improves the system throughput, defined as the number of epochs (the processing of all data samples in the dataset) the system handles per second. Yet, due to the presence of a *single centralized* generator, **this architecture does not scale**, i.e., adding devices does not improve the throughput, as illustrated in Figure 6.1b. This scalability bottleneck is due to the huge volumes of data to be transferred over the network, essentially to synchronize the generator and the multiple discriminators, making the central generator a bottleneck. Indeed, there is a huge room for improvement here.

Another architecture for distributed training is federated learning (FL) [176], in which training happens on the edge devices that own private data, assisted by a central server. Combining GAN training with this architecture can yield impressive applications on edge devices, including text-to-image translation and generating new human poses. Yet, approaches that followed such an architecture for GAN training (e.g., running diagnostics [23] or for data privacy goals [235]) also reported **difficulties due to learning divergence, vanishing gradients, and mode collapse problems**: *learning divergence* happens when neither the generator nor the discriminator reaches its goal, i.e., Nash equilibrium is not reached [20]. A second salient problem is *vanishing gradients*, which occurs when the discriminator is much more powerful than the generator, which, in this case, always fails to generate convincing samples for the discriminator, where the feedback from the discriminator does not help the generator to learn [185]. Last but not least, the problem of *mode collapse* happens when the generator learns to generate only a few classes of data input rather than learning the true distribution of data [46]. Such problems manifest clearly in the FL context. For instance, the server cannot communicate with *all* devices in *all* rounds for scalability reasons; some form of device sampling is needed. Yet, the sampling strategy should not be uniform because of the data distribution skewness on the devices hosting the GAN model. Unfortunately, such data skewness can easily lead to mode collapse. Thus, the mere application of the FL approach [265, 257, 254] to GANs training is bound to failure [100] or underperformance, as we shall confirm empirically.

To overcome the aforementioned challenges, we introduce FEGAN, a distributed system that enables GAN training to scale and cope with GAN-specific issues such as mode collapse, vanishing gradients, and learning divergence. Our main contributions are three-fold.

1) We revisit the FL paradigm, traditionally dedicated to deep networks, to make it suitable for GANs. Essentially, we fully distribute both the generator and the discriminator so that a private GAN can be locally trained on each device. This distribution scheme helps scale the system and prevent the vanishing gradients problem, as we show in Sections 6.5 and 6.6.

2) We design mechanisms to make FEGAN resilient to GAN-specific issues. In particular, we devise two techniques to resist the mode collapse and the learning divergence problems in a distributed setup. FEGAN prioritizes updates from some devices over others, using *Kullback-Leibler (KL) weighting* scheme, and it carefully schedules the application of devices' updates on the global trained model, using the *balanced sampling* scheme. Both schemes not only boost the learning quality but also help save compute time and communication resources. FEGAN is tolerant to devices' heterogeneity in terms of memory and computing power and server and network failures through periodic checkpointing of the learning state.

3) We conduct an extensive experimental evaluation of FEGAN and compare it to a state–of–the–art GAN distribution approach (MD-GAN), as well as with a centralized approach and a baseline application of federated learning to GANs. Our code is publicly available at https://github.com/LPD-EPFL/FeGAN.

6.2 Background

6.2.1 Generative Adversarial Networks

A GAN [84], made of a generator $\mathscr{G}$ and a discriminator $\mathscr{D}$, targets the learning of a dataset distribution in space X, where $\boldsymbol{x} \in X$ follows a distribution probability P_{data}. Figure 6.2 depicts a typical GAN architecture. The generator is modeled by the function $\mathscr{G}_{\boldsymbol{w}} : R^{\ell} \longrightarrow X$, where $\boldsymbol{w}$ contains the parameters of its neural network $\mathscr{G}_{\boldsymbol{w}}$, and ℓ is fixed. Similarly, for the discriminator $\mathscr{D}_{\boldsymbol{\theta}} : X \longrightarrow [0,1]$, where $\mathscr{D}_{\boldsymbol{\theta}}(\boldsymbol{x})$ is the probability that $\boldsymbol{x}$ is a data from the training dataset, and $\boldsymbol{\theta}$ contains the parameters of the discriminator $\mathscr{D}_{\boldsymbol{\theta}}$. The objective consists in finding the parameters $\boldsymbol{w}^*$ for the generator: $\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \max_{\boldsymbol{\theta}} (A_{\boldsymbol{\theta}} + B_{\boldsymbol{\theta},\boldsymbol{w}})$, with $A_{\boldsymbol{\theta}} = \mathbb{E}_{\boldsymbol{x}\sim P_{\text{data}}} \left[\log \mathscr{D}_{\boldsymbol{\theta}}(\boldsymbol{x})\right]$ and $B_{\boldsymbol{\theta},\boldsymbol{w}} = \mathbb{E}_{\boldsymbol{z}\sim\mathscr{N}_{\ell}} \left[\log(1 - \mathscr{D}_{\boldsymbol{\theta}}(\mathscr{G}_{\boldsymbol{w}}(\boldsymbol{z})))\right]$, where $\boldsymbol{z} \sim \mathscr{N}_{\ell}$ means that each entry of the ℓ-dimensional

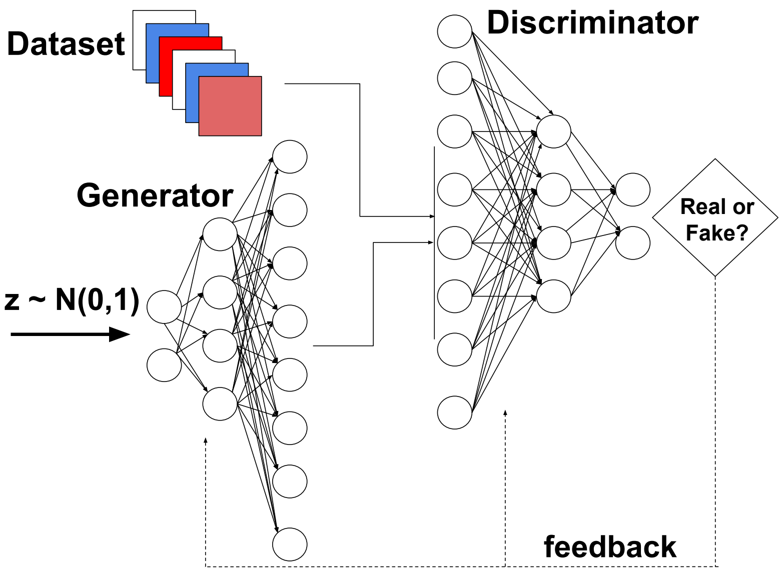


Figure 6.2: A typical GAN architecture. The generator generates data (from a noisy signal) that looks like real data. The discriminator tries to differentiate between real and generated data. Both networks learn simultaneously from the feedback, typically using a backpropagation algorithm [107].

random vector $\boldsymbol{z}$ follows a normal distribution with fixed parameters. In this equation, $\mathscr{D}$ adjusts its parameters $\boldsymbol{\theta}$ to maximize $A_{\boldsymbol{\theta}}$, i.e., the expected good classification on real data and $B_{\boldsymbol{\theta},\boldsymbol{w}}$, the expected good classification on the generated data. $\mathscr{G}$ adjusts its parameters $\boldsymbol{w}$ to minimize $B_{\boldsymbol{\theta},\boldsymbol{w}}$ ($\boldsymbol{w}$ does not have impact on A), which means that it tries to minimize the expected good classification of $\mathscr{D}$ on generated data. This competitive scheme ends, if convergence occurs, to the learning of the dataset distribution P_{data}.

6.2.2 Distributed GANs

One way to distribute GANs training is to follow the celebrated parameter server model (see Section 2.3.1), as in, e.g., [273, 101]. This setup decouples the generator and the discriminator by having a single generator placed on the central server. Discriminators are, however, dispatched on devices and access local data. This setup reduces the processing load on the devices but also results in more network communication in order to maintain the generator-discriminator game. To avoid the mode collapse problem [46], some approaches periodically swap discriminators between devices in a peer-to-peer fashion, essentially to augment the diversity of the data used by the discriminators. In this chapter, we take another route to distribute GANs: we follow the federated learning model (see Section 2.3.3).

6.3 The FEGAN Design and Implementation

In this section, we first provide the system setup and an overview of FEGAN. Then, we focus on our design choices, which make FEGAN scalable and resilient to GAN training issues, and

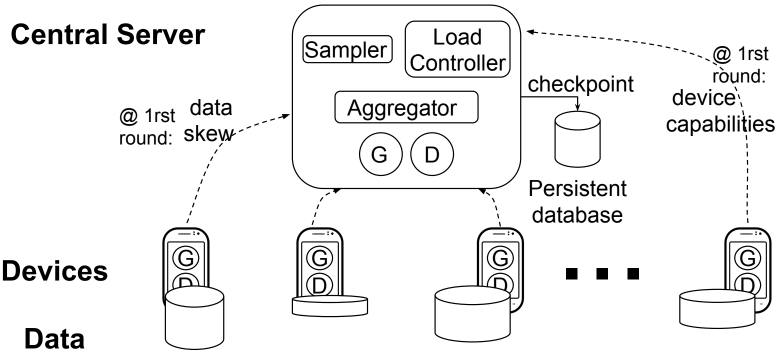


Figure 6.3: FEGAN architecture: $\mathscr{D}$ and $\mathscr{G}$ are co-located on all devices. Dashed lines denote the communication of metadata about devices' datasets and specifications. Data across devices might be unbalanced and skewed.

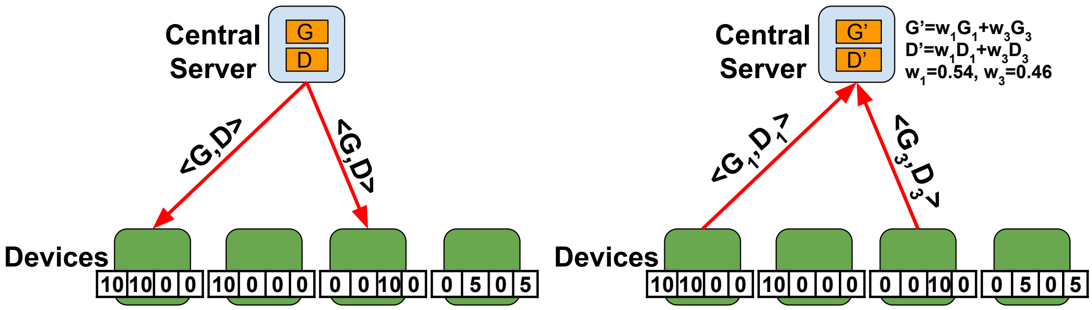
the system aspects of FEGAN.

6.3.1 Overview

FEGAN revisits the general federated learning (FL) paradigm [139], which consists of (1) a central server (hereafter, server) hosting the up-to-date global model and (2) a set of computing devices that can be mobile phones or arbitrary computing nodes (hereafter called devices). Figure 6.3 depicts the FEGAN architecture. The training model contains two neural networks: the generator $\mathscr{G}$ and the discriminator $\mathscr{D}$. The server orchestrates the communication load by selecting which devices should contribute to updating the model at a given round. In each round, the server chooses a constant fraction C of all the devices. The server is also in charge of aggregating the computation from devices to update the global model. Each group of local iterations E (on devices), ending with a global model update, is called an *FL round* (or simply *round*). Note that each worker uses a local batch size B in each of these local iterations. The server may instruct *weak* workers to use less number of batches in order to avoid the *stragglers*' problem, as discussed in Section 6.3.4. On top of this, we design training and system specifics to reach our scalability and resilience goals, as we show later.

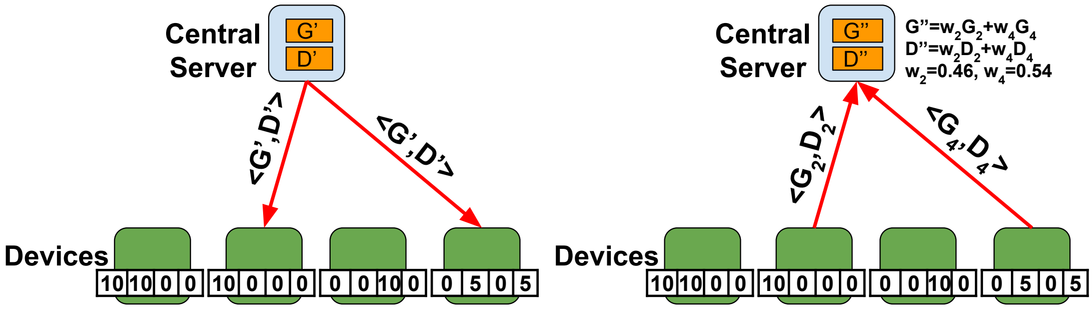
Each device owns a GAN locally, composed of a local generator and a local discriminator. The input data to the global GAN is the aggregate of the data stored locally on each device. We assume that the data stored on each device remains local: only the output of the local computation is sent to the server. Given its local nature, data might be unbalanced and possibly not identically nor independently distributed (*non-iid*) across devices. For this, we allow the server to gather *metadata* from devices regarding the number of local samples per class. To satisfy this step, we assume each device can label its data or at least can model its data distribution, using generative or clustering schemes like Gaussian Mixture Models (GMM) [247, 210] or K-means [75].

Devices only communicate with the server. As they do not communicate with each other, they do not need to trust each other. Communication links between the devices and the server could be unreliable and asymmetric with limited bandwidth [139]. The server can keep the devices' updates secure using off–the–shelf *secure aggregation* protocols [35].



(a) FL round 1 starts

(b) Server updates G and D



(c) FL round 2 starts

(d) Server updates G and D

Figure 6.4: A running example of how FEGAN works. Assume a dataset of 50 samples in 4 classes, distributed on 4 devices with $C = 0.5$ (where C is the fraction of devices sampled by the server in each round). Each device has an array of 4 entries depicting how many samples this device has from each class. In the first round, the server chooses device 1 (with the highest number of samples) and device 3 (with samples from a class not represented by device 1). When the devices reply, the server updates the model, applying *KL weighting* to their replies. In the second round, the server chooses the unsampled devices to maintain fairness among devices.

6.3.2 The FEGAN Algorithm

In this section, we explain the high-level algorithm of FEGAN. Figure 6.4 depicts a running example of FEGAN.

Initialization. Before starting the training process, each device informs the server of its local data distribution (typically how many classes and how many samples per class it owns locally). Such a step could be repeated should the dataset change during the training process. The goal of collecting metadata from all devices is for the server to account for the data imbalance across devices.

To this end, the server computes the *Kullback–Leibler (KL) divergence* [146, 145] scores, which reflect the degree of divergence of the devices' local data distribution from the global distribution. In particular, the KL divergence of device k is computed as follows:

$$D_{KL}(P_k||Q) = \sum_{x\in\mathscr{X}} P_k(x)\log(\frac{P_k(x)}{Q(x)}), \tag{6.1}$$

where P_k is the normalized vector of samples per class x at device k, Q is the normalized vector of the total number of samples per class (in the global dataset), and $\mathscr{X}$ is the collection of all classes. For instance, $P_k(x) = {}^{n_{k_x}}/_{n_k}$, where n_{k_x} is the number of samples of class x at device k, and n_k is the total number of samples at device k. The server then assigns a score s_k to each device k as follows:

$$s_k = \frac{n_k}{n} \times D_{KL}(P_k||Q), \tag{6.2}$$

where n is the total number of samples in the dataset.

All devices also send to the server their capabilities in terms of memory size and compute power available for training. The server then uses this information to choose a training load (in terms of batch size and local number of iterations) for each device. As for the current version of FeGAN, the server follows a linear model to compute this load, which we assume fixed throughout the training process. Finally, the server initializes the model, i.e., $\mathscr{G}$ and $\mathscr{D}$ and selects an optimizer and a loss function to use.

Server operation. The server starts each FL round by choosing a group of devices, using balanced sampling (Section 6.3.3), to update collectively the model state. The server then asks the sampled devices to contribute to the current round and assigns the pre-computed training load to each device. At the end of an FL round, the server collects the updates of the devices and aggregates them using the KL weighting scheme (Section 6.3.3). The server also weighs such updates by the devices' respective assigned batch sizes (Section 6.3.4). The server repeats this procedure till convergence.

Device operation. Devices remain idle until they are contacted by the server to participate in one FL round. A device selected for a given round receives the updated model state (of $\mathscr{D}$ and $\mathscr{G}$) and the values for E, the number of local training iterations to be done by this device, and B, the training batch size. After running E local training iterations, each device sends back the updated model to the server.

6.3.3 FeGAN Design Choices

Co-locating networks. Training vanilla deep neural networks with backpropagation sometimes encounters the vanishing gradients problem. This problem happens when the computed gradients are extremely small, hindering the update of the network's parameters and hence, stopping the learning. In the context of GANs, such a problem usually happens due to having a very strong discriminator [185]. Given that training GANs is naturally formulated as a *game* between two players/networks, a player gains *strength* with more *experience* (i.e., with more training iterations). This *strength* then reflects how good the player is (either the generator or the discriminator) in doing its task. In the situation of a powerful discriminator, the generator fails to learn as it cannot get much information from the discriminator's feedback. As a result, the discriminator always wins in this case, i.e., the generator always fails to generate *fake* samples that can trick the discriminator. In the centralized setup, this problem can be mitigated by choosing the *correct* set of hyperparameters and the loss functions [84, 168]. Yet, this is not sufficient in the distributed setup; it is also crucial to balance the strength of both networks. FEGAN solves this problem by co-locating the discriminator with the generator so that both networks can be trained simultaneously, increasing their strength at the same time.

Importantly, networks co-location reduces the communication overhead compared to other designs that centralize the generator in the server while distributing the discriminators on the devices [101, 23]. In the latter case, the distributed discriminators should periodically send their feedback, whose size is the same as that of the input space that is desired to be learned (e.g., pixels of an image), to the centralized generator. By co-locating networks, FEGAN drastically reduces such an overhead, as we show in Section 6.6.

Kullback-Leibler (KL) weighting. Data is more likely to follow different distributions on different devices and hence, using vanilla averaging of devices' local models might lead to learning divergence [163]. Vanilla *federated averaging (FedAvg)* [176] is also not sufficient for convergence in some cases [281] as it does not take data distribution discrepancies into consideration while weighting the devices' updates.

FEGAN mitigates learning divergence by relying on Kullback-Leibler (KL) weighting, which is applied as follows. Upon the receipt of the updated models from the devices, the server weighs them by applying the *softmax* function on the *KL divergence score* of each device. More specifically, the model received from device k is given a weight w_k as follows:

$$w_k = \frac{\exp^{-s_k}}{\sum_{i \in \mathscr{I}} \exp^{-s_i}}, \tag{6.3}$$

where s_k is the pre-computed KL score of device k (see Equation 6.2), and $\mathscr{I}$ is the set of devices contributing to the current FL round. The *softmax* function acts as a normalization[1] to the weights given to the devices' updates so as to make these weights sum to 1. In other words, the output of the *softmax* function (i.e., the weights) can be seen as the contribution of each device to updating the global model state. The negative sign given to the score is to *penalize* the devices with more divergence from the global data distribution as their updates are less useful than the other devices' updates. Such a design of a weighting scheme is crucial

---

[1] We experimented with other normalization schemes (e.g., linear averaging and step-wise averaging), and we found that the *softmax* normalization gives the best results.

to learning the *global distribution* of data.

Balanced sampling. Mode collapse [46] is arguably the hardest problem that one can encounter while training a GAN. Such a problem happens when the generator learns only partially the data distribution and hence, can only generate data from a limited number of classes. In our distributed, uncontrolled environment, the situation could be exacerbated due to the skewness of data distribution on the devices. Our *balanced sampling* scheme aims at solving this problem.

At each FL round, the server *samples* $C \times n_d$ devices to do the training procedure, where C denotes the fraction of devices doing the training in each round with $C \in]0, 1]$, and n_d is the total number of devices. Note that unlike vanilla FL (which relies on random sampling [139]), in FEGAN, the server samples devices with specific data distribution to avoid falling into mode collapse. The novelty of FEGAN's sampling is that it favors (1) devices with more samples, (2) devices with a balanced dataset (i.e., almost equal number of samples per class), and (3) unsampled devices in the previous rounds. Thus, our sampling technique always ensures a balanced number of samples per class in the updates applied to the model ($\mathscr{G}$ and $\mathscr{D}$). Such a technique plays an important role in the robustness and the convergence speed, as we show in our evaluation of FEGAN.

Technically, FEGAN applies balanced sampling as follows: (1) The server maintains a priority queue with the number of accumulated samples per class from sampled devices (from previous rounds) stored in it. (2) At each round, the server picks the class with the least number of accumulated samples and chooses a device that declares to have this class to include it in the new round. (3) The server accounts for the samples from other classes that will be included in the new round (due to including the chosen device) and updates the queue accordingly. (4) The server repeats this procedure until it samples $C \times n_d$ devices. It then sends the updated state of the model, i.e., $\mathscr{D}$ and $\mathscr{G}$ networks to the sampled devices.

6.3.4 FEGAN System Aspects

Handling heterogeneity. Devices in the wild are not equal in many respects, including their hardware specifications (e.g., compute power and memory) and the available network bandwidth. Tolerating device heterogeneity in an optimal manner is an open problem with many heuristics [5, 163]. Because we are targeting GANs training, it is crucial to design a solution that handles device heterogeneity without falling into GAN-specific problems. For instance, a solution that prefers sampling strong devices over weak devices e.g., [186], can lead to mode collapse if the weak devices own data classes that are not represented by the strong devices.

FEGAN handles this issue by controlling the *training load* given to the contributing devices in one round. More specifically, the central server dictates the number of iterations and the batch size used for local training on devices. The server chooses less number of iterations and smaller data batches for weaker devices. This helps mitigate the straggler issue [157], which happens with synchronous training systems when some devices are significantly slow, degrading the whole system throughput. In the aggregation phase, devices' updates are also weighted based on the load given to each of them. Such weights are multiplied by the KL weights (see Equation 6.3).

Fault tolerance. The central server represents a single point of failure: if it crashes, without any fault tolerance technique, the whole learning stops, as devices do not communicate with nor trust each other. FEGAN uses periodic checkpointing for the server's fault tolerance. Initially, FEGAN starts multiple servers, which number is configurable, yet, only one of them acts as a primary, and the rest are backups. Only the primary communicates with the devices while the other servers remain idle. After each FL round, the primary stores the learning state in a persistent database. The learning state includes the weights of both networks, $\mathscr{G}$ and $\mathscr{D}$, the optimizer, the number of epochs passed, the data distribution on the devices, the devices' capabilities, and the history of the chosen devices in all previous rounds.

In the case of the crash of the primary server, a backup server takes over. First, it loads the latest learning state from the persistent database and announces to the devices that it is the new primary. The new primary then starts a new round by choosing a new set of devices to do the training.

Network failure is detected through a standard timeout mechanism. Such a failure could be either due to slow communication, network partition, or a crashing device. In the event of a failure, the server calls for completion or abortion of the current round. Such a decision relies on the server configuration and the learning state. The server then records the actual devices that participated in this round rather than the intended set of devices to participate.

Implementation. We implemented FEGAN on PyTorch [194]. We integrated our implementation with already-existing public implementations of different GAN architectures implemented for training on one machine. We now describe the communication abstractions we implemented to allow the distribution of GAN via FEGAN. Our source code is available at [93].

We rely on the notion of PyTorch distributed groups.[2] Any invoked communication abstraction takes as an input a *group* defining the set of devices that will be involved in this communication. As we rely on a centralized architecture (see Figure 6.3), we allow only the centralized server to create groups to avoid any conflicts with the devices and hence, achieve strong consistency on the group formation.

We introduce two abstractions that can be used off–the–shelf by both the server and the devices: *multicast model* and the *average models. Multicast model* is used by the server at the beginning of any FL round to multicast the current state of the model to a group of devices

---

[2] https://pytorch.org/docs/stable/distributed.html

(similar to *map* in MapReduce notation [64]). To comply with PyTorch distributed runtime, the chosen devices should also invoke this abstraction (i.e., *multicast model*) to receive the models from the server. We use the NCCL backend of PyTorch [188] so that the models stored on a GPU need not to be copied to the CPU memory before being sent back to the devices and hence, saving time and memory.[3]

The second abstraction, *average models*, is used at the end of an FL round (similar to *reduce* in MapReduce notation). Typically, this function is invoked by the server to aggregate devices' updates. As we consider synchronous training, the server waits for all contributing devices to send back their results before proceeding to the next round. If our weighting technique is not activated, we use the *reduce* operation from the distributed backend with *ReduceOp.SUM* as a reduce operator and then divide by the number of contributing devices to this round to get the average of the updates. We use this operation as it is always faster than the other operations. Otherwise, we use the *gather* operation to first collect all the updates from the devices and then apply the weights calculated by the server. The main problem with the latter operation is that it does not work on GPUs,[4] and hence the updates need to be first copied to the main memory before sending them to the server. At the other end, the server collects all the updates, copies them to the GPU memory, and then calculates the weighted average to get the new model. We could not overcome this inherent problem of the PyTorch distributed backend as *gather* is the only appropriate abstraction we can use to implement our *average model* abstraction while applying weighted averaging.

Finally, we report on the number of lines of code (LoC) required to port two publicly available[5] GAN implementations to FEGAN. Factoring out the common code for dataset partitioning, group initialization, and performance measurements (e.g., *FID* calculation), it takes less than 70 LoC (which constitutes around 5% of the whole code) to port a GAN implementation to FEGAN.

6.4 Performance Evaluation Methodology

In this section, we describe our experimental setup, baselines, and the configurations we used to evaluate FEGAN.

Testbed. We evaluate FEGAN on the Grid5000 platform [89], using machines from the same cluster. Each machine has 1 CPU (Intel Xeon Gold 6126) with 2 cores, 16 GiB RAM and 2 Gbps Ethernet, and one Nvidia Tesla P100 GPU. Unless stated otherwise, we run experiments on 80 devices. This number reaches up to 176 in some experiments.

---

[3] This is only true if the used abstraction is implemented by NCCL; otherwise, we use the GLOO [118] backend.

[4] https://pytorch.org/docs/stable/distributed.html.

[5] https://github.com/eriklindernoren/PyTorch-GAN.

Evaluation metrics. Although evaluating GANs has been an issue for ML practitioners, requiring human judgment to assess the quality of the generated data [37, 168, 173], robust metrics are now commonly used to assess the performance of GANs [273, 101, 147] such as *Frechet Inception Distance (FID)*. We evaluate FEGAN observing two main metrics: *FID* to assess the GAN convergence (or quality of the learned distribution) and *throughput* to measure the system's efficiency. We chose FID as it is one of the most stable, robust, and widely-used metrics for GAN [37]; it was the one used by our predecessors [101, 273]. Besides, FID has an official implementation in the popular ML frameworks, including TensorFlow and PyTorch.

1. FID is a metric that computes the distance between feature vectors of real images and generated ones. The score reflects the statistical divergence between the generated images and the raw images using the *Inception-v3* model [232], primarily designed for image classification. The lower the score, the better (e.g., a 0.0 score indicates that the two groups of images are statistically identical).

2. Throughput defines the number of updates the server can process per second. Such a metric reflects the scalability of the system and the efficiency of the communication.

Note that we observe training convergence with time, the number of training epochs, and FL rounds. However, in our experiments, we maintain a one-to-one mapping between the number of epochs and the number of FL rounds, so we always show only one of them. The shown results are the average of 6 runs per experiment; we omit error bars for better readability.

Datasets. We evaluate FEGAN with three datasets: *MNIST* [152], *Fashion-MNIST* [276], and *ImageNet* [65]. In our evaluation, we focus on image generation not only because it is very challenging and resource-demanding but also due to its multiple applications in the real world. *MNIST* and *Fashion-MNIST* both describe grey-scale images of 10 classes representing handwritten digits and clothes, respectively. Each dataset has 28x28 60K training images and 10K testing images; we use the latter for computing the *FID* score. *ImageNet* [65] is a large dataset with around 14M images of a 256x256 resolution, dispatched into more than 21K classes. ImageNet classification challenges usually use a subset of this dataset [213, 230]. We use a subset of ImageNet with 100K images, distributed among 200 classes with 500 samples per class. Reducing the dataset only allows for faster convergence. Unless otherwise stated, we use Fashion-MNIST as a default dataset throughout our evaluation.

Non-iidness. In all the considered datasets, the data is balanced: the average number of samples per class is almost the same in all classes. To our knowledge, there is no publicly available dataset with inherent *non-iidness* or imbalanced data. We designed a distribution engine to emulate imbalanced and skewed distributions of data among devices.

Our engine accepts two input parameters: *max_class* and *max_samples*. The former defines the maximum number of classes any device can have, and the latter defines the maximum number of samples per class on any device. For each device, the engine generates a random number $r_c \in [1, \frac{\text{max\_class} \times i}{n}]$, where i is the index of the device, i.e., $i \in [1, n]$ and n is the total number of devices. r_c defines the number of classes this device will have. Then, the engine randomly samples r_c classes from the dataset. Similarly, the engine (at each device) generates a random number r_s for each selected class with $r_s \in [1, \min(i^2, \frac{\text{max\_samples} \times i}{n})]$. Finally, the engine randomly samples r_s samples for each selected class from the dataset. Note that the engine generates a different value for r_s per class. Note that all our random choices and samplings follow the *uniform* distribution. Using this engine, we managed to emulate several cases of data imbalance and *non-iid* distribution of data among devices, including the typical *non-iid* workloads reported before in the literature [101, 215, 273, 176].

GAN architecture. Without loss of generality, we evaluate FEGAN with two popular GAN architectures. Since FEGAN is GAN-internals agnostic, our work can be ported to any other GAN architecture. The two experimented architectures are: Least Squares Generative Adversarial Networks (LSGAN) [173] for the MNIST and Fashion-MNIST datasets and a Deep Convolutional Generative Adversarial Network (DCGAN) [205] for the ImageNet dataset. Both architectures are lightweight: they do not require more than 3 MBs of memory.

Hyperparameters. Unless otherwise stated, we use the following hyperparameter values in our experiments. We set E, the local number of iterations per each device, to 30, C, the fraction of devices chosen each round (by the server) for the training, to 0.025, B, the batch size, to 50, and *FID* batch size, the number of samples used to calculate the *FID*, to 10K. We discuss the effect of changing the values of E and C in Section 6.6. We monitor the progress of *FID* every 1000 training iterations (not to confuse with *epochs* nor with *FL rounds*). We use the *Adam* optimizer [136] for both the generator and the discriminator with an initial learning rate of 0.0002 and values for betas 0.5 and 0.999. As instructed in their original papers, we use Mean Square Error (MSE) loss function with LSGAN [173] and Binary Cross-Entropy (BCE) with DCGAN [205].

Baselines. We compare FEGAN to three competitors:

1. Centralized GAN. We use a centralized GAN as a baseline to be able to compare the resulting *FID* with FEGAN as well as to illustrate the throughput gain of distributing the computation. We train a GAN on a single machine with two GPUs. The hyperparameters are set to the same value as FEGAN, including the batch size for fairness.

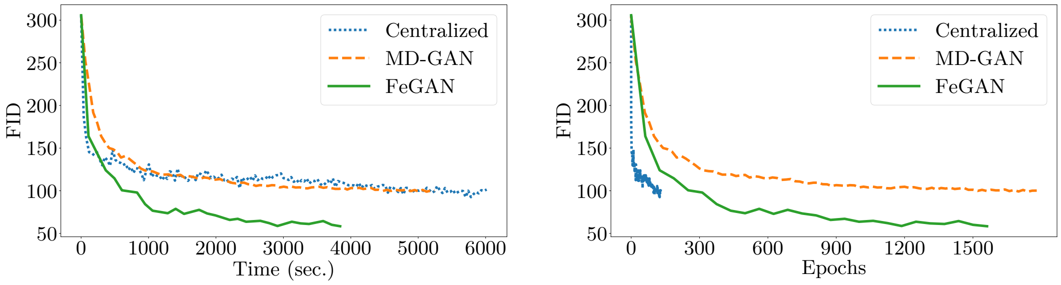
2. Parameter Server-based GAN. Existing approaches to distribute GANs rely on the parameter server architecture [157] to handle the distributed training [101, 273]. Note that the evaluation of such systems (in their original papers) has been conducted by emulation. Since none of these systems is open-sourced yet and rely on the same architecture (with minor nuances), we picked **MD-GAN** [101] as a representative of this class of systems. Like FEGAN, MD-GAN assumes that data on devices is never shared with any other machine. We implemented MD-GAN in our distributed framework using the same networking abstractions, models, and datasets that we used in FEGAN. For fairness, we put *tensors* on GPUs and handle communication using the same GPU-to-GPU abstractions. We also use the best values for the hyperparameters as instructed by the authors in their original paper [101].

*3. Federated Averaging (*FL-VANILLA*).* We compare to a strawman FL setup, where discriminators and generators remain on devices. Such devices send gradient updates to a central server that hosts the up-to-date generator and discriminator. This constitutes the FL baseline, inspired by the *FedAvg* algorithm [176], which we coin FL-VANILLA.

FEGAN configurations. We report on all the variants of our FEGAN system with all combinations of *balanced sampling* and *KL weighting* techniques being used or not. We abbreviate *sampling* with s and *weighting* with w in the figures with 0 means disabled and 1 means enabled. For example, $s = 1, w = 0$ denotes FEGAN deployment that uses our *balanced sampling* scheme but not *KL weighting*. This is completely independent of the distribution of data among devices. Note that setting $s = 0, w = 0$ is equivalent to employing FL-VANILLA.

6.5 Convergence Comparison

In this section, we show the quality of the data generated by FEGAN, measured by *FID*, as well as the convergence behavior compared to other baselines in both *iid* and *non-iid* data settings.



(a) Time (b) Training epochs

Figure 6.5: Convergence of FEGAN compared to the *Centralized* approach and MD-GAN. We distributed the Fashion-MNIST dataset identically and independently (*iid*) on 80 devices for the distributed deployments. *FID* metric: the lower, the better the generated samples.

6.5.1 Convergence in *iid* Settings

Comparison with competitors. We run a head-to-head comparison with a centralized GAN and MD-GAN w.r.t. convergence performance over time and the number of training epochs.

In this experiment, we distribute data identically and independently (*iid*) over devices in the distributed deployments, namely MD-GAN and FEGAN. As we focus on the benefit obtained from FEGAN architecture, namely the fact that both the generator and the discriminator are distributed, we voluntarily disable the weighting and sampling schemes in this experiment ($s = 0$, $w = 0$). Indeed, distribution leads to communication overhead, which FEGAN minimizes (by the co-location of the generator with the discriminator on all devices in addition to sampling a limited number of devices for training in each round).

Results displayed in Figure 6.5 show that FEGAN significantly outperforms both competitors. In both figures, we observe that FEGAN converges to an *FID* value of around 50, whereas the other approaches converge to an *FID* value of around 100. Compared to the centralized deployment, FEGAN achieves a better *FID* in less time (Figure 6.5a). At the first glance, this result looks counter-intuitive. Yet, this is because, in FEGAN, the server aggregates updates based on more data samples (i.e., collected from more devices), highlighting the advantage of distributing the training process.[6] The distribution also gives FEGAN a more diverse view of the dataset and ensures better convergence than in the centralized case [282, 260]. This result is consistent with the observations reported before in the literature [68, 108]. Since MD-GAN relies on a single generator at the server, it fails to benefit from the data diversity on multiple devices and hence achieves a performance similar to the centralized approach. FEGAN achieves a much faster convergence because the server communicates only with a fraction of devices in each round as opposed to MD-GAN, where it communicates with all of them; this enables FEGAN to achieve faster iterations.

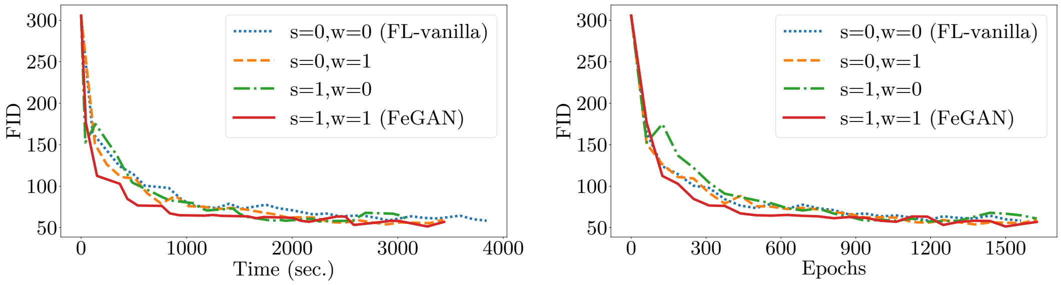
The results depicted in Figure 6.5b show that FEGAN consistently outperforms MD-GAN. However, the centralized approach converges faster than the distributed approaches in the first few hundreds of epochs. After that, FEGAN achieves a better *FID* value. The reason for this behavior is that in the first few epochs, FEGAN contacts different devices per round, and hence, the server aggregates updates from different data, trying to adapt both networks (the generator and the discriminator) to this data, which leads to the observed slow start. In the subsequent iterations, as the weights of both networks become more stable, FEGAN leads to better-generated data (and hence, a better *FID*).

FEGAN configurations. We now assess the impact of the core mechanisms of FEGAN, namely the impact of s and w in an *iid* setting.

Figure 6.6 displays the results of different FEGAN deployments with all combinations of enabling and disabling the *balanced sampling* and the *KL weighting* schemes. We observe that in terms of convergence speed (w.r.t. both time and training epochs), they perform approximately the same. FEGAN (with $s = 1, w = 1$) performs slightly better and provides quickly a smaller *FID*. Those results demonstrate that, while we aimed at sustaining *non-iid*

---

[6] Note that we could not arbitrarily increase the training batch size on the centralized setup, which is prone to memory constraints.

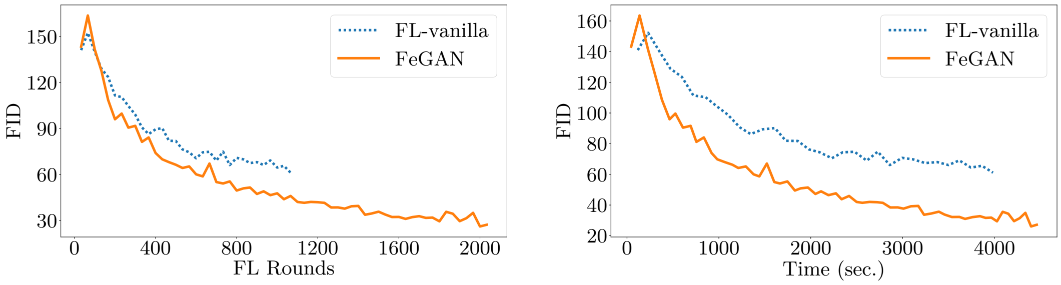


(a) Time

(b) Training epochs

Figure 6.6: Convergence of a few FEGAN configurations in a distributed setup with the Fashion-MNIST dataset distributed identically and independently (*iid*) on 80 devices.

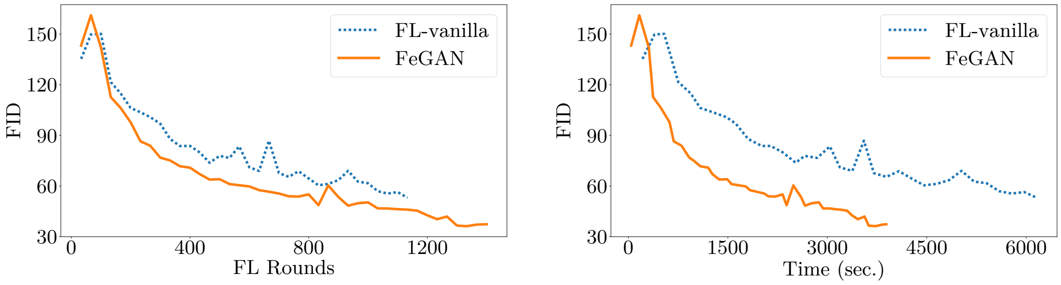
data when designing our system, FEGAN is equally good in *iid* setups. This makes FEGAN an excellent system across data distributions.



(a) Convergence with FL rounds

(b) Convergence with time

max_class=20 - max_samples=3000

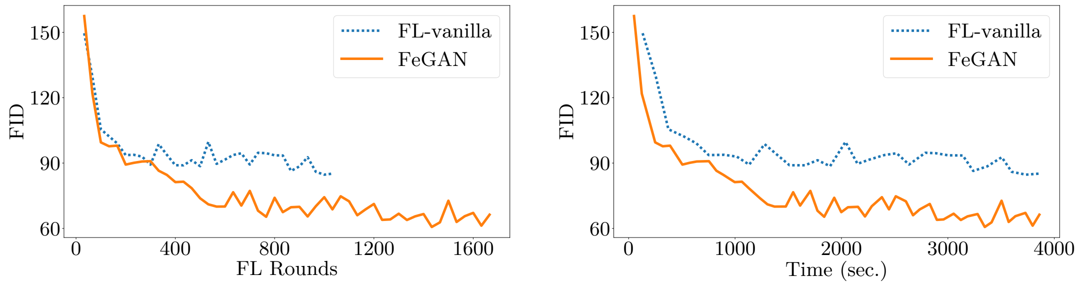


(c) Convergence with FL rounds

(d) Convergence with time

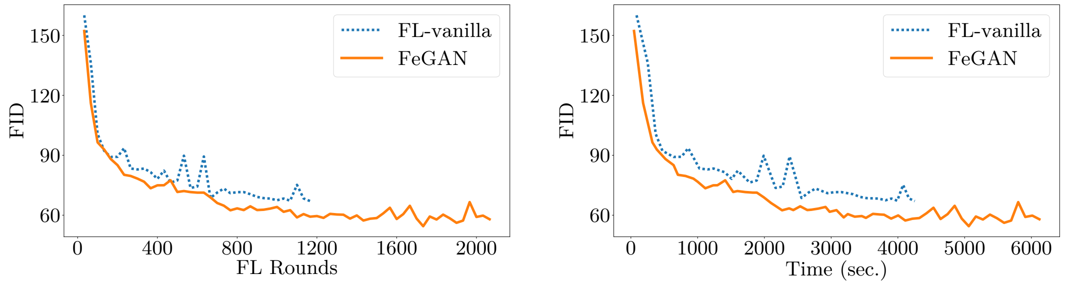
max_class=40 - max_samples=3000

Figure 6.7: Convergence of FEGAN in a distributed setup with MNIST; data is not distributed identically and independently (*non-iid*) on devices.



(a) Convergence with FL rounds (b) Convergence with time

max_class=10 - max_samples=2000



(c) Convergence with FL rounds (d) Convergence with time

max_class=40 - max_samples=4000

Figure 6.8: Convergence of FeGAN in a distributed setup with Fashion-MNIST; data is not distributed identically and independently (*non-iid*) on devices.

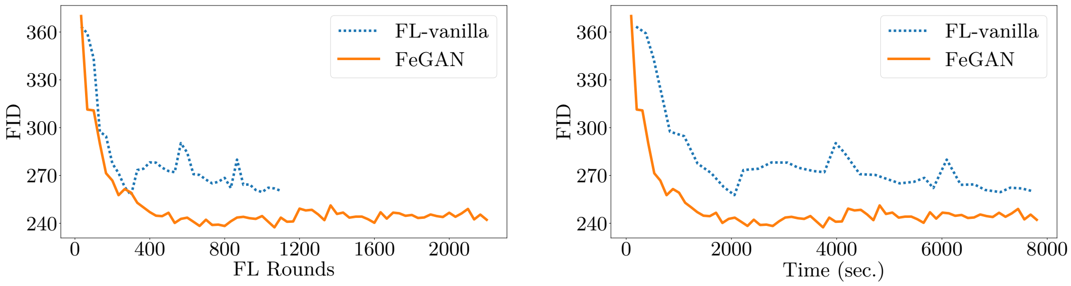
6.5.2 Convergence in *Non-iid* Settings

In this section, we extensively evaluate the performance of FeGAN in a *non-iid* setup. We compare FeGAN in its fully-fledged version (with $s = 1$, $w = 1$) to the FeGAN version where the sampling and the weighting schemes have been disabled (i.e., the FL-vanilla competitor). We run these experiments in three different datasets and different data distributions. We control the data distribution (and hence, the degree of *non-iidness* and data imbalance) using the data distribution engine described in Section 6.4. Figures 6.7, 6.8, and 6.9 show the results on the MNIST, Fashion-MNIST, and ImageNet datasets, respectively, each in two settings where we vary the maximum number of classes and samples per device. The first two datasets are used to show the effectiveness of FeGAN on well-known baseline datasets, whereas the ImageNet experiment shows the performance of FeGAN on a large-scale dataset. As the results show similar behavior across datasets, we discuss the three figures collectively, highlighting the main performance gains of FeGAN.

Faster model updates. Our first observation is that FeGAN runs epochs *faster*, i.e., leading to a faster convergence than FL-vanilla.[7] The *balanced sampling* (as compared to random

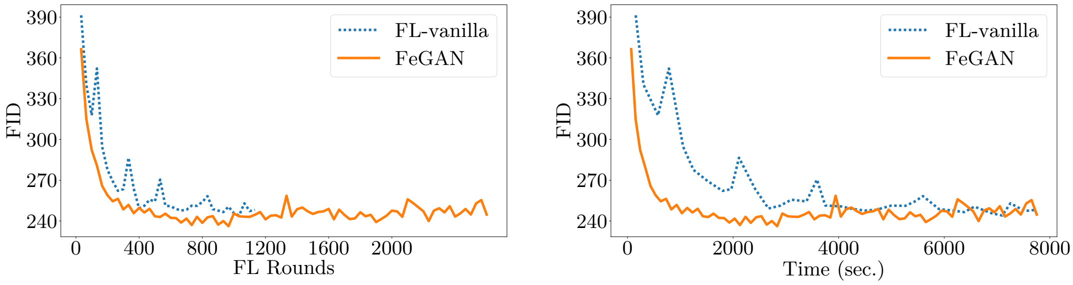
---

[7] Note that we run both systems for the same amount of time per experiment; that is why the lines in all sub-figures (b) are balanced, unlike sub-figures (a).



(a) Convergence with FL rounds (b) Convergence with time

max_class=200 - max_samples=500



(c) Convergence with FL rounds (d) Convergence with time

max_class=200 - max_samples=50

Figure 6.9: Convergence of FEGAN in a distributed setup with ImageNet; data is not distributed identically and independently (*non-iid*) on devices.

sampling in FL-VANILLA) of FEGAN favors devices that were not visited in previous rounds, and hence, it has a higher probability of sweeping over the whole dataset (distributed among devices) faster than FL-VANILLA. We quantify this throughput gain to $1.27-1.8\times$ with MNIST, $1.57-1.7\times$ with Fashion-MNIST, and $1.9-2.13\times$ with ImageNet. We discuss this observation in detail in Section 6.6.

Better *FID*. Not only does FEGAN achieve faster epochs in all datasets, but we also observe that it converges to a better *FID* value, compared to FL-VANILLA, after training for the same number of epochs. This outperformance is enabled by our *balanced sampling* and *KL weighting* schemes. The fact that the server adjusts the weight of each device depending on its data distribution helps achieve a better generation of data (and hence, a lower value for *FID*), as opposed to FL-VANILLA, which considers devices equal regardless of the number of classes and the number of samples per class they hold. This, in turn, enables the server of FEGAN to consider faster a diverse set of data with a higher probability, contributing to learning faster the global data distribution.

Convergence gain. To demonstrate the superiority of FEGAN over FL-VANILLA, we compare the time required to achieve a given *FID* value (usually the best value achieved by FL-VANILLA)

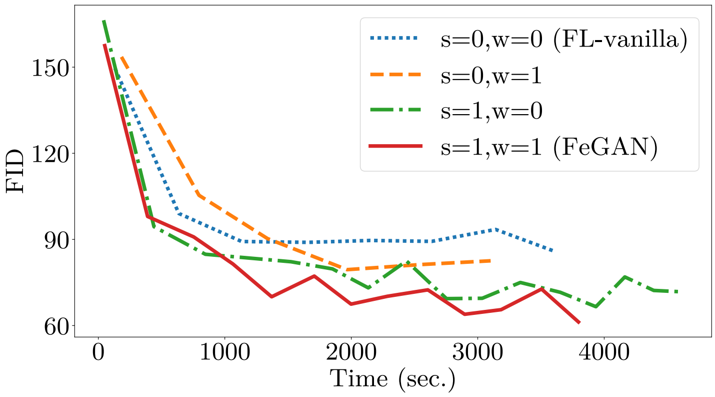


Figure 6.10: Convergence of FEGAN with *non-iid* Fashion-MNIST data distributed on 80 devices, showing all cases of enabling and disabling weighting and sampling schemes.

and call it *convergence gain*. Results show that the convergence gain of FEGAN over FL-VANILLA is $1.27 - 1.57\times$, $1.71 - 3\times$, and $2.75 - 3.67\times$ when training MNIST, Fashion-MNIST, and ImageNet, respectively. Beyond the ultimate *FID* value that a system can achieve eventually, the convergence gain is an important end–metric for ML practitioners. Such a metric combines the system's performance aspect (e.g., throughput or latency) and the ML algorithm performance aspect (i.e., convergence over training epochs).

The less uniform, the more effective FEGAN is. When comparing the first and second rows of Figures 6.7 and 6.8, we observe that the performance gap between FEGAN and FL-VANILLA is larger for lower values of *max_class*. In other words, the effectiveness of FEGAN is even clearer when each device owns only a few classes from the whole dataset. This result demonstrates the ability of FEGAN to account for a higher degree of data imbalance over devices; we believe the latter is a realistic scenario as we do not expect that all classes would be represented on distributed devices. Figures 6.9c and 6.9d show a real-world scenario, where each device has at most 50 samples from any of the 200 classes of ImageNet. Results from both figures show that FEGAN converges faster than FL-VANILLA in terms of both the number of rounds and time.

6.5.3 FEGAN's Tolerance to GAN-specific Problems

The contribution of sampling and weighting. We evaluate the relative impact of activating each mechanism (i.e., sampling and weighting) in all datasets. The results for Fashion-MNIST are depicted in Figure 6.10. Our general observation is that: while using only one mechanism is enough for FEGAN to achieve a lower *FID* than FL-VANILLA (where using *sampling* alone is slightly better than using *weighting* alone), using both mechanisms achieves the lowest *FID*.

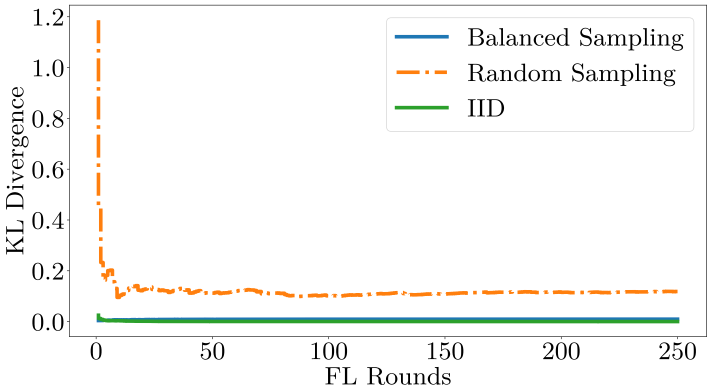
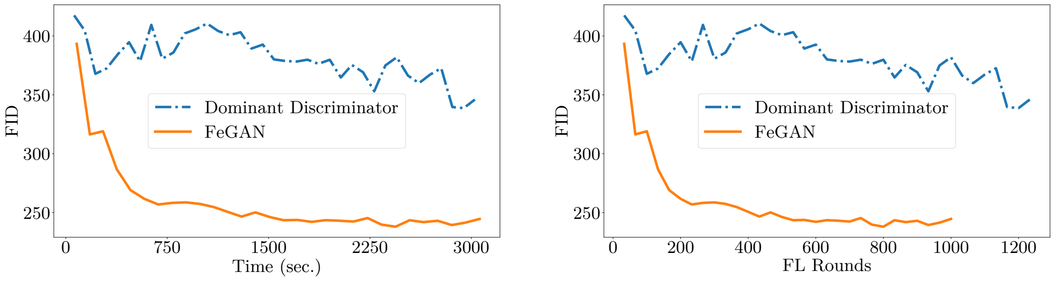


Figure 6.11: KL divergence of the real distribution from the distribution seen by the centralized server. The smaller the KL divergence, the lower the probability of mode collapse.



(a) Time

(b) Training epochs

Figure 6.12: The co-location of both networks on all devices avoids yielding a powerful discriminator (as in MD-GAN), preventing the problem of vanishing gradients.

Preventing mode collapse. Figure 6.11 shows the KL divergence of the data distribution seen by the server (over FL rounds) compared to the real data distribution. We compare *balanced sampling* (used by FEGAN) to random sampling (used by FL with *FedAvg* [139]), both with *non–iid* data, and the *iid* case (when data is distributed identically on all devices). The latter constitutes the optimal baseline in this experiment. On the one hand, the figure shows that using random sampling with *non–iid* data will let the server see a diverging distribution compared to the real distribution of data. On the other hand, balanced sampling of *non–iid* data allows the server to see an almost–exact distribution of data (compared to the optimal case, i.e., with *iid* data). As demonstrated by previous work [163], the smaller the KL divergence, the less the probability of experiencing mode collapse. In that sense, FEGAN then shows resilience to problems caused by *non–iid* data.

Preventing vanishing gradients. FEGAN co-locates the discriminator with the generator on all devices so that the training of both networks can happen simultaneously in all training iterations, allowing both networks to have the same power in confronting each other. To

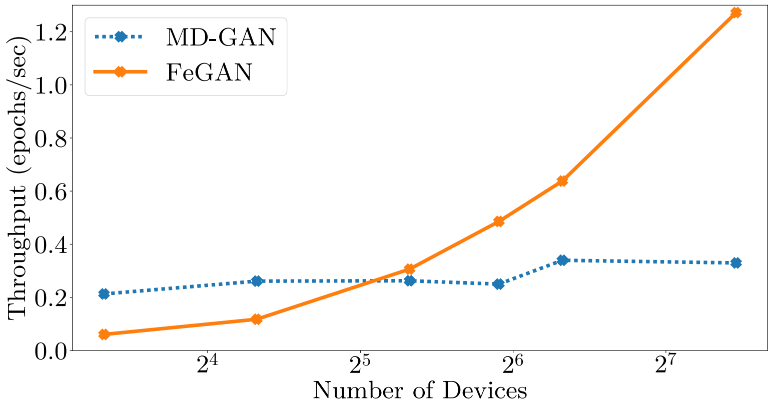


Figure 6.13: Scalability of FeGAN with the number of devices.

demonstrate the efficiency of this approach, we run an experiment in which the discriminator is trained in all iterations, whereas the generator is trained only once each E iterations. Such a design reflects the setup of having one generator in the central server and multiple discriminators on the devices, which was proposed, e.g., in [235, 23]. Figure 6.12 shows that the latter design creates a dominant discriminator that can always defeat the generator, i.e., hindering the generator from generating output data that looks real. Yet, the co-location of both networks on all devices (applied by FeGAN) allows the generator to be powerful enough to generate such data, as demonstrated by the lower *FID* values. FeGAN thus clearly avoids that problem.

6.6 FeGAN System Performance

In this section, we report on the gains (throughput and bandwidth) achieved by FeGAN compared to existing approaches. In addition, we convey FeGAN's performance with heterogeneous devices and in the case of a server's crash. The results being similar across datasets, we only present the results obtained with the Fashion-MNIST dataset unless stated otherwise.

Throughput. We measure the system's throughput as the number of epochs processed per second. Figure 6.13 shows the scalability (in terms of throughput) of both MD-GAN and FeGAN with the number of devices. We observe a stable throughput for MD-GAN, which is almost non–sensitive to an increasing number of devices. This is because the single generator on the server has bounded processing capacity regardless of the number of devices interacting with it. Results show that FeGAN, however, exhibits better scalability: FeGAN's throughput increases linearly with the number of devices, fully leveraging the potential of distributing the training. With 176 devices, FeGAN achieves a 5× throughput gain over MD-GAN. Notably, this scalability behavior relies heavily on the values of E and C. For instance, we do not expect FeGAN to scale with $C = 1, E = 1$ since it will incur the same amount of communication as with MD-GAN in this case. Hence, the values of C and E largely control whether the bottleneck

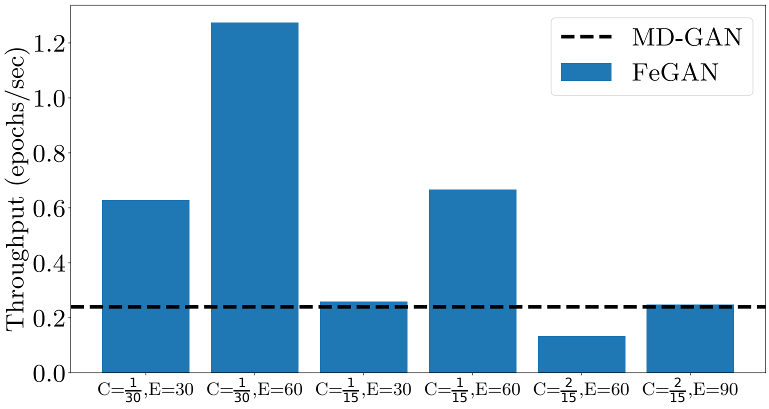


Figure 6.14: The impact of the values of E and C (60 devices).

is the communication or the computation.

Figure 6.14 presents the throughput results obtained with different values of E, the local number of iterations performed on each device in one FL round, and C, the fraction of devices chosen by the server in each round, for FEGAN and MD-GAN (as a baseline) in a 60–device configuration. We observe that the performances of FEGAN are significantly better in several configurations and, in any case, at least as good as the ones of MD-GAN, but in the $C = 2/15$, $E = 60$ configuration. We believe this latter issue is a pure implementation issue, which we detail in the following. On the one hand, since the server in MD-GAN only averages the devices' updates, we use the *reduce* abstraction in PyTorch, which has an efficient GPU-to-GPU NCCL [188] implementation. On the other hand, we use the *gather* abstraction with FEGAN to enable *KL weighting*, i.e., the server first gathers the updates and then applies a weighted sum to them. Such a communication abstraction is not supported on GPUs (to–date in PyTorch), and hence, the models are copied to the CPU memory first, leading to slower end-to-end communication.

From these results, we observe that lower values of C give a significant advantage to FEGAN, with one configuration ($C = 1/30$, $E = 60$) even performing close to 4× faster than MD-GAN. This behavior is explained by the communication overhead between the server and the devices: the lower the value of C, the lower the number of devices chosen per round, and hence, the lower the communication overhead. Since the communication overhead is the main bottleneck in distributed machine learning applications [254, 110], reducing the communication drastically improves the system throughput without impacting the convergence, as demonstrated in the previous section.

Figure 6.14 also shows a positive correlation between the value of E and the throughput. The higher the value of E, the more local iterations per device per round, the lower the communication between the server and the devices per unit time, and hence, the higher the system's throughput. Note that while studying the impact of the values of C and E gives us some indication, choosing the best values for C and E remains data and model-dependent.

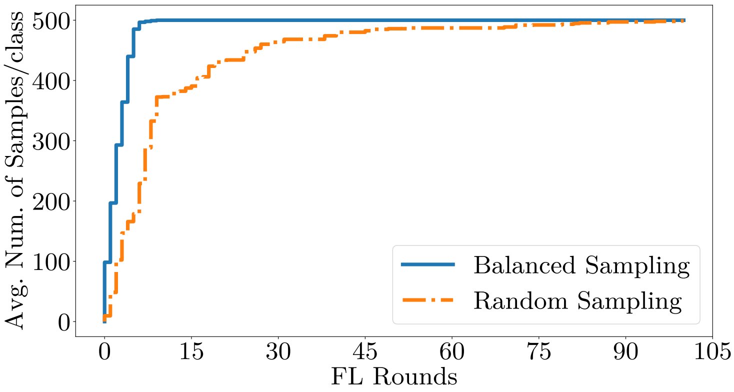
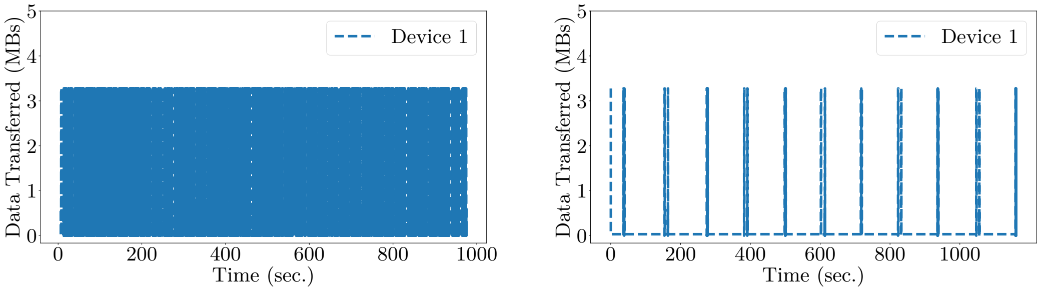


Figure 6.15: The average number of samples per class seen by the server, through devices' updates, as a function of FL rounds. Here, the maximum number of samples at each device is 50. (ImageNet, 80 devices).



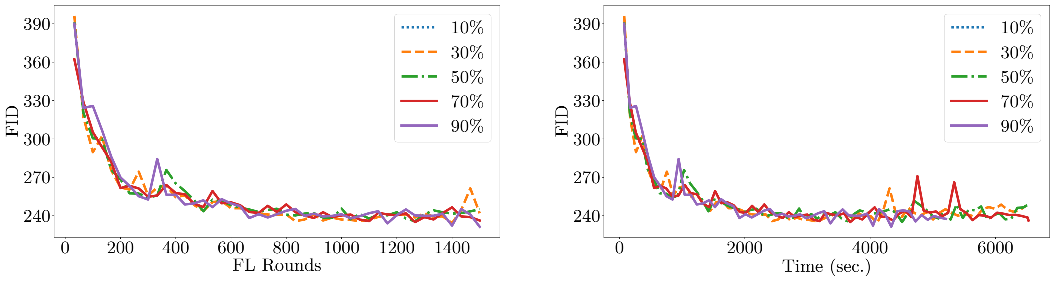
(a) MD-GAN (b) FEGAN

Figure 6.16: Bandwidth consumption at one random device for both MD-GAN and FEGAN. We observe the same distribution (as plotted) at different sampled devices. Using FEGAN, each device is picked (by the server), on average, less than 10 times in a 200-second interval.

Setup	5 - 3000	10 - 2000	20 - 3000	40 - 4000
FL-VANILLA	3596.9s	3733.3s	4825.7s	3815.5s
FEGAN	2441.4s	2379.9s	3031.7s	2560.2s
Gain	1.47×	1.57×	1.59×	1.49×

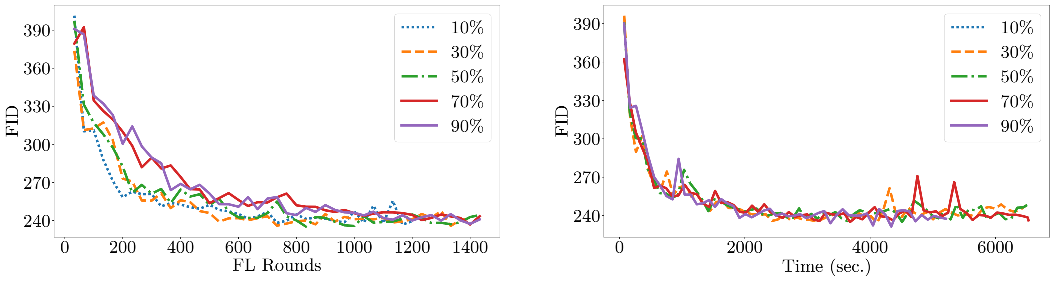
Table 6.1: Time to finish 200 epochs by FEGAN and FL-VANILLA. We use *balanced sampling* with the former and random sampling with the latter. The given two numbers in the first row represent *max_class* and *max_samples*, respectively. The gain in time to finish one epoch can be also viewed as a gain in consumed bandwidth over random sampling for not looking at diverse data batches.

Network bandwidth. Figure 6.15 shows the impact of *balanced sampling* on the data diversity (number of samples per class) observed by the server as a function of the number of FL rounds. We observe that *balanced sampling* significantly impacts the training time: FEGAN takes 10 rounds to come across 500 samples against 10× this figure when random sampling is applied.



(a) Convergence with FL rounds (b) Convergence with time

Limited memory



(c) Convergence with FL rounds (d) Convergence with time

Limited processing power

Figure 6.17: Convergence of FEGAN in the presence of weak devices in terms of memory or processing power. FEGAN is adaptive to heterogeneous devices. (ImageNet, 60 devices)

The effectiveness of *balanced sampling* directly translates into network bandwidth gain. Effectively, random sampling has a higher probability to visit the same data batch twice, consuming unnecessary bandwidth. Table 6.1 summarizes the gains in terms of time and communication. Such results confirm the results displayed in Figure 6.15. Based on these numbers, *random sampling* requires roughly 1.5× more bandwidth compared to our *balanced sampling* per epoch.

Moreover, as Figure 6.16 shows, devices in FEGAN consume less bandwidth than with MD-GAN as each device is selected only once every few FL rounds as opposed to MD-GAN, in which all devices train the model each round. In this figure, each device is sampled less than 10 times every 200 seconds.

Heterogeneous devices. Figure 6.17 shows an experiment with a set of devices, some of which with limited memory or processing power (we denote those as *weak* devices). Based on FEGAN's design, such devices are assigned smaller batch sizes (to respect the memory constraints) or less number of local iterations (to respect the limited processing power). In this experiment, weak devices are assigned up to half of the batch size and half of the local number of iterations. The figure also shows the effect of having different ratios of weak devices compared to the total number of devices (in the range of 10–90%). All figures show that

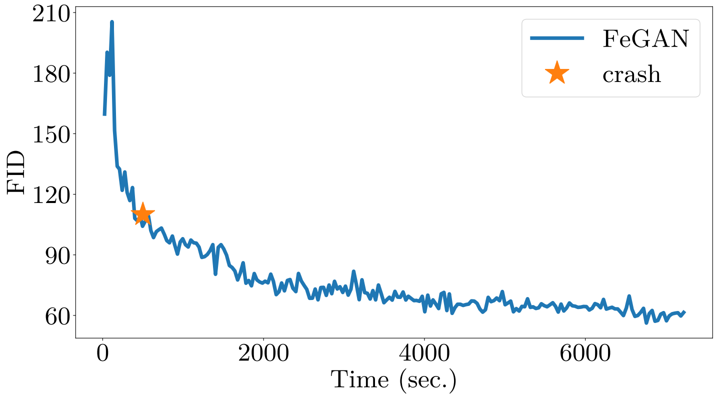


Figure 6.18: FEGAN tolerates the server's crash.

the presence of weak devices almost does not affect FEGAN's convergence. This shows the efficiency and the practicality of the adaptive methods included in the design of FEGAN.

Tolerating server's crashes. As the central server failure is more critical than the devices' failures, we focus here on the former case. Figure 6.18 shows an experiment in which we crash the central server after 500 seconds from the beginning of the experiment. In such a case, a backup server loads the latest stored checkpoint, announces to the devices that it is the new primary, and continues the training normally from that checkpoint. The figure shows that the server failure/crash does not cause troubles (e.g., big stalls) to the training process, which continues normally after the crash.

6.7 Related Work

Few proposals suggested distributing GANs computation on multiple machines. MD-GAN [101] aims at reducing the computation on workers by relying on a single generator, hosted on the central server of the parameter server model. In this setup, every worker hosts a discriminator. This architecture then breaks the usual generator–discriminator couple by setting up a 1-to-n game. The discriminators perform local learning steps on their datasets (that are never shared with other machines) and compute the error feedback on the generated samples they are given by the server. Discriminators are periodically swapped between the workers in a peer-to-peer fashion in order to avoid overfitting the local datasets. MD-GAN is built with scalability to a few tens of workers in mind for targeting the within-datacenter learning scheme; it also only targets the *iid* distribution of data.

A recent approach by Yonetani et al. [273] builds on the architecture of MD-GAN, with also one single generator. They aim at tackling some distribution skew on worker nodes by proposing two unsupervised learning approaches: F2U and F2A. F2U is designed to fool the most *forgiving* discriminator for a given sample (i.e., the discriminator that judge the sample as real and close to their data). F2A adaptively aggregates the feedback of discriminators by emphasizing those from the more forgiving ones. Augenstein et al. [23] proposed a similar federated GAN algorithm with one generator at the server and multiple discriminators at the devices. Such an algorithm focuses on the privacy of users' data on devices with the goal of synthesizing data from similar distribution of real data for diagnosis. The scalability of a single generator here is also at stake; we propose instead to have multiple generator-discriminator couples.

Hardy et al. [100] proposed a fully decentralized, peer-to-peer architecture for distributing GANs. The authors experimented with GANs being gossiped and averaged between independent devices. While acceptable on small datasets, the performances are inferior to the baselines we use to evaluate FEGAN with larger datasets.

6.8 Concluding Remarks

We evaluated FEGAN extensively, and here is our conclusion: *(i)* its careful design allows for scaling the training of GANs. While MD-GAN provides better throughput at a small scale (up to 32 devices), FEGAN then largely prevails, achieving a linear improvement in throughput with the number of devices (experimented with up to 176 devices) with even a lower bandwidth consumption. This scaling does not come at the cost of learning quality; on the opposite, we show lower FID values in both *iid* data and *non–iid* data contexts. *(ii)* We have shown that the proposed components of FEGAN can circumvent GAN-specific issues that a vanilla deployment of an FL-based scheme can encounter. In that respect, both the *balanced sampling* and the *KL weighting* schemes are instrumental in the effectiveness of FEGAN. Although our main ideas are designed to combat GAN-specific issues, they can be still used in vanilla federated learning systems (for deep networks training). For instance, the weighting and the sampling schemes can be used to ensure learning convergence while training deep networks.

Such components essentially utilize the skew information released by the devices at the beginning of the learning with regard to their local data. The requirement for extra information might constitute a privacy concern. Yet, recent work has shown how to employ differential privacy with training vanilla federated GANs [261]. Exploring approaches that tolerate data imbalance while ensuring the privacy of devices' data is an avenue for future research.

Concluding Remarks Part IV

7 Conclusions and Future Work

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 tackles a few practical challenges facing distributed machine learning (ML) systems in different environments, namely in datacenters, in the cloud, and on edge devices. We first addressed the general problem of making distributed ML systems robust against arbitrary (i.e., Byzantine) failures. Then, we presented practical insights that help improve the training performance in each of the three environments we consider.

Chapter 3 presented GARFIELD, our *one-size-fits-all* library for Byzantine resilience. GARFIELD allows ML practitioners to write their training applications from a system's point of view rather than from an algorithmic perspective. GARFIELD also provides access to various robust *gradient aggregation rules (GARs),* which have been shown effective against arbitrary failures. With the ability to capture different architectures, network patterns, hardware capabilities, and data distributions, we demonstrated how GARFIELD can achieve Byzantine resilience in multiple setups. Essentially, GARFIELD helps quantify the benefits and the costs of Byzantine resilience in practice. We have integrated GARFIELD with two popular ML frameworks: TensorFlow and PyTorch. GARFIELD is an open-source project that has already been used multiple times in different projects at EPFL. In addition, the center for digital trust (C4DT)[1] has featured it as a secure ML solution against arbitrary failures. A demo on GARFIELD (built by C4DT) can be found at https://factory.c4dt.org/incubator/garfield/demo/.

Chapter 4 showed how *elasticity* can improve training time and resource utilization on datacenters' shared ML clusters. We demonstrated this with our elastic resource allocator (ERA), using which we showed the benefits of dual scaling by dynamically adapting (a) the number of workers and (b) the batch size. Our evaluation of ERA with a 64-GPU cluster shows that it

---

[1] https://show-new.c4dt.org/showcase/garfield/presentation

improves the throughput of training jobs, minimizes the average job completion time, and maximizes resource utilization, notably without hampering the training accuracy.

Chapter 5 showed how *cloud object stores* can contribute to ML computation and the implication of this contribution in alleviating the typical network bottleneck of training in the cloud. Essentially, the idea is to partially offload the ML computation to the storage tier to reduce the data transfer between storage and compute. We identified transfer learning (TL) as the killer application for this setup. With HAPI, our TL processing system that splits the computation between the storage and the compute tiers, we showed improvements in throughput, training time, communication load, and scalability while being completely transparent to the user.

Chapter 6 showed how to robustly and efficiently train generative adversarial networks (GANs) in the federated learning (FL) paradigm using FEGAN. The key to FEGAN's efficiency is to co-locate both components of the GAN (the generator and the discriminator) on all devices contributing to training. FEGAN also relies on KL weighting and balanced sampling to cope with heterogeneous devices and skewed datasets. We showed that FEGAN scales well with hundreds of devices while also achieving better learning quality.

In summary, this thesis provides practical insights that advance the state of the art of robust and efficient distributed ML systems in different environments. We have open-sourced the code of GARFIELD and FEGAN, and both of them went through the *Artifact Evaluation* process in their respective conferences and were awarded a couple of *ACM badges*[2] accordingly. We have seen some of our prototypes have already been extended further by the community.

7.2 Future Directions

While this thesis touched on crucial and practical problems for diverse settings of distributed training, research in this area is young and more efforts are needed to extend it further. In this section, we give some ideas to extend the work of this thesis.

Elastic training with cloud object stores. In Chapter 5, we show how we can use cloud object stores to improve training time, and in Chapter 4, we demonstrate how elasticity can help adapt training to existing resources, further expediting the training time. An interesting future direction is to see the result of integrating these (seemingly-orthogonal) two ideas. First, consider an extension to our HAPI framework (i.e., by processing the first part of a neural network on the object store) that uses multiple workers on the compute tier. An interesting question then is: how would that extension affect the splitting algorithm? Intuitively, this extension will release the stress from the storage tier as the compute tier can handle more computation with more workers. A second step would be to explore how *elasticity* affects the optimal solution. Indeed, the server load can play a role in scaling decisions. For example, if the server is serving many requests, it might decide to decrease its computation load and ask

---

[2] https://www.acm.org/publications/policies/artifact-review-badging

the elastic controller to increase the number of workers running training on the compute tier. Exploring the tradeoffs in this setup is a promising research direction.

Elastic Byzantine resilience. So far, all the research on Byzantine-resilient ML assumes a fixed set of participants throughout the training process. This assumption might not hold in permission-less systems, e.g., if random devices on the Internet want to train a common model collaboratively. Keeping such cases in mind and with the benefits of elasticity we have shown in Chapter 4, we need to explore how our resilience guarantees change with elastic training. For example, should workers keep a history of participants in previous rounds for better Byzantine resilience guarantees? If we model the pattern of participation of Byzantine workers (based on the elasticity decisions), can we design faster algorithms with the same resilience guarantees? What would be the correct assumptions on the number of Byzantine participants in that case? Achieving Byzantine resilience in permission-less settings is a plausible direction for Byzantine ML research.

Byzantine-resilient federated GANs. In Chapter 3, we show how to use GARFIELD to achieve Byzantine resilience in different settings. All these settings, though, focus only on deep networks training. An important research question is: how can we extend this work to training GANs, e.g., using FEGAN? Although we can straightforwardly apply the same approaches of GARFIELD to FEGAN, the effects of the robust algorithms of GARFIELD on GAN-specific problems (such as mode collapse and vanishing gradients) are not clear. Introducing Byzantine resilience to training GANs (especially those trained in the federated learning setup like FEGAN) is a crucial next research step.

Extending GARFIELD. While Chapter 3 shows how GARFIELD supports different environments and architectures, Byzantine resilience is far from being practical in some cases. For instance, data is typically not identical on workers contributing to training in the real world. Existing solutions require a prohibitive number of communication messages to cope with that while providing Byzantine resilience [69]. Coming up with better algorithms and baking them into GARFIELD would be an interesting future direction. For example, one idea that improves the communication complexity (in the case of training neural networks) is to pipeline communication with gradient computation [278, 102, 192]. This approach would decrease the idle time, improving the throughput of the whole system. Yet, this improvement comes at the expense of breaking transparency [121]. The same applies to traffic compression and sparsification methods [214, 76, 262, 162]: while such methods would reduce the communication complexity, it is not obvious how these methods change the resilience guarantees. Another direction is to enhance GARFIELD with privacy-preserving techniques, e.g., [106]. While an EPFL project has already started doing that,[3] further research is still needed to make it fully practical.

---

[3] https://github.com/LPD-EPFL/Garfield/tree/Secure/tensorflow_impl/applications/Secure

Extending ERA. In Chapter 4, we focused on specific aspects of ERA, namely the benefits of our dual scaling in boosting training and cluster performance. We give three directions to extend ERA. First, we can enhance other aspects of ERA using existing techniques in the literature on ML jobs scheduling (see Section 4.6), including time-slicing of GPUs, packing multiple jobs on the same GPU, and migrating jobs to specific GPUs. Moreover, we can improve ERA by using new trends for multi-user support on GPUs (e.g., the MIG feature from NVIDIA [54]) to better utilize the cluster resources and share cluster GPUs. Integrating these techniques with ERA is not straightforward and can result in compelling contributions on their own. Another limitation of ERA is that it relies on the stop-resume approach of PyTorch-elastic (for scaling), which might not be the best technique for scaling [115]. Since ERA's design is orthogonal to the elastic engine choice, integrating it with other sophisticated engines is a reasonable next step. Third, we consider a homogeneous cluster: we assume that all GPUs are alike and network connections in the cluster are reliable and have the same speed. This assumption might not hold in some cases. Assuming a heterogeneous cluster further complicates the problem. In this case, ERA should utilize the locality of the GPUs and should take more fine-grained measurements for the optimal allocation plan. Extending ERA to the heterogeneous cluster case is also a natural research direction.

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*, 2016.

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

[3] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review*, 39:59–74, 2005.

[4] Ahmed M Abdelmoniem, Chen-Yu Ho, Pantelis Papageorgiou, and Marco Canini. Empirical analysis of federated learning in heterogeneous environments. In *Proceedings of the 2nd European Workshop on Machine Learning and Systems*, pages 1–9, 2022.

[5] Ahmed M Abdelmoniem, Atal Narayan Sahu, Marco Canini, and Suhaib A Fahmy. Resource-efficient federated learning. *arXiv preprint arXiv:2111.01108*, 2021.

[6] Alim Al Ayub Ahmed, Ayman Aljabouh, Praveen Kumar Donepudi, and Myung Suh Choi. Detecting fake news using machine learning: A systematic literature review. *arXiv preprint arXiv:2102.04458*, 2021.

[7] Michael Ahn et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

[8] Monther Aldwairi and Ali Alwahedi. Detecting fake news in social media networks. *Procedia Computer Science*, 141:215–222, 2018.

[9] Adlich Alex et al. Driving impact at scale from automation and AI. https://mck.co/3IQIBVi, 2019. Last accessed: Apr, 2022.

[10] Alibaba. Alibaba OSS. https://www.alibabacloud.com/product/object-storage-service. Last accessed: Dec, 2021.

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

[12] Zeyuan Allen-Zhu and Elad Hazan. Variance reduction for faster non-convex optimization. In *International conference on machine learning*, pages 699–707, 2016.

[13] Amazon. Amazon S3. https://aws.amazon.com/s3/. Last accessed: Dec, 2021.

[14] Amazon. Amazon SageMaker JumpStart. https://docs.aws.amazon.com/sagemaker/latest/dg/studio-jumpstart.html. Last accessed: Apr, 2022.

[15] Amazon. Introducing Amazon S3 Object Lambda – Use Your Code to Process Data as It Is Being Retrieved from S3. https://aws.amazon.com/blogs/aws/introducing-amazon-s3-object-lambda-use-your-code-to-process-data-as-it-is-being-retrieved-from-s3/. Last accessed: Dec, 2021.

[16] AMD. Amd ai hardware. https://www.amd.com/en/graphics/servers-instinct-deep-learning.

[17] Syed Muhammad Anwar, Muhammad Majid, Adnan Qayyum, Muhammad Awais, Majdi Alnowami, and Muhammad Khurram Khan. Medical image analysis using convolutional neural networks: a review. *Journal of medical systems*, 42(11):1–13, 2018.

[18] Apple. Apple unveils m1 ultra, the worlds most powerful chip for a personal computer. https://www.apple.com/newsroom/2022/03/apple-unveils-m1-ultra-the-worlds-most-powerful-chip-for-a-personal-computer/.

[19] Amazon AQUA. AWS Advanced Query Accelerator. https://aws.amazon.com/redshift/features/aqua/.

[20] Martin Arjovsky and Léon Bottou. Towards principled methods for training generative adversarial networks, 2017.

[21] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja uszczak, et al. Delta lake: high-performance acid table storage over cloud object stores. *Proceedings of the VLDB Endowment*, 13(12):3411–3424, 2020.

[22] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 472–487, 2022.

[23] Sean Augenstein, H Brendan McMahan, Daniel Ramage, Swaroop Ramaswamy, Peter Kairouz, Mingqing Chen, Rajiv Mathews, et al. Generative models for effective ml on private, decentralized datasets. *arXiv preprint arXiv:1911.06679*, 2019.

[24] Isaac Bankman. *Handbook of medical image processing and analysis*. Elsevier, 2008.

[25] Gilad Baruch, Moran Baruch, and Yoav Goldberg. A little is enough: Circumventing defenses for distributed learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[26] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

[27] Jeremy Bernstein, Jiawei Zhao, Kamyar Azizzadenesheli, and Anima Anandkumar. signSGD with majority vote is communication efficient and fault tolerant. In *International Conference on Learning Representations*, 2018.

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

[29] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317–331, 2018.

[30] Reuben Binns, Michael Veale, Max Van Kleek, and Nigel Shadbolt. Like trainer, like bot? inheritance of bias in algorithmic content moderation. In *International conference on social informatics*, pages 405–415. Springer, 2017.

[31] Chris M Bishop. Neural networks and their applications. *Review of scientific instruments*, 65(6):1803–1832, 1994.

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

[33] AWS News Blog. S3 Select and Glacier Select – Retrieving Subsets of Objects. https://aws.amazon.com/blogs/aws/s3-glacier-select/. Last accessed: Apr, 2022.

[34] Cara Bloom, Joshua Tan, Javed Ramjohn, and Lujo Bauer. Self-driving cars and data collection: Privacy perceptions of networked autonomous vehicles. In *Thirteenth Symposium on Usable Privacy and Security ({SOUPS} 2017)*, pages 357–375, 2017.

[35] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for federated learning on user-held data. *arXiv preprint arXiv:1611.04482*, 2016.

[36] 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 *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1175–1191. ACM, 2017.

[37] Ali Borji. Pros and cons of gan evaluation measures. *Computer Vision and Image Understanding*, 179:41–65, 2019.

[38] Amaury Bouchra Pilet, Davide Frey, and François Taïani. Simple, efficient and convenient decentralized multi-task learning for neural networks. In *International Symposium on Intelligent Data Analysis*, pages 37–49. Springer, 2021.

[39] Andrew Brock, Jeff Donahue, and Karen Simonyan. Large scale GAN training for high fidelity natural image synthesis. In *International Conference on Learning Representations*, 2019.

[40] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[41] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.

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

[43] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 143–157, New York, NY, USA, 2011. Association for Computing Machinery.

[44] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. {POLARDB} meets computational storage: Efficiently support analytical workloads in {Cloud-Native} relational database. In *18th USENIX Conference on File and Storage Technologies (FAST)*, 2020.

[45] Chan Jung Chang, Jerry Chou, Yu-Ching Chou, and I-Hsin Chung. Ecs2: A fast erasure coding library for gpu-accelerated storage systems with parallel & direct io. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020.

[46] Tong Che, Yanran Li, Athul Paul Jacob, Yoshua Bengio, and Wenjie Li. Mode regularized generative adversarial networks. In *ICLR*, 2017.

[47] 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.

[48] Tianlong Chen, Sijia Liu, Shiyu Chang, Yu Cheng, Lisa Amini, and Zhangyang Wang. Adversarial robustness: From self-supervised pre-training to fine-tuning. In *Proceedings*

of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 699–708, 2020.

[49] Alvin Cheung, Owen Arden, Samuel Madden, and Andrew C Myers. Automatic partitioning of database applications. *Proceedings of the VLDB Endowment*, 5(11), 2012.

[50] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, Broomfield, CO, October 2014. USENIX Association.

[51] 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.

[52] Ting-Wu Chin, Cha Zhang, and Diana Marculescu. Renofeation: A simple transfer learning method for improved adversarial robustness. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3243–3252, 2021.

[53] Dooseop Choi, Taeg-Hyun An, Kyounghwan Ahn, and Jeongdan Choi. Driving experience transfer method for end-to-end control of self-driving cars. *arXiv preprint arXiv:1809.01822*, 2018.

[54] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.

[55] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[56] Alibaba Cloud. Real-Time Image Processing by Object Storage Service . https://www.alibabacloud.com/blog/real-time-image-processing-by-object-storage-service_597996. Last accessed: Dec, 2021.

[57] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 177–190. USENIX Association, 2006.

[58] 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 *ACM MobiSys 2010*, June 2010.

[59] Harshit Daga, Patrick K. Nicholson, Ada Gavrilovska, and Diego Lugones. Cartel: A system for collaborative transfer learning at the edge. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 25–37, New York, NY, USA, 2019. Association for Computing Machinery.

[60] Georgios Damaskinos, El-Mahdi El-Mhamdi, Rachid Guerraoui, Arsany Guirguis, and Sébastien Rouault. Aggregathor: Byzantine machine learning via robust gradient aggregation. In *Proceedings of Machine Learning and Systems*, volume 1, pages 81–106, 2019.

[61] 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.

[62] Georgios Damaskinos, Sébastien Rouault, and Arsany Guirguis. Aggregathor source code. https://github.com/LPD-EPFL/AggregaThor.

[63] Jeff Dean. Designs, lessons and advice from building large distributed systems, 2009.

[64] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

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

[66] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[67] Ilias Diakonikolas, Gautam Kamath, Daniel M Kane, Jerry Li, Ankur Moitra, and Alistair Stewart. Robustly learning a gaussian: Getting optimal error, efficiently. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2683–2702. Society for Industrial and Applied Mathematics, 2018.

[68] Ishan Durugkar, Ian Gemp, and Sridhar Mahadevan. Generative multi-adversarial networks. *arXiv preprint arXiv:1611.01673*, 2016.

[69] El Mahdi El-Mhamdi, Sadegh Farhadkhani, Rachid Guerraoui, Arsany Guirguis, Lê-Nguyên Hoang, and Sébastien Rouault. Collaborative learning in the jungle (decentralized, byzantine, heterogeneous, asynchronous and nonconvex learning). *Advances in Neural Information Processing Systems*, 34, 2021.

[70] El-Mahdi El-Mhamdi, Rachid Guerraoui, Arsany Guirguis, Lê Nguyên Hoang, and Sébastien Rouault. Genuinely distributed byzantine machine learning. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 355–364, 2020.

[71] El-Mahdi El-Mhamdi, Rachid Guerraoui, Arsany Guirguis, Lê-Nguyên Hoang, and Sébastien Rouault. Genuinely distributed byzantine machine learning (extended version). *Distributed Computing*, pages 1–27, 2022.

[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. Distributed momentum for byzantine-resilient learning. *arXiv preprint arXiv:2003.00010*, 2020.

[74] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542(7639):115, 2017.

[75] Vance Faber. Clustering and the continuous k-means algorithm. *Los Alamos Science*, 22(138144.21):67, 1994.

[76] Jiawei Fei, Chen-Yu Ho, Atal N Sahu, Marco Canini, and Amedeo Sapio. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 676–691, 2021.

[77] Clement Fung, Chris JM Yoon, and Ivan Beschastnikh. The limitations of federated learning in sybil settings. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*, pages 301–316, 2020.

[78] Zhe Gan, Yen-Chun Chen, Linjie Li, Chen Zhu, Yu Cheng, and Jingjing Liu. Large-scale adversarial training for vision-and-language representation learning. *Advances in Neural Information Processing Systems*, 33:6616–6628, 2020.

[79] Amirmasoud Ghiassi, Taraneh Younesian, Zhilong Zhao, Robert Birke, Valerio Schiavoni, and Lydia Y Chen. Robust (deep) learning framework against dirty labels and beyond. In *2019 First IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, pages 236–244. IEEE, 2019.

[80] Tarleton Gillespie. Content moderation, ai, and the question of scale. *Big Data & Society*, 7(2):2053951720943234, 2020.

[81] Justin Gilmer, Luke Metz, Fartash Faghri, Samuel S Schoenholz, Maithra Raghu, Martin Wattenberg, and Ian Goodfellow. Adversarial spheres. *arXiv preprint arXiv:1801.02774*, 2018.

[82] Christos Gkantsidis, Dimitrios Vytiniotis, Orion Hodson, Dushyanth Narayanan, Florin Dinu, and Ant Rowstron. Rhea: automatic filtering for unstructured cloud storage. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’13)*. USENIX, April 2013.

[83] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[84] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[85] Google. Google Cloud AutoML. https://cloud.google.com/automl/. Last accessed: Apr, 2022.

[86] Google. Google Cloud Storage. https://cloud.google.com/storage. Last accessed: Dec, 2021.

[87] Google. Tensor processing units. https://cloud.google.com/tpu/docs/tpus.

[88] 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.

[89] Grid5000. Grid5000. https://www.grid5000.fr/.

[90] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A {GPU} cluster manager for distributed deep learning. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 485–500, 2019.

[91] Rachid Guerraoui, Arsany Guirguis, Anne-Marie Kermarrec, and Erwan Le Merrer. Fegan: Scaling distributed gans. In *Proceedings of the 21st International Middleware Conference*, pages 193–206, 2020.

[92] Rachid Guerraoui, Arsany Guirguis, Jérémy Plassmann, Anton Ragot, and Sébastien Rouault. Garfield: System support for byzantine machine learning (regular paper). In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 39–51. IEEE, 2021.

[93] Arsany Guirguis. FEGANsource code. https://github.com/LPD-EPFL/FeGAN.

[94] Arsany Guirguis, Florin Dinu, Diana Petrescu, Do Le Quoc, Javier Picorel, and Rachid Guerraoui. Accelerating transfer learning with cloud object stores. *Under submission*, 2022.

[95] Arsany Guirguis, Yitzchak Lockerman, Chengjian Zhheng, David Eis, and Rachid Guerraoui. Elastic resource allocation for multi-tenant deep learning clusters. *Under submission*, 2022.

[96] Arsany Guirguis, Jérémy Plassmann, Anton Ragot, and Sébastien Rouault. GARFIELD source code. https://github.com/LPD-EPFL/Garfield.

[97] Walker Haddock, Matthew L Curry, Purushotham V Bangalore, and Anthony Skjellum. Gpu erasure coding for campaign storage. In *International Conference on High Performance Computing*, 2017.

[98] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[99] 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.

[100] Corentin Hardy, Erwan Le Merrer, and Bruno Sericola. Gossiping gans. In *Proceedings of the Second Workshop on Distributed Infrastructures for Deep Learning: DIDL*, volume 22, 2018.

[101] Corentin Hardy, Erwan "Le Merrer", and Bruno Sericola. Md-gan: Multi-discriminator generative adversarial networks for distributed datasets. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 866–877. IEEE, 2019.

[102] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288*, 2018.

[103] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629. IEEE, 2018.

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

[105] Lie He, An Bian, and Martin Jaggi. Cola: Decentralized linear learning. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 4536–4546. Curran Associates, Inc., 2018.

[106] Lie He, Sai Praneeth Karimireddy, and Martin Jaggi. Secure byzantine-robust machine learning. *arXiv preprint arXiv:2006.04747*, 2020.

[107] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[108] Quan Hoang, Tu Dinh Nguyen, Trung Le, and Dinh Phung. Multi-generator generative adversarial nets. *arXiv preprint arXiv:1708.02556*, 2017.

[109] Matthew B Hoy. Alexa, siri, cortana, and more: an introduction to voice assistants. *Medical reference services quarterly*, 37(1):81–88, 2018.

[110] 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.

[111] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[112] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[113] Huawei. Huawei launches ascend 910, the world's most powerful ai processor, and mindspore, an all-scenario ai computing framework. https://www.huawei.com/en/news/2019/8/huawei-ascend-910-most-powerful-ai-processor.

[114] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. {ZooKeeper}: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[115] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. Elastic resource sharing for distributed deep learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 721–739. USENIX Association, April 2021.

[116] IBM. IBM Storage supports NVIDIA DGX A100 AI infrastructure. https://www.ibm.com/blogs/systems/ibm-storage-supports-nvidia-dgx-a100-ai-infrastructure/. Last accessed: Dec, 2021.

[117] IBM. Optimize running NVIDIA GPU-enabled AI workloads with data orchestration solution. https://community.ibm.com/community/user/storage/blogs/pallavi-galgali1/2020/10/05/optimize-running-nvidia-gpu-enabled-ai-workloads-w?CommunityKey=1142f81e-95e4-4381-95d0-7977f20d53fa&tab=recentcommunityblogsdashboard. Last accessed: Apr, 2022.

[118] Facebook Incubator. Gloo. https://github.com/facebookincubator/gloo.

[119] Intel. Intel ai hardware. https://www.intel.com/content/www/us/en/artificial-intelligence/hardware.html.

[120] Alexander Isenko, Ruben Mayer, Jeffrey Jedele, and Hans-Arno Jacobsen. Where is my training bottleneck? hidden trade-offs in deep learning preprocessing pipelines. *arXiv preprint arXiv:2202.08679*, 2022.

[121] Nikita Ivkin, Daniel Rothchild, Enayat Ullah, Vladimir Braverman, Ion Stoica, and Raman Arora. Communication-efficient distributed sgd with sketching. *arXiv preprint arXiv:1903.04488*, 2019.

[122] KR Jayaram, Vinod Muthusamy, Parijat Dube, Vatche Ishakian, Chen Wang, Benjamin Herta, Scott Boag, Diana Arroyo, Asser Tantawi, Archit Verma, et al. Ffdl: A flexible multi-tenant deep learning platform. In *Proceedings of the 20th International Middleware Conference*, pages 82–95, 2019.

[123] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant {GPU} clusters for {DNN} training workloads. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 947–960, 2019.

[124] Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang. Multi-tenant gpu clusters for deep learning workloads: Analysis and implications. *Technical report, Microsoft Research*, 2018.

[125] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[126] Yihan Jiang, Jakub Konečnỳ, Keith Rush, and Sreeram Kannan. Improving federated learning personalization via model agnostic meta learning. *arXiv preprint arXiv:1909.12488*, 2019.

[127] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479. USENIX Association, November 2020.

[128] Yuchen Jin, Tianyi Zhou, Liangyu Zhao, Yibo Zhu, Chuanxiong Guo, Marco Canini, and Arvind Krishnamurthy. Autolrs: Automatic learning-rate schedule by bayesian optimization on the fly. *arXiv preprint arXiv:2105.10762*, 2021.

[129] Tyler Johnson, Pulkit Agrawal, Haijie Gu, and Carlos Guestrin. Adascale sgd: A user-friendly algorithm for distributed training. In *International Conference on Machine Learning*, pages 4911–4920. PMLR, 2020.

[130] M. Kachelrieß. Branchless vectorized median filtering. In *2009 IEEE Nuclear Science Symposium Conference Record (NSS/MIC)*, pages 4099–4105, Oct 2009.

[131] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News*, 45(1):615–629, 2017.

[132] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT*, pages 4171–4186, 2019.

[133] Salman Khan, Muzammal Naseer, Munawar Hayat, Syed Waqas Zamir, Fahad Shahbaz Khan, and Mubarak Shah. Transformers in vision: A survey. *ACM Computing Surveys (CSUR)*, 2021.

[134] Douwe Kiela, Hamed Firooz, Aravind Mohan, Vedanuj Goswami, Amanpreet Singh, Pratik Ringshia, and Davide Testuggine. The hateful memes challenge: Detecting hate speech in multimodal memes. *Advances in Neural Information Processing Systems*, 33:2611–2624, 2020.

[135] Larry Kim. How many ads does google serve in a day?, 2012.

[136] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[137] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. Understanding ephemeral storage for serverless analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[138] Jakub Konečný, Brendan McMahan, and Daniel Ramage. Federated optimization: Distributed optimization beyond the datacenter. *arXiv preprint arXiv:1511.03575*, 2015.

[139] Jakub Konečný, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

[140] Gunjae Koo, Kiran Kumar Matam, Te I., H.V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: Trading communication with computing near storage. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[141] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar dataset. https://www.cs.toronto.edu/~kriz/cifar.html.

[142] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Neural Information Processing Systems*, pages 1097–1105, 2012.

[143] Ben Kröse, Ben Krose, Patrick Van der Smagt, and Patrick Smagt. An introduction to neural networks, 1993.

[144] Kubernetes. Configmaps. https://kubernetes.io/docs/concepts/configuration/configmap/.

[145] Solomon Kullback. *Information theory and statistics*. Courier Corporation, 1997.

[146] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[147] Karol Kurach, Mario Lučić, Xiaohua Zhai, Marcin Michalski, and Sylvain Gelly, editors. *A Large-Scale Study on Regularization and Normalization in GANs*, 2019.

[148] San Francisco Laboratory of Cell Geometry, University of California. Pytorch model size estimator. https://github.com/jacobkimmel/pytorch_modelsize.

[149] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[150] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *TOPLAS*, 4(3):382–401, 1982.

[151] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[152] Yann Lecun, Corinna Cortes, and Christopher J.C. Burges. Mnist dataset. http://yann.lecun.com/exdb/mnist/.

[153] C. Ledig, L. Theis, F. Huszar, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi. Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network. *CVPR*, 2017.

[154] George Leopold. Nvidia buys object storage vendor swiftstack. https://www.enterpriseai.news/2020/03/06/nvidia-buys-object-storage-vendor-swiftstack/.

[155] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics. *Proceedings of the VLDB Endowment*, 9(14):1647–1658, 2016.

[156] 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*, volume 1, page 3, 2014.

[157] 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*, 2014.

[158] 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.

[159] Yuezun Li and Siwei Lyu. Exposing deepfake videos by detecting face warping artifacts. In *IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2019.

[160] Ming Liang and Xiaolin Hu. Recurrent convolutional neural network for object recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3367–3375, 2015.

[161] Kunal Lillaney, Vasily Tarasov, David Pease, and Randal Burns. The case for dual-access file systems over object storage. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[162] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 3lc: Lightweight and effective traffic compression for distributed machine learning. *arXiv preprint arXiv:1802.07389*, 2018.

[163] Wei Yang Bryan Lim, Nguyen Cong Luong, Dinh Thai Hoang, Yutao Jiao, Ying-Chang Liang, Qiang Yang, Dusit Niyato, and Chunyan Miao. Federated learning in mobile edge networks: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 2020.

[164] Haibin Lin, Hang Zhang, Yifei Ma, Tong He, Zhi Zhang, Sheng Zha, and Mu Li. Dynamic mini-batch sgd for elastic distributed training: Learning in the limbo of resources. *arXiv preprint arXiv:1904.12043*, 2019.

[165] Bo Liu, Ming Ding, Sina Shaham, Wenny Rahayu, Farhad Farokhi, and Zihuai Lin. When machine learning meets privacy: A survey and outlook. *ACM Computing Surveys (CSUR)*, 54(2):1–36, 2021.

[166] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

[167] Irene Lopatovska, Katrina Rink, Ian Knight, Kieran Raines, Kevin Cosenza, Harriet Williams, Perachya Sorsche, David Hirsch, Qi Li, and Adrianna Martinez. Talk to me: Exploring user interactions with the amazon alexa. *Journal of Librarianship and Information Science*, 51(4):984–997, 2019.

[168] Mario Lucic, Karol Kurach, Marcin Michalski, Sylvain Gelly, and Olivier Bousquet. Are gans created equal? a large-scale study. In *Advances in neural information processing systems*, pages 700–709, 2018.

[169] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient {GPU} cluster scheduling. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 289–304, 2020.

[170] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. Kungfu: Making training in distributed machine learning adaptive. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 937–954, 2020.

[171] Shervin Malmasi and Marcos Zampieri. Detecting hate speech in social media. *arXiv preprint arXiv:1712.06427*, 2017.

[172] Yishay Mansour, Mehryar Mohri, Jae Ro, and Ananda Theertha Suresh. Three approaches for personalization with applications to federated learning. *arXiv preprint arXiv:2002.10619*, 2020.

[173] Xudong Mao, Qing Li, Haoran Xie, Raymond YK Lau, Zhen Wang, and Stephen Paul Smolley. Least squares generative adversarial networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2794–2802, 2017.

[174] Iacopo Masi, Yue Wu, Tal Hassner, and Prem Natarajan. Deep face recognition: A survey. In *2018 31st SIBGRAPI conference on graphics, patterns and images (SIBGRAPI)*, pages 471–478. IEEE, 2018.

[175] H Brendan McMahan, Gary Holt, David Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, et al. Ad click prediction: a view from the trenches. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM, 2013.

[176] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *AISTATS*, 2017.

[177] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *JMLR*, 17(1):1235–1241, 2016.

[178] Stuart Mitchell, Anita Kean, Andrew Mason, Michael O'Sullivan, Antony Phillips, and Franco Peschiera. Optimization with pulp. https://coin-or.github.io/pulp/.

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

[180] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. *arXiv preprint arXiv:2104.05158*, 2021.

[181] Luis Muñoz-González, Kenneth T Co, and Emil C Lupu. Byzantine-robust federated machine learning through adaptive model averaging. *arXiv preprint arXiv:1909.05125*, 2019.

[182] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. Tf.data: A machine learning data processing framework. *Proc. VLDB Endow.*, 14(12):2945–2958, jul 2021.

[183] David R Musser. Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993, 1997.

[184] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[185] Behnam Neyshabur, Srinadh Bhojanapalli, and Ayan Chakrabarti. Stabilizing gan training with multiple random projections. *arXiv preprint arXiv:1705.07831*, 2017.

[186] Takayuki Nishio and Ryo Yonetani. Client selection for federated learning with heterogeneous resources in mobile edge. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2019.

[187] NVIDIA. GPUDirect Storage: A Direct Path Between Storage and GPU Memory. https://developer.nvidia.com/blog/gpudirect-storage/. Last accessed: Apr, 2022.

[188] NVIDIA. Nccl. https://developer.nvidia.com/nccl, 2017.

[189] Openstack. OpenStack Swift. https://docs.openstack.org/swift/latest/. Last accessed: Dec, 2021.

[190] Andrew Or, Haoyu Zhang, and Michael Freedman. Resource elasticity in distributed deep learning. *Proceedings of Machine Learning and Systems*, 2:400–411, 2020.

[191] 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.

[192] Jay H Park, Sunghwan Kim, Jinwon Lee, Myeongjae Jeon, and Sam H Noh. Accelerated training for cnn distributed deep learning through automatic resource-aware layer placement. *arXiv preprint arXiv:1901.05803*, 2019.

[193] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[194] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[195] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[196] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[197] Ivens Portugal, Paulo Alencar, and Donald Cowan. The use of machine learning algorithms in recommender systems: A systematic review. *Expert Systems with Applications*, 97:205–227, 2018.

[198] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, February 2019. USENIX Association.

[199] PyTorch. Torch distributed elastic. https://pytorch.org/docs/stable/distributed.elastic.html.

[200] PyTorch. Torchelastic controller for kubernetes. https://github.com/pytorch/elastic/tree/master/kubernetes.

[201] 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 *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 631–644, Boston, MA, July 2018. USENIX Association.

[202] Aurick Qiao, Bryon Aragam, Bingjing Zhang, and Eric Xing. Fault tolerance in iterative-convergent machine learning. In *International Conference on Machine Learning*, pages 5220–5230, 2019.

[203] Minghui Qiu, Peng Li, Chengyu Wang, Haojie Pan, Ang Wang, Cen Chen, Xianyan Jia, Yaliang Li, Jun Huang, Deng Cai, et al. Easytransfer: A simple and scalable deep transfer learning platform for nlp applications. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, pages 4075–4084, 2021.

[204] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. Odessa: Enabling interactive perception applications on mobile devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, page 43–56, New York, NY, USA, 2011. Association for Computing Machinery.

[205] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[206] Maithra Raghu, Chiyuan Zhang, Jon Kleinberg, and Samy Bengio. Transfusion: Understanding transfer learning for medical imaging. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2019.

[207] Shashank Rajput, Hongyi Wang, Zachary Charles, and Dimitris Papailiopoulos. Detox: A redundancy-based framework for faster and more robust gradient aggregation. *Advances in Neural Information Processing Systems*, 32, 2019.

[208] Qing Rao and Jelena Frtunikj. Deep learning for self-driving cars: chances and challenges. In *2018 IEEE/ACM 1st International Workshop on Software Engineering for AI in Autonomous Systems (SEFAIAS)*, pages 35–38. IEEE, 2018.

[209] S. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele, and H. Lee. Generative Adversarial Text to Image Synthesis. *ArXiv e-prints*, May 2016.

[210] Sylvia Richardson and Peter J Green. On bayesian analysis of mixtures with an unknown number of components (with discussion). *Journal of the Royal Statistical Society: series B (statistical methodology)*, 59(4):731–792, 1997.

[211] Peter J Rousseeuw. Multivariate estimation with high breakdown point. *Mathematical statistics and applications*, 8:283–297, 1985.

[212] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[213] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[214] Atal Sahu, Aritra Dutta, Ahmed M Abdelmoniem, Trambak Banerjee, Marco Canini, and Panos Kalnis. Rethinking gradient sparsification as total error minimization. *Advances in Neural Information Processing Systems*, 34, 2021.

[215] Felix Sattler, Simon Wiedemann, Klaus-Robert Müller, and Wojciech Samek. Robust and communication-efficient federated learning from non-iid data. *IEEE transactions on neural networks and learning systems*, 2019.

[216] V. Saxena, K. R. Jayaram, S. Basu, Y. Sabharwal, and A. Verma. Effective elastic scaling of deep learning workloads. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8, Los Alamitos, CA, USA, nov 2020. IEEE Computer Society.

[217] Vaibhav Saxena, KR Jayaram, Saurav Basu, Yogish Sabharwal, and Ashish Verma. Effective elastic scaling of deep learning workloads. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8. IEEE, 2020.

[218] Linus E Schrage and Louis W Miller. The queue m/g/1 with the shortest remaining processing time discipline. *Operations Research*, 14(4):670–684, 1966.

[219] Christopher J Shallue, Jaehoon Lee, Joe Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E Dahl. Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600*, 2018.

[220] Yotam Shmargad and Samara Klar. Sorting the news: How ranking by popularity polarizes our politics. *Political Communication*, 37(3):423–446, 2020.

[221] Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In *SIGSAC*, pages 1310–1321, 2015.

[222] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 349–362, Hollywood, CA, October 2012. USENIX Association.

[223] Singh Chouhan Siddharth, Kaul Ajay, Pratap Singh Uday, and Jain Sanjeev. A database of leaf images: Practice towards plant conservation with plant pathology. Mendeley Data, 2019.

[224] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[225] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't decay the learning rate, increase the batch size. In *International Conference on Learning Representations*, 2018.

[226] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[227] Samuel L Smith and Quoc V Le. A bayesian perspective on generalization and stochastic gradient descent. *arXiv preprint arXiv:1710.06451*, 2017.

[228] Liuyihan Song, Pan Pan, Kang Zhao, Hao Yang, Yiming Chen, Yingya Zhang, Yinghui Xu, and Rong Jin. Large-scale training system for 100-million classification at alibaba. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2909–2930, 2020.

[229] Lili Su and Shahin Shahrampour. Finite-time guarantees for byzantine-resilient distributed state estimation with noisy measurements. *IEEE Transactions on Automatic Control*, 65(9):3758–3771, 2019.

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

[231] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016.

[232] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[233] Tesla. Tesla ai hardware. https://www.tesla.com/AI.

[234] Kjerstin Thorson, Kelley Cotter, Mel Medeiros, and Chankyung Pak. Algorithmic inference, political interest, and exposure to news and politics on facebook. *Information, Communication & Society*, 24(2):183–200, 2021.

[235] Aleksei Triastcyn and Boi Faltings. Federated generative privacy. *arXiv preprint arXiv:1910.08385*, 2019.

[236] Grant Van Horn, Oisin Mac Aodha, Yang Song, Yin Cui, Chen Sun, Alex Shepard, Hartwig Adam, Pietro Perona, and Serge Belongie. The inaturalist species classification and detection dataset. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[237] Paul Vanhaesebrouck, Aurélien Bellet, and Marc Tommasi. Decentralized collaborative learning of personalized models over networks. In *AISTATS*, 2017.

[238] Kenton Varda. Protocol buffers. https://github.com/protocolbuffers/protobuf.

[239] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, ukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[240] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S Rellermeyer. A survey on distributed machine learning. *ACM Computing Surveys (CSUR)*, 53(2):1–33, 2020.

[241] C. Vondrick, H. Pirsiavash, and A. Torralba. Generating Videos with Scene Dynamics. *ArXiv e-prints*, September 2016.

[242] Pooja Vyavahare, Lili Su, and Nitin H Vaidya. Distributed learning with adversarial agents under relaxed network condition. *arXiv preprint arXiv:1901.01943*, 2019.

[243] Chong Wang, Xi Chen, Alexander J Smola, and Eric P Xing. Variance reduction for stochastic gradient optimization. *Advances in Neural Information Processing Systems*, 26:181–189, 2013.

[244] Yaqing Wang, Quanming Yao, James T Kwok, and Lionel M Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM Computing Surveys (CSUR)*, 53(3):1–34, 2020.

[245] William Warner and Julia Hirschberg. Detecting hate speech on the world wide web. In *Proceedings of the second workshop on language in social media*, pages 19–26, 2012.

[246] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big data*, 3(1):1–40, 2016.

[247] Christopher KI Williams and Carl Edward Rasmussen. Gaussian processes for regression. In *Advances in neural information processing systems*, pages 514–520, 1996.

[248] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices. In *CVPR*, pages 4820–4828, 2016.

[249] Yidi Wu, Kaihao Ma, Xiao Yan, Zhi Liu, Zhenkun Cai, Yuzhen Huang, James Cheng, Han Yuan, and Fan Yu. Elastic deep learning in multi-tenant gpu clusters. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[250] Huang Xiao, Battista Biggio, Gavin Brown, Giorgio Fumera, Claudia Eckert, and Fabio Roli. Is feature selection secure against training data poisoning? In *ICML*, pages 1689–1698, 2015.

[251] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 595–610, 2018.

[252] Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. Generalized Byzantine-tolerant sgd. *arXiv preprint arXiv:1802.10116*, 2018.

[253] Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. Zeno: Byzantine-suspicious stochastic gradient descent. *arXiv preprint arXiv:1805.10032*, 2018.

[254] Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. Slsgd: Secure and efficient distributed on-device machine learning. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 213–228. Springer, 2019.

[255] Cong Xie, Oluwasanmi O Koyejo, and Indranil Gupta. Faster distributed synchronous sgd with weak synchronization. *OpenReview*, 2018.

[256] Cong Xie, Sanmi Koyejo, and Indranil Gupta. Fall of empires: Breaking byzantine-tolerant sgd by inner product manipulation. *arXiv preprint arXiv:1903.03936*, 2019.

[257] Cong Xie, Sanmi Koyejo, and Indranil Gupta. Practical distributed learning: Secure machine learning with communication-efficient local updates. *arXiv preprint arXiv:1903.06996*, 2019.

[258] Cong Xie, Sanmi Koyejo, and Indranil Gupta. Zeno++: Robust fully asynchronous sgd. *arXiv preprint arXiv:1903.07020*, 2019.

[259] Lei Xie, Jidong Zhai, Baodong Wu, Yuanbo Wang, Xingcheng Zhang, Peng Sun, and Shengen Yan. Elan: Towards generic and efficient elastic training for deep learning. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 78–88. IEEE, 2020.

[260] Pengtao Xie. *Diversity-promoting and Large-scale Machine Learning for Healthcare*. PhD thesis, University of Pittsburgh Medical Center, 2018.

[261] Bangzhou Xin, Wei Yang, Yangyang Geng, Sheng Chen, Shaowei Wang, and Liusheng Huang. Private fl-gan: Differential privacy synthetic data generation based on federated learning. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2927–2931. IEEE, 2020.

[262] Hang Xu, Chen-Yu Ho, Ahmed M Abdelmoniem, Aritra Dutta, El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini, and Panos Kalnis. Grace: A compressed communication framework for distributed machine learning. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 561–572. IEEE, 2021.

[263] Shuotao Xu, Thomas Bourgeat, Tianhao Huang, Hojun Kim, Sungjin Lee, and Arvind Arvind. Aquoman: An analytic-query offloading machine. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 386–399. IEEE, 2020.

[264] Mengjia Yan, Mengao Zhao, Zining Xu, Qian Zhang, Guoli Wang, and Zhizhong Su. Vargfacenet: An efficient variable group convolutional neural network for lightweight face recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, pages 0–0, 2019.

[265] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. Federated machine learning: Concept and applications. *ACM Trans. Intell. Syst. Technol.*, 10(2):12:1–12:19, January 2019.

[266] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. Flexpushdowndb: hybrid pushdown and caching in a cloud dbms. *Proceedings of the VLDB Endowment*, 2021.

[267] Zhixiong Yang and Waheed U Bajwa. Bridge: Byzantine-resilient decentralized gradient descent. *arXiv preprint arXiv:1908.08098*, 2019.

[268] Zhixiong Yang and Waheed U Bajwa. Byrdie: Byzantine-resilient distributed coordinate descent for decentralized learning. *IEEE Transactions on Signal and Information Processing over Networks*, 5(4):611–627, 2019.

[269] Sarita Yardi, Daniel Romero, Grant Schoenebeck, et al. Detecting spam in a twitter network. *First monday*, 2010.

[270] Dong Yin, Yudong Chen, Kannan Ramchandran, and Peter Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates. *arXiv preprint arXiv:1803.01498*, 2018.

[271] Peifeng Yin, Ping Luo, and Taiga Nakamura. Small batch or large batch? gaussian walk with rebound can teach. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1275–1284, 2017.

[272] Peifeng Yin, Ping Luo, and Taiga Nakamura. Small batch or large batch? gaussian walk with rebound can teach. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 1275–1284, New York, NY, USA, 2017. Association for Computing Machinery.

[273] Ryo Yonetani, Tomohiro Takahashi, Atsushi Hashimoto, and Yoshitaka Ushiku. Decentralized learning of generative adversarial networks from multi-client non-iid data. *CoRR*, abs/1905.09684, 2019.

[274] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained gpu sharing primitives for deep learning applications. *arXiv preprint arXiv:1902.04610*, 2019.

[275] Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. Pushdowndb: Accelerating a dbms using s3 computation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1802–1805. IEEE, 2020.

[276] Zalando. Fashion-mnist dataset. https://research.zalando.com/welcome/mission/research-projects/fashion-mnist/.

[277] Zalando. Kopf: Kubernetes operators framework. https://kopf.readthedocs.io/en/stable/.

[278] 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.

[279] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C.M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. Hived: Sharing a GPU cluster for deep learning with guarantees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 515–532. USENIX Association, November 2020.

[280] Mark Zhao, Niket Agarwal, Aarti Basant, Bugra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, et al. Understanding and co-designing the data ingestion pipeline for industry-scale recsys training. *arXiv preprint arXiv:2108.09373*, 2021.

[281] Yue Zhao, Meng Li, Liangzhen Lai, Naveen Suda, Damon Civin, and Vikas Chandra. Federated learning with non-iid data. *CoRR*, abs/1806.00582, 2018.

[282] Tianyi Zhou, Shengjie Wang, and Jeff A Bilmes. Diverse ensemble evolution: Curriculum data-model marriage. In *Advances in Neural Information Processing Systems*, pages 5905–5916, 2018.

[283] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. A comprehensive survey on transfer learning. *CoRR*, abs/1911.02685, 2019.

Arsany Guirguis

Chemin de Veilloud 12,
1024 Ecublens Vd, Switzerland
(+41) 787325160
WWW: https://aguirguis.netlify.app
E-mail: arsany.guirguis91@gmail.com

EDUCATION

Ph.D. candidate, September 2017 - Now
School of Computer and Communication Sciences,
EPFL, Switzerland.
Thesis: System Support for Robust Distributed Learning.
Adviser: Prof. Rachid Guerraoui.

M.Sc., Computer and Systems Engineering, September 2014 - February 2017
Alexandria University, Egypt.
GPA: 4.00/4.00.
Thesis: Cooperation-based Routing Protocol for Cognitive Radio Networks.
Adviser: Prof. Mustafa ElNainay.

B.Sc., Computer and Systems Engineering, September 2009 - June 2014
Alexandria University, Egypt.
Rank: First, Distinction with degree of honor.
GPA: 3.97/4.00.
Graduation Project: BARDI: A Real Interactive Board.
Adviser: Prof. Mohamed Hussein.

CONFERENCE PAPERS

1. **Arsany Guirguis**, Yitzchak Lockerman, Chengjian Zheng, David Eis, Rachid Guerraoui, “Elastic Resource Allocation for Multi-tenant Deep Learning” Clusters, Under submission.
2. **Arsany Guirguis**, Florin Dinu, Diana Petrescu, Do Le Quoc, Javier Picorel, Rachid Guerraoui, “Accelerating Transfer Learning with Cloud Object Stores”, Under submission.
3. El Mahdi El Mhamdi[*], Sadegh Farhadkhani, Rachid Guerraoui, **Arsany Guirguis**, Lê Nguyên Hoang, and Sébastien Rouault, “Collaborative Learning in the Jungle”, Thirty-fifth Conference on Neural Information Processing Systems (NeurIPS) 2021.
4. Rachid Guerraoui[*], **Arsany Guirguis**, Jérémy Plassmann, Anton Ragot, and Sebastien Rouault, “Garfield: System Support for Byzantine Machine Learning”, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) 2021. **Acceptance rate 16.3%**.
5. Rachid Guerraoui[*], **Arsany Guirguis**, Anne-Marie Kermarrec, and Erwan Le Merrer, “FeGAN: Scaling Distributed GANs”, ACM/IFIP International Middleware Conference (Middleware) 2020.
6. El Mahdi El Mhamdi[*], Rachid Guerraoui, **Arsany Guirguis**, Lê Nguyên Hoang, and Sébastien Rouault, “Genuinely Distributed Byzantine Machine Learning”, ACM Symposium on Principles of Distributed Computing (PODC) 2020. [**Invited to the International Journal Distributed Computing**].
7. Georgios Damaskinos[*], El Mahdi El Mhamdi, Rachid Guerraoui, **Arsany Guir-**

---

[*] Authors are listed alphabetically.

guis, and Sébastien Rouault, "AGGREGATHOR: Byzantine Machine Learning via Robust Gradient Aggregation", The Conference on Systems and Machine Learning (MLSys/SysML) 2019. **Acceptance rate 16.9%**.

8. **Arsany Guirguis**, Mustafa El-Nainay, "Channel Selection Scheme for Cooperative Routing Protocols in Cognitive Radio Networks", IEEE ICNC 2017.
9. Samer S. Hanna, **Arsany Guirguis**, Mahmoud A. Mahdi, Yaser A. El-Nakieb, Mahmoud Alaa Eldin, Dina M. Saber, Moustafa Youssef, Mustafa Y. ElNainay, Amr A. El-Sherif, and Karim G. Seddik, "CRC: Collaborative Research and Teaching Testbed for Wireless Communications and Networks", ACM WiNTECH 2016 in conjunction with ACM MobiCom 2016.
10. **Arsany Guirguis**, Mohamed Ibrahim, Karim Seddik, Khaled Harras, Fadel Digham, and Moustafa Youssef, "Primary User Aware k -hop Routing for Cognitive Radio Networks", IEEE GLOBECOM 2015.
11. **Arsany Guirguis**, Raymond Guirguis, and Moustafa Youssef, "Primary User Aware Network Coding for Multi-hop Cognitive Radio Networks", IEEE GLOBECOM 2014.

JOURNAL PAPERS

1. El Mahdi El Mhamdi*, Rachid Guerraoui, **Arsany Guirguis**, Lê Nguyên Hoang, and Sébastien Rouault, "Genuinely Distributed Byzantine Machine Learning (extended version)", Distributed Computing, 2022.
2. **Arsany Guirguis**, Fadel Digham, Karim Seddik, Mohamed Ibrahim, Khaled Harras, and Moustafa Youssef, "Primary User-aware Optimal Discovery Routing for Cognitive Radio Networks", IEEE Transactions on Mobile Computing (TMC), 2018.
3. **Arsany Guirguis**, Mohammed Karmoose, Karim Habak, Mustafa ElNainay, and Moustafa Youssef, "Cooperation-based Routing in Cognitive Radio Networks", Elsevier Journal of Network and Computer Applications (JNCA), 2018.
4. Youssef Khazbak, Mostafa Izz, Tamer ElBatt, Abdulrahman Fahim, **Arsany Guirguis**, and Moustafa Youssef, "Cost-Effective Data Transfer for Mobile Healthcare", IEEE Systems Journal, 2016.

INVITED TALKS

1. System Support for Robust Machine Learning, KAUST, June 2022. Saudi Arabia.
2. Robust Machine Learning in the Jungle, Amazon ATS Tech Talk, March 2022. Amazon, Luxembourg.
3. Byzantine Machine Learning, Workshop on Systems (WOS9), December 2019. Inria Rennes.

EXPERIENCE

Bloomberg LP, London, UK
Data Science Research Intern May 2021 - August 2021

- Designed and built an elastic resource allocator for multi-tenant deep learning clusters.

Huawei Technologies Research Center, Munich, Germany
Ph.D. Visiting Researcher February 2021 - April 2021

- Explored the benefits of near data processing (NDP) for machine learning applications.

INRIA, Rennes, France
Visiting Scholar July 2019 - September 2019

- Designed and built a scalable system for distributed Generative Adversarial Networks.

Alexandria University, Alexandria, Egypt
Research Assistant August 2014 - May 2017

- Designed and implemented open-access testbed for Cognitive Radio Networks to be available for educational and research purposes.
- Designed and implemented new routing protocols for Cognitive Networks.

Egypt-Japan University of Science and Technology, Alexandria, Egypt
Undergraduate Research Assistant July 2013 - September 2013

- Designed and implemented a network coding flavored forwarding algorithm for Cognitive Radio Networks. This work was published in GLOBECOM 2014.

Egypt-Japan University of Science and Technology, Alexandria, Egypt
Undergraduate Research Assistant July 2012 - September 2012

- Implemented a routing protocol (SEARCH) for Cognitive Radio Networks over Click Modular Router. This work was a part of the CogFrame project.

HONORS AND AWARDS

1. **EPFL, Switzerland:** EDIC fellowship, September 2017.
2. **Alexandria University, Egypt:** University fund winner to attend IEEE ICNC 2017.
3. **Alexandria University, Egypt:** runner-up, outstanding teaching assistant in Computer and Systems Engineering Department, April 2016.
4. **Alexandria University, Egypt:** Prof. Khalil award, August 2014 (for ranking first on Computer and Systems Engineering Department).
5. **Alexandria University, Egypt:** Best Graduation Project award, July 2014.
6. **Microsoft Research Advanced Technology Lab, Cairo, Egypt:** Graduation Project fund winner, December 2013.

TECHNICAL SKILLS

- **Languages:** Python, Java, C/C++, P4.
- **Frameworks:** PyTorch, TensorFlow, Kubernetes, OpenStack Swift.

PROFESSIONAL SERVICES

- Reviewer for ICLR 2022 (**Highlighted Reviewer**), NeurIPS 2022.
- Artifact Evaluation Board member at the Journal of Systems Research 2021.
- Reviewer for IEEE Transactions on Parallel and Distributed Systems.
- Reviewer for IEEE Transactions on Dependable and Secure Computing.
- Reviewer for IEEE Systems Journal.
- Reviewer for Elsevier Computer Standards & Interfaces.
- Reviewer for IEEE Transactions on Vehicular Technology.
- Reviewer for IEEE International Wireless Communications and Mobile Computing Conference.
- Reviewer for IEEE Communications Letters.