# Minimalist Systems for Pervasive Machine Learning

by

Fan Lai

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2023

Doctoral Committee:

Associate Professor Mosharaf Chowdhury, Chair
Professor Aditya Akella, The University of Texas at Austin
Assistant Professor Raed Al Kontar
Associate Professor Harsha V. Madhyastha

Fan Lai

fanlai@umich.edu

ORCID iD: 0009-0005-0472-107X

© Fan Lai 2023

*To my family*

## *Acknowledgments*

Pursuing a PhD is a challenging yet very enjoyable experience. Over the past six years, I have been filled with great gratitude toward my advisor, family, friends, colleagues, and collaborators. Their help and support make both this moment possible and this period of my life transformative.

Foremost, I am immensely grateful to Professor Mosharaf Chowdhury and Professor Harsha V. Madhyastha, for their invaluable guidance, tremendous support, and generous freedom they granted me to work on the problems that seem most exciting to me. They instilled in me a deep appreciation for simplicity in design, solidity in theoretical backing, and clarity in presentation. Their insistence on doing great work has significantly elevated my research. Going beyond research, I am particularly grateful for their honest feedback that improved me as a resilient and conscientious person. I am truly fortunate to have enjoyed my PhD under their guidance.

The work in this dissertation is the culmination of many successful collaborations: Chapter 2 was joint work with Jie You, Xiangfeng Zhu, Harsha V. Madhyastha and Mosharaf Chowdhury [161]; Chapter 3 builds upon joint work with Yinwei Dai, Harsha V. Madhyastha and Mosharaf Chowdhury [159]; Chapter 4 includes materials from joint work with Wei Zhang, Rui Liu, Jongsoo Park, Jie You, Mosharaf Chowdhury, and many collaborators at Meta AI [162]; Chapters 5 and 6 highlight parts of the FedScale project developed with Yinwei Dai, Sanjay S. Singapuram, Jiachen Liu, Xiangfeng Zhu, Harsha V. Madhyastha, Mosharaf Chowdhury, and our LinkedIn partners [160].

I am thankful to my dissertation committee members, Aditya Akella and Raed Al Kontar, for their useful feedback, as well as many other wonderful collaborators: Kai Chen, Yiding Wang, Haizhong Zheng, Atul Prakash, Yiwen Zhang, Jiachen Liu, Jason Chen, Yuxi Hu, William Tsai, Sabin Devkota, Xiaohan Wei, Jianyu Huang, Naichen Shi, Boyi Chen, and Souvik Ghosh. With their help, I am very fortunate to have multiple systems deployed in big companies such as Meta and LinkedIn, which now cater to millions of users daily.

I would also like to extend my sincere thanks to everyone in the Symbiotic Lab: Dr. Juncheng Gu, Dr. Peifeng Yu, Dr. Jie You, Dr. Hasan Al Muraf, Yiwen Zhang, Jiachen Liu, Jae-Won Chung, Insu Jang, as well as our FedScale contributors Ewen Wang, Yile Gu, Yuxuan Zhu, and Chengsong Zhang. Our discussions have always been enlightening, and the shared moments during deadline pressures are unforgettable.

And finally, my deepest thanks go to my family and friends for their consistent support throughout my PhD journey. I'm truly blessed to have had such a supportive environment and to be a part of the University of Michigan. Forever Go Blue!

# TABLE OF CONTENTS

**Dedication** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **ii**

**Acknowledgments** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **iii**

**List of Figures** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **viii**

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

**List of Algorithms** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **xv**

**List of Appendices** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **xvi**

**Abstract** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **xvii**

**Chapter**

**1 Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1**

1.1 Background: Scaling Trends in Machine Learning . . . . . . . . . . . . . . . . . . 2
1.1.1 Lifecycle of Machine Learning Development . . . . . . . . . . . . . . . . 2
1.1.2 Machine Learning Development is Hitting Limits . . . . . . . . . . . . . . 3
1.2 Thesis Statement and Contributions . . . . . . . . . . . . . . . . . . . . . . . . 4
1.3 Dissertation Plan . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

**2 Minimizing Data Migration via Federated Data Preprocessing** . . . . . . . . . . . **8**

2.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.2 Solution Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.3 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.3.1 Execution Engines . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.3.2 Inefficiencies in Constrained Network Conditions . . . . . . . . . . . . . . 12
2.4 Sol: A Federated Execution Engine . . . . . . . . . . . . . . . . . . . . . . . . 15
2.5 Sol Control Plane . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.5.1 Early-Binding to Avoid High-Latency Coordination . . . . . . . . . . . . . 17
2.5.2 Why Does Early-Binding Help? . . . . . . . . . . . . . . . . . . . . . . . 19
2.5.3 How to Push the Right Number of Tasks? . . . . . . . . . . . . . . . . . . 20
2.5.4 How to Push Tasks with Dependencies? . . . . . . . . . . . . . . . . . . . 21
2.5.5 How to Handle Failures and Uncertainties? . . . . . . . . . . . . . . . . . 22
2.6 Sol Data Plane . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.6.1 How to Decouple the Provisioning of Resources? . . . . . . . . . . . . . . 23

2.6.2 How Many Communication Tasks to Create? . . . . . . . . . . . . . . . . 24
2.6.3 How to Recover CPUs for Computation? . . . . . . . . . . . . . . . . . 24
2.6.4 Who Gets the Freed up CPUs? . . . . . . . . . . . . . . . . . . . . . 25
2.7 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
2.8 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
2.8.1 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
2.8.2 Performance Across Diverse Workloads in EC2 . . . . . . . . . . . . . 27
2.8.3 Online Performance Breakdown . . . . . . . . . . . . . . . . . . . . 29
2.8.4 Sol's Performance Across the Design Space . . . . . . . . . . . . . . 31
2.8.5 Sol's Performance Under Uncertainties . . . . . . . . . . . . . . . . 32
2.9 Discussion and Future Work . . . . . . . . . . . . . . . . . . . . . . . 33
2.10 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
2.11 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

**3 Minimizing Training Execution via Automated Training Warmup . . . . . . . . . . 36**

3.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
3.2 Solution Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
3.3 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
3.3.1 DNN Model Training . . . . . . . . . . . . . . . . . . . . . . . . . 39
3.3.2 Opportunities for Repurposing Models . . . . . . . . . . . . . . . . 39
3.4 ModelKeeper Overview . . . . . . . . . . . . . . . . . . . . . . . . . . 42
3.5 ModelKeeper Design . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
3.5.1 Matcher: Identify Similar Models . . . . . . . . . . . . . . . . . . . 45
3.5.2 Mapper: Transform Maximal Parent Information . . . . . . . . . . . . 47
3.5.3 Zoo Manager: Transform Effectively At Scale . . . . . . . . . . . . . 50
3.6 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
3.7 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
3.7.1 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
3.7.2 End-to-End Performance . . . . . . . . . . . . . . . . . . . . . . . 56
3.7.3 Performance Breakdown . . . . . . . . . . . . . . . . . . . . . . . 59
3.7.4 Sensitivity and Ablation Studies . . . . . . . . . . . . . . . . . . . 61
3.8 Discussion and Future Work . . . . . . . . . . . . . . . . . . . . . . . 62
3.9 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
3.10 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64

**4 Minimizing Model Size via In-Training Model Pruning . . . . . . . . . . . . . . 65**

4.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
4.2 Solution Outlines . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
4.3 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
4.3.1 Deep Learning Recommendation Models . . . . . . . . . . . . . . . . 68
4.3.2 Challenges in DLRM Deployment . . . . . . . . . . . . . . . . . . . 69
4.3.3 Opportunities for In-Training Pruning . . . . . . . . . . . . . . . . . 70
4.4 AdaEmbed Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
4.5 AdaEmbed Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
4.5.1 Embedding Monitor: Identify Important Embeddings . . . . . . . . . . 74

4.5.2 AdaEmbed Coordinator: Prune at Right Time . . . . . 77
4.5.3 Memory Manager: Prune Weights at Scale . . . . . 80
4.6 Implementation . . . . . 83
4.7 Evaluation . . . . . 83
4.7.1 Methodology . . . . . 84
4.7.2 End-to-End Performance . . . . . 85
4.7.3 Performance Breakdown . . . . . 86
4.7.4 Sensitivity and Ablation Studies . . . . . 88
4.8 Related Work . . . . . 90
4.9 Summary . . . . . 91

**5 Minimizing Data Collection via Enabling Cross-Device Federated Learning . . . . 92**

5.1 Introduction . . . . . 92
5.2 Background . . . . . 95
5.3 FedScale Dataset: Realistic FL Workloads . . . . . 96
5.3.1 Client Statistical Dataset . . . . . 96
5.3.2 Client System Behavior Dataset . . . . . 98
5.4 FedScale Runtime: Execution Platform . . . . . 99
5.4.1 FedScale Runtime: Mobile Backend . . . . . 99
5.4.2 FedScale Runtime: Cluster Backend . . . . . 101
5.5 Experiments . . . . . 105
5.5.1 How Does FedScale Help FL Benchmarking? . . . . . 105
5.5.2 Opportunities for Future FL Optimizations . . . . . 107
5.6 Summary . . . . . 109

**6 Minimizing Cloud-Fed ML Accuracy Gap via Guided Participant Selection . . . . 110**

6.1 Background . . . . . 110
6.2 Solution Outlines . . . . . 111
6.3 Motivation . . . . . 112
6.4 Oort Overview . . . . . 114
6.4.1 Architecture . . . . . 114
6.4.2 Oort Interface . . . . . 115
6.5 Federated Model Training . . . . . 116
6.5.1 Tradeoff Between Statistical and System Efficiency . . . . . 116
6.5.2 Client Statistical Utility . . . . . 117
6.5.3 Trading off Statistical and System Efficiency . . . . . 118
6.5.4 Adaptive Participant Selection . . . . . 119
6.6 Federated Model Testing . . . . . 122
6.6.1 Preserving Data Representativeness . . . . . 123
6.6.2 Enforcing Diverse Data Distribution . . . . . 124
6.7 Implementation . . . . . 125
6.8 Evaluation . . . . . 125
6.8.1 Methodology . . . . . 126
6.8.2 FL Training Evaluation . . . . . 127
6.8.3 FL Testing Evaluation . . . . . 134

6.9 Related Work . . . . . 136
6.10 Summary . . . . . 137

**7 Conclusions . . . . . 138**

7.1 Lessons Learned . . . . . 138
7.2 Broader Impact . . . . . 140
7.3 Future Work . . . . . 140
7.3.1 Virtual Cloud for Pervasive Data Analytics . . . . . 140
7.3.2 Systems for Self-Adaptive ML . . . . . 141
7.3.3 ML for Self-Adaptive Systems . . . . . 142

**Appendices . . . . . 144**

A.1 Benefits of Pipelining . . . . . 144
A.2 Impact of Queue Length . . . . . 144
A.3 Determining the Queue Length . . . . . 145
B.1 ModelKeeper Analysis . . . . . 147
B.1.1 Design Criteria . . . . . 147
B.1.2 Information-Preserving Transformation . . . . . 148
C.1 Experiment Setup . . . . . 150
C.2 Introduction of FedScale Datasets . . . . . 150
C.3 Comparison with Existing FL Benchmarks . . . . . 152
C.4 Examples of New Plugins . . . . . 154
D.1 Proving Benefits of Statistical Utility . . . . . 156
D.2 Privacy Concern in Collecting Feedbacks . . . . . 157
D.3 Determining Size of Participants . . . . . 159

**Bibliography . . . . . 160**

# LIST OF FIGURES

FIGURE


1.1 Lifecycles of ML development. With the rapid growth of ML applications, ML development is facing scaling challenges in every stage. . . . . . . . . . . . . . . . . 2
1.2 This dissertation introduces minimalist systems designed to reduce ML resource demands and data collection. They span every stage of the ML lifecycle and enable efficient and pervasive ML in the cloud (§2-§4) and up to the planetary scale (§5-§6). . 5

2.1 Execution engine forms the narrow waist between diverse applications and resources. . 9
2.2 While execution engines are widely deployed on cloud platforms, the underlying network conditions can be diverse in latency and bandwidth. . . . . . . . . . . . . . . 11
2.3 TPC query completion times in different network settings using different execution engines (scale factor is 100). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.4 CPU utilization throughout a machine learning job. . . . . . . . . . . . . . . . . . . 13
2.5 Resource utilization over a bandwidth-bound query's lifespan (scale factor is 1000). . 14
2.6 Sol components and their interactions. Low-latency sites synchronously coordinate within themselves and asynchronously coordinate across high-latency links. . . . . . . 16
2.7 Task execution control flows in traditional designs vs. Sol. In Sol, tasks (denoted by colored rectangles) are queued at a site manager co-located with workers. . . . . . . . 18
2.8 Sol adopts the push-based model to pipeline task scheduling and data fetch. . . . . . . 18
2.9 Job performance with Site- and Worker-Queue approaches. Variance in task durations increases from (a) to (c). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.11 (b) The recovery process is shown in red. (0) $W_1$ detects large output, and sends CANCEL message to the coordinator $C$ for task rescheduling. (1) Upon receiving the update, $C$ waits until it gathers required information, then reschedules task, and (3) cancels task in $W_2$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
2.12 High-level overview of data plane decoupling. . . . . . . . . . . . . . . . . . . . . . 23
2.13 Three strategies to manage decoupled jobs. We adopt the Greedy strategy: ① when the downstream tasks start, they hold the reserved CPUs. ② But the task manager will reclaim the unused CPUs, and ③ activate the computation task once its input data is ready. The first trough marks the stage boundary. . . . . . . . . . . . . . . . . . . . 25
2.14 Performance of Sol, Spark, and Tez on TPC query processing benchmark. . . . . . . . 26
2.15 Performance on machine learning. . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
2.16 Performance on stream processing. Higher is better. . . . . . . . . . . . . . . . . . . 29
2.17 Resource utilization over time. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

2.18 JCT with online job arrival using different cluster schedulers. Sol- is Sol without data plane decoupling. . . . . . 31
2.19 Sol performance in other design space. . . . . . 31
2.20 Sol performance under latency variations. . . . . . 32
2.21 Impact of different failures on iteration duration for Sol and Spark: (a) Sol site manager failure, (b) task failure, (c) node failure , and (d) site-wide failure. . . . . . 33

3.1 A DNN model is essentially a graph of tensors. Model outputs are determined by tensor weights and their control flow. . . . . . 39
3.2 Pervasive model similarity in today's model zoos. We measure the top-1 and top-5 architectural similarities of each model to other models, and report the distribution across models. 1 indicates identical model architectures. . . . . . 40
3.3 Transferring model weights from well-trained models with similar architectures can accelerate new model training. . . . . . 41
3.4 Warm start provides better initial weights search space. We use RoR3 to warm start ResNet101. . . . . . 41
3.5 ModelKeeper architecture. It can run as a cluster-wide service to serve different users and/or frameworks. . . . . . 43
3.6 Code snippet of ModelKeeper client service APIs. . . . . . 44
3.7 ModelKeeper relies on dynamic programming-like heuristics to measure graph-level model architectural similarity. . . . . . 45
3.8 Keeper is order-of-magnitude more scalable than existing GED. V-Ensemble is a model zoo for ensemble training (§3.7.1). . . . . . 47
3.9 Models vary in accuracy and architecture (Imgclsmob zoo). We measure their similarity w.r.t.ResNet101, and prefer to transform a parent model with better similarity and accuracy. . . . . . 48
3.10 Width and depth operators to transform the parent model. . . . . . 50
3.11 Matcher clusters models into groups to reduce the search space, and then performs model matching within groups. . . . . . 51
3.12 ModelKeeper can find the optimal number of clusters $K^*$ in hierarchical matching, and can identify the most similar models with fewer zoo models (e.g., 5% in NASBench) needed to explore. . . . . . 51
3.13 A few zoo models are more frequently repurposed as the parent by Keeper. Numbers are from our evaluations (§3.7.2). . . . . . 53
3.14 ModelKeeper outperforms existing warmup training. . . . . . 57
3.15 ModelKeeper improves general training tasks. . . . . . 58
3.16 Breakdown of Keeper components. . . . . . 59
3.17 Faster training with higher model similarity. . . . . . 60
3.18 Keeper introduces negligible overhead. . . . . . 60
3.19 Keeper is robust in the presence of poor performance models (NAS-Grid). . . . . . 61
3.20 Keeper improves training execution time across the different numbers of buckets. . . . 61
3.21 Impact of zoo capacity on execution time. Error bars report standard deviation. . . . . 62
3.22 Keeper accelerates model training on CIFAR-100 using ImageNet32 model zoo. . . . 62

4.1 DLRM models consist of large embedding tables. . . . . . 68

4.2 The number of sparse feature instances (IDs) increases rapidly over time, while the lifespan of instances is heterogeneous. . . . . . 69
4.3 Compared to the full (1×) model, smaller embedding sizes hurt model NE (i.e., larger NE regression), but improve QPS. 0.25× and 0.5× denote using 25% and 50% of the full model size, respectively. . . . . . 70
4.4 Embedding access varies across IDs and over time, leading to distinct table utilization in existing embedding designs. . . . . . 71
4.5 AdaEmbed overview and its in-training execution flow. AdaEmbed components are in red. . . . . . 72
4.6 AdaEmb supports existing DLRMs with minor changes. . . . . . 73
4.7 (a) Embedding gradient and access frequency are heterogeneous, (b) while their combination reports a larger correlation to the embedding weights. A correlation value > 0.4 indicates a positive, medium to strong association. . . . . . 74
4.8 Magnitudes of embedding access frequencies and gradients vary across features, making it hard to compare $EI(i)$. . . . . . 76
4.9 (a) Each iteration accesses millions of embeddings. (b) Pruning needs to reallocate a large amount of embedding weights. . . . . . 77
4.10 Profiling can get accurate results with little overhead. . . . . . 79
4.11 VHPI employs lookup table to link each embedding to the weight vector, and recycles the vector of pruned embeddings without intense memory allocation. . . . . . 80
4.12 Zero initialization performs better (0.5× model). . . . . . 82
4.13 VHPI operations introduce little overhead. . . . . . 82
4.14 Example embedding configuration in AdaEmbed. . . . . . 84
4.15 AdaEmbed achieves better lifetime NE and evaluation NE. Better lifetime NE implies potentially better model accuracy for online learning deployment, while better evaluation NE indicates better accuracy after offline training (i.e., prior to launching online training). Both NEs are important metrics. . . . . . 86
4.16 Models with AdaEmbed achieve consistently better NE over time. Troughs are due to data distribution shifting over days. . . . . . 87
4.17 For the same NE w.r.t.1× model, AdaEmbed learns better per-feature embedding configuration using smaller size. . . . . . 87
4.18 For the same size (0.5× model), AdaEmbed retains more important embeddings to achieve better NE (Model-XS). . . . . . 88
4.19 Performance breakdown of AdaEmbed (AE) design. . . . . . 89
4.20 AdaEmbed achieves improvement across settings. . . . . . 89
4.21 AdaEmbed outperforms importance alternatives. . . . . . 90
4.22 AE outperforms post-training pruning (PTP). . . . . . 90

5.1 Standard FL protocol. . . . . . 93
5.2 Existing benchmarks can be misleading. We train ShuffleNet on OpenImage classification (Detailed setup in Section 5.5). . . . . . 95
5.3 Non-IID client data. . . . . . 97
5.4 Heterogeneous client system speed. . . . . . 98
5.5 Client availability is dynamic. . . . . . 98

5.7 FedScale Runtime can benchmark the mobile runtime of power, energy, and latency. We train Resnet34 and Shufflenet on ImageNet and CIFAR-10 on Xiaomi mi10 and Samsung S10e. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100
5.6 Training on mobile client. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100
5.8 FedScale Runtime enables the developer to benchmark various FL efforts with practical FL data and metrics. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
5.9 Add plugins by inheritance. . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
5.10 FedScale Runtime can run thousands of clients/round on 10 GPUs where others fail.[1]More results are in Appendix C.3. . . . . . . . . . . . . . . . . . . . . . . 104
5.11 FedScale can benchmark the statistical FL performance. (c) shows existing benchmarks can under-report the FedYoGi performance as they cannot support a large number of participants. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
5.12 FedScale can benchmark realistic FL runtime. (a) and (b) report FedYoGi results on OpenImage with different number of local steps (K); (b) reports the FL runtime to reach convergence. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
5.13 FedScale can benchmark privacy efforts in more realistic FL settings. . . . . . . . . . 108
5.14 FedScale can benchmark security optimizations with realistic FL data. . . . . . . . . . 108
5.15 System stragglers slow down practical FL greatly. . . . . . . . . . . . . . . . . . . . 108
5.16 Biased accuracy distributions of the trained model across clients (Shufflenet on OpenImage). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108

6.1 Existing works are suboptimal in: (a) round-to-accuracy performance and (b) final model accuracy. (a) reports number of rounds required to reach the highest accuracy of Prox on MobileNet (i.e., 74.9%). . . . . . . . . . . . . . . . . . . . . . . . . . 113
6.2 Participant selection today leads to (a) deviations from developer requirements, and thus (b) affects testing result. Shadow indicates the [min, max] range of y-axis values over 1000 runs given the same x-axis input; each line reports the median. . . . . . . . 114
6.3 Oort architecture. The driver of the FL framework interacts with Oort using a client library. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
6.4 Code snippet of Oort interaction during FL training. . . . . . . . . . . . . . . . . . . 116
6.5 Existing FL training randomly selects participants, whereas Oort navigates the sweet point of statistical and system efficiency to optimize their circled area (i.e., time to accuracy). Numbers are from the MobileNet on OpenImage dataset (§6.8.2.1). . . . . 117
6.6 Key Oort APIs for supporting federated testing. . . . . . . . . . . . . . . . . . . . . 122
6.7 Time-to-Accuracy performance. A lower perplexity is better in the language modeling (LM) task. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 128
6.8 Breakdown of Time-to-Accuracy performance with YoGi, when using different participant selection strategies. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129
6.9 Number of rounds to reach the target accuracy. . . . . . . . . . . . . . . . . . . . . 130
6.10 Breakdown of final model accuracy. . . . . . . . . . . . . . . . . . . . . . . . . . . 130
6.11 Oort outperforms in different scales of participants. . . . . . . . . . . . . . . . . . . 131
6.12 Oort improves performance across penalty factors. . . . . . . . . . . . . . . . . . . 131
6.13 Oort still improves performance under outliers. . . . . . . . . . . . . . . . . . . . . 132
6.14 Oort improves performance even under noise. . . . . . . . . . . . . . . . . . . . . . 133

6.15 Oort can cap data deviation for all targets. Shadow indicates the empirical [min, max] range of the x-axis values over 1000 runs given the y-axis input. . . . . . . . . . . . . . 134
6.16 Oort outperforms MILP in clairvoyant FL testing. . . . . . . . . . . . . . . . . . . . . . . . . 135
6.17 Oort scales to millions of clients, while MILP did not complete on any query. . . . . . 135

7.1 LLaMA model weights are less sparse, but runtime attention weights are highly sparse. 141

A.1 Improvement of Push-based Model in (§2.5.2) . . . . . . . . . . . . . . . . . . . . . . . . . 144
A.2 Impact of Queue Size on Job Completion Time (JCT). . . . . . . . . . . . . . . . . . . . . 145
A.3 JCT performance with different utilization targets. . . . . . . . . . . . . . . . . . . . . . . 145

C.1 Evaluate client selection algorithm [163]. . . . . . . . . . . . . . . . . . . . . . . . . . . . 154
C.2 Evaluate model compression [222]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 154
C.3 Evaluate security enhancement [232]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 154
C.4 Evaluate model compression with Flower [50]. The developer needs to implement the functions in grey by his own. Note that each function can take tens of lines of code. . . 155

D.1 Oort outperforms with different utility definitions. . . . . . . . . . . . . . . . . . . . . . 159

# LIST OF TABLES

TABLE


3.1 Summary of improvements. ModelKeeper improves training execution time without accuracy drop, by reducing the amount of training needed (i.e., GPU Saving). The accuracy difference is defined by Acc.(Keeper) - Acc.(Baseline), and smaller perplexity (ppl) is better. . . . . 55
3.2 Keeper saves training execution time of individual jobs without accuracy drop. Smaller perplexity (ppl) is better. . . . . 59

4.1 Summary of improvements. AdaEmbed reduces the embedding size needed for the same model accuracy (NE), while improving NE using the same embedding size. We report the approximate memory savings, since evaluating all memory settings is unaffordable. . . . . 83

5.1 Comparing FedScale with existing FL benchmarks and libraries. ◯ implies limited support. . . . . 94
5.2 Statistics of *partial* FedScale datasets (the full list with more details is available in Appendix C.2). Currently, FedScale has 20 real-world federated datasets; each dataset is partitioned by its real client-data mapping, and we have removed sensitive information in these datasets. . . . . 96
5.3 Some example APIs. FedScale provides APIs to deploy new plugins for various designs. We omit input arguments for brevity here. . . . . 101
5.4 Benchmarking of different FL algorithms across realistic FL datasets. We report the mean test accuracy over 5 runs. . . . . 105

6.1 Statistics of the dataset in evaluations. . . . . 126
6.2 Summary of improvements on time to accuracy.[2]We tease apart the overall improvement with statistical and system ones, and take the highest accuracy that Prox can achieve as the target, which is moderate due to the high task complexity and lightweight models. 126
6.3 Oort improves time to accuracy (TTA) across different fairness knobs ($f$). Random reports the performance of random participant selection. The variance of rounds reports how fairness is enforced in terms of the number of participating rounds across clients. A smaller variance implies better fairness. . . . . 134

C.1 Statistics of FedScale datasets. FedScale has 20 realistic client datasets, which are from the real-world collection, and we partitioned each dataset using its real client-data mapping. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 151
C.2 FedScale is more scalable and faster. Image classification on iNature dataset using MobileNet-V2 on 20-GPU setting. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 153

# LIST OF ALGORITHMS

ALGORITHM


2.1 The interaction between the central coordinator, site manager, and task manager. . . 17

3.1 Select the parent model to transform. . . . . . . . . . . . . . . . . . . . . . . . . . 49

4.1 Pseudo-code of AdaEmbed runtime . . . . . . . . . . . . . . . . . . . . . . . . . . 78

6.1 Participant selection w/ exploration-exploitation. . . . . . . . . . . . . . . . . . . . . 121

# LIST OF APPENDICES


**A Sol Appendix** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **144**

**B ModelKeeper Appendix** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **147**

**C FedScale Appendix** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **150**

**D Oort Appendix** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **156**

# ABSTRACT

Skyrocketing data volumes, growing hardware capabilities, and the revolution in machine learning (ML) theory have collectively driven the latest leap forward in ML. Despite our hope to realize the next leap with new hardware and a broader range of data, ML development is reaching scaling limits in both realms. First, the exponential surge in ML workload volumes and their complexity far outstrip hardware improvements, leading to hardware resource demands surpassing the sustainable growth of capacity. Second, the mounting volumes of edge data, increasing awareness of user privacy, and tightening government regulations render conventional ML practices, which centralize all data into the cloud, increasingly unsustainable due to escalating costs and scrutiny.

This dissertation surmounts these resource and data limits using a minimalist approach – reducing complexity and eliminating bloating features – to develop minimalist systems. The thesis provides evidence that by co-designing ML, systems, and networking, we can (1) minimize ML resource demands by removing bloating system execution without compromising ML performance; (2) minimize data collection by effectively offloading ML to the planet-scale data source; and (3) minimize human effort by automatically discovering the sweet spot of ML and system efficiency.

The minimalist systems developed in this thesis span each stage of ML development, facilitating the transition to the era of pervasive ML. The thesis commences with the data preprocessing stage and introduces a network-aware execution engine called Sol. Sol empowers distributed ML clusters to efficiently perform collaborative data processing over the Internet to minimize data migration. The second and third parts of this thesis optimize the subsequent training stage, by introducing ModelKeeper and AdaEmbed to minimize ML resource demands. ModelKeeper repurposes the weights of previously trained models to warm up model training, reducing the amount of training execution needed. During model training, AdaEmbed automatically identifies the model weights that contribute more to model accuracy and removes less important weights, reducing model size without compromising model accuracy.

The fourth and fifth parts introduce FedScale and Oort to complement and extend today's ML training stage and the subsequent deployment stage up to the planetary scale. They enable federated model training and testing across millions of clients at the edge. FedScale supports on-device model execution and integrates Oort to orchestrate clients. At runtime, Oort cherry-picks the clients, who have the data that offers better utility in improving model accuracy and the capability to execute the ML task quickly, to minimize the performance gap between cloud ML and federated ML.

# CHAPTER 1

# Introduction

The last ten years have witnessed a monumental surge in machine learning (ML) performance and adoption. This leap was made possible by the cloud, especially hardware improvements, big data, and the evolution of ML theories. These three driving forces empower ML developers to develop powerful ML models in the cloud and subsequently deploy a plethora of ML-based applications for end users at the edge.

As today's machine learning heavily relies on the cloud to explore various models and to collect real-life data from the edge to improve model quality, this has placed great strain on the system infrastructure. For instance, over the recent three years, the size of the largest models has grown more than 1000× [213], training jobs surged by over 7× [199], and the influx of edge data for cloud models has escalated by 2.4× [257].

In the face of these scalability demands, extensive efforts have been channeled into designing systems adept at efficiently scaling ML. To make the best use of hardware resources, a variety of solutions perform advanced scheduling to scale up the resource utilization of hardwares [67, 259, 212, 46]. As the model and data sizes continue their growth trajectory, recent systems have focused on scaling out ML execution across hundreds of machines [210, 144, 197, 168, 267]. By scaling up resources and data capacity, we are hoping to usher in the next leap in ML.

However, this scaling strategy is becoming untenable for at least two of the aforementioned drivers. First, we are grappling with stalling hardware improvements, yet experiencing an explosion in both job numbers and model sizes. The increase in total hardware resource demands is outpacing the sustainable growth in overall resource capacity [257]. Second, with the proliferation of end-user devices and applications, the conventional way to improve cloud ML – by collecting real-life data from the edge – is increasingly infeasible due to the exploding data volumes [54, 244], regulatory restrictions (e.g., GDPR [10], CCPA [5]), and mounting privacy concerns [116, 128, 243].

So, how can we unblock resource and data limits to herald the next leap forward in ML? This thesis contends that we can design *minimalist systems* to minimize the resource demands and data collection needs of ML models without compromising model accuracy. The crux of this philosophy – *reducing complexity and eliminating bloating features* – lies in the fact that not all ML components

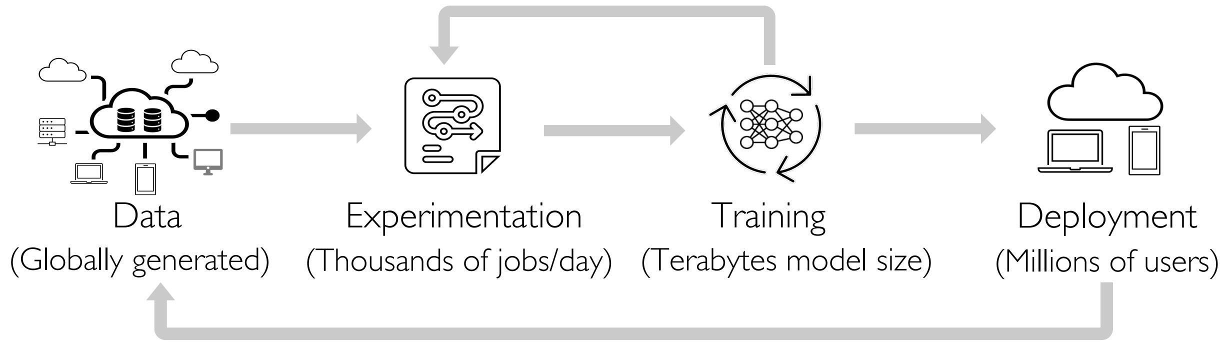



Figure 1.1: Lifecycles of ML development. With the rapid growth of ML applications, ML development is facing scaling challenges in every stage.

contribute equally to model accuracy. Thus, we can build *minimalist systems* to only capture pivotal components and jointly adapt ML and systems to expunge the bloating system execution.

In the remainder of this chapter, we provide background information on ML development, highlight the motivations for minimalist systems, and summarize our contributions.

## 1.1 Background: Scaling Trends in Machine Learning

Modern ML applications are now often referred to as "Software 2.0" to highlight the radical shift they represent compared to conventional computing applications [215]. Such applications are developed using new end-to-end workflows, introduced to a broader range of users and settings such as AI for science, and accommodate multifaceted considerations such as privacy, fairness, and robustness. This section introduces the lifecycle and trends in ML development.

### 1.1.1 Lifecycle of Machine Learning Development

As shown in Figure 1.1, in contrast to conventional big data analytics, ML development often involves an iterative lifecycle to pivot on the latest data trends (e.g., user preferences). It primarily comprises four stages: Data Processing, Experimentation, Model Training, and Deployment, each can involve computation on the globally generated data across many machines.

❶ **Data Processing**: ML development aims to train and deploy models with high quality (e.g., model accuracy). The performance of an ML model hinges on both the volume and quality of the data. As such, ML developers often initiate by collecting and processing data. This involves tasks such as cleaning and transforming the raw data into suitable formats, performing big data analytics for feature engineering (e.g., analyzing data distribution), and splitting the data for training, validation, and testing.

❷ **Experimentation**: Post data preparation, this stage delves into various configurations, such as model architectures, hyperparameters, and input features, to optimize model performance. It is an iterative process that compares results from various models, based on pre-selected evaluation metrics.

❸ **Model Training**: Concurrently with experimentation, a variety of models are trained using the designated hyperparameters and techniques. Once training completes, ML developers evaluate the model performance on the validation set, which informs further iterations in the experimentation stage.

❹ **Model Deployment**: This stage delivers the best performing model into service, which can be in the cloud (e.g., gigantic recommendation models [194]) and/or at the edge (e.g., word prediction on keyboard [116]). The decision of deployment depends on factors such as latency requirements, computational resources, and request volumes. The model performs inference tasks for the application request, and the data generated during service can refine the model.

The lifecycle does not end with deployment. Persistent monitoring and iterative model training with fresh data are vital to adapt to evolving data patterns and ensure sustained model accuracy. Thus, ML resource costs and data volumes inflate over time.

### 1.1.2 Machine Learning Development is Hitting Limits

With the proliferation of ML-based applications and the increasing demands for performance, ML is experiencing a surge in data volumes, model sizes, and system resources [255, 139, 199]. Although recent systems have made considerable progress in scaling ML, ML development is confronting resource and data limits:

**Resource Limits** The growth in the volumes of ML workloads and the complexity thereof far outstrip hardware improvements, making it increasingly infeasible to scale out and democratize ML advances. For instance, ML model sizes from 2019 to 2021 increased by 20×, whereas GPU memory capacity growth lags, increasing by less than 2 × every 2 years [257], e.g., 32 GB (NVIDIA V100, 2018) to 80 GB (NVIDIA A100, 2021); large language models (LLMs) such as Llama 2 require millions of GPU hours to train [235], implying prohibitively high resource costs even for model inference, due to the skyrocketing number of user requests. Indeed, the growth in ML resource demands is outpacing the affordable increase in total resource capacity (4× vs. 2.9× in the past 18 months), even with the addition of more machines [199].

**Data Limits** Although the cloud has successfully accommodated the three "V"s (volume, velocity, and variety) of Big Data, the conventional ML development that centralizes edge data to the cloud becomes increasingly untenable. As applications shift to the edge for end-user service – Gartner predicts that 75% of enterprise data will be created at the edge by 2025 – centralizing edge data not only incurs high costs to keep up with the velocity of data generation, but also suffers from various restrictions on the variety of data it can collect. For example, Meta is expected to lose $10 billion in advertising revenue due to Apple's recent privacy restrictions on collecting iOS users' tracking data to the platform; application-integrated LLMs have caused new privacy threats [166]. The cost and scrutiny of how data is collected and used will only increase with growing awareness of user privacy and stronger government regulations. Making sense of data closer to its home is more appealing than ever.

## 1.2 Thesis Statement and Contributions

Historically, ML system design trends have revolved around optimizing hardware resource utilization and subsequently executing ML across hundreds of machines [259, 276, 108, 210, 197, 292]. However, in the face of such resource and data limits in scaling, even with the optimal solution for both aspects, ML development is still capped by the sluggish growth of the resource capacity, due to factors such as stalling hardware improvements and/or cost concerns, as well as the data available in the cloud. This calls for revisiting the design philosophy to unblock the next leap forward in ML performance and adoption.

**Thesis Statement –** *By co-designing machine learning, systems, and networking, we can remove the bloating system execution in machine learning to minimize the amount of system execution needed to get the same model accuracy, and can enable efficient federated machine learning up to the planetary scale to minimize data collection.*

In lieu of solely focusing on scaling, this dissertation adopts a minimalist approach to unblock the resource and data limits, by *building systems to enable more efficient, sustainable ML with reduced resource demands and data collection*. Our systems can not only improve the efficiency of the same ML algorithm by orders of magnitude, but also enable new advances in theory and applications up to the planetary scale.

Our minimalist systems optimize every stage of the ML lifecycle, promoting pervasive ML in the cloud and beyond (Figure 1.2):

❶ **Data Processing:** As many of today's organizations host their services globally and (therefore) generate massive amounts of data of interest worldwide, centralized data processing

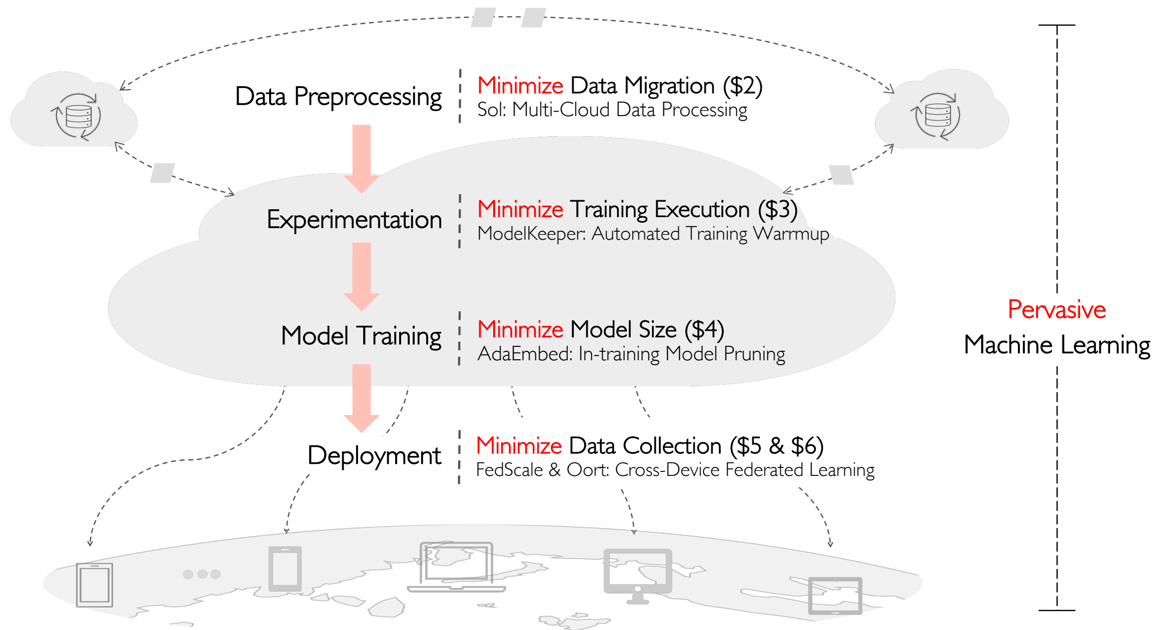



Figure 1.2: This dissertation introduces minimalist systems designed to reduce ML resource demands and data collection. They span every stage of the ML lifecycle and enable efficient and pervasive ML in the cloud (§2-§4) and up to the planetary scale (§5-§6).

becomes prohibitively expensive. In reality, we could *minimize data migration* by enabling distributed ML clusters to perform in-situ data processing and subsequently aggregate their intermediate results. However, modern execution engines (e.g., Spark [278] or PyTorch [24]) implicitly rely on ultra-fast datacenter networks, leading to great CPU/GPU underutilization when their computation is bottlenecked by slow Internet.

Sol is a network-aware execution engine capable of executing distributed data processing over the Internet. It pipelines task execution and expensive coordination over high-latency networks, while dynamically scaling down a task's CPU/GPU requirements to match the available bandwidth, thereby boosting execution throughput. Sol allows ML data processing, model training, and inference jobs to run with an order-of-magnitude reduction in data migration costs.

❷ **Experimentation:** After getting the processed dataset, developers often prototype and/or (re)train various models to identify the best one. Since an ML model is essentially a graph of tensors with associated weight values, this raises a natural question: can we reuse such data to reduce the total amount of training execution?

We show that initializing a training job's model by transforming a previously-trained model's weights can jump-start training. We present ModelKeeper, the *first* system to automatically

warm up the training of new jobs at the cluster level. Given a new model to train, ModelKeeper scalably selects a parent model with high similarity in model architecture and good model accuracy. Then it warms up the new model's weights by transforming the parent model's weights, thereby reducing the amount of training execution needed for model convergence.

❸ **Model Training:** Other than the large number of training jobs, the ever-growing model size also leads to great resource demands. Notable examples include deep learning recommendation models (DLRMs) and LLMs, which can span tens of terabytes and require hundreds of GPUs to run. Understandably, not all components of the model (e.g., tensor weights) contribute equally to model accuracy at all times.

We introduce AdaEmbed, the first *in-training* pruning system for embedding-based models like DLRMs and LLMs, to further *minimize ML resource demands*. It considers the tensor weights that are frequently accessed and accumulate large gradient norms to be more important, and efficiently orchestrates hundreds of GPUs to remove less important weights during training. It not only offers large resource savings and better model accuracy but also reduces human effort by automatically searching for better embedding configurations.

❹ **Model Deployment and Data Collection:** Minimizing ML resource demands enables developing personalized models for various settings (e.g., incorporating personal features). But still, ML performance is capped by the data that can be collected to the cloud, especially when we desire more real-life or even sensitive data (e.g., user location data in DLRMs).

To *minimize data collection*, we developed FedScale, a scalable federated learning (FL) platform that enables distributed edge devices, such as smartphones and laptops, to collaboratively learn models without disclosing their raw data. FedScale spins up on-device execution and communication over the Internet, orchestrating pervasive ML up to the global scale. To bridge the performance gap between FL and cloud ML, we developed Oort, a client manager to coordinate model training across millions of heterogeneous clients. At scale, Oort explores and exploits those clients, who have both the data that offers the greatest utility in improving model accuracy and the capability to execute training quickly, and prioritizes their use in model training, achieving close-to-cloud ML performance.

**Real-world Adoptions** Our minimalist systems have received many adoptions in industry and the open-source community, attracting hundreds of worldwide researchers and contributors in many fields [8]. In terms of large-scale deployments, AdaEmbed is currently deployed at Meta, supporting multiple production models and catering to millions of users. FedScale and Oort are deployed at LinkedIn to support efficient federated learning. Recently, Oort is adopted by Cisco.

## 1.3 Dissertation Plan

This dissertation is organized in terms of the lifecycle of ML development. Chapter 2 presents Sol for the data processing stage, a network-aware execution engine for fast distributed computation over the Internet, so that distributed ML clusters can collaboratively perform data processing and model training across the globe to minimize the cost of data migration. In Chapter 3, we introduce ModelKeeper for the experimentation stage, a cluster-wide model manager designed to accelerate model training by automatically repurposing pre-trained model weights before training starts, thereby minimizing the amount of training execution to model convergence. Chapter 4 moves on to the training stage and discusses AdaEmbed, an in-training pruning system to reduce the size of DLRM models during training. Chapter 5 delves into FedScale to augment today's training and deployment stage up to the planetary scale. FedScale is a platform enabling practical federated ML across end-user devices, and its client manager, Oort, is discussed in Chapter 6 for efficient participant selection. We conclude the dissertation and discuss future directions in Chapter 7.

# CHAPTER 2

## Minimizing Data Migration via Federated Data Preprocessing

This chapter focuses on optimizing the preliminary stage of ML development, i.e., data preprocessing, and introduces a *federated execution engine* named Sol. Sol enables data preprocessing, ML training, and inference on data sourced globally throughout ML lifecycle. It is designed to bridge the ever-widening gap between data generation and ML development – data of interest is being generated over the globe, while the execution engines available today (e.g., Spark) are ones primarily designed for ultra-fast datacenter networks. Sol aspires to expand ML execution capabilities globally.

The next section introduces the need for a network-aware execution engine for efficient ML execution across diverse network conditions. Section 2.2 provides an outline of our system. Section 2.5 and Section 2.6, respectively, present the design of the control plane and data plane to tackle various network latency and bandwidth conditions. We then implement and evaluate Sol in Section 2.7 and Section 2.8 through global-scale deployments. We then discuss future work in Section 2.9 and survey related work in Section 2.10.

### 2.1 Background

Execution engines form the narrow waist of modern ML software stacks (Figure 2.1). Given a user-level intent and corresponding input for an ML pipeline – be it running a SQL query to understand data characteristics [42], training ML models across many GPUs [36], or realtime stream processing to monitor model performance [60] – an execution engine orchestrates the execution of tasks across many distributed workers until the job runs to completion even in the presence of failures and stragglers.

Modern execution engines have primarily targeted datacenters with low latency and high bandwidth networks. The absence of noticeable network latency has popularized the *late-binding* task execution model in the control plane [207, 181, 241, 52] – pick the worker which will run a task only when the worker is ready to execute the task – which maximizes flexibility. At the same time, the impact of the network on task execution time is decreasing with increasing network

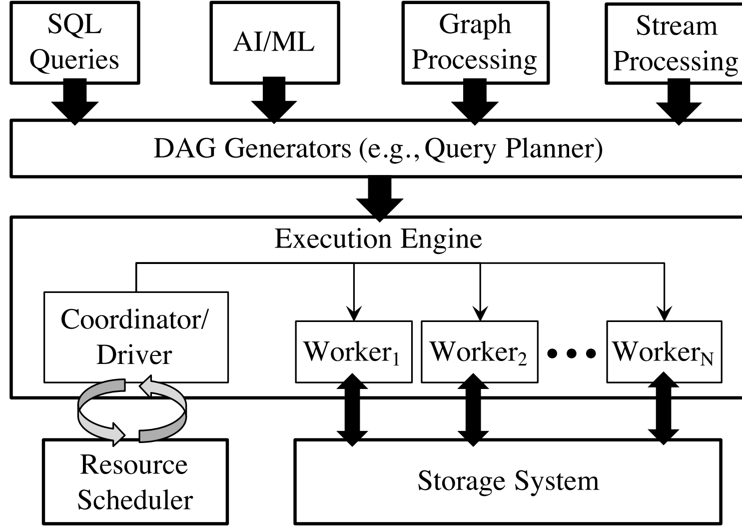



Figure 2.1: Execution engine forms the narrow waist between diverse applications and resources.

bandwidth; most datacenter-scale applications today are compute- or memory-bound [36, 206]. The availability of high bandwidth has led to *tight coupling* of a task's roles to hide design complexity in the data plane, whereby the same task reads remote input and computes on it too. Late-binding before execution and tight coupling during execution work well together when the network is well-provisioned.

Many emerging workloads, however, have to run on networks with high latency, low bandwidth, or both. Large organizations often perform interactive SQL and iterative machine learning between on- and off-premise storage [133, 128, 31, 74]. For example, Google uses federated model training on globally distributed data subject to privacy regulations [55, 293]; telecommunications companies perform performance analysis of radio-access networks (RAN) [137, 136]; while others troubleshoot their appliances deployed in remote client sites using ML [195, 264]. Although these workloads are similar to those running within a datacenter, the underlying network can be significantly constrained in bandwidth and/or latency (§2.3). In this chapter, *we investigate the impact of low bandwidth and high latency on latency-sensitive interactive and iterative workloads*.

While recent works have proposed solutions for bandwidth-sensitive workloads, the impact of network constraints on latency-sensitive workloads has largely been overlooked. Even for bandwidth-sensitive workloads, despite many resource schedulers [134, 280], query planners [243, 211], or application-level algorithms [128, 284], the underlying execution engines of existing solutions are still primarily the ones designed for datacenters. For example, Iridium [211], Tetrium [134], and Pixida [155] rely on the execution engine of Apache Spark [278], while many others (e.g., Clarinet [243], Geode [244]) are built atop the execution engine of Apache Tez [223].

Unfortunately, under-provisioned networks can lead to large CPU underutilization in today's

execution engines. First, in a high-latency network, late-binding suffers significant coordination overhead, because workers will be blocked on receiving updates from the coordinator; this leads to wasted CPU cycles and inflated completion times of latency-sensitive tasks (e.g., ML inference). Indeed, late-binding of tasks to workers over the WAN can slow down the job by 8.5×–30× than running it within the local-area network (LAN). Moreover, for bandwidth-intensive tasks (e.g., ML training), coupling the provisioning of communication and computation resources at the beginning of a task's execution leads to head-of-line (HOL) blocking: bandwidth-sensitive jobs hog CPUs even though they bottleneck on data transfers, which leads to noticeable queuing delays for the rest.

## 2.2 Solution Outline

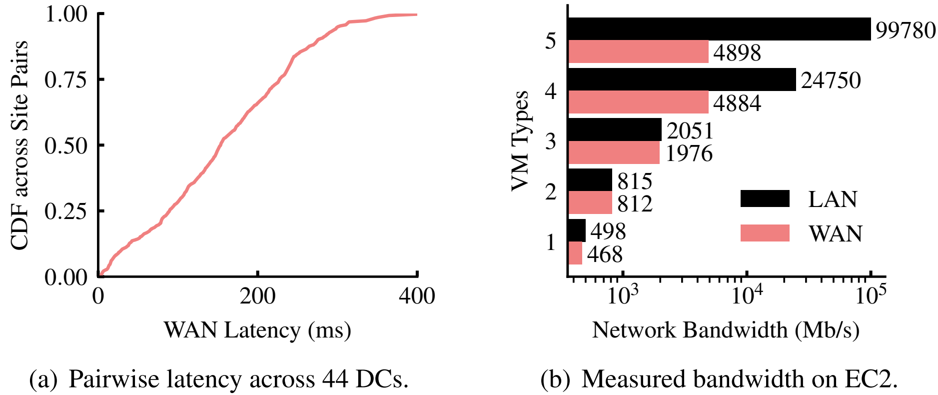
By accounting for network conditions, we present a federated execution engine, Sol, which is API-compatible with Apache Spark [278]. [1] Our design of Sol, which can transparently run existing jobs and WAN-aware optimizations in other layers of the stack, is based on two high-level insights to achieve better job performance and resource utilization.

First, we advocate *early-binding control plane decisions over the WAN* to save expensive round-trip coordinations, while continuing to late-bind workers to tasks within the LAN for the flexibility of decision making. By promoting early-binding in the control plane, we can pipeline different execution phases of the task. In task scheduling, we subscribe tasks for remote workers in advance, which creates a tradeoff: binding tasks to a remote site too early may lead to sub-optimal placement due to insufficient knowledge, but deferring new task assignments until prior tasks complete leaves workers waiting for work to do, thus underutilizing them. Our solution deliberately balances efficiency and flexibility in scheduling latency-bound tasks, while retaining high-quality scheduling for latency-insensitive tasks even under uncertainties.

Second, *decoupling the provisioning of resources for communication and computation within data plane task executions* is crucial to achieving high utilization. By introducing dedicated communication tasks for data reads, Sol decouples computation from communication and can dynamically scale down a task's CPU requirement to match its available bandwidth for bandwidth-intensive communications; the remaining CPUs can be redistributed to other jobs with pending computation.

Our evaluations show that Sol can automatically adapt to diverse network conditions while largely improving application-level job performance and cluster-level resource utilization. Using representative industry benchmarks on a 40-machine EC2 cluster across 10 regions, we show that Sol speeds up SQL and machine learning jobs by 4.9× and 16.4× on average in offline and online settings, respectively, compared to Apache Spark in resource-constrained networks. Even in

[1] Sol is available at https://github.com/SymbioticLab/Sol.




(a) Pairwise latency across 44 DCs.

(b) Measured bandwidth on EC2.

Figure 2.2: While execution engines are widely deployed on cloud platforms, the underlying network conditions can be diverse in latency and bandwidth.

datacenter environments, Sol outperforms Spark by 1.3× to 3.9×. Sol offers these benefits while effectively handling uncertainties and gracefully recovering from failures.

## 2.3 Motivation

### 2.3.1 Execution Engines

The execution engine takes a graph of *tasks* – often a directed acyclic graph (DAG) – from the higher-level scheduler as its primary input. Tasks performing the same computation function on different data are often organized into *stages*, with dependencies between the stages represented by the edges of the execution DAG. Typically, a central *coordinator* in the execution engine – often referred to as the driver program of a job – interacts with the cluster resource manager to receive required resources and spawns *workers* across one or more machines to execute runnable tasks.[2] As workers complete tasks, they notify the coordinator to receive new runnable tasks to execute.

**Design space.** The design of an execution engine should be guided by how the environment and workload characteristics affect delays in the *control plane* (i.e., coordinations between the coordinator and workers as well as amongst the workers) and in the *data plane* (i.e., processing of data by workers). Specifically, a task's lifespan consists of four key components:

- *Coordination time* ($t_{coord}$) represents the time for orchestrating task executions across workers. This is affected by two factors: network latency, which can vary widely between different pairs of sites (Figure 2.2(a)), and the inherent computation overhead in making decisions. While the latter can be reduced by techniques like reusing schedules [242, 184], the former is determined by the environment.

[2] A task becomes runnable whenever its dependencies have been met.

- *Communication time* ($t_{comm}$) represents the time spent on reading input and writing output of a task over the network and to the local storage.[3] For the same amount of data, time spent in communication can also vary widely based on Virtual Machine (VM) instance types and LAN-vs-WAN (Figure 2.2(b)).
- *Computation time* ($t_{comp}$) represents the time spent in running every task's computation.
- *Queuing time* ($t_{queue}$) represents the time spent waiting for resource availability before execution. Given a fixed amount of resources, tasks of one job may have to wait for tasks of other jobs to complete.

We first take into account $t_{comp}$, $t_{coord}$, and $t_{comm}$ in characterizing the design of execution engines for a single task. By assuming $t_{comp} \gg t_{coord}, t_{comm}$ (i.e., by focusing on the execution of *compute-bound* workloads such as HPC [107], AI training [36] and in many cases within datacenters), existing execution engines have largely ignored two settings in the design space.

First, the performance of jobs can be dominated by the coordination time (i.e., $t_{coord} \gg t_{comm}, t_{comp}$), and more time is spent in the control plane than the data plane. An example of such a scenario within a datacenter would be stream processing using mini-batches, where scheduling overhead in the coordinator is the bottleneck [242]. As $t_{coord} \to O(100)$ ms over the WAN, coordination starts to play a bigger role even when scheduler throughput is not an issue. As $\frac{t_{comp}}{t_{coord}}$ and $\frac{t_{comm}}{t_{coord}}$ decrease, e.g., in interactive analytics [137, 136] and federated learning [55], coordination time starts to dominate the end-to-end completion time of each task.

Second, in *bandwidth-bound* workloads, more time is likely to be spent in communication than computation (i.e., $t_{comm} > t_{coord}, t_{comp}$). Examples of such a scenario include big data jobs in resource-constrained private clusters [181] or across globally distributed clouds [211, 244, 243, 128], and data/video analytics in a smart city [127].

In the presence of multiple jobs, inefficiency in one job's execution engine can lead to inflated $t_{queue}$ for other jobs' tasks. For latency-sensitive jobs waiting behind bandwidth-sensitive ones, $t_{queue}$ can quickly become non-negligible.

### 2.3.2 Inefficiencies in Constrained Network Conditions

While there is a large body of work reasoning about the performance of existing engines in high-bandwidth, low-latency datacenters [206, 205], the rest of the design space remains unexplored. We show that existing execution engines suffer significant resource underutilization and performance loss in other settings.

[3] Most transfers after the input-reading stages happen over the network.

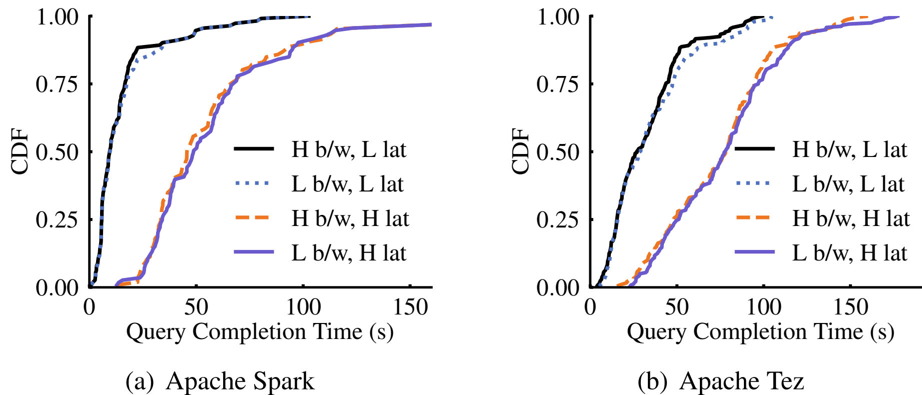



Figure 2.3: TPC query completion times in different network settings using different execution engines (scale factor is 100).

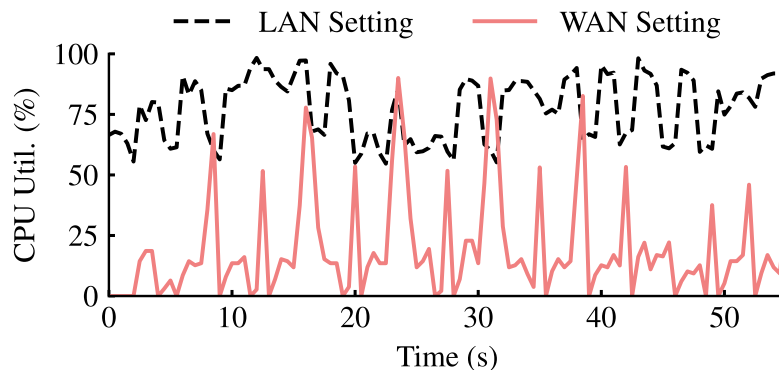



Figure 2.4: CPU utilization throughout a machine learning job.

**Performance degradation due to high latency.** To quantify the impact of high latency on job performance, we analyzed the individual query completion times of 110 queries on two industrial benchmarks: TPC-DS and TPC-H. We use two popular execution engines – Apache Spark[278] and Apache Tez[223] – on a 10-site deployment; each site has 4 machines, each with 16 CPU cores and 64 GB of memory. We consider four network settings, each differing from the rest in terms of either bandwidth or latency as follows: [4]

- *Bandwidth:* Each VM has a 10 Gbps NIC in the high-bandwidth and 1 Gbps in the low-bandwidth setting.
- *Latency:* Latency across machines is <1 ms in the low-latency setting, while latencies across sites vary from $O(10)$–400 ms in the high-latency setting.

Figure 2.3 shows the distributions of average query completion times of Spark and Tez, where

[4]We use the latency profile of 10 sites on EC2 and set a large TCP window size to reach the network capacity [158]. For the high-bandwidth setting and the low-bandwidth one, we refer to the available LAN bandwidth on *m4.10xlarge* and *m4.2xlarge* instances, respectively.

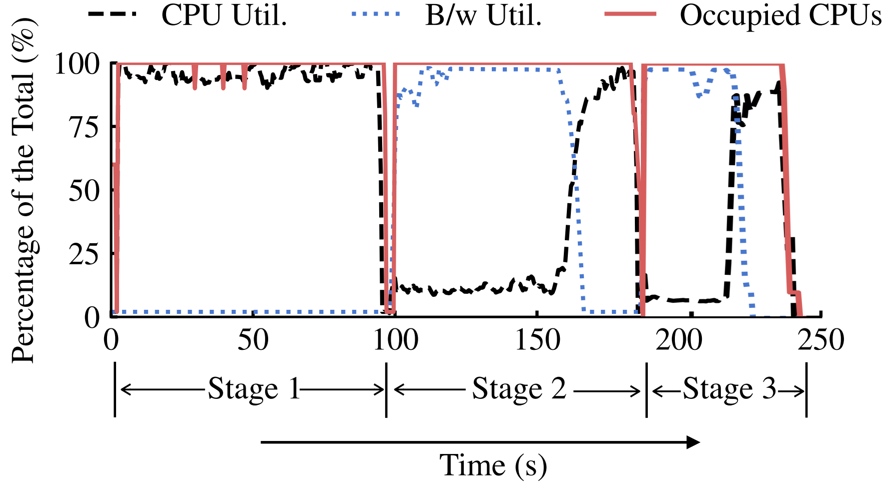



Figure 2.5: Resource utilization over a bandwidth-bound query's lifespan (scale factor is 1000).

we use a dataset of scale factor 100.[5] We found that the availability of more bandwidth has little impact on query completion times; different query plans and task placement decisions in Spark and Tez did not improve the situation either. However, job completion times in the high-latency setting are significantly inflated – up to $20.6\times$ – than those in the low-latency setting. Moreover, we observe that high network latency can lead to inefficient use of CPUs for a latency-bound machine learning job (Figure 2.4).

The root cause behind this phenomenon is the *late-binding* of tasks to workers in the control plane. In existing execution engines, decision making in the control plane, such as task scheduling [243] and straggler mitigation [41], often requires realtime information from data plane executions, whereas data processing is initiated by control plane decisions. With high coordination latency, this leads to wasted CPU cycles as each blocks on acquiring updates from the other.

**CPU underutilization due to low bandwidth.** To understand the impact of low bandwidth on resource efficiency, we analyzed bandwidth-sensitive workloads using a scale factor of 1000 in the same experimental settings as above.

Figure 2.5 reports both the CPU and network utilizations throughout the execution of a representative query (query-25 from the TPC-DS benchmark), which involves two large shuffles (in stage 2 and stage 3) over the network. During task executions, large data reads over the network are communication-intensive, while computations on the fetched data are CPU-intensive. We observe that, when tasks are bandwidth-constrained, their overall CPU utilization plummets even though they continue to take up all the available CPUs. This is due to the coupling of communication with computation in tasks. In other words, the number of CPUs involved in communication is

[5] A scale factor of $X$ means a $X$ GB dataset.

independent of the available bandwidth. The end result is head-of-line (HOL) blocking of both latency- and bandwidth-sensitive jobs (not shown) by bandwidth-bound underutilized CPUs of large jobs.

**Shortcomings of existing works.** Existing works on WAN-aware query planning and task placement [134, 244, 211, 243] cannot address the aforementioned issues because they focus on managing and/or minimizing bandwidth usage during task execution, not on the impact of latency before execution starts or CPU usage during task execution.

## 2.4 Sol: A Federated Execution Engine

To address the aforementioned limitations, we present Sol, a *federated execution engine* which is aware of the underlying network's characteristics (Figure 2.6). It is primarily designed to facilitate efficient execution of emerging distributed workloads across a set of machines which span multiple sites (thus, have high latency between them) and/or are interconnected by a low bandwidth network. Sol assumes that machines within the same site are connected over a low-latency network. As such, it can perform comparably to existing execution engines when deployed within a datacenter.

**Design goals.** In designing Sol, we target a solution with the following properties:

- *High-latency coordinations should be pipelined.* Coordinations between control and data planes should not stall task executions. As such, Sol should avoid synchronous coordination (e.g., workers blocking to receive tasks) to reduce overall $t_{coord}$ for latency-bound tasks. This leads to early-binding of tasks over high-latency networks.
- *Underutilized resources should be released.* Sol should release unused resources to the scheduler, which can be repurposed to optimize $t_{queue}$ for pending tasks. This calls for decoupling communication from computation in the execution of bandwidth-intensive tasks without inflating their $t_{comm}$ and $t_{comp}$.
- *Sol should adapt to diverse environments automatically.* The network conditions for distributed computation can vary at different points of the design space. Sol should, therefore, adapt to different deployment scenarios with built-in runtime controls to avoid reinventing the design.

**System components.** At its core, Sol has three primary components:

- *Central Coordinator:* Sol consists of a logically centralized coordinator that orchestrates the input job's execution across many remote compute sites. It can be located at any of the sites; each application has its own coordinator or driver program. Similar to existing coordinators, it interacts with a resource manager for resource allocations.

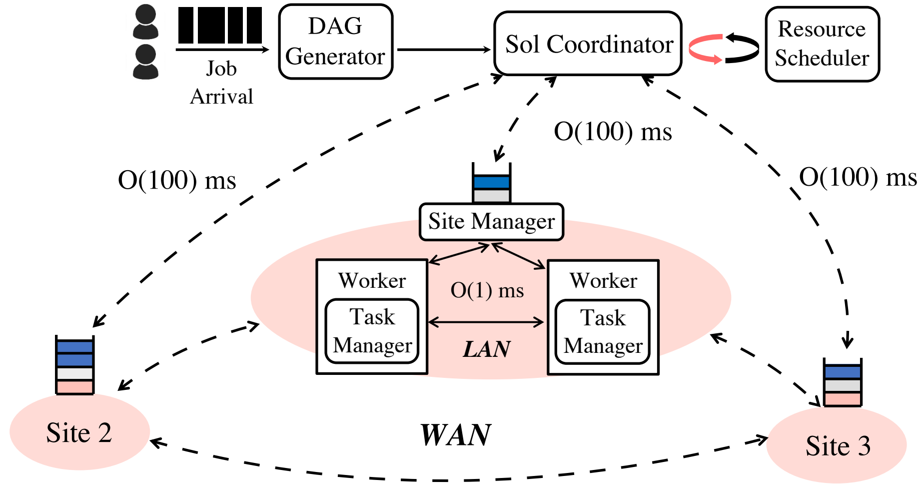



Figure 2.6: Sol components and their interactions. Low-latency sites synchronously coordinate within themselves and asynchronously coordinate across high-latency links.

- *Site Manager:* Site managers in Sol coordinate local workers within the same site. Each site manager has a shared queue, where it enqueues tasks assigned by the central coordinator to the workers in this site. This allows for late-binding of tasks to workers within the site and ensures high resource utilization, wherein decision making can inherit existing designs for intra-datacenter systems. The site manager also detects and tackles failures and stragglers that are contained within the site.
- *Task Manager:* At a high level, the task manager is the same as today: it resides at individual workers and manages tasks. However, it manages compute and communication resources independently.

Algorithm 2.1 shows a high-level overview explaining the interaction among these components throughout our design in the control plane (§2.5) and the data plane (§2.6).

## 2.5 Sol Control Plane

Modern execution engines primarily target datacenters with low latency networks [278, 223, 60, 36], wherein late-binding of tasks to workers maximizes flexibility. For example, the coordinator assigns new tasks to a worker after it is notified of new resource availability (e.g., due to task completion) from that worker. Moreover, a variety of on-demand communication primitives, such as variable broadcasts and data shuffles, are also initiated lazily by the coordinator and workers. In the presence of high latency, however, late-binding results in expensive coordination overhead (§2.3.2).

In this section, we describe how Sol pushes tasks to sites to hide the expensive coordination

```
/* Operations in Central Coordinator                                    */
 1: for Site s in all sites do
 2:   while currentTaskNum(s) < targetQueLen(s) do
 3:     if Exist available tasks t for scheduling to s then
 4:       Push t to Site Manager in s;                              ▷ §2.5.3
 5:     else
 6:       Break down task dependency judiciously;                   ▷ §2.5.4
/* Operations in Site Manager                                           */
 7: if Receive task assignment then
 8:   Queue up task;
 9: else if Receive task completion then
10:   Notify coordinator and schedule of next task t;
11:   if Task t requires large remote read then
12:     Issue fetch request to the scheduled worker;
13:   else
14:     Launch task t;
15: else if Input is ready for computation task t then
16:   Activate and launch task t;
/* Operations in Task Manager                                           */
17: if Receive task assignment t then
18:   Execute task t;
19: else if Detect task completion then
20:   Notify Site Manager of the new task assignment;
21: else if Receive data fetch request then
22:   Initiate communication task;                                  ▷ §2.6.2
```

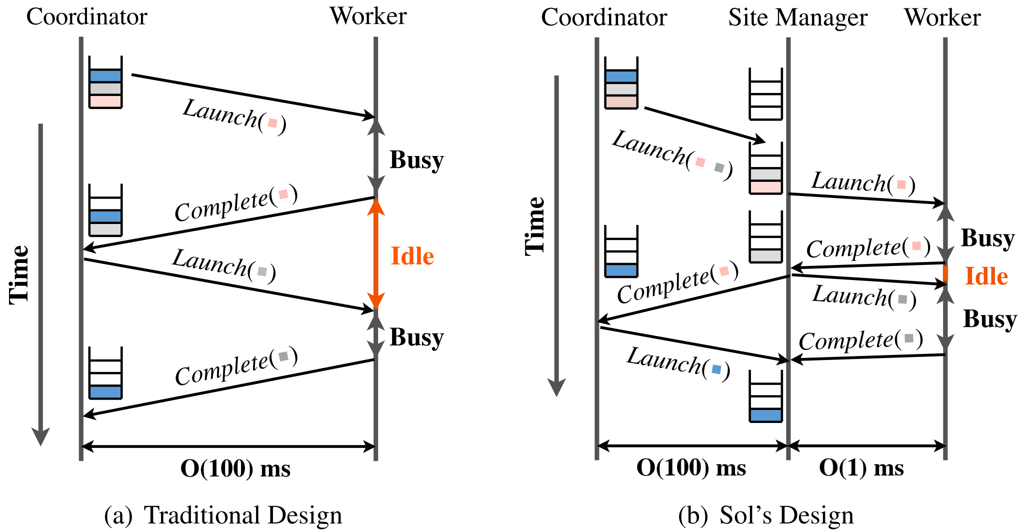
**Algorithm 2.1:** The interaction between the central coordinator, site manager, and task manager.

latency (§2.5.1), the potential benefits of push-based execution (§2.5.2) as well as how we address the challenges in making it practical; i.e., how to determine the right number of tasks to push to each site (§2.5.3), how to handle dependencies between tasks (§2.5.4), and how to perform well under failures and uncertainties (§2.5.5).

### 2.5.1 Early-Binding to Avoid High-Latency Coordination

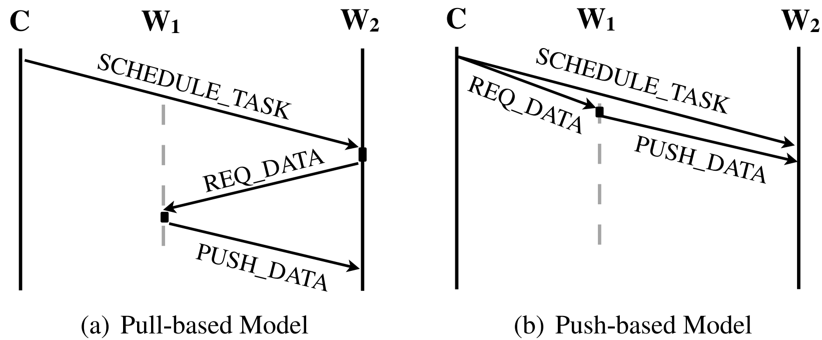
Our core idea to hide coordination latency ($t_{coord}$) over high-latency links is early-binding tasks to sites. Specifically, Sol optimizes $t_{coord}$ between the central coordinator and remote workers (i) by *pushing* and queuing up tasks in each site; and (ii) by *pipelining* task scheduling and execution across tasks.

Figure 2.7 compares the control flow in traditional designs with that in Sol. In case of late-




(a) Traditional Design (b) Sol's Design

Figure 2.7: Task execution control flows in traditional designs vs. Sol. In Sol, tasks (denoted by colored rectangles) are queued at a site manager co-located with workers.




(a) Pull-based Model (b) Push-based Model

Figure 2.8: Sol adopts the push-based model to pipeline task scheduling and data fetch.

binding, workers have to wait for new task assignments from the remote coordinator. In contrast, the site manager in Sol directly dispatches a task already queued up in its local queue and asynchronously notifies the coordinator of task completions as well as the resource status of the site. The coordinator makes new task placement decisions and queues up tasks in site managers asynchronously, while workers in that site are occupied with task executions.

Furthermore, this execution model enables us to pipeline $t_{coord}$ and $t_{comm}$ for each individual task's execution. When the coordinator assigns a task to a site, it notifies the corresponding upstream tasks (i.e., tasks in the previous stage) of this assignment. As such, when upstream tasks complete, they can proactively push their latency-bound output partitions directly to the site where their downstream tasks will execute, even though the control messages containing task placements may still be on-the-fly. As shown in Figure 2.8, pull-based data fetches experience three sequential phases of communication; in contrast, the scheduling of downstream tasks and their remote data

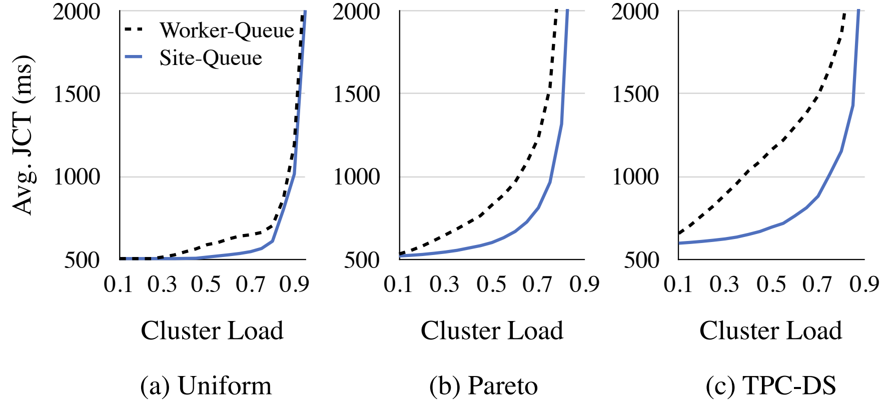



Figure 2.9: Job performance with Site- and Worker-Queue approaches. Variance in task durations increases from (a) to (c).

reads are pipelined in the push-based model, improving their completion times.

### 2.5.2 Why Does Early-Binding Help?

Assume that the coordinator continuously allocates tasks to a remote site's $k$ CPUs. For each task completion, the pull-based model takes one RTT for the coordinator to receive the task completion notification and then send the next task assignment; during this period, the task is queued up in the coordinator for scheduling. Hence, on average, $\frac{i-1}{k}$ RTTs are wasted before the $i^{th}$ task runs. The push-based model can save up to $\frac{i-1}{k}$ RTTs for the $i^{th}$ task by pipelining inter-task assignments and executions. Our analysis over 44 datacenters using the measured inter-site latencies shows that, compared to late-binding, the push-based model can achieve an average improvement of 153 ms for the data fetch of every downstream task (more details in Appendix A.1). Such gaps become magnified at scale with a large number of small tasks, e.g., a large fan-out, latency-sensitive job.

One may consider pulling multiple tasks for individual workers at a time, but the push model provides more flexibility. Pushing from the coordinator can react to online scheduling better. When tasks arrive in an online manner, multiple tasks may not be available for pulling at a time. e.g., when a new task keeps arriving right after serving a pull request, pulling multiple tasks degenerates into pulling one by one.

Moreover, by late-binding task assignments within the site, our site-manager approach enables more flexibility than pushing tasks to individual workers (i.e., maintaining one queue per worker). To evaluate this, we ran three workloads across 10 EC2 sites, where all workloads have the same average task duration from TPC-DS benchmarks but differ in their distributions of task durations. Figure 2.9 shows that the site-manager approach achieves superior job performance owing to better work balance, particularly when task durations are skewed.

### 2.5.3 How to Push the Right Number of Tasks?

Determining the number of queued-up tasks for site managers is crucial for balancing worker utilization versus job completion times. On the one hand, queuing up too few tasks leads to underutilization, inflating $t_{queue}$ due to lower system throughput. On the other hand, queuing up too many leads to sub-optimal work assignments because of insufficient knowledge when early-binding, which inflates job completion times as well (see Appendix A.2 for more details).

**Our target:** Intuitively, as long as a worker is not waiting to receive work, queuing more tasks does not provide additional benefits for improving utilization. To fully utilize the resource, we expect the total execution time of the queued-up tasks will occupy the CPU before the next task assignment arrives, which is the key to strike the balance between utilization and job performance.

**Our solution:** When every task's duration is known, the number of queued-up tasks can adapt to the instantaneous load such that the total required execution time of the queued-up tasks keeps all the workers in a site busy, but not pushing any more to retain maximum flexibility for scheduling across sites. However, individual task durations are often highly skewed in practice [41], while the overall distribution of task durations is often stable over a short period [208, 106].

Even without presuming task-specific characteristics or distributions, we can still approximate the ideal queue length at every site dynamically for a given resource utilization target. We model the total available cycles in each scheduling round as our target, and the duration of each queued-up task is a random variable. This can be mapped into a packing problem, where we have to figure out how many random variables to sum up to achieve the targeted sum.

When the individual task duration is not available, we extend Hoeffding's inequality, and inject the utilization target into our model to determine the desired queue length (Appendix A.3 for a formal result and performance analysis). Hoeffding's inequality is known to characterize how the sum of random variables deviates from its expected value with the minimum, the average, and the maximum of variables [125]. We extend it but filter out the outliers in tasks, wherein we rely on three statistics – 5th percentile, average, and 95th percentile (which are often stable) – of the task duration by monitoring the tasks over a period. As the execution proceeds, the coordinator in Sol inquires the model to generate the target queue size, whereby it dynamically pushes tasks to each site to satisfy the specified utilization. Note that when the network latency becomes negligible, our model outputs zero queue length as one would expect.

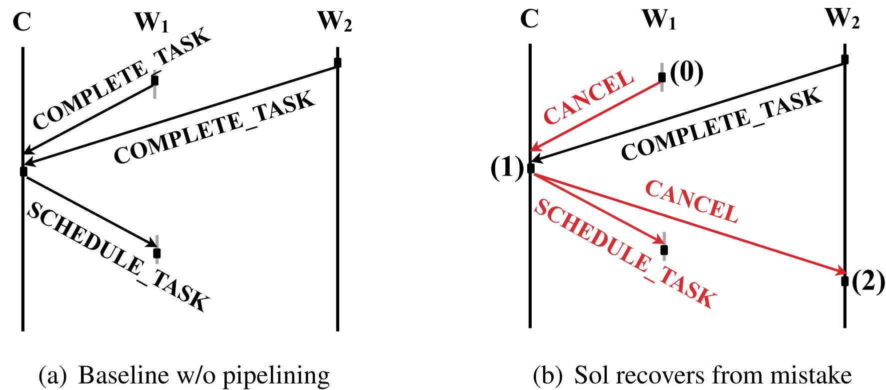



Figure 2.11: (b) The recovery process is shown in red. (0) $W_1$ detects large output, and sends CANCEL message to the coordinator $C$ for task rescheduling. (1) Upon receiving the update, $C$ waits until it gathers required information, then reschedules task, and (3) cancels task in $W_2$.

### 2.5.4 How to Push Tasks with Dependencies?

In the presence of task dependencies, where tasks may depend on those in their parent stage(s), pushing tasks is challenging, since it creates a tradeoff between the efficiency and quality of pipelining. For latency-sensitive tasks, we may want to push downstream tasks to save round-trip coordination even before the upstream output is available. However, for bandwidth-intensive tasks, pushing their downstream tasks will not bring many benefits; this may even miss optimal task placements due to insufficient knowledge about the outputs from all upstream tasks [243, 211]. Sol, therefore, has to reconcile between latency- and bandwidth-sensitive tasks at runtime without presuming task properties.

To achieve the desired pipelining efficiency for latency-bound tasks, Sol speculates the best placements for downstream tasks. Our straw-man heuristic is first pushing the downstream task to the site with the least work, with an aim to minimize the queueing time on the site. Moreover, Sol can refine its speculation by learning from historical trends (e.g., recurring jobs) or the iterative nature of many jobs. For example, in stream processing and machine learning, the output partitions computed for every batch are largely similar [279], so are their task placements [242]. As such, Sol can reuse the placement decisions in the past run.

However, even when a bandwidth-intensive task is pushed to a suboptimal site, Sol can gracefully retain the scheduling quality via worker-initiated re-scheduling. Figure 2.11 shows the control flow of the recovery process in Sol and the baseline. In Figure 2.11(b), we push upstream tasks to workers $W_1$ and $W_2$, and downstream tasks are pushed only to $W_2$. As the upstream task in $W_1$ proceeds, the task manager detects large output is being generated, which indicates we are in the regime of bandwidth-intensive tasks. (0) Then $W_1$ notifies the coordinator C of task completion and a CANCEL message to initiate rescheduling for the downstream task. (1) Upon receiving the

CANCEL message, the coordinator will wait until it collects output metadata from $W_1$ and $W_2$. The coordinator then reschedules the downstream task, and (2) notifies $W_2$ to cancel the pending downstream task scheduled previously. Note that the computation of a downstream task will not be activated unless it has gathered the upstream output.

As such, even when tasks are misclassified, Sol performs no worse than the baseline (Figure 2.11(a)). First, the recovery process does not introduce more round-trip coordinations due to rescheduling, so it does not waste time. Moreover, even in the worst case, where all upstream tasks have preemptively pushed data to the downstream task by mistake, the total amount of data transfers is capped by the input size of the downstream. However, note that the output is pushed only if it is latency-sensitive, so the amount of wasted bandwidth is also negligible as the amount of data transfers is latency-bound.

### 2.5.5 How to Handle Failures and Uncertainties?

Fault tolerance and straggler mitigation are enforced by the local site manager and the global coordinator. Site managers in Sol try to restart a failed task on other local workers; failures of long tasks or persistent failures of short tasks are handled via coordination with the remote coordinator. Similarly, site managers track the progress of running tasks and selectively duplicate small tasks when their execution lags behind. Sol can gracefully tolerate site manager failures by redirecting workers' control messages to the central coordinator, while a secondary site manager takes over (§2.8.5).

Moreover, Sol can guarantee a bounded performance loss due to early-binding even under uncertainties. To ensure a task at the site manager will not experience arbitrarily large queueing delay, the site manager can withdraw the task assignment when the task queueing delay on this site exceeds $\Delta$. As such, the total performance loss due to early-binding is ($\Delta$ + RTT), since it takes one RTT to push and reclaim the task.

## 2.6 Sol Data Plane

In existing execution engines, the amount of CPU allocated to a task when it is reading data is the same as when it later processes the data. This tight coupling between resources leads to resource underutilization (§2.3.2). In this section, we introduce how to improve $t_{queue}$ by mitigating HOL blocking.

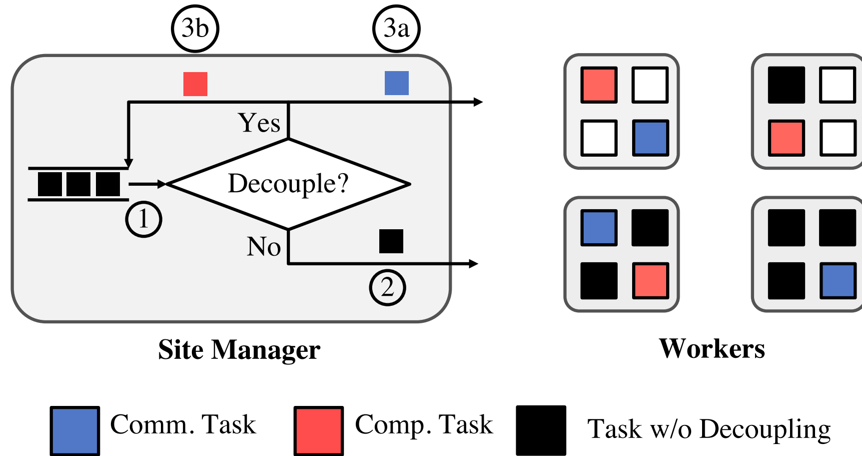



Figure 2.12: High-level overview of data plane decoupling.

### 2.6.1 How to Decouple the Provisioning of Resources?

To remove the coupling in resource provisioning, Sol introduces dedicated communication tasks,[6] which fetch task input from remote worker(s) and deserialize the fetched data, and computation tasks, which perform the computation on data. The primary goal of decoupling is to scale down the CPU requirements when multiple tasks have to fetch data over low-bandwidth links.

Communication and computation tasks are internally managed by Sol without user intervention. As shown in Figure 2.12, ① when a task is scheduled for execution, the site manager checks its required input for execution and reserves the provisioned resource on the scheduled worker. ② Bandwidth-insensitive tasks will be dispatched directly to the worker to execute. (3a) However, for tasks that need large volumes of remote data, the site manager will notify the task manager on the scheduled worker to set up communication tasks for data preparation. (3b) At the same time, the corresponding computation tasks are marked as inactive and do not start their execution right away. Once input data is ready for computation, the site manager will activate corresponding computation tasks to perform computation on the fetched data.

Although decoupling the provisioning of computation and communication resources will not speed up individual tasks, it can greatly improve overall resource utilization. When the input for a task's computation is being fetched by the communication task, by oversubscribing multiple computation tasks' communication to fewer communication tasks, Sol can release unused CPUs and repurpose them for other tasks. In practice, even the decoupled job can benefit from its own decoupling; e.g., when tasks in different stages can run in parallel, which is often true for jobs with complicated DAGs, computation tasks can take up the released CPUs from other stages in decoupling.

[6]Each communication task takes one CPU core by default in our design.

### 2.6.2 How Many Communication Tasks to Create?

Although decoupling is beneficial, we must avoid hurting the performance of decoupled jobs while freeing up CPUs. A key challenge in doing so is to create the right number of communication tasks to fully utilize the available bandwidth. Creating too many communication tasks will hog CPUs, while creating too few will slow down the decoupled job.

We use a simple model to characterize the number of required communication tasks. There are two major operations that communication tasks account for: (i) fetch data with CPU cost $C_{I/O}$ every time unit; (ii) deserialize the fetched data simultaneously with CPU cost $C_{deser}$ in unit time. When the decoupling proceeds with I/O bandwidth $B$, the total requirement of communication tasks $N$ can be determined based on the available bandwidth ($N = B \times (C_{I/O} + C_{deser})$).

Referring to the network throughput control, we use an adaptive tuning algorithm. When a new task is scheduled for decoupling, the task manager first tries to hold the provisioned CPUs to avoid resource shortages in creating communication tasks. However, the task manager will opportunistically cancel the launched communication task after its current fetch request completes, and reclaim its CPUs if launching more communication tasks does not improve bandwidth utilization any more.[7] During data transfers, the task manager monitors the available bandwidth using an exponentially weighted moving average (EWMA).[8] As such, the task manager can determine the number of communication tasks required currently: $N_{current} = \lceil \frac{B_{current}}{B_{old}} \times N_{old} \rceil$. Therefore, it will launch more communication tasks when more bandwidth is available and the opposite when bandwidth decreases. Note that the number of communication tasks is limited by the total provisioned CPUs for that job to avoid performance interference.

### 2.6.3 How to Recover CPUs for Computation?

Sol must also ensure that the end-to-end completion time on computation experiences negligible inflation. This is because when the fetched data is ready for computation, the decoupled job may starve if continuously arriving computation tasks take up its released computation resources.

We refer to not decoupling as the *baseline* strategy, while waiting for the entire communication stage to finish as the *lazy* strategy. The former wastes resources, while the latter can hurt the decoupled job. Figure 2.13 depicts both.

Instead, Sol uses a *greedy* strategy, whereby as soon as some input data becomes ready (from upstream tasks), the site manager will prioritize the computation task corresponding to that data over other jobs and schedule it. As such, we can gradually increase its CPU allocation instead of trying to acquire all at once or holding onto all of them throughout.

[7] This introduces little overhead, since the data fetch is in a streaming manner, wherein the individual block is small.

[8] $B_{current} = \alpha\ B_{measured} + (1 - \alpha)\ B_{old}$, where $\alpha$ is the smoothing factor ($\alpha$ = 0.2 by default) and $B$ denotes the available bandwidth over a period.

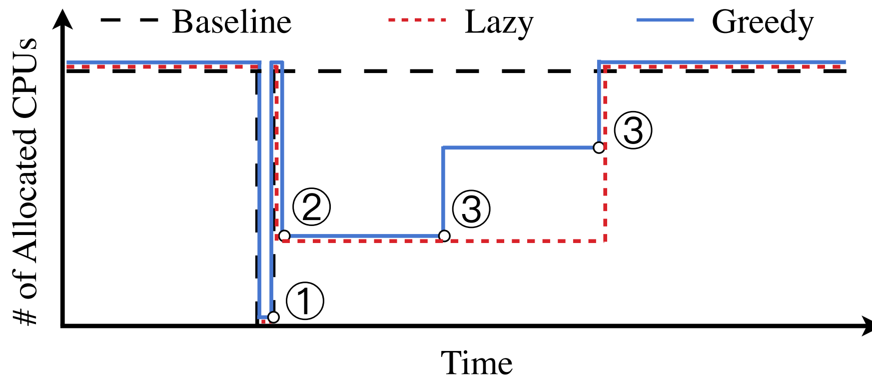



Figure 2.13: Three strategies to manage decoupled jobs. We adopt the Greedy strategy: ① when the downstream tasks start, they hold the reserved CPUs. ② But the task manager will reclaim the unused CPUs, and ③ activate the computation task once its input data is ready. The first trough marks the stage boundary.

### 2.6.4 Who Gets the Freed up CPUs?

Freed up CPUs from the decoupled jobs introduce an additional degree of freedom in scheduling. Resource schedulers can assign them in a FIFO or fair manner. As the duration of communication tasks can be estimated by the remaining data fetches and the available bandwidth, the scheduler can plan for the extra resources into the future, e.g., similar to [105].

## 2.7 Implementation

While our design criteria are not married to specific execution engines, we have implemented Sol in a way that keeps it API compatible with Apache Spark [4] in order to preserve existing contributions in the big data stack.

**Control and Data Plane** To implement our federated architecture, we add site manager modules to Spark, wherein each site manager keeps a state store for necessary metadata in task executions, and the metadata is shared across tasks to avoid redundant requests to the remote coordinator. The central coordinator coordinates with the site manager by heartbeat as well as the piggyback information in task updates. During executions, the coordinator monitors the network latency using EWMA in a one second period. This ensures that we are stable despite transient latency spikes. When the coordinator schedules a task to the site, it assigns a dummy worker for the pipelining of dependent tasks (e.g., latency-bound output will be pushed to the dummy worker). Similar to delay scheduling, we set the queueing delay bound $\Delta$ to 3 seconds [280]. Upon receiving the completion of upstream tasks, the site manager can schedule the downstream task more intelligently with late-binding. Meanwhile, the output information from upstream tasks is backed up in the state

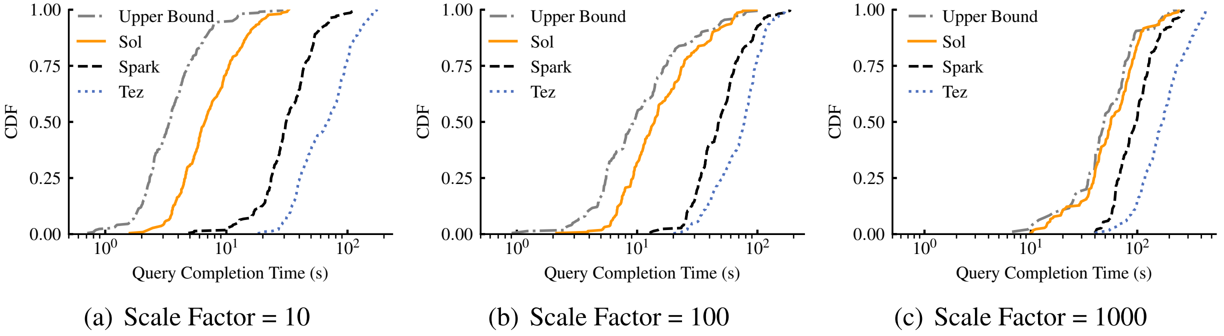



Figure 2.14: Performance of Sol, Spark, and Tez on TPC query processing benchmark.

store until their completion.

**Support for Extensions** Our modifications are to the core of Apache Spark, so users can enjoy existing Spark-based frameworks on Sol without migrations of their codebase. Moreover, for recent efforts on WAN-aware optimizations, Sol can support those more educated resource schedulers or location-conscious job schedulers by replacing the default, but further performance analysis of higher-layer optimizations is out of the scope of this paper. To the best of our knowledge, Sol is the first execution engine that can optimize the execution layer across the design space.

## 2.8 Evaluation

In this section, we empirically evaluate Sol through a series of experiments using micro and industrial benchmarks. Our key results are as follows:

- Sol improves performance of individual SQL and machine learning jobs by 4.9×–11.5× w.r.t. Spark and Tez execution engines in WAN settings. It also improves streaming throughput by 1.35×–3.68× w.r.t. Drizzle (§2.8.2).
- In online experiments, Sol improves the average job performance by 16.4× while achieving 1.8× higher utilization (§2.8.3).
- Even in high bandwidth-low latency (LAN) setting, Sol improves the average job performance by 1.3× w.r.t. Spark; its improvement in low bandwidth-low latency setting is 3.9× (§2.8.4).
- Sol can recover from failures faster than its counterparts, while effectively handling uncertainties (§2.8.5).

### 2.8.1 Methodology

**Deployment Setup** We first deploy Sol in EC2 to evaluate individual job performance using instances distributed over 10 regions.[9] Our cluster allocates 4 *m4.4xlarge* instances in each region. Each has 16 vCPUs and 64GB of memory. To investigate Sol performance on multiple jobs in diverse network settings, we set up a 40-node cluster following our EC2 setting, and use *Linux Traffic Control* to perform network traffic shaping to match our collected profiles from 10 EC2 regions.

**Workloads** We use three types of workloads in evaluations:

1. *SQL*: we evaluate 110 industry queries in TPC-DS/TPC-H benchmarks [34, 35]. Performance on them is a good demonstration of how good Sol would perform in real-world applications handling jobs with complicated DAGs.
2. *Machine learning*: we train three popular federated learning applications: linear regression, logistic regression, and k-means, from Intel's industry benchmark [132]. Each training data consists of 10M samples, and the training time of each iteration is dominated by computation.
3. *Stream processing*: we evaluate the maximum throughput that an execution design can sustain for *WordCount* and *TopKCount* while keeping the end-to-end latency at a given target. We define the end-to-end latency as the time from when records are sent to the system to when results incorporating them appear.

**Baselines** We compare Sol to the following baselines:

1. *Apache Spark* [278] and *Apache Tez* [223]: the mainstream execution engines for generic workloads in datacenter and wide-area environments.
2. *Drizzle* [242]: a recent engine tailored for streaming applications, optimizing the scheduling overhead.

**Metrics** Our primary metrics to quantify performance are the overarching user-centric and operator-centric objectives, including *job completion time (JCT)* and *resource utilization*.

### 2.8.2 Performance Across Diverse Workloads in EC2

In this section, we evaluate Sol's performance on individual jobs in EC2, with query processing, machine learning, and streaming benchmarks.

[9] California, Sydney, Oregon, Ohio, Tokyo, Mumbai, Seoul, Singapore, Sao Paulo and Frankfurt.

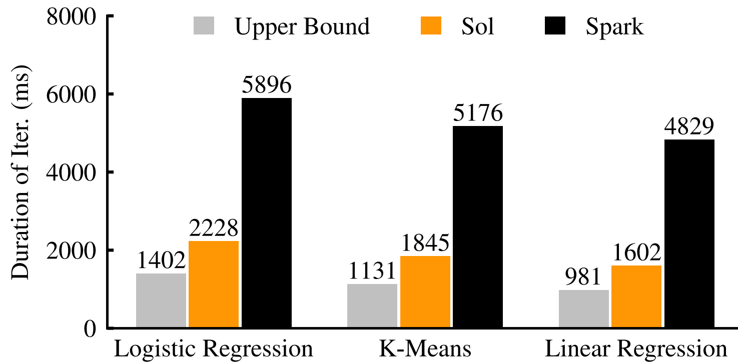



Figure 2.15: Performance on machine learning.

**Sol outperforms existing engines** Figure 2.14 shows the distribution of query completion times of 110 TPC queries individually on (10, 100, 1000) scale factor datasets. As expected, Sol and Spark outperform Tez by leveraging their in-memory executions. Meanwhile, Sol speeds up individual queries by 4.9× (11.5×) on average and 8.6× (23.3×) at the 95th percentile over Spark (Tez) for the dataset with scale factor 10. While these queries become more bandwidth- and computation-intensive as we scale up the dataset, Sol can still offer a noticeable improvement of 3.4× and 1.97× on average compared to Spark on datasets with scale factors 100 and 1000, respectively. More importantly, Sol outperforms the baselines across all queries.

Sol also benefits machine learning and stream processing. For such predictable workloads, Sol pipelines the scheduling and data communication further down their task dependencies. Figure 2.15 reports the average duration across 100 iterations in machine learning benchmarks, where Sol improves the performance by 2.64×-3.01× w.r.t. Spark.

Moreover, Sol outperforms Drizzle [242] in streaming workloads. Figure 2.16 shows that Sol achieves 1.35×–3.68× higher throughput than Drizzle. This is because Sol follows a *push-based* model in both control plane coordinations and data plane communications to pipeline round-trips for inter-site coordinations, while Drizzle optimizes the coordination overhead between the coordinator and workers. Allowing a larger target latency improves the throughout, because the fraction of computation time throughout the task lifespan increases, and thus the benefits from Sol become less relevant.

**Sol is close to the upper bound performance** To explore how far Sol is from the optimal, we compare Sol's performance in the high latency setting against Sol in a hypothetical latency-free setting [10] , which is a straightforward upper bound on its performance. While high latencies lead to an order-of-magnitude performance degradation on Spark, Sol is effectively approaching the optimal. As shown in Figure 2.14 and Figure 2.15, Sol's performance is within 3.5× away from the

[10] We create a 40-node cluster in a single EC2 region.

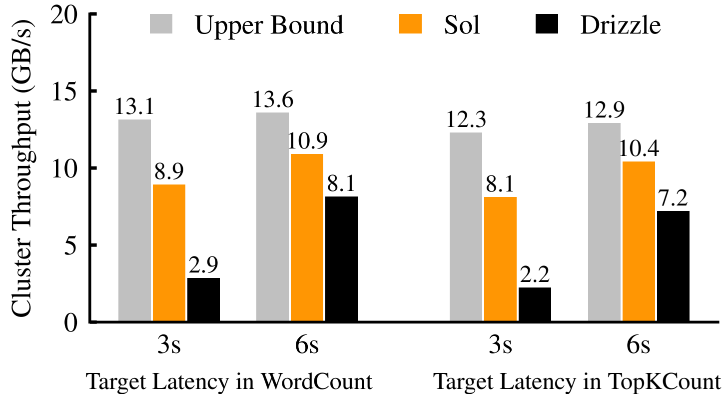



Figure 2.16: Performance on stream processing. Higher is better.

upper bound. As expected in Figure 2.14(c), this performance gap narrows down as Sol has enough work to queue-up for hiding the coordination delay.

### 2.8.3 Online Performance Breakdown

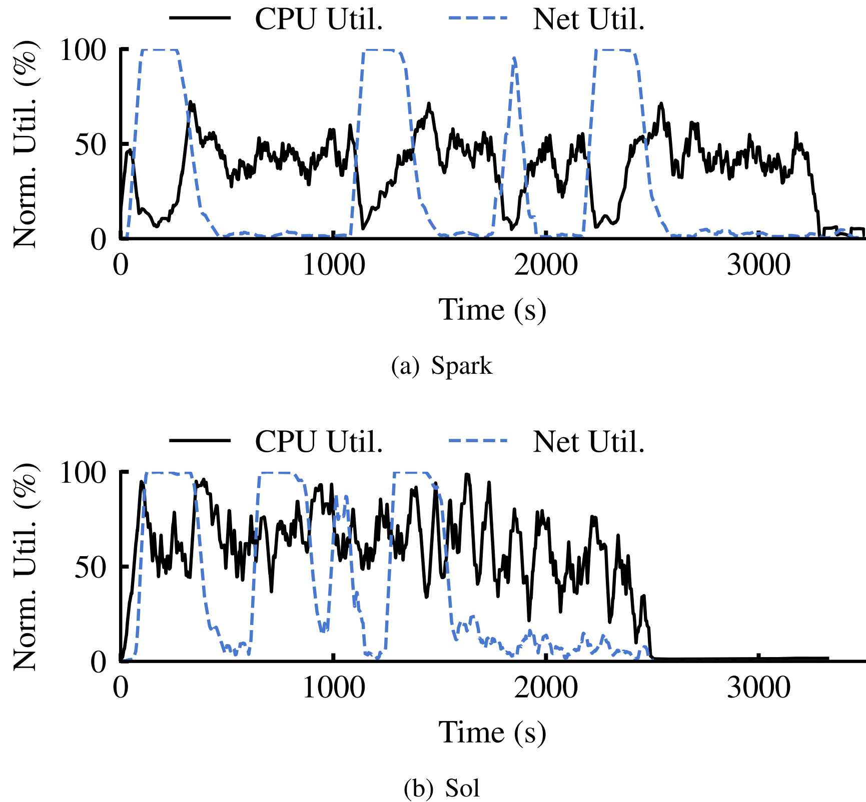
So far, we have evaluated Sol in the offline setting with individual jobs. Here we move on to evaluate Sol with diverse workloads running concurrently and arriving in an online fashion with our cluster. Specifically, we evaluate Sol in an online setting, where we run 160 TPC queries – randomly drawn from the (10, 100)-scale TPC benchmarks – run as foreground, interactive jobs, and bandwidth-intensive CloudSort jobs [27] – each has 200 GB or 1 TB GB input – in the background. The TPC queries are submitted following a Poisson process with an average inter-arrival time of 10 seconds, while the CloudSort jobs are submitted every 300 seconds.

We evaluate Sol and Spark using two job schedulers:

1. *FIFO*: Jobs are scheduled in the order of their arrivals, thus easily resulting in Head-of-Line (HOL) blocking;
2. *Fair sharing*: Jobs get an equal share of resources, but the execution of early submitted jobs will be prolonged.

These two schedulers are prevalent in real deployments[123, 3], especially when job arrivals and durations are unpredictable.

**Improvement of resource utilization** Figure 2.17 shows a timeline of normalized resource usage for both network bandwidth and total CPUs with the FIFO scheduler. A groove in CPU utilization and a peak in network utilization dictate the execution of bandwidth-intensive background jobs. Similarly, a low network utilization but high CPU utilization implicates the execution of foreground jobs. We observe Sol improves the CPU utilization by 1.8× over Spark. The source of this




(a) Spark

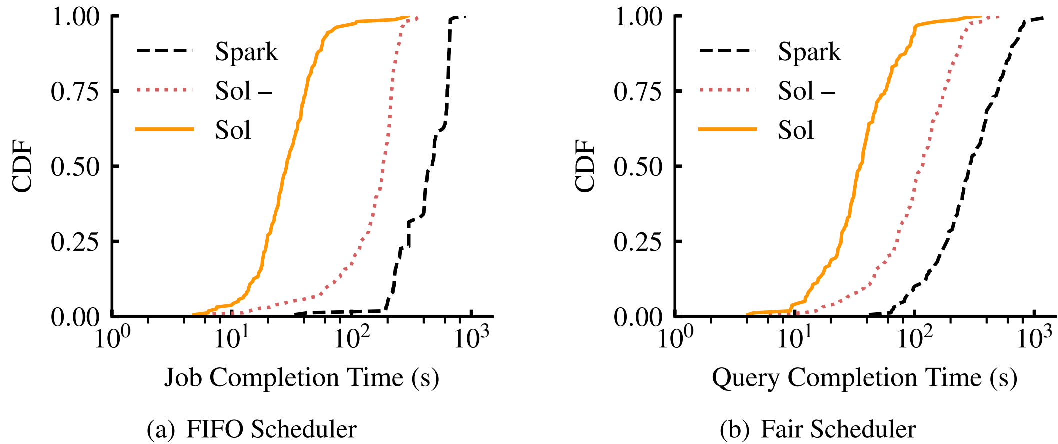
(b) Sol

Figure 2.17: Resource utilization over time.

improvement comes from both control and data planes: (i) Sol pipelines high-latency coordinations, and thus workers are busy in running tasks all the time. (ii) Sol flexibly repurposes the idle CPU resources in the presence of bandwidth-intensive jobs, thus achieving higher utilizations by orchestrating all jobs. Note that the CPU resource is not always fully saturated in this evaluation, because the cluster is not extremely heavy-loaded given the arrival rate. Therefore, we believe Sol can provide even better performance with heavy workloads, wherein the underutilized resource can be repurposed for more jobs with decoupling. Results were similar for the fair scheduler too.

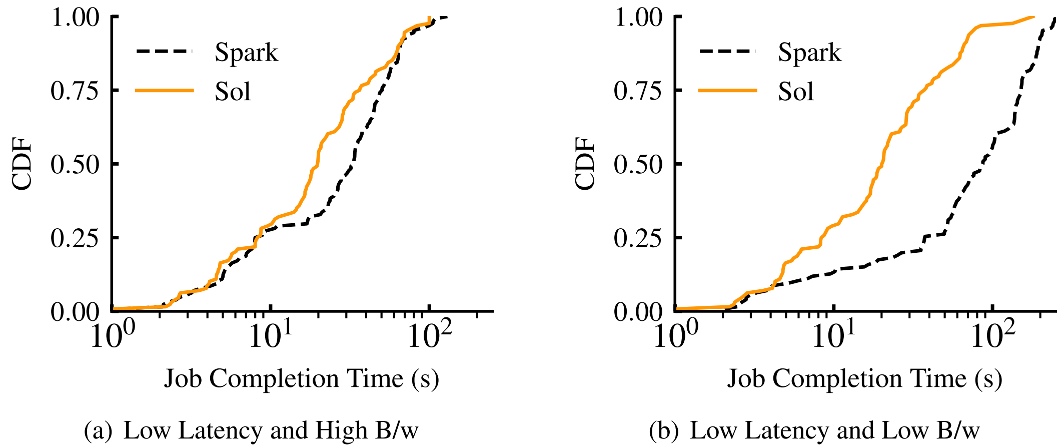
**Improvement of JCTs** Figure 2.18(a) and Figure 2.18(b) report the distribution of job completion times with FIFO and fair schedulers respectively. The key takeaways are the following. First, simply applying different job schedulers is far from optimal. With the FIFO scheduler, when CloudSort jobs are running, all the frontend jobs are blocked as background jobs hog all the available resources. While the fair scheduler mitigates such job starvation by sharing resources across jobs, it results in a long tail as background jobs are short of resources.

Instead, Sol achieves better job performance by improving both the intra-job and inter-job completions in the task execution level: (i) Early-binding in the control plane improves small jobs,




(a) FIFO Scheduler (b) Fair Scheduler

Figure 2.18: JCT with online job arrival using different cluster schedulers. Sol- is Sol without data plane decoupling.




(a) Low Latency and High B/w (b) Low Latency and Low B/w

Figure 2.19: Sol performance in other design space.

whereby the cluster can finish more jobs in a given time. Hence, even the simple pipelining can achieve an average improvement of 2.6× with the FIFO scheduler and 2.5× with the fair scheduler. (ii) With data plane decoupling, the latency-sensitive jobs can temporarily enjoy under-utilized resource without impacting the bandwidth-intensive jobs. We observe that the performance loss of bandwidth-intensive jobs is less than 0.5%. As the latency-sensitive jobs complete faster, bandwidth-intensive jobs can take up more resources. As such, Sol further improves the average JCTs w.r.t. Spark with both FIFO (average 16.4×) and fair schedulers (average 8.3×).

### 2.8.4 Sol's Performance Across the Design Space

We next rerun the prior online experiment to investigate Sol's performance in different network conditions with our cluster.

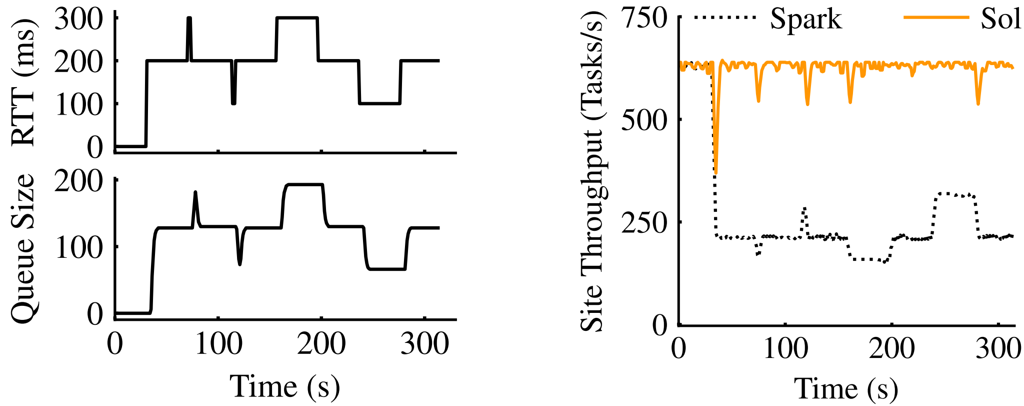



Figure 2.20: Sol performance under latency variations.

**High bandwidth-low latency network** In this evaluation, each machine has 10 Gbps bandwidth, and the latency across machines is <1 ms. Figure 2.19 shows the distribution of JCTs. The benefits of data plane decoupling depend on the time spent on data exchanges over the network. Although jobs are prone to finishing faster in this favorable environments, Sol can still improve over Spark by 1.3× on average by mitigating the HOL blocking with the decoupling in task executions.

**Low bandwidth-low latency network** In practice, users may deploy cheap VMs to perform time-insensitive jobs due to budgetary constraints. We now report the JCT distribution in such a setting, where each machine has 1 Gbps low bandwidth and negligible latency. As shown in Figure 2.19(b), Sol largely outperforms Spark by 3.9×. This gain is again due to the presence of HOL blocking in Spark, where bandwidth-intensive jobs hog their CPUs when tasks are reading large output partitions over the low-bandwidth network.

Note that the high latency-high bandwidth setting rarely exists. As such, Sol can match or achieve noticeable improvement over existing engines across all practical design spaces.

### 2.8.5 Sol's Performance Under Uncertainties

As a network-aware execution engine, Sol can tolerate different uncertainties with its federated design.

**Uncertainties in network latency** While Sol pushes tasks to site managers with early-binding under high network latency, its performance is robust to latency jitters. We evaluate Sol by continuously feeding our cluster with inference jobs; each scans a 30 GB dataset and the duration of each task is around 100 ms. We snapshot a single site experiencing transient or lasting latency

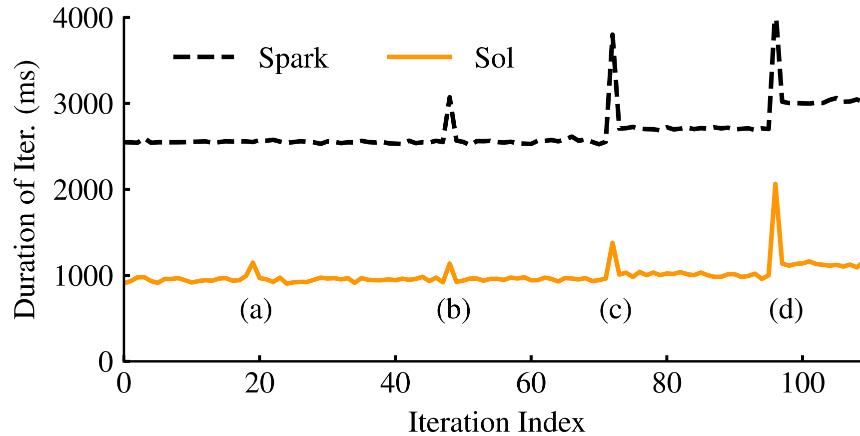



Figure 2.21: Impact of different failures on iteration duration for Sol and Spark: (a) Sol site manager failure, (b) task failure, (c) node failure , and (d) site-wide failure.

variations. As shown in Figure 2.20, Sol proceeds more tasks than Spark with early-binding of tasks. Moreover, Sol can efficiently react to RTT variations by adaptively tuning its queue size.

**Uncertainties in failure** Figure 2.21 compares Sol's performance with Spark under different failures. In this evaluation, we train a long running linear regression in our 10-site deployment, and each iteration performs two stages: training on the data and the aggregation of updates. When the site manager fails (a), Sol restarts the site manager on other local machines, and reschedules the missing queued-up tasks. The recovery of site managers is pipelined with task executions, experiencing little overhead in job performance. Task failures (b) and machine failures (c) in Spark require a tight coordination with the remote coordinator, but Sol handles such failures by coordinating the site manager. Upon detecting task failures, the site manager restarts the task on other locally available machines with its metadata, while asynchronously notifying the coordinator. As such, Sol suffers little overhead by hiding the failures silently. Although the coordinator needs to take charge of rescheduling in both Sol and Spark under site-wide failures (d), tasks in Sol complete faster.

## 2.9 Discussion and Future Work

**Fine-grained queue management.** By capturing the range of task durations, Sol pushes the right number of tasks to site managers at runtime. However, Hoeffding's inequality can be suboptimal, especially when the variance of task durations becomes much greater than their average [125]. Further investigations on the queue management of site managers are needed. To this end, one possible approach is to build a context-aware machine learning model (e.g., reinforcement learning) to decide the optimal queue length [63].

**Performance analysis of geo-aware efforts.** As the first federated execution engine for diverse network conditions, Sol can serve a large body of existing efforts for geo-distributed data analytics [211, 243, 134]. Although these works do not target latency-bound tasks, for which Sol shows encouraging improvements with control plane optimizations, it would be interesting to investigate Sol's improvement for bandwidth-intensive workloads after applying techniques from existing geo-distributed frameworks.

## 2.10 Related Work

**Geo-distributed storage and data analytics** Numerous efforts strive to build frameworks operating on geo-distributed data. Recent examples include geo-distributed data storage [258, 177] and data analytics frameworks [128, 31]. Geode [244] aims at generating query plans that minimize data transfers over the WAN, while Clarinet [243] and Iridium [211] develop the WAN-aware query optimizer to optimize query response time subject to heterogeneous WAN bandwidth. These optimizations for data analytics lie on the scheduler layer and could transparently leverage Sol for further gains (§2.7).

**Data Processing Engines** The explosion of data volumes has fostered the world of MapReduce-based parallel computations [80]. Naiad [196] and Flink [60] express data processing as pipelined fault-tolerant data flows, while the batch processing on them performs similarly to Spark [278]. The need for expressive user-defined optimizations motivates Dryad [135] and Apache Tez [223] to enable runtime optimizations on execution plans. These paradigms are designed for well-provisioned networks. Other complementary efforts focus on reasoning about system performance [206, 205], or decoupling communication from computation to further optimize data shuffles [71, 70]. Our work bears some resemblance, but our focus is on designing a network-aware execution engine.

**Speeding up data-parallel frameworks** Although Nimbus [184] and Drizzle [242] try to speed up execution engines, they focus on amortizing the computation overhead of scheduling iterative jobs. Hydra [79] democratizes the resource management for jobs across multiple groups. While Yaq-c [214] discusses the tradeoff between utilization and job performance in queue management, its solution is bound to specific task durations without dependencies. Moreover, we optimize task performance inside execution engines.

## 2.11 Summary

As modern data processing expands due to changing workloads and deployment scenarios, existing execution engines fall short in meeting the requirements of diverse design spaces. In this

chapter, we explored the possible designs beyond those for datacenter networks and presented Sol, a federated execution engine that emphasizes an early-binding design in the control plane and decoupling in the data plane. In comparison to the state-of-the-art, Sol can match or achieve noticeable improvements in job performance and resource utilization across all practical points in the design space.

# CHAPTER 3

## Minimizing Training Execution via Automated Training Warmup

In the previous chapter, we showed how to efficiently preprocess data for ML development at a low cost – e.g., by performing in-situ computation on data and only aggregating the processed data. This chapter moves on to the experimentation stage, in which developers often train hundreds of models to find the best performing model on the processed data [281, 239]. In this process, we develop ModelKeeper, the first automated training warmup system that accelerates the training of deep neural networks (DNNs) by repurposing previously-trained models. In a shared cluster, ModelKeeper reduces the cost of model training by exploiting the insight that *initializing a training job's model by transforming an already-trained model's weights can jump-start it and reduce the total amount of training needed.*

The rest of this chapter is organized as follows. The next section introduces the potential to leverage model similarity. Section 3.2 outlines the proposed system. Section 3.3 presents analyses into real-world model zoos, showcasing the opportunity of model transformation. The design section, Section 3.5, illustrates how ModelKeeper scalably selects a parent model with high similarity and good model accuracy, and performs structure-aware transformation across model architectures. We then implement and ModelKeeper in Section 3.6 and Section 3.7, respectively. Next, we discuss in Section 3.8 its current limitations and future research directions, survey related work in Section 3.9, and summarize our findings in Section 3.10.

### 3.1 Background

Modern machine learning (ML) clusters train thousands of deep neural networks (DNNs) every day [255, 139]. For a specific ML task, ML developers often start with exploring various model architectures using Neural Architecture Search (NAS) to find the one with desired accuracy [286]. In preparation for model serving, developers may train tens of models to customize the latency-accuracy trade-off across hardware [127, 78], to organize weak and powerful DNNs into different inference stages for fast feature extraction [77], and/or to dynamically select tens of models and combine their predictions to maximize ensemble accuracy [253, 110, 90]. Overall, from inception to deployment,

ML development often requires training hundreds of models across developers [281, 239].

Naturally, many recent advances in ML training optimizations have focused on faster DNN execution, e.g., by increasing parallelism [197, 267], improving communication [210, 144], or increasing GPU utilization [259, 108, 260, 276]. However, little has been done to exploit the natural similarity between models that are trained as part of the same NAS process, models targeting the same ML task in different hardware, or models embedded in different applications. Indeed, our analysis of three large CV and NLP model zoos shows that more than 60% of widely-used models can find an architecturally similar counterpart within the same zoo (§3.3.2).

In fact, *one can reduce the amount of training needed for model convergence by leveraging a well-trained model's weights to warm up the training of a new model.* This is because any DNN model is fundamentally a computation graph of tensor weights and operators; transforming the weights of trained models with similar architectures to a new model can accelerate model convergence (similar to transfer learning [272, 236] but across architectures).

Despite the potential for large benefits, there exists little systematic support for automated repurposing of weights. Today's frameworks may provide pre-trained models, but are limited to a few models and specific datasets, and/or require domain knowledge to manually search, transfer, and contribute a trained model's weights [20]. As such, ML developers have to train models from scratch more often [212]. A few recent AutoML frameworks (e.g., Retiarii [286]) repurpose trained models, but they are limited to individual jobs within a NAS task because they rely on the lineage of model mutation to enable the transfer. When models are submitted by various developers and/or frameworks with distinct architectures and performance requirements, these solutions do not apply.

## 3.2 Solution Outline

We introduce ModelKeeper, a cluster-wide training warmup system, to reduce the training execution needed for model convergence via automated model weight transformation (§3.4). ModelKeeper adaptively manages a collection of trained models (i.e., *model zoo*) from prior training jobs corresponding to different ML tasks. For a new training job, ModelKeeper selects and transforms a trained model's weights (i.e., *parent model*) to the training model (i.e., *query model*) before training takes place. It can benefit various ML applications, including exploratory training (e.g., improving Retiarii [286] further) and general training (e.g., using PyTorch [24]) of CV/NLP models, with few-lines-of-code change.

ModelKeeper addresses two primary challenges toward selecting a suitable parent model and repurposing its weights. First, ModelKeeper must determine similarity between two models (§3.5.1). Intuitively, we can treat each DNN model as a directed graph, where nodes represent tensors (layers) and edges represent data flows, and use heuristics for the classic NP-hard graph edit distance

problem [113] to find the matching similarity. However, maximizing matching by skipping nodes can be harmful because the computation of each tensor affects that of the subsequent ones in a trained parent model. To this end, we present a structure-aware dynamic programming approach to capture the similarity (transformable tensor weights) between two models. To scale to real-world zoos with thousands of models, we then introduce a two-stage hierarchical search algorithm to identify similar models efficiently.

Second, perfect matching is unlikely as two models are seldom identical. Therefore, given many candidate parent models with different similarity scores and each with different accuracy, which one to pick and then how to transform its weights to the query model (§3.5.2)? A more similar parent model enables transforming more weights, while a more accurate one implies a better training jump start after the transformation. When the two are at odds, we adopt a bucketing heuristic: potential parent models are put into different buckets in terms of their similarity to the query model, grouping comparable parent models together. We then pick the most accurate parent from the bucket containing the most similar parent models. Nevertheless, tensor mappings from the parent to the query model can be incomplete (e.g., due to non-identical architectures). To preserve maximal parent model information, we introduce width and depth operators to transform parent model weights into the query model with negligible overhead.

We have integrated ModelKeeper with four popular ML frameworks (§3.6): Ray [192], AutoKeras [145], MLFlow [281], and Microsoft NNI with Retiarii backend [286].[1] Our evaluations across thousands of DNN training jobs in CV and NLP applications (§3.7) show that ModelKeeper can save 23%–77% total amount of training needed (i.e., 1.3×–4.3× faster training) than the state-of-the-art without model accuracy drop, while efficiently serving cluster-scale warmup requests.

Overall, we make the following contributions in this chapter:

1. We present ModelKeeper, a system to enable automated training warmup for faster DNN training in clusters;
2. In order to maximize training speedup, we demonstrate how to scalably compute similarities between models and how to transform an already-trained model's weights to a yet-to-be trained model with little overhead;
3. We integrate ModelKeeper with multiple advanced ML frameworks, and evaluate it across thousands of CV and NLP models to show large improvements.

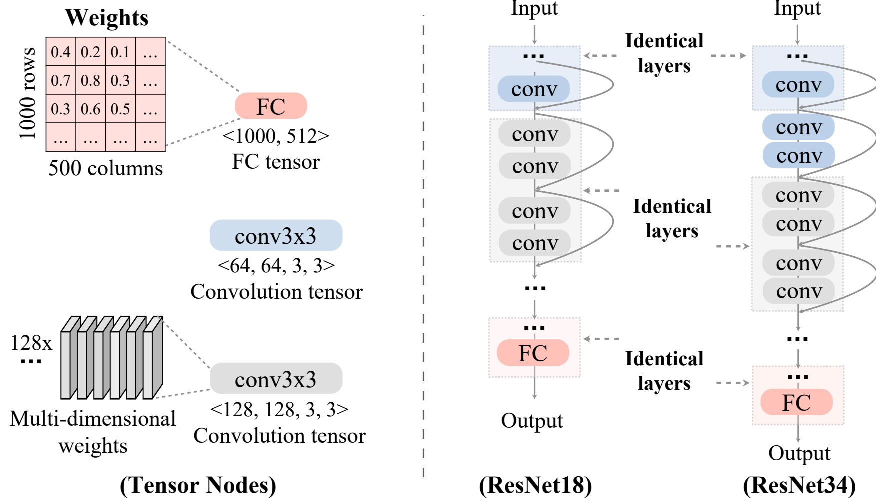



Figure 3.1: A DNN model is essentially a graph of tensors. Model outputs are determined by tensor weights and their control flow.

## 3.3 Motivation

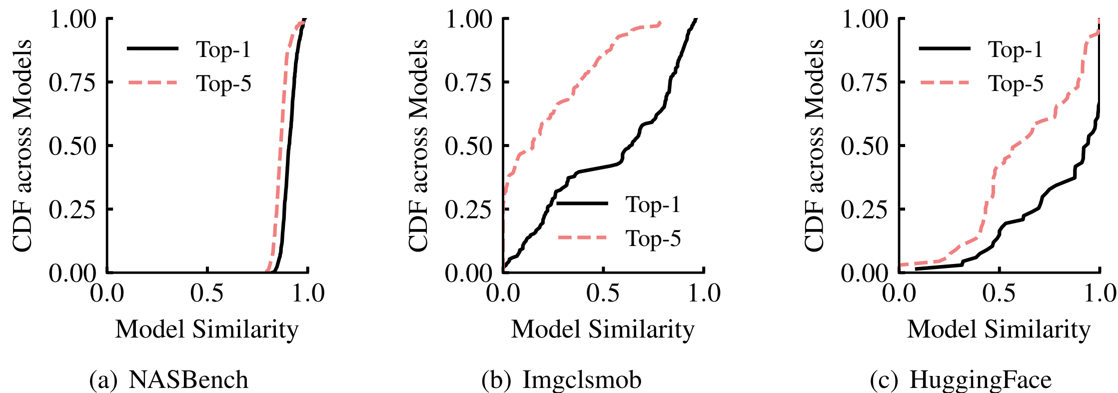
### 3.3.1 DNN Model Training

Modern DNN frameworks represent DNN computations as a directed computation graph with tens to thousands of nodes across branches (Figure 3.1) [140, 24]. Each node implies a mathematical tensor operation (e.g., matrix multiplication or convolution) along with its tensor weights and input, where weights are n-dimensional arrays typically consisting of floats. DNN training often covers thousands of iterations across mini-batches of data to minimize the training loss. In each iteration, the computation graph takes a data mini-batch as the input, and performs a (1) *forward pass*, where each node conducts the tensor operation on the output of parent nodes to get the training loss regarding the model output and ground truth; and a (2) *backward pass*, which updates the weight values, from the last to front tensors, using the gradients derived by the training loss with respect to the current weight. Therefore, the DNN model is essentially a graph of weights orchestrated by tensor operators, and training searches for the best weight values.

### 3.3.2 Opportunities for Repurposing Models

In this chapter, we focus on *reducing the amount of training needed* to train a new model by automatically repurposing the weights of previously trained models. Our approach of warming up the weights of a new model before its training starts is based on the following observations.

[1]ModelKeeper is available at https://github.com/SymbioticLab/ModelKeeper.




(a) NASBench (b) Imgclsmob (c) HuggingFace

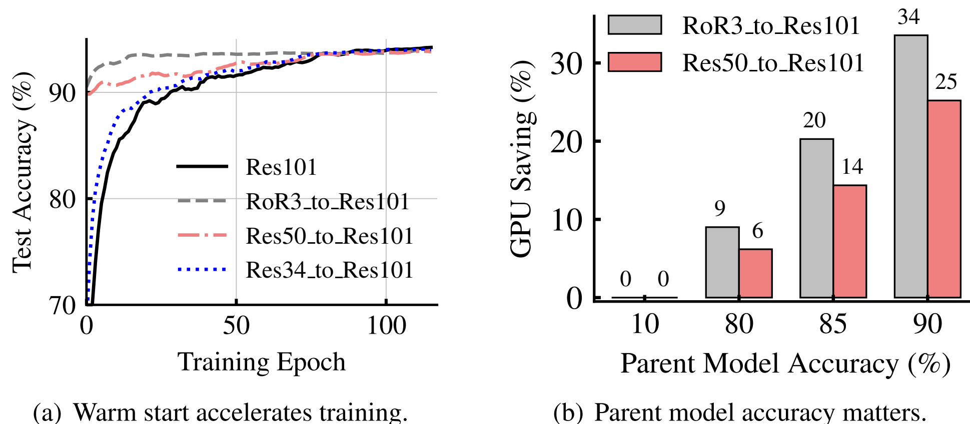
Figure 3.2: Pervasive model similarity in today's model zoos. We measure the top-1 and top-5 architectural similarities of each model to other models, and report the distribution across models. 1 indicates identical model architectures.

**Pervasive model similarity.** With the rapid increase in the number of ML training jobs in datacenters [139, 108], similarities between training jobs are increasing too [255]:

- First, for a specific ML task, ML developers often explore various model architectures using Neural Architecture Search (NAS) to find the preferred model architecture (e.g., better capacity-accuracy frontier [286]), or to investigate the performance consistency of new optimizations across models (e.g., ML ablation study) [190]. For example, Microsoft tuning clusters perform as many as 75 exploratory training jobs in median for user apps [180].
- Second, in preparation for ML deployment, developers can train dozens of models to either customize the latency-accuracy tradeoff across hardware (e.g., in video analysis systems [127, 141]), or to dynamically select tens of models and combine their predictions in order to maximize accuracy under changing loads in today's ensemble-serving systems [253, 110] (e.g., AWS Autogluon [90]).
- Third, the potential for similar models increases with increasing users. For example, over 100K ML models are submitted to Kaggle competitions each month [16]. Each competition can have thousands of participants developing their models independently, and participants are reported to have trained many similar models [81].

Indeed, our analysis of three large public model zoos – Imgclsmob [26] for ImageNet classification (435 models), HuggingFace [14] for English text generation (2.5K models), and NASBench [84] for NAS task (16K models) – reinforces these observations. Figure 3.2 reports the pairwise architectural similarity across models in each model zoo. We measure the similarity of each model to other zoo models[2] in terms of the normalized graph edit distance of two directed model computation

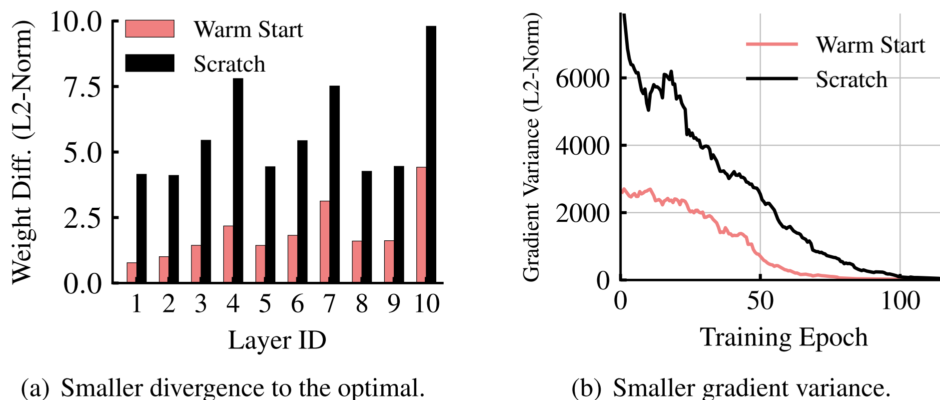
[2]To avoid over-optimistic identification of model similarity, we removed identical models in each zoo and focus on different model architectures.




(a) Warm start accelerates training.

(b) Parent model accuracy matters.

Figure 3.3: Transferring model weights from well-trained models with similar architectures can accelerate new model training.




(a) Smaller divergence to the optimal.

(b) Smaller gradient variance.

Figure 3.4: Warm start provides better initial weights search space. We use RoR3 to warm start ResNet101.

graphs [96] ($\in$ [0, 1]), where 1 indicates identical graphs. We observe that more than 60% of the models have at least one other model (top-1) in the zoo with a similarity over 0.6. Pervasive similarity is prominent in all these model zoos because modern models often rely on similar architecture designs but with wider/deeper layers or branches. For example, convolution layers are widely-used in CV models [120, 131], while NLP models are often stacked by attention layers [240].

**Similar models can warm-start training.** Recent theoretical [236] and empirical [272] efforts from the transfer learning community show that inheriting well-trained parent model weights can speed up the training of a new model, because this warm start enables an informed weight initialization (e.g., training from the basin of loss curvature). Yet, different from their focus on manually transfering the same model across datasets for better model generalization accuracy [202, 288], we notice that transforming a trained model's weights to a new model (i.e., across architectures)

can accelerate its training.

Consider the training of ResNet101 on CIFAR-10 dataset as an example. We copy the tensor weights of a well-trained parent model (e.g., ResNet50 or RoR3 [285]) to the ResNet101 tensor if two tensors have identical properties (e.g., same operator and weight dimensions), while the rest of the training proceeds as normal. We notice that (1) warmup training can reduce the amount of training needed, while obtaining the same final accuracy as training from scratch with random weight initialization (Figure 3.3(a)); and (2) the savings are more encouraging when inheriting from more similar models – similarity of ResNet34, ResNet50, and RoR3 to ResNet101 is 0.19, 0.48, and 0.85, respectively – and better performing models (Figure 3.3(b)), which respectively determine whether it is possible and beneficial to transform the weights.

These improvements are because they speed up the search in the space of weight values. If we consider ResNet101 as an example, (1) warming it up using RoR3's weights before training starts results in a smaller distance to the final weights achieved when the model converges (Figure 3.4(a)), and (2) during the training, this informed weight initialization enables smaller gradient variance (i.e., more consistent gradient directions) towards the basin of loss curvature (Figure 3.4(b))), thus requiring fewer iterations to convergence in theory [40, 202].

## 3.4 ModelKeeper Overview

ModelKeeper is an automated training warmup system for various ML tasks that accelerates DNN training by warm-starting models with weights from already-trained models.

**Design Space** Large training clusters are shared between users with varying expertise, and they can train a large number of jobs with different model architectures. Consequently, ModelKeeper must minimize the information needed and overhead incurred for each training model (i.e., *query model*), while offering users the flexibility in their request (e.g., using ImageNet model zoo to warm start models on other image datasets). In fact, determining which dataset (model zoo) as the source to transfer is as yet an open problem in the transfer learning community [272, 178, 288]. ModelKeeper is complementary to and benefits existing ML efforts as it automates training warmup (e.g., searching, transforming, and contributing a trained parent model's weights) for a given model zoo, instead of making the developer keep tracking all models and handcraft which model to repurpose [20]. We empirically show that ModelKeeper can benefit the model training across datasets too (§3.7.4).

For a given model zoo, the effectiveness of transforming parent model weights relies on two key aspects: (i) *Model similarity*: it dictates the similarity of two model architectures, including the weights shape and operation type of a tensor; and (ii) *Parent model accuracy*: it determines the

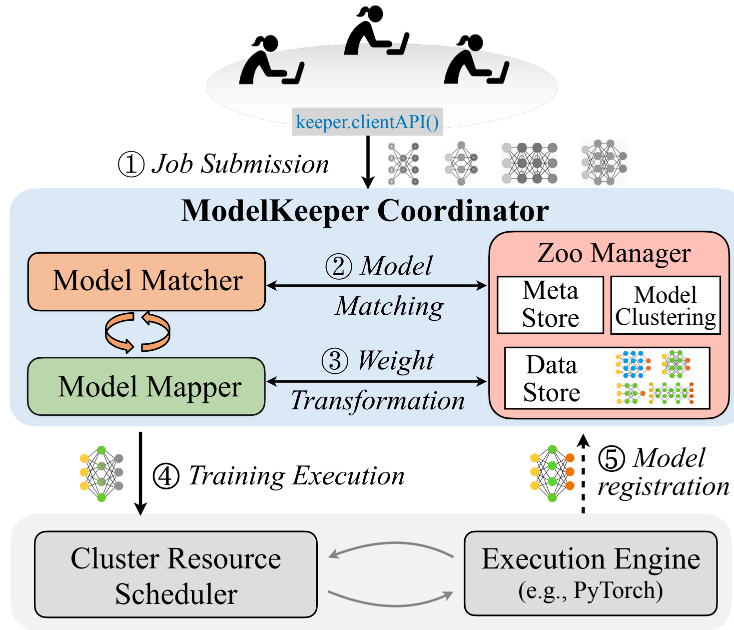



Figure 3.5: ModelKeeper architecture. It can run as a cluster-wide service to serve different users and/or frameworks.

value of transformation. Having architectural similarity is the prerequisite to transforming more weights information of a parent model, while better parent model accuracy implies a potentially better warm start after transformation.

As such, ModelKeeper should repurpose a parent model with large similarity and better accuracy. We provide the theoretical analysis to support why ModelKeeper can benefit model convergence following this principle in Appendix B.1.

**System Components** ModelKeeper is a complementary system to existing ML training (Figure 6.3), and has integrations with various frameworks (e.g., Microsoft NNI [17] and Ray [192]). It consists of the remote coordinator, which serves user query models before their training executes, and the client agent that allows users to submit model warmup requests. ModelKeeper coordinator employs three key components to warm up models by transforming a trained model's weights:

- *Model Matcher*: to identify architecturally similar models in the zoo of trained models;
- *Model Mapper*: to select a zoo model with good architectural similarity and accuracy as the parent model, and transforms the parent model weights to the query model;
- *Zoo Manager*: to adaptively manage zoo models that can be submitted from users to support transformation at scale.

Figure 3.6 reports the example interface on the client agent, where the user benefits from ModelKeeper with a few lines of code in training (Coordinator interfaces are in Section 3.6).

```
1  from modelkeeper import ModelKeeperClient
2
3  def training_with_keeper(model, dataset):
4    # Create client session to keeper coordinator
5    keeper_client = ModelKeeperClient(coordinator_ip)
6    warmed_model, meta = keeper_client.query_for_model(
7                model, meta={'data': 'Flowers102',
8                'task':'classification', 'tags': None})
9
10   acc = train(warmed_model, dataset) # Training starts
11
12   # Register model to ModelKeeper when training ends
13   keeper_client.register_model(warmed_model,
14           meta={'data': 'Flowers102', 'accuracy': acc,
15           'task': 'classification', 'tags': None})
16   keeper_client.stop()
```

Figure 3.6: Code snippet of ModelKeeper client service APIs.

**Training Lifecycle** When the developer creates a new training job, ① she first initiates a client connection to the remote ModelKeeper coordinator, and then issues a query with the specified job meta information. ModelKeeper client agent will automatically extract the model information needed (e.g., model computation graph) and issue a request (mostly size < 1 MB) to the coordinator. ② Upon receiving the request, Matcher consults its metadata store, identifies zoo models that the user can access and that meet the specified tag (e.g., name of the preferred parent models), and measures their architectural similarity to the query model. ③ Mapper selects a parent model with a large architectural similarity and good accuracy out of these zoo models. Thereafter, it loads the model weights of this parent model from the data store, and transforms the model weights, based on pairwise tensor mapping from the Matcher, to the query model. Note that this transformation only updates tensor weight values, while others (e.g., model architecture) remain the same. ④ The coordinator responds to the developer with warmup model weights, and the rest of the training remains as usual. ⑤ When the training completes, ModelKeeper can automatically register the trained model to the Zoo Manager to benefit future jobs.

### 3.5 ModelKeeper Design

In large shared clusters, models are often submitted by various developers and/or frameworks with diverse architectures at different points in time. The large variety in model architectures and accuracy characteristics leads to novel system challenges in automating weight transformation from a parent model with high similarity and better accuracy:

- Having similar model architectures is the prerequisite to transforming weights across architectures. How to identify more architecturally similar zoo models (§3.5.1)?
- As the similarity and accuracy of many potential parent models can come at odds, which one to pick and then how to transform its weights to the query model even in the presence of

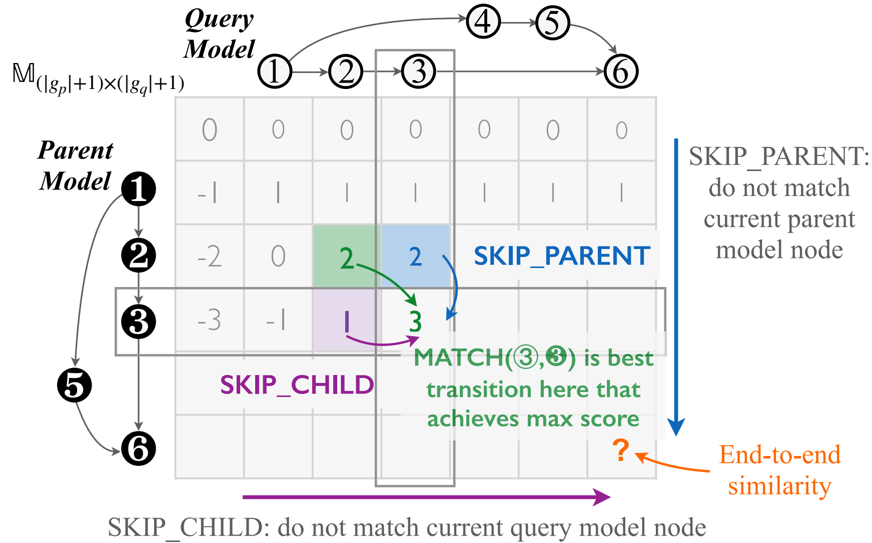



Figure 3.7: ModelKeeper relies on dynamic programming-like heuristics to measure graph-level model architectural similarity.

non-identical architectures (§3.5.2)?

- New training jobs and trained models can join the cluster on the fly. How to serve user warmup requests at scale for high throughput clusters in the wild (§3.5.3)?

### 3.5.1 Matcher: Identify Similar Models

The similarity between two model architectures determines the number of tensor weights that we can transform. Hence, we need to identify the graph-level architectural similarity of each parent and query model pair (Figure 3.7) and their pairwise tensor mappings. It is tempting to model it as a classic NP-hard graph edit distance (GED) problem [51] by treating tensors as nodes and data flows as edges with the goal to morph one graph to the other with minimum edits (e.g., add/delete a node). However, model matching encounters new challenges: (i) *Prefix Preference*: we prefer to match the prefix over the suffix of model graphs. Because prefix tensors are more transferable since they capture general input features (e.g., image color blobs) [272, 202]. Moreover, model weights are trained systematically over tensors, so any edit on prefixes can result in information loss to subsequent tensors [89]; (ii) *Partial Matching*: we can partially transform the weights of a smaller dimensional tensor to a wider one to match more tensors, or postpone its matching to preserve its exact weights information; and (iii) *Scalability*: as each model can consist of thousands of nodes across branches, capturing the similarity to thousands of zoo models is challenging.

ModelKeeper Matcher measures the graph-level similarity of models, in terms of the total number of transformable weights after mapping tensor pairs from the parent to the query model. It uses the widely-used ONNX tool [21] to extract the computation graph. ONNX supports various

model formats (e.g., Tensorflow and PyTorch), which allows us to perform the cross-framework transformation.

**Structure-Aware Pairwise Model Matching** We introduce a dynamic programming-based heuristic to measure the end-to-end similarity (i.e., number of weights to transform) of two models. It relies on a similarity table $\mathbb{M}_{(|g_p|+1)\times(|g_q|+1)}(i,j)$ to record the best similarity after matching the prefixes of the parent and the query model. Here, $|g_p|$ and $|g_q|$ respectively denote the number of tensors of the parent model and the query model. Then, it enumerates plausible matching operations from previous states (e.g., $\mathbb{M}(i-1,j-1)$), and takes the operation that can acquire the maximum similarity to enter the next state (i.e., $\mathbb{M}(i,j)$).

Figure 3.7 shows the execution of our structure-aware matching algorithm. It traverses the similarity table in the topological order of graph tensors. This allows us to embed graph-level information while prioritizing the matching of prefixes. To advance to the current tensor pair ($i$, $j$), it enumerates three plausible operations:

1. *MATCH*: transform weights of $i$'s parent to $j$'s parents.
2. *SKIP_PARENT*: give up transforming tensor $i$'s parent;
3. *SKIP_CHILD*: give up transforming to tensor $j$'s parent;

Then, it updates the table to obtain the maximum similarity after each step based on previous states as follows:

$$\mathbb{M}(i,j) = \max_{k\in parent(i)} \begin{cases} \mathbb{M}(k, j_{parent}) + MATCH(k, j_{parent}) & (3.1) \\ \mathbb{M}(k,j) + SKIP_PARENT & (3.2) \\ \mathbb{M}(i, j_{parent}) + SKIP_CHILD & (3.3) \end{cases}$$

To get the overall transformable weights, we can reward each operation based on the number of tensor weights transformed. When tensor $i$ and $j$ belong to the same operator (e.g., convolution), the fraction of transformed weights along each weights dimension in *MATCH* operation (3.1) is:

$$MATCH(i,j) = \frac{\prod_{dim=1} \min(dim(i), dim(j))}{\prod_{dim=1} \max(dim(i), dim(j))} \quad (\in [0,1]) \tag{3.4}$$

Otherwise, we assign *MATCH(i, j)* to -1, as this transformation is useless and even loses the weights of that parent model tensor. Similarly, SKIP_PARENT is set to -1 as it loses the parent model tensor, and SKIP_CHILD is 0, since it does not transform the parent model tensor.

Capturing the graph-level similarity is more challenging when tensor $j$ of the query model is the intersection of multiple upstream branches. Because different upstream branches to $j$ may follow the same branch of the parent model during their matching, leading to repetitive (conflicting)

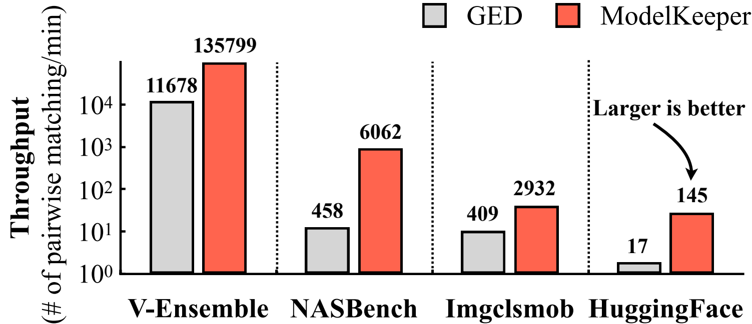



Figure 3.8: Keeper is order-of-magnitude more scalable than existing GED. V-Ensemble is a model zoo for ensemble training (§3.7.1).

matching. As shown in Figure 3.7, when we reach (❻, ⑥), branch (②→③) and (④→⑤) may both be matched to (❷→❸) that maximizes their own similarity. To avoid conflicting matching, the similarity to $j$ is the sum of upstream branches ($\mathbb{M}(i,j) = \sum_{k \in parent(j)} \mathbb{M}(i,k)$), and we greedily adopt the matching of a branch to tensor $j$, whose trajectory achieves the largest similarity (i.e., match ②→③ to ❷→❸), to maximize their sum. Meanwhile, we discard the trajectory of other branches where conflict exists. As such, branch ④→⑤ takes the inferior match (❺), where ④ is skipped.

The last entry of the table, i.e., $\mathbb{M}(|g_p|, |g_q|)$, gives the end-to-end similarity. Note that we can learn the pairwise tensor mappings by backtracking operations taken to reach $\mathbb{M}(|g_p|, |g_q|)$ over the table in linear time. For a specific model, our heuristic will naturally treat the model itself as the most similar model, because matching will always take operation $MATCH(i, i)$ in each step to maximize the similarity.

Figure 3.8 reports that our pairwise matching can match thousands of model pairs in a second, and achieves higher throughput than the state-of-the-art GED solution [218] in this model matching scenario. More importantly, our empirical results report that our structure-aware matching achieves better training warmup than the GED solution (§3.7.3).

### 3.5.2 Mapper: Transform Maximal Parent Information

The effectiveness of weight transformation is determined by the similarity (transfer more weights) and accuracy (transfer better weights) of parent models. Unfortunately, it is impractical to pick the optimal parent model, since the performance of transformation can only be known after training each derived warmup model to converge. Worse, the variety of model similarity and performance leads to the tussle in selecting the parent model. As shown in Figure 3.9, while some models (e.g., ResNet34) possess high accuracy, their low similarity to ResNet101 can cap the number of weights that can transform. Next, we introduce Mapper to exploit the sweet spot of both

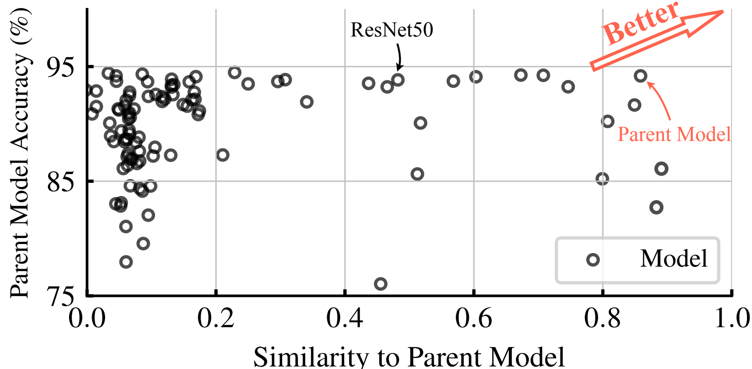



Figure 3.9: Models vary in accuracy and architecture (Imgclsmob zoo). We measure their similarity w.r.t.ResNet101, and prefer to transform a parent model with better similarity and accuracy.

aspects, and then to transform maximal parent model weights in the presence of partial matching.

As shown in Algorithm 3.1, Mapper relies on Matcher to identify similar models (Line 2). As having a good similarity is the prerequisite for transformation, we need to first ensure picking similar models. To this end, Mapper takes the popular bucketing strategy to allocate models into $B$ buckets in terms of their similarity (Line 14). Taking Figure 3.9 as an example, with $B = 10$ by default, *bucket* 10 will accommodate models with similarities between 0.9 and 1.0, so models in the same bucket have comparable similarities. Then, Mapper traverses from the last bucket (*bucket 10*) to the first until reaching the first one with nonempty models (*bucket 9*), from which it selects the model with the best performance as the parent model (Line 15). As such, the parent model approaches the boundary of better model similarity and accuracy. Later, Mapper performs structure-aware weight transformation to initialize the query model weights (Line 18).

**Information-Preserving Weight Transformation** To maximize the end-to-end number of weights to transform, Matcher allows partial matching: it may map a small tensor of the parent to a wider one of the query model, or skip the mapping of some tensors in the parent or the query model. Here, the straw-man solution (e.g., in Retiarii [286]), which transfers the weights of parent model tensors if and only if two tensors are identical, can be suboptimal (§3.7.3), since losing the parent model tensor can make the transfer of its subsequent tensors useless.

To preserve maximal parent model information under partial mappings, Mapper employs a width operator and a depth operator, which extend the well-known ML technique for function-preserving model transformation (e.g., Net2Net [66] and Network Morphism [254]). But unlike existing model transformation techniques [145], which are limited to *expand* the depth and width of a pre-determined model, or complicated transfer learning (e.g., knowledge distillation [124]) that requires additional computation (e.g., pre-training) and/or intrusive implementation, our operators transform the parent model weights into the query model with little overhead.

**Input:** Query model $q$, Model Zoo $M$
**Output:** Warmup Model Weights

```
1:  NumOfBuckets B = 10                                            ▷ Model similarity ∈ [0, 1]
2:  Function GetModelSim (Query q, Models M)
      /* Structure-aware matching for model similarity.                                 */
3:    topo_query_tensors = SortByTopo(q)
4:    model_similarity = { }
5:    for model m ∈ M do
6:      similarity_table = zeros(|g_m|+1, |g_q|+1)
7:      for tensor i ∈ CachedModelTopo(m) do
8:        for tensor j ∈ topo_query_tensors do
            /* Enumerate and merge intersection.                                        */
9:          similarity_table[i][j] = Equation (3.1-3.3)
10:     model_similarity[m] = similarity_table[|g_m|][|g_q|]
11:   return model_similarity
12: Function QueryForModel (Query q, Model Zoo M)
      /* Bucket models in terms of similarities. Pick the model in the top-similar bucket with the best
         performance.                                                                   */
13:   model_similarity = GetModelSim(q, M)
14:   top_similar_bucket = BucketBySimilarity(model_similarity, B).first
15:   for model ∈ top_similar_bucket do
16:     if model.perf > best_parent.perf then
17:       best_parent = model
      /* Perform width and depth weight transformation                                  */
18:   warmup_weights = TransWeight(best_parent, q)
19:   return warmup_weights
```

**Algorithm 3.1:** Select the parent model to transform.

Our graph-level transformation proceeds in the topological order of tensors. Mapper handles the expanding case similarly to today's function-preserving transformation (Figure 3.10): (i) to transform a parent model tensor into a wider query model tensor, the width operator copies the parent model weights to its mapping tensor of the query model, and pads the rest of the columns via weighted replication from other columns; and (ii) when the mapping requires inserting a new tensor into the parent model (i.e., SKIP_CHILD), the depth operator will initialize the weights of this mapping tensor to be an identity tensor. i.e., this tensor will directly pass the output of its parent tensors to the child tensors, in order to keep the same parent model's output. Readers can refer to

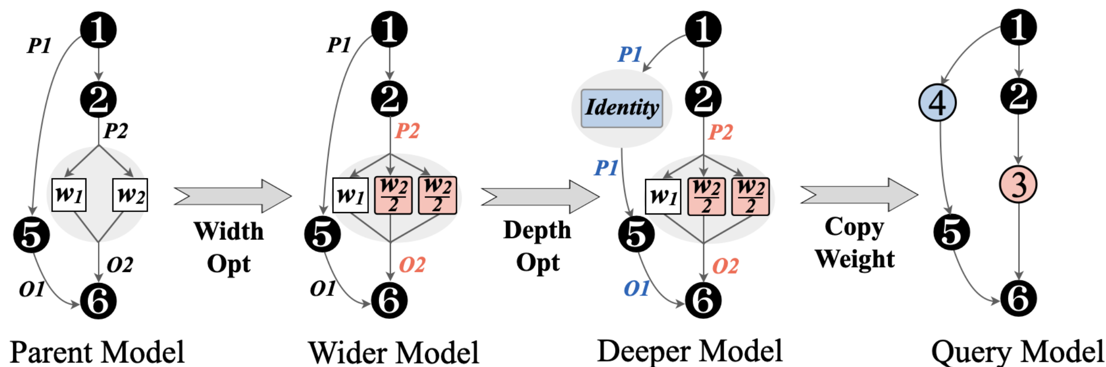



Figure 3.10: Width and depth operators to transform the parent model.

Net2Net [66] for more details. We note that both expanding operations, in theory, can preserve the parent model information (i.e., with the same tensor output) for many tensor operators (e.g., the wide-used full connection and convolution layers).

The pruning case, however, cannot preserve full parent model information, because we lose some tensor weights of the parent model in transformation. Our solution is inspired by today's ML model pruning criteria [114]. Specifically, when we need to fit wider tensor weights (i.e., with larger array dimensions) to a smaller dimensional tensor of the query model, the width operator will progressively pick and copy the largest weight values of the parent model tensor to the mapping tensor of the query model. This is because, intuitively and empirically, larger magnitude values often have more impact on the model output [48]. From the depth perspective, when we skip transforming (i.e., SKIP_PARENT) a parent model tensor, the depth operator will add noise to the weight values of that tensor's neighbors. It disturbs the affinity of trained parent model weights so that neighboring tensors can still keep most information while being able to learn new weights [200].

Our transformation can be applied to various models for informed weight initialization. Thereafter, training proceeds as normal, and the warmup model will gradually converge on the weight values that fit its architecture the best. As a generic system, ModelKeeper can accommodate other transformation techniques too as they become available. We provide a theoretical analysis of our transformation in Appendix B.1.2, and empirically show performance improvements using our transformation over its counterparts (§3.7.3).

### 3.5.3 Zoo Manager: Transform Effectively At Scale

In reality, cluster users register their trained models to the model zoo on the fly, leading to scalability and performance challenges. First, while gathering more models increases the opportunity to transform better parents, the ever-growing number of models (e.g., > 70K models in the HuggingFace model hub of all tasks [14]) and model size (e.g., NLP models can be tens of

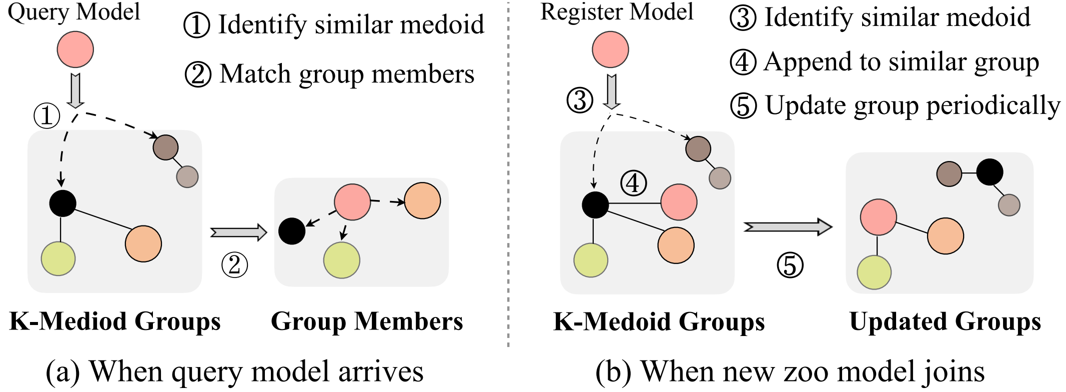



Figure 3.11: Matcher clusters models into groups to reduce the search space, and then performs model matching within groups.

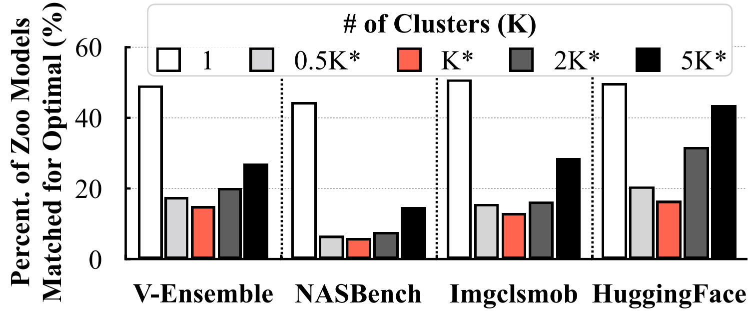



Figure 3.12: ModelKeeper can find the optimal number of clusters $K^*$ in hierarchical matching, and can identify the most similar models with fewer zoo models (e.g., 5% in NASBench) needed to explore.

GBs [58]) can lead to large matching overhead and storage costs. Moreover, models registering to the zoo may have low accuracy (e.g., due to insufficient training), which can harm the effectiveness of weight transformation. As such, ModelKeeper employs a Zoo Manager to support effective transformation at scale under dynamics.

**Two-Stage Hierarchical Model Matching** Despite being able to match thousands of lightweight CV models every minute (Figure 3.8), our pairwise matching heuristic can still be insufficient for model zoos with tens of thousands of models or complicated model architectures (e.g., NLP models). For example, to serve a query model using the HuggingFace model zoo for English text generation (2.5K models), performing pairwise matching on these zoo models can take $\sim$17 minutes, namely, 2.5K models over the throughput (145 matching/minute). This long search time is further exacerbated in today's large cluster with sub-minute job arrivals [139], eventually hurting the user experience.

To ensure an *interactive* service, Zoo Manager adaptively clusters zoo models into a well-defined

number of groups, whereby Matcher can perform *two-stage* matching to reduce the number of matching pairs needed to identify more similar models. Intuitively, models with similar architectures would have comparable model similarity to the same query model, so we may be able to cluster zoo models into multiple groups, and then perform pairwise matching on the group members of top similar model groups. However, it is non-trivial to decide what features to use for clustering models, and how many groups are needed. Clustering too few groups does not scale down the problem enough, while too many can lead to a large overhead in identifying which group to prioritize.

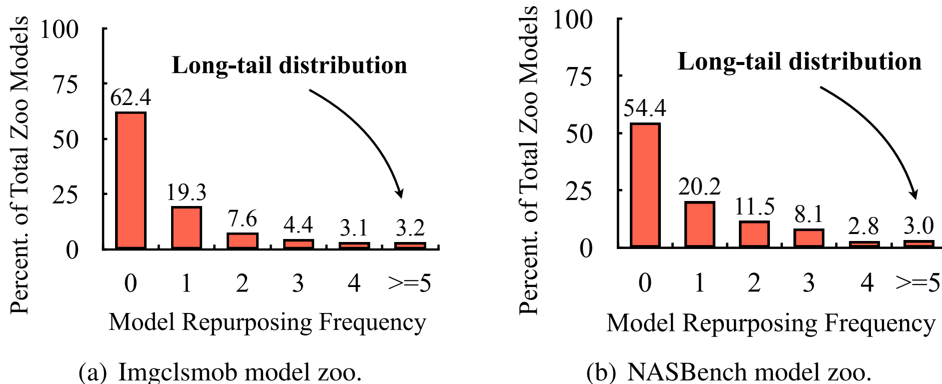
We deploy K-medoids clustering [201] to combine pairwise model matching and clustering to find a sweet spot. K-medoids can directly take the distance of two points as input to minimize the distance between data points and their cluster center. Here, models can be taken as different points, and the distance is the reciprocal of their similarity. Compared to other clustering methods (e.g., K-means), K-medoids circumvents the need for embedding complicated model graphs, and it is more compatible with pairwise model matching.

As shown in Figure 3.11, when a query model arrives, Matcher identifies its similarity to each group medoid, and then conducts pairwise matching on the members of top similar groups. Similarly, when a new model registers, Matcher measures its similarity to group medoids, and assigns this new model to the group whose medoid is the most similar. This enables interactive queries to the latest models. Later, Matcher periodically triggers K-medoids to update the clustering.

To select the most similar models for each query model, Matcher identifies the best group medoid $i$ by performing $K$ pairwise matching, followed by $K_i$ runs to match the members of group $i$. Assuming a zoo of $M$ models ($M = \sum_i^k K_i$), to minimize the average matching runs on each group (i.e., $\min\big((K + K_i)/K\big)$), we can get the optimal number of groups $K^* = \sqrt{M}$. Figure 3.12 reports that, compared to the non-clustering design (i.e., $K = 1$), this two-stage design requires matching only 5%-16% of all zoo models to identify the most similar models, thus reducing the query hang time (§3.7.3).

**Capping Zoo Size** Hosting all zoo models can consume noticeable storage space. For example, the HuggingFace model zoo takes tens of TBs of storage [14]. In fact and understandably, we notice that a small portion of zoo models are more frequently repurposed than others (Figure 3.13). This is because certain models contain more similar blocks to other models (e.g., ResNet50 is more likely to be used to warm up other large ResNet models than ResNet18).

To harvest more warmup opportunities subject to the zoo capacity limit, we can formulate it as a knapsack packing problem, where each item (model) is associated with a weight (model size) and a value (repurposing frequency as the parent model), and our goal is to maximize the total value achieved. Namely, warm up as many jobs as possible (aka maximum total repurposing frequency). As such, solving this packing problem enables us to identify which item (model) to keep in the




(a) Imgclsmob model zoo. (b) NASBench model zoo.

Figure 3.13: A few zoo models are more frequently repurposed as the parent by Keeper. Numbers are from our evaluations (§3.7.2).

knapsack (model zoo). But on the other hand, models that are popular to train can change over time. For example, users incline to train more recent and/or advanced models. To account for the temporal variation in the repurposing frequency of each zoo model, we take the moving average of model values (e.g., decaying their repurposing frequency by 0.9 every day), and trigger the packing solver upon reaching the storage limit. We show that ModelKeeper can perform well even under severe storage limits (§3.7.4).

**Avoiding Low-Accuracy Models** Low accuracy models registering to the zoo (e.g., due to user error) not only wastes storage but can harm the transformation, so we need to ensure the zoo uses models with decent accuracy. To this end, other than selecting the model with better accuracy as the parent using the bucketing design at query time, Zoo Manager evicts zoo models with outlier accuracy at runtime. By default, we take the popular Z-score criteria (i.e., model accuracy below the mean by more than two standard deviations) to identify outliers [221]. Moreover, for the same model architecture, it only keeps the model with the best accuracy. We show that ModelKeeper can accelerate training even in the presence of low accuracy models in unfavorable environments (§3.7.4).

**Complexity Analysis** The complexity of pairwise model matching is $O(|g_p| \times |g_q|)$,[3] and that of model clustering is $O(M^{2.5})$ for the zoo with $M$ models. Mapper takes linear time to select and transform the parent model. The magnitude of these factors is mostly within O(1K) (§3.7.1). Our evaluations show that ModelKeeper incurs negligible overhead (§3.7.3).

[3]We omit the complexity in enumerating tensor parents (i.e., $k \in parent(i)$), since the node degree is orders of magnitude smaller than $|g_p|$.

## 3.6 Implementation

We have implemented a system prototype of ModelKeeper, with around 2K lines of Python code as the frontend library and 1K lines of C++ code as the backend. Our implementation provides user-friendly APIs and supports many popular ML frameworks, such as Microsoft NNI [17], AutoKeras [145], Ray [192], and MLflow [18], with few-lines-of-code plugins.

**ModelKeeper Components** ModelKeeper coordinator supports distributed deployment across machines. Each coordinator controller processes a single scheduling thread to poll client requests from its queue, and reserves a thread pool for Matcher. Matcher performs pairwise model matching in parallel for each query model, and then Mapper creates a worker thread to transform parent model weights using *numpy* format. Zoo Manager updates the model clustering every 5 minutes, and uses *ortools* library to solve the knapsack problem. The client agent communicates with the coordinator via TCP connections.

**Fault Tolerance** ModelKeeper uses *Redis* in the coordinator to store the metadata and model weights in a fault-tolerant manner, and this metadata is cached in memory with small footprints. Changes to the model zoo (e.g., registering new models) follow the write-ahead transaction to the storage. At runtime, the coordinator runs a daemon process to monitor the liveness of the service, which will create a new service process if the existing one crashes. The new process then fetches the latest checkpoint from *Redis* to catch up.

**Interfaces** We pack interfaces into a Python library. The cluster manager can initiate the coordinator in three lines:

```
from modelkeeper import ModelKeeperCoordinator
keeper_service = ModelKeeperCoordinator(config)
keeper_service.start()
```

Users can initiate the client agent in a few lines (Figure 3.6).

## 3.7 Evaluation

We evaluate the effectiveness of ModelKeeper on three mainstream frameworks for exploratory and general DNN training, using five large-scale CV and NLP model zoos across thousands of models. We summarize the results as follows:

- ModelKeeper saves 23%-77% total amount of training execution needed (i.e., 1.3×-4.3× faster training) than the state-of-the-art without accuracy drop of models (§3.7.2).

| Category | Task | Workload | # of Models | Dataset | Avg. Time Improvement | Avg. Acc. Difference |
|---|---|---|---|---|---|---|
| Exploratory Training | Grid Search NAS | NASBench [84] | 1,000 | CIFAR-100 | 2.9× | 0.39% |
| | Evolution NAS | | | | 2.4× | 0.38% |
| | Bayesian NAS [145] | AutoKeras Zoo [145] | 500 | | 4.3× | 0.31% |
| General Training | Image Classification | Imgclsmob [26] | 389 | Flowers102 [203] | 2.8× | 0.23% |
| | | Imgclsmob-Small | 179 | CIFAR-10 | 2.1× | 0.02% |
| | | | | CIFAR-100 | 1.6× | 0.18% |
| | | | | ImageNet32 [72] | 1.3× | 0.03% |
| | Ensemble Training | V-Ensemble [253] | 104 | CIFAR-100 | 1.7× | 0.08% |
| | Language Modeling | HuggingFace [14] | 69 | WikiText-103 [189] | 1.8× | -0.13 ppl |

Table 3.1: Summary of improvements. ModelKeeper improves training execution time without accuracy drop, by reducing the amount of training needed (i.e., GPU Saving). The accuracy difference is defined by Acc.(Keeper) - Acc.(Baseline), and smaller perplexity (ppl) is better.

- ModelKeeper outperforms its counterparts by exploiting the parent model with high similarity and better accuracy using different design components (§3.7.3).
- ModelKeeper improves performance over a wide range of parameters and practical cluster setups in the wild (§3.7.4).

### 3.7.1 Methodology

**Cluster setup.** We evaluate ModelKeeper on an 80-node cluster (40 GPU nodes and 40 CPU nodes). Each GPU node has a Tesla P100 GPU with 16 GB GPU memory and 16-core CPUs. Since most HuggingFace NLP models exceed our GPU memory capacity, we resort to CPU nodes. Each node has 32-core CPUs and 384 GB of memory. ModelKeeper coordinator runs on a 32-CPU server with 10 Gbps bandwidth.

**Workloads.** We evaluate ModelKeeper using five widely-used CV/NLP model zoos and realistic workloads (Table 6.2):

- *NASBench* [84]: an image classification model zoo with thousands of lightweight models for NAS task.
- *AutoKeras Zoo* [145]: a CNN model zoo generated by AutoKeras during the Bayesian NAS searching.
- *Imgclsmob* [26]: a popular zoo of state-of-the-art CV models (e.g., DenseNet [131]). Most models are heavyweight.
- *V-Ensemble* [253]: a benchmarking workload for ensemble training, which has hundreds of variants of VGG models.

- *HuggingFace* [14]: a collection of advanced HuggingFace NLP models (e.g., Bert [82]) for next word prediction.

We train Imgclsmob-Small models on the CIFAR dataset and the ImageNet32 dataset for 32×32 small image inputs, Imgclsmob models on Flowers102 dataset for 224×224 large images, and HuggingFace models on the large WikiText dataset. ImageNet32 is a downsampled 120-category ImageNet dataset (e.g., smaller input size) for efficient computation.

To emulate practical cluster setups, NAS models are generated by the searching algorithm on the fly, and training jobs are submitted following the arrival of Microsoft Trace [139]. The same workload does not contain identical model architectures. ModelKeeper model zoo starts empty for each workload, and jobs contribute (upload) their trained models to the zoo as they complete over time.

**Parameters.** We follow the default setting specified in each model zoo: (1) *CV models*: the SGD optimizer with a minibatch size 64 and an initial learning rate 0.01; and (2) *NLP models*: the AdamW optimizer with a minibatch size 32 and an initial learning rate 8e-5. We use the ReduceLROnPlateau scheduler to decay the learning rate by 0.5 once the training loss stagnates.

**Baselines.** We compare ModelKeeper to the following:

- *Retiarii [286]*: Microsoft's training framework that relies on the lineage of graph mutation to warm up NAS models.
- *AutoKeras [145]*: An advanced AutoML system based on Keras that applies lineage-based warmup for NAS models.
- *MotherNet [253]*: An ad-hoc ensemble training algorithm that trains a model subnet, which introduces intrusive overhead and implementation, to warm start models.

Existing efforts limit to individual NAS/ensemble jobs, while ModelKeeper can support various tasks across jobs and users.

**Metrics.** We care about the *training execution time* needed to train to converge and the *model convergence accuracy*.

We run with five realistic Microsoft Traces [212], and report the average over 5 runs.

### 3.7.2 End-to-End Performance

In this section, we evaluate how ModelKeeper (Keeper) is complementary to and benefits today's ML frameworks. Here, we run the NAS task using Microsoft NNI (with Retiarii backend [286]) and

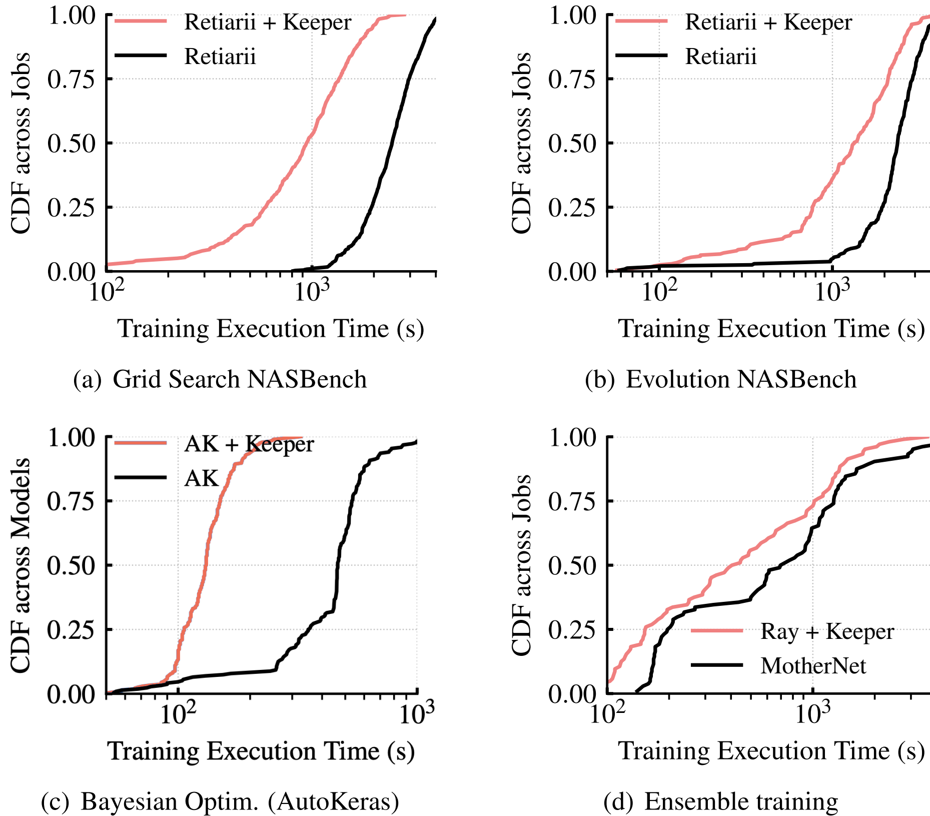



Figure 3.14: ModelKeeper outperforms existing warmup training.

AutoKeras, and other training tasks on Ray [192]. Table 6.2 summarizes the average improvement on each training workload after applying ModelKeeper.

**ModelKeeper outperforms existing warmup solutions.** ModelKeeper outperforms existing training warmup solutions in Retiarii, AutoKeras, and MotherNet by 1.7×-4.3× (Figure 3.14), by saving 43.1%-76.7% total amount of training needed. Their inefficiency is due to two primary reasons:

(i) Suboptimal parent model selection: Retiarii and AutoKeras track the lineage of graph mutation and treat the base model in evolution as the parent model. However, as multiple layers can be modified on the base model in searching new models, such rigid parent selection can miss better parent models out of other explored NAS models. Similarly, MotherNet not only requires additional training of the model subnet, but can not repurpose better-trained models on the fly.

(ii) Insufficient weight transformation: Their design, which simply copies the weights from the parent model when two tensors are identical, is lossier than ModelKeeper. For example, inserting randomly initialized prefix tensors can make the copy of subsequent tensors useless.

Meanwhile, they are limited to specific NAS or ensemble training tasks and cannot serve various

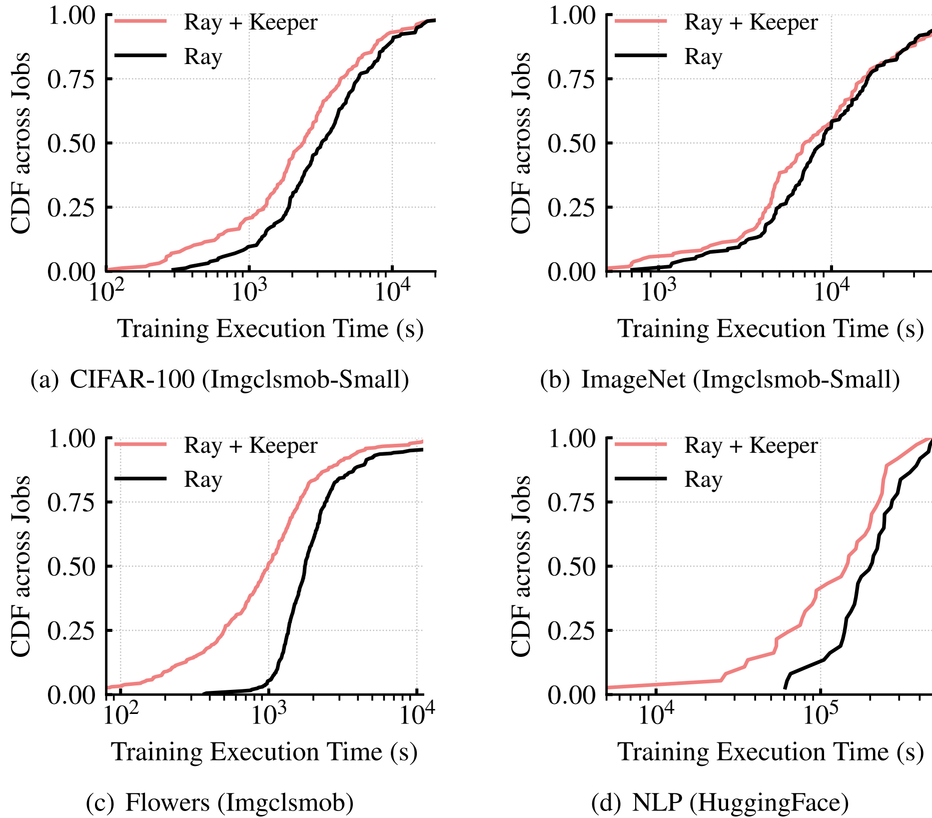



Figure 3.15: ModelKeeper improves general training tasks.

DNN training jobs on the fly in the cluster wide.

**ModelKeeper accelerates ML training for various tasks.** Figure 3.15 and Table 3.2 report the performance of individual jobs. Compared to training from scratch, we observe that: (i) ModelKeeper achieves 1.3×-4.3× faster training, saving 23%-77% training execution, across a wide range of workloads. This improvement is more pronounced in a larger zoo because of having more trained models to repurpose. (ii) Improvements on different workloads report a positive correlation with the prevalence of model similarity in that model zoo. Here, ModelKeeper achieves larger improvement on NASBench, which is consistent with the fact that this model zoo owns higher inter-model similarities (Figure 3.2). (iii) Although ModelKeeper starts from an empty zoo and jobs arrive on the fly, we can still save the training execution for 70%-95% individual jobs (Table 3.2). We note that the 25th percentile improvement of small-scale model zoos (e.g., Imgclsmob) is inferior to others. This is because not all training models are warmed up due to the cold start of this online setting, and the fact that ModelKeeper will not warm start the model that does not have a similar parent (similarity > 0).

| Workload | Time Improvement | | | Acc.(Keeper) - Acc.(Baseline) | | |
|---|---|---|---|---|---|---|
| | 25th | 50th | 75th | 25th | 50th | 75th |
| NAS-Grid | 1.5× | 2.0× | 3.1× | 0.01% | 0.25% | 0.42% |
| NAS-Evol | 1.2× | 1.6× | 3.0× | 0.03% | 0.19% | 0.48% |
| Flowers102 | 1.2× | 2.1× | 3.3× | 0.0% | 0.16% | 0.37% |
| CIFAR-100 | 1.1× | 1.5× | 2.0× | -0.04% | 0.08% | 0.32% |
| ImageNet32 | 1.0× | 1.2× | 1.6× | -0.07% | 0.0% | 0.11% |
| V-Ensemble | 1.1× | 1.5× | 1.9× | 0.02% | 0.07% | 0.65% |
| HuggingFace | 1.2× | 1.4× | 2.1× | 0.2 ppl | -1.3 ppl | -3.87 ppl |

Table 3.2: Keeper saves training execution time of individual jobs without accuracy drop. Smaller perplexity (ppl) is better.

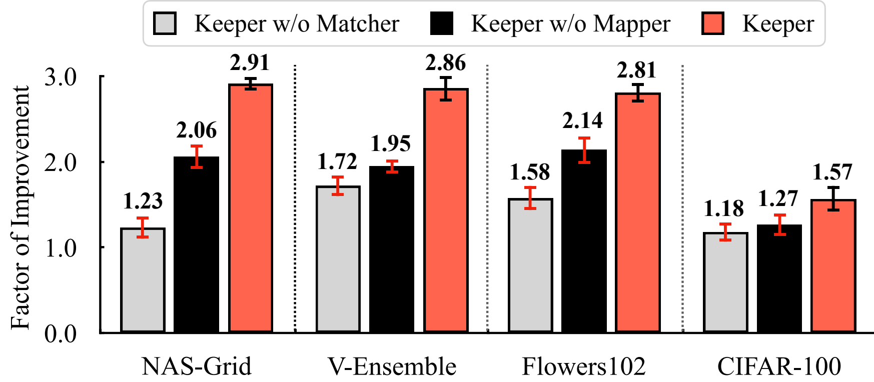



Figure 3.16: Breakdown of Keeper components.

**ModelKeeper speeds up training without accuracy drop.** Table 6.2 and Table 3.2 report that, on average, ModelKeeper can achieve similar (or even slightly better) final model accuracy. Intuitively, ModelKeeper should perform no worse than baseline accuracy, since the rest of the training (e.g., data) remains the same. However, we note that this slightly better model performance is consistent with the observations in ML network morphism [254], which interprets it as the internal regularization ability. Specifically, by transferring from well-trained models, model weights have been placed in a good position in the space, resulting in a more regularized network to reach a better basin of the loss curvature [254, 89]. In contrast, training from scratch can get stuck in local minima.

### 3.7.3 Performance Breakdown

In the rest of the evaluations, we refer to the improvement on V-Ensemble as that over training from scratch for brevity.

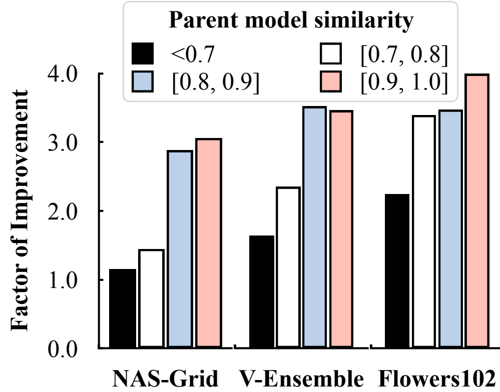



Figure 3.17: Faster training with higher model similarity.

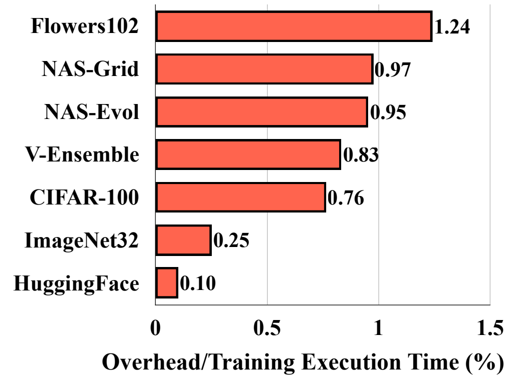



Figure 3.18: Keeper introduces negligible overhead.

**Breakdown of Components** We break down ModelKeeper by disabling Matcher and Mapper respectively: (1) *Keeper w/o Matcher*: remove our Matcher design, and instead resort to a state-of-the-art graph matching strategy [218] to select a parent model with the most pairwise tensor mappings; and (2) *Keeper w/o Mapper*: disable our Mapper design, so only transform the parent model weight if and only if two tensors are identical. Figure 3.16 reports the improvement of these variants. We notice: (i) the classic GED solution, in *Keeper w/o Matcher*, achieves suboptimal performance, since model matching prefers to match prefixes, and partial matching allows better overall similarity. (ii) transforming weights only for identical tensors, in *Keeper w/o Mapper*, is inferior to Keeper information-preserving transformation. (iii) Matcher and Mapper contribute comparable improvements.

**Breakdown of Improvement Characteristics** Figure 3.17 reports the average improvement after categorizing training models by their similarity to the parent model. We note that: (1) Keeper tends to achieve better execution saving for models with a higher parent model similarity. This again supports our parent model selection criteria that prioritize models with higher architectural similarity. (2) Improvements of different similarity regimes (e.g., [0.7, 0.8] vs. [0.8, 0.9]) are often distinct, and this becomes vague as similarity over 0.8. Because most layers have been largely warmed up, and deeper layers are too specific to the parent model to be transferable [272].

**Overhead Analysis** Figure 3.18 reports Keeper's overhead, i.e., the time taken between initiating the query and starting to train, over the training execution time. We report the average of all jobs, and notice that Keeper introduces less than 1.5% overhead (< 43 s) across all workloads.

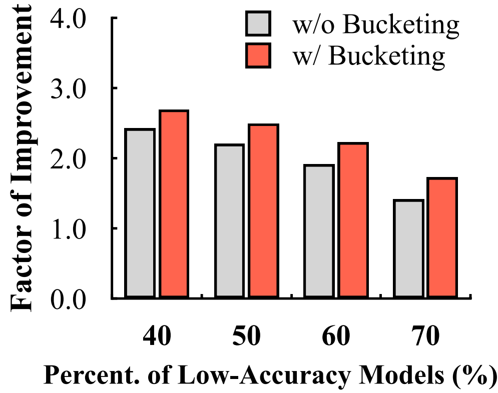



Figure 3.19: Keeper is robust in the presence of poor performance models (NAS-Grid).

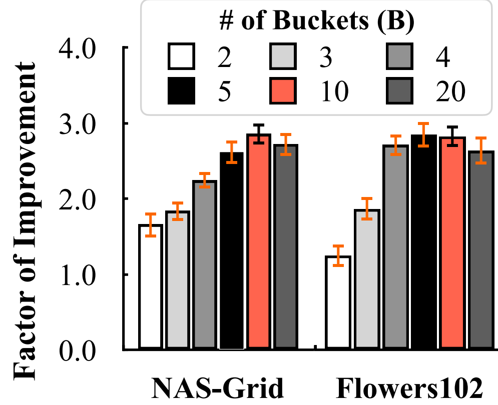



Figure 3.20: Keeper improves training execution time across the different numbers of buckets.

### 3.7.4 Sensitivity and Ablation Studies

**Impact of Low-Accuracy Models** As a cluster-wide service, ModelKeeper should be robust to unfavorable settings where the accuracy of user-registered zoo models can be low (e.g., due to insufficient training). We follow the popular early-stop design in ML [167] to simulate unfavorable setups, where model registration takes place when jobs run to at most $X$ minutes. Figure 3.19 reports the improvement of execution time across different degrees of unfavorable settings. Here, the x-axis value $40\%$ indicates $X$ is set to be the 60th percentile value in execution time distributions, so only $40\%$ zoo models are trained to converge. We observe that: (i) improvement decreases as more zoo models have low accuracy. (ii) Keeper is more robust with our bucketing design as it exploits the similarity-accuracy sweet spot.

**Impact of Bucketing** Figure 3.20 reports that ModelKeeper delivers consistent improvement across a wide range of number of buckets $B$. Meanwhile, we notice that using the most accurate parent model (i.e., $\sim B$=2) or the most architecturally similar parent (i.e., $\sim B$=20) achieves suboptimal improvement, since it respectively undervalues the model similarity and accuracy in selecting the parent model.

**Impact of Zoo Capacity** Figure 3.21 reports the average improvement under different zoo capacities. The total size (i.e., $100\%$) of model zoos in NAS-Grid, V-Ensemble, and Flowers102 are 1.6GB, 17GB, and 31 GB, respectively. We observe that: (i) as expected, the improvement is more pronounced as we allocate more storage to ModelKeeper's model zoo, but (ii) we can still achieve $\sim 2\times$ improvement under severe capacity limits (e.g., $5\%$ capacity aka < 2GB storage), since Keeper adaptively evicts suboptimal zoo models.

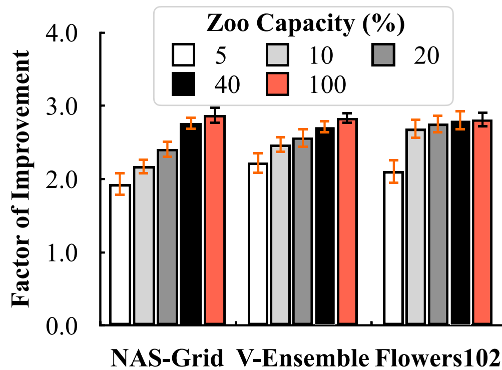



Figure 3.21: Impact of zoo capacity on execution time. Error bars report standard deviation.

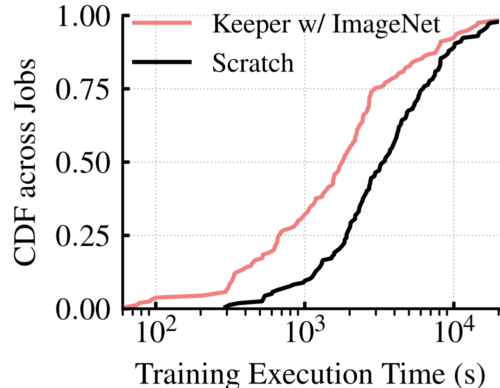



Figure 3.22: Keeper accelerates model training on CIFAR-100 using ImageNet32 model zoo.

**Cross-Dataset Training Warmup** Figure 3.22 reports that ModelKeeper can benefit DNN training across datasets. Here, we warm start the training of Imgclsmob-small models on CIFAR-100 using zoo models from the ImageNet32 workload, and notice 2.5× faster training on average. This is because front DNN layers capture general input features (e.g., color blobs of images), which are transferable to similar datasets [272]. While picking which dataset to use as the source for warmup is still an open ML problem [178, 288], ModelKeeper provides systems support for automated warmup transformation across ML tasks and datasets using the given model zoo.

## 3.8 Discussion and Future Work

**Support for Hyperparameter Tuning** ModelKeeper by default automatically searches and transforms the parent model for various training tasks. Meanwhile, the developer can specify which parent model to repurpose using the tag configuration in their request too (Figure 3.6), while enjoying the automated weights transformation. For example, we may want the same parent model for hyperparameter tuning jobs to eliminate the comparison bias and/or to ensure reproducibility. Moreover, as the training of the query model will be jump-started, it would be interesting to investigate how to adapt to better job configurations (e.g., scaling the learning rate in terms of the number of transformed layers [212, 254]) to further improve the training convergence.

**Model Sharing in the Wild** ModelKeeper repurposes a zoo of trained models to warm start the new training job. These zoo models can be maintained by the cluster provider, and/or contributed by users. For example, AWS SageMaker offers hundreds of pre-trained models for tasks like object detection and natural language processing [22]; HuggingFace Model Hub has gathered ∼70K models shared by the community [14]. The former is more managed but expensive to include

extensive models and tasks, while the latter has good extensibility but can exhibit great uncertainties (e.g., low-accuracy models). To the best of our knowledge, ModelKeeper moves the first step to *automatically* warm start the cluster-wide model training. However, further investigations on how to democratize it in the wild, such as for privacy and security concerns, are needed. To this end, one possible approach is to develop differential privacy-like solutions [37] (e.g., adding noise to the weights of the contributed models), which naturally leads to an interesting trade-off between privacy and the model quality.

## 3.9 Related Work

**Deep Learning Frameworks** Recent ML efforts have made considerable progress toward efficient inter-job scheduling [108, 212, 192, 277], intra-job computation placement [197, 163], communication optimization [210, 144], specialized execution backend [24, 67, 161], and timely inference [109]. However, they are mostly in-execution optimizations, and/or the total amount of training remains the same. Different from transforming tensors for faster computation (e.g., TASO [140] and PET [245]), ModelKeeper operates on model weights, and acts as a complementary service to accelerate cluster-wide DNN training.

**AutoML Systems** Retiarii [286] and AutoKeras [145] rely on the lineage of graph mutation to repurpose trained models, whereas they are limited to NAS tasks within individual jobs. Experiment Graph [81] identifies the reusable ML scripts and artifacts in platforms to speed up repeated executions, so it focuses on the same job execution. As recent AutoML platforms, such as AzureML [234], Amazon SageMaker [2] and MLflow [281], provide a collaborative environment to simplify ML deployments, reusing artifacts can greatly speed up repeated executions (e.g., reuse scripts [81]). ModelKeeper is the first automated training warmup system to accelerate cluster-wide DNN training jobs across users, and improves Retiarii and AutoKeras further (§3.7.2).

**Transfer Learning** Transfer learning today mostly transfers the weight of the same model [272], from one source task to another target to alleviate the need for large training data. For a given parent model, network morphism [66, 254] introduces function-preserving transformation to construct child models while preserving the parent information. MotherNet [253] further applies the network morphism to warm start model training, but is limited to ensemble training tasks. ModelKeeper tackles a more challenging scenario for various tasks in the wild, and achieves better performance (§3.7.2).

**Graph Matching** Graph matching is one of the NP-hard fundamental problems in graph analysis [237]. To speed up the matching, DAF [113] decomposes the graph into forests. Similarly,

AED [218] divides global matching into local matching, and then aggregates the local matching decisions. However, they are insufficient due to the novel properties of DNN graphs, where pair-wise matching prefers ordered alignment and allows partial weights transformation. ModelKeeper outperforms them in training speedup and throughput (§3.7.3).

## 3.10 Summary

In this chapter, we introduce ModelKeeper to enable automated warmup of DNN training jobs at the cluster scale. ModelKeeper manages a collection of already-trained models from different developers and/or frameworks. Before training a model, it selects a high-quality trained parent model and performs structure-aware transformation of parent model weights to warm up the weights of new training models. Our evaluations across thousands of CV/NLP models show that ModelKeeper achieves 1.3×-4.3× faster training completion.

# CHAPTER 4

## Minimizing Model Size via In-Training Model Pruning

So far we have considered how to leverage the large number of trained models to optimize training execution before training starts. Yet, the surging model size, which can span tens of terabytes, significantly escalates the cost of model training. Next, we target a dominant model type in production, deep learning recommendation models (DLRMs), and propose an in-training pruning system to reduce ML resource demands during training.

The remainder of this chapter is organized as follows. We begin by presenting state-of-the-art DLRMs prevalent in big companies. Section 4.2 sketches the design of AdaEmbed, aiming to support efficient in-training pruning. The motivations behind AdaEmbed, drawn from the analysis of industrial DLRMs, are explored in Section 4.3. We follow this up with AdaEmbed's design overview in Section 4.4, an in-depth look at its scalable pruning in flight in Section 4.5, and implementation in Section 4.6. We then evaluate AdaEmbed's performance in industrial settings at Meta in Section 4.7 and survey related work in Section 4.8.

### 4.1 Background

Deep learning recommendation models (DLRMs) are important to many online services, including Google advertisement display [76, 68], Netflix movie recommendations [101, 157], and Amazon e-commerce [231], and comprise up to 65% of AI cycles in Meta's datacenters [111, 88]. Unlike conventional machine learning (ML) counterparts that train models on continuous input features (e.g., color values of images), DLRM inputs consist of continuous dense features (e.g., timestamp) and categorical sparse features (e.g., video genres). Each sparse feature is often associated with an embedding table, where each instance of that feature is represented by a trainable embedding row (weight vector). In the forward and backward passes of model execution, the model reads and updates the embedding weights of accessed rows.

Because the accuracy of a DLRM typically increases with larger embeddings (e.g., by considering more feature instances), modern DLRM embedding size is ever growing (up to terabytes and billions of embeddings [283, 88]). This introduces multiple challenges. First, DLRMs often have

stringent throughput and latency requirements for (online) training and inference [262, 154], but gigantic embeddings make computation [193], communication [38, 229] and memory optimizations [290, 88] challenging. To achieve the desired model throughput, practical deployments often have to use hundreds of GPUs to hold embeddings [194]. Meanwhile, designing better embeddings (e.g., number of per-feature embedding rows and which embedding weights to retain) remains challenging because the exploration space increases with larger embeddings and requires intensive manual efforts [271, 176].

Unlike existing DLRM efforts that have primarily focused on optimizing the model's execution speed for the given embeddings – e.g., by balancing embedding sharding [194, 290], accelerating embedding retrieval [261, 229], compressing embeddings [112, 270], or elastic resource scaling [262, 289] – we explore a complementary opportunity: *Can we fundamentally reduce the size of embedding needed for the same accuracy, by dynamically optimizing the per-feature embedding during model training?* Or, equivalently, *can we improve model accuracy for the given embedding size?* This is because unlike classic ML models, the DLRM model output (accuracy) is determined by the input data (e.g., accessed instances) and their embedding weights, and the input data is typically organized chronologically during training to account for the diverse and non-stationary user preferences [294]. Therefore, the access patterns and the weights of embeddings vary across embedding rows and the training process (§4.3.2). This implies an opportunity to admit and prune embedding rows based on their heterogeneous importance to improve model accuracy.

## 4.2 Solution Outlines

In this chapter, we introduce an *automated in-training pruning system* to Adaptively optimize per-feature Embeddings (AdaEmbed) for better model accuracy. For the given embedding size, AdaEmbed scalably identifies and retains embeddings that have larger importance to model accuracy at particular times during training. As a result, not only does it reduce human effort in embedding design, but it also cuts down the embedding size, thus the computational, network, and memory resources, needed to achieve the same accuracy. AdaEmbed is complementary to and supports existing DLRM efforts with a few lines of code changes (§4.4).

Unfortunately, identifying important embeddings out of billions is non-trivial. To maximize the overall model accuracy, we should retain the embedding rows that affect model inputs more often (e.g., are frequently accessed) and that affect model outputs the most (e.g., have larger weights) (§4.5.1). However, the non-stationary data distribution during training leads to the spatiotemporal variation in the access frequency of different embeddings. e.g., new videos are posted and become popular, while some old ones lose popularity. Moreover, embedding weights change over training iterations and so does their impact. Once we prune an embedding's weights from the GPU memory,

we cannot accurately capture their importance to model accuracy as training moves on. Based on our analytical insights, embeddings with larger runtime gradients and higher access frequencies tend to accumulate larger embedding weights, and AdaEmbed prioritizes them when deciding which ones to retain. Moreover, we group features with similar feature-level characteristics (e.g., vector dimensions), and then identify important embeddings across feature groups to optimize the per-feature embedding size and which embedding to retain (§4.5.1).

Enforcing in-training pruning after identifying important embeddings is not straightforward either. Pruning for practical DLRMs can require reallocating millions of embedding rows and tens of gigabytes of embedding weights per training iteration, whereas each iteration takes only a few hundred milliseconds [38, 194]. While frequent pruning allows admitting important embeddings in a timely manner, thereby improving model accuracy, it can slow down model training by many hundred times (§4.5.2). To achieve a sweet spot between timely pruning and low overhead, AdaEmbed initiates pruning selectively when perceiving big changes in the importance distribution of all embeddings, thus reducing the number of pruning rounds needed while ensuring high accuracy. However, existing DLRM systems face difficulty in dynamically admitting and pruning embeddings at scale because they often rely on static and/or fixed-size embedding storage [36, 33, 13, 261]. AdaEmbed introduces a shim layer, Virtually Hashed Physically Indexed (VHPI) embedding, to support various embedding designs. VHPI decouples the management of embeddings from their physical weights, whereby it recycles the weight vector of embeddings to avoid intense memory allocation (§4.5.3).

We have implemented a system prototype of AdaEmbed (§4.6) and evaluated it using five industry models and months of data across hundreds of GPUs (§4.7). Our evaluations show that AdaEmbed can reduce 35-60% embedding size, implying comparable resource savings, and improve model execution speed by 11-34% without compromising model accuracy. Meanwhile, it achieves noticeable accuracy gains under the same embedding size, thus being able to reduce manual efforts by automatically finding better per-feature embeddings.

Overall, we make the following contributions in this chapter:

1. We propose an in-training pruning system, AdaEmbed, to automatically optimize DLRM embeddings.
2. We introduce embedding importance to capture important embeddings and employ VHPI embedding to enforce scalable pruning, with few changes to existing designs.
3. We evaluate AdaEmbed in various real-world settings to show its resource savings and accuracy gains.

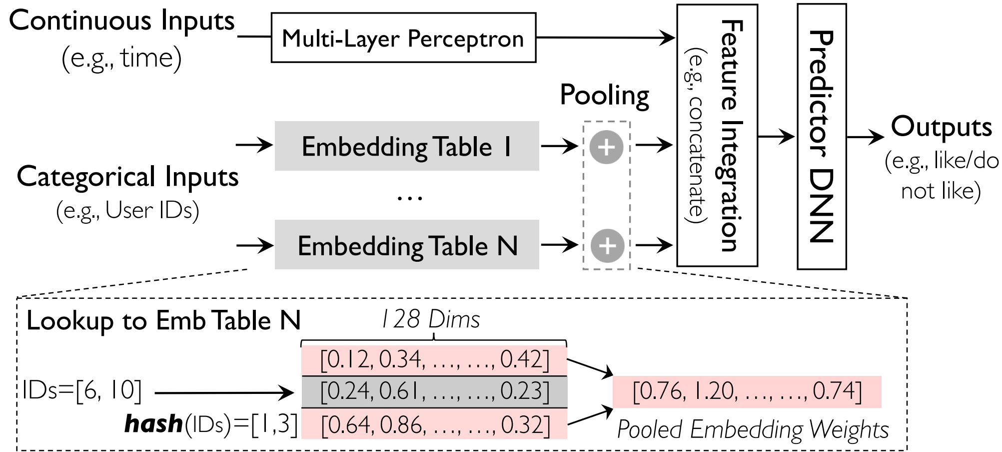



Figure 4.1: DLRM models consist of large embedding tables.

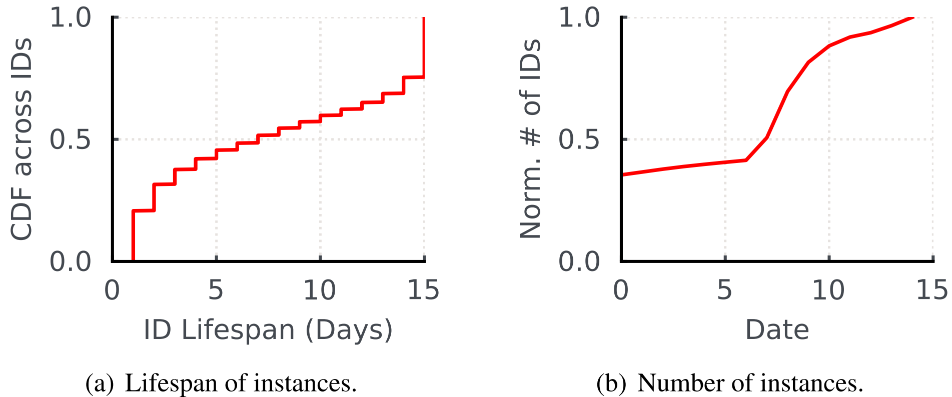
## 4.3 Motivation

We start with a quick primer on DLRMs (§4.3.1), followed by the challenges it faces and inefficiencies of the state-of-the-art based on our analysis of real-world experiments (§4.3.2). Next, we highlight the opportunities that motivate our work (§4.3.3).

### 4.3.1 Deep Learning Recommendation Models

As shown in Figure 4.1, a DLRM consists of a combination of fully connected multiple-layer perceptrons (MLPs) to capture continuous dense features (e.g., timestamp), and a set of embedding tables to map various categorical sparse features (e.g., user and video IDs) to a dense representation. DLRMs can contain up to thousands of sparse features: each feature is typically associated with an embedding table, and each table can have millions of rows [290, 101, 194]. Each embedding row is a multi-dimensional weight vector (e.g., 128 floats) corresponding to a specific feature instance (e.g., a specific user ID of feature "User IDs").

DLRMs differ from traditional computer vision (CV) and natural language processing (NLP) models in that they require training on large volumes of data organized chronologically, to keep up with the latest recommendation trends. Hence, the distribution of training data changes over the training process. In the forward pass of model computation, each input sample includes a set of embedding IDs for each table to extract the corresponding embedding weights (vectors). To reduce the computation complexity, embedding weights of a sample will be pooled per table using the element-wise *pooling* operator, which typically takes the sum or maximum along each vector dimension (Figure 4.1). The pooled embedding weights of mini-batch samples are packed together with their intermediate outputs of dense features, forming a batch input to deeper layers. In the backward pass, the weights of the accessed embeddings are updated using the gradient.



(a) Lifespan of instances.

(b) Number of instances.

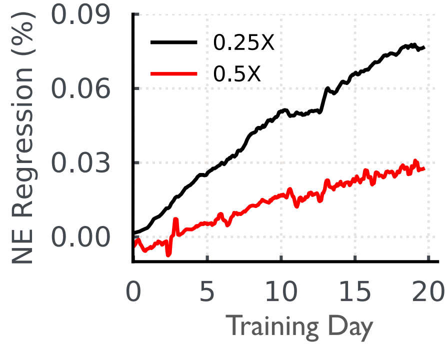
Figure 4.2: The number of sparse feature instances (IDs) increases rapidly over time, while the lifespan of instances is heterogeneous.

Due to the enormous number of sparse feature instances, their embedding weights can occupy more than 99% size of a commonly used model (up to several terabytes) [118]; so DLRMs exhibit much larger memory intensity than conventional ML models (e.g., ResNet). As such, practical DLRM deployments use a combination of model parallelism for sparse feature layers and data parallelism for MLPs. The former allocates different embedding partitions across workers to avoid replicating them, and the latter enables concurrent processing of dense feature inputs [88]. Even so, model deployments often require hundreds of GPUs to achieve the desired model throughput (a few hundred milliseconds per iteration) [38, 194].

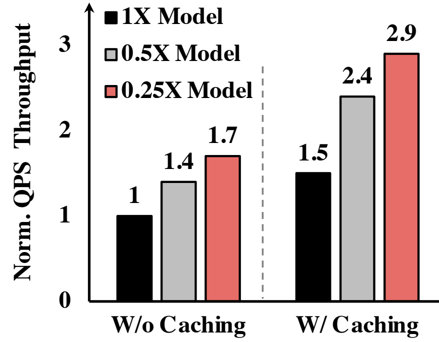
### 4.3.2 Challenges in DLRM Deployment

Due to its significant impact on revenue and numerous iterations needed to train a DLRM model, DLRM deployments follow the "*achieve better accuracy and run as fast as possible*" paradigm [194, 262, 290]. The execution speed and accuracy of a DLRM model are respectively measured by Query-Per-Second (QPS) throughput and Normalized Entropy (NE) loss [121]. Larger QPS and smaller NE indicate better performance, and any relative > 0.02% NE gain is considered to be significant [88, 265]. However, optimizing both aspects leads to novel tussles and challenges in real-world deployments.

**Larger embedding sizes improve NE** Embedding size of modern DLRMs is ever-growing to accommodate more embedding rows for sparse features and their instances [194, 261]. Figure 4.2 reports the size of the instance set over 15 days' data in a real-world DLRM system. We observe that even though a small portion of the trained instances will seldom be accessed again in later days (Figure 4.2(a)), the total number of unique instances increases by 1.5× every week (Figure 4.2(b)). As DLRMs are often trained on months of data and retrained over time, the size of the instance set




(a) Large embeddings improve NE.



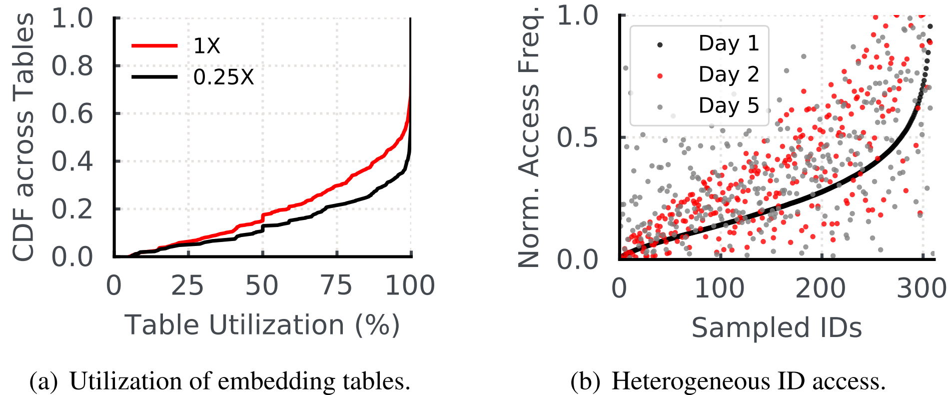

(b) Large embeddings hurt QPS.

Figure 4.3: Compared to the full (1×) model, smaller embedding sizes hurt model NE (i.e., larger NE regression), but improve QPS. 0.25× and 0.5× denote using 25% and 50% of the full model size, respectively.

will eventually far exceed the embedding size. To cap the embedding size, existing designs often perform hashing on the raw instance IDs, and then use the hashed IDs to access their embedding rows [33].

Intuitively, using more embedding rows implies more instances are considered, thus enabling better data coverage for better NE. Figure 4.3(a) reports the impact of the embedding size on the NE regression at different times of training. NE regression denotes the accuracy degradation of using a smaller embedding size w.r.t.the full-size model.We notice that (i) using a smaller embedding size can greatly hurt NE. For example, reducing the number of embedding rows by 75% (i.e., 0.25× model) results in ∼0.02% NE regression on Day 2; Worse, (ii) this NE regression inflates as the training evolves over time as more instances are spawned.

**Large embedding sizes hurt QPS** However, using more embeddings can slow down model execution and consume more machine resources in multiple execution phrases: (i) slower embedding access if we can not retain all embeddings in high-bandwidth GPUs; (ii) longer communication as we may need to transfer more embeddings over the network [38, 283]; and (iii) longer computation as more embeddings need to be computed on. Figure 4.3(b) shows, compared to the full model, 0.5× model achieves 1.4× QPS speedup in the same resource setting. Here, we note that state-of-the-art DLRM optimizations [194, 261], which cache and prefetch the embeddings to be accessed in future batches, cannot eliminate the QPS drop (Figure 4.3(b)). More importantly, they can be insufficient for online training and model serving as we may not know the input data in advance.




(a) Utilization of embedding tables. (b) Heterogeneous ID access.

Figure 4.4: Embedding access varies across IDs and over time, leading to distinct table utilization in existing embedding designs.

### 4.3.3 Opportunities for In-Training Pruning

For a given DLRM, recent advances have made considerable progress for efficient communication [38, 112, 229] and/or computation [154, 194, 88]. Instead, we focus on a complementary opportunity that *reduces the embedding size needed without NE regression, by adaptively pruning embeddings during model training*. Our approach is based on the following observations.

**Handcrafted embeddings are suboptimal** Designing optimal embeddings (e.g., deciding the number of per-feature embedding rows and which embedding weights to retain) is as yet an open problem in the ML community [100]. Hence, DLRM systems often decide the embedding size using human-defined rules, e.g., by estimating the feature popularity [100] and/or hyper-parameter tuning by model experts before training takes place [290]. Not only does this require great human effort and resources to explore, but it can also be suboptimal due to limited adaptivity at runtime (e.g., deciding which instance's embedding to retain if many instances are generated).

Worse, existing DLRM systems often treat per-feature embedding tables individually for ease of management. This can underutilize or overload individual tables as data distribution changes over time. Indeed, when we analyze the table utilization in a one-day training window (i.e., number of accessed embeddings over the total number of embeddings on that day), we notice large heterogeneity (Figure 4.4(a)). Intuitively, tables that are fully utilized can degrade NE because many instances are hashed to the same embedding row, leading to hash collisions. However, underutilized tables cannot trade in their space during training because of the suboptimal predetermined embedding size and inelastic embedding designs.

**Embeddings have heterogeneous characteristics** Figure 4.4(b) zooms into individual embedding rows, where the sampled IDs (i.e., x-axis) are ordered based on their access frequency on Day 1.

We notice that the access frequency of embeddings varies across embedding IDs and over time, since user preferences change over time. We have similar observations on embedding weights too (§4.5.1). Since the model output (accuracy) is determined by the input instance (e.g., which embedding is accessed) and embedding weights, this implies a potential to identify and retain more important embeddings during training to maximize final model accuracy.

## 4.4 AdaEmbed Overview

In this chapter, we introduce an *automated in-training pruning system*, AdaEmbed, to adaptively optimize per-feature embeddings at scale for better model accuracy. Unlike existing efforts for model pruning, which focus on conventional models [85, 115, 62] and/or prune model size when training completes [176], AdaEmbed automatically identifies and retains important embeddings for the given embedding size to improve performance while training is ongoing. Our evaluations in industrial settings show that in addition to saving resources throughout training, AdaEmbed provides superior model accuracy to its post-training pruning counterparts (§4.7.4).

AdaEmbed is a complementary system that acts as a shim layer atop today's embedding designs (Figure 4.5). It has a central coordinator and a set of distributed on-worker agents:

- *AdaEmbed Coordinator*: It gathers the embedding information from agents, determines the global pruning decision, and orchestrates the agent to enforce the pruning.
- *Memory Manager*: It is located inside each AdaEmbed agent and manages the physical memory for today's embedding designs. At runtime, it receives the pruning decision from the coordinator and executes pruning on local embedding weights.
- *Embedding Monitor*: It resides along with the memory manager to track embedding importance and reports the profiling results of the importance to the coordinator.

Figure 4.6 illustrates the interface of AdaEmbed, which supports existing DLRM systems in a few lines of code.

**Training Lifecycle** Similar to current DLRM deployments, ❶ each worker is in charge of a subset of sparse features, which is determined by the embedding partition of model parallelism. The worker processes the input data (i.e., a list of embedding IDs) of those features. ❷ However, the inputs are first forwarded to AdaEmbed agent to look up the physical address of each embedding's weights (Line 12). ❸ The physical address is then used to fetch the embedding weights for read and write operations. The rest of model training adheres to existing designs. ❹ After each training iteration, the embedding monitor updates the embedding importance with the training feedback (Line 16). Periodically, it samples the importance of different embedding rows and notifies the coordinator of the profiling results. The coordinator determines how to prune embeddings subject to

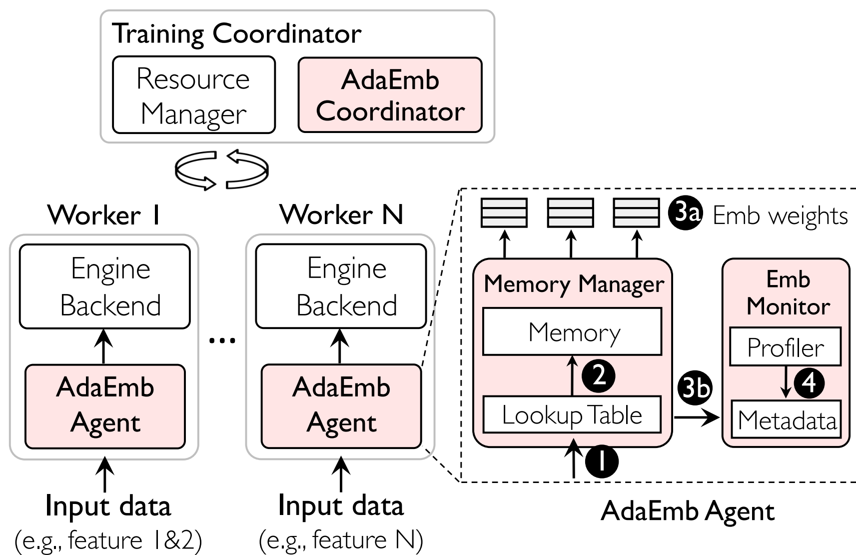



Figure 4.5: AdaEmbed overview and its in-training execution flow. AdaEmbed components are in red.

the total embedding size and guides the memory manager to admit and prune embeddings at scale.

## 4.5 AdaEmbed Design

Practical DLRMs often contain hundreds of sparse features and up to billions of embedding rows [283, 88]. They run across hundreds of GPUs on non-stationary model inputs to get the desired model execution speed [194, 38]. These lead to the following challenges toward practical in-training pruning of embedding rows:

- *Heterogeneity*: The characteristics of embeddings (e.g., data distribution and embedding weights) vary across instances of the same feature. This, as well as the physical size of the embedding row, differs across features too. How to measure which embeddings are important to retain for better model accuracy (§4.5.1)?
- *Dynamics and Scalability*: The importance of individual embeddings varies over iterations at a sub-second speed. As such, improving model accuracy requires pruning in a timely manner to maximize the number of important embeddings. However, identifying important embeddings out of billions distributed across hundreds of workers, and then pruning on terabytes of embedding weights can lead to large overhead. How to orchestrate pruning under training dynamics (§4.5.2)? Additionally, how to efficiently enforce pruning on each worker's memory to avoid throughout degradation (§4.5.3)?
- *Extensibility*: Existing systems are built atop a variety of embedding designs, such as key-value storage [290, 261] or highly optimized but fixed-size tensors [36, 33]. How to provide generic

```
import AdaEmbed

def dlrm_model_training():
  # Wrap existing embedding modules
  emb_agent = AdaEmbed.create_agent(
    emb_tables=model.embs, pruning_config=config)

  for _ in range(num_iterations):
    input_ids = get_next_data_batch()

    # Look up physical embedding address
    emb_physical_ids = emb_agent.look_up(input_ids)
    feedback = model.train_step(emb_physical_ids)

    # Update embedding importance with feedback
    emb_agent.update_importance(input_ids, feedback)
```

Figure 4.6: AdaEmb supports existing DLRMs with minor changes.

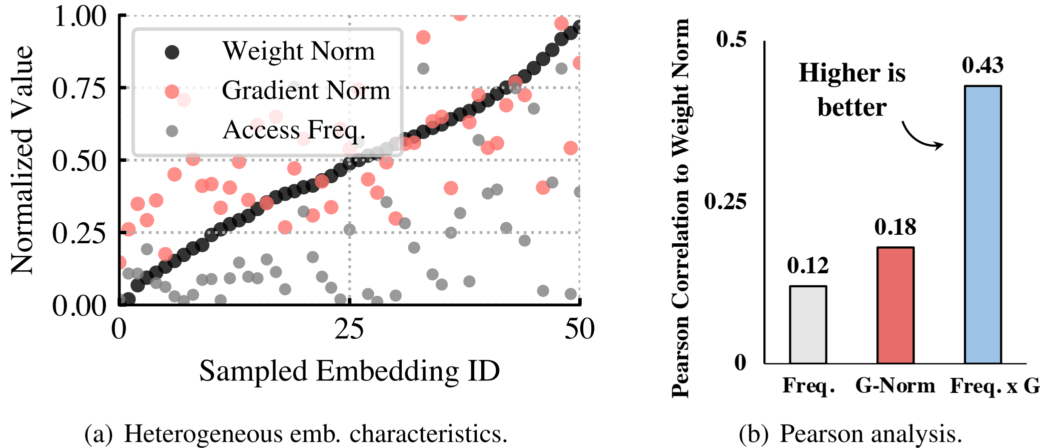
systems support to minimize modifications to existing DLRM systems (§4.5.3)?

### 4.5.1 Embedding Monitor: Identify Important Embeddings

Given the embedding size, we aim to trade the less important embedding rows for the more important ones. This requires us to consider the importance of each embedding row in terms of the contribution of its embedding weights to model accuracy, as well as its physical size. However, determining the optimal pruning strategy during training is challenging. First, the model output (accuracy) is affected by the complex interplay between input feature instances (e.g., which item IDs appear) and their embedding weights. Even with full model information after training completes, pruning is still a fundamental open problem in the ML literature [85, 176]. Second, during model training, this interplay becomes more intractable because of the large spatiotemporal variations in the distribution of model inputs and embedding weights (Figure 4.7(a)). Worse, once we prune an embedding's weight vector, it is difficult to assess its impact on model accuracy as training moves on. These challenges are amplified by the need to account for feature-level heterogeneity too (e.g., different weight vector sizes across features).

AdaEmbed employs the embedding monitor to capture the embedding importance of individual rows within the feature, and then extends it to identify important rows across features.

**Intra-Feature Embedding Importance** For embeddings of the same feature, we introduce a data- and model-aware importance metric $EI(i)$ to capture the importance of each row $i$ to model accuracy. Instead of relying on the embedding weights that become stale after being pruned, $EI(i)$ is the runtime combination of access frequency and gradient, i.e., $EI(i) = freq_t(i) \times \|\nabla g_t(i)\|$. $\|\nabla g_t(i)\|$ is the L2-norm of $i$'s gradient in iteration $t$, and $freq_t(i)$ is the access frequency. So the embedding with a higher access frequency and a larger gradient norm is deemed more




(a) Heterogeneous emb. characteristics.

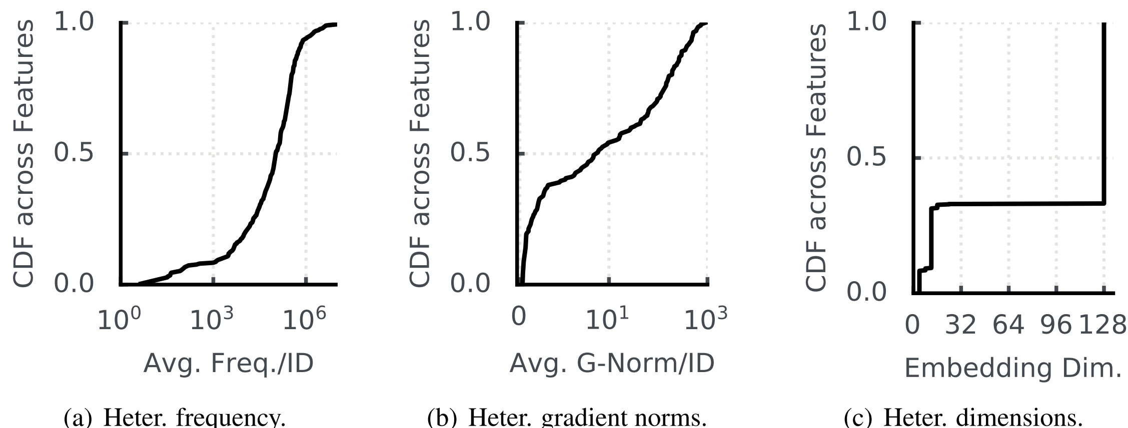
(b) Pearson analysis.

Figure 4.7: (a) Embedding gradient and access frequency are heterogeneous, (b) while their combination reports a larger correlation to the embedding weights. A correlation value > 0.4 indicates a positive, medium to strong association.

important. Here, collecting $EI(i)$ introduces negligible overhead, because the embedding gradient is already generated during back-propagation of training regardless of AdaEmbed. Since the gradient is generated and shared by mini-batch samples [194], the importance of pruned-but-accessed embeddings will continue to be updated.

Our importance design is motivated by multiple factors:

- Intuitively, the output of sparse feature layers (i.e., pooled embedding weights) is often derived by taking the sum or maximum of input embedding weights (§4.3.1); so we should retain the embeddings that affect many model inputs (i.e., frequently accessed) and that affect model outputs more (i.e., larger weights). While we do not have information about future weights after pruning an embedding, we observe a strong correlation between our frequency-gradient combined metric and the final embedding weights when training converges (Figure 4.7). This is because frequent weight updates with large gradients typically result in larger weights.
- Theoretically, embedding rows are designed for training different bins of data instances: each bin holds only one type of category instance (i.e., a specific ID), and bins can have different data volumes (i.e., different access frequencies of IDs). Now, we want to select and retain certain bins (embeddings). This, in concept, is similar to the importance sampling problem in the ML literature [153, 103]: To improve model convergence by selecting the right bins to train the model, the optimal solution is to select bin $i$ with a probability proportional to the aggregate gradients of training all that bin's data. In our formulation, the training samples within the same bin are identical, because they correspond to the same specific ID. Therefore, the aggregate gradients herein is equivalent to the product of the number of training samples and the gradient of the individual sample (i.e., $EI(i) = freq_t(i) \times \|\nabla g_t(i)\|$).




(a) Heter. frequency. (b) Heter. gradient norms. (c) Heter. dimensions.

Figure 4.8: Magnitudes of embedding access frequencies and gradients vary across features, making it hard to compare $EI(i)$.

Empirically, our fleet-wide evaluations show that our importance design outperforms its alternatives (§4.7.4).

Since the gradient and access frequency can fluctuate during training (e.g., due to the randomness in sampling mini-batches), we need to account for these uncertainties in $EI(i)$. Here, the embedding monitor considers $EI(i)_t = freq_t(i) \times \|\nabla g_t(i)\| + EI(i)_{t-1}$, whereby we reduce uncertainties in individual iterations and only need to update the importance of accessed embeddings. This is because the importance of not accessed embeddings remains unchanged as $freq_t(i) = 0$. In reality, only a subset of embeddings are accessed, so we can reduce the overhead significantly (§4.5.2). Moreover, to account for the temporal variation, we use a moving average that decays $EI(i)$ by a factor of 0.8 every $T$ iterations.

**Inter-Feature Group Pruning** Retaining important embeddings subject to the total size naturally leads to a global pruning design, in which we hope to allocate different embedding sizes to individual features. However, the values of embedding importance can vary across features by orders of magnitude. This can be due to features with fewer instances often having larger average access frequencies per embedding, and/or different initialization mechanisms of the embedding weights leading to gradients of different magnitudes (Figure 4.8). As such, directly using the intra-feature embedding importance for comparison across features can result in a large bias, as embeddings with greater importance values are not necessarily more important than those of other features. Moreover, as the dimension of embedding vectors of different features can vary (Figure 4.8(c)), deciding which embeddings are more valuable to retain becomes intricate when large embedding importance and vector size are in conflict.

Because we rely on the relative ranking of importance to determine pruning (e.g., prune the tail 40% less important embeddings), we can tackle the comparison bias across features using the

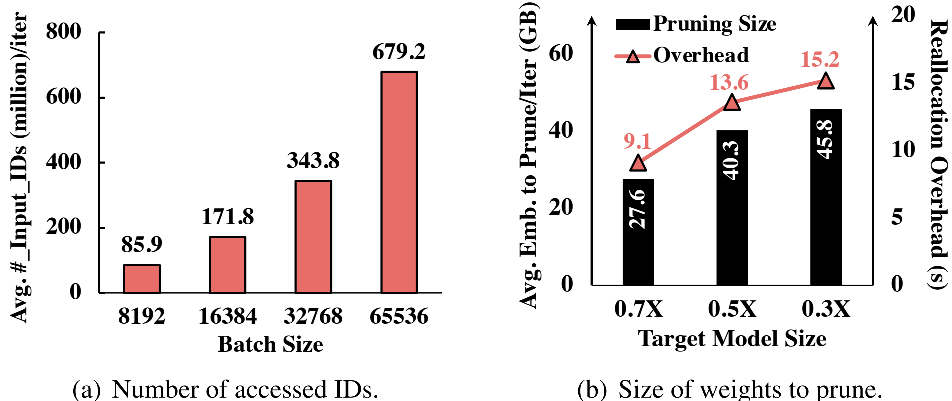



Figure 4.9: (a) Each iteration accesses millions of embeddings. (b) Pruning needs to reallocate a large amount of embedding weights.

popular normalization philosophy [102]; i.e., by normalizing each embedding's importance by that feature's distribution of all embeddings' importance. This way, the embedding importance of different features takes on similar ranges of values, and the more important embeddings of each feature are still prioritized because of having larger relative importance values after normalization. The embedding monitor normalizes the embedding importance of each feature by the 95th percentile of its distribution (i.e., $EI(i)/EI_{95th}(feature(i))$) to avoid outliers.

Next, to account for different weight vector sizes across features, AdaEmbed groups features with the same embedding dimension and then performs global pruning within the feature group. In reality, DLRMs are configured with only a handful of distinct embedding dimensions (Figure 4.8(c)) to reduce hyper-parameter tuning and/or to achieve better parallelism (e.g., balancing embedding sharding [194, 290]). This implies a big opportunity to group many features, which already forms a large shared embedding size for inter-feature group pruning. By default, AdaEmbed initializes the per-group embedding size based on the number of in-group features and the total embedding size (i.e., $\frac{num_group_features \times group_feature_dim}{num_features \times avg_feature_dim} \times total_size$) to uniformly allocate the space to each dimension. Note that unused embedding storage will be picked up by other groups (§4.5.3). When developers have more advanced information about features (e.g., feature importance), AdaEmbed provides APIs for customizing feature groups and sizes (§4.6).

Our evaluations show that with importance normalization and group pruning, AdaEmbed achieves better resource savings and model accuracy (§4.7.3).

### 4.5.2 AdaEmbed Coordinator: Prune at Right Time

In real-world DLRM systems, each training iteration involves updating the importance of millions of embedding rows in terabyte-sized models (Figure 4.9(a)). At that scale, orchestrating hundreds of workers to prune leads to a trade-off between the pruning overhead and quality. Frequent

```
1:  weight_table ← EmbWeights()                                   ▷ Physical weight tables
2:  emb_meta ← Init(weight_table)                                 ▷ VHPI metadata
3:  pruning_start ← false                                         ▷ Enforce pruning or not
4:  Function UpdateEmbs (input_ids, feedback) :
      /* Monitor: Update embedding importance asynchronously to model training.      */
5:    UpdateImport(input_ids, feedback)
6:    if pruning_start == true then
7:      EnforcePruning()                                          ▷ Stall training
8:      pruning_start ← false
9:  Function MonitorImportance (ProfilingInterval Δ) :
      /* Coordinator: asynchronously inspect big changes in importance distribution via profiling
         across workers.                                                             */
10:   last_dist ← null
11:   while training == true do
12:     if mod(current_time, Δ) == 0 then
13:       cur_dist← ProfileImportance()
14:       pruning_start←Diff(last_dist, cur_dist) > p
15:       last_dist ← cur_dist
16: Function EnforcePruning () :
      /* Memory manager: Identify embedding rows to admit and prune subject to the given
         embedding size.                                                             */
17:   admit_emb, evict_emb ← IdentifyRecycleEmbs(
18:               emb_meta, weight_table.size)
      /* Redistribute the lookup mapping from the embedding ID to the weight vector, whereby
         admitted embedding rows can recycle the weight vector of pruned ones.       */
19:   RedistLookup(emb_meta, admit_emb, evict_emb)
      /* Reset embedding weights for admitted embeddings.                            */
20:   weight_table.ResetEmbs(admit_emb)
```

**Algorithm 4.1:** Pseudo-code of AdaEmbed runtime

pruning allows for better decision quality, i.e., maximizing the number of important embeddings all the time for potentially better model accuracy. Yet, pruning can require cleaning up and creating tens of gigabytes of embedding weights, which can take many seconds and significantly slow down the sub-second training iterations (Figure 4.9(b)). This trade-off becomes more intractable as a result of training dynamics; e.g., stochastic gradient descent can introduce large noise to embedding gradients, thus the embedding importance. As such, pruning too frequently can also be suboptimal (§4.7.4).

To find the sweet spot between pruning overhead and quality, AdaEmbed Coordinator decides

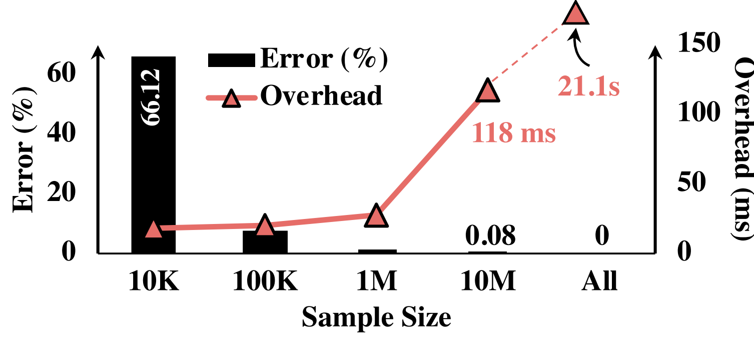



Figure 4.10: Profiling can get accurate results with little overhead.

the right time to prune to reduce the number of pruning rounds needed, and instructs the memory manager to minimize the overhead in each pruning round when pruning embedding weights (§4.5.3). Algorithm 4.1 outlines how AdaEmbed Coordinator orchestrates efficient embedding pruning. The embedding monitor updates the importance of accessed embeddings after each training iteration (Line 4), and periodically profiles embedding importance (Line 9). The results of the profiling will be sent to the coordinator. In the event of big changes in the importance distribution, the coordinator initiates a new pruning round and notifies the memory manager of the pruning decision (Line 9). The memory manager on each worker then executes pruning and admits new embedding weights at scale (Line 16).

Intuitively, pruning cares about the importance ranking of individual embeddings instead of their dynamic importance. Therefore, AdaEmbed coordinator relies on the importance distribution of all embeddings again, and initiates pruning if the importance distribution has changed greatly since the last pruning round. To effectively gather the importance distribution across hundreds of machines, each local agent samples a small portion, $P$, of embedding importance values on that agent. The coordinator then can estimate how many embeddings have crossed the pruning boundary, i.e., the number of embedding rows whose importance ranking has fallen below or risen above the $X_{th}$ percentile of the distribution since the last pruning round. $X_{th}$ is the cut-off importance boundary determined by the size limit (i.e., $\sum_{emb \in top_Xth} size(emb) < total_size$), and the agent will prune the weight vector of the embeddings whose importance value is smaller than the cut-off importance.

As shown in Figure 4.10, while more samples, $P$, allow for a more precise estimate of $X_{th}$ importance, this will also increase the coordination overhead, such as in collecting importance distributed across hundreds of machines and then computing distribution changes. In fact, we can use the concentration theorem in the probability sampling [266] to decide the right number of samples.[1] This gives us $\sim$ 5M embedding rows out of billions to sample on each machine, in order

[1] The minimum number of samples $P$ needed to ensure $Pr[|\bar{X} - E[\bar{X}]| < \epsilon] > \delta$ is $P = \frac{(X_{max} - X_{min})^2 \ln{(2/\delta)}}{2\epsilon^2}$ for the distribution of variable $X$. $E[\bar{X}]$, $X_{max}$ and $X_{min}$ are the expectation, maximum and minimum of $X$, respectively.

to ensure a deviation from the global ground truth of less than 1%. In addition to having a smaller computation overhead, this results in negligible network traffic, $5M \times 4bytes \sim 20$ megabytes as $EI(i)$ is a 4-byte float, over tens of Gbps network to the coordinator. As suggested by today's data validation systems [57, 182], we consider a big change to have occurred and initiate pruning when more than $c$ =5% of the total embeddings cross the boundary (i.e., we need to prune and admit more than c% embeddings), and issue this lightweight profiling per minute. This avoids the large overhead caused by pruning in each training iteration, while ensuring that the current embedding allocation is at most $c$% worse than what we can achieve through pruning in each iteration. We show that profiling achieves a small deviation and little overhead (i.e., the 5M sample size in Figure 4.10).

**Convergence Analysis** As described in §4.5.1, our design of embedding importance draws inspiration from importance sampling, which has been shown in ML theory [153, 103, 175] for its ability to reduce gradient variance and accelerate training convergence. Empirically, our extensive evaluations using months of real-world data and models demonstrate that AdaEmbed consistently improves model accuracy by pruning at the right time, as opposed to pruning too frequently or infrequently (§4.7.4).

### 4.5.3 Memory Manager: Prune Weights at Scale

As the reallocation of embedding weights is hundreds of times slower than each training iteration (Figure 4.9(b)), reducing the number of pruning rounds needed is still far from achieving negligible overhead in practice (§4.7.2). To avoid intense memory reallocation, the memory manager of AdaEmbed employs a Virtually Hashed Physically Indexed (VHPI) design to decouple the management of embeddings from their physical weight vectors, whereby AdaEmbed can recycle the weight vectors of different embeddings to enable efficient pruning for a variety of existing embedding designs.

VHPI primarily consists of two parts (Figure 4.11):

- *Lookup table*: It stores the metadata information of each embedding instance, including the embedding importance (a `float32`), and the physical address (a `int64`) to that embedding's weight vector. Compared to the weight vector, often a vector of 128 `float`, this payload information introduces a negligible memory footprint ($\frac{3}{128} \sim 2\%$).
- *Weight table*: It is a monolithic physical table for embedding weight vectors. It remains the same as the embedding table of today's DLRM systems, but it is shared across features under the orchestration of the memory manager.

Weights vectors of the pruned embeddings are not retained, while the metadata of all embeddings is always maintained in the lookup table. So the lookup table can include more entries (i.e.,

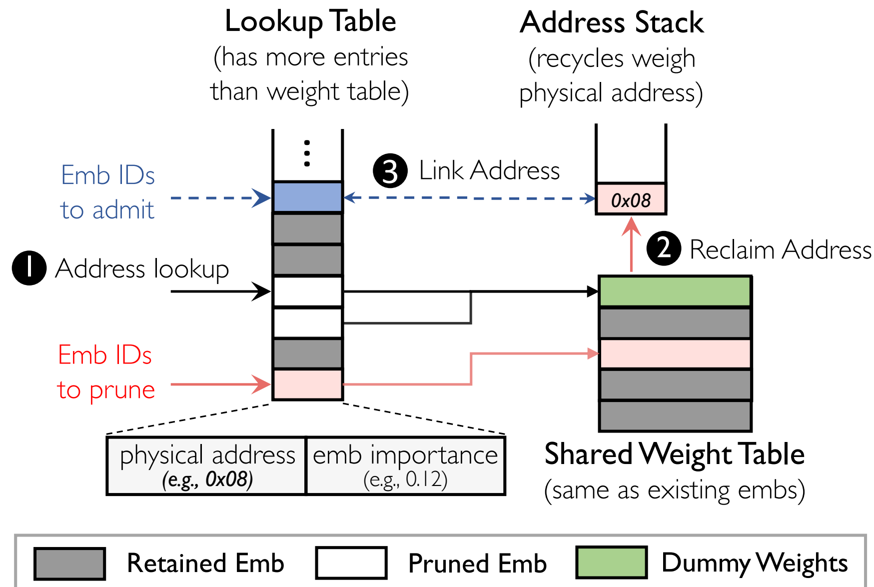



Figure 4.11: VHPI employs lookup table to link each embedding to the weight vector, and recycles the vector of pruned embeddings without intense memory allocation.

embedding IDs) than the weight table. This allows us to adaptively determine the link between embeddings and weight vectors to recycle weight vectors. Moreover, this can improve model accuracy by reducing hash collision (§4.7.4), as we can make the lookup table very large to accommodate many embedding entries without expanding the weight table.

The memory manager performs two primitive operations for weight pruning at runtime (Figure 4.11):

- *Address lookup*: It looks up the physical weight address for each embedding ID to access its embedding weights. ❶ If that embedding row is pruned, to avoid breaking existing designs (e.g., missing weights due to pruning), the lookup returns a shared physical address that points to a weights vector containing constant zeros. Access to this dummy vector will be folded on the execution backend due to the same entry, reducing redundant execution.
- *Weight allocation*: It executes the pruning decision to prune and admit embeddings. ❷ To prune an embedding row, VHPI first de-links and reclaims the current physical address of that embedding's weight. It then sets the address of the pruned embedding's lookup entry to the address of the shared dummy vector, redirecting the future access. ❸ To admit an embedding, VHPI pops an available physical address and links this address with the lookup entry, thereby recycling the physical memory. Meanwhile, the memory manager resets the weight vector values to clean up the previously pruned weight state.

However, it is not straightforward to reset (i.e., reinitialize) the weight values for admitted embeddings, because the model herein is partially trained and the values of embedding weights already

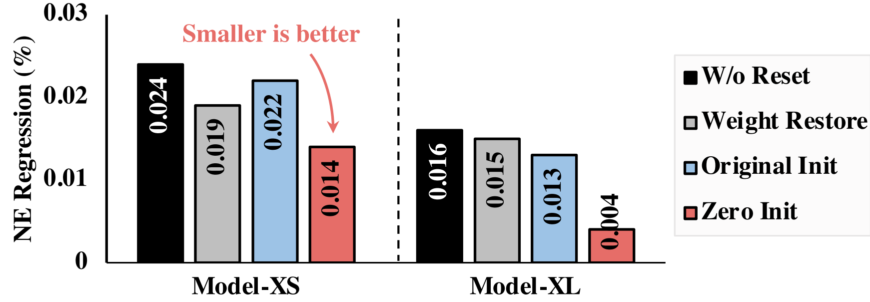
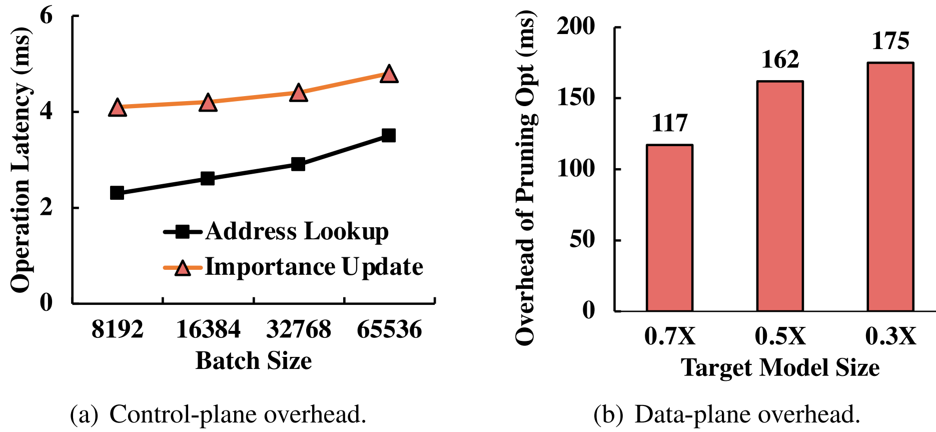



Figure 4.12: Zero initialization performs better (0.5× model).




(a) Control-plane overhead. (b) Data-plane overhead.

Figure 4.13: VHPI operations introduce little overhead.

differ by orders of magnitude (Figure 4.7(a)). Improper initialization (e.g., random initialization) can introduce a large amount of noise to the retained embeddings. Eventually, this will hurt model accuracy, especially considering the noise from millions of admitted embeddings in each pruning round.

Here, we investigated four popular strategies to reset weight vectors (Figure 4.12): (1) *w/o reset*: inherit the weights of pruned embeddings without resetting them; (2) *weight restore*: evict previously pruned weights to extra storage (e.g., disk) and reinstate the weights when that embedding is reclaimed; (3) *original initialization*: randomly initialize embedding weights as at the start of training; and (4) *zero initialization*: reset embedding weights to zeros. Intuitively, the restored weights will become too stale since they were pruned (often thousands of iterations ago). Original initialization and w/o reset can introduce large noise, as the weights have already been of differing magnitudes. Here, we advocate resetting the weight vector values to zeros, as this can avoid large noise while allowing the admitted embedding to learn from scratch. Indeed, our real-world evaluations report that zero initialization outperforms its alternatives (Figure 4.12).

| Model | # of Sparse Features (Approximate Value) | Raw Emb Size (Approximate Value) | # of GPUs | w/ Same Model NE | | w/ Same Emb Size | |
|---|---|---|---|---|---|---|---|
| | | | | Memory Saving | QPS Speedup | Avg. NE Gain (%) | QPS Overhead |
| Model-XS | 1000 | 200 GB | 32 | $\approx$ 35% | 1.1× | 0.015 | 0.4% |
| Model-S | 600 | 350 GB | 32 | $\approx$ 45% | 1.2× | 0.018 | 0.2% |
| Model-M | 1000 | 1 TB | 64 | $\approx$ 40% | 1.2× | 0.028 | 1.6% |
| Model-L | 1000 | 1.1 TB | 64 | $\approx$ 55% | 1.3× | 0.021 | 1.3% |
| Model-XL | 800 | 1.5 TB | 128 | $\approx$ 60% | 1.3× | 0.026 | 1.1% |

Table 4.1: Summary of improvements. AdaEmbed reduces the embedding size needed for the same model accuracy (NE), while improving NE using the same embedding size. We report the approximate memory savings, since evaluating all memory settings is unaffordable.

**Overhead Analysis** Among all the operations involved in VHPI, address lookup and importance update to the lookup table take place every iteration and consume a few milliseconds (Figure 4.13(a)). Weight pruning to the weight table consumes a few hundred milliseconds (Figure 4.13(b)), but it occurs every hundreds of iterations. Overall, these operations lead to little end-to-end overhead in large-scale deployments (§4.7.2).

## 4.6 Implementation

We implemented a system prototype of AdaEmbed to support distributed DLRM deployment across GPUs. Our implementation requires minor changes to existing DLRM systems.

**AdaEmbed Backend** AdaEmbed backend is implemented as GPU operators for fast execution. The VHPI metadata (e.g., embedding importance and weight address) are hosted on GPUs to process embeddings in parallel. The address lookup and importance update operations require no change to existing DLRM systems. As we need to reset the weight vector, the weight allocation operation requires existing frameworks to expose an API to access their weight table, but this requires a few lines of code change. The local agent interacts with the coordinator via TCP connections.

**Fault Tolerance** As a shim layer, AdaEmbed can be integrated into existing DLRM checkpoints by adding its state information to the model state. This not only minimizes the modification to existing designs, but also ensures that the saved AdaEmbed state conforms to the embedding weights at that time. When training is resumed, the model reloads the checkpoint, which restores the AdaEmbed state too. At runtime, AdaEmbed runs a lightweight daemon to back up VHPI metadata after each pruning round, and to resume its components if the current instances crash.

**Interfaces** AdaEmbed exposes Python APIs as the frontend (Figure 4.6), and it can also take *json* as input (Figure 4.14).

```
"adaembed_configs": {
  "total_emb_size": "1 TB",// Total embedding size
  "feature_configs": {
    "default_group": {...},
    "group_1": {  // Features to use group pruning
      "features": ["feature1", ...],
      "total_emb_size": "200 GB",
    } ...          // Other feature groups
  }
}
```

Figure 4.14: Example embedding configuration in AdaEmbed.

## 4.7 Evaluation

We evaluate AdaEmbed in real-world DLRM systems across hundreds of GPUs. Our evaluation results on different industrial models and months of data are summarized as follows:

- AdaEmbed can reduce 35-60% embedding size and improve model execution speed by 11-34% without compromising model accuracy (§4.7.2).
- AdaEmbed can reduce manual efforts by automatically finding better per-feature embeddings, achieving noticeable accuracy improvements (§4.7.2-§4.7.3);
- AdaEmbed improves performance over a wide range of settings and outperforms its design counterparts (§4.7.4);

### 4.7.1 Methodology

**Experimental setup** We use models and data from industry DLRM systems in the evaluation. Table 6.2 depicts high-level statistics of the model. They span different scales and recommendation tasks, including click-through rate prediction and ranking. We train each model on 14 days' data to obtain the model *lifetime NE*, which indicates the cumulative model accuracy throughout training, and then test the model on the 15th day's data to get the *evaluation NE*. Each day has many terabytes of data input.

The training batch size of each model is 65536, requiring tens of GPU nodes for the desired QPS. Each GPU node has 8 A100 GPUs with 40 GB of GPU memory. The GPUs are interconnected using 200 Gbps RoCE NICs.

**Baselines** To the best of our knowledge, AdaEmbed is the first system to support in-training embedding pruning, and is complementary to existing DLRM efforts. Our evaluations cover two primary baselines: (i) *w/o AdaEmbed*: an industry DLRM system without AdaEmbed support. Based on the access frequency of embedding rows in previous days, rows that are less frequently accessed are removed before training starts. This generates a pruned model derived from the full

model; and (ii) different variants of AdaEmbed with changes in the pruning algorithm (§4.7.4). Here, we focus on the performance improvement of the *w/ AdaEmbed* setup, i.e., the setup using AdaEmbed.

**Metrics** We care about the (i) *memory saving* to achieve the same model accuracy as with the full model (i.e., without NE regression)[2], because we want to minimize the embedding size for better model throughput and resource savings in deployment; (ii) *NE gain* that we can achieve using the same embedding size, since it not only minimizes manual efforts in configuring DLRM embeddings, but also implies higher revenues; and (iii) *overhead* that AdaEmbed introduces in model execution speed (i.e., QPS).

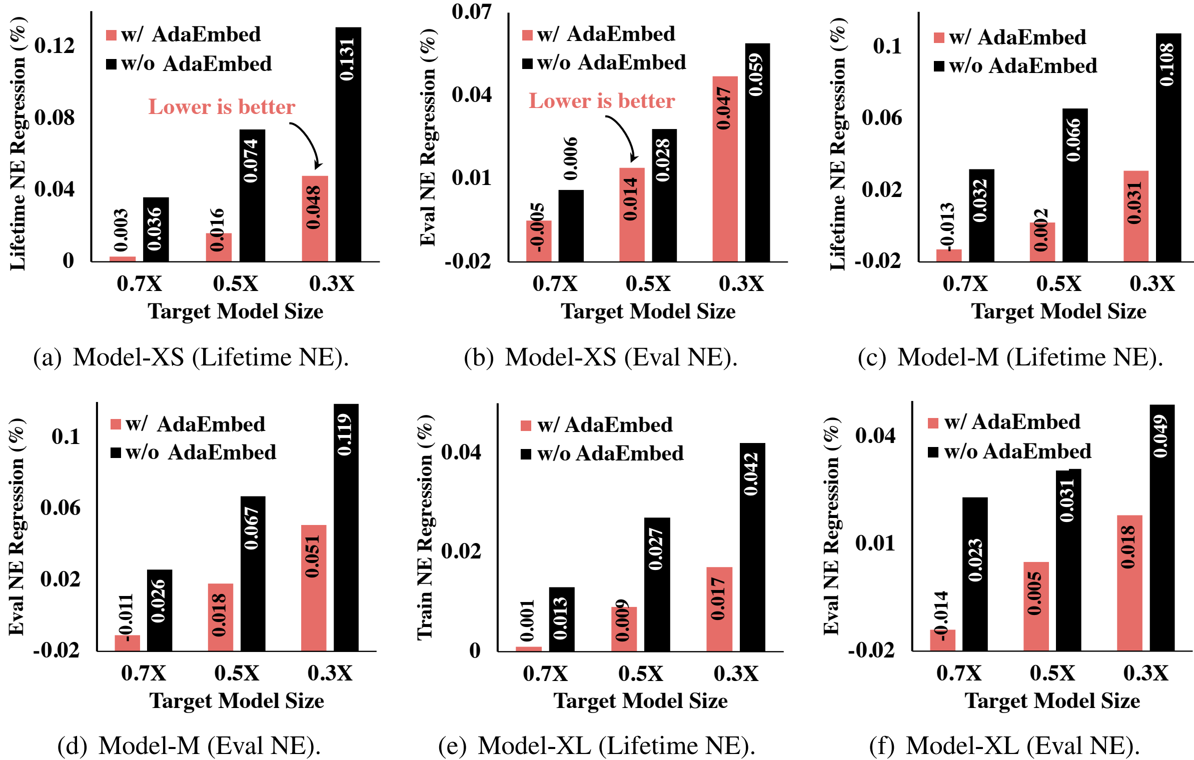
### 4.7.2 End-to-End Performance

Table 6.2 summarizes the key memory saving, NE gain, and overhead of five models at different scales. Meanwhile, Figure 4.15 zooms into three representative models and reports their performance under different target embedding sizes. In our evaluations, NE regression measures the accuracy loss w.r.t.the full model (i.e., 1× model), and any > 0.02% NE gap is considered to be significant [88, 290, 172].

**AdaEmbed cuts resource needs and improves QPS** We first evaluate how many embedding sizes we can reduce without sacrificing model NE. Yet, evaluating all embedding sizes to get accurate memory saving is unaffordable because training with each setup takes thousands of GPU hours. So, we enumerate 0.7× (i.e., cut the embedding size by 30%), 0.6×, 0.5×, 0.4×, and 0.3× of the full model size to approximate this embedding saving with no accuracy drop. Table 6.2 reports that (i) AdaEmbed reduces the model embedding size by 35-60% with no reduction in model accuracy. This implies that we can reduce the machine usage by nearly the same amount (e.g., using 50% fewer GPUs); (ii) the resource savings are more encouraging for large models (e.g., Model-XL vs. Model-XS). One reason behind this is that large models provide gigantic GPU memory for AdaEmbed to reallocate embeddings via inter-feature group pruning (§4.7.3); and (iii) alternatively, reducing the fundamental embedding size provides 1.1-1.3× faster model execution speed (i.e., QPS) when running the model on the same machines.

**AdaEmbed achieves better NE under the same size** Figure 4.15 illustrates that with AdaEmbed, models can achieve 0.011-0.077% better NE using the same embedding size. We notice that (i) AdaEmbed achieves consistently better NE across models and under different target embedding sizes than the baseline; (ii) we can achieve NE gains with smaller embedding sizes (e.g., 0.7×

[2] A smaller Normalized Entropy (NE) loss indicates better model accuracy.



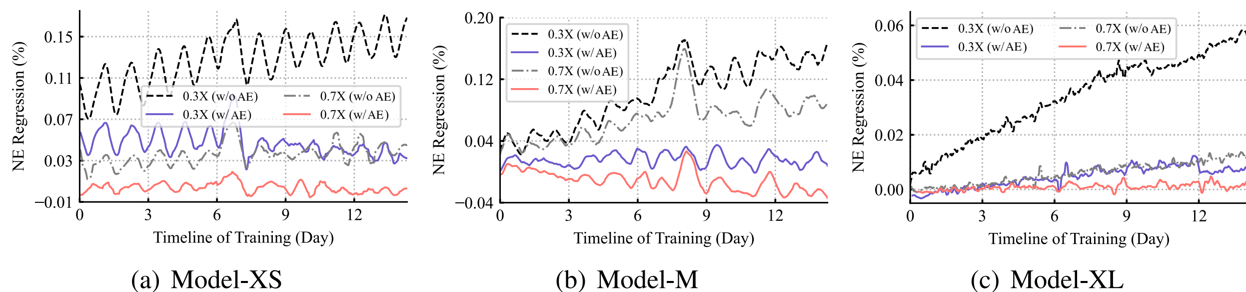

(a) Model-XS (Lifetime NE). (b) Model-XS (Eval NE). (c) Model-M (Lifetime NE).

(d) Model-M (Eval NE). (e) Model-XL (Lifetime NE). (f) Model-XL (Eval NE).

Figure 4.15: AdaEmbed achieves better lifetime NE and evaluation NE. Better lifetime NE implies potentially better model accuracy for online learning deployment, while better evaluation NE indicates better accuracy after offline training (i.e., prior to launching online training). Both NEs are important metrics.

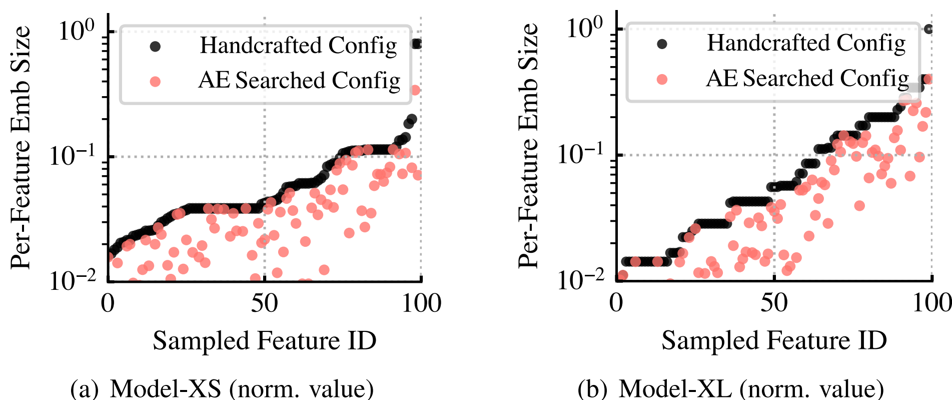
models) even when compared to the full model. This is because AdaEmbed can automatically learn better per-feature embeddings, like the size and which embeddings to retain. Meanwhile, pruning less important embeddings can reduce model overfitting, thereby improving model generalization (accuracy) [48]; and (iii) the lifetime NE gain is more prominent than that of the evaluation NE, because the former is closer to the online deployment (i.e., retraining on real-time data), where AdaEmbed is able to adapt to the latest data distribution.

**AdaEmbed introduces negligible overhead** As shown in Table 6.2, compared to the same-size model in the baseline, AdaEmbed introduces negligible ($< 2\%$) QPS overhead across scales of the deployment (e.g., from 32 to 128 GPUs and 200 GB to 1.5 TB models), because (i) AdaEmbed largely parallelizes operations (e.g., asynchronous importance update and multi-threading); (ii) coordinator selectively initiates pruning rounds; and (iii) the memory manager introduces VHPI to avoid intense reallocation of the physical weight. Note that the memory overhead is $\sim$2% as AdaEmbed introduces only two small buffers (i.e., the lookup address and embedding importance) in VHPI lookup table (§4.5.3).




(a) Model-XS (b) Model-M (c) Model-XL

Figure 4.16: Models with AdaEmbed achieve consistently better NE over time. Troughs are due to data distribution shifting over days.




(a) Model-XS (norm. value) (b) Model-XL (norm. value)

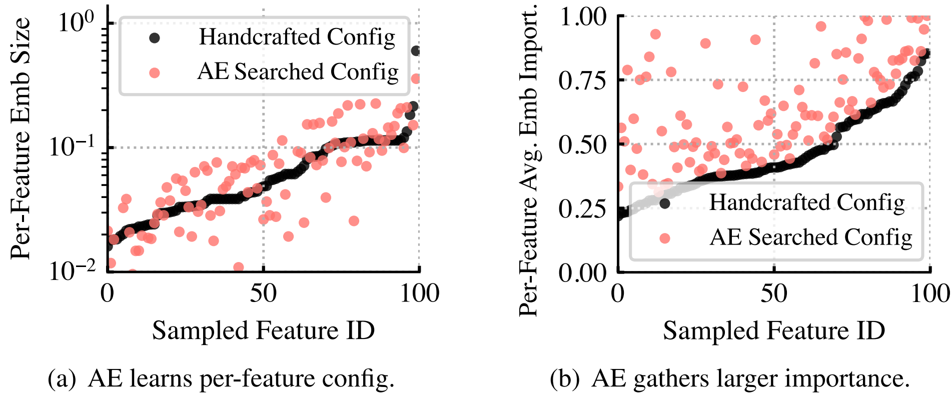
Figure 4.17: For the same NE w.r.t.1× model, AdaEmbed learns better per-feature embedding configuration using smaller size.

### 4.7.3 Performance Breakdown

We next break down AdaEmbed performance by time, the characteristics of sparse features, and design components.

**Breakdown by Time** Figure 4.16 breaks down model NE by time, with each data point on the line representing the moving average of the NE over hourly data (i.e., window NE regression). The training encompasses 14 days of data. We observe that with AdaEmbed, we can achieve consistently small NE regression than the baseline over time.

Moreover, we notice that this NE regression exhibits diurnal variation (e.g., in Model-XS and Model-M). This is because the data distribution (e.g., user preference) of recommendation tasks can change drastically over days. As such, at the beginning of training on a new day's data, the smaller model (e.g., 0.3× model) will experience a larger NE regression as it has less space to accommodate new embedding IDs. However, as the model gradually adapts to the new distribution, this regression tones down. We note that AdaEmbed experiences less NE fluctuation due to its ability to identify and retain important embeddings on the fly.




(a) AE learns per-feature config. (b) AE gathers larger importance.

Figure 4.18: For the same size (0.5× model), AdaEmbed retains more important embeddings to achieve better NE (Model-XS).

**Breakdown by Embedding Features** We next investigate whether AdaEmbed can reduce manual efforts by learning to use better embedding configurations. First, in achieving the same NE as the 1× model, AdaEmbed learns to use smaller embeddings for many features (Figure 4.17). Moreover, using the same embedding size w.r.t.the 0.5× model, AdaEmbed gathers larger average embedding importance on each feature than the handcrafted setup (Figure 4.18), implying that more important embeddings are retained under the same total size. More importantly, we notice that (i) our group pruning shares similar preferences to the handcrafted configuration. Specifically, AdaEmbed tends to allocate more embeddings to those features that the model expert also values highly. However, (ii) some features are allocated fewer embeddings, but AdaEmbed eventually achieves better NE, indicating that AdaEmbed can automatically find better embedding configurations.

**Breakdown by Components** We break down our design into two variants (i) (AdaEmbed w/o Norm): disable importance normalization in group pruning; and (ii) (AdaEmbed w/o Group): completely disable group pruning, so the per-feature embedding size is resized to X% of the full model. We notice that both normalization and group pruning contribute to better NE (Figure 4.19). This is because (i) group pruning allows greater flexibility to resize the per-feature embedding using the shared gigantic weight table; and (ii) importance normalization helps to reduce the inter-feature heterogeneity by prioritizing important embeddings of each feature when comparing embedding importance globally.

### 4.7.4 Sensitivity and Ablation Studies

**Impact of pruning frequency** AdaEmbed Coordinator initiates a pruning round when the importance distribution radically changes. Next, we evaluate the impact of pruning frequency by deterministically enforcing pruning after training every-minute ($\sim$ 50 training iterations) and

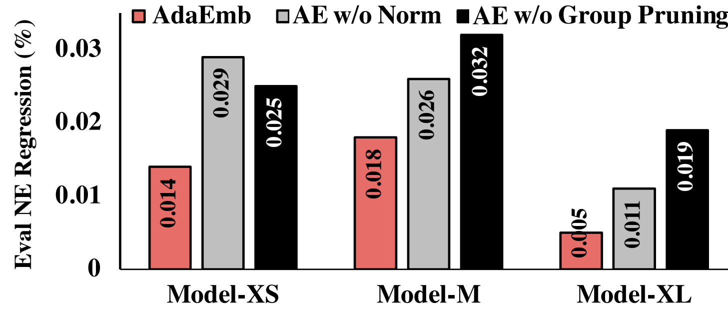
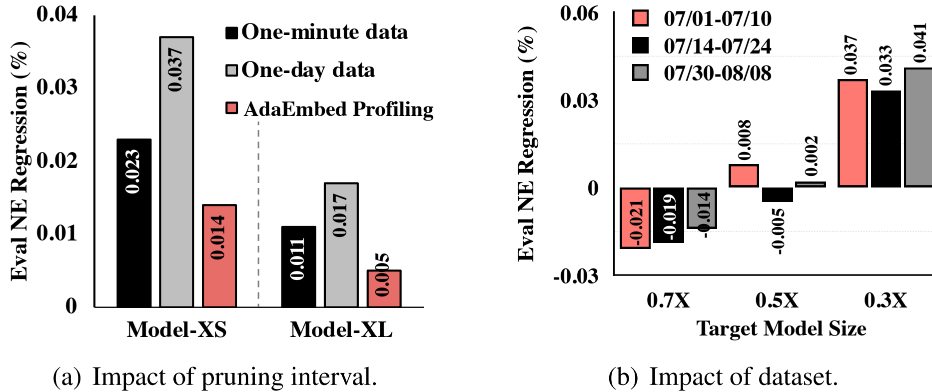



Figure 4.19: Performance breakdown of AdaEmbed (AE) design.




(a) Impact of pruning interval.

(b) Impact of dataset.

Figure 4.20: AdaEmbed achieves improvement across settings.

every-day data ($\sim$ 70K training iterations). Figure 4.20(a) reports that pruning too frequently and infrequently (i.e., pruning every one-minute and one-day data) both lead to suboptimal NE. The former is due to large training noise affecting instantaneous embedding importance, while the latter is due to AdaEmbed missing to admit important embeddings in a timely manner. Instead, the selective pruning of AdaEmbed achieves better performance by relying on the overall importance distribution at runtime.

**Impact of different data** Figure 4.20(b) reports the NE performance of model-S on three distinct datasets. Each training spans 10 days' training data, and we report the evaluation NE on the data of day 11. While the NE gain varies slightly as the data distribution varies across dates, AdaEmbed consistently achieves 50% memory savings with no NE regression.

**Alternatives of embedding importance** We next experiment with different embedding importance designs in training 10 days' data. Here, we consider using the frequency, gradient, and their

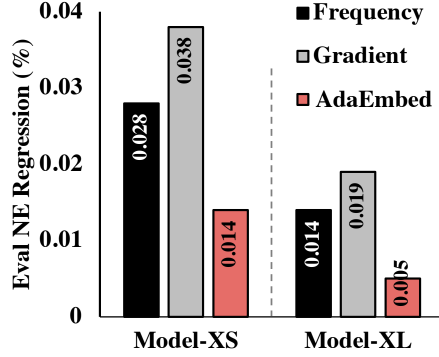



Figure 4.21: AdaEmbed outperforms importance alternatives.

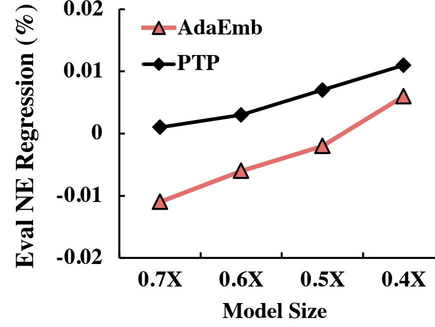



Figure 4.22: AE outperforms post-training pruning (PTP).

combination (i.e., AdaEmbed design) as the embedding importance. We notice our frequency-gradient combination outperforms the alternatives. We note that this is consistent with the results of our Pearson analysis too, i.e., their combination has a stronger correlation to final embedding weights (Figure 4.8(b)). Instead, the access frequency and gradient only consider the data distribution and model characteristics, respectively, while DLRM accuracy depends on both aspects.

**In-training vs. post-training pruning** We compare AdaEmbed to its post-training pruning (PTP) counterpart like [115]. After model training is complete, PTP reduces the embedding size by pruning less important embeddings, as measured by our importance design. In fact, deploying PTP in real is often impractical (e.g., due to the need for online learning), and cannot achieve memory savings and/or QPS improvement during model training. Moreover, Figure 4.22 reports that AdaEmbed (i.e., in-training embedding pruning) can achieve better NE than PTP under the same embedding size, as the in-training design can adapt to the model performance at runtime and continuously optimize embeddings.

## 4.8 Related Work

**Deep Learning Recommendation Systems** Existing systems primarily focus on accelerating DLRM execution. NEO [111] co-optimizes embedding sharding and data parallelism. AIBox [290] and HierPS [289] overlap training execution on CPUs (using solid-state drives) and GPUs. Ekko [229] accelerates DLRM training over wide-area networks. TT-Rec [270] replaces embedding tables with matrix products to reduce memory footprints. Check-N-Run [88] reduces the bandwidth consumption for model checkpoints. Fleche [261] and Kraken [262] share the idea of sharing the weight table across features, but they focus on caching frequently accessed embeddings. AdaEmbed goes one step further by identifying the heterogeneous embedding importance to improve model accuracy during model training.

**Optimizations for Deep Learning** Recent ML advances have proposed various innovations for deep learning. TASO [140] and PET [246] perform tensor optimizations to improve model computation. Superneurons [250] and PipeSwitch [46] optimize instantaneous GPU memory by prefetching model layers based on their computation order. Similarly, ByteScheduler [210] and BytePS [144] accelerate the communication of distributed DNN training. ModelKeeper [159] warms up model training to reduce the amount of training execution needed. Egeria [251] adaptively freezes the training of model layers and bypasses their computation. These existing works focus primarily on conventional models, whereas DLRM models are often bottlenecked by memory-intensive embeddings. Moreover, AdaEmbed is complementary to these efforts as AdaEmbed can further improve their optimized DLRM models.

**Model Pruning** Model pruning has been extensively studied to reduce model computation during training [85, 176], or to generate smaller models after training completes [225, 62]. Importance sampling [163, 153] performs weighted sampling on training data to achieve faster training convergence. Existing pruning systems and theories primarily focus on conventional CV and/or NLP counterparts by pruning only the dense layers [85, 204, 115]. However, in DLRMs, the gigantic embedding tables have become the bottleneck. This difference introduces novel challenges since the dense layers and embedding tables are distinct components with unique characteristics. For instance, dense layers are shared and accessed by all input samples, whereas each embedding row corresponds to a specific feature instance and is only accessed by it, leading to the heterogeneous importance of embeddings. Therefore, existing solutions are ill-suited for DLRMs.

## 4.9 Summary

This chapter introduces AdaEmbed, an in-training embedding pruning system for better DLRM accuracy. AdaEmbed identifies embedding rows with larger importance to model accuracy, and then adaptively prunes less important embeddings to cap the total embedding size at scale. Our evaluations demonstrate that AdaEmbed can reduce manual efforts by automatically learning to use better per-feature embeddings, whereby it saves 35-60% embedding size needed in deployment, and achieves noticeable improvements on model accuracy and model execution speed.

# CHAPTER 5

## Minimizing Data Collection via Enabling Cross-Device Federated Learning

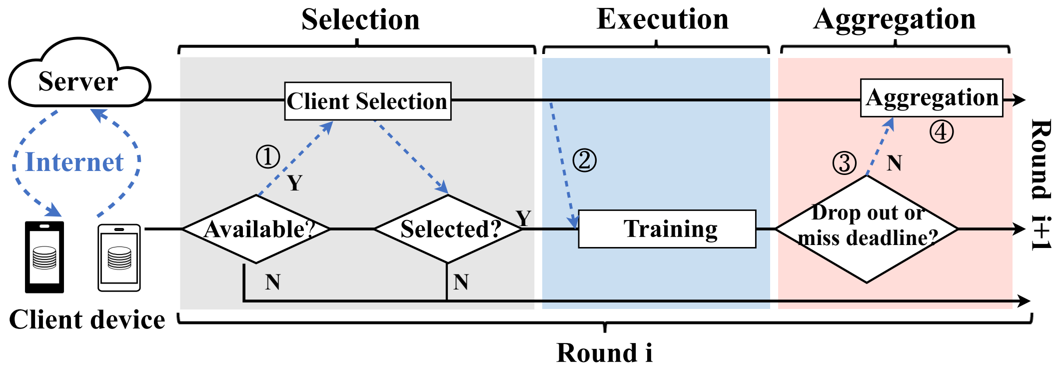
While the previous three chapters zeroed in on minimizing ML resource demands in cloud or multi-cloud environments, the proliferation of end-user devices and applications makes the traditional way to improve cloud ML, by collecting real-life data from the edge, prohibitively expensive. Exploding data volumes and tightening regulations regarding privacy concerns further exacerbate this data collection process. To *minimize the cost of data collection*, machine learning today is experiencing a paradigm shift from cloud ML toward edge-based federated learning (FL) [55, 148].

Though cloud ML and FL share similar model architectures and goals (e.g., less training time and better model accuracy), the underlying model training and testing in FL are orchestrated by a logically centralized coordinator, spanning across potentially millions of devices in the wild [55, 148]. Understandably, this presents new fundamental challenges due to the heterogeneity and dynamics of client system performance and data distribution, as well as the massive scale of clients. All these raise important questions: What are the key characteristics of this new workload? And how can we develop a platform to effectively evaluate and deploy FL efforts?

Starting from this chapter, we extend today's ML training stage and deployment stage up to the global scale. Herein, we first study these questions through a comprehensive analysis of realistic FL system and data traces (Section 5.3). Then, we develop an easily deployable platform designed to simplify FL deployment (Section 5.4).

### 5.1 Introduction

Federated learning (FL) is an emerging ML setting that enables model training and evaluation on end-user data, while circumventing high costs and privacy risks in gathering the raw data from clients (Figure 5.1). Large companies such as Google and Apple deploy FL for computer vision (CV) and natural language processing (NLP) tasks across user devices [83, 269, 7, 117]; NVIDIA applies FL to create medical imaging AI [171]; smart cities perform in-situ image training and testing on AI cameras to avoid expensive data migration [127, 143, 179]; and video streaming and




① Devices check-in with server; then server selects a subset of clients

② Model and configuration are sent to selected devices

③ On-device training is performed; then model update is reported back if training succeeds

④ Server aggregates updates into the global model; then training moves to next round

Figure 5.1: Standard FL protocol.

networking communities use FL to interpret and react to network conditions [29, 264].

In the presence of heterogeneous execution speeds of client devices as well as non-IID data distributions, existing efforts have focused on optimizing different aspects of FL: (1) *System efficiency*: reducing computation load (e.g., using smaller models [224]) or communication traffic (e.g., local SGD [187]) to achieve shorter round duration; (2) *Statistical efficiency*: designing data heterogeneity-aware algorithms (e.g., client clustering [99]) to obtain better training accuracy with fewer training rounds; (3) *Privacy and security*: developing reliable strategies (e.g., differentially private training [147]) to make FL more privacy-preserving and robust to potential attacks.

A comprehensive benchmark to evaluate an FL solution must investigate its behavior under the practical FL setting with (1) *data heterogeneity* and (2) *device heterogeneity* under (3) *heterogeneous connectivity* and (4) *availability* conditions at (5) *multiple scales* on a (6) *broad variety of ML tasks*. While the first two aspects are oft-mentioned in the literature [169], realistic network connectivity and the availability of client devices can affect both types of heterogeneity (e.g., distribution drift [87]), impairing model convergence. Similarly, evaluation at a large scale can expose an algorithm's robustness, as practical FL deployment often runs across thousands of concurrent participants out of millions of clients [269]. Overlooking any one aspect can mislead FL evaluation (§5.2).

Unfortunately, existing FL benchmarks often fall short across multiple dimensions (Table 5.1). First, they are limited in the versatility of data for various real-world FL applications. Indeed, even though they may have quite a few datasets and FL training tasks (e.g., LEAF [59]), their datasets often contain synthetically generated partitions derived from conventional datasets (e.g., CIFAR) and do not represent realistic characteristics. This is because these benchmarks are mostly borrowed

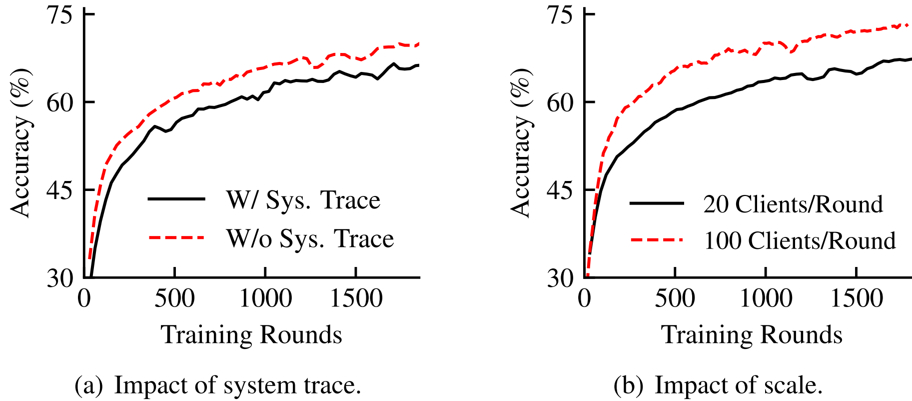
| Features | LEAF | TFF | Flower | **FedScale** |
|---|---|---|---|---|
| Heter. Client Dataset | ◯ | ✗ | ◯ | ✔ |
| Heter. System Speed | ✗ | ✗ | ◯ | ✔ |
| Client Availability | ✗ | ✗ | ✗ | ✔ |
| Scalable Platform | ✗ | ✔ | ✔ | ✔ |
| Real FL Runtime | ✗ | ✗ | ✗ | ✔ |
| Flexible APIs | ✗ | ✔ | ✔ | ✔ |

Table 5.1: Comparing FedScale with existing FL benchmarks and libraries. ◯ implies limited support.

from traditional ML benchmarks (e.g., MLPerf [185]) or designed for simulated FL environments like TensorFlow Federated (TFF) [31] or PySyft [23]. Second, existing benchmarks often overlook system speed, connectivity, and availability of the clients (e.g., Flower [50]). This discourages FL efforts from considering system efficiency and leads to overly optimistic statistical performance (§5.2). Third, their datasets are primarily small-scale, because their experimental environments are unable to emulate large-scale FL deployments. While real FL often involves thousands of participants in each training round [148, 269], most existing benchmarking platforms can merely support the training of tens of participants per round. Finally, most of them lack user-friendly APIs for automated integration, resulting in great engineering efforts for benchmarking at scale. We attach a detailed comparison of existing benchmarks against FedScale in Appendix C.3.

**Contributions** We introduce an FL benchmark and accompanying runtime, FedScale, to enable comprehensive and standardized FL evaluations:

- To the best of our knowledge, FedScale presents the most comprehensive collection of FL datasets for evaluating different aspects of real FL deployments. It currently has 20 realistic FL datasets with small, medium, and large scales for a wide variety of task categories, such as image classification, object detection, word prediction, speech recognition, and reinforcement learning. To account for practical client behaviors, we include real-world measurements of mobile devices and associate each client with their computation and communication speeds, as well as the availability status over time.
- We present an automated evaluation platform, FedScale Runtime, to simplify and standardize FL evaluation in more realistic settings. FedScale Runtime provides a mobile backend to enable on-device FL evaluation and a cluster backend to benchmark various practical FL metrics (e.g., real client round duration) on GPUs/CPUs using real FL statistical and system datasets. The cluster backend can efficiently train thousands of clients in each round on a handful of GPUs.




(a) Impact of system trace.

(b) Impact of scale.

Figure 5.2: Existing benchmarks can be misleading. We train ShuffleNet on OpenImage classification (Detailed setup in Section 5.5).

FedScale Runtime is also extensible, allowing easy deployment of new algorithms and ideas with flexible APIs.

- We perform systematic experiments to show how FedScale facilitates comprehensive FL benchmarking and highlight the pressing need for co-optimizing system and statistical efficiency, especially in tackling system stragglers, accuracy bias, and device energy trade-offs.

## 5.2 Background

**Existing efforts toward practical FL** To tackle heterogeneous client data, FedProx [169], FedYoGi [216] and Scaffold [150] introduce adaptive client/server optimizations that use control variates to account for the 'drift' in model updates. Instead of training a single global model, some efforts enforce guided client selection [163], train a mixture of models [227, 93], or cluster clients over training [99]; To tackle the scarce and heterogeneous device resource [161], FedAvg [187] reduces communication cost by performing multiple local SGD steps, while some works compress the model update by filtering out or quantizing unimportant parameters [222, 151]; After realizing the privacy risk in FL [97, 247], DP-SGD [98] enhances the privacy by employing differential privacy, and DP-FTRL [147] applies the tree aggregation to add noise to the sum of mini-batch gradients. These FL efforts often navigate accuracy-computation-privacy trade-offs. As such, a realistic FL setting is crucial for comprehensive evaluations.

**Existing FL benchmarks can be misleading** Existing benchmarks often lack realistic client statistical and system behavior datasets and/or fail to reproduce large-scale FL deployments. As a result, they are not only insufficient for benchmarking diverse FL optimizations but can even mislead performance evaluations. For example, (1) As shown in Figure 5.2(a), statistical performance

becomes worse when encountering realistic client behavior (e.g., training failures and availability dynamics), which indicates existing benchmarks that do not have systems traces can produce overly optimistic statistical performance; (2) FL training with hundreds of participants each round performs better than that with tens of participants (Figure 5.2(b)). As such, existing benchmark platforms can under-report FL optimizations as they cannot support the practical FL scale with a large number of participants.

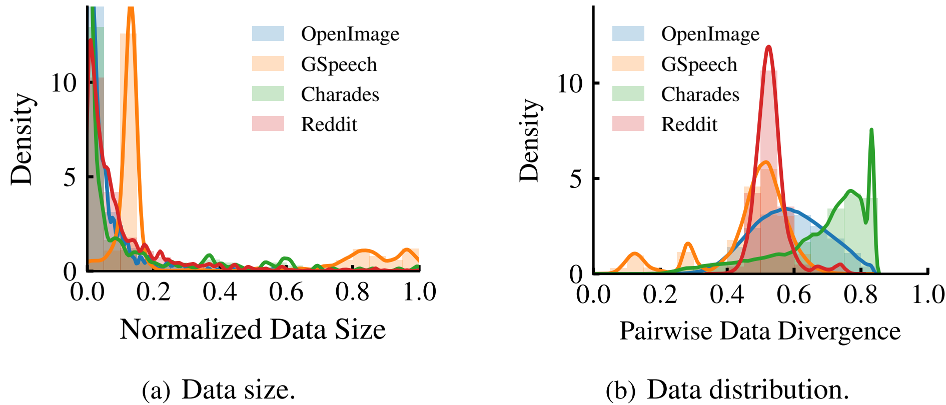
## 5.3 FedScale Dataset: Realistic FL Workloads

We next introduce how we curate realistic datasets in FedScale to fulfill the desired properties of FL datasets.

### 5.3.1 Client Statistical Dataset

| Category | Name | Data Type | #Clients | #Instances | Example Task |
|---|---|---|---|---|---|
| **CV** | *OpenImage* | Image | 13,771 | 1.3M | Classification, Object detection |
| | *Google Landmark* | Image | 43,484 | 3.6M | Classification |
| | *Charades* | Video | 266 | 10K | Action recognition |
| | *VLOG* | Video | 4,900 | 9.6K | Classification, Object detection |
| | *Waymo Motion* | Video | 496,358 | 32.5M | Motion prediction |
| **NLP** | *Europarl* | Text | 27,835 | 1.2M | Text translation |
| | *Reddit* | Text | 1,660,820 | 351M | Word prediction |
| | *LibriTTS* | Text | 2,456 | 37K | Text to speech |
| | *Google Speech* | Audio | 2,618 | 105K | Speech recognition |
| | *Common Voice* | Audio | 12,976 | 1.1M | Speech recognition |
| **Misc ML** | *Taobao* | Text | 182,806 | 20.9M | Recommendation |
| | *Puffer Streaming* | Text | 121,551 | 15.4M | Sequence prediction |
| | *Fox Go* | Text | 150,333 | 4.9M | Reinforcement learning |

Table 5.2: Statistics of *partial* FedScale datasets (the full list with more details is available in Appendix C.2). Currently, FedScale has 20 real-world federated datasets; each dataset is partitioned by its real client-data mapping, and we have removed sensitive information in these datasets.




(a) Data size.

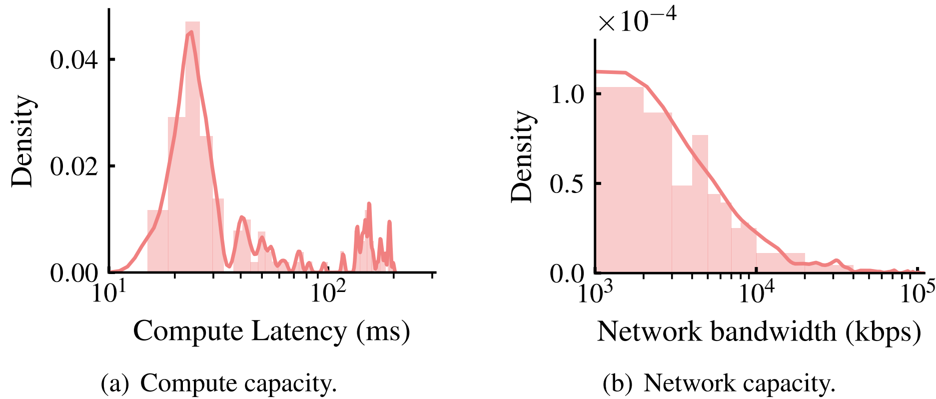
(b) Data distribution.

Figure 5.3: Non-IID client data.

FedScale currently has 20 realistic FL datasets (Table 6.1) across diverse practical FL scenarios and disciplines. For example, the Puffer dataset [264] is from FL video streaming deployed to edge users over the Internet. The raw data of FedScale datasets is collected from different sources and stored in various formats. We clean up the raw data, partition them into new FL datasets, streamline new datasets into consistent formats, and categorize them into different FL use cases. Moreover, FedScale provides standardized APIs, a Python package, for the user to easily leverage these datasets (e.g., using different distributions of the same data or new datasets) in other frameworks.

**Realistic data and partitions** We target realistic datasets with client information, and partition the raw dataset using the unique client identification. For example, OpenImage is a vision dataset collected by Flickr, wherein different mobile users upload their images to the cloud for public use. We use the *AuthorProfileUrl* attribute of the OpenImage data to map data instances to each client, whereby we extract the realistic distribution of the raw data. Following the practical FL deployments [269], we assign the clients of each dataset into the training, validation, and testing groups, to get its training, validation, and testing set. Here, we pick four real-world datasets – video (Charades), audio (Google Speech), image (OpenImage), and text (Reddit) – to illustrate practical FL characteristics. Each dataset consists of hundreds or up to millions of clients and millions of data points. Figure 5.3 reports the *Probability Density Function* (PDF) of the data distribution, wherein we see a high statistical deviation (e.g., wide distribution of the density) across clients not only in the quantity of samples (Figure 5.3(a)) but also in the data distribution (Figure 5.3(b)).[1] We notice that realistic datasets mostly have unique Non-IID patterns, implying the impracticality of existing artificial FL partitions.

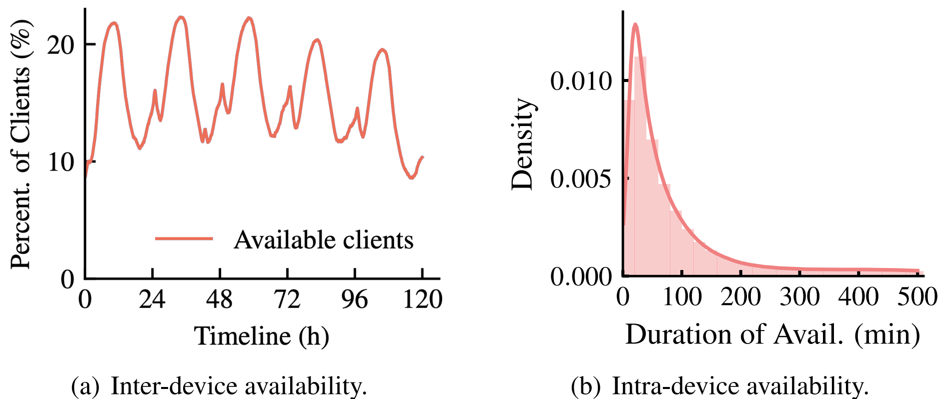
[1]We report the pairwise Jensen–Shannon distance of the label distribution between two clients.




(a) Compute capacity.

(b) Network capacity.

Figure 5.4: Heterogeneous client system speed.




(a) Inter-device availability.

(b) Intra-device availability.

Figure 5.5: Client availability is dynamic.

**Different scales across diverse task categories** To accommodate diverse scenarios in practical FL, FedScale includes small-, medium-, and large-scale datasets across a wide range of tasks, from hundreds to millions of clients. Some datasets can be applied in different tasks, as we enrich their use cases by deriving different metadata from the same raw data. For example, the raw OpenImage dataset can be used for object detection, and we extract each object therein and generate a new dataset for image classification. Moreover, we provide APIs for the developer to customize their dataset (e.g., enforcing new data distribution or taking a subset of clients for evaluations with a smaller scale).

### 5.3.2 Client System Behavior Dataset

**Client device system speed is heterogeneous** We formulate the system trace of different clients using *AI Benchmark* [1] and *MobiPerf Measurements* [19] on mobiles. *AI Benchmark* provides the training and inference speed of diverse models (e.g., MobileNet) across a wide range of device models (e.g., Huawei P40 and Samsung Galaxy S20), while *MobiPerf* has collected the available

cloud-to-edge network throughput of over 100k world-wide mobile clients. As specified in real FL deployments [55, 269], we focus on mobile devices that have larger than 2GB RAM and connect with WiFi; Figure 5.4 reports that their compute and network capacities can exhibit order-of-magnitude differences. As such, how to orchestrate scarce resources and mitigate stragglers is paramount.

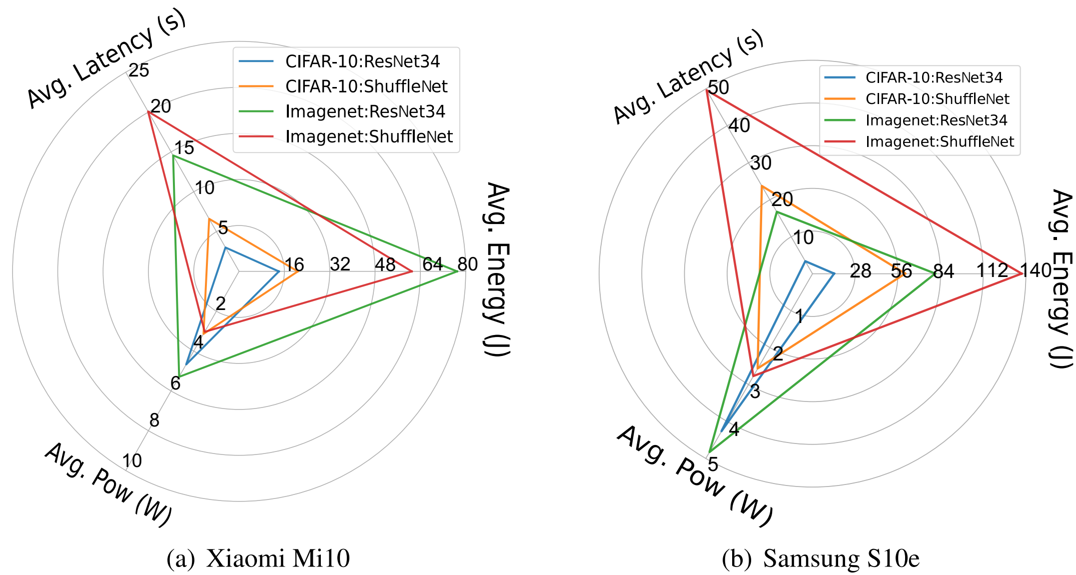
**Client device availability is dynamic** We incorporate a large-scale user behavior dataset spanning across 136k users [268] to emulate the behaviors of clients. It includes 180 million trace items of client devices (e.g., battery charge or screen lock) over a week. We follow the real FL setting, which considers the device in charging to be available [55] and observe great dynamics in their availability: (i) the number of available clients reports diurnal variation (Figure 5.5(a)). This confirms the cyclic patterns in the client data, which can deteriorate the statistical performance of FL [87]. (ii) the duration of each available slot is not long-lasting (Figure 5.5(b)). This highlights the need of handling failures (e.g., clients become offline) during training, as the round duration (also a few minutes) is comparable to that of each available slot. This, however, is largely overlooked in the literature.

## 5.4 FedScale Runtime: Execution Platform

Existing FL evaluation platforms are poor at reproducing practical, large-scale FL deployment scenarios. Worse, they often lack user-friendly APIs and require significant developer effort to introduce new plugins. We introduce, FedScale Runtime, an automated, extensible, and easily-deployable evaluation platform equipped with mobile and cluster backends, to simplify and standardize FL evaluation under realistic settings.

### 5.4.1 FedScale Runtime: Mobile Backend

FedScale Runtime deploys a mobile backend to enable on-device FL evaluation on smartphones. The first principle in building our mobile backend is to minimize any engineering effort for the developer (e.g., without reinventing their Python code) to benchmark FL on mobiles. To this end, FedScale mobile backend [230] is built atop the Termux app [32], an Android terminal that supports Linux environment.




(a) Xiaomi Mi10 (b) Samsung S10e

Figure 5.7: FedScale Runtime can benchmark the mobile runtime of power, energy, and latency. We train Resnet34 and Shufflenet on ImageNet and CIFAR-10 on Xiaomi mi10 and Samsung S10e.

```
from fedscale.core.client import Client

class Mobile_Client(Client):
  def train(self,client_data,model,conf):
    for local_step in range(conf.local_steps):
      optimizer.zero_grad()
      ...
      loss.backward()
      optimizer.step()

    # Results will be sent to cloud aggregator via gRPC
    return gradient_update
```

Figure 5.6: Training on mobile client.

Figure 5.6 shows a snippet of code running on FedScale mobile backend. By integrating with Termux, FedScale Runtime allows the developer to run an unmodified version of Python script (e.g., PyTorch) built from source on the mobile device; the full-operator set (e.g., PyTorch modules) is available too. This speeds up the deployment cycle: FL models and algorithms that were prototyped on server GPUs/CPUs can also be deployed using FedScale Runtime. We are currently implementing the Google Remote Procedure Call (gRPC) for distributed mobile devices to interface with FedScale Runtime cloud server.

**Benchmarking the mobile backend** FedScale mobile backend enables developers to benchmark realistic FL training/testing performance on mobile phones. For example, Figure 5.7 reports the

performance metrics of training ShuffleNet and ResNet34 on one mini-batch (batch size 32), drawn from the ImageNet and CIFAR-10 datasets, on Xiaomi Mi10 and Samsung S10e Android devices. We benchmark the average training time. We notice that ResNet34 runs at higher instantaneous power than ShuffleNet on both devices, but it requires less total energy to train since it takes shorter latency. ImageNet takes longer than CIFAR-10 per mini-batch, as the larger training image sizes lead to longer execution. The heterogeneity in computational capacity is evident as the Xiaomi Mi10 device outperforms the Samsung S10e device due to a more capable processor. As such, we believe that FedScale mobile backend can facilitate future on-device FL optimizations (e.g., hardware-aware neural architecture search [122]).

### 5.4.2 FedScale Runtime: Cluster Backend

FedScale Runtime provides an automated cluster backend that can support FL evaluations in real deployments and in-cluster simulations. In the *deployment mode*, FedScale Runtime acts as the cloud aggregator and orchestrates FL executions across real devices (e.g., laptops, mobiles, or even cloud servers). To enable cost-efficient FL benchmarking, FedScale Runtime also includes a *simulation mode* that performs FL training/testing on GPUs/CPUs, while providing various practical FL metrics by emulating realistic FL behaviors, such as computation/communication cost, latency, and wall clock time. To the best of our knowledge, FedScale Runtime is the first platform that enables FL benchmarking with practical FL runtime on GPUs/CPUs. More importantly, FedScale Runtime can run the same code with a few changes in both modes to minimize the migration overhead.

Throughout the rest of the chapter, we focus on the simulation mode as benchmarking is the primary focus of this chapter.

| Module | API Name | Example Use Case |
|---|---|---|
| **Aggregator** | `round_completion_handler()` | Adaptive/secure aggregation |
| **Simulator** | `client_completion_handler()` | Straggler mitigation |
| **Client** | `select_participants()` | Client selection |
| **Manager** | `select_model_for_client()` | Adaptive model selection |
| **Client** | `train()` | Local SGD/malicious attack |
| **Simulator** | `serialize_results()` | Model compression |

Table 5.3: Some example APIs. FedScale provides APIs to deploy new plugins for various designs. We omit input arguments for brevity here.

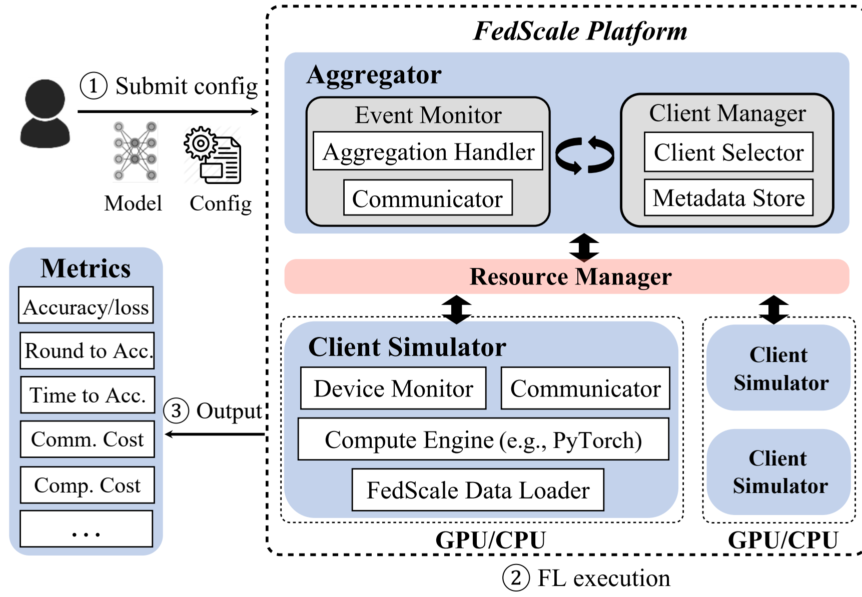



Figure 5.8: FedScale Runtime enables the developer to benchmark various FL efforts with practical FL data and metrics.

```
1  from fedscale.core.client import Client
2
3  class Customized_Client(Client):
4  # Redefine training (e.g., for local SGD/gradient compression)
5
6    def train(self,client_data,model,conf):
7      # Code of plugin
8          ...
9
10     # Results will be serialized, and then sent to aggregator
11     return training_result
```

Figure 5.9: Add plugins by inheritance.

**Overview of the simulation mode** FedScale Runtime consists of three primary components (Figure 5.8):

- *Aggregator Simulator*: It acts as the aggregator in practical FL, which selects participants, distributes execution profiles (e.g., model weight), and handles result (e.g., model updates) aggregation. In each round, its client manager uses the client behavior trace to monitor whether a client is available; then it selects the specified number of clients to participate in that round. Once receiving new events, the event monitor activates the handler (e.g., aggregation handler to perform model aggregation), or the gRPC communicator to send/receive messages. The communicator records the size (cost) of every network traffic, and its runtime latency in FL wall-clock time ($\frac{traffic_size}{client_bandwidth}$).

- *Client Simulator*: It works as an FL client. FedScale data loader loads the federated dataset of that client and feeds this data to the compute engine to run real training/testing. The computation latency is determined by ($\#_processed_sample \times latency_per_sample$), and the communicator handles network traffic and records the communication latency ($\frac{traffic_size}{client_bandwidth}$). The device monitor will terminate the simulation of a client if the current FL runtime exceeds his available slot (e.g., client drops out), as indicated by the availability trace.
- *Resource Manager*: It orchestrates the available physical resources for evaluation to maximize resource utilization. For example, when the number of participants/round exceeds the resource capacity (e.g., simulating thousands of clients on a few GPUs), the resource manager queues the overcommitted tasks of clients and schedules new client simulation from this queue once resources become available. Note that this queuing will not affect the simulated FL runtime, as this runtime is controlled by a global virtual clock, and the event monitor will manage events in the correct runtime order.

Note that capturing runtime performance (e.g., wall clock time) is rather slow and expensive in practical FL – each mobile device takes several minutes to train a round – but our simulator enables *fast-forward* simulation, as training on CPUs/GPUs takes only a few seconds per round, while providing simulated runtime using realistic traces.

**FedScale Runtime enables automated FL simulation** FedScale Runtime incorporates realistic FL traces, using the aforementioned trace by default or the developer-specified profile from the mobile backend, to automatically emulate the practical FL workflow: ① *Task submission*: FL developers specify their configurations (e.g., model and dataset), which can be federated training or testing, and the resource manager will initiate the aggregator and client simulator on available resources (GPU, CPU, other accelerators, or even smartphones); ② *FL simulation*: Following the standardized FL lifecycle (Figure 5.1), in each training round, the aggregator inquires the client manager to select participants, whereby the resource manager distributes the client configuration to the available client simulators. After the completion of each client, the client simulator pushes the model update to the aggregator, which then performs the model aggregation. ③ *Metrics output*: During training, the developer can query the practical evaluation metrics on the fly. Figure 5.8 lists some popular metrics in FedScale.

**FedScale Runtime is easily deployable and extensible with plugins** FedScale Runtime provides flexible APIs, which can accommodate with different execution backends (e.g., PyTorch) by design, for the developer to quickly benchmark new plugins. Table 5.3 illustrates some example APIs that can facilitate diverse FL efforts, and Figure 5.9 dictates an example showing how these APIs help to benchmark a new design of local client training with a few lines of code by inheriting the

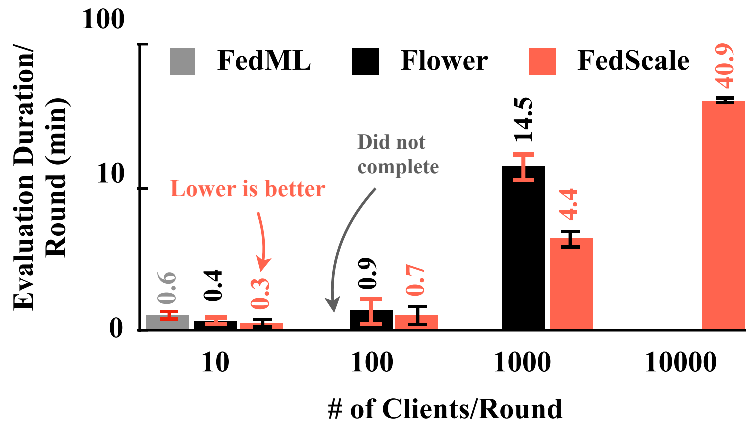



Figure 5.10: FedScale Runtime can run thousands of clients/round on 10 GPUs where others fail.[3] More results are in Appendix C.3.

base Client module. Moreover, FedScale Runtime can embrace new realistic (statistical client or system behavior) datasets with the built-in APIs. For example, the developer can import his own dataset of the client availability with the API (`load_client_availability`), and FedScale Runtime will automatically enforce this trace during evaluations. We provide more examples and a comparison with other frameworks, in Appendix C.4 to show the ease of evaluating various today's FL work in FedScale – a few lines are all we need [163, 165]!

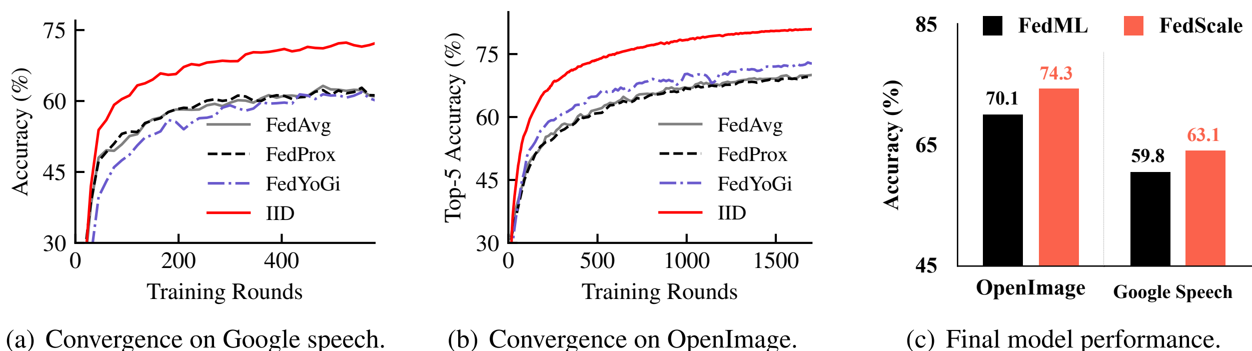
**FedScale Runtime is scalable and efficient** In the simulation mode, FedScale Runtime can perform large-scale simulations (thousands of clients per round) in both standalone (single CPU/GPU) and distributed (multiple machines) settings. This is because: (1) FedScale Runtime uses GPU sharing techniques [276] to divide GPU among tasks so that multiple client simulators can co-locate on the same GPU; (2) our resource manager monitors the fine-grained resource utilization of machines, queues the overcommitted execution requests, adaptively dispatches requests of the client across machines to achieve load balance, and then orchestrates the simulation based on the client virtual clock. Instead, state-of-the-art platforms can hardly support the practical FL scale, due to their limited support for distributed evaluations (e.g., FedJax [219]), and/or the reliance on the traditional ML architecture that trains on a few workers with long-running computation, whereas FedScale Runtime minimizes the overhead (e.g., frequent data serialization) in the fleet training of FL clients. As shown in Figure 5.10 [4], other than being able to evaluate the practical FL runtime, FedScale Runtime not only runs faster than FedML [119] and Flower [50], but it can support large-scale evaluations efficiently.

[3] We finished these evaluations in Dec. 2021.

[4] We train ShuffleNet on OpenImage classification task on 10 GPU nodes. Detailed experimental setups are available in Appendix C.1.

| Task | Dataset | Model | IID | FedAvg | FedProx | FedYoGi |
|---|---|---|---|---|---|---|
| Image Classification | FEMNIST | ResNet-18 | 86.40% | **78.50%** | 78.40% | 76.30% |
| | OpenImage | ShuffleNet-V2 | 81.37% | 70.27% | 69.54% | **74.04%** |
| | | MobileNet-V2 | 80.83% | 70.09% | 70.34% | **75.25%** |
| Text Classification | Amazon Review | Logistic Regression | 66.10% | **65.80%** | 65.10% | 65.30% |
| Language Modeling | Reddit | Albert | 73.5 ppl | 77.3 ppl | **76.6 ppl** | 81.6 ppl |
| Speech Recognition | Google Speech | ResNet-34 | 72.58% | **63.37%** | 63.25% | 62.67% |

Table 5.4: Benchmarking of different FL algorithms across realistic FL datasets. We report the mean test accuracy over 5 runs.




(a) Convergence on Google speech. (b) Convergence on OpenImage. (c) Final model performance.

Figure 5.11: FedScale can benchmark the statistical FL performance. (c) shows existing benchmarks can under-report the FedYoGi performance as they cannot support a large number of participants.

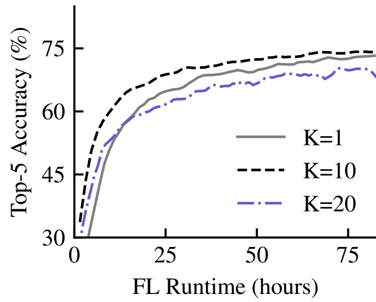
## 5.5 Experiments

In this section, we show how FedScale can facilitate better benchmarking of FL efforts over its counterparts.

**Experimental setup** We use 10 NVIDIA Tesla P100 GPUs in our evaluations. Following the real FL deployments [55, 269], the aggregator collects updates from the first $N$ completed participants out of $1.3N$ participants to mitigate system stragglers in each round, and $N = 100$ by default. We experiment with representative FedScale datasets in different scales and tasks (detailed experiment setup in Appendix C.1).

### 5.5.1 How Does FedScale Help FL Benchmarking?

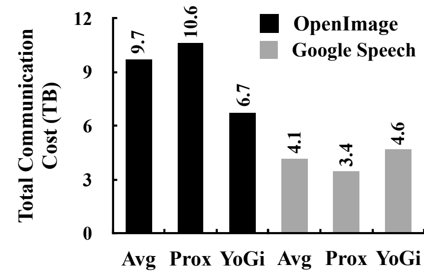
Existing benchmarks are insufficient to evaluate the various metrics needed in today's FL. We note that the performance of existing benchmarks and FedScale are quite close in the same settings if we turn off the optional system traces in FedScale. Because the underlying training and FL




(a) FedScale reports realistic FL clock.

| Setting | FL runtime (hours) | Evaluation time (hours) |
|---|---|---|
| K=1 | 97 | 17 |
| K=10 | 52 | 9 |
| K=20 | 130 | 47 |

(b) FedScale enables fast-forward eval.




(c) FedScale reports FL communication cost.

Figure 5.12: FedScale can benchmark realistic FL runtime. (a) and (b) report FedYoGi results on OpenImage with different number of local steps (K); (b) reports the FL runtime to reach convergence.

protocols in evaluations are the same. However, the limited scalability can mislead the practical FL performance. Next, we show the effectiveness of FedScale in benchmarking different FL aspects over its counterparts.

**Benchmarking FL statistical efficiency.** FedScale provides various realistic client datasets to benchmark the FL statistical efficiency. Here, we experiment with state-of-the-art optimizations (FedAvg, FedProx and FedYoGi) – each aims to mitigate the data heterogeneity – and the traditional IID data setting. Figure 5.11 and Table 5.4 report that: (1) the round-to-accuracy performance and final model accuracy of the non-IID setting is worse than that of the IID setting, which is consistent with existing findings [148]; (2) different tasks can have different preferences on the optimizations. For example, FedYoGi performs the best on OpenImage, but it is inferior to FedAvg on Google Speech. With much more FL datasets, FedScale enables extensive studies of the sweet spot of different optimizations; and (3) existing benchmarks can under-report the FL performance due to their inability to reproduce the FL setting. Figure 5.11(c) reports the final model accuracy using FedML and FedScale, where we attempt to reproduce the scale of practical FL with 100 participants per round in both frameworks, but FedML can only support 30 participants because of its suboptimal scalability, which under-reports the FL performance that the algorithm can indeed achieve.

**Benchmarking FL system efficiency.** Existing system optimizations for FL focus on the practical runtime (e.g., wall-clock time in real FL training) and the FL execution cost. Unfortunately, existing benchmarks can hardly evaluate the FL runtime due to the lack of realistic system traces, but we now show how FedScale can help such benchmarking: (1) FedScale Runtime enables fast-forward evaluations of the practical FL wall-clock time with fewer evaluation hours. Taking different number

of local steps $K$ in local SGD as an example [187], Figure 5.12(a) and Table 5.12(b) illustrate that FedScale can evaluate this impact of $K$ on practical FL runtime in a few hours. This allows the developer to evaluate large-scale system optimizations efficiently; and (2) FedScale Runtime can dictate the FL execution cost by using realistic system traces. For example, Figure 5.12(c) reports the practical FL communication cost in achieving the performance of Figure 5.11, while Figure 5.15 reports the system duration of individual clients. These system metrics can facilitate developers to navigate the accuracy-cost trade-off.

**Benchmarking FL privacy and security.** FedScale can evaluate the statistical and system efficiency for privacy and security optimizations more realistically. Here, we give an example of benchmarking DP-SGD [98, 147], which applies differential privacy to improve the client privacy. We experiment with different privacy targets $\sigma$ ($\sigma$=0 indicates no privacy enhancement) and different number of participants per round $N$. Figure 5.13 shows that the scale of participants (e.g., $N$=30) that today's benchmarks can support can mislead the privacy evaluations too: for $\sigma$=0.01, while we notice great performance degradation (12.8%) in the final model accuracy when $N$=30, this enhancement is viable in practical FL ($N$=100) with decent accuracy drop (4.6%). Moreover, FedScale is able to benchmark more practical FL metrics, such as wall-clock time, communication cost added in privacy optimizations, and the number of rounds needed to leak the client privacy under realistic individual client data and Non-IID distributions.

As for benchmarking FL security, we follow the example setting of recent backdoor attacks [232, 247] on the OpenImage, where corrupted clients flip their ground-truth labels to poison the training. We benchmarked two settings: one without security enhancement, while the other clips the model updates as [232]. As shown in Figure 5.14, while state-of-the-art optimizations report this can mitigate the attacks without hurting the overall performance on their synthesized datasets, large accuracy drops can occur in more practical FL settings.

### 5.5.2 Opportunities for Future FL Optimizations

Next, we show FedScale can shine light on the need for yet unexplored optimizations owing to its realistic FL settings.

**Heterogeneity-aware co-optimizations of communication and computation** Existing optimizations for the system efficiency often apply the same strategy on all clients (e.g., using the same number of local steps [187] or compression threshold [222]), while ignoring the heterogeneous client system speed. When we outline the timeline of 5 randomly picked participants in training of the ShuffleNet (Figure 5.15), we find: (1) system stragglers can greatly slow down the round aggregation in practical FL; and (2) simply optimizing the communication or computation efficiency

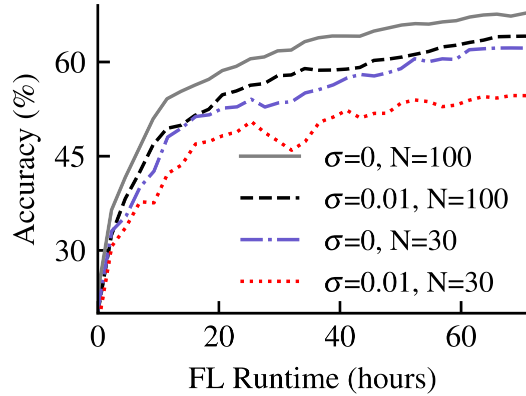



Figure 5.13: FedScale can benchmark privacy efforts in more realistic FL settings.

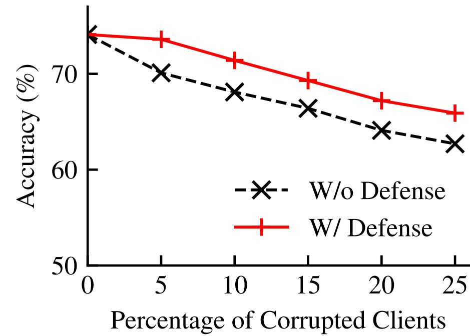



Figure 5.14: FedScale can benchmark security optimizations with realistic FL data.

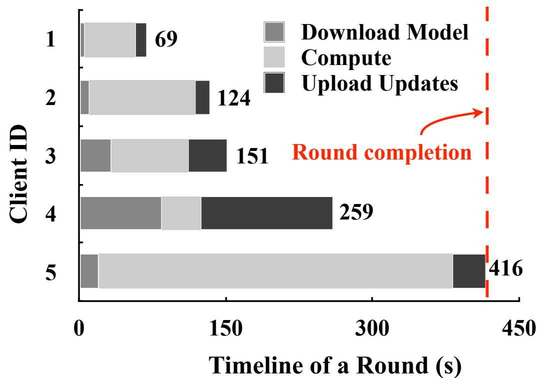



Figure 5.15: System stragglers slow down practical FL greatly.

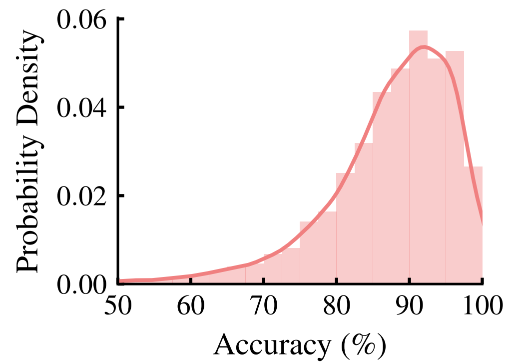



Figure 5.16: Biased accuracy distributions of the trained model across clients (Shufflenet on OpenImage).

may not lead to faster rounds, as the last participant can be bottlenecked by the other resource. Here, optimizing the communication can greatly benefit *Client 4*, but it achieves marginal improvement on the round duration as *Client 5* is computation-bound. This implies an urgent need of heterogeneity-aware co-optimizations of communication and computation efficiency.

**Co-optimizations of statistical and system efficiency** Most of today's FL efforts focus on either optimizing the statistical or the system efficiency, whereas we observe a great opportunity to jointly optimize both efficiencies: (1) As the system behavior determines the availability of client data, predictable system performance can benefit statistical efficiency. For example, in alleviating the biased model accuracy (Figure 5.16), we may prioritize the use of upcoming offline clients to curb the upcoming distribution drift of client data; (2) Statistical optimizations should be aware of the heterogeneous client system speed. For example, instead of applying one-fit-all strategies (e.g., local steps or gradient compression) for all clients, faster workers can trade more system latency for

more statistical benefits (e.g., transferring more traffics with less intensive compression).

**FL design-decisions considering mobile environment** Existing efforts have largely overlooked the interplay of client devices and training speed (e.g., using a large local steps to save communication [187]), however, as shown in Figure 5.7, running intensive on-device computation for a long time can quickly drain the battery, or even burn the device, leading to the unavailability of clients. Therefore, we believe that a power and temperature-aware training algorithm (e.g., different local steps across clients or device-aware NAS) can be an important open problem.

## 5.6 Summary

To enable scalable and reproducible FL research, we introduce FedScale, a diverse set of realistic FL datasets in terms of scales, task categories, and client system behaviors, along with a scalable and extensible evaluation platform, FedScale Runtime. FedScale Runtime performs fast-forward evaluation of the practical FL runtime metrics needed in today's works. More importantly, FedScale Runtime provides ready-to-use realistic datasets and flexible APIs to allow more FL applications, such as benchmarking NAS, model inference, and a broader view of federated computation (e.g., multi-party computation). FedScale is open-source, and we hereby invite the community to develop and contribute state-of-the-art FL efforts.

# CHAPTER 6

## Minimizing Cloud-Fed ML Accuracy Gap via Guided Participant Selection

In the pursuit of FedScale, we realize a new system component in the FL software stack, i.e., the client manager. This is because as the underlying execution in FL spans across millions of heterogeneous clients, how to orchestrate FL client participation in each execution round can lead to distinct statistical efficiency (e.g., round-to-accuracy performance) and system efficiency (e.g., duration of each round). This chapter propounds the minimalist philosophy – do less to gain more – to minimize the performance gap between cloud ML and federated learning by cherry-picking FL participants. Our approach takes into account both the client's data toward improving model accuracy and its system's capability to run execution efficiently.

We start by analyzing how state-of-the-art FL optimizations can be far from optimal in Section 6.3, due to the deficiency of existing client orchestration strategies. Section 6.4 outlines our system called Oort, which performs guided participant selection by exploring and exploiting those clients who contribute more toward improving model performance (Section 6.5 and Section 6.6). We then evaluate Oort's performance across diverse FL deployment scales and tasks in Section 6.8.

### 6.1 Background

Similar to the life cycle of traditional ML, the FL developer often first prototypes model architectures and hyperparameters with a proxy dataset. After selecting a suitable configuration, she can use federated training to improve model performance by training across a crowd of participants [55, 148]. The wall clock time for training a model to reach an accuracy target (i.e., time-to-accuracy) is still a key performance objective, even though it may take significantly longer than centralized training [148]. To circumvent biased or stale proxy data in hyperparameter tuning [191], to inspect these models being trained, or to validate deployed models after training [264, 263], developers may want to perform federated testing on the real-life client data, wherein enforcing their requirements on the testing set (e.g., $N$ samples for each category or following the representative categorical distribution[1]) is crucial for them to reason about model performance under different data

[1] A categorical distribution is a discrete probability distribution showing how a random variable can take the result from one of $K$ possible categories.

characteristics [57, 191].

Unfortunately, clients may not all be simultaneously available for FL training or testing [148]; they may have heterogeneous data distributions and system capabilities [129, 55]; and including too many may lead to wasted work and suboptimal performance [55] (§6.3). Consequently, a fundamental problem in practical FL is the *selection of a "good" subset of clients as participants*, where each participant locally processes its own data, and only their results are collected and aggregated at a (logically) centralized coordinator.

Existing works optimize for *statistical model efficiency* (i.e., better training accuracy with fewer training rounds) [248, 169, 209, 65] or *system efficiency* (i.e., shorter rounds) [187, 233], while randomly selecting participants. Although random participant selection is easy to deploy, unfortunately, it results in poor performance of federated training because of large heterogeneity in device speed and/or data characteristics. Worse, random participant selection can lead to biased testing sets and loss of confidence in results. As a result, developers often resort to more participants than perhaps needed [249, 191], leading to wasteful FL training and testing.

## 6.2 Solution Outlines

We present Oort for FL developers to enable guided participant selection throughout the life cycle of an FL model (§6.4). Specifically, Oort cherry-picks participants to improve time-to-accuracy performance for federated training, and it enables developers to specify testing criteria for federated model testing. It makes informed participant selection by relying on the information already available in existing FL solutions [148] with little modification.

Selecting participants for federated training is challenging because of the trade-off between heterogeneous system and statistical model utilities both across clients and of any specific client over time (as the trained model changes). First, simply picking clients with high statistical utility can lead to longer training rounds due to the coupled nature of client data and system performance. The challenge is further exacerbated by the large population, as capturing the latest utility of all clients is impractical. As such, we identify clients with high statistical utility, which is measured in terms of their most recent aggregate training loss, adjusted for spatiotemporal variations, and penalize the utility of a client if her system speed is likely to elongate the duration necessary to complete global aggregation. To navigate the sweet spot of jointly maximizing statistical and system efficiency, we adaptively allow for longer training rounds to admit clients with higher statistical utility. We then employ an online exploration-exploitation strategy to probabilistically select participants among high-utility clients for robustness to outliers. Our design can accommodate diverse selection criteria (e.g., fairness), and deliver improvements while respecting privacy (§6.5).

Although FL developers often have well-defined requirements on their testing data, satisfying

these requirements is not straightforward. Similar to traditional ML, developers may request a testing dataset that follows the global distribution to avoid testing on all clients [130, 191]. However, clients' data characteristics in some private FL scenarios may not be available [98, 269]. To preserve the deviation target of participant data from the global, Oort performs participant selection by bounding the number of participants needed. Second, for cases where clients' data characteristics are provided [179], developers can specify specific distribution of the testing set to debug model efficiency (e.g., using balanced distribution) [45, 274]. At scale, satisfying this requirement in FL suffers from large overhead. Therefore, we propose a scalable heuristic to efficiently enforce developer requirements, while optimizing the duration of testing (§6.6).

We have integrated Oort with PySyft (§6.7) and evaluated it across various FL tasks with real-world workloads (§6.8). [2] Compared to the state-of-the-art selection techniques used in today's FL deployments [249, 269, 64], Oort improves time-to-accuracy performance by 1.2×-14.1× and final model accuracy by 1.3%-9.8% for federated model training, while achieving close to upper-bound statistical performance. For federated model testing, Oort can efficiently respond to developer-specified data distribution across millions of clients, and improves the end-to-end testing duration by 4.7× on average over state-of-the-art solutions.

Overall, we make the following contributions in this chapter:

1. We highlight the tension between statistical and systems efficiency when selecting FL participants and present Oort to effectively navigate the tradeoff.
2. We propose participant selection algorithms to improve the time-to-accuracy performance of training and to scalably enforce developers' FL testing criteria.
3. We implement and evaluate these algorithms at scale in Oort, showing both statistical and systems performance improvements over the state-of-the-art.

## 6.3 Motivation

While existing FL solutions have made considerable progress in tackling some of the above challenges (§6.9), they mostly rely on hindsight – given a pool of participants, they optimize model performance [209, 170] or system efficiency [187] to tackle data and system heterogeneity. However, the potential for curbing these disadvantages by cherry-picking participants before execution has largely been overlooked. For example, FL training and testing today still rely on randomly picking participants [55], which leaves large room for improvements.

**Suboptimality in maximizing efficiency.** We first show that today's participant selection underperforms for FL solutions. Here, we train two popular image classification models tailored

[2] Oort is available at https://github.com/SymbioticLab/Oort.

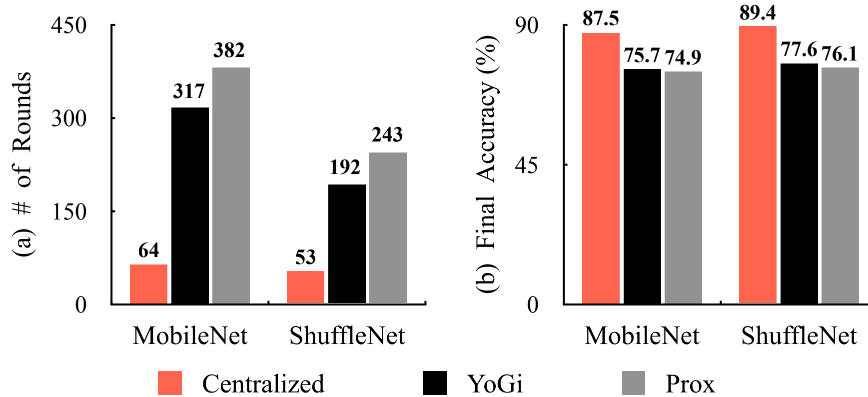


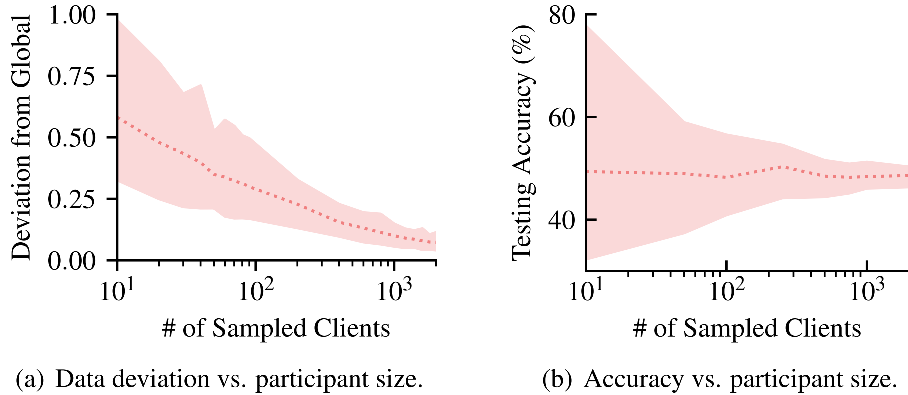

Figure 6.1: Existing works are suboptimal in: (a) round-to-accuracy performance and (b) final model accuracy. (a) reports number of rounds required to reach the highest accuracy of Prox on MobileNet (i.e., 74.9%).

for mobile devices (i.e., MobileNet [224] and ShuffleNet [287]) with 1.6 million images of the OpenImage dataset, and randomly pick 100 participants out of more than 14k clients in each training round. We consider a performance *upper bound* by creating a hypothetical centralized case where images are evenly distributed across only 100 clients, and train on all 100 clients in each round. As shown in Figure 6.1, even with state-of-the-art optimizations, such as YoGi [216] and Prox [169],[3] the round-to-accuracy and final model accuracy are both far from the upper-bound. Moreover, overlooking the system heterogeneity can elongate each round, further exacerbating the suboptimality of time-to-accuracy performance.

**Inability to enforce data selection criteria.** While an FL developer often fine-tunes her model by understanding the input dataset, existing solutions do not provide any systems support for her to express and reason about what data her FL model was trained or tested on. Even worse, existing participant selection not only inflates the execution, but can lead to bias and loss of confidence in results [57, 129].

To better understand how existing works fall short, we take the global categorical distribution as an example requirement, and experiment with the above pre-trained ShuffleNet model. Figure 6.2(a) shows that: (i) even for the same number of participants, random selection can result in noticeable data deviations from the target distribution; (ii) while this deviation decreases as more participants are involved, it is non-trivial to quantify how it varies with different number of participants, even if we ignore the cost of enlarging the participant set. Worse, when even selecting many participants, developers can not enforce other distributions (e.g., balanced distribution for debugging [45]) with random selection. One natural effect of violating developer specification is bias in results

[3]These two adapt traditional stochastic gradient descent algorithms to tackle the heterogeneity of the client datasets.




(a) Data deviation vs. participant size. (b) Accuracy vs. participant size.

Figure 6.2: Participant selection today leads to (a) deviations from developer requirements, and thus (b) affects testing result. Shadow indicates the [min, max] range of y-axis values over 1000 runs given the same x-axis input; each line reports the median.

(Figure 6.2(b)), where we test the accuracy of the same model on these participants. We observe that a biased testing set results in high uncertainties in testing accuracy.

## 6.4 Oort Overview

Oort improves FL training and testing performance by judiciously selecting participants while enabling FL developers to specify data selection criteria. In this section, we provide an overview of how Oort fits in the FL life cycle to help the reader follow the subsequent sections.

### 6.4.1 Architecture

At its core, Oort is a participant selection framework that identifies and cherry-picks valuable participants for FL training and testing. It is located inside the coordinator of an FL framework and interacts with the driver of an FL execution (e.g., PySyft [23] or Tensorflow Federated [31]). Given developer-specified criteria, it responds with a list of participants, whereas the driver is in charge of initiating and managing execution on the Oort-selected remote participants.

Figure 6.3 shows how Oort interacts with the developer and FL execution frameworks. ① *Job submission*: the developer submits and specifies the participant selection criteria to the FL coordinator in the cloud. ② *Participant selection*: the coordinator enquires the clients meeting eligibility properties (e.g., battery level), and forwards their characteristics (e.g., liveness) to Oort. Given the developer requirements (and execution feedbacks in case of training (2a)), Oort selects participants based on the given criteria and notifies the coordinator of this participant selection ((2b)). ③ *Execution*: the coordinator distributes relevant profiles (e.g., model) to these participants, and then each participant independently computes results (e.g., model weights in training) on her data;

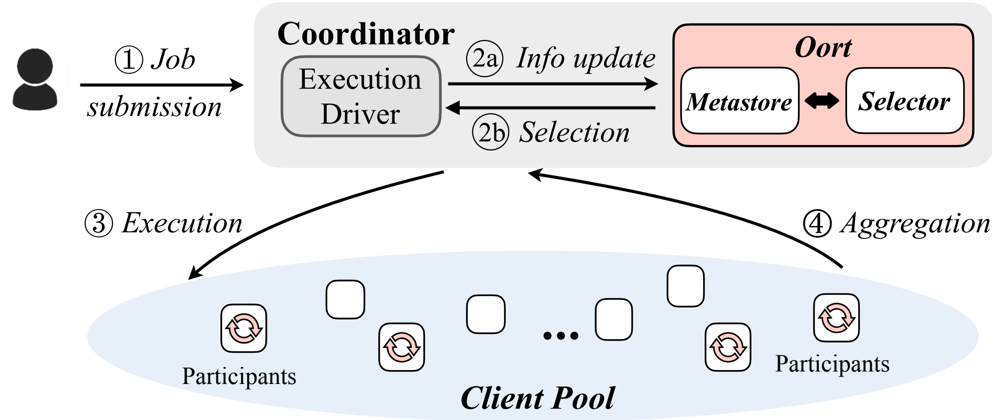



Figure 6.3: Oort architecture. The driver of the FL framework interacts with Oort using a client library.

④ *Aggregation*: when participants complete the computation, the coordinator aggregates updates from participants.

During federated training, where the coordinator initiates the next training round after aggregating updates from enough number of participants [55], it iterates over ②-④ in each round. Every few training rounds, federated testing is often used to detect whether the cut-off accuracy has been reached.

### 6.4.2 Oort Interface

Oort employs two distinct selectors that developers can access via a client library during FL training and testing.

**Training selector.** This selector aims to improve the time-to-accuracy performance of federated training. To this end, it captures the utility of clients in training, and efficiently explores and selects high-utility clients at runtime.

Figure 6.4 presents an example of how FL developers and frameworks interact with Oort during training. In each training round, Oort collects feedbacks from the engine driver, and updates the utility of individual clients (Line 15-17). Thereafter, it cherry-picks high-utility clients to feed the underlying execution (Line 20). We elaborate more on client utility and the selection mechanism in Section 6.5.

**Testing selector.** This selector currently supports two types of selection criteria. When the individual client data characteristics (e.g., categorical distribution) are not provided, the testing selector determines the number of participants needed to cap the data deviation of participants

```
1  import Oort
2
3  def federated_model_training():
4    selector = Oort.create_training_selector(config)
5
6    # Train to target testing accuracy
7    while federated_model_testing() < target:
8
9      # Train 50 rounds before testing
10     for _ in range(50):
11       # Collect feedbacks of last round
12       feedbacks = engine.get_participant_feedback()
13
14       # Update the utility of clients
15       for clientId in feedbacks:
16         selector.update_client_util(
17                     clientId, feedbacks[clientId])
18
19       # Pick 100 high-utility participants
20       participants = selector.select_participant(100)
21       ... # Activate training on remote clients
```

Figure 6.4: Code snippet of Oort interaction during FL training.

from the global. Otherwise, it cherry-picks participants to serve the exact developer-specified requirements on data while minimizing the duration of testing. We elaborate more on selection for federated testing in Section 6.6.

## 6.5 Federated Model Training

In this section, we first outline the trade-off in selecting participants for FL training (§6.5.1), and then describe how Oort quantifies the client utility while respecting privacy (§6.5.2 and §6.5.3), how it selects high-utility clients at scale despite staleness in client utility as training evolves (§6.5.4).

### 6.5.1 Tradeoff Between Statistical and System Efficiency

Time-to-accuracy performance of FL training relies on two aspects: (i) *statistical efficiency*: the number of rounds taken to reach target accuracy; and (ii) *system efficiency*: the duration of each training round. The data stored on the client and the speed with which it can perform training determine its utility with respect to statistical and system efficiency, which we respectively refer to as statistical and system utility.

Due to the coupled nature of client data and system performance, cherry-picking participants for better time-to-accuracy performance requires us to jointly consider both forms of efficiency. We visualize the trade-off between these two with our breakdown experiments on the MobileNet model with OpenImage dataset (§6.8.2.1). As shown in Figure 6.5, while optimizing the system efficiency ("Opt-Sys. Efficiency") can reduce the duration of each round (e.g., picking the fastest clients), it can lead to more rounds than random selection as that client data may have already been

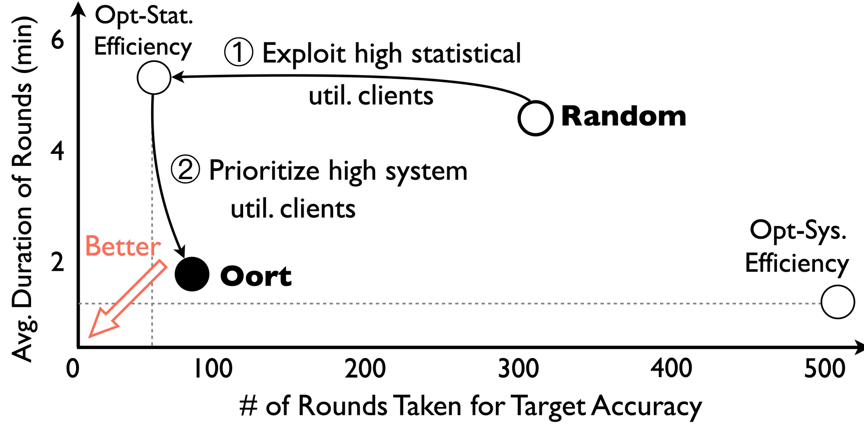



Figure 6.5: Existing FL training randomly selects participants, whereas Oort navigates the sweet point of statistical and system efficiency to optimize their circled area (i.e., time to accuracy). Numbers are from the MobileNet on OpenImage dataset (§6.8.2.1).

overrepresented by other participants over past rounds. On the other hand, using a client with high statistical utility ("Opt-Stat. Efficiency") may lead to longer rounds if that client turns out to be the system bottleneck in global model aggregation.

**Challenges.** To improve time-to-accuracy performance, Oort aims to find a sweet spot in the trade-off by associating with every client its *utility* toward optimizing each form of efficiency (Figure 6.5). This leads to three challenges:

- In each round, how to determine which clients' data would help improve the statistical efficiency of training the most while respecting client privacy (§6.5.2)?
- How to take a client's system performance into account to optimize the global system efficiency (§6.5.3)?
- How to account for the fact that we don't have up-to-date utility values for all clients during training (§6.5.4)?

Next, we integrate system designs with ML principles to tackle the heterogeneity, the massive scale, the runtime uncertainties and privacy concerns of clients for practical FL.

### 6.5.2 Client Statistical Utility

An ideal design of statistical utility should be able to efficiently capture the client data utility toward improving model performance for various training tasks, and respect privacy.

To this end, we leverage importance sampling used in the ML literature [146, 153]. Say each client $i$ has a bin $B_i$ of training samples locally stored. Then, to improve the round-to-accuracy performance via importance sampling, the optimal solution would be to pick bin $B_i$ with a

probability proportional to its importance $|B_i|\sqrt{\frac{1}{|B_i|}\sum_{k\in B_i} \parallel \nabla f(k) \parallel^2}$, where $\parallel \nabla f(k) \parallel$ is the L2-norm of the unique sample $k$'s gradient $\nabla f(k)$ in bin $B_i$. Intuitively, this means selecting the bin with larger aggregate gradient norm across all of its samples.

However, taking this importance as the statistical utility is impractical, since it requires an extra time-consuming pass over the client data to generate the gradient norm of every sample,[4] and this gradient norm varies as the model updates.

To avoid extra cost, we introduce a pragmatic approximation of statistical utility instead. At its core, the gradient is derived by taking the derivative of training loss with respect to current model weights, wherein training loss measures the estimation error between model predictions and the ground truth. Our insight is that a larger gradient norm often attributes to a bigger loss [146]. Therefore, we define the statistical utility $U(i)$ of client $i$ as $U(i) = |B_i|\sqrt{\frac{1}{|B_i|}\sum_{k\in B_i} Loss(k)^2}$, where the training loss $Loss(k)$ of sample $k$ is automatically generated during training with negligible collection overhead. As such, we consider clients that currently accumulate a bigger loss to be more important for future rounds.

Our statistical utility can capture the heterogeneous data utility across and within categories and samples for various tasks. We present the theoretical insights for its effectiveness over random sampling in Appendix D.1, and empirically show its close-to-optimal performance (§6.8.2.2).

**How Oort respects privacy?** Training loss measures the prediction confidence of a model without revealing the raw data and is often collected in real FL deployments [117, 269]. We further provide three ways to respect privacy. First, we rely on *aggregate* training loss, which is computed locally by the client across *all* of her samples without revealing the loss distribution of individual samples either. Second, when even the aggregate loss raises a privacy concern, clients can add noise to their loss value before uploading, similar to existing local differential privacy [98]. Third, we later show that Oort can flexibly accommodate other definitions of statistical utility used in our generic participant selection framework (§6.5.4). We provide detailed theoretical analyses for each strategy (e.g., using gradient norm of batches) of how Oort can respect privacy (e.g., amenable under noisy utility value) in Appendix D.2, while empirically showing its superior performance even under noisy utility value (§6.8.2.3).

### 6.5.3 Trading off Statistical and System Efficiency

Simply selecting clients with high statistical utility can hamper the system efficiency. To reconcile the demand for both efficiencies, we should maximize the statistical utility we can achieve per unit time (i.e., the division of statistical utility and its round duration). As such, we formulate

[4] ML models generate the training loss of each sample during training, but calculate the gradient of the mini-batch instead of individual samples.

the utility of client $i$ by associating her statistical utility with a global system utility in terms of the duration of each training round:

$$Util(i) = \underbrace{|B_i|\sqrt{\frac{1}{|B_i|}\sum_{k\in B_i} Loss(k)^2}}_{Statistical\ utility\ U(i)} \times \underbrace{(\frac{T}{t_i})^{\mathbb{1}(T<t_i)\times\alpha}}_{Global\ sys\ utility} \tag{6.1}$$

where $T$ is the developer-preferred duration of each round, $t_i$ is the amount of time that client $i$ takes to process the training, which has already been collected by today's coordinator from past rounds,[5] and $\mathbb{1}(x)$ is an indicator function that takes value 1 if $x$ is true and 0 otherwise. This way, the utility of those clients who may be the bottleneck of the desired speed of current round will be penalized by a developer-specified factor $\alpha$, but we do not reward the non-straggler clients because their completions do not impact the round duration.

This formulation assumes that all samples at a client are processed in that training round. Even if the estimated $t_i$ for a client is greater than the desired round duration $T$, Oort might pick that client if the statistical utility outweighs its slow speed. Alternatively, if the developer wishes to cap every round at a certain duration [187], then either only clients with $t_i < T$ can be considered (e.g., by setting $\alpha \rightarrow \infty$) or a subset of a participant's samples can be processed [169, 216], and only the aggregate training loss of those trained data in that round is considered in measuring the statistical utility.

**Navigating the trade-off.** Determining the preferred round duration $T$ in Equation (6.1), which strikes the trade-off between the statistical and system efficiency in aggregations, is non-trivial. Indeed, the total statistical utility (i.e., $\sum U(i)$) achieved by picking high utility clients can decrease round by round, because the training loss decreases as the model improves over time. If we persist in suppressing clients with high statistical utility but low system speed, the model may converge to suboptimal accuracy (§6.8.2.2).

To navigate the optimal trade-off – maximizing the total statistical utility achieved without greatly sacrificing the system efficiency – Oort employs a pacer to determine the preferred duration $T$ at runtime. The intuition is that, when the accumulated statistical utility in the past rounds decreases, the pacer allows a larger $T \leftarrow T + \Delta$ by $\Delta$ to bargain with the statistical efficiency again. We elaborate more in Algorithm 6.1.

[5] We only care whether a client can complete by the expected duration $T$. So, a client can even mask its precise speed by deferring its report.

### 6.5.4 Adaptive Participant Selection

Given the above definition of client utility, we need to address the following practical concerns in order to select participants with the highest utility in each training round.

- *Scalability*: a client's utility can only be determined after it has participated in training; how to choose from clients at scale without having to try all clients once?
- *Staleness*: since not every client participates in every round, how to account for the change in a client's utility since its last participation?
- *Robustness*: how to be robust to outliers in the presence of corrupted clients (e.g., with noisy data)?

To tackle these challenges, we develop an exploration-exploitation strategy for participant selection (Algorithm 6.1).

**Online exploration-exploitation of high-utility clients.** Selecting participants out of numerous clients can be modeled as a multi-armed bandit problem, where each client is an "arm" of the bandit, and the utility obtained is the "reward" [44]. In contrast to sophisticated designs (e.g., reinforcement learning), the bandit model is scalable and flexible even when the solution space (e.g., number of clients) varies dramatically over time. Next, we adaptively balance the exploration and exploitation of different arms to maximize the long-term reward.

Similar to the bandit design, Oort efficiently explores potential participants under spatial variation, while intelligently exploiting observed high-utility participants under temporal variation. At the beginning of each selection round, Oort receives the feedback of the last training round, and updates the statistical utility and system performance of clients (Line 6). For the explored clients, Oort calculates their client utility and narrows down the selection by exploiting the high-utility participants (Line 9-15). Meanwhile, Oort samples $\epsilon(\in [0, 1])$ fraction of participants to explore potential participants that had not been selected before (Line 16), which turns to full exploration as $\epsilon \rightarrow 1$. Although we cannot learn the statistical utility of not-yet-tried clients, one can decide to prioritize the unexplored clients with faster system speed when possible (e.g., by inferring from device models), instead of performing random exploration (Line 16).

**Exploitation under staleness in client utility.** Oort employs two strategies to account for the dynamics in client utility over time. First, motivated by the confidence interval used to measure the uncertainty in bandit reward, we introduce an incentive term, which shares the same shape of the confidence in bandit solutions [142], to account for the staleness (Line 10), whereby we gradually increase the utility of a client if she has been overlooked for a long time. So those clients accumulating high utility since their last trial can still be repurposed again. Second, instead of

**Input:** Client set $\mathbb{C}$, sample size $K$, exploitation factor $\epsilon$, pacer step $\Delta$, step window $W$, penalty $\alpha$
**Output:** Participant set $\mathbb{P}$

```
    /* Initialize global variables. */
1:  E ← ∅; U ← ∅                                   ▷ Explored clients and statistical utility.
2:  L ← ∅; D ← ∅                                   ▷ Last involved round and duration.
3:  R ← 0; T ← Δ                                   ▷ Round counter and preferred round duration.
    /* Participant selection for each round. */
4:  Function SelectParticipant (C, K, ε, T, α)
5:  |   Util ← ∅; R ← R + 1
        /* Update and clip the feedback; blacklist outliers. */
6:  |   UpdateWithFeedback(E, U, L, D)
        /* Pacer: Relaxes global system preference T if the statistical utility achieved decreases in last
           W rounds. */
7:  |   if ∑U(R − 2W : R − W) > ∑U(R − W : R) then
8:  |   |   T ← T + Δ
        /* Exploitation #1: Calculate client utility. */
9:  |   for client i ∈ E do
10: |   |   Util(i) ← U(i) + sqrt(0.1 log R / L(i))     ▷ Temporal uncertainty.
11: |   |   if T < D(i) then                           ▷ Global system utility.
12: |   |   |   Util(i) ← Util(i) × (T / D(i))^α
        /* Exploitation #2: admit clients with greater than c% of cut-off utility; then sample (1 − ε)K
           clients by utility. */
13: |   Util ← SortAsc(Util)
14: |   W ← CutOffUtil(E, c × Util((1 − ε) × K))
15: |   P ← SampleByUtil(W, Util, (1 − ε) × K)
        /* Exploration: sample unexplored clients by speed. */
16: |   P ← P ∪ SampleBySpeed(C − E, ε × K)
17: |   return P
```

**Alg. 6.1:** Participant selection w/ exploration-exploitation.

picking clients with top-k utility deterministically, we allow a confidence interval $c$ on the cut-off utility (95% by default in Line 13-14). Namely, we admit clients whose utility is greater than the $c\%$ of the top $((1-\epsilon)\times K)$-th participant. Among this high-utility pool, Oort samples participants with probability proportional to their utility (Line 15). This adaptive exploitation mitigates the uncertainties in client utility by prioritizing participants opportunistically, thus relieving the need for accurate estimations of utility as we do not require the exact ordering among clients, while

```
1  def federated_model_testing():
2    selector = Oort.create_testing_selector()
3
4    # Type 1: subset w/ < X deviation from the global
5    participants = selector.select_by_deviation(
6      dev_target, range_of_capacity, total_num_clients)
7
8    # Provide individual client data characteristics
9    selector.update_client_info(client_id, client_info)
10   # Type 2: [5k, 5k] samples of category [i, j]
11   participants = selector.select_by_category(
12                   request_list, testing_config)
```

Figure 6.6: Key Oort APIs for supporting federated testing.

preserving a high quality as a whole.

**Robust exploitation under outliers.** Simply prioritizing high utility clients can be vulnerable to outliers in unfavorable settings. For example, corrupted clients may have noisy data, leading to high training loss, or even report arbitrarily high training loss intentionally. For robustness, Oort (i) removes the client in selection after she has been picked over a given number of rounds. This helps to remove the perceived outliers in terms of participation (Line 6); (ii) clips the utility value of a client by capping it to no more than an upper bound (e.g., 95% value in utility distributions). With probabilistic participant selection among the high-utility client pool (Line 15), the chance of selecting outliers is significantly decreased under the scale of clients in FL. We show that Oort outperforms existing mechanisms while being robust (§6.8.2.3).

**Accommodation to diverse selection criteria.** Our adaptive participant selection is generic for different utility definitions of diverse selection criteria. For example, developers may hope to reconcile their demand for time-to-accuracy efficiency and fairness, so that some clients are not underrepresented (e.g., more fair resource usage across clients) [148, 170]. Although developers may have various fairness criterion $fairness(\cdot)$, Oort can enforce their demands by replacing the current utility definition of client $i$ with $(1-f) \times Util(i) + f \times fairness(i)$, where $f \in [0,1]$ and Algorithm 6.1 will naturally prioritize clients with the largest fairness demand as $f \to 1$. For example, $fairness(i) = max_resource_usage - resource_usage(i)$ motivates fair resource usage for each client $i$. Note that existing participant selection provides no support for fairness, and we show that Oort can efficiently enforce diverse developer-preferred fairness while improving performance (§6.8.2.3).

## 6.6 Federated Model Testing

Enforcing developer-defined requirements on data distribution is a first-order goal in FL testing, whereas existing mechanisms lead to biased testing results (§6.3). In this section, we elaborate on how Oort serves the two primary types of queries. As shown in Figure 6.6, we start with how Oort preserves the representativeness of testing set even without individual client data characteristics (§6.6.1), and how it efficiently enforces developer's testing criteria for specific data distribution when the individual information is provided (§6.6.2).

### 6.6.1 Preserving Data Representativeness

Learning the individual data characteristics (e.g., categorical distribution) can be too expensive or even prohibited [220, 91]. Without knowing data characteristics, the developer has to be conservative and selects many participants to gain more confidence for query *"a testing set with less than X% data deviation from the global"*, as selecting too few can lead to a biased testing result (§6.3). However, admitting too many may inflate the budget and/or take too long because of the system heterogeneity. Next, we show how Oort can enable guided participant selection by determining the number of participants needed to guarantee this deviation target.

We consider the deviation of the data formed by all participants from the global dataset (i.e., representative) using L1-distance, a popular distance metric in FL [129, 191, 130]. For category $X$, its L1-distance ($|\bar{X} - E[\bar{X}]|$) captures how the average number of samples of all participants (i.e., empirical value $\bar{X}$) deviates from that of all clients (i.e., expectation $E[\bar{X}]$). Note that the number of samples $X_n$ that client $n$ holds is independent across clients. Namely, the number of samples that one client holds will not be affected by the selection of any other clients at that time, so it can be viewed as a random instance sampled from the distribution of variable $X$.

Given the developer-specified tolerance $\epsilon$ on data deviation and confidence interval $\delta$ (95% by default [188]), our goal is to estimate the number of participants needed such that the deviation from the representative categorical distribution is bounded (i.e., $Pr[|\bar{X} - E[\bar{X}]| < \epsilon] > \delta$). To this end, we formulate it as a problem of sampling stochastic variables, and apply the Hoeffding bound [47] to capture how this data deviation varies with different number of participants.

**Estimating the number of participants to cap deviation.** Even when the individual data characteristics are not available, the developer can specify her tolerance $\epsilon$ on the deviation from the global categorical distribution, whereby Oort outputs the number of participants needed to preserve this preference. To use our model, the developer needs to input the global range (i.e., global maximum - global minimum) of the number of samples that one client can hold, and the total number of clients. Learning this global information securely is well-established [75, 220], and the developer

can assume a plausible limit (e.g., according to the capacity of device models) too.

Our model does not require any collection of the distribution of global or participant data. As a straw-man participant selection design, the developer can randomly distribute her model to this Oort-determined number of participants. After collecting results from this number of participants, she can confirm the representativeness of computed data.

### 6.6.2 Enforcing Diverse Data Distribution

When the individual data characteristics are provided (e.g., FL across enterprise AI cameras [179, 130]), Oort can enforce the exact data preference on specific categorical distribution, and improve the duration of testing by cherry-picking participants.

Satisfying queries like *"[5k, 5k] samples of class [x, y]"* can be viewed as a multi-dimensional bin covering problem, where a subset of data bins (i.e., participants) are selected to cover the requested quantity of data. For each category $i (\in I)$ of interest, the developer has preference $p_i$ (preference constraint), and an upper limit $B$ (referred to as budget) on how many participants she can have [45]. Each participant $n (\in N)$ can contribute $n_i$ samples out of her capacity ${c_n}^i$ (capacity constraint). Given her compute speed $s_n$, the available bandwidth $b_n$ and the size of data transfers $d_n$, we aim to minimize the duration of model testing:

$$
\begin{aligned}
& \min \left\{ \max_{n \in N} \left( \frac{\sum_{i \in I} n_i}{s_n} + \frac{d_n}{b_n} \right) \right\} && \triangleright \text{ Minimize duration} \\
\text{s.t.} \quad & \forall i \in I, \sum_{n \in N} n_i = p_i && \triangleright \text{ Preference Constraint} \\
& \forall i \in I, \forall n \in N, n_i \le c_n^i && \triangleright \text{ Capacity Constraint} \\
& \forall i \in I, \sum_{n \in N} \mathbb{1}(n_i > 0) \le B && \triangleright \text{ Budget Constraint}
\end{aligned}
$$

The max-min formulation stems from the fact that testing completes after aggregating results from the last participant. While this mixed-integer linear programming (MILP) model provides high-quality solutions, it has prohibitively high computational complexity for large $N$.

**Scalable participant selection.** For better scalability, we present a greedy heuristic to scale down the search space of this strawman. We (1) first group a subset of feasible clients to satisfy the preference constraint. To this end, we iteratively add to our subset the client which has the most number of samples across all not-yet-satisfied categories, and deduct the preference constraint on each category by the corresponding capacity of this client. We stop this greedy grouping until the preference is met, or request a new budget if we exceed the budget; and (2) then optimize job duration with a simplified MILP among this subset of clients, wherein we have removed the budget

constraint and reduced the search space of clients. We show that our heuristic can outperform the straw-man MILP model in terms of the end-to-end duration of model testing owing to its small overhead (§6.8.3.2).

## 6.7 Implementation

We have implemented Oort as a Python library, with 2617 lines of code, to friendly support FL developers. Oort provides simple APIs to abstract away the problem of participant selection, and developers can import Oort in their application codebase and interact with FL engines (e.g., PySyft [23] or TensorFlow Federated [31]).

We have integrated Oort with PySyft. Oort operates on and updates its client metadata (e.g., data distribution or system performance) fed by the FL developer and PySyft at runtime. The metadata of each client in Oort is an object with a small memory footprint. Oort caches these objects in memory during executions and periodically backs them up to persistent storage. In case of failures, the execution driver will initiate a new Oort selector, and load the latest checkpoint to catch up. We employ Gurobi solver [12] to solve the MILP. The developer can also initiate a Oort application beyond coordinators to avoid resource contention. We use *xmlrpc* library to connect to the coordinator, and these updates will activate Oort to write these updates to its metastore. In the coordinator, we use the PySyft API *model.send(client_id)* to direct which client to run given the Oort decision, and *model.get(client_id)* to collect the feedback.

## 6.8 Evaluation

We evaluate Oort's effectiveness for four different ML models on four CV and NLP datasets. We organize our evaluation by the FL activities with the following key results.

**FL training results summary:**

- Oort outperforms existing random participant selection by 1.2×-14.1× in time-to-accuracy performance, while achieving 1.3%-9.8% better final model accuracy (§6.8.2.1).
- Oort achieves close-to-optimal model efficiency by adaptively striking the trade-off between statistical and system efficiency with different components (§6.8.2.2).
- Oort outperforms its counterpart over a wide range of parameters and different scales of experiments, while being robust to outliers (§6.8.2.3).

**FL testing results summary:**

- Oort can serve testing criteria on data deviation while reducing costs by bounding the number of participants needed without individual data characteristics (§6.8.3.1).

| Dataset | # of Clients | # of Samples |
|---|---|---|
| Google Speech [252] | 2,618 | 105,829 |
| OpenImage-Easy [11] | 14,477 | 871,368 |
| OpenImage [11] | 14,477 | 1,672,231 |
| StackOverflow [28] | 315,902 | 135,818,730 |
| Reddit [25] | 1,660,820 | 351,523,459 |

Table 6.1: Statistics of the dataset in evaluations.

- With the individual information, Oort improves the testing duration by 4.7× w.r.t. Mixed Integer Linear Programming (MILP) solver, and is able to efficiently enforce developer preferences across millions of clients (§6.8.3.2).

### 6.8.1 Methodology

**Experimental setup.** Oort is designed to operate in large deployments with potentially millions of edge devices. However, such a deployment is not only prohibitively expensive, but also impractical to ensure the reproducibility of experiments. As such, we resort to a cluster with 68 NVIDIA Tesla P100 GPUs, and emulate up to 1300 participants in each round. We simulate real-world heterogeneous client system performance and data in both training and testing evaluations using FedScale: (1) Heterogeneous device runtimes of different models, network throughput/connectivity, device model and availability are emulated using data from AI Benchmark [1] and Network Measurements on mobiles [19]; (2) We distribute each real dataset to clients following the corresponding raw placement (e.g., using <*authors_ID*> to allocate OpenImage), where client data can vary in quantities, distribution of outputs and input features; (3) The coordinator communicates with clients using the parameter server architecture. These follow the PySyft and real FL deployments. To mitigate stragglers, we employ the widely-used mechanism specified in real FL deployments [55], where we collect updates from the first $K$ completed participants out of $1.3K$ participants in each round, and $K$ is 100 by default. We report the simulated clock time of clients in evaluations.

**Datasets and models.** We run three categories of applications with four real-world datasets of different scales, and Table 6.1 reports the statistics of each dataset:

- *Speech Recognition*: the small-scale Google speech dataset [252]. We train a convolutional neural network model (ResNet-34 [120]) to recognize the command among 35 categories.
- *Image Classification*: the middle-scale OpenImage [11] dataset, with 1.5 million images spanning 600 categories, and a simpler dataset (OpenImage-Easy) with images from the most popular 60 categories. We train MobileNet [224] and ShuffleNet [287] models to classify the image.

| Task | Dataset | Accuracy Target | Model | Speedup for Prox [169] | | | Speedup for YoGi [216] | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | Stats. | Sys. | Overall | Stats. | Sys. | Overall |
| Image Classification | OpenImage-Easy [11] | 74.9% | MobileNet [224] | 3.8× | 3.2× | 12.1× | 2.4× | 2.4× | 5.7× |
| | | | ShuffleNet [287] | 2.5× | 3.5× | 8.8× | 1.9× | 2.7× | 5.1× |
| | OpenImage [11] | 53.1% | MobileNet | 4.2× | 3.1× | 13.0× | 2.3× | 1.5× | 3.3× |
| | | | ShuffleNet | 4.8× | 2.9× | 14.1× | 1.8× | 3.2× | 5.8× |
| Language Modeling | Reddit [25] | 39 perplexity | Albert [164] | 1.3× | 6.4× | 8.4× | 1.5× | 4.9× | 7.3× |
| | StackOverflow [28] | 39 perplexity | Albert | 2.1× | 4.3× | 9.1× | 1.8× | 4.4× | 7.8× |
| Speech Recognition | Google Speech [252] | 62.2% | ResNet-34 [120] | 1.1× | 1.1× | 1.2× | 1.2× | 1.1× | 1.3× |

Table 6.2: Summary of improvements on time to accuracy.[7] We tease apart the overall improvement with statistical and system ones, and take the highest accuracy that Prox can achieve as the target, which is moderate due to the high task complexity and lightweight models.

- *Language Modeling*: the large-scale StackOverflow[28] and Reddit[25] dataset. We train next word predictions with Albert model [164] using the top-10k popular words.

These applications are widely used in real end-device applications [263], and these models are designed to be lightweight.

**Parameters.** The minibatch size of each participant is 16 in speech recognition, and 32 in other tasks. The initial learning rate for Albert model is 4e-5, and 0.04 for other models. These configurations are consistent with those reported in the literature [116]. In configuring the training selector, Oort uses the popular time-based exploration factor [44], where the initial exploration factor is 0.9, and decreased by a factor 0.98 after each round when it is larger than 0.2. The step window of pacer $W$ is 20 rounds. We set the pacer step $\Delta$ in a way that it can cover the duration of next $W \times K$ clients in the descending order of explored clients' duration, and the straggler penalty $\alpha$ to 2. We remove a client from Oort's exploitation list once she has been selected over 10 times.

**Metrics.** We care about the *time-to-accuracy* performance and *final model accuracy* of model training tasks on the testing set. For model testing, we measure the *end-to-end* testing duration, which consists of the computation overhead of the solution and the duration of actual computation.

For each experiment, we report the mean value over 5 runs, and error bars show the standard deviation.

### 6.8.2 FL Training Evaluation

In this section, we evaluate Oort's performance on model training, and employ Prox [169] and YoGi [216]. We refer Prox as Prox running with existing random participant selection, and Prox + Oort is Prox running atop Oort. We use a similar denotation for YoGi. Note that Prox and

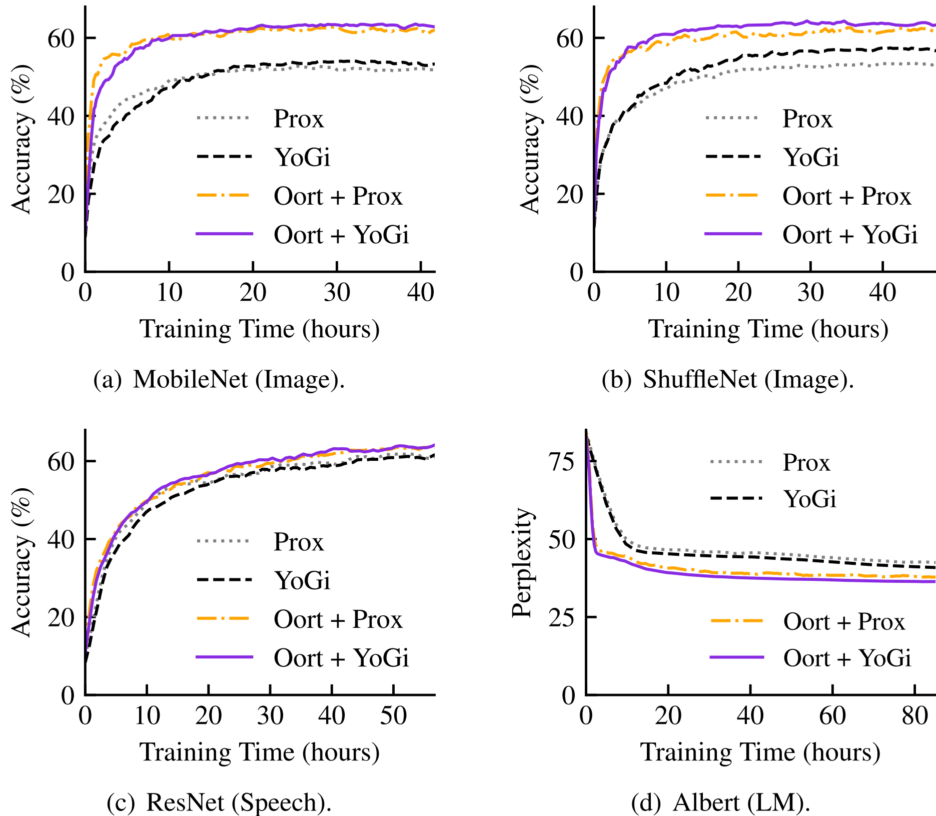



Figure 6.7: Time-to-Accuracy performance. A lower perplexity is better in the language modeling (LM) task.

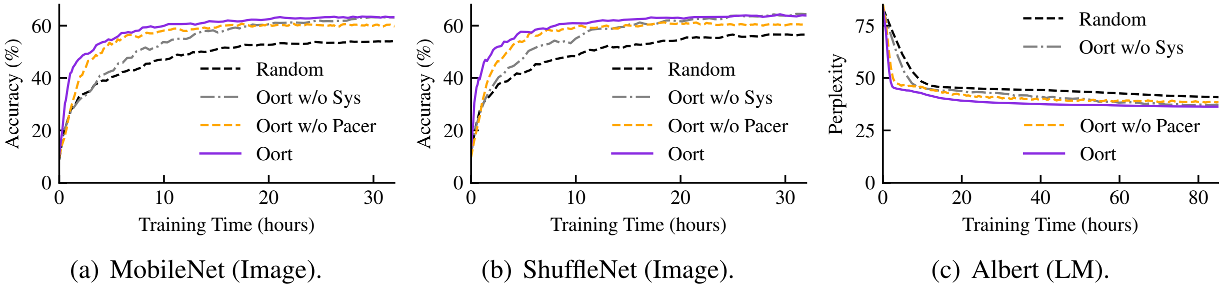
YoGi optimize the statistical model efficiency for the given participants, while Oort cherry-picks participants to feed them.

#### 6.8.2.1 End-to-End Performance

Table 6.2 summarizes the key time-to-accuracy performance of all datasets. In the rest of the evaluations, we report the ShuffleNet and MobileNet performance on OpenImage, and Albert performance on Reddit dataset for brevity. Figure 6.7 reports the timeline of training to achieve different accuracy.

**Oort improves time-to-accuracy performance.** We notice that Oort achieves large speedups to reach the target accuracy (Table 6.2). Oort reaches the target 3.3×-14.1× faster in terms of wall clock time on the middle-scale OpenImage dataset; speedup on the large-scale Reddit and StackOverflow dataset is 7.3×-9.1×. Understandably, these benefits decrease when the total number

[7]We set the target accuracy to be the highest achievable accuracy by all used strategies, which turns out to be Prox accuracy. Otherwise, some may never reach that target.



(a) MobileNet (Image).
(b) ShuffleNet (Image).
(c) Albert (LM).

Figure 6.8: Breakdown of Time-to-Accuracy performance with YoGi, when using different participant selection strategies.

of clients is small, as shown on the small-scale Google Speech dataset (1.2×-1.3×).

These time-to-accuracy improvements stem from the comparable benefits in statistical model efficiency and system efficiency (Table 6.2). Oort takes 1.8×-4.8× fewer training rounds on OpenImage dataset to reach the target accuracy, which is better than that of language modeling tasks (1.3×-2.1×). This is because real-life images often exhibit greater heterogeneity in data characteristics than the language dataset, whereas the large population of language datasets leaves a great potential to prioritize clients with faster system speed.

**Oort improves final model accuracy.** When the model converges, Oort achieves 6.6%-9.8% higher final accuracy on OpenImage dataset, and 3.1%-4.4% better perplexity on Reddit dataset (Figure 6.7). Again, this improvement on Google Speech dataset is smaller (1.3% for Prox and 2.2% for YoGi) due to the small scale of clients. These improvements attribute to the exploitation of high statistical utility clients. Specifically, the statistical model accuracy is determined by the quality of global aggregation. Without cherry-picking participants in each round, clients with poor statistical model utility can dilute the quality of aggregation. As such, the model may converge to suboptimal performance. Instead, models running with Oort concentrate more on clients with high statistical utility, thus achieving better final accuracy.

#### 6.8.2.2 Performance Breakdown

We next delve into the improvement on middle- and large-scale datasets, as they are closer to real FL deployments. We break down our knobs designed for striking the balance between statistical and system efficiency: (i) (*Oort w/o Pacer*): We disable the pacer that guides the aggregation efficiency. As such, it keeps suppressing low-speed clients, and the training can be restrained among low-utility but high-speed clients; (ii) (*Oort w/o Sys*): We further totally remove our benefits from system efficiency by setting $\alpha$ to 0, so Oort blindly prioritizes clients with high statistical utility. We take YoGi for analysis, because it outperforms Prox most of the time.

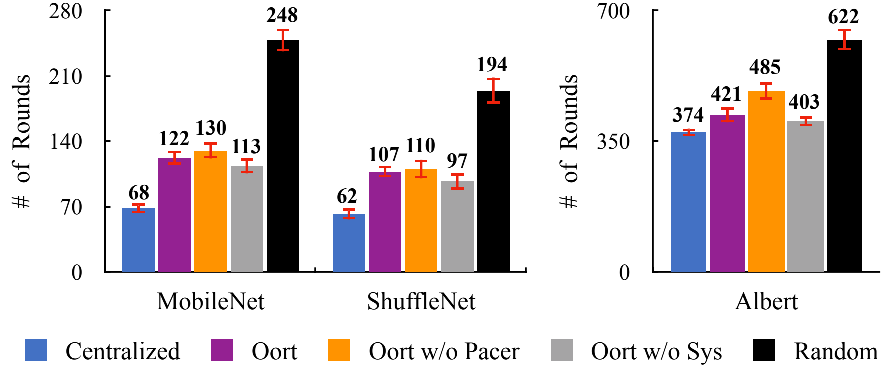



Figure 6.9: Number of rounds to reach the target accuracy.

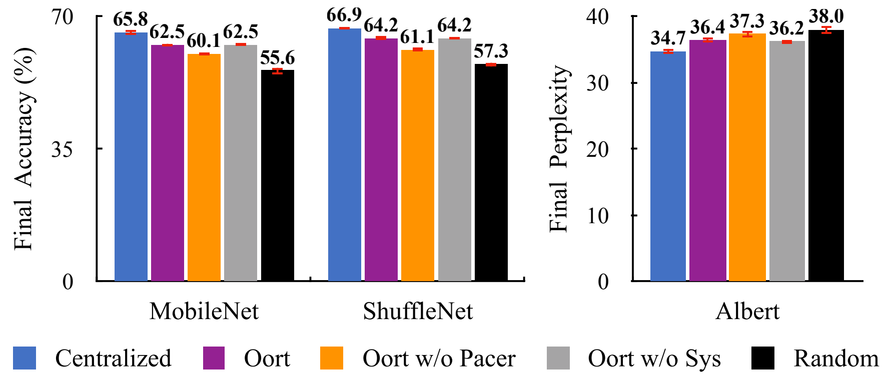



Figure 6.10: Breakdown of final model accuracy.

**Breakdown of time-to-accuracy efficiency.** Figure 6.8 reports the breakdown of time-to-accuracy performance, where Oort achieves comparable improvement from statistical and system optimizations. Taking Figure 6.8(b) as an example, (i) At the beginning of training, both Oort and (Oort w/o Pacer) improve the model accuracy quickly, because they penalize the utility of stragglers and select clients with higher statistical utility and system efficiency. In contrast, (Oort w/o Sys) only considers the statistical utility, resulting in longer rounds. (ii) As training evolves, the pacer in Oort gradually relaxes the constraints on system efficiency, and admits clients with relatively low speed but higher statistical utility, which ends up with the similar final accuracy of (Oort w/o Sys). However, (Oort w/o Pacer) relies on a fixed system constraint and suppresses valuable clients with high statistical utility but low speed, leading to suboptimal final accuracy.

**Oort achieves close to upper-bound statistical performance.** We consider an *upper-bound* statistical efficiency by creating a centralized case, where all data are evenly distributed to $K$ participants. Using the target accuracy in Table 6.2, Oort can efficiently approach this upper bound by incorporating different components (Figure 6.9). Oort is within $2\times$ of the upper-bound to achieve

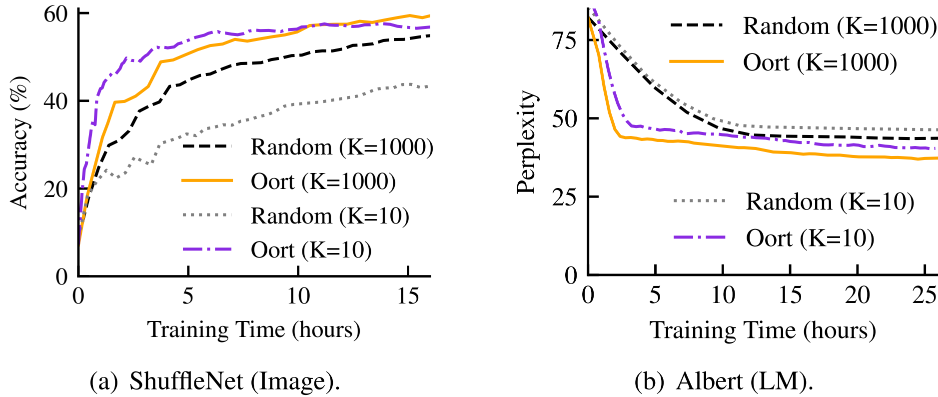



Figure 6.11: Oort outperforms in different scales of participants.

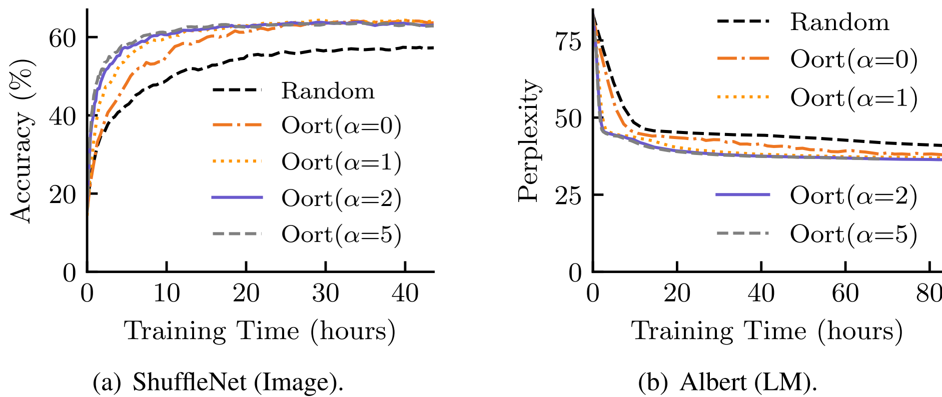
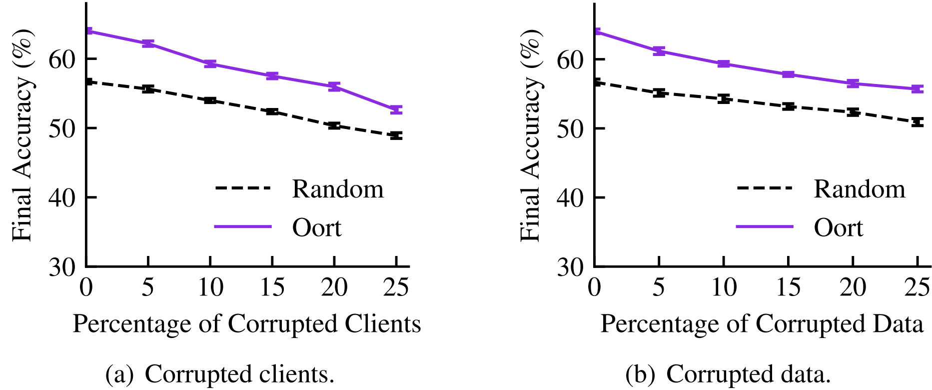



Figure 6.12: Oort improves performance across penalty factors.

the target accuracy, and (Oort w/o Sys) performs the best in statistical model efficiency, because (Oort w/o Sys) always grasps clients with higher statistical utility. However, it is suboptimal in our targeted time-to-accuracy performance because of ignoring the system efficiency. Moreover, by introducing the pacer, Oort achieves 2.4%-3.1% better accuracy than (Oort w/o Pacer), and is merely about 2.7%-3.3% worse than the upper-bound final model accuracy (Figure 6.10).

#### 6.8.2.3 Sensitivity Analysis

**Impact of number of participants $K$.** We evaluate Oort across different scales of participants in each round, where we cut off the training after 200 rounds given the diminishing rewards. We observe that Oort improves time-to-accuracy efficiency across different number of participants (Figure 6.11), and having more participants in FL indeed receives diminishing rewards. This is because taking more participants (i) is similar to having a large batch size, which is confirmed to be even negative to round-to-accuracy performance [173]; (ii) can lead to longer rounds due to stragglers when the number of clients is limited (e.g., $K$=1000 on OpenImage dataset).



(a) Corrupted clients.

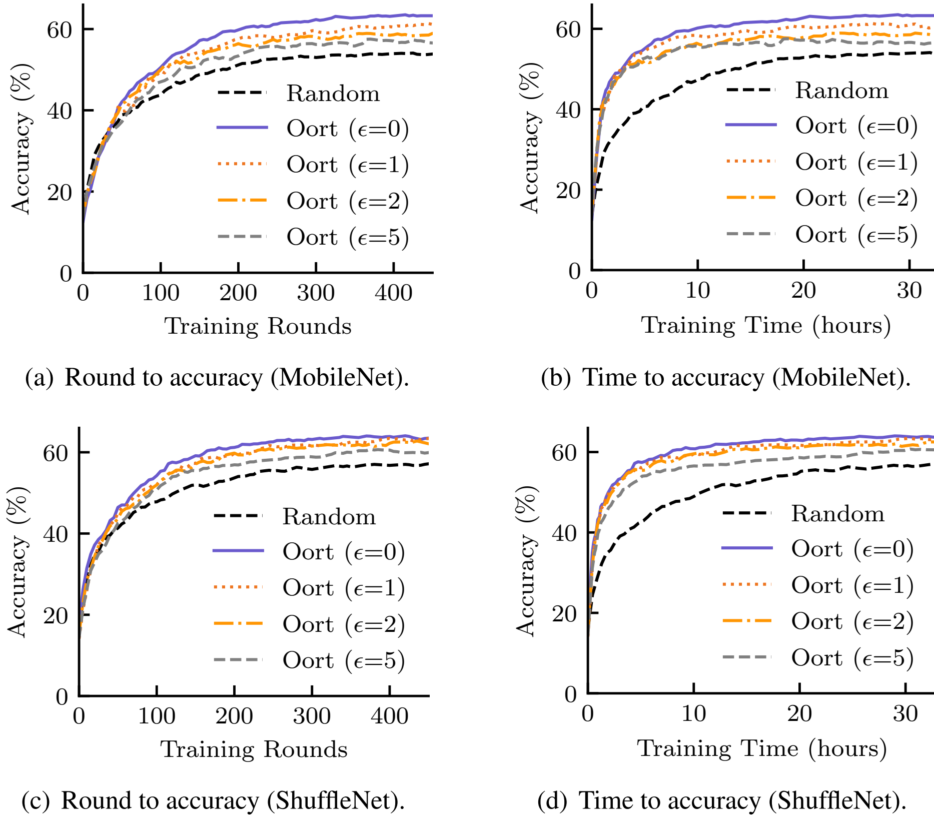
(b) Corrupted data.

Figure 6.13: Oort still improves performance under outliers.

**Impact of penalty factor $\alpha$ on stragglers.** Oort uses the penalty factor $\alpha$ to penalize the utility of stragglers in participant selection, whereby it adaptively prioritizes high system efficiency participants. Figure 6.12 shows that Oort outperforms its counterparts across different $\alpha$. Note that Oort orchestrates its components to automatically navigate the best performance across parameters: larger $\alpha$ (i.e., overemphasizing system efficiency) drives the Pacer to relax the system constraint $T$ more frequently to admit clients with higher statistical efficiency, and vice versa. As such, Oort achieves similar performance across all non-zero $\alpha$.

**Impact of outliers.** We investigate the robustness of Oort by introducing outliers manually. Following the popular adversarial ML setting [94], we randomly flip the ground-truth data labels of the OpenImage dataset to any other categories, resulting in artificially high utility. We consider two practical scenarios with the ShuffleNet model: (i) Corrupted clients: labels of all training samples on these clients are flipped (Figure 6.13(a)); (ii) Corrupted data: each client uniformly flips a subset of her training samples (Figure 6.13(b)). We notice Oort still outperforms across all degrees of corruption.

**Impact of noisy utility.** We next show the superior performance of Oort over its counterparts under noisy utility value. In this experiment, we add noise from the Gaussian distribution $Gaussian(0, \sigma^2)$, and investigate Oort's performance with different $\sigma$. Similar to differential FL [98], we define $\sigma = \epsilon \times Mean(real_value)$, where $Mean(real_value)$ is the average real value without noise. Note that we take this $real_value$ as reference for the ease of presentations, and developers can refer to other values. As such, a large $\epsilon$ implies larger variance in noise, thus providing better privacy by disturbing the real value significantly. We report the statistical efficiency after adding noise to the statistical utility (Fig 6.14(a) and Fig 6.14(c)), as well as the time-to-accuracy performance (Fig 6.14(b) and Fig 6.14(d)). We observe that Oort still improves performance across different




(a) Round to accuracy (MobileNet).
(b) Time to accuracy (MobileNet).
(c) Round to accuracy (ShuffleNet).
(d) Time to accuracy (ShuffleNet).

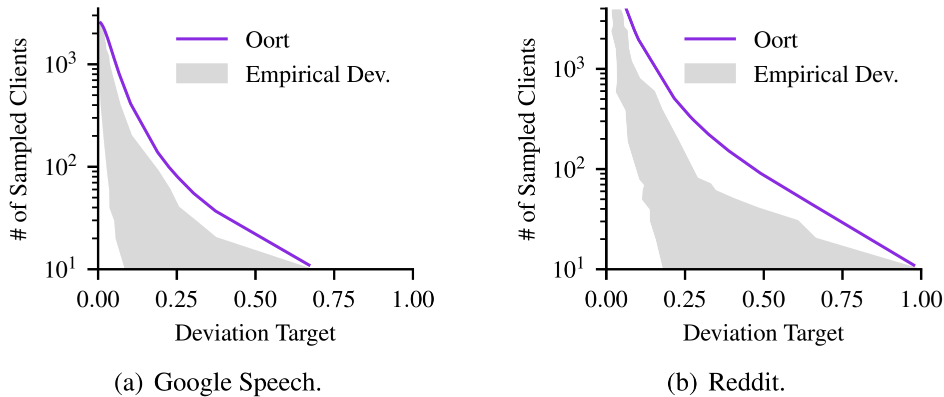
Figure 6.14: Oort improves performance even under noise.

amount of noise, and is robust even when the noise is large (e.g., $\epsilon = 5$ is often considered to be very large noise [37]).

**Oort can respect developer-preferred fairness.** In this experiment, we expect all clients should have participated training with the same number of rounds (Table 6.3), implying a fair resource usage [148]. We train ShuffleNet model on OpenImage dataset with YoGi. To this end, we sweep different knobs $f$ to accommodate the developer demands for the time-to-accuracy efficiency and fairness. Namely, we replace the current utility definition of client $i$ with $(1-f) \times Util(i) + f \times fairness(i)$, where $fairness(i) = max_resource_usage - resource_usage(i)$. Understandably, time-to-accuracy efficiency significantly decreases as $f \to 1$, since we gradually end up with round-robin participant selection, totally ignoring the utility of clients. Note that Oort still achieves better time-to-accuracy even when $f \to 1$ as it prioritizes high system utility clients at the beginning of training, thus achieving shorter rounds. Moreover, Oort can enforce different fairness preferences while improving efficiency across fairness knobs.

| Strategy | TTA (h) | Final Accuracy (%) | Var. (Rounds) |
|---|---|---|---|
| Random | 36.3 | 57.3 | 0.39 |
| $f = 0$ | 5.8 | 64.2 | 6.52 |
| $f = 0.25$ | 6.1 | 62.4 | 5.1 |
| $f = 0.5$ | 13.1 | 59.7 | 2.03 |
| $f = 0.75$ | 25.4 | 58.6 | 0.65 |
| $f = 1$ | 30.1 | 57.2 | 0.31 |

Table 6.3: Oort improves time to accuracy (TTA) across different fairness knobs ($f$). Random reports the performance of random participant selection. The variance of rounds reports how fairness is enforced in terms of the number of participating rounds across clients. A smaller variance implies better fairness.



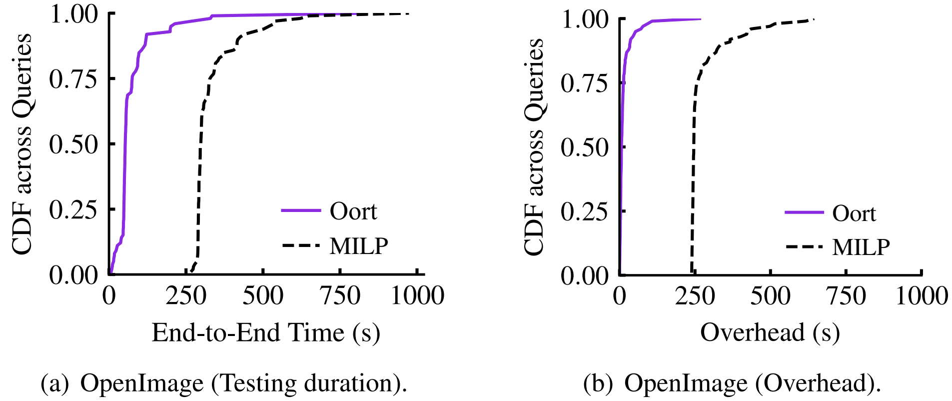

(a) Google Speech. (b) Reddit.

Figure 6.15: Oort can cap data deviation for all targets. Shadow indicates the empirical [min, max] range of the x-axis values over 1000 runs given the y-axis input.

### 6.8.3 FL Testing Evaluation

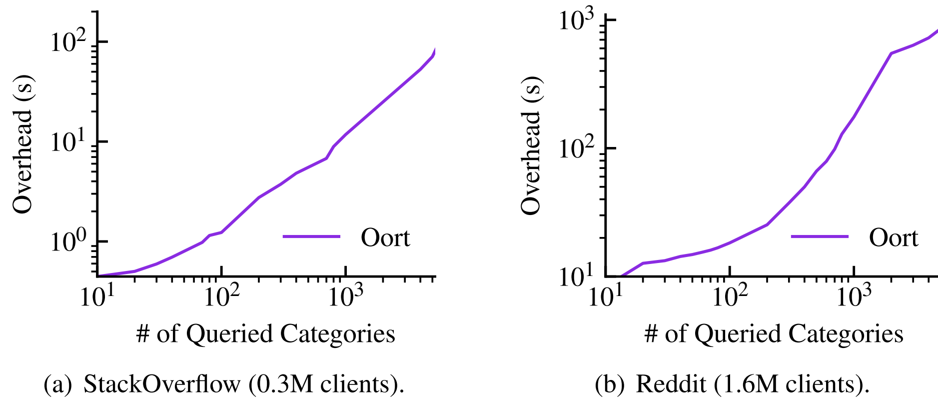
#### 6.8.3.1 Preserving Data Representativeness

**Oort can cap data deviation.** Figure 6.15 reports Oort's performance on serving different deviation targets, with respect to the global distribution. We sweep the number of selected clients from 10 to 4k, and randomly select each given number of participants over 1k times to empirically search their possible deviation. We notice that for a given deviation target, (i) different workloads require distinct number of participants. For example, to meet the target of 0.05 divergence, the Speech dataset uses 6× less participants than the Reddit attributing to its smaller heterogeneity (e.g., tighter range of the number of samples); (ii) with the Oort-determined number of participants, no empirical deviation exceeds the target, showing the effectiveness of Oort in satisfying the deviation target, whereby Oort reduces the cost of expanding participant set arbitrarily and improves the testing duration.




(a) OpenImage (Testing duration). (b) OpenImage (Overhead).

Figure 6.16: Oort outperforms MILP in clairvoyant FL testing.




(a) StackOverflow (0.3M clients). (b) Reddit (1.6M clients).

Figure 6.17: Oort scales to millions of clients, while MILP did not complete on any query.

#### 6.8.3.2 Enforcing Diverse Data Distribution

**Oort outperforms MILP.** We start with the middle-scale OpenImage dataset and compare the end-to-end testing duration of Oort and MILP. Here, we generate 200 queries using the form "Give me $X$ representative samples", where we sweep $X$ from 4k to 200k and budget $B$ from 100 participants to 5k participants. We report the validation time of MobileNet on participants selected by these strategies.

Figure 6.16(a) shows the end-to-end testing duration. We observe Oort outperforms MILP by 4.7× on average. This is because Oort suffers little computation overhead by greedily reducing the search space of MILP. As shown in Figure 6.16(b), MILP takes 274 seconds on average to complete the participant selection, while Oort only takes 15 seconds.

**Oort is scalable.** We further investigate Oort's performance on the large-scale StackOverflow and Reddit dataset with millions of clients, where we take 1% of the global data as the requirement, and sweep the number of interested categories from 1 to 5k. Figure 6.17 shows even though we

gradually magnify the search space of participant selection by introducing more categories, Oort can serve our requirement in a few minutes at the scale of millions of clients, while MILP fails to generate the solution decision for any query.

## 6.9 Related Work

**Federated Learning** Federated learning [148] is a distributed machine learning paradigm in a network of end devices, wherein Prox [169] and YoGi [216] are state-of-the-art optimizations in tackling data heterogeneity. Recent efforts in FL have been focusing on improving communication efficiency [187, 128] or compression schemes [39], ensuring privacy by leveraging multi-party computation (MPC) [56] and differential privacy [98], or tackling heterogeneity by reinventing ML algorithms [248, 170]. However, they underperform in FL because of the suboptimal participant selection they rely on, and lack systems supports for developers to specify their participant selection criteria.

**Datacenter Machine Learning** Distributed ML in datacenters has been well-studied [210, 197, 140], wherein they assume relatively homogeneous data and workers [180, 108]. While developer requirements and models can still be the same, the heterogeneity of client system performance and data distribution makes FL much more challenging. We aim at enabling them in FL. To accelerate traditional model training, some techniques bring up importance sampling to prioritize important training samples in selecting mini-batches for training [146, 153]. While bearing some resemblance in prioritizing data, Oort adaptively considers both statistical and system efficiency in formulating the client utility at scale.

**Geo-distributed Data Analytics** Federated data analytics has been a topic of interest in geo-distributed storage [238] and data processing systems [284, 161] that attempt to reduce latency [243] and/or save bandwidth [211, 244, 158]. Gaia [128] reduces network traffics for model training across datacenters, while Sol [161] enables generic federated computation on data with sub-second latency in the execution layer. These work back up Oort with cross-layer system support, whereas Oort cherry-picks participants before execution.

**Privacy-preserving Data Analytics** To gather sensitive statistics from user devices, several differentially private systems add noise to user inputs locally to ensure privacy [91], but this can reduce the accuracy. Some assume a trusted third party, which only adds noise to the aggregated raw inputs [53], or use MPC to enable global differential privacy without a trusted party [220]. While our goal is not to address the security and privacy issue in these solutions, Oort enables

informed participant selection by leveraging the information already available in today's FL, and can reconcile with them (e.g., to deliver improvement under outliers while respecting privacy).

## 6.10 Summary

While today's FL efforts have been optimizing the statistical model and system efficiency by reinventing traditional ML designs, the participant selection mechanisms they rely on underperform for federated training and testing, and fail to enforce diverse data selection criteria. In this chapter, we present Oort to enable guided participant selection for FL developers. Compared to existing mechanisms, Oort achieves large speedups in time-to-accuracy performance for federated training by picking clients with high statistical and system utility, and it allows developers to specify their selection criteria on data while efficiently serving their requirements on data distribution during FL testing, even at the scale of millions of clients.

# CHAPTER 7

# Conclusions

Over the past decade, skyrocketing data volumes, growing hardware capabilities, and the revolution in ML theory have propelled the last leap forward in ML development. However, enabling the next leap is facing scaling limits in hardware resources and data due to the plateauing of hardware improvement, while resource demands and data volumes continue to grow exponentially. How should we design systems to support the next leap?

This dissertation shows that the answer hinges on recognizing that not all ML components contribute equally to ML performance, whereby we can embrace the minimalist philosophy – reducing complexity and eliminating bloating features – to co-design ML, systems, and networking. The minimalist systems introduced in this thesis span different stages of ML development, minimizing ML resource demands and the need for data collection without sacrificing ML performance. Our real-world deployments demonstrate that these minimalist systems not only improve ML efficiency by orders of magnitude but also enable new advances in theory and applications up to the planetary scale.

In the rest of this chapter, we summarize a few of the lessons distilled from this work, elaborate on its broader impacts, and finally outline several avenues for future work.

## 7.1 Lessons Learned

**Leveraging Cross-Layer Optimizations** The main thread of this dissertation is discerning the impact of underlying system execution on ML performance and leveraging this knowledge to improve efficiency. In this process, we found that there exists a large headroom for improvement through cross-layer optimizations, primarily due to the scaling law in ML [149] – even slight accuracy improvements beyond certain thresholds can lead to substantial increases in system loads (e.g., exponential growth in model size and training data). Despite – or perhaps, because of – the fast evolution of ML algorithms, software, and hardware support, the ever-growing ML accuracy demands have led to a scaling process yet without a well-crafted tradeoff between accuracy and costs. Incidentally, the misalignment in the evolving speed of different ML layers further exacerbates this

issue, leading to bloating system execution. For example, ML algorithms generally evolve faster than the corresponding systems support.

As developing and democratizing large-scale and multi-modal ML models is likely to be the biggest challenge in the coming decade, a multi-disciplinary approach is recommended to align different layers rather than solely tailoring systems to fixed ML workloads.

**Embracing Heterogeneity** Today's ML, both cloud-based and beyond, is witnessing increasing heterogeneities in model, data, and the underlying execution environments. This arises from the proliferation of specialized accelerators and their legacy (e.g., hardware degradation), diverse user demands, and multi-tenancy deployments (e.g., resource contention), alongside the inherent heterogeneity of data [163] and model performance [159]. Over time, we found that this heterogeneity implies varying importance of ML components, often following a long-tail distribution, thereby enabling us to cherry-pick components.

In tandem with the aforementioned misalignment, we believe that this heterogeneity is expected to expand, necessitating ML systems to *automatically* customize optimizations for different (cohorts of) users and environments, thereby reducing the associated costs.

**Capitalizing on ML Dynamics** In the presence of heterogeneities, existing ML systems have primarily focused on optimizing for specific user-defined resources, data, and many other settings. This static space simplifies system designs, so most of the existing system optimizations take place either pre-training (e.g., determining model parallelism) or post-training (e.g., compressing models). However, we are reaching a tipping point where models are becoming more gigantic and dynamic. For example, LLMs may employ mixture-of-expert (MoE) models to dynamically route model input to different expert models [213]. Consequently, current static and user-specified paradigms are no longer tenable.

The future calls for *systems that can collect runtime feedback and adaptively optimize for peak efficiency in real-time*, supporting the growing demand for "no-code ML" – developers focus on desired outcomes rather than the intricacies of achieving them. These self-adaptive systems can also unlock huge optimization potential, as they tap into the heterogeneous importance of various ML components to maximize overall performance. For example, when routing model input to different expert models, can we consider their system load and training convergence to find the sweet spot of ML and system efficiency?

**Simplicity Increases Applicability** Lastly, the growing adoption of our minimalist systems in the open-source community and industry is attributed to their simplicity. Our implementations are primarily built atop widely-used frameworks in today's landscape and provide familiar interfaces. For

example, Sol replaces the core execution engine of Apache Spark within the execution layer [278]; ModelKeeper serves as a lightweight plugin to Ray, offering efficient model management [192]; FedScale acts as an umbrella that bridges cloud ML and edge FL. These systems themselves compose with each other and form a minimalist software stack that can be easily deployed in ML lifecycle, while benefiting from ongoing advances without reinventing the wheel.

## 7.2 Broader Impact

**Impact on Industry** The software artifacts developed throughout this dissertation, largely open-source, have garnered widespread attention and contributors from academic and industrial circles, such as HKUST, University of Toronto, and LinkedIn. In terms of large-scale industrial deployments, FedScale and Oort are deployed at LinkedIn to support efficient federated learning. Recently, Oort is adopted by Cisco, while AdaEmbed is deployed at Meta, catering to millions of users.

**Outreach and Societal Implications** Ensuring data privacy while personalizing everything with ML is likely to be the biggest challenge in the coming decade. This dissertation open-sources a minimalist software stack, democratizing sustainable, pervasive, and private ML without having to maintain large software engineering teams that only the likes of Google and Intel can afford. It's poised to influence sectors beyond tech; for example, the University of Michigan's medical school leverages FedScale for patient studies, ensuring data privacy; MakeMyLetters is integrating FedScale, aiming to support a K-12 education app to help kid's handwriting.

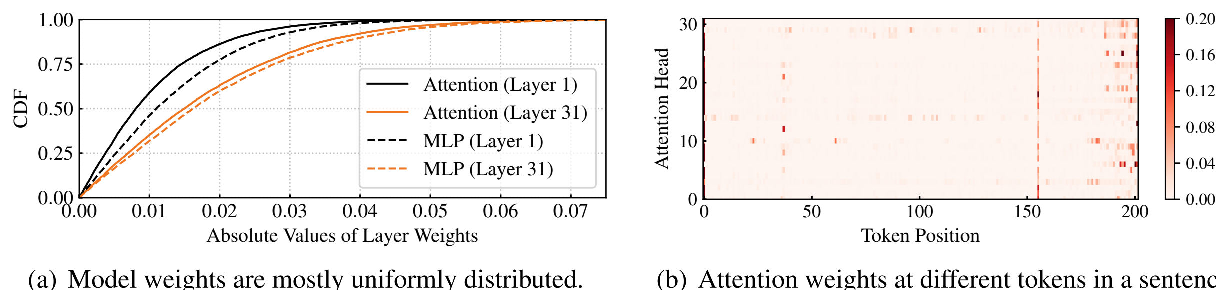
## 7.3 Future Work

In this section, we identify several avenues for future work. Whenever possible, we highlight the primary challenges and possible solutions to address them.

### 7.3.1 Virtual Cloud for Pervasive Data Analytics

The data we want to make sense of is dispersed beyond traditional data centers. Although cloud computing provides manageable resource provisioning, it grapples with costly data collection and limited insights into sensitive data. Computing on the edge, however, is restricted by device resources and faces challenges in orchestrating massive clients over the slow Internet. Consequently, analytics on these data calls for methods that enable collaboration between the cloud and the edge in the near future.

While federated learning sets a promising stage, realizing collaborative computing in the wild remains intricate and open-ended. Challenges range from determining which edge data to collect




(a) Model weights are mostly uniformly distributed. (b) Attention weights at different tokens in a sentence.

Figure 7.1: LLaMA model weights are less sparse, but runtime attention weights are highly sparse.

to the cloud (e.g., for edge video analysis, augmented/virtual reality), how to minimize changes to established applications, design and schedule tasks for faster execution (e.g., splitting models between the cloud and the edge and co-training the model), personalize for large device populations (e.g., using different models [174]), and find the sweet spot between edge privacy and the fidelity of results (e.g., via differential privacy), are all areas needing exploration. Our preliminary results show that solving these at scale requires significant effort in application-aware resource scheduling [273] and on-device adaptation [126].

### 7.3.2 Systems for Self-Adaptive ML

The adaptive nature of ML, i.e., a stochastic process that learns an approximated function for the data, presents a unique opportunity to co-design ML algorithms, models, software, and hardware to alter the semantics of model execution. We should fundamentally rethink existing system designs to make use of such adaptability in ML and prospective big data analytics.

**Adaptive Model Training** One possible direction involves leveraging the heterogeneous importance of data and model components to overall performance. Could we identify and prioritize the use of more important data to accelerate model convergence during training [291]? What happens if the model input data has dependencies (e.g., the input nodes of graph neural networks have different neighboring nodes)? Similarly, is it possible to identify and fuse the less important pieces of the model to reduce execution without losing final model accuracy (e.g., compress the model gradients or perform partial model aggregation *adaptively* in terms of training convergence)? For example, LLMs often report a high sparsity during execution (around 80% sparsity, as shown in Figure 7.1(b)), suggesting potential reductions in computational and memory requirements.

**Adaptive Model Inference** Similarly, ML serving is undergoing a paradigm shift with models catering to a wide range of tasks. For example, LLMs are capable of handling tasks such as text classification, translation, and coding. Understandably, tasks often vary in terms of complexity,

user's quality of service agreements (e.g., latency and accuracy). So routing all requests to a single LLM can result in substantial serving costs without delivering significant accuracy benefits.

As such, we can develop an inference scheduler that *adaptively* routes user requests to a zoo of serving models, in order to reduce ML deployment costs for providers while preserving service quality for users. This will be achieved through the following two designs:

- *Efficient Model Personalization*: Developing a zoo of models is the first step towards cherry-picking models. However, training large models, in particular, is extremely expensive. We can develop an automated training warmup system to transform the model weights of pretrained models to accelerate model training. This will perform structure-aware model matching, such as measuring the L1 distance of embedding weights and the number of transformable weights in the model graph, to determine which pretrained model to repurpose. Additionally, when transforming models across datasets, we will consider the data's similarity.
- *Effective Model Selection*: Selecting which model to use for individual user requests is challenging due to the inherent heterogeneity of tasks and even of different requests within the same task (e.g., unpredictable LLM output length). We can take inspiration from the large volume of daily requests and the fact that requests often exhibit similarities, whereby we can apply latent-factor collaborative filtering, a lightweight ML technique commonly used in recommender systems. It characterizes requests in terms of latent (i.e., "hidden") features and then learns models' performance on requests from past requests and user feedback. Thereafter, we can perform request scheduling while considering the instantaneous workload of models.

### 7.3.3 ML for Self-Adaptive Systems

As workloads and system complexity increase, it becomes difficult to optimize systems. This is further exacerbated by the dynamic nature of system deployment (e.g., the difficulty in predicting new events in scheduling) and the large overhead in finding the optimal solution (e.g., scheduling 1000 jobs can take tens of minutes [198]). Existing solutions often resort to long-standing heuristics for faster speed, leading to inferior performance.

In fact, optimizations over time and at a large scale are producing enormous amounts of data that can be exploited using ML techniques. We could use ML to augment system optimizations along three lines. First, *can we achieve long-term benefits by taking the future into account?* For example, in designing the ML serving pipeline, we may use multi-armed bandit optimization to determine the best collection of models on the fly for better accuracy and low latency, even if the data distribution changes over time. Second, *can learning from past decisions help reduce the search ambit toward the optimal solution?* As an example, we may apply contextual bandit to learn which model to repurpose in ModelKeeper, with feedback from the warmup improvement, in order

to consider advanced transfer learning techniques (e.g., knowledge distillation). Third, *how can we design an elastic ML technique that can quickly transfer knowledge to new environments (like foundation models)?* This is very likely to be the biggest challenge for applying ML to system optimizations in the coming decade, particularly considering the dynamic nature and multi-tenancy of system deployment (e.g., in serverless computing).

# APPENDIX A

# Sol Appendix

## A.1 Benefits of Pipelining

To investigate the benefit of pipelining the scheduling and data fetch of a downstream task, we assume zero queuing time and emulate the inter-site coordinations with our measured latency across 44 datacenters on AWS, Azure and Google Cloud. We define the improvement as the difference between the duration of the pull-based model and that of our proposed push-based model for every data fetch (Figure 2.8). Our results in Figure A.1 report we can achieve an average improvement of 153 ms.

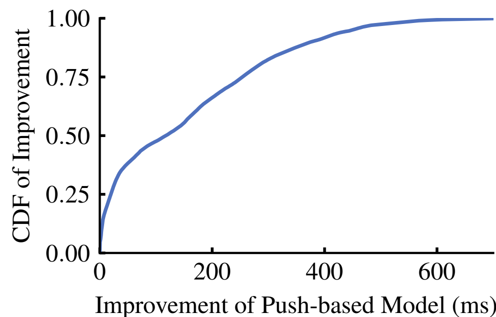
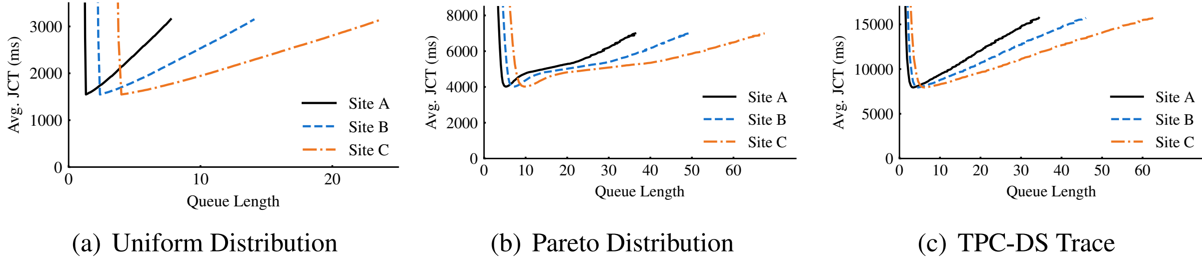



Figure A.1: Improvement of Push-based Model in (§2.5.2) .

Understandably, such benefit is more promising in consideration of the task queuing time, as pushing the remote data is even pipelined with the task spin-wait for scheduling.

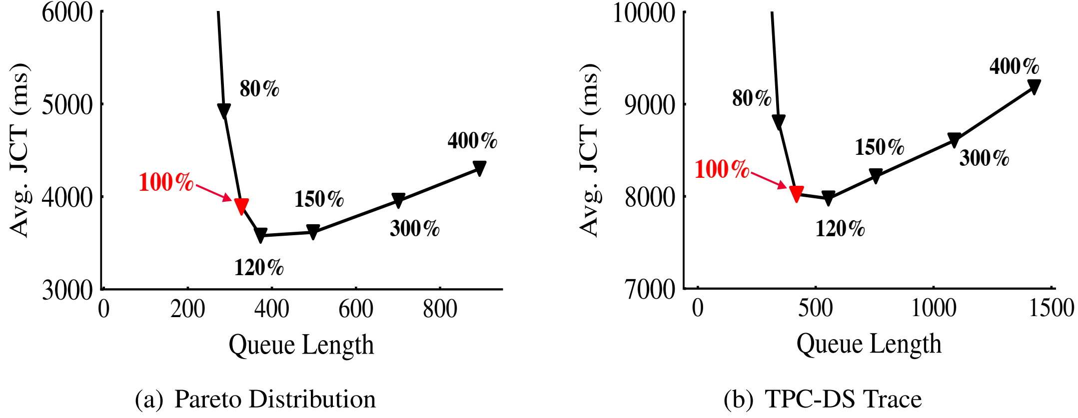
## A.2 Impact of Queue Length

We quantify the impact of queue size with the aforementioned three workloads (in (§2.5.2)). In this experiment, we analyze three distinct sites, where the network latency from Site A, B and C to the centralized coordinator is 130 ms, 236 ms and 398 ms, respectively. As shown in Figure A.2, queuing up too many or too few tasks can hurt job performance.




(a) Uniform Distribution (b) Pareto Distribution (c) TPC-DS Trace

Figure A.2: Impact of Queue Size on Job Completion Time (JCT).




(a) Pareto Distribution (b) TPC-DS Trace

Figure A.3: JCT performance with different utilization targets.

## A.3 Determining the Queue Length

**Lemma 1.** *For a given utilization level $\delta$ and confidence interval $\alpha$ (i.e., $\Pr[Util. > \delta] > \alpha$), the queue length $K$ for $S$ working slots satisfies:*

$$K \geq M - \frac{C}{2} + \frac{1}{2D_{avg}}\sqrt{(CD_{avg})^2 - 4MCD_{avg}} \tag{A.1}$$

*where $D_{5th}$ and $D_{95th}$ denote the 5th and 95th percentile of task durations respectively, and $D_{avg}$ denotes the average task duration. $M = \frac{\delta RTT \times S}{D_{avg}}$, $C = \frac{1}{2}\left(\frac{D_{95th} - D_{5th}}{D_{avg}}\right)^2 \cdot \log \alpha$. The first term $M$ depicts the expectation, while the rest capture the skewness of distributions and confidence.*

This is true for any distribution of task durations. Unfortunately, we omit the proof for brevity. $D_{avg}$, $D_{5th}$ and $D_{95th}$ are often stable in a large cluster, and thus available from the historical data. $\alpha$ is the configurable confidence level, which is often set to 99% [188, 138], and $\delta$ is set to $\geq 100\%$ to guarantee full utilization. Note that from Eq. D.3, when task durations follow the uniform distribution, our model ends up with the expectation $M$. Similarly, when the RTT becomes negligible, this outputs zero queue length.

Our evaluations show that this model can provide encouraging performance, wherein we repeated

the experiments with the workloads mentioned in §2.5.3. We provide the results for workloads with Pareto and TPC-DS distributions by injecting different utilizations in theory, since results for the Uniform distribution are concentrated on a single point (i.e., the expectation $M$). As shown in Figure A.3, the queue length with 100% utilization target locates in the sweet spot of JCTs.

Note that when more task information is available, one can refine this range better; e.g., the bound of Eq. (D.3) can be improved with Chernoff's inequality when the distribution of task durations is provided.

# APPENDIX B

# ModelKeeper Appendix

## B.1 ModelKeeper Analysis

### B.1.1 Design Criteria

At its core, ModelKeeper performs informed weight initialization for DNN models by repurposing a well-trained model's weights. Intuitively, we note that

- This can be viewed as an instance of existing transfer learning (TL), where we transform the weight of a model on one dataset to train on "another" dataset (i.e., the warmup model has not viewed that dataset before its training takes place) [43]. More subtly, it is a simplified and complementary TL scenario under homogeneous data distribution and features, so existing TL theories can be applied to validate our effectiveness too.
- ModelKeeper transformation is an informed weight initialization, thus a special case of random initialization. As the rest of the training remains the same, the model should be able to reach similar final accuracy when the model converges.

**Why ModelKeeper Can Help Convergence?** We next present the theoretical analysis of model convergence to show why ModelKeeper can achieve faster convergence.

**Corollary 1.1.** *(Theorem 1 in [275]). Under widely-used DL assumptions (1) Smoothness: loss function $f(\boldsymbol{w})$ is L-Lipschitz smooth; (2) Bounded gradient variances: with constants $G > 0$, $\sigma > 0$, we assume $\mathbb{E}[\|\nabla f(\boldsymbol{w})\|^2] \leq G^2$ and $\mathbb{E}[\|\nabla F(\boldsymbol{w})\|^2 - \nabla f(\boldsymbol{w})\|^2] \leq \sigma^2$; and (3) Unbiased estimation: on mini-batch $\xi$, we have $\mathbb{E}_{\xi|\boldsymbol{w}}[\nabla f(\boldsymbol{w})] = \nabla F(\boldsymbol{w})$.*

*With learning rate $\gamma$ to $0 < \gamma \leq \frac{1}{L}$, then for iteration $T$, the model training convergence rate is:*

$$\frac{1}{T}\sum_{t=1}^{T}\mathbb{E}[\|\nabla f(\boldsymbol{w}^{t-1})\|^2] \leq \frac{2}{\gamma T}(f(\boldsymbol{w}^0) - f^*) + 4\gamma^2 I^2 G^2 L^2 + \frac{L}{N}\gamma\sigma^2$$

Where $N$ is the number of workers in synchronized data-parallel training, $f^*$ is the optimal training loss.

From Corollary 1.1, we can notice that, for the same model, training achieves faster convergence with a smaller initial loss value $f(\mathbf{w}^0)$ in theory (similar to [272, 202]). Indeed, existing gradient variance reduction techniques in the ML community report a similar theory analysis [40]. Here, we empirically show that the initial training loss of the warmup query model, $f(\mathbf{w}^0)$, will start from some basin of loss curvature (e.g., better accuracy in Figure 3.3 and smaller gradient variance in Figure 3.4), and theoretically analyze why this enables starting from the loss basin in Appendix B.1.2.

Admittedly, weight transformation can be lossy (e.g., due to incomplete matching), which breaks the parent model information. We note that capturing the exact convergence comparison herein is extremely challenging, which indeed is a fundamental open problem even in today's transfer learning [89]. Nevertheless, many empirical analyses have reported consistently encouraging improvement [272], and transfer learning is widely-used. Intuitively, for front tensors that enjoy full information-preserving transformation, we can consider them as a prefix subnet, and this subnet holds the same output as the corresponding parent subnet. Therefore, these tensors can still potentially achieve faster convergence according to Corollary 1.1.

**How to Select Parent Models?** In selecting the parent model, ModelKeeper prioritizes the model with (1) *better model accuracy*: this is because parent models with better accuracy enable smaller initial loss $f(\mathbf{w}^0)$, thus allowing better convergence speed (Corollary 1.1); and (2) *larger architectural similarity* and *prefix preference*: If we dive to the fundamental of model training, the output activations of a specific model tensor $i$ is $\mathbf{y}^i = f_i(\mathbf{y}^{(i-1)T}\mathbf{w}_i + \mathbf{b}_i)$. Here, assuming the front $l-1$ tensor are warmup, while $\mathbf{w}_i$ is randomly initialized. The front subnet still enjoys better convergence, so we prefer a model with architectural similarity to maximize this potential. In the forward training propagation, $\mathbf{w}_i$ leads to cascading information loss to subsequent tensors, so we prioritize the match of prefixes to minimize this loss. On the other hand, training front tensors is more difficult but more transferable, because gradient information becomes less informative as it is backpropagated through more subsequent tensors [89], which requires us to match subsequent tensors as many as possible to curb this divergence to the front tensors in backward propagation.

### B.1.2 Information-Preserving Transformation

ModelKeeper employs width and depth operators to perform structure-aware weight transformation, wherein expanding the parent model performs the same to Net2Net [66]. Net2Net theoretically grounds that expanding transformation (e.g., more convolution channels or new convolution tensors) can preserve the parent model information for a wide range of tensors. Specifically, the depth operator tries to deepen a tensor $\mathbf{y}^i = f_i(\mathbf{y}^{(i-1)T}\mathbf{w}_i + \mathbf{b}_i)$ using two tensors $\mathbf{y}^i = f_i\big(\mathbf{U}^{(i)T} f_i\big(\mathbf{y}^{(i-1)T}\mathbf{w}_i + \mathbf{b}_i\big)\big)$, where $f_i$, $\mathbf{w}_i$, $\mathbf{b}_i$ are the activation function, tensor weights, and

bias vectors, respectively. When matrix $\mathbf{U}$ is initialized to an identity matrix, adding $\mathbf{U}$ preserves the same output of its input tensor if $f_i$ is chosen such that $f_i(\mathbf{U}f_i(\mathbf{v})) = f_i(\mathbf{v})$ for all vectors $\mathbf{v}$. This property, $f_i$, holds for widely-used rectified linear activation in today's DNN models. For example, to insert a new convolution tensor, we should set the convolution kernels to be identity filters. Readers can refer to Net2Net [66] for the theoretical analysis for the width operator. As such, in expanding the parent model, we may preserve the full parent model information.

# APPENDIX C

# FedScale Appendix

## C.1 Experiment Setup

**Scalability Evaluations** We evaluate the scalability of FedScale Runtime, FedML (GitHub *commit@2ee0517*) and Flower (v0.17.0 atop Ray v1.9.2) using a cluster with 10 GPU nodes. Each GPU node has a P100 GPU with 12GB GPU memory and 192GB RAM. We train the ShuffleNet-V2 model on the OpenImage dataset. We set the minibatch size of each participant to 32, and the number of local steps $K$ to 20, which takes around 2800MB GPU memory for each model training. As such, we allow each GPU node to run 4 processes in benchmarking these three frameworks.

**Evaluation Setup** Applications and models used in our evaluations are widely used on mobile devices. We set the minibatch size of each participant to 32, and the number of local steps $K$ to 20. We cherry-pick the hyper-parameters with grid search, ending up with an initial learning rate 0.04 for CV tasks and 5e-5 for NLP tasks. The learning rate decays by 0.98 every 10 training rounds. These settings are consistent with the literature. More details about the input dataset are available in Appendix C.2.

## C.2 Introduction of FedScale Datasets

FedScale currently has 20 realistic federated datasets across a wide range of scales and task categories (Table C.1). Here, we provide the description of some representative datasets.

**Google Speech Commands.** A speech recognition dataset [252] with over ten thousand clips of one-second-long duration. Each clip contains one of the 35 common words (e.g., digits zero to nine, "Yes", "No", "Up", "Down") spoken by thousands of different people.

**OpenImage.** OpenImage [11] is a vision dataset collected from Flickr, an image and video hosting service. It contains a total of 16M bounding boxes for 600 object classes (e.g., Microwave oven). We clean up the dataset according to the provided indices of clients.

| Category | Name | Data Type | #Clients | #Instances | Example Task |
|---|---|---|---|---|---|
| **CV** | *iNature* [15] | Image | 2,295 | 193K | Classification |
| | *FEMNIST* [73] | Image | 3,400 | 640K | Classification |
| | *OpenImage* [11] | Image | 13,771 | 1.3M | Classification, Object detection |
| | *Google Landmark* [256] | Image | 43,484 | 3.6M | Classification |
| | *Charades* [228] | Video | 266 | 10K | Action recognition |
| | *VLOG* [95] | Video | 4,900 | 9.6K | Classification, Object detection |
| | *Waymo Motion* [92] | Video | 496,358 | 32.5M | Motion prediction |
| **NLP** | *Europarl* [156] | Text | 27,835 | 1.2M | Text translation |
| | *Blog Corpus* [226] | Text | 19,320 | 137M | Word prediction |
| | *Stackoverflow* [28] | Text | 342,477 | 135M | Word prediction, Classification |
| | *Reddit* [25] | Text | 1,660,820 | 351M | Word prediction |
| | *Amazon Review* [186] | Text | 1,822,925 | 166M | Classification, Word prediction |
| | *CoQA* [217] | Text | 7,685 | 116K | Question Answering |
| | *LibriTTS* [282] | Text | 2,456 | 37K | Text to speech |
| | *Google Speech* [252] | Audio | 2,618 | 105K | Speech recognition |
| | *Common Voice* [6] | Audio | 12,976 | 1.1M | Speech recognition |
| **Misc ML** | *Taxi Trajectory* | Text | 442 | 1.7M | Sequence prediction |
| | *Taobao* [30] | Text | 182,806 | 20.9M | Recommendation |
| | *Puffer Streaming* [264] | Text | 121,551 | 15.4M | Sequence prediction |
| | *Fox Go* [9] | Text | 150,333 | 4.9M | Reinforcement learning |

Table C.1: Statistics of FedScale datasets. FedScale has 20 realistic client datasets, which are from the real-world collection, and we partitioned each dataset using its real client-data mapping.

**Reddit and StackOverflow.** Reddit [25] (StackOverflow [28]) consists of comments from the Reddit (StackOverflow) website. It has been widely used for language modeling tasks, and we consider each user as a client. In this dataset, we restrict to the 30k most frequently used words, and represent each sentence as a sequence of indices corresponding to these 30k frequently used words.

**VLOG.** VLOG [95] is a video dataset collected from YouTube. It contains more than 10k Lifestyle Vlogs, videos that people purportedly record to show their lives, from more than 4k actors. This dataset is aimed at understanding everyday human interaction and contains labels for scene classification, hand-state prediction, and hand detection tasks.

**LibriTTS.** LibriTTS [282] is a large-scale text-to-speech dataset. It is derived from audiobooks in LibriVox project. There are 585 hours of read English speech from 2456 speakers at a 24kHz

sampling rate.

**Taobao.** Taobao Dataset [30] is a dataset of click rate prediction about display Ad, which is displayed on the website of Taobao. It is composed of 1,140,000 users ad display/click logs for 8 days, which are randomly sampled from the website of Taobao.

**Waymo Motion.** Waymo Motion [92] is composed of 103,354 segments each containing 20 seconds of object tracks at 10Hz and map data for the area covered by the segment. These segments are further broken into 9 second scenarios, and we consider each scenario as a client.

**Puffer Streaming.** Puffer is a Stanford University research study about using machine learning (e.g., reinforcement learning [183]) to improve video-streaming algorithms: the kind of algorithms used by services such as YouTube, Netflix, and Twitch. Puffer dataset [264] consists of 15.4M sequences of network throughput on edge clients over time.

### C.3 Comparison with Existing FL Benchmarks

In this section, we compare FedScale with existing FL benchmarks in more details.

**Data Heterogeneity** Existing benchmarks for FL are mostly limited in the variety of realistic datasets for real-world FL applications. Even they have various datasets (e.g., LEAF [59]) and FedEval [61]), their datasets are mostly synthetically derived from conventional datasets (e.g., CIFAR) and limited to quite a few FL tasks. These statistical client datasets can not represent realistic characteristics and are inefficient to benchmark various real FL applications. Instead, FedScale provides 20 comprehensive realistic datasets for a wide variety of tasks and across small, medium, and large scales, which can also be used in data analysis to motivate more FL designs.

**System Heterogeneity** The practical FL statistical performance also depends on the system heterogeneity (e.g., internet bandwidth [158] and client availability), which has inspired lots of optimizations for FL system efficiency. However, existing FL benchmarks have largely overlooked the system behaviors of FL clients, which can produce misleading evaluations, and discourage the benchmarking of system efforts. To emulate the heterogeneous system behaviors in practical FL, FedScale incorporates real-world traces of mobile devices, associates each client with his system speeds, as well as the availability. Moreover, we develop a more efficient evaluation platform, FedScale Runtime, to automate FL benchmarking.

**Scalability** Existing frameworks, perhaps due to the heavy burden of building complicated system support, largely rely on the traditional ML architectures (e.g., the primitive parameter-server architecture of Pytorch). These architectures are primarily designed for the traditional large-batch training on a number of workers, and each worker often trains a single batch at a time. However, this is ill-suited to the simulation of thousands of clients concurrently: (1) they lack tailored system implementations to orchestrate the synchronization and resource scheduling, for which they can easily run into synchronization/memory issues and crash down; (2) their resource can be under-utilized, as FL evaluations often use a much smaller batch size.

Tackling all these inefficiencies requires domain-specific system designs. Specifically, we built a cluster resource scheduler that monitors the fine-grained resource utilization of machines, queues the overcommitted simulation requests, adaptively dispatches simulation requests of the client across machines to achieve load balance, and then orchestrates the simulation based on the client virtual clock. Moreover, given a much smaller batch size in FL, we maximize the resource utilization by overlapping the communication and computation phrases of different client simulations. The former and the latter make FedScale more scalable across machines and on single machines, respectively.

Empirically, we have run the 20-GPU set up on different datasets and models in Figure 5.10 and Table C.2, and are aware of at least one group who ran FedScale Runtime with more than 60 GPUs [163].

| | Eval. Duration/Round vs. # of Clients/Round | | | |
|---|---|---|---|---|
| | 10 | 100 | 1K | 10K |
| **FedScale** | **0.03 min** | **0.16 min** | **1.14 min** | **10.9 min** |
| FedML | 0.58 min | 4.4 min | fail to run | fail to run |
| Flower | 0.05 min | 0.23 min | 2.4 min | fail to run |

Table C.2: FedScale is more scalable and faster. Image classification on iNature dataset using MobileNet-V2 on 20-GPU setting.

**Modularity** As shown in Table 5.1, some existing frameworks (e.g., LEAF and FedEval) do not provide user-friendly modularity, which requires great engineering efforts to benchmark different components, and we recognize that FedML and Flower provide the API modularity too.

On the other hand, FedScale Runtime's modularity for easy deployments and broader use cases is not limited to APIs (Figure 5.8): (1) FedScale Runtime Data Loader: it simplifies and expands the use of realistic datasets. e.g., developers can load and analyze the realistic FL data to motivate new algorithm designs, or imports new datasets/customize data distributions in FedScale evaluations; (2) Client simulator: it emulates the system behaviors of FL clients, and developers can customize their system traces in evaluating the FL system efficiency too; (3) Resource manager: it hides the system complexity in training large-scale participants.

```
1  from fedscale.core.client_manager import ClientManager
2  import Oort
3
4  class Customized_ClientManager(ClientManager):
5    def __init__(self, *args):
6      super().__init__(*args)
7      self.oort_selector = Oort.create_training_selector(*args)
8
9    # Replace default client selection algorithm w/ Oort
10   def select_participants(self, numOfClients, cur_time, feedbacks):
11     # Feed Oort w/ execution feedbacks
12     oort_selector.update_client_info(feedbacks)
13     selected_clients = oort_selector.select_participants(
14       numOfClients, cur_time)
15
16     return selected_clients
```

Figure C.1: Evaluate client selection algorithm [163].

```
1  from fedscale.core.client import Client
2
3  class Customized_Client(Client):
4  # Customize the training on each client
5    def train(self,client_data,model,conf):
6        # Get the training result from
7        # the default training component
8      training_result = super().train(
9          client_data, model, conf)
10
11     # Implementation of compression
12     compressed_result = compress_impl(
       training_result)
13     return compressed_result
```

Figure C.2: Evaluate model compression [222].

```
1  from fedscale.core.client import Client
2
3  class Customized_Client(Client):
4  # Customize the training on each client
5    def train(self,client_data,model,conf):
6      training_result = super().train(
7          client_data, model, conf)
8
9      # Clip updates and add noise
10     secure_result = secure_impl(training_result)
11     return secure_result
```

Figure C.3: Evaluate security enhancement [232].

## C.4 Examples of New Plugins

In this section, we demonstrate examples to show the ease of integrating today's FL efforts in FedScale evaluations.

At its core, FedScale Runtime provides flexible APIs on each module so that the developer can access and customize methods of the base class. Table 5.3 illustrates some example APIs that can facilitate diverse FL efforts. Note that FedScale Runtime will automatically integrate new plugins into evaluations, and then produces practical FL metrics. Figure C.1 demonstrates that we can easily evaluate new client selection algorithms, Oort [163], by modifying a few lines of the `clientManager` module. Similarly, Figure C.2 and Figure C.3 show that we can extend the basic `Client` module to apply new gradient compression [222] and enhancement for malicious attack [232], respectively.

**Comparison with other work** Figure C.4 shows the same evaluation of gradient compression [222] as that in Figure C.2 using flower [50], which requires much more human efforts than Figure C.2. The gray components in figure C.4 requires implementation. Flower falls short in

```
1     import flwr as fl
2
3     def get_config_fn():
4       # Implementation of randomly selection
5       client_ids = random_selection()
6       config = {"ids": client_ids}
7       return config
8
9     # Customized Strategy
10    strategy = CustomizedStrategy(
11      on_fit_config_fn=get_config_fn())
12
13    fl.server.start_server(
14      config={"num_rounds":args.round},
15      strategy=strategy)
16
```

```
1     import flwr as fl
2
3     class Customized_Client():
4       def fit(self, config, net):
5         # Customization of client data
6         trainloader = select_dataset(
7           config["ids"][args.partition])
8         train(net, trainloader)
9         compressed_result = self.get_parameters()
10        # Implementation of compression
11        compressed_result = compress_impl(
12            training_result)
13        return compressed_result
14
15    fl.client.start_numpy_client(
16      args.address, client=CustomizedClient())
17
```

Figure C.4: Evaluate model compression with Flower [50]. The developer needs to implement the functions in grey by his own. Note that each function can take tens of lines of code.

providing APIs for passing metadata between client and server, for example client id, which makes server and running workers client-agnostic. To customized anything for clients during FL training, developers have to go through the source code and override many components to share client meta data between server and workers.

# APPENDIX D

# Oort Appendix

## D.1 Proving Benefits of Statistical Utility

We follow the proof of importance sampling to show the advantage of our statistical utility in theory. The convergence speed of Stochastic Gradient Descent (SGD) can be defined as the reduction $R$ of the divergence of model weight $\mathbf{w}$ from its optimal $\mathbf{w}^*$ in two consecutive round $t$ and $t+1$ [153, 146] :

$$R = \mathbb{E}\big[\parallel \mathbf{w_t} - \mathbf{w}^* \parallel_2^2 - \parallel \mathbf{w_{t+1}} - \mathbf{w}^* \parallel_2^2 \big] \tag{D.1}$$

**How does oracle sampling help in theory?** If the learning rate of SGD is $\eta$ and we use loss function $L$ to measure the training loss between input features $x$ and the label $y$, then $\mathbf{w_{t+1}} = \mathbf{w_t} - \eta \nabla L(\mathbf{w_t}(x_i), y_i)$. We now set the gradient $G_t = \nabla L(\mathbf{w_t}(x_i), y_i)$ for brevity, then from Eq. (D.1):

$$\begin{aligned}
R &= -\mathbb{E}\big[(\mathbf{w_{t+1}} - \mathbf{w}^*)^T(\mathbf{w_{t+1}} - \mathbf{w}^*) - (\mathbf{w_t} - \mathbf{w}^*)^T(\mathbf{w_t} - \mathbf{w}^*)\big] \\
&= -\mathbb{E}\big[\mathbf{w_{t+1}^T}\mathbf{w_{t+1}} - 2\mathbf{w_{t+1}}\mathbf{w}^* - \mathbf{w_t^T}\mathbf{w_t} + 2\mathbf{w_t}\mathbf{w}^*\big] \\
&= -\mathbb{E}\big[(\mathbf{w_t} - \eta\mathbf{G_t})^T(\mathbf{w_t} - \eta\mathbf{G_t}) + 2\eta G_t^T\mathbf{w}^* - \mathbf{w_t}^T\mathbf{w_t}\big] \\
&= -\mathbb{E}\big[-2\eta(\mathbf{w_t} - \mathbf{w}^*)G_t + \eta^2 G_t^T G_t\big] \\
&= 2\eta(\mathbf{w_t} - \mathbf{w}^*)\mathbb{E}[G_t] - \eta^2\mathbb{E}[G_t]^T\mathbb{E}[G_t] - \eta^2 Tr(\mathbb{V}[G_t])
\end{aligned} \tag{D.2}$$

It has been proved that optimizing the first two terms of Eq. (D.2) is intractable due to their joint dependency on $\mathbb{E}[G_t]$, however, one can gain a speedup over random sampling by intelligently sampling important data bins to minimize $Tr(\mathbb{V}[G_t])$ (i.e., reducing the variance of gradients while respecting the same expectation $\mathbb{E}[G_t]$) [146, 153]. Here, the oracle is to pick bin $B_i$ with a probability proportional to its importance $|B_i|\sqrt{\frac{1}{|B_i|}\sum_{k\in B_i} \parallel G(k) \parallel^2}$, where $\parallel G(k) \parallel$ is the L2-norm of the unique sample $k$'s gradient $G(k)$ in bin $B_i$ (Please refer to [104] for detailed proof).

**How does loss-based approximation help?** We have shown the advantage of importance sampling by sampling the larger gradient norm data, and next we present the theoretical insights that motivates our loss-based utility design.

**Corollary 1.2.** *(Theorem 2 in [152]). Let* $\| G(k) \|$ *denote the gradient norm of any sample* $k$*,* $M = \max \| G(k) \|$*. There exists* $K > 0$ *and* $C < M$ *such that* $\frac{1}{K} L(\mathbf{w}(x_k), y_k) + C \geq \| G(k) \|$*.*

This corollary implies that a bigger loss leads to a large upper bound of the gradient norm. To sample data with a larger gradient norm, we prefer to pick the one with bigger loss. Moreover, it has been empirically shown that sampling high loss samples exhibits similar variance reducing properties to sampling according to the gradient norm, resulting in better convergence speed compared to naive random sampling [153]. Readers can refer to the recent FL work [69] for the detailed proof.

By taking account of the oracle and the effectiveness of loss-based approximation, we propose our loss-based statistical utility design, whereby we achieve the close to upper-bound statistical performance (§6.8.2.2).

## D.2 Privacy Concern in Collecting Feedbacks

Depending on different requirements on privacy, we elaborate on how Oort respects privacy while outperforming existing mechanisms (§6.5.2).

**Compute aggregated training loss on clients locally.** Our statistical utility of a client relies on the aggregated training loss of all samples on that client. The training loss of each sample measures the prediction uncertainties of model on every possible output (e.g., category), and even the one with a correct prediction can generate non-zero training loss [86]. So it does not reveal raw data inputs. Moreover, it does not leak the categorical distribution either, since samples of the same category can own different training losses due to their heterogeneous input features. Note that even when we consider homogeneous input features, a high total loss can be from several high loss samples, or many moderate loss samples.

**Add noise to hide the real aggregated loss.** Even when clients have very stringent privacy concerns on their aggregated loss, clients can add noise to the exact value. Similar to the popular differentially private FL [98], clients can disturb their real aggregated loss by adding Gaussian noise (i.e., noise from the Gaussian distribution). In fact, Oort can tolerate this noisy statistical utility well, owing to its probabilistic selection from the pool of high-utility clients, wherein teasing apart the top-k% utility clients from the rest is the key.

We first prove that Oort is still very likely to select high-utility clients even in the presence of noise. To pick $K$ participants out of $N$ all feasible clients, there are totally $\binom{N}{K}$ possible combinations. We denote these combinations as $X_i$ and sort them $X_1 \leq \ldots \leq X_n$ by the ascending order of total utility. Adding noise to each client ends up with an accumulated noise on $X_i$. Thereafter, $X_i$ turns to random variables $\mathbf{X}_i$ that follow the distribution of accumulated noise. Specifically, distribution of $\mathbf{X}_i$ is equivalent to shifting the distribution of noise horizontally by a constant $X_i$. Given that noise added to $X_i$ follows the same distribution, (i) $\mathbf{X}_i$ experiences the same standard deviation for every $i$; (ii) the expectation of $\mathbf{X}_i$ is the sum of $X_i$ and the expectation of noise. Note that adding a constant (i.e., the expectation of noise here) to the inequality does not change its properties, so we still have $\mathbb{E}[\mathbf{X}_1] \leq \ldots \leq \mathbb{E}[\mathbf{X}_n]$. As such, we are more likely to select high-utility clients (i.e., combination $\mathbf{X}_i$ with higher $\mathbb{E}[\mathbf{X}_i]$) in sampling when picking $i$ with the highest value of $X_i$. Moreover, we have show the superior empirical performance of Oort over its counterparts under noise in Section 6.8.2.3.

**Rely on gradient norm of batches.** For case where clients are even reluctant to report the noisy loss, we introduce an alternative statistical utility to drive our exploration-exploitation based client selection. Our intuition is to use the gradient norm of batches to approximate the gradient norm of individual samples $\nabla f(k)$ in the oracle importance $|B_i|\sqrt{\frac{1}{|B_i|}\sum_{k\in B_i} \| \nabla f(k) \|^2}$. In mini-batch SGD, we have

$$\mathbf{w}_{t+1} = \mathbf{w}_t - learning_rate \times \frac{1}{batch_size} \times \sum_{k\in batch} \nabla f(k)$$

where $w_t$ is the model weights at time $t$. Now, we can use the gradient norm of batches (i.e., $\| \mathbf{w}_{t+1} - \mathbf{w}_t \|^2$) to approximate $\| \nabla f(k) \|^2$, and they become equivalent when the batch size is 1. Note that today's FL is already collecting the model updates (i.e., $\mathbf{w}_{t+1} - \mathbf{w}_t$), so we are not introducing additional information. As such, we consider the client with larger accumulated gradient norm of batches to be more important.

We report the empirical performance of this approximation and the loss-based statistical utility using YoGi. As shown in Fig D.1, Oort achieves superior performance over the random selection, and the loss-based utility is better than its counterparts. This is because the approximation accuracy with the norm of batches decreases when using mini-batch SGD, whereas mini-batch SGD is more popular than the single-sample batch in ML.

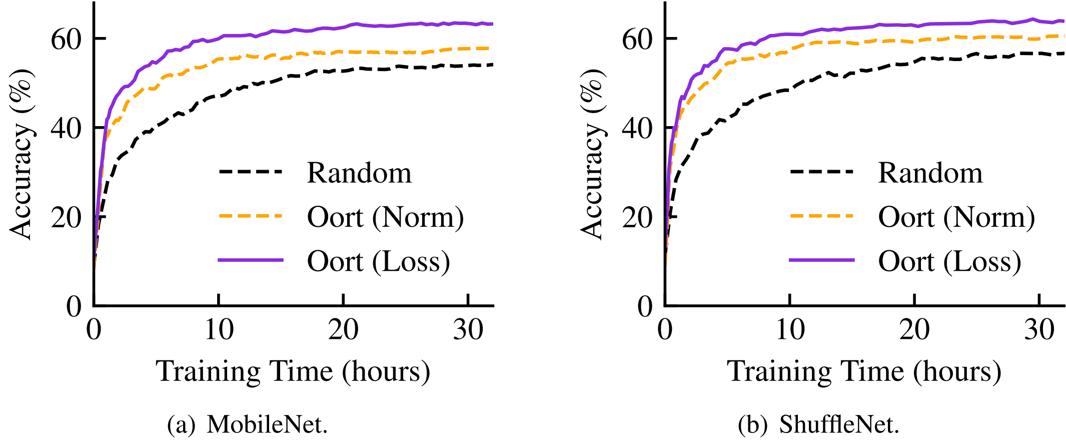



Figure D.1: Oort outperforms with different utility definitions.

### D.3 Determining Size of Participants

We next introduce Lemma 2, which captures how the empirical value of $\bar{X}$ (i.e., average number of samples of participants for category $X$) deviates from the expectation $E[\bar{X}]$ (i.e., average number of samples of all clients) as the size of participants $n$ varies.

**Lemma 2.** *For a given tolerance on deviation $\epsilon$ and confidence interval $\delta$ for category $X$, the number of participants $n$ we need to achieve $Pr[|\bar{X} - E[\bar{X}]| < \epsilon] > \delta$ requires:*

$$n \geq (N+1) \times \frac{1}{1 - \frac{2N}{\log(1-\delta)} \times (\frac{\epsilon}{\max\{X\} - \min\{X\}})^2} \tag{D.3}$$

*where $N$ is the total number of feasible clients, and $\max\{X\}$ and $\min\{X\}$ denote the global maximum and minimum possible number of samples that all clients can hold, respectively.*

Lemma 2 is a corollary of Hoeffding-Serfling Bound [47], and we omit the detailed proof for brevity. Intuitively, when we have an extremely stringent requirement (i.e., $\epsilon \to 0$), we have to include more participants (i.e., $n \to N$). When more information of the client data characteristics is available, one can refine this range better. For example, the bound of Eq. (D.3) can be improved with Chernoff's inequality [47] when the distribution of sample quantities is provided.

Similarly, the multi-category scenario proves to be an instance of multi-variate Hoeffding Bound. Given the developer-specific requirement on each category, the developer may want to figure out how many participants needed to satisfy all these requirements simultaneously (e.g., $Pr[|\bar{X} - E[\bar{X}]| < \epsilon_x \wedge |\bar{Y} - E[\bar{Y}]| < \epsilon_y] > \delta$). More discussions are out of the scope of this paper, but readers can refer to [49] for detailed discussions and a complete solution.

# BIBLIOGRAPHY


[1] AI Benchmark: All About Deep Learning on Smartphones. http://ai-benchmark.com/ranking_deeplearning_detailed.html.

[2] Amazon SageMaker. https://aws.amazon.com/sagemaker/.

[3] Apache Hadoop NextGen MapReduce (YARN). http://goo.gl/etTGA.

[4] Apache spark. https://spark.apache.org/.

[5] California Consumer Privacy Act (CCPA). https://oag.ca.gov/privacy/ccpa.

[6] Common Voice Data. https://commonvoice.mozilla.org/en/datasets.

[7] Federated AI Technology Enabler. https://www.fedai.org/.

[8] FedScale: a scalable and extensible open-source federated learning platform. https://github.com/SymbioticLab/FedScale.

[9] Fox Go Dataset. https://github.com/featurecat/go-dataset.

[10] General Data Protection Regulation (GDPR). https://gdpr-info.eu/.

[11] Google Open Images Dataset. https://storage.googleapis.com/openimages/web/index.html.

[12] Gurobi. https://www.gurobi.com/.

[13] HugeCTR: a high efficiency GPU framework designed for Click-Through-Rate (CTR) estimating training. https://developer.nvidia.com/nvidia-merlin/hugectr.

[14] HuggingFace Model Hub. https://huggingface.co/models?sort=downloads.

[15] iNaturalist 2019. https://sites.google.com/view/fgvc6/competitions/inaturalist-2019.

[16] Kaggle Competition. https://www.kaggle.com/docs/competitions.

[17] Microsoft NNI. https://github.com/microsoft/nni.

[18] MLflow. https://mlflow.org/.

[19] MobiPerf. https://www.measurementlab.net/tests/mobiperf/.

[20] Model Zoo: Discover open source deep learning code and pretrained models. https://modelzoo.co/.

[21] Open Neural Network Exchange (ONNX). https://github.com/onnx/onnx.

[22] Pre-trained machine learning models available in AWS Marketplace. https://aws.amazon.com/marketplace/solutions/machine-learning/pre-trained-models/.

[23] PySyft. https://github.com/OpenMined/PySyft.

[24] PyTorch. https://pytorch.org/.

[25] Reddit Comment Data. https://files.pushshift.io/reddit/comments/.

[26] Sandbox for training deep learning networks. https://github.com/osmr/imgclsmob.

[27] Sort Benchmark. http://sortbenchmark.org/.

[28] Stack Overflow Data. https://cloud.google.com/bigquery/public-data/stackoverflow.

[29] Stanford Puffer. https://puffer.stanford.edu/.

[30] Taobao Dataset. https://tianchi.aliyun.com/dataset/dataDetail?dataId=56&lang=en-us.

[31] TensorFlow Federated. https://www.tensorflow.org/federated.

[32] Termux. https://termux.com/.

[33] TorchRec. https://github.com/pytorch/torchrec.

[34] TPC Benchmark DS (TPC-DS). http://www.tpc.org/tpcds.

[35] TPC Benchmark H (TPC-H). http://www.tpc.org/tpch.

[36] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, and et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

[37] Martín Abadi, Andy Chu, Ian Goodfellow, and et al. Deep learning with differential privacy. In *CCS*, 2016.

[38] Saurabh Agarwal, Ziyi Zhang, and Shivaram Venkataraman. Bagpipe: Accelerating deep recommendation model training. *arXiv preprint arXiv:2202.12429*, 2022.

[39] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. QSGD: Communication-efficient sgd via gradient quantization and encoding. In *NeurIPS*, 2017.

[40] Zeyuan Allen-Zhu and Elad Hazan. Variance reduction for faster non-convex optimization. In *ICML*, 2016.

[41] G. Ananthanarayanan, A. Ghodsi, Scott Shenker, and Ion Stoica. Effective Straggler Mitigation: Attack of the Clones. In *NSDI*, 2013.

[42] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational data processing in Spark. In *SIGMOD*, 2015.

[43] Jordan Ash and Ryan P Adams. On warm-starting neural network training. *NeurIPS*, 2020.

[44] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. In *Machine Learning*, 2002.

[45] Sean Augenstein, H. Brendan McMahan, and et al. Generative models for effective ML on private, decentralized datasets. In *ICLR*, 2020.

[46] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. Pipeswitch: Fast pipelined context switching for deep learning applications. In *OSDI*, 2020.

[47] Rémi Bardenet and Odalric-Ambrym Maillard. *Concentration inequalities for sampling without replacement*. Bernoulli Society for Mathematical Statistics and Probability, 2015.

[48] Brian R. Bartoldson, Ari S. Morcos, Adrian Barbu, and Gordon Erlebacher. The generalization-stability tradeoff in neural network pruning. In *NeurIPS*, 2020.

[49] Patrice Bertail, Emmanuelle Gautherat, and Hugo Harari-Kermadec. Exponential bounds for multivariate self-normalized sums. *Electron. Commun. Probab.*, 13:no. 57, 628–640, 2008.

[50] Daniel J. Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Titouan Parcollet, Pedro Porto Buarque de Gusmao, and Nicholas D. Lane. FLOWER: A friendly federated learning framework. *arXiv preprint arXiv:2007.14390*, 2021.

[51] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *SIGMOD*, 2016.

[52] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. Rock you like a hurricane: Taming skew in large scale analytics. In *EuroSys*, 2018.

[53] Andrea Bittau, Úlfar Erlingsson, and et al. Prochlo: Strong privacy for analytics in the crowd. In *SOSP*, 2017.

[54] Thomas Bittman and Bob Gill. The future shape of edge computing: Five imperatives. https://www.gartner.com/doc/3880015/future-shape-edge-computing-imperatives, 2018.

[55] Keith Bonawitz, Hubert Eichner, and et al. Towards federated learning at scale: System design. In *MLSys*, 2019.

[56] Keith Bonawitz, Vladimir Ivanov, and et al. Practical secure aggregation for privacy-preserving machine learning. In *CCS*, 2017.

[57] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. Data validation for machine learning. In *MLSys*, 2019.

[58] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. 2020.

[59] Sebastian Caldas, Sai Meher, Karthik Duddu, and et al. Leaf: A benchmark for federated settings. *NeurIPS' Workshop*, 2019.

[60] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. In *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2015.

[61] Di Chai, Leye Wang, Kai Chen, and Qiang Yang. FedEval: A benchmark system with a comprehensive evaluation model for federated learning. In *arxiv.org/abs/2011.09655*, 2020.

[62] Shih–Kang Chao, Zhanyu Wang, Yue Xing, and Guang Cheng. Directional pruning of deep neural networks. In *NeurIPS*, 2020.

[63] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. AuTO: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *SIGCOMM*, 2018.

[64] Mingqing Chen, Rajiv Mathews, Tom Ouyang, and Françoise Beaufays. Federated learning of out-of-vocabulary words. In *arxiv.org/abs/1903.10635*, 2019.

[65] Mingqing Chen, Ananda Theertha Suresh, Rajiv Mathews, Adeline Wong, Cyril Allauzen, Francoise Beaufays, and Michael Riley. Federated learning of n-gram language models. In *ACL*, 2019.

[66] Tianqi Chen, Ian J. Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. In *ICLR*, 2016.

[67] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.

[68] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. Wide & deep learning for recommender systems. In *DLRS*, 2016.

[69] Yae Jee Cho, Jianyu Wang, and Gauri Joshi. Towards understanding biased client selection in federated learning. In *AISTATS*, 2022.

[70] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In *SIGCOMM*, 2015.

[71] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with Varys. In *SIGCOMM*, 2014.

[72] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. A downsampled variant of imagenet as an alternative to the cifar datasets. In *arxiv.org/abs/1707.08819*, 2017.

[73] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. EMNIST: an extension of MNIST to handwritten letters. In *arxiv.org/abs/1702.05373*, 2017.

[74] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. *ACM TOCS*, 31(3):8, 2013.

[75] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI*, 2017.

[76] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *RecSys*, 2016.

[77] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: Latency-aware provisioning and scaling for prediction serving pipelines. In *SoCC*, 2020.

[78] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency online prediction serving system. In *NSDI*, 2017.

[79] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, and et al. Hydra: a federated resource manager for data-center scale analytics. In *NSDI*, 2019.

[80] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[81] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawasch Abedjan, Tilmann Rabl, and Volker Markl. Optimizing machine learning workloads in collaborative environments. In *SIGMOD*, 2020.

[82] J. Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*, 2019.

[83] Apple Differential Privacy Team. Learning with privacy at scale. In *Apple Machine Learning Journal*, 2017.

[84] Xuanyi Dong and Yi Yang. NAS-Bench-201: Extending the scope of reproducible neural architecture search. In *ICLR*, 2020.

[85] Xiaocong Du, Bhargav Bhushanam, Jiecao Yu, Dhruv Choudhary, Tianxiang Gao, Sherman Wong, Louis Feng, Jongsoo Park, Yu Cao, and Arun Kejariwal. Alternate model growth and pruning for efficient training of recommendation systems. In *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2021.

[86] S. Dutta, D. Wei, H. Yueksel, P. Y. Chen, S. Liu, and K. R. Varshney. Is there a trade-off between fairness and accuracy? a perspective using mismatched hypothesis testing. In *ICML*, 2020.

[87] Hubert Eichner, Tomer Koren, H. Brendan McMahan, Nathan Srebro, and Kunal Talwar. Semi-cyclic stochastic gradient descent. In *ICML*, 2019.

[88] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-n-run: a checkpointing system for training deep learning recommendation models. In *NSDI*, 2022.

[89] Dumitru Erhan, Pierre-Antoine Manzagol, Yoshua Bengio, Samy Bengio, and Pascal Vincent. The difficulty of training deep architectures and the effect of unsupervised pre-training. In David van Dyk and Max Welling, editors, *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, volume 5 of *Proceedings of Machine Learning Research*, pages 153–160, Hilton Clearwater Beach Resort, Clearwater Beach, Florida USA, 16–18 Apr 2009. PMLR.

[90] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander J. Smola. Autogluon-tabular: Robust and accurate automl for structured data. *CoRR*, abs/2003.06505, 2020.

[91] Ulfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In *CCS*, 2014.

[92] Scott Ettinger, Shuyang Cheng, Benjamin Caine, Chenxi Liu, Hang Zhao, Sabeek Pradhan, Yuning Chai, Benjamin Sapp, Charles Qi, Yin Zhou, Zoey Yang, Aurelien Chouard, Pei Sun, Jiquan Ngiam, Vijay Vasudevan, Alexander McCauley, Jonathon Shlens, and Dragomir Anguelov. Large scale interactive motion forecasting for autonomous driving : The waymo open motion dataset. *CoRR*, abs/2104.10133, 2021.

[93] Alireza Fallah, Aryan Mokhtari, and Asuman Ozdaglar. Personalized federated learning with theoretical guarantees: A model-agnostic meta-learning approach. In *NeurIPS*, 2020.

[94] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. Local model poisoning attacks to byzantine-robust federated learning. In *USENIX Security Symposium*, 2020.

[95] David F. Fouhey, Weicheng Kuo, Alexei A. Efros, and Jitendra Malik. From lifestyle vlogs to everyday interactions. In *CVPR*, 2018.

[96] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. A survey of graph edit distance. *Pattern Analysis and Applications*, 13, 113–129 (2010), 2010.

[97] Jonas Geiping, Hartmut Bauermeister, Hannah Dröge, and Michael Moeller. Inverting gradients - how easy is it to break privacy in federated learning? In *NeurIPS*, 2020.

[98] Robin C. Geyer, Tassilo Klein, and Moin Nabi. Differentially private federated learning: A client level perspective. In *NeuIPS*, 2017.

[99] Avishek Ghosh, Jichan Chung, Dong Yin, and Kannan Ramchandran. An efficient framework for clustered federated learning. In *NeurIPS*, 2020.

[100] A.A. Ginart, Maxim Naumov, Dheevatsa Mudigere, Jiyan Yang, and James Zou. Mixed dimension embeddings with application to memory-efficient recommendation systems. In *ISIT*, 2021.

[101] Carlos A. Gomez-Uribe and Neil Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Trans. Manage. Inf. Syst.*, 6(4), December 2016.

[102] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. 2016.

[103] Siddharth Gopal. Adaptive sampling for sgd by exploiting side information. In *ICML*, 2016.

[104] Siddharth Gopal. Adaptive sampling for SGD by exploiting side information. In *ICML*, 2016.

[105] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *OSDI*, 2016.

[106] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *OSDI*, 2016.

[107] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[108] 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 *NSDI*, 2019.

[109] Arpan Gujarati, Reza Karimi, Safya Alzayat, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *OSDI*, 2020.

[110] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. Cocktail: A multidimensional optimization for model serving in cloud. In *NSDI*, 2022.

[111] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, and et al. The architectural implications of facebook's dnn-based personalized recommendation. In *HPCA*, 2020.

[112] Vipul Gupta, Dhruv Choudhary, Peter Tang, Xiaohan Wei, Xing Wang, Yuzhen Huang, Arun Kejariwal, Kannan Ramchandran, and Michael W. Mahoney. Training recommender systems at scale: Communication-efficient model and data parallelism. KDD, 2021.

[113] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *SIGMOD*, 2019.

[114] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. NIPS'15, 2015.

[115] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In *NIPS*, 2015.

[116] Andrew Hard, Kanishka Rao, and et al. Federated learning for mobile keyboard prediction. In *arxiv.org/abs/1811.03604*, 2018.

[117] Florian Hartmann, Sunah Suh, Arkadiusz Komarzewski, Tim D. Smith, and Ilana Segall. Federated learning for ranking browser history suggestions. In *arxiv.org/abs/1911.11807*, 2019.

[118] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, and et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In *HPCA*, 2018.

[119] Chaoyang He, Songze Li, Jinhyun So, and Xiao Zeng. FedML: A research library and benchmark for federated machine learning. In *arxiv.org/abs/2007.13518*, 2020.

[120] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[121] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, and Joaquin Quiñonero Candela. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, ADKDD'14. Association for Computing Machinery, 2014.

[122] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *ECCV*, 2018.

[123] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A.D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[124] Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. In *NIPS Deep Learning and Representation Learning Workshop*, 2015.

[125] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. In *Journal of the American Statistical Association*, 1963.

[126] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM*, 2013.

[127] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *OSDI*, 2018.

[128] 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*, 2017.

[129] Kevin Hsieh, Amar Phanishayee, Onur Mutlu, and Phillip B. Gibbons. The Non-IID data quagmire of decentralized machine learning. In *ICML*, 2020.

[130] Harry Hsu, Hang Qi, and Matthew Brown. Federated visual classification with real-world data distribution. In *ECCV*, 2020.

[131] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *CVPR*, 2017.

[132] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *ICDE Workshop*, 2010.

[133] Yuzhen Huang, Yingjie Shi, Zheng Zhong, and et al. Yugong: Geo-Distributed data and job placement at scale. In *VLDB*, 2019.

[134] Chien-Chun Hung, Ganesh Ananthanarayanan, Leana Golubchik, Minlan Yu, and Mingyang Zhang. Wide-area analytics with multiple resources. In *EuroSys*, 2018.

[135] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[136] Anand Padmanabha Iyer, Li Erran Li, Mosharaf Chowdhury, and Ion Stoica. Mitigating the latency-accuracy trade-off in mobile data analytics systems. In *MobiCom*, 2018.

[137] Anand Padmanabha Iyer, Li Erran Li, and Ion Stoica. Celliq : Real-time cellular network analytics at scale. In *NSDI*, 2015.

[138] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. ASAP: Fast, approximate graph pattern mining at scale. In *OSDI*, 2018.

[139] 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 *ATC*, 2019.

[140] Zhihao Jia, Oded Padon, and et al. TASO: Optimizing deep learning computation with automatic generation of graph substitutions. In *SOSP*, 2019.

[141] Angela H. Jiang, Daniel L.-K. Wong, Christopher Canel, Lilia Tang, Ishan Misra, Michael Kaminsky, Michael A. Kozuch, Padmanabhan Pillai, David G. Andersen, and Gregory R. Ganger. Mainstream: Dynamic Stem-Sharing for Multi-Tenant video processing. In *ATC*, 2018.

[142] Junchen Jiang, Rajdeep Das, and et al. Via: Improving internet telephony call quality using predictive relay selection. In *SIGCOMM*, 2016.

[143] Junchen Jiang, Yuhao Zhou, Ganesh Ananthanarayanan, Yuanchao Shu, and Andrew A. Chien. Networked cameras are the new big data clusters. In *HotEdgeVideo*, 2019.

[144] 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 *OSDI*, 2020.

[145] Haifeng Jin, Qingquan Song, and Xia Hu. Auto-keras: An efficient neural architecture search system. In *KDD*, 2019.

[146] Tyler B. Johnson and Carlos Guestrin. Training deep models faster with robust, approximate importance sampling. In *NeurIPS*, 2018.

[147] Peter Kairouz, Brendan McMahan, Shuang Song, Om Thakkar, Abhradeep Thakurta, and Zheng Xu. Practical and private (deep) learning without sampling or shuffling. In *arxiv.org/abs/2103.00039*, 2021.

[148] Peter Kairouz, H. Brendan McMahan, and et al. Advances and open problems in federated learning. In *Foundations and Trends® in Machine Learning*, 2021.

[149] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. 2020.

[150] Sai Praneeth Karimireddy, Satyen Kale, Mehryar Mohri, Sashank J. Reddi, Sebastian U. Stich, and Ananda Theertha Suresh. SCAFFOLD: Stochastic controlled averaging for federated learning. In *ICML*, 2020.

[151] Sai Praneeth Karimireddy, Quentin Rebjock, Sebastian U Stich, and Martin Jaggi. Error feedback fixes signsgd and other gradient compression schemes. In *arXiv preprint arXiv:1901.09847*, 2019.

[152] Angelos Katharopoulos and Francois Fleuret. Biased importance sampling for deep neural network training. In *arxiv.org/abs/1706.00043*, 2017.

[153] Angelos Katharopoulos and Francois Fleuret. Not all samples are created equal: Deep learning with importance sampling. In *ICML*, 2018.

[154] Liu Ke, Udit Gupta, Carole-Jean Wu, Benjamin Youngjae Cho, Mark Hempstead, Brandon Reagen, Xuan Zhang, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, and Xiaodong Wang. Recnmp: Accelerating personalized recommendation with near-memory processing. In *ISCA*, 2020.

[155] Konstantinos Kloudas, Margarida Mamede, Nuno Preguica, and Rodrigo Rodrigues. Pixida: Optimizing data parallel jobs in wide-area data analytics. In *VLDB*, 2015.

[156] Philipp Koehn. Europarl: A Parallel Corpus for Statistical Machine Translation. In *Conference Proceedings: the tenth Machine Translation Summit*, 2005.

[157] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[158] Fan Lai, Mosharaf Chowdhury, and Harsha Madhyastha. To relay or not to relay for inter-cloud transfers? In *HotCloud*, 2018.

[159] Fan Lai, Yinwei Dai, Harsha V. Madhyastha, and Mosharaf Chowdhury. ModelKeeper: Accelerating dnn training via automated training warmup. In *NSDI*, 2023.

[160] Fan Lai, Yinwei Dai, Sanjay S. Singapuram, Jiachen Liu, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Fedscale: Benchmarking model and system performance of federated learning at scale. In *ICML*, 2022.

[161] Fan Lai, Jie You, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Sol: A federated execution engine for fast distributed computation over slow networks. In *NSDI*, 2020.

[162] Fan Lai, Wei Zhang, Rui Liu, William Tsai, Xiaohan Wei, Yuxi Hu, Sabin Devkota, Jianyu Huang, Jongsoo Park, Xing Liu, Zeliang Chen, Ellie Wen, Paul Rivera, Jie You, Chun cheng Jason Chen, and Mosharaf Chowdhury. AdaEmbed: Adaptive embedding for Large-Scale recommendation models. In *OSDI*, 2023.

[163] Fan Lai, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Oort: Efficient federated learning via guided participant selection. In *OSDI*, 2021.

[164] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. In *ICLR*, 2020.

[165] Chenning Li, Xiao Zeng, Mi Zhang, and Zhichao Cao. Pyramidfl: Fine-grained data and system heterogeneity-aware client selection for efficient federated learning. In *MobiCom*, 2022.

[166] Haoran Li, Dadi Guo, Wei Fan, Mingshi Xu, Jie Huang, Fanpu Meng, and Yangqiu Song. Multi-step jailbreaking privacy attacks on chatgpt. 2023.

[167] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018.

[168] Mu Li, David G. Andersen, and et al. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[169] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. Federated optimization in heterogeneous networks. In *MLSys*, 2020.

[170] Tian Li, Manzil Zaheer, Ahmad Beirami, and Virginia Smith. Fair resource allocation in federated learning. In *ICLR*, 2020.

[171] Wenqi Li, Fausto Milletari, and Daguang Xu. Privacy-preserving federated brain tumour segmentation. In *Machine Learning in Medical Imaging*, 2019.

[172] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, et al. Persia: a hybrid system scaling deep learning based recommenders up to 100 trillion parameters. *arXiv preprint arXiv:2111.05897*, 2021.

[173] Tao Lin, Sebastian U. Stich, Kumar Kshitij Patel, and Martin Jaggi. Don't use large mini-batches, use local SGD. In *ICLR*, 2020.

[174] Jiachen Liu, Fan Lai, Yinwei Dai, Aditya Akella, Harsha Madhyastha, and Mosharaf Chowdhury. Auxo: Heterogeneity-mitigating federated learning via scalable client clustering. *arXiv preprint:2210.16656*.

[175] Rui Liu, Tianyi Wu, and Barzan Mozafari. Adam with bandit sampling for deep learning. In *NeurIPS*, 2020.

[176] Siyi Liu, Chen Gao, Yihong Chen, Depeng Jin, and Yong Li. Learnable embedding sizes for recommender systems. In *ICLR*, 2021.

[177] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[178] Mingsheng Long, Han Zhu, Jianmin Wang, and Michael I. Jordan. Deep transfer learning with joint adaptation networks. In *ICML*, 2017.

[179] Jiahuan Luo, Xueyang Wu, Yun Luo, Anbu Huang, Yunfeng Huang, Yang Liu, and Qiang Yang. Real-world image datasets for federated learning. In *arxiv.org/abs/1910.11089*, 2019.

[180] Kshiteej Mahajan, Arjun Balasubramanian, and et al. Themis: Fair and efficient GPU cluster scheduling. In *NSDI*, 2020.

[181] Kshiteej Mahajan, Mosharaf Chowdhury, Aditya Akella, and Shuchi Chawla. Dynamic query re-planning using QOOP. In *OSDI*, 2018.

[182] Ankur Mallick, Kevin Hsieh, Behnaz Arzani, and Gauri Joshi. Matchmaker: Data drift mitigation in machine learning for large-scale systems. *MLSys*, 2022.

[183] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *SIGCOMM*, 2017.

[184] Omid Mashayekhi, Hang Qu, Chinmayee Shah, and Philip Levis. Execution templates: Caching control plane decisions for strong scaling of data analytics. In *ATC*, 2017.

[185] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. Mlperf training benchmark. In *MLSys*, 2020.

[186] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton van den Hengel. Image-based recommendations on styles and substitutes. In *SIGIR*, 2015.

[187] 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.

[188] William Mendenhall, Robert J Beaver, and Barbara M Beaver. *Introduction to probability and statistics*. Cengage Learning, 2012.

[189] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. 2016.

[190] Richard Meyes, Melanie Lu, Constantin Waubert de Puiseau, and Tobias Meisen. Ablation studies in artificial neural networks. 2019.

[191] Mehryar Mohri, Gary Sivek, and Ananda Theertha Suresh. Agnostic federated learning. In *ICML*, 2019.

[192] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *OSDI*, 2018.

[193] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, and et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *KDD*, 2021.

[194] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie (Amy) Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani,

Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models. ISCA, 2022.

[195] Matthew K Mukerjee, David Naylor, Junchen Jiang, Dongsu Han, Srinivasan Seshan, and Hui Zhang. Practical, real-time centralized control for cdn-based live video delivery. In *ACM SIGCOMM Computer Communication Review*, 2015.

[196] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.

[197] Deepak Narayanan, Aaron Harlap, and et al. Pipedream: Generalized pipeline parallelism for dnn training. In *SOSP*, 2019.

[198] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving large-scale granular resource allocation problems efficiently with pop. In *SOSP*, 2021.

[199] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, Krishnakumar Nair, Isabel Gao, Bor-Yiing Su, Jiyan Yang, and Mikhail Smelyanskiy. Deep learning training in facebook data centers: Design of scale-up and scale-out systems. *arXiv preprint arXiv:2003.09518*, 2020.

[200] Kirill Neklyudov, Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Structured bayesian pruning via log-normal multiplicative noise. In *NeurIPS*, 2017.

[201] James Newling and François Fleuret. K-medoids for k-means seeding. 2017.

[202] Behnam Neyshabur, Hanie Sedghi, and Chiyuan Zhang. What is being transferred in transfer learning? *NeurIPS*, 2020.

[203] M-E. Nilsback and A. Zisserman. Automated flower classification over a large number of classes. In *Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing*, Dec 2008.

[204] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning. In *ASPLOS*, 2020.

[205] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *SOSP*, 2017.

[206] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *NSDI*, 2015.

[207] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *SOSP*, 2013.

[208] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In *EuroSys*, 2018.

[209] Xingchao Peng, Zijun Huang, Yizhe Zhu, and Kate Saenko. Federated adversarial domain adaptation. In *ICLR*, 2020.

[210] Yanghua Peng, Yibo Zhu, and et al. A generic communication scheduler for distributed dnn training acceleration. In *SOSP*, 2019.

[211] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Victor Bahl, and Ion Stoica. Low latency geo-distributed data analytics. In *SIGCOMM*, 2015.

[212] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *OSDI*, 2021.

[213] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. 2022.

[214] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. Efficient queue management for cluster scheduling. In *EuroSys*, 2016.

[215] Alexander Ratner, Dan Alistarh, Gustavo Alonso, David G. Andersen, Peter Bailis, Sarah Bird, Nicholas Carlini, Bryan Catanzaro, Jennifer Chayes, Eric Chung, Bill Dally, Jeff Dean, Inderjit S. Dhillon, Alexandros Dimakis, Pradeep Dubey, Charles Elkan, Grigori Fursin, Gregory R. Ganger, Lise Getoor, Phillip B. Gibbons, Garth A. Gibson, Joseph E. Gonzalez, Justin Gottschlich, Song Han, Kim Hazelwood, Furong Huang, Martin Jaggi, Kevin Jamieson, Michael I. Jordan, Gauri Joshi, Rania Khalaf, Jason Knight, Jakub Konečný, Tim Kraska, Arun Kumar, Anastasios Kyrillidis, Aparna Lakshmiratan, Jing Li, Samuel Madden, H. Brendan McMahan, Erik Meijer, Ioannis Mitliagkas, Rajat Monga, Derek Murray, Kunle Olukotun, Dimitris Papailiopoulos, Gennady Pekhimenko, Theodoros Rekatsinas, Afshin Rostamizadeh, Christopher Ré, Christopher De Sa, Hanie Sedghi, Siddhartha Sen, Virginia Smith, Alex Smola, Dawn Song, Evan Sparks, Ion Stoica, Vivienne Sze, Madeleine Udell, Joaquin Vanschoren, Shivaram Venkataraman, Rashmi Vinayak, Markus Weimer, Andrew Gordon Wilson, Eric Xing, Matei Zaharia, Ce Zhang, and Ameet Talwalkar. MLSys: The new frontier of machine learning systems. 2019.

[216] Sashank Reddi, Zachary Charles, and et al. Adaptive federated optimization. In *arxiv.org/abs/2003.00295*, 2020.

[217] Siva Reddy, Danqi Chen, and Christopher D. Manning. Coqa: A conversational question answering challenge. *arXiv preprint arXiv:1808.07042*, 2019.

[218] Kaspar Riesen and Horst Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision Computing*, 27(7):950–959, 2009. 7th IAPR-TC15 Workshop on Graph-based Representations (GbR 2007).

[219] Jae Hun Ro, Ananda Theertha Suresh, and Ke Wu. Fedjax: Federated learning simulation with jax. 2021.

[220] Edo Roth, Daniel Noble, Brett Hemenway Falk, and Andreas Haeberlen. Honeycrisp: Large-scale differentially private aggregation without a trusted core. In *SOSP*, 2019.

[221] Kevin Roth, Yannic Kilcher, and Thomas Hofmann. The odds are odd: A statistical test for detecting adversarial examples. In *ICML*, 2019.

[222] Daniel Rothchild, Ashwinee Panda, Enayat Ullah, Nikita Ivkin, Ion Stoica, Vladimir Braverman, Joseph Gonzalez, and Raman Arora. Fetchsgd: Communication-efficient federated learning with sketching. In *ICML*, 2020.

[223] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curinom. Apache tez: A unifying framework for modeling and building data processing applications. In *SIGMOD*, 2015.

[224] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.

[225] Victor Sanh, Thomas Wolf, and Alexander M. Rush. Movement pruning: Adaptive sparsity by fine-tuning. In *NeurIPS*, 2020.

[226] J. Schler, M. Koppel, S. Argamon, and J. Pennebaker. Effects of age and gender on blogging. In *Proceedings of AAAI Spring Symposium on Computational Approaches for Analyzing Weblogs*, 2006.

[227] Naichen Shi, Fan Lai, Raed Al Kontar, and Mosharaf Chowdhury. Ensemble models in federated learning for improved generalization and uncertainty quantification. In *IEEE Transactions on Automation Science and Engineering*, 2023.

[228] Gunnar A. Sigurdsson, Gül Varol, Xiaolong Wang, Ali Farhadi, Ivan Laptev, and Abhinav Gupta. Hollywood in homes: Crowdsourcing data collection for activity understanding. In *ECCV*, 2016.

[229] Chijun Sima, Yao Fu, Man-Kit Sit, Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, and Luo Mai. Ekko: A Large-Scale deep learning recommender system with Low-Latency model update. In *OSDI*, 2022.

[230] Sanjay Sri Vallabh Singapuram, Fan Lai, Chuheng Hu, and Mosharaf Chowdhury. Swan: A neural engine for efficient dnn training on smartphone socs. 2022.

[231] Brent Smith and Greg Linden. Two decades of recommender systems at amazon.com. In *IEEE Internet Computing*, 2017.

[232] Ziteng Sun, Peter Kairouz, Ananda Theertha Suresh, and H Brendan McMahan. Can you really backdoor federated learning. In *arxiv.org/abs/1911.07963*, 2019.

[233] Ananda Theertha Suresh, Felix X. Yu, Sanjiv Kumar, and H. Brendan McMahan. Distributed mean estimation with limited communication. In *ICML*, 2017.

[234] AzureML Team. Azureml: Anatomy of a machine learning service. In *PAPIs*, 2015.

[235] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. 2023.

[236] Nilesh Tripuraneni, Michael I. Jordan, and Chi Jin. On the theory of transfer learning: The importance of task diversity. In *NeurIPS*, 2020.

[237] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976.

[238] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V. Madhyastha. Near-optimal latency versus cost tradeoffs in geo-distributed storage. In *NSDI*, 2020.

[239] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. Modeldb: A system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, 2016.

[240] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, 2017.

[241] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J Franklin, and Ion Stoica. The power of choice in data-aware cluster scheduling. In *OSDI*, 2014.

[242] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, and Michael J. Franklin. Drizzle: Fast and adaptable stream processing at scale. In *SOSP*, 2017.

[243] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. Clarinet: WAN-aware optimization for analytics queries. In *OSDI*, 2016.

[244] Ashish Vulimiri, Carlo Curino, B Godfrey, J Padhye, and G Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *NSDI*, 2015.

[245] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *OSDI*, 2021.

[246] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *OSDI*, 2021.

[247] Hongyi Wang, Kartik Sreenivasan, Shashank Rajput, Harit Vishwakarma, Saurabh Agarwal, Jy yong Sohn, Kangwook Lee, and Dimitris Papailiopoulos. Attack of the tails: Yes, you really can backdoor federated learning. In *NeurIPS*, 2020.

[248] Jianyu Wang and Gauri Joshi. Adaptive communication strategies to achieve the best error-runtime trade-off in local-update SGD. In *MLSys*, 2019.

[249] Kangkang Wang, Rajiv Mathews, Chloe Kiddon, Hubert Eichner, Francoise Beaufays, and Daniel Ramage. Federated evaluation of on-device personalization. In *arxiv.org/abs/1910.10252*, 2019.

[250] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *PPoPP*, 2018.

[251] Yiding Wang, Decang Sun, Kai Chen, Fan Lai, and Mosharaf Chowdhury. Egeria: Efficient dnn training with knowledge-guided layer freezing. In *EuroSys*, 2023.

[252] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition. In *arxiv.org/abs/1804.03209*, 2018.

[253] Abdul Wasay, Brian Hentschel, Yuze Liao, Sanyuan Chen, and Stratos Idreos. Mothernets: Rapid deep ensemble learning. In *MLSys*, 2020.

[254] Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. Network morphism. In *ICML*, 2016.

[255] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *NSDI*, 2022.

[256] Tobias Weyand, Andre Araujo, Bingyi Cao, and Jack Sim. Google landmarks dataset v2 a large-scale benchmark for instance-level recognition and retrieval. In *arxiv.org/abs/2004.01804*, 2020.

[257] Carole-Jean Wu, Ramya Raghavendra, Udit Gupta, Bilge Acun, Newsha Ardalani, Kiwan Maeng, Gloria Chang, Fiona Aga Behram, James Huang, Charles Bai, Michael Gschwind, Anurag Gupta, Myle Ott, Anastasia Melnikov, Salvatore Candido, David Brooks, Geeta

Chauhan, Benjamin Lee, Hsien-Hsin S. Lee, Bugra Akyildiz, Maximilian Balandat, Joe Spisak, Ravi Jain, Mike Rabbat, and Kim Hazelwood. Sustainable ai: Environmental implications, challenges and opportunities. MLSys, 2022.

[258] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *SOSP*, 2013.

[259] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *OSDI*, 2018.

[260] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *OSDI*, 2020.

[261] Minhui Xie, Youyou Lu, Jiazhen Lin, Qing Wang, Jian Gao, Kai Ren, and Jiwu Shu. Fleche: An efficient gpu embedding cache for personalized recommendations. EuroSys, 2022.

[262] Minhui Xie, Kai Ren, Youyou Lu, Guangxu Yang, Qingxing Xu, Bihai Wu, Jiazhen Lin, Hongbo Ao, Wanhong Xu, and Jiwu Shu. Kraken: Memory-efficient continual learning for large-scale real-time recommendations. In *SC*, 2020.

[263] Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaozhu Lin, Yunxin Liu, and Xuanzhe Liu. A first look at deep learning apps on smartphones. In *WWW*, 2019.

[264] Francis Y. Yan, Hudson Ayers, and et al. Learning in situ: a randomized experiment in video streaming. In *NSDI*, 2020.

[265] Li Yan, Choudhary Dhruv, Wei Xiaohan, Yuan Baichuan, Bhushanam Bhargav, Zhao Tuo, and Lan Guanghui. Frequency-aware sgd for efficient embedding learning with provable benefits. In *ICLR*, 2022.

[266] Ying Yan, Liang Jeff Chen, and Zheng Zhang. Error-bounded sampling for analytics on big sparse data. In *VLDB Endowment*, 2014.

[267] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Re, Christopher Aberger, and Christopher De Sa. Pipemare: Asynchronous pipeline parallel dnn training. In *MLSys*, 2021.

[268] Chengxu Yang, Qipeng Wang, and et al. Characterizing impacts of heterogeneity in federated learning upon large-scale smartphone data. In *WWW*, 2021.

[269] Timothy Yang, Galen Andrew, Hubert Eichner, Haicheng Sun, Wei Li, Nicholas Kong, Daniel Ramage, and Françoise Beaufays. Applied federated learning: Improving Google keyboard query suggestions. In *arxiv.org/abs/1812.02903*, 2018.

[270] Chunxing Yin, Bilge Acun, Carole-Jean Wu, and Xing Liu. TT-Rec: Tensor train compression for deep learning recommendation models. In *MLSys*, 2021.

[271] Zi Yin and Yuanyuan Shen. On the dimensionality of word embedding. NIPS'18, 2018.

[272] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *ArXiv*, abs/1411.1792, 2014.

[273] Jie You, Sheng Yang, Fan Lai, Mosharaf Chowdhury, and Samir Khuller. System H: A framework for optimizing hybrid multi-cloud analytics. *arXiv preprint:1901.08644*.

[274] Felix X. Yu, Ankit Singh Rawat, Aditya Krishna Menon, and Sanjiv Kumar. Federated learning with only positive labels. In *ICML*, 2020.

[275] Hao Yu, Sen Yang, and Shenghuo Zhu. Parallel restarted sgd with faster convergence and less communication: Demystifying why model averaging works for deep learning. In *AAAI*, 2019.

[276] Peifeng Yu and Mosharaf Chowdhury. Fine-grained gpu sharing primitives for deep learning applications. In *MLSys*, 2020.

[277] Peifeng Yu, Jiachen Liu, and Mosharaf Chowdhury. Fluid: Resource-aware hyperparameter tuning engine. In *MLSys*, 2021.

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

[279] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant stream computation at scale. In *SOSP*, 2013.

[280] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[281] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41(4):39–45, 2018.

[282] Heiga Zen, Viet Dang, Rob Clark, Yu Zhang, Ron J. Weiss, Ye Jia, Zhifeng Chen, and Yonghui Wu. Libritts: A corpus derived from librispeech for text-to-speech. *arXiv preprint arXiv:1904.02882*, 2019.

[283] Chaoliang Zeng, Layong Luo, Qingsong Ning, Yaodong Han, Yuhang Jiang, Ding Tang, Zilong Wang, Kai Chen, and Chuanxiong Guo. FAERY: An FPGA-accelerated embedding-based retrieval system. In *OSDI*, 2022.

[284] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A. Lee. AWStream: Adaptive wide-area streaming analytics. In *SIGCOMM*, 2018.

[285] Ke Zhang, Miao Sun, Tony X. Han, Xingfang Yuan, Liru Guo, and Tao Liu. Residual networks of residual networks: Multilevel residual networks. *IEEE Transactions on Circuits and Systems for Video Technology*, 28(6):1303–1314, Jun 2018.

[286] Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, and Lidong Zhou. Retiarii: A deep learning exploratory-training framework. In *OSDI*, 2020.

[287] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *CVPR*, 2018.

[288] Han Zhao, Remi Tachet des Combes, Kun Zhang, and Geoffrey J. Gordon. On learning invariant representation for domain adaptation. ICML'19, 2019.

[289] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. *MLSys*, 2020.

[290] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. AIBox: CTR prediction model training on a single node. CIKM, 2019.

[291] Haizhong Zheng, Rui Liu, Fan Lai, and Atul Prakash. Coverage-centric coreset selection for high pruning rates. In *ICLR*, 2023.

[292] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *OSDI*, 2022.

[293] Wenting Zheng, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *IEEE S&P*, 2019.

[294] Guorui Zhou, Chengru Song, Xiaoqiang Zhu, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *KDD*, 2018.