ABSTRACT

Title of Dissertation: EFFICIENT MODELS AND LEARNING STRATEGIES FOR RESOURCE-CONSTRAINED SYSTEMS

Tahseen Rabbani
Doctor of Philosophy, 2024

Dissertation Directed by: Assistant Professor Furong Huang
Department of Computer Science

The last decade has seen sharp improvements in the performance of machine learning (ML) models, but at the cost of vastly increased complexity and size of their underlying architectures. Advances in high-performance computing have enabled researchers to train and deploy models composed of hundreds of billions of parameters. However, harvesting the full utility of large models on smaller clients, such as Internet of Things (IoT) devices, without resorting to external hosting will require a significant reduction of parameters and faster, cheaper inference. In addition to augmenting IoT, efficient models and learning paradigms can reduce energy consumption, encourage technological equity, and are well-suited for deployment in real-world applications which require fast response in low-resource settings.

To address these challenges, we introduce multiple, novel strategies for (1) reducing the scale of deep neural networks and (2) faster learning. For the size problem (1), we leverage tools such as tensorization, randomized projections, and locality-sensitive hashing to train on reduced

representations of large models without sacrificing performance. For learning efficiency (2), we develop algorithms for cheaper forward passes, accelerated PCA, and asynchronous gradient descent. Several of these methods are tailored for federated learning (FL), a private, distributed learning paradigm where data is decentralized among resource-constrained edge clients. We are exclusively concerned with improving efficiency *during* training – our techniques do not process pre-trained models or require a device to train over an architecture in its full entirety.

EFFICIENT MODELS AND LEARNING STRATEGIES FOR RESOURCE-CONSTRAINED SYSTEMS

by

Tahseen Rabbani

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2024

Advisory Committee:
Assistant Professor Furong Huang, Chair/Advisor
Professor Thomas Goldstein
Assistant Professor Nirupam Roy
Associate Professor Haizhao Yang
Assistant Professor Tianyi Zhou

© Copyright by
Tahseen Rabbani
2024

Acknowledgments

This dissertation is more than just mine. It was made by possible by a network of individuals who celebrated all my successes and pushed me through all my failures.

First and most importantly, I would like to thank my advisor, Dr. Furong Huang. She shepherded my entry into machine learning and has been a strong advocate of all my work. Our collaboration has been successful thanks to her egalitarian approach to advising: she is not only my mentor, but also my colleague and friend. Her wide breadth of knowledge and strong sense of curiosity have created a group dynamic that encourages a judgment-free submission of wildly imaginative ideas. She has served as my navigator in this young, fast-growing field and I owe much of my professional growth to her.

I would like to thank several departmental representatives for guiding me through my doctoral career. Tom Hurst was always a thoughtful source of guidance on any logistical hurdle. Jodie Gray always made time on the phone or in-person to help me work through election benefits and financial questions. Dr. Ramani Duraiswaimi graciously took a chance on a kid from across the street wanting to transfer into the CS department.

It is fortunate that I rarely had to leave UMD to find incredibly talented colleagues and researchers to work on every single project with. My academic older siblings, Dr. Jiahao Su and Dr. Yanchao Sun served as role models for every pivotal stage of my doctoral career. I am thankful for my frequent collaborator on all things related to federated learning, Marco Bornstein.

To my brilliant undergraduate co-authors, Arjun Rajkumar (Mr. Page 2) and Kyle Sang: thank you for helping me grow as a mentor and for pushing through arduous publication deadlines.

I owe endless gratitude to my friends, family, and signifcant other. To my partner, Meaghan, your companionship keeps me grounded and gave me the emotional strength to approach the latter half of my doctoral work with renewed optimism and joy. I appreciate Dr. James Davis for serving as my first research mentor and endless source of career advice. I thank my childhood best friends, Ronald, Jack, and Mark for helping me find sanity and humor through every stage of this long, twisting journey. To my doctoral buddies, Nelson and Josue, who entered graduate school with me – where did time go? Thank you, Mom, Dad, and Tahrin for your unconditional love and support: my intense love of learning began at home with you three.

I would like to acknowledge financial support from the Laboratory of Physical Sciences (LPS) for nearly every project discussed herein and the National Science Foundation (NSFDGE-1632976) for support on much of my initial federated learning research.

I sincerely apologize if I have inadvertently left out anyone who has contributed to the completion of this thesis. You are a significant component of this journey.

Table of Contents

Acknowledgements ii

Table of Contents iv

List of Tables viii

List of Figures xii

List of Abbreviations xix

Chapter 1: Introduction 1
- 1.1 Background & Motivation . . . 1
- 1.2 Organization of Dissertation . . . 4

Chapter 2: Practical and Fast Eigenvector Discovery 6
- 2.1 Introduction . . . 6
- 2.2 Related Works . . . 9
- 2.3 Accelerated Momentum-Based Power Methods via Inexact Deflation . . . 12
  - 2.3.1 Delayed Momentum Power Method (DMPower) . . . 16
  - 2.3.2 Delayed Momentum Streaming Power Method (DMStream) . . . 18
- 2.4 Convergence Analysis . . . 20
  - 2.4.1 Delayed Momentum Power Method (DMPower) . . . 20
  - 2.4.2 Delayed Momentum Streaming Power Method (DMStream) . . . 21
  - 2.4.3 Precision of Inexact Deflation . . . 23
- 2.5 Experiments . . . 26
  - 2.5.1 DMPower Experiments . . . 26
  - 2.5.2 DMStream Experiments . . . 28
  - 2.5.3 Application: Spectral Clustering . . . 29
- 2.6 Conclusion . . . 31
- 2.7 Supplemental Materials . . . 32
  - 2.7.1 Preliminaries and Facts . . . 32
  - 2.7.2 Noisy Deflation . . . 34
  - 2.7.3 Proof of Theorem 4 . . . 38
  - 2.7.4 Proof of Theorem 5 . . . 41
  - 2.7.5 Experimental Data . . . 50
  - 2.7.6 Precision Bounds for Momentum . . . 54
  - 2.7.7 Data Matrix Generation for Non-Streaming Experiments . . . 55

2.7.8 Spectral Clustering & Experimental Data . . . 56

Chapter 3: Representation and FLOPs-Minimal Evaluation of Multilinear Operations 58
3.1 Introduction . . . 58
3.2 Related works . . . 60
3.3 Tensor Operations and einsum . . . 62
3.3.1 Representations of Multilinear Operations . . . 62
3.3.2 From einsum to conv_einsum . . . 64
3.3.3 Compact Neural Networks via conv_einsum . . . 65
3.4 Algorithms . . . 67
3.4.1 Atomic Operations . . . 67
3.4.2 Optimal Sequencer . . . 70
3.4.3 Checkpointing . . . 73
3.5 Experiments . . . 74
3.5.1 Runtime and FLOPs results . . . 76
3.5.2 Memory Results . . . 79
3.6 Conclusions and Discussions . . . 80
3.7 Supplemental Material . . . 81
3.7.1 Examples of primitive operations . . . 81
3.7.2 Tensorial Neural Networks . . . 85
3.7.3 Optimal Sequencer . . . 90
3.7.4 Proof of Theorem 25 and 26 . . . 93
3.7.5 Supplementary Experimental Data . . . 97

Chapter 4: Federated Compression of Large Networks via Sketching 99
4.1 Introduction . . . 99
4.2 Preliminaries and Problem Setup . . . 101
4.2.1 Federated Learning Setup . . . 101
4.2.2 Network Architectures . . . 102
4.2.3 Count Sketch . . . 103
4.3 Comfetch . . . 104
4.3.1 Model Transmission and Download . . . 104
4.3.2 Client Update . . . 106
4.3.3 Backwards pass and uplink . . . 107
4.3.4 Model Update . . . 108
4.4 Related Work . . . 109
4.5 Convergence Guarantee . . . 110
4.6 Experiments . . . 112
4.6.1 CIFAR10/100 Experiments . . . 113
4.6.2 FedAvg Pruning . . . 115
4.7 Conclusion . . . 116
4.8 Supplemental Material . . . 116
4.8.1 Further Related Work . . . 116
4.8.2 Sketches . . . 117
4.8.3 Proof of Theorem 27 . . . 121

4.8.4 Multi-Sketch Comfetch . . . 125
4.8.5 Model Transmission and Download . . . 126
4.8.6 Model Update . . . 129
4.8.7 Additional Language Task Data . . . 130
4.8.8 Prediction Error Bound . . . 131
4.8.9 Multi-Sketch Ablation & ResNet-9 Experiments . . . 134

Chapter 5: Decentralized Federated Learning via Wait-Free Model Communication 136
5.1 Introduction . . . 136
5.2 Related Works . . . 138
5.3 Problem Formulation . . . 140
5.4 Shared Wait-Free Transmission (SWIFT) Federated Learning . . . 142
5.5 SWIFT Theoretical Analysis . . . 149
5.6 Experiments . . . 152
5.6.1 Baseline Comparison . . . 152
5.6.2 Varying Heterogeneities . . . 154
5.6.3 Varying Numbers of Clients & Network Topologies . . . 157
5.7 Conclusion . . . 159
5.8 Supplemental Material . . . 160
5.8.1 Additional Experimental Details and Setup . . . 160
5.8.2 Additional Properties and Algorithm Details . . . 162
5.8.3 Review of Existing Inter-Client Communication in Decentralized FL . . . 164
5.8.4 Proof of the Main Theorem . . . 166
5.8.5 Additional Lemmas . . . 179

Chapter 6: Large-Scale Distributed Learning via Private On-Device LSH 185
6.1 Introduction . . . 185
6.2 Related work . . . 187
6.3 Preliminaries . . . 189
6.4 PGHash . . . 192
6.4.1 PGHash theory . . . 193
6.4.2 PGHash algorithm . . . 197
6.5 Experiments . . . 199
6.6 Conclusion . . . 204
6.7 Supplemental Material . . . 206
6.7.1 Experiment details . . . 206
6.7.2 PGHash: angle versus Hamming distance . . . 210
6.7.3 Additional proofs . . . 212
6.7.4 Proof of Theorem 50 . . . 212
6.7.5 Additional theory . . . 215

Chapter 7: Benchmarking the Robustness of Image Watermarks 217
7.1 Introduction . . . 217
7.2 Image Watermarks . . . 220
7.3 Standardized Evaluation through WAVES . . . 221

7.3.1 Standardized Evaluation Workflow and Metrics . . . 221
7.3.2 Stress-testing Watermarks . . . 225
7.4 Benchmarking Results and Analysis . . . 232
7.4.1 Benchmarking Watermark Robustness . . . 232
7.4.2 Benchmarking Attacks . . . 234
7.4.3 Benchmarking Results for User Identification . . . 235
7.4.4 Summary of Takeaway Messages . . . 236
7.5 Conclusion . . . 237
7.6 Supplementary Material . . . 238
7.6.1 A Mini Survey of Image Watermarks . . . 238
7.6.2 Formalism of Watermark Detection and Identification . . . 242
7.6.3 Details on Performance Metrics . . . 245
7.6.4 Design Choices of WAVES . . . 249
7.6.5 Evaluation Details . . . 251
7.6.6 Details of Benchmarking Watermarks . . . 256
7.6.7 Details of Benchmarking Attacks . . . 258
7.7 Details of Attacks . . . 258
7.7.1 Distortion Attacks . . . 258
7.7.2 Regeneration Attacks . . . 260
7.7.3 Adversarial Attacks . . . 264
7.8 Additional Results . . . 265
7.8.1 More Results for Identification . . . 265
7.8.2 More Analyses on Surrogate Detector Attacks . . . 265
7.8.3 Visualization of Attacks . . . 269
7.8.4 Full Results on DiffusionDB, MS-COCO and DALL·E3 . . . 278
7.9 Limitations . . . 278

Chapter 8: Conclusion 281

Bibliography 283

List of Tables

2.1 Iterations required for Power+M to converge at various sub-optimal and optimal β assignments. $\beta = \lambda_2^2/4 = 0.2025$ is the optimal momentum coefficient at this setting, where $\lambda_2 = 0.9$. 51

2.2 Absolute difference between final approximation of λ_2 and true value of λ_2. . . . 51

2.3 Iterations for Vanilla PM, Power+M, and DMPower at `spec`=[1, 0.99, 0.98,. . . , 0.98]. 51

2.4 Percentage of iterations in the *momentum phase* for `spec`=[1, 0.99, 0.98,. . . , 0.98] with $A \in \mathbb{R}^{100\times 100}$. DMPower across a variety of ρ settings spends the vast majority of its time in the less computationally-intensive momentum phase. . 52

2.5 Recorded wall-time convergence speeds (in nanoseconds) for vanilla power method, Power+M with optimal β assignment, and various settings of DMPower, and the Lanczos algorithm. Performed 1000 calls using random PSD of various sizes with fixed spectrum $\lambda_1 = 1, \lambda_2 = 0.99, \lambda_3 = 0.98$, and remaining eigenvalues set to 0.98. In several instances DMPower exhibits faster wall-time speeds than Power+M with optimal β assignment, consistently outperforms Lanczos, and markedly accelerates the vanilla power method at all error thresholds (ϵ) tighter than 0.1. 52

2.6 Averaged $\log$ error, batch size = 500, with performance measured by $\log_{10}\big(1 - \frac{||X^\top q_K||}{||X^\top v_1||}\big)$. DMStream registers much better accuracy than Oja's algorithm and emulates the performance of optimal Mini-Batch Power+M. We notice that accuracy does not improve as we increase the number of epochs, which is commonly observed for streaming algorithm running with small batch sizes. We demonstrate in Figure 2.4 that increasing batch size results in improved accuracy. 53

2.7 Proportion of data points correctly classified by spectral clustering combined with k-means. As the error-threshold (and therefore, accuracy) of the eigenvector output is tightened, accuracy improves, eventually achieving perfect classification. See Figure 2.5 for a visual depiction of data separation using DMPower ($\rho = \sqrt[3]{\epsilon}$). We further note that although the vanilla power method achieves perfect classification at lower error-thresholds, it is at the cost of significantly more iterations as indicated in Table 2.8. 57

2.8 Iteration complexity required to recover principal components. DMPower requires significantly fewer iterations for eigenvector recovery when compared to the vanilla power method, and closely mimics the performance of Power+M with $\beta = \lambda_2^2/4$. 57

3.1 **Run-time (minutes per epoch) comparison between conv_einsum and PyTorch for ImageNet Classiciation w/ checkpointing.** We use a RCP ($M = 3$) ResNet-34 architecture and batch size of 256. Left-to-right evaluation runs into memory overflow without checkpointing. conv_einsum significantly improves training time. Runtimes are averaged over 3 runs. 78
3.2 **FLOPs per CP convolutional layer in ResNet-34.** We calculate the FLOPs incurred by forward passes through CP convolutional layers (CR$= 100\%$), using a batch size of 128. Layer nomenclature and associated kernel structure correspond with architecture description in [1]. 79
3.3 **Maximum batch size for a speech and video task.** The maximal batch size allowed for data under varying compression rates and using different libraries on **(1)** an ASR on LibriSpeech and **(2)** a VC task for spatial (S) and temporal (T) streams of a two-stream network on UCF-101. conv_einsum permits larger batch sizes. 80
3.4 **Primitive tensor operations**. $\mathcal{T}^{(0)} \in \mathbb{R}^{I_0 \times \cdots \times I_{m-1}}, \mathcal{T}^{(1)} \in \mathbb{R}^{J_0 \times \cdots \times J_{n-1}}$ are the input tensors and $\mathcal{T}$ is the output tensor. Note that both mode-(I_k, J_l) tensor contraction, mode-(I_k, J_l) tensor batch product are legal only if $I_k = J_l$. The outer product can be performed with any two tensors. Red highlighted modes are owned by $\mathcal{T}^{(0)}$ while blue highlighted modes belong to $\mathcal{T}^{(1)}$. 83
3.5 **Run-time (seconds per epoch) comparison between conv_einsum and PyTorch implementation of TNNs using different tensor decomposition forms in the image classification task on the CIFAR-10 dataset**. The base architecture used is ResNet-34. We observe that conv_einsum outperforms PyTorch under different tensor decompositions. Here, RXX denotes the reshaped XX decomposition, with reshaping factor M=3. 97
3.6 **TNN performance under various model scales for image classification under low resources (4-core CPU).** CIFAR10 classification using RCP ($M = 3$) ResNet-34 and a batch size of 128. Result is seconds per epoch. 97
3.7 **TNN performance under various model scales for diverse machine learning tasks.** Automatic Speech Recognition (ASR) on LibriSpeech is measured by Word Error Rate (WER) (the lower, the better). Image classification (IC) on CIFAR10 (results from [2]) is measured by top-1 precision. Video classification (VC) on UCF-101 is measured by top-1 accuracy. Averaged over 3 runs. 98

4.1 **IID CIFAR-10 Top accuracy.** Average top test accuracy over three runs for 10 client `Comfetch` classifying CIFAR-10 [3] using ResNet-18. We analyze how `Comfetch` performs over a range of compression rates as well as IID and non-IID ($\alpha = 1$) dataset splits. 113
4.2 **Non-IID CIFAR-100 Top Accuracy.** Average test accuracy over three random runs for 10 clients classifying CIFAR-100 [3] using ResNet18. We analyze how `Comfetch` performs over a range of compression rates as well as IID and non-IID ($\alpha = 1$) dataset splits. 114
4.3 Model accuracies under different memory footprints in clients, for predicting part of speech taggings for MNLI [4] sentences using an LSTM. We exclude embeddings from parameter counts as they serve as inputs and can be pretrained. 131

4.4 Test accuracy under different memory footprints in clients for the CIFAR-10 [3] image classification task. `Comfetch` maintains a large and powerful global model under communication compression and client memory compression. The no sketch model is compressed via random pruning, which performs much worse than our sketch-compressed models. 135

5.1 Rate and complexity comparisons for decentralized FL algorithms. 138
5.2 Baseline average epoch and communication times on CIFAR-10 for 16 client ring. 154
5.3 Average epoch and communication times for non-IID setting on CIFAR-10. . . . 156
5.4 Average epoch and communication times on CIFAR-10 for 16 client ring with slowdown. 156
5.5 Average epoch and communication times on CIFAR-10 with varying clients in ring topology. 158
5.6 Average epoch and communication times on CIFAR-10 for varying network topologies. 159
5.7 Hyperparameters for all experiments. 161

6.1 Hyper-parameters for Federated Experiments (PGHash and Federated SLIDE). . 206
6.2 **Hyper-parameters for Compression Experiments (PGHash).** 207
6.3 **Average LSH time for PGHash over a range of tables.** We compute the average μ time (and standard deviation σ) it takes for PGHash to perform *vanilla sampling* (exact matches) between the hash codes of sample x and each weight w_i in the final dense layer. Times are sampled for PGHash on Delicious-200k for batch size $M = 128$, $k = 9$, and $c = 8$ for one device. 207
6.4 **PGHash and SLIDE performance on Amazon-670K using SimHash.** Accuracy across the first 5,000 iterations for a single device. Batch size $M = 1024$, $k = 8$, and $c = 8$. 209

7.1 **Comparison of robustness evaluations with existing works.** For *categories of attacks*, D denotes distortions, R denotes image regeneration attacks, and A denotes adversarial attacks. *Reference image source* shows whether to use non-watermarked images in the evaluation and what is the source of those images. For the *number of quality metrics*, we use 8 in the paper but test 10 in total. *Joint test* means whether the performance and quality are jointly tested under a range of attack strengths. Our benchmark is the most comprehensive one, with a large scale of attacks, data, metrics, and more realistic evaluation setups. 219
7.2 A taxonomy of all the attacks in our stress-testing set. Novel attacks proposed by WAVES are marked with *. 223

7.3 **Comparison of attacks across three watermarking methods under the detection setup.** Q denotes the normalized quality degradation, and P denotes the performance as derived from Figure 7.7. Q@0.95P measures quality degradation at a 0.95 performance threshold where "inf" denotes cases where all tested attack strengths yield performance above 0.95, and "-inf" where all are below. A similar notation applies to Q@0.7P. Avg P and Avg Q are the average performance and quality over all the attack strengths. The lower the performance and the smaller the quality degradation, the stronger the attack is. For each watermarking method, we rank attacks by Q@0.95P, Q@0.7P, Avg P, Avg Q, in that order, with lower values (↓) indicating stronger attacks. The top 5 attacks of each watermarking method are highlighted in red. 279
7.4 A list of alternative watermarking algorithms not tested by WAVES in this work. . 279
7.5 **Comparison of attacks across three watermarking methods under the identification setup (one million users).** Q denotes the normalized quality degradation and P denotes the performance as derived from aggregated 2D plots. Q@0.7P measures quality degradation at a 0.7 performance threshold where "inf" denotes cases where all tested attack strengths yield performance above 0.7, and "-inf" where all are below. Q@0.4P is defined analogously. Avg P and Avg Q are the average performance and quality over all the attack strengths. The lower the performance and the smaller the quality degradation, the stronger the attack. For each watermarking method, we rank attacks by Q@0.7P, Q@0.4P, Avg P, Avg Q, in that order, with lower values (↓) indicating stronger attacks. The top 5 attack of each watermarking method are highlighted in red. 280

List of Figures

1.1 The parameter counts of notable machine learning models grow with time [5] . . 2

2.1 **Sub-optimal β Selection for Power+M**. We measure convergence speeds at various momentum assignments. The X-axis is the error threshold ϵ between approximates q_k, q_{k-1} of v_1 needed for Power+M to terminate, i.e., we run until $||q_k - q_{k-1}|| < \epsilon$. The Y-axis is the total number of iterations k needed to meet this condition according to the update equation (2.4). Averaged over 1000 runs, each time run on a random PSD $A \in \mathbb{R}^{10\times 10}$ with spectrum $\lambda_1 = 1, \lambda_2 = 0.9, \lambda_3 = 0.8$, and remaining eigenvalues set to 0.8. We observe that increased deviation from the optimal β assignment results in worsened and eventually divergent performance. 14

2.2 **Accuracy vs Simultaneous Power**. We measure the Rayleigh quotient accuracy in determining λ_2 between the Simultaneous Power Iteration (SimPM) and DMPower. The X-axis is the error threshold ϵ of the entire DMPower run. We report the true accuracy of $\widehat{\lambda}_2$ output by the pre-momentum phase, which is reflected in the Y-axis. Averaged over 1000 runs, each time run on a random PSD $A \in \mathbb{R}^{10\times 10}$. We varied ρ according to ϵ for convenience, but ρ does not depend on ϵ in implementation. We observe that DMPower is overall the most accurate, while SimPM is inaccurate at low thresholds. 15

2.3 **Iteration Complexity Comparisons**. We compare DMPower (with different ρ settings) against the vanilla power method, Power+M with optimal β, and the Lanczos algorithm. The X-axis corresponds to ϵ, and the Y-axis measures performance by iteration count. Our algorithms demonstrate consistently favorable performance against the vanilla power method baseline and match optimal convergence speeds established by Power+M at medium and tight eigengap settings. *Note 1*: For the Lanczos algorithm, we start by taking 100 tri-diagonalization iterations, which is the recommended number for numerical stability. *Note 2:* We vary ρ according to ϵ for convenience, but they are independent precision thresholds. As stated in Proposition 6, ρ is dependent on $\Delta_{1,2}$. 27

2.4 **Performance by batch size**. In this experiment we measure the performance of DMStream versus Oja's algorithm (with various step sizes) and Mini-Batch Power+M (optimal $\beta = \lambda_2^2/4$). We measure performance by the log error $\log_{10}(1 - \frac{||X^\top q_K||}{||X^\top v_1||})$. DMStream exhibits a consistent increase in accuracy as batch-size is increased and mimics the performance of optimal Mini-Batch Power+M. 28

2.5 **Performance of DMPM on Spectral Clustering.** We depict the performance of DMPM ($\rho = \sqrt[3]{\epsilon}$) on dividing a collection of data points into their natural, geometrically-partitioned clusters. The first series (a-e) is the concentric circles dataset while the second series (f-j) is the half-moons dataset. In both series, the first image depicts the unlabeled arrangement, the second depicts a naïve application of k-means without spectral clustering, while the last three images depict the performance of k-means assisted by spectral clustering over progressively tighter ϵ error thresholds (thus, requiring more accurate affinity matrix eigenvector approximations). As ϵ grows smaller, the data separation improves, eventually achieving perfect classification by $\epsilon = 10^{-8}$. 30

3.1 **conv_einsum sample code.** The figure depicts the generation and analysis of a set of tensors coalesced into one tensor sequence (Python). 69

3.2 **Optimal sequencer example**. conv_einsum deploys the optimal sequencer to analyze the path tree of an abstract tensor sequence $\mathcal{A} \circ_1 \mathcal{B} \circ_2 \mathcal{C} \circ_3 \mathcal{D} \circ_4 \mathcal{E}$, where $\circ_i$ for $1 \leq i \leq 4$ is any collection of multi-linear operations, including convolutions, batch products, contractions, and outer products. The green path indicates the optimal path and the orange path indicates a path which satisfies a user-specified cost cap c at each node. 70

3.3 **Run-time comparison between conv_einsum and naive evaluation for image classification (IC) and automatic speech recognition (ASR) (CP-TNN) tasks**. A RCP-TNN ($M = 3$) and CIFAR-10 is used for IC while a CP-TNN and LibriSpeech dataset isused for ASR. 'ckpt' denotes checkpointing. Runtimes are averaged over 3 runs. 76

3.4 **Run-time comparison between conv_einsum and naive left-to-right PyTorch implementations for a video classification machine learning task** An RCP-TNN ($M = 3$) is trained on the UCF-101 dataset. All tests were run using the maximum allowable batch size. Naive w/ ckpt only ran without memory overflow for compression rates 1% - 10% and naive w/o ckpt was only able to run for compression rate 1%. 'ckpt' denotes checkpointing. Averaged over 3 runs. . . . 77

4.1 The forward pass of a fully-connected layer. 106

4.2 **IID CIFAR-10/100 Training/Test Curves.** Accuracy convergence of `Comfetch` under varying compression rates with $N = 10$ clients. **(a)-(b)** corresponds to CIFAR-10 [3] image classification, while **(c)-(d)** correspond to CIFAR-100 image classification. In these experiments, only a single sketch is used and the datasets are IID. Similar accuracy with different `Comfetch` compression rates suggests that our method retains the expressive power of non-sketched models, while simultaneously reducing their storage size. 114

4.3 **Non-IID CIFAR-10/100 Training/Test Curves.** Accuracy convergence of Comfetch under varying compression rates in non-IID settings with $N = 10$ clients. **(a)-(b)** corresponds to CIFAR-10 image classification, while **(c)-(d)** corresponds to CIFAR-100 image classification. Datasets are non-IID with a Dirichlet split ($\alpha = 1$). Only a single sketch is used. Similar accuracy with different Comfetch compression rates suggests that our method retains the expressive power of the model while reducing the parameter counts even in the non-IID domain. 114
4.4 **Pruned Models.** $N = 4$ clients are given iid distributions of CIFAR-10 (12500 samples each) split across classes. Client model weights are then pruned randomly or based on ℓ_1 magnitude at compression rates of 0.1, 0.25 and 0.5. 115
4.5 Test accuracy achieved on predicting part of speech taggings for MNLI [4] sentences. On the left, we present quantitative comparisons to other methods. The horizontal line reflects the simple baseline where no compression is applied during training. On the right, we report the test accuracy with different compression rate while varying the K ratio. FetchSGD only compresses the network weights during communication, but each client still needs to decompress the entire network locally to perform training. 131
4.6 **Multi-Sketch Comfetch.** We assess the affect of using multiple sketches during training, for a single client training of CIFAR-10 with 87.5% compression. The use of multiple sketches has no noticeable effect on model performance. 134
4.7 Test accuracy convergence of 1-client ResNet-9 Comfetch under varying compression rates. **(a)-(b)** correspond to the CIFAR-10 [3] image classification. Similar accuracy with 1) different Comfetch compression rates suggests that our method retains the expressive power of the model while reducing the parameter sizes. 135

5.1 SWIFT schematic with $\mathcal{C}_s = \mathcal{C}_1$ (i.e., clients communicate every two local update steps). 143
5.2 Baseline performance comparison on CIFAR-10 for 16 client ring. 153
5.3 Average test loss for varying degrees of non-IIDness on CIFAR-10, 10 client ROC-3C. 155
5.4 SWIFT vs. D-SGD for CIFAR-10 in 16 client ring with varying slowdown. 157
5.5 Average communication and epoch times for increasing numbers of clients. . . . 158
5.6 Average test loss for varying network topologies on CIFAR-10. 159
5.7 An 8 Client Ring. 162
5.8 **(Left)** 10 Client, 3-Cluster. **(Middle)** 16 Client, 2-Cluster. **(Right)** 16 Client, 4-Cluster. 162
5.9 I_1, I_2 Depiction (from [6]). 165

6.1 **Hash-based dropout.** Two procedures for conducting cosine similarity estimation between full weight matrix $W \in \mathbb{R}^{d \times n}$ (column w_i denotes the weight of neurons i) and an input $x \in \mathbb{R}^d$. **(a)** SimHash generates a hash table via left-multiplication by a randomized rectangular Gaussian matrix $S \sim \mathcal{N}(0, I_d)$ onto the *fully-sized* W. **(b)** PGHash generates a hash table via left-multiplication by a randomized square Gaussian matrix $S \sim \mathcal{N}(0, I_c)$ onto a base projection $BW \in \mathbb{R}^{c \times n}$ of W. In both procedures, weight w_i is selected for activation if its signed hash matches the signed hash of x. 188

6.2 **Correlation between angle and Hamming distance.** We plot the average Hamming distance (x-axis) between a PG/SimHash hashed fixed vector x and collection of vectors $\mathcal{V}$ versus their true angles (y-axis). Vectors are unit, length 100, and hashed down to dimension $k = 25$ binary vectors according to $\tau = 10$ or 100 hash tables. PGHash has a sketch dimension of $c = 25$. Both PGHash and SimHash show strong correlation between Hamming distance and angular similarity. 199

6.3 **Compressed PGHash.** We record model accuracy of a large recommendation network on an extreme classification task (Delicious-200K) using PGHash for varying compression rates (CR). Compressed PGHash, even at 90% compression, is competitive with full training (*without even including effects of sparsity-induced neuronal drop out*). Hyperparameters are in Appendix 6.7.1.1. 202

6.4 **Federated Delicious-200K PGHash.** We record model accuracy of a large recommendation network on an extreme classification task (Delicious-200K) trained in a federated setting. PGHash achieves competitive accuracies compared with Federated SLIDE and FedAvg. In fact, PGHash converges quicker to near-optimal accuracy. Hyperparameters are in Appendix 6.7.1.1. 203

6.5 **Federated Amazon-670K PGHash-D.** We record model accuracy of a large recommendation network on Amazon-670K trained in a federated setting. PGHash-D matches the convergence of Federated SLIDE without requiring LSH to be performed by the central server. 204

6.6 **PGHash(-D) Computational Efficiency.** In Figure **6.6a**, we showcase that PGHash(-D) activates only a fraction of the total final layer neurons *even without compression*. Through this induced sparsity, PGHash(-D) greatly reduces the computational complexity of forward and backward training compared to full training. In Figures **6.6b** and **6.6c**, we compare PGHash(-D) with the Sampled Softmax heuristic [7] (randomly sampling 10% of the total neurons) for efficiently training recommender networks. PGHash(-D) outperforms the Sampled Softmax baseline, as it selects a better set of activated neurons via LSH to more efficiently train the recommender network. 205

6.7 **Wiki-325K PGHash-D.** We record model accuracy of a large recommendation network on Wiki-325K trained in a federated setting. PGHash-D matches or outperforms the convergence of SLIDE without requiring LSH to be performed by the central server. 209

6.8 **Angle/Hamming Distance as a function of sketch dimension.** The average Hamming distance between a PGHashed fixed unit vector $x \in \mathbb{R}^{100}$ and a collection of vectors $y_i \in \mathbb{R}^{100}$ which form different angles with x. Increasing sketch dimension c smooths and reduces the variance of the scatter towards linear correlation. Furthermore, the Hamming scales linearly with c, improving discernibility. (a)-(c) & (d)-(f) are independent series. 211
6.9 **Angle/Hamming Distance as a function of tables** The average Hamming distance between a PGHashed fixed unit vector $x \in \mathbb{R}^100$ and a collection of vectors $y_i \in \mathbb{R}^{100}$ which form different angles with x and fixed sketch dimension $c = 10$. Increasing the number of tables reduces variance. 211
6.10 **Distribution of folded norms.** 100k randomly drawn unit vectors ($d = 128$) are folded down to length 16 by are usual (d, c)-folding procedure. Depicted is a binned histogram of the norms. As predicted by the statistical description of $||BX||^2$, where X is a randomly drawn unit vector, the mass is centered at 1, i.e., most norms are preserved. Empirically, we observe that the folded norms rarely exceed $\sqrt{12816}$, although the theoretical support is $[0, 8]$: this concurs with the pdf. 215

7.1 **WAVES** establishes a standardized evaluation framework that encompasses a comprehensive suite of stress tests including both existing and newly proposed stronger attacks (denoted by *). 218
7.2 Evaluation workflow. 222
7.3 Regeneration attacks on Tree-Ringk. Regen-Diff is a single diffusive regeneration and Rinse-[N]xDiff is a rinsing one with N repeated diffusions, with the number of noising steps as attack strength. Regen-VAE uses a pre-trained VAE with quality factor as strength and Regen-KLVAE uses pre-trained KL-VAEs with bottleneck size as strength. RinseD-VAE applies a VAE as a denoiser after Rinse-4xDiff. 225
7.4 Adversarial embedding attacks target Tree-Ring at strengths of {2/255, 4/255, 6/255, 8/255}. Tree-Ring shows vulnerability to embedding attacks, especially when the adversary can access the VAE being used. 227
7.5 Three settings for training the surrogate detector. The Generator is the victim generator under attack. We externalize the watermarking process for simplicity, but it could be in-processing watermarks. After training the surrogate detectors, the adversary performs PGD attacks on them to flip the labels. 229
7.6 Adversarial surrogate detector attacks on Tree-Ring. 230
7.7 Unified performance vs. quality degradation 2D plots under detection setup. We evaluate each watermarking method under various attacks. Two dashed lines show the thresholds used for ranking attacks. 231
7.8 (left) Detection performance of three watermarks after attacks, measured by Average TPR@0.1%FPR with lower values (near center) indicating higher vulnerabilities. (right) The distribution of quality degradation. The lower the better. . . . 233
7.9 Identification accuracy of three watermarks after attacks. 235

7.10 **An illustration of a robust watermarking workflow.** An AI company provides two services: (1) generate watermarked images, i.e., embed invisible messages, and (2) detect these messages when shown any of their watermarked images. There is an attack stage between the watermarking and detection stages. The watermarked images may experience natural distortions (e.g., compression, rescaling) or manipulated by malicious users attempting to remove the watermarks. A robust watermarking method should still be able to detect the original message after an attack. 238
7.11 Word clouds of DiffusionDB, MS-COCO, and DALL·E3 prompts. 251
7.12 Image examples of DiffusionDB, MS-COCO, and DALL·E3. 251
7.13 **Ranking attacks with different quality metrics** on DiffusionDB images watermarked by Tree-Ring. Attack potency is ranked by image quality at 0.95 TPR@0.1%FPR. Colors indicate the ranks (1=best, 9=worst), and values show the measured quality. We use 'NA' to label an attack if its attack curve lies entirely above TPR=0.95; the attack is automatically ranked last. 254
7.14 Cumulative distribution functions (CDFs) for eight image quality metrics across all attacked watermarked images. The horizontal dashed lines mark the 10% and 90% quantiles, and the intersecting vertical dashed lines delineate the bounds of the normalization intervals. Values at the lower bound are normalized to 0.1, and those at the upper bound to 0.9. 257
7.15 Distortions and their combinations. We combine three types of distortions: geometric, photometric, and degradation, both individually and collectively. By comparing quality-performance plots, we see combinations of distortions do not necessarily lead to better attacks. 260
7.16 Regenerative diffusion with varying depth of noising steps and a VAE regeneration with a low quality factor. 261
7.17 4x rinsing regeneration with varying depth of noising steps per diffusion. 261
7.18 An image of a dragon attacked using a 3x rinsing regeneration. Pushing the image through a VAE restores image quality, noticeable in the eye color of the dragon (indicated by the green box). Image is drawn from the Gustavosta Stable Diffusion dataset available. 262
7.20 The spoofing attack fails for AdvCls-UnWM&WM. 265
7.19 **Aggregated performance vs. quality degradation 2D plots under identification setup (one million users).** We evaluate each watermarking method under various attacks. Two dashed lines show to thresholds used for ranking attacks. . . 266
7.21 Visualization of AdvCls-UnWM&WM attack. (a) shows the watermarking mask of Tree-Ring where there are four channels, and we only watermark the last channel. The watermark message is the rings, which contain ten complex numbers that are not shown in the figure. (b) and (c) show the inversed latent before and after the attack in the Fourier space. We only show the real part of the latent. Clearly, the rings exist before the attack and vanish after the attack. (d) shows the magnitude of the element-wise difference before and after the attack. The attack not only perturbs the watermark part but also other features. The average magnitude change of the watermark-part and non-watermark-part is around 2:1. The attack successfully disturbs the watermark, albeit in an imprecise manner. 267

7.22 Visualization of AdvCls-WM1&WM2 attack. (a) and (b) are the same as that in Figure 7.21. (c) shows the inversed latent after the attack, where the watermark vanishes instead of changing to another watermark. (d) shows the magnitude of the element-wise difference before and after the attack. The attack not only perturbs the watermark part but also other features. The average magnitude change of the watermark-part and non-watermark-part is also around 2:1. Although the surrogate detector is trained to classify two different watermark messages. The attack based on it cannot change the watermark message from one to another but can effectively disturb the watermark. 268
7.23 The user identification results for Tree-Ring under AdvCls-WM1&WM2 attacks. The original watermarked images are embedded with User1's message. AdvCls-WM1&WM2 tries to disrupt the latent feature of those images so that they can be misidentified as User2 generated. We simulate two settings: 100 users and 1000 users in total. The blue curves represent the proportion of images correctly identified as belonging to User1, while the orange curves show those misidentified as User2's. Note that, the axes for blue and orange curves have different ranges in the figure. With increasing attack strengths, the likelihood of correctly identifying them as User1's decreases significantly under both 100 and 1K user scenarios. However, misidentification as User2's images occurs notably only when the total number of users is small (e.g., 100 users). 270
7.24 A visual demonstration of various adversarial, regeneration, and distortion attacks on a Tree-Ring watermarked image. **Figure (a)** is the base unattacked image. The base prompt, drawn from DiffusionDB, is "digital painting of a lake at sunset surrounded by forests and mountains," along with further styling details. 271
7.25 Evaluation on DiffusionDB dataset under the detection setup (part 1). 272
7.26 Evaluation on DiffusionDB dataset under the detection setup (part 2). 273
7.27 Evaluation on MS-COCO dataset under the detection setup (part 1). 274
7.28 Evaluation on MS-COCO dataset under the detection setup (part 2). 275
7.29 Evaluation on DALL·E3 dataset under the detection setup (part 1). 276
7.30 Evaluation on DALL·E3 dataset under the detection setup (part 2). 277

List of Abbreviations

ADPSGD	Asynchronous Decentralized Stochastic Gradient Descent
AI	Artificial Intelligence
CS	Count Sketch
CNN	Convolutional Neural Network
DFL	Decentralized Federated Learning
DNN	Deep Neural Network
FL	Federated Learning
FLOP	Floating Point Operation
HPC	High-Performance Computing
IoT	Internet of Things
LSH	Locality-Sensitive Hashing
ML	Machine Learning
PCA	Principal Component Analysis
SGD	Stochastic Gradient Descent
SotA	State of the Art

Chapter 1: Introduction

1.1 Background & Motivation

Over the last decade, two trends have become readily apparent in machine learning (ML): models (1) continue to improve in task performance and (2) they continue to grow larger. Figure 1.1 depicts a seemingly exponential growth in the parameter count of architectures within the last 5 years. Reasons for this pattern are subject to debate, but this phenomenon may be (somewhat circularly) encouraged by the empirical observation that larger models perform better, especially on language and vision tasks [8, 9, 10, 11, 12].

Research and development on the Internet of Things (IoT) has similarly seen dramatic leaps in quality and popularity. The general public is now able to build at-home multi-device networks capable of wirelessly delivering personalized services. Wireless networks composed of low-power sensors (WSNs) are used in both healthcare and telecommunications to collect vast amounts of data over a wide geographic scale [13]. However, due to computational and memory constraints, deep learning on clients such as mobile devices [14] is limited to comparatively smaller model sizes. Popular vision models such as the MobileNet family [15] are light ($<$10M parameters), but cannot perform the same feats as massive, multi-billion parameter diffusion models. The compute required in the large-scale era of ML is similarly infeasible for low-resource clients; popular generative models typically require upwards of 4×10^{21} FLOPs for

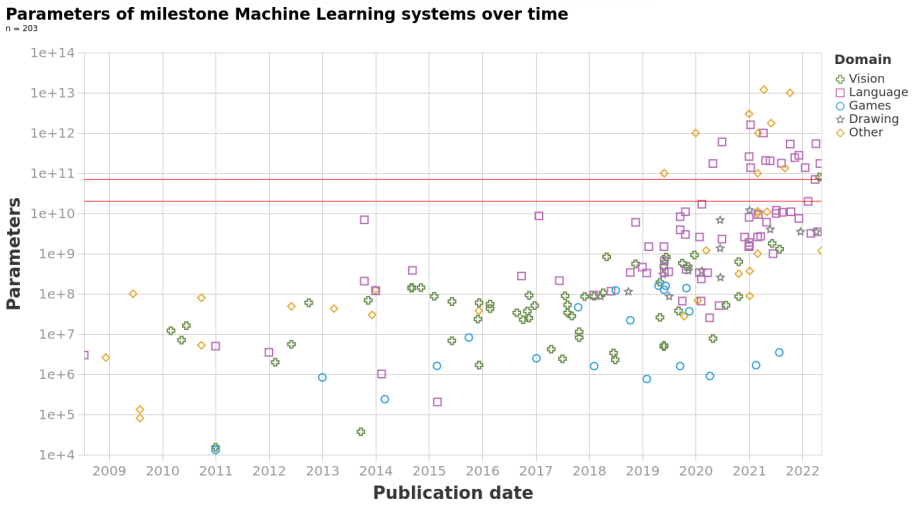


Figure 1.1: The parameter counts of notable machine learning models grow with time [5]

.

training. Clusters of dedicated graphics cards enable within-lifetime inference and training over these networks, but even state-of-the-art models tailored for mobile devices require off-device pre-training via modern GPUs and TPUs [15, 16, 17]. In light of this hardware dilemma, we focus on two general strategies for resource-constrained systems to conduct on-device training large models:

- ***Go smaller.*** A distinctive feature of most deep neural architectures is its weights, which are matrices whose entries comprise the parameters of the model. In the large-scale era of ML [18], recommender networks, large language models (LLMs), and diffusion model weights can consist of consist of tens to hundreds of thousands of parameters [18, 19, 20]. Inference and training on large weights through modern DL library requires large amounts of RAM and tensor parallelism [21]. Indeed, the blessing of dimension appears to be at work [18]; larger models seemingly generalize better on a myriad of tasks [22, 23, 24] and may induce strong convexity-like landscapes [25]. However, it is both theorized and observed that large pre-trained networks contain small subnetworks with generalization capabilities similar to

their progenitors [26, 27]. Similarly, sparsified and pruned pre-trained networks can be highly performant [28, 29].

In this thesis, we consider the natural setting in which devices cannot ever host a large model in its entirety for any portion of training, thereby rendering typical procedures such as knowledge distillation [30], quantization [31, 32], and gradual dropout [33, 34, 35, 36, 37] inapplicable. This dissertation answers several questions regarding during-training compression. In the context of federated learning: can using a dynamically-changing, small, and sparse architecture converge to an acceptable global model [38]? Do randomized-projection techniques such as the Count Sketch assist with performance-preserving model shrinkage [39]? Does compactification of weights via tensorization leave capacity intact [40]?

- ***Go faster.*** This approach focuses on schemes that reduce the overall iteration complexity of the task. In algorithmic terms, we are lowering the number of iterations required for convergence in the core for-loop associated with the optimization program defining the task. The net computational complexity of the proposed algorithm should be lower than a competitor or vanilla execution of the program, even if the per-loop computational complexity is increased. This strategy may also refer to a reduced expenditure of floating point operations per second (FLOPs) per iteration, but again, with the assurance that overall computation is reduced. As many typical learning tasks require many redundant, yet cumulatively expensive BLAS-like calculations [41], minimizing execution time drastically reduces energy expenditure thereby preserving the hardware health and battery life of the device.

In this work, we seek to speed up training using novel variations of classical approaches. For principal component analysis or spectral clustering, we develop a practical momentum-based power method [42, 43] to accelerate eigenvector discovery [44]. In an FL scenario, we develop a wait-free, peer-to-peer training procedure which employs asynchronous parallel stochastic gradient descent (ADPSGD) [45, 46] to improve iteration complexity [47]. For tensorized networks, we speed up inference through FLOPs-minimising evaluation order of tensorial multilinear operations [40].

Both of the above approaches are complementary; reduction in model size and the number of parameters inherently results in less taxing forward passes, backpropagation, and inference. Consequently, these two threads of research will often intertwine throughout this manuscript.

1.2 Organization of Dissertation

In Chapter 2, we set the stage for efficient processing of large neural weights by considering eigenvector extraction of arbitrarily large matrices. We introduce a practical variation of a momentum-based power method [42], which traditionally relies on unrealistic spectral knowledge (the second eigenvalue, λ_2) at runtime. Instead, we use inexact approximations of λ_2 obtained through a traditional power method. In Chapter 3, we develop a unifying framework for representing tensorial convolution layers as einsum-like strings and a meta-algorithm conv_einsum which is able to evaluate these strings in a FLOPs-minimizing manner.

Chapters 4 to 7 are focused on acceleration and compression in an FL environment. These chapters enforce private training protocols via collaborative training of clients over a connected communication graph. In particular, client-held data is never exposed. For model compression,

this setting poses several obstacles to conventional approaches due to the following: training data only lives on resource-constrained clients rendering conventional knowledge distillation infeasible, and fully-sized models cannot ever be stored in the RAM of the clients, thus eliminating post-hoc compression and gradual dropout as viable options. In Chapter 4, we propose a simple yet effective novel algorithm that allows clients to train large networks using reduced representations of the global architecture via Count Sketch. In Chapter 5, we consider the peer-to-peer FL setting (decentralized FL). We propose an asynchronous, wait-free decentralized FL algorithm that allows clients to conduct training at their own speed without being bottlenecked by slow neighbors. In Chapter 6, using a new family of hash functions, we develop one of the first private, personalized, and memory-efficient on-device locality-sensitive hashing (LSH) frameworks. Our framework allows clients to significantly prune a massive classification layer weight by up to 90% without ever hosting the fully-sized model, which is especially useful for training over extreme multi-label recommender datasets that contain hundreds-of-thousands possible labels.

Chapter 7 continues along the theme of privacy, but this time with a focus on watermarking generative imagery. We introduce novel benchmark of stress tests for assessing watermark robustness, overcoming the limitations of current evaluation methods. Several of these tests make use of compression algorithms to alter the latent representation of target images thereby corrupting the watermark. Our comprehensive evaluation reveals previously undetected vulnerabilities of several modern watermarking algorithms. We conclude with a summary of this dissertations tying the contributions of all chapters together.

Chapter 2: Practical and Fast Eigenvector Discovery

2.1 Introduction

Approximating the dominant eigenvector of a matrix $A \in \mathbb{R}^{d \times d}$ is a task common to many statistical and industrial applications. The vanilla power method is a simple and inexpensive algorithm for computing the dominant eigenvector v_1 of a matrix A. For an initial $q_0 \in \mathbb{R}^d$ non-orthogonal to v_1, the power method performs the following update which converges to v_1,

$$q_k = A^k q_{k-1} / ||A^k q_{k-1}||. \tag{2.1}$$

Owing to its ease of implementation and modest assumptions for convergence, the power method has found its use in a variety of machine learning tasks. It can be used to assist the k-means algorithm for class separation of large datasets, which is referred to as power iteration clustering (PIC) [48, 49, 50]. It is also used in sparse PCA, which projects data onto sparse principal components, that is, components with small ℓ_0 norm [51, 52]. In particular, word-embedding matrices for NLP models can be dimensionally-reduced via sparse PCA [53, 54].

De Sa et al. [42] introduced the *power method with momentum*, abbreviated as Power+M along with a stochastic variant, Mini-Batch Power+M. Both of these methods outperform their vanilla counterparts (the stochastic version of equation 2.1 uses instead an unbiased estimate $\widehat{A}$ of

A). Here, speed is measured in the sense of iteration complexity, i.e., the number of outer loop iterations/updates required to output a vector q_k with precision ϵ, i.e., $\sin^2\theta(q_k, v_1) \triangleq 1-(q_k^\top v_1)^2 < \epsilon$. Recently, other algorithms have been developed which adopt a momentum-based scheme [55, 56]. However, a notable drawback to Power+M, Mini-Batch Power+M, and other existing momentum-based methods is that achieving accelerated iteration complexity requires knowledge of λ_2, the second greatest eigenvalue of A, which is an impractical assumption. Specifically, the momentum coefficient β, a hyperparameter of momentum-based power methods, must be set near $\lambda_2^2/4$ to achieve improved iteration complexity over the vanilla power method.

In this work, we develop a scheme which enjoys a near-optimal acceleration without the strict spectral knowledge requirements of other momentum-based methods. Our scheme consists of two phases. In the *pre-momentum phase*, we run a vanilla (or stochastic for the online setting) power method to approximate the dominant eigenvector and an inexact Hotelling deflation [57] to estimate λ_2 and later assign $\beta \approx \lambda_2^2/4$ (our momentum coefficient). In the *momentum phase*, we run Power+M (or Mini-Batch Power+M for the online setting) with the near-optimal β assignment taken from the previous phase for the remaining iterations until convergence. Our main contributions, the *delayed momentum power method* (DMPower) and the *delayed momentum streaming power method* (DMStream), are realizations of this scheme.

Relaxing Spectral Knowledge. As explained above, an optimal acceleration of momentum-based methods relies on the proper selection of β. Previous approaches rely on expensive guess-and-check auto-tuning, whereby the user chooses β, and at each round conducts many experimental iterations with 0.67β, 0.99β, β, 1.01β, etc., revising the coefficient after determining which adjustment results in the largest Rayleigh quotient [42]. To the best of our knowledge,

DMPower and DMStream are the first momentum-based algorithms to approximate optimal β in a partially-adaptive manner while still benefiting from acceleration. We present an informal version of our major results.

Theorem 1 (Informal). *Let $\Delta = |\lambda_1 - \lambda_2|$ denote the absolute difference between the largest and second-largest eigenvalues. With high probability, our proposed practical* DMPower*, after an efficient pre-momentum warm-up stage, outputs an ϵ-close estimate of the leading eigenvector within the state-of-the-art $\mathcal{O}\left(\frac{1}{\sqrt{\Delta}}\log\left(\frac{1}{\epsilon}\right)\right)$ iteration complexity using a momentum acceleration, without requiring knowledge of λ_2 or hyperparameter selection for λ_2.* DMPower *is extended to* DMStream *in the streaming setting, with similar iteration complexity.*

In section 2.4 we provide the full version the above theorem along with a companion streaming theorem. Although the momentum phases utilize existing methods, neither of our algorithms are true hybrids; DMPower and DMStream are the first of their kind to utilize inexact Hotelling deflation for second eigenvalue recovery with guarantees. We will show that the iteration complexity of the DMPower and DMStream momentum phases respectively match the iteration complexity of Power+M and Mini-Batch Power+M without having to assign β at initialization.

Our experiments show that DMPower converges faster than the vanilla power method to various precisions of estimation and for matrices with tighter eigengaps, $\Delta = 0.01, \Delta = 0.001$, our DMPower matches the convergence speed of Power+M running with optimal β. DMPower also outperforms than the vanilla power method and closely mimics the performance of optimal Power+M when employed in the unsupervised learning task of spectral clustering. Additionally, in comparison to another concurrent iteration method, the simultaneous power iteration, our

experiments demonstrate that DMPower determines a more accurate approximation of λ_2. DMStream outputs a more accurate estimation of the dominant eigenvector for a variety of batch sizes when compared against Oja's algorithm and performs nearly identically to Mini-Batch Power+M initialized with optimal β.

Summary of Contributions.

1. Our proposed algorithms DMPower and DMStream achieve close to optimal performance when compared against other momentum-based power methods, which have been initialized with a priori unknowable optimal hyperparameters.

2. To the best of our knowledge, we are to first to provide a convergence analysis of an inexact deflation receiving approximate dominant eigenvectors supplied by a power iteration, both in the deterministic setting (Lemma 12) and streaming setting (Proposition 21).

3. Our Proposition 7 provides a guarantee for acceleration if one can provide lower bounds on $|\lambda_1 - \lambda_2|$ and $|\lambda_2 - \lambda_3|$, which is a far more practical requirement at run-time.

4. For many estimation precision requirements, DMPower outperforms the state-of-the-art Lanczos algorithm in iteration complexity when factoring in the recommended number d of tri-diagonalization iterations needed for numerical stability of the Lanczos algorithm. Additionally, DMPower runs noticeably faster (in seconds) than the Lanczos algorithm.

2.2 Related Works

Speedy Deterministic Power Methods. Variations of the vanilla power method intended to improve its iteration complexity of $\mathcal{O}(\frac{1}{\lambda_1 - \lambda_2} \log \frac{1}{\epsilon})$ have been suggested. The Lanczos al-

gorithm [58] itself may be thought of as a fast variation of the power method, which has iteration complexity $\mathcal{O}(\frac{1}{\sqrt{\lambda_1-\lambda_2}}\log\frac{1}{\epsilon})$ but only after a tri-diagonalization process which is expensive in high dimensions. Lei et al. [59] consider a coordinate-wise update with complexity $\mathcal{O}\big(\frac{\lambda_1}{\lambda_1-\lambda_2}\log\frac{\tan\theta_0}{\epsilon}\big)$. Based on the heavy ball method first studied by Polyak [43], De Sa et al. [42] first proposed the addition of a momentum term to accelerate the basic power method. The term is controlled by a momentum coefficient, β, which is a hyperparameter selected at initialization. Their method achieves $\mathcal{O}\big(\frac{1}{\sqrt{\lambda_1-\lambda_2}}\log\frac{1}{\epsilon}\big)$, but only with a precise selection of β requiring unrealistic spectral knowledge at run-time. Mai and Johannson [56] created a momentum-based algorithm `NAPI` for solving the canonical correlation analysis (CCA) problem similar requiring impractical hyperparameter selection. In contrast, our proposed DMPower requires no spectral knowledge and achieves the same speed as existing momentum methods running optimally.

Block Iterations. To enjoy the effects of acceleration, DMPower concurrently runs a second power iteration for a finite number of rounds and extracts an optimal momentum coefficient. This approach is reminiscent, though not the same as block power iterations such as the simultaneous power iteration for matrices [60], extended to tensors by Wang and Lu [61], which intends to recover multiple eigenvectors at once. The simultaneous power method for matrices is well-known to suffer from rounding errors [58][62]. In contrast, our inexact deflation is experimentally shown to achieve greater accuracy in extracting the second eigenvalue, which is critically important when eigengaps are tight. The improved "practical" simultaneous iteration and QR algorithm [62] compute a QR factorization followed by a reversed RQ computation, which requires $O(d^2)$ more flops than inexact deflation. Furthermore, these block iterations do not employ any acceleration schemes: the convergence rate of an ϵ-close approximation of v_1 in a 2-vector simultaneous iteration is $\mathcal{O}\big(\frac{1}{\min\{\lambda_1-\lambda_2,\lambda_2-\lambda_3\}}\log\frac{1}{\epsilon}\big)$.

Inexact Power Methods. In the pre-momentum phase, DMPower relies on an inexact deflation step every round. This step may be regarded as a Hardt-Price noisy power method, which has been studied in differential privacy-preserving PCA [63] [64], but primarily as a meta-algorithm. The DMPower may be regarded then as one practical application of the noisy power method, with an added momentum phase and thus requiring novel analysis beyond the scope of existing noisy power method literature. Furthermore, most existing results on deflation schemes operate under the assumption that eigenvectors and eigenvalues have been exactly recovered, but in practice, only approximate eigenvectors are used. To the best of our knowledge, we are the first to provide a guarantee on obtaining a second eigenvalue through deflation using inaccurate dominant eigenvectors (Lemma 12 and Proposition 21). Our algorithm is the first in the class of momentum-based power methods to incorporate an acceleration scheme without the requirement of guessing an approximation of the second eigenvalue at runtime.

Streaming Methods. DMStream is a streaming companion to DMPower, which instead uses unbiased estimates of the underlying covariance matrix A for all of its updates. The streaming setting assumes that A is either inaccessible or expensive to obtain, and several algorithms have been developed to address this situation [55, 65, 66, 67]. The momentum phase of DMStream is built upon Mini-Batch Power+M [42], whose iteration complexity is $\mathcal{O}\left(\frac{1}{\sqrt{\lambda_1-\lambda_2}}\log\left(\frac{1}{\epsilon}\right)\right)$, which is desirable in that it matches the offline state-of-the-art Lanczos complexity, but similar to Power+M, requires a momentum hyperparameter β close to $\lambda_2^2/4$. Just like DMPower, DMStream efficiently approximates λ_2 in its pre-momentum phase and then drops into a momentum phase where it enjoys the aforementioned optimal offline iteration complexity.

Gradient Descent Methods. From the optimization perspective, leading eigenvector computation is equivalent to minimizing $-\mathbf{x}^\top A\mathbf{x}$ where $\mathbf{x} \in S^2$, the unit sphere. Although this

problem is non-convex, it is possible to use gradient descent to solve this problem [68, 69, 70].In particular, the global convergence rate has been shown to be $\mathcal{O}\left(\left(\frac{\lambda_1}{\lambda_1-\lambda_2}\right)^2 \log \frac{1}{\epsilon}\right)$, which is generally incomparable to the guarantee of DMPower [71]. Conjugate gradient (CG) methods have also been employed to compute the dominant eigenvector. The Fletcher-Reeves Gradient Descent (FRGD) [72], a fully-adaptive momentum-based CG method, achieves a convergence rate of $\mathcal{O}\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)$, where $\kappa = \lambda_1/\lambda_n$ and thus this rate is also not directly comparable to the convergence rate of DMPower and other classical power methods, since they largely depend on the first eigengap.

2.3 Accelerated Momentum-Based Power Methods via Inexact Deflation

Problem Setup. We first outline the setting and assumptions typical of deterministic momentum-based PCA. We discuss the streaming setting in section 2.3.2. Let $x_1, x_2, \ldots, x_n \in \mathbb{R}^d$ be data points. Our goal is to recover the top eigenvector and dominant eigenvalue of the symmetric PSD covariance matrix $A = \frac{1}{n}\sum_{i=1}^n x_i x_i^\top \in \mathbb{R}^{d\times d}$. We assume that A has eigenvalues $1 \geq \lambda_1 > \lambda_2 > \lambda_3 \geq \lambda_4 \geq \cdots \geq \lambda_d \geq 0$ with associated orthonormal eigenvectors $v_1, v_2, \ldots, v_d$. Unless noted otherwise, $||\cdot||$ refers to the 2-norm for vectors and matrices. We let $\Delta_{1,2} := \lambda_1 - \lambda_2$ and $\Delta_{2,3} := \lambda_2 - \lambda_3$. The vanilla power method, at each round $k = 1, 2, \ldots$ performs the following update,

$$q_k = \frac{A^k q_{k-1}}{||A^k q_{k-1}||} \tag{2.2}$$

$$\nu_k = q_k^\top A q_k \tag{2.3}$$

where q_0 is a random unit vector non-orthogonal to v_1. Under these conditions, $q_k \to v_1$ and $\nu_k \to \lambda_1$ at a geometric convergence rate, with ratio $\left(\frac{\lambda_2}{\lambda_1}\right)^2$. Here, error is measured as the sine squared of the angle $\theta(q_k, v_1)$ between our unit q_k and v_1, that is, $\sin^2\theta(q_k, v_1) \triangleq 1 - (q_k^\top v_1)^2$. The power method with momentum uses the alternative update,

$$q_k = \frac{Aq_{k-1} - \beta q_{k-2}}{||Aq_{k-1} - \beta q_{k-2}||} \tag{2.4}$$

where $q_{-1} = \mathbf{0}$ and β is the *momentum coefficient* chosen by the user at initialization. Sa et al. [42] establish the following theorem and its corollary,

Theorem 2 (Convergence of Power+M [42])**.** *Given a PSD matrix $A \in \mathbb{R}^{d\times d}$ with eigenvalues $1 \geq \lambda_1 > \lambda_2 \geq \lambda_3 \ldots \lambda_d \geq 0$, running with $\lambda_2 < 2\sqrt{\beta} \leq \lambda_1$ results in q_k with*

$$\sin^2\theta(q_k, v_1) = 1 - (q_k^\top v_1)^2 \leq \frac{4}{|q_0^\top v_1|^2}\left(\frac{2\sqrt{\beta}}{\lambda_1 + \sqrt{\lambda_1^2 - 4\beta}}\right)^{2k} \tag{2.5}$$

Corollary 3 ([42])**.** *For $\epsilon \in (0, 1)$ after $T = \mathcal{O}\left(\frac{\sqrt{\beta}}{\sqrt{\lambda_1^2 - 4\beta}} \log\frac{1}{\epsilon}\right)$ iterations,* $\sin^2\theta(q_T, v_1) \leq \epsilon$.

We have from these results that $\beta \in [\lambda_2^2/4, \lambda_1^2/4)$, and minimizing $\frac{\sqrt{\beta}}{\sqrt{\lambda_1 - 4\beta}}$ as in Corollary 3 over this interval results in an optimal assignment $\beta = \lambda_2^2/4$.

Practical Considerations. The power method with momentum is capable of achieving a faster convergence rate than the vanilla power iteration. For $\beta = \lambda_2^2/4$, we have, as in Theorem 2, a geometric convergence with ratio $\left(\frac{\lambda_2}{\lambda_1 + \sqrt{\lambda_1^2 - \lambda_2^2}}\right)^2$, which is smaller and thus faster than the vanilla convergence ratio $\left(\frac{\lambda_2}{\lambda_1}\right)^2$. However, therein lies the impracticality of the algorithm: the user will generally not have knowledge of λ_2. Guessing the momentum coefficient $2\sqrt{\beta} > \lambda_1$

can result in extremely slow convergence and in some cases, divergence, as shown in Figure 2.1. In fact, there is currently no known convergence guarantee for $2\sqrt{\beta} > \lambda_1$; we notice that in the setting of Theorem 2 and its associated Corollary, that such selection of β results in an imaginary ratio, rendering the guarantee uninformative. For practical use, we should remove β as a hyperparameter.

Ultimately then, we must approximate λ_2. This naturally leads us to consider a concurrent/block iteration scheme that synchronously converges towards v_1 and v_2. From our approximations of v_2, we may obtain Rayleigh quotient estimations of λ_2. Accuracy is critical when the eigengap $\Delta_{1,2}$ is small, and since the simultaneous power iteration experiences rounding errors as shown in Figure 2.2, it is not appropriate for use. The more accurate *practical simultaneous power iteration* [62] employs a thin QR factorization at each step, which is expensive. Our instinct, then, is to adopt a concurrent iteration which utilizes *matrix deflation* to approximate λ_2.

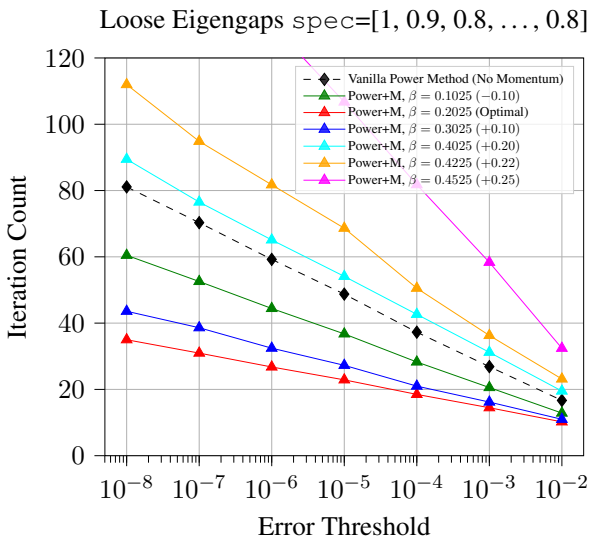


Figure 2.1: **Sub-optimal β Selection for Power+M**. We measure convergence speeds at various momentum assignments. The X-axis is the error threshold ϵ between approximates q_k, q_{k-1} of v_1 needed for Power+M to terminate, i.e., we run until $||q_k - q_{k-1}|| < \epsilon$. The Y-axis is the total number of iterations k needed to meet this condition according to the update equation (2.4). Averaged over 1000 runs, each time run on a random PSD $A \in \mathbb{R}^{10\times 10}$ with spectrum $\lambda_1 = 1, \lambda_2 = 0.9, \lambda_3 = 0.8$, and remaining eigenvalues set to 0.8. We observe that increased deviation from the optimal β assignment results in worsened and eventually divergent performance.

Approach for Smart Selection of β. Deflation methods extract further eigenvalues along the spectrum (ordered in absolute modulus), once previous eigenvalues and eigenvectors are de-

termined. Hotelling deflation [58] is one such scheme upon which we model our algorithms. Assume for our symmetric PSD A that it also has a positive second eigengap, i.e., $\lambda_2 > \lambda_3$. If we form the deflation matrix $B = A - \lambda_1 v_1 v_1^\top$, then for w_0 non-orthogonal to v_2, we have that $w_k = \frac{Bw_{k-1}}{||Bw_{k-1}||} \to v_2$ and $q_k^\top B q_k \to \lambda_2$ as $k \to \infty$. Clearly, we do not have access to λ_1 and v_1 (their approximation is the entire purpose of PCA), but at each round of a vanilla power iteration, we *do* have approximations ν_k and q_k as in update equations (2.2) and (2.3), so we may instead form an inexact deflation matrix $A - \nu_k q_k q_k^\top$.

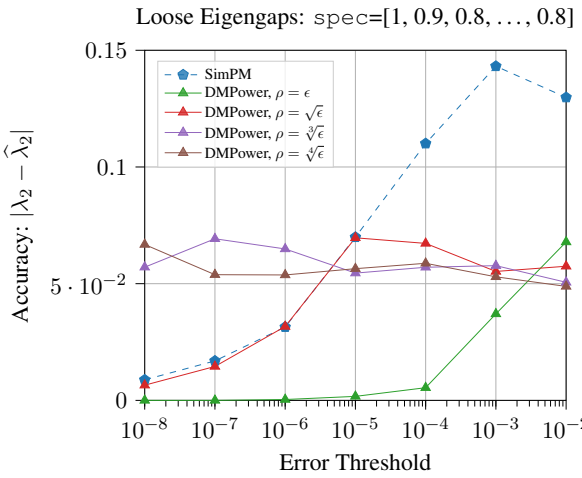


Figure 2.2: **Accuracy vs Simultaneous Power**. We measure the Rayleigh quotient accuracy in determining λ_2 between the Simultaneous Power Iteration (SimPM) and DMPower. The X-axis is the error threshold ϵ of the entire DMPower run. We report the true accuracy of $\widehat{\lambda}_2$ output by the pre-momentum phase, which is reflected in the Y-axis. Averaged over 1000 runs, each time run on a random PSD $A \in \mathbb{R}^{10\times 10}$. We varied ρ according to ϵ for convenience, but ρ does not depend on ϵ in implementation. We observe that DMPower is overall the most accurate, while SimPM is inaccurate at low thresholds.

One might wonder the implications of setting β as $q_J^\top A q_J$. Setting $\beta = q_J^\top A q_J$ will result in a momentum parameter converging towards λ_1, which would not lie in the convergence zone $[\lambda_2^2/4, \lambda_1^2/4)$ of Theorem 2. The next logical idea then would be to set $\beta = (q_J^\top A q_J)^2/4$, which would converge towards $\lambda_1^2/4$, but employing this coefficient very nearly follows the same dynamics as the vanilla power method (see Theorem 2) and hence, no improvement.

Our first proposed algorithm the *delayed momentum power method* (DMPower) is a realization of the above discussion and considerations, using inexact deflation to progressively approximate λ_2. Experimentally, DMPower outperforms the vanilla power method for all speci-

fied error thresholds, running near-optimally at tighter eigengaps $\Delta_{1,2} = 0.01$ and $\Delta_{1,2} = 0.001$. DMPower experiences decayed noise at each step, making it possible to establish a convergence guarantee, which we present in Theorem 4.

2.3.1 Delayed Momentum Power Method (DMPower)

Intuitions Behind DMPower. The delayed momentum power method, our primary contribution, experiences momentum-based acceleration after an initial waiting period with no required selection of β at initialization. As stated before, we use inexact Hotelling deflation to approximate $\lambda_2^2/4$. It is known that inexact deflation can succeed with controlled noise [63, 64], and we prove in Section 2.7.2 that the inexact deflation step in DMPower satisfies such conditions. We will eventually obtain an estimate $\beta \in [\lambda_2^2/4, \lambda_1^2/4)$. Using this well-behaved β, we can transition to a momentum-based update exhibiting acceleration.

Overview. Algorithm 1 describes DMPower. It proceeds as a routine vanilla power method, but at each step, our approximate top eigenvector and eigenvalue q_k and ν_k are used to form an inexact deflation matrix $(A - P) = A - \nu_k q_k q_k^\top$, which is run in a power-iterative manner on an initial vector w_0, which is non-orthogonal to v_2. We refer to this portion of DMPower as the *pre-momentum phase*. For a practical implementations, the first for-loop would exit once $|\mu_{k+1} - \mu_k| \leq \rho$, for some user-specified ρ, i.e., once our λ_2 approximates are close to each other. We discuss at length the practical and theoretical considerations of selecting ρ in section 2.4.3.

After achieving ρ-accuracy between our approximates μ_k, we set $\beta \leftarrow \mu_k^2/4$, and $q_0 \leftarrow q_j$ as our new initial vector. We then proceed to the *momentum phase*, which runs Power+M updates

Algorithm 1 Delayed Momentum Power Method (DMPower)

Require: $A \in \mathbb{R}^{d\times d}$ symmetric, unit $q_0 \in \mathbb{R}^d$, pre-momentum phase iterations J, momentum phase iterations K, unit $w_0 \in \mathbb{R}^d$

for $j = 1, 2, \ldots, J$ **do**
  $q_j \leftarrow Aq_{j-1}$
  $q_j \leftarrow q_j/||q_j||$
  $\nu_j \leftarrow q_j^\top Aq_j$ ▷ Rayleigh Quotient estimate of λ_1
  $P \leftarrow \nu_j q_j q_j^\top$
  $w_j \leftarrow (A - P)w_{j-1}$ ▷ Inexact deflation
  $w_j \leftarrow w_j/||w_j||$
  $\mu_j \leftarrow w_j^\top Aw_j$ ▷ Rayleigh Quotient estimate of λ_2
end for
$\widehat{\lambda}_2 = \mu_J$
$\beta \leftarrow \widehat{\lambda}_2^2/4$ ▷ Approximated optimal momentum coefficient
$q_1 \leftarrow q_J$ ▷ Current estimate of v_1
$q_0 \leftarrow \mathbf{0}$
for $k = 1, 2, \ldots, K$ **do** ▷ while $\|q_k - q_{k-1}\| > \epsilon$
  $q_{k+1} \leftarrow Aq_k - \beta q_{k-1}$ ▷ Momentum update
  $q_{k+1} \leftarrow q_{k+1}/||q_{k+1}||$
  $\nu_k \leftarrow q_{k+1}^\top Aq_{k+1}$
end forreturn q_K, ν_K

until ϵ-accuracy is achieved among the q_k. Notice that q_j has already made progress towards v_1. This greatly benefits our Power+M updates according to Theorem 2, where $\sin^2\theta(q_t, v_1)$ is limited at each step by the constant $\frac{4}{|q_j^\top v_1|}$. We provide a convergence guarantee in Theorem 4 for DMPower.

Complexity. The vanilla power iteration costs $\mathcal{O}(d^2)$ flops per round, and since we are concurrently running two vanilla power methods in addition to a Rayleigh quotient, we perform two additional $\mathcal{O}(d^2)$ flops. Although we are not asymptotically increasing the time complexity, we justify the increased flops: we prove in Theorem 4 that for a fixed ρ, we will achieve our desired approximation of λ_2 after a finite number of rounds. Furthermore, we are only interested in the Rayleigh quotient approximations μ_j of λ_2 from each deflation step, and it is well-known that if $||v_2 - w_k|| = \rho$, then $|\lambda_2 - \mu_k| = \mathcal{O}(\rho^2)$ [60]. That is, the Rayleigh quotient is a quadratically-

accurate estimate of v_2.

2.3.2 Delayed Momentum Streaming Power Method (DMStream)

Overview. We first review a typical setting for streaming PCA. Assume we have a stream of $\mathbb{R}^d$ inputs $x_1, x_2, \ldots$ drawn from some unknown distribution $\mathcal{D}$ with underlying covariance matrix A. We wish to recover the dominant eigenvector v_1 of A.

Streaming algorithms have risen to address this challenge, which instead use a different unbiased estimate $\widehat{A}_t = \frac{1}{n}\sum_{i=1}^{n} x_i x_i^\top$ of A at each round t to conduct their updates, where n is a fixed batch size, and the x_i's are selected in a uniformly random manner [42, 65, 66, 67]. In general, the total iteration complexity and runtime is dependent on several factors, including the variance $\mathbb{E}[(\widehat{A}_t - A) \otimes (\widehat{A}_t - A)]$ of each unbiased estimate (where $\otimes$ denotes the Kronecker product) and the batch size. The sample complexity is the total number of streaming inputs need overall to output an ϵ-close estimate of v_1 for $\epsilon < 1$.

Our second algorithm which we call the *delayed momentum streaming power method* (DMStream) shown in Algorithm 2 is a streaming companion to DMPower. Theorem 16 due to De Sa et. al [42] provides a convergence guarantee on a momentum-based streaming algorithm referred to as Mini-Batch Power+M. In particular, significant acceleration is experienced if $2\sqrt{\beta} \in [\lambda_2, \lambda_1)$, but as in the case of Power+M, selection of such a β at run-time is an impractical ask. The full convergence guarantee of Mini-Batch Power+M is provided in Theorem 16. Similar to DMPower, DMStream approximates a converging momentum coefficient in a pre-momentum phase and then uses that coefficient to accelerate convergence in a secondary momentum phase.

DMStream superficially resembles DMPower, instead using unbiased estimates for its up-

Algorithm 2 Streaming Delayed Momentum Power Method (DMStream)

Require: Streaming samples $x_1, x_2, \ldots, x_l \in \mathbb{R}^d$, batch size n, unit $q_0 \in \mathbb{R}^d$, pre-momentum phase iterations J, momentum phase iterations K, unit $w_0 \in \mathbb{R}^d$

for $j = 1, 2, \ldots, J$ **do**
  Generate unbiased estimate $\widehat{A}_j = \frac{1}{n}\sum_{i=(j-1)n+1}^{jn} x_i x_i^\top$
  $q_j \leftarrow \widehat{A}_j q_{j-1}$
  $q_j \leftarrow q_j / ||q_j||$
  $\nu_j \leftarrow q_j^\top \widehat{A}_j q_j$ ▷ Rayleigh Quotient estimate of λ_1
  $P \leftarrow \nu_j q_j q_j^\top$
  $w_j \leftarrow (\widehat{A}_j - P) w_{j-1}$ ▷ Inexact deflation
  $w_j \leftarrow w_j / ||w_j||$
  $\mu_j \leftarrow w_j^\top \widehat{A}_j w_j$ ▷ Rayleigh Quotient estimate of λ_2
end for
$\hat{\lambda}_2 \leftarrow \mu_J$
$\beta \leftarrow \mu_J^2/4$ ▷ Approximated optimal momentum coefficient
$q_1 \leftarrow q_J$ ▷ Current estimate of v_1
$q_0 \leftarrow \mathbf{0}$
for $k = 1, 2, \ldots, K$ **do** ▷ while $\|q_k - q_{k-1}\| > \epsilon$
  Generate unbiased estimate $\widehat{A}_k = \frac{1}{n}\sum_{i=(k-1)n+1}^{kn} x_i x_i^\top$
  $q_{k+1} \leftarrow \widehat{A}_k q_k - \beta q_{k-1}$ ▷ Momentum update
  $q_{k+1} \leftarrow q_{k+1} / ||q_{k+1}||$
  $\nu_k \leftarrow q_{k+1}^\top \widehat{A}_k q_{k+1}$
end forreturn q_K, ν_K

dates. However, due to the noise introduced by estimation error of A by $\widehat{A}_t$ in conjunction with the noise introduced by our imperfect estimations of v_1 by q_t, the inexact deflation step is more challenging to analyze and results in a distinct guarantee, which we provide in Theorem 5.

Complexity. Each matrix-vector multiplication cost $\mathcal{O}\left(d^2\right)$ with three such multiplications in every round of the pre-momentum phase (power iteration, Hotelling iteration, and an inexact Rayleigh quotient). Akin to DMPower, we justify these increased FLOPS by noting that the pre-momentum phase will terminate in a finite number of rounds, which is shown in Theorem 5. We empirically observe that even a rough selection of β provides us with noticeable acceleration, resulting in lower iteration complexity overall, and thus a decreased total runtime when compared

to conventional streaming PCA as in Algorithm 4.

2.4 Convergence Analysis

We now provide our two major convergence theorems. We adopt the same notations as in Algorithm 1 and Algorithm 2. Both algorithms are divided into J steps of a pre-momentum phase and K steps of a momentum phase. As a reminder, μ_j is the Rayleigh quotient approximation of λ_2 at step j and $\beta := \widehat{\lambda}^2/4 = \mu_J^2/4$. For notational convenience, we let $\theta_0 := \arccos |q_0^\top v_1|$.

2.4.1 Delayed Momentum Power Method (DMPower)

Theorem 4 (Convergence of DMPower)**.** *Let J represent the number of steps in the pre-momentum phase and K the number of steps in the momentum phase as in Algorithm 1. Let $\epsilon < 1$ represent the desired error threshold of our v_1 estimates, i.e., $\sin^2\theta(q_t, v_1) < \epsilon$ and $\rho < \min\{1/2, \sqrt{\frac{\lambda_1-\lambda_2}{\lambda_2-\lambda_d}}\}$ represent the desired error threshold of our λ_2 estimates, i.e., $|\mu_k - \lambda_2| < \rho$. Further fix $\tau > 1$ and $\delta = \min\{\rho, \frac{1}{\tau\sqrt{d}}\}$. Then after*

$$J = \mathcal{O}\Big(\frac{1}{\lambda_1 - \lambda_2}\log\frac{\tan^2\theta_0}{\delta(\lambda_2 - \lambda_3)} + \frac{\lambda_2}{\lambda_2 - \lambda_3}\log\frac{d\tau}{\rho}\Big), \tag{2.6}$$

$$K = \mathcal{O}\Big(\frac{\beta}{\sqrt{\lambda_1^2 - 4\beta^2}}\log\frac{1}{\epsilon}\Big) \tag{2.7}$$

pre-momentum and momentum steps, respectively, where $\beta = \widehat{\lambda}_2^2/4 = \mu_J^2/4$, with all but $\tau^{-\Omega(1)} + e^{-\Omega(d)}$ probability, DMPower *outputs a vector q_K with $\sin^2\theta(q_K, v_1) < \epsilon$.*

Remark 1: Our step count for Power+M convergence implicitly assumes $\lambda_2 \le 2\sqrt{\beta_J}$, i.e., our β approximation lives on the right of $\lambda_2^2/4$. It is possible for $2\sqrt{\beta_J} \le \lambda_2$, in which case we will still

appreciate the effects of acceleration (see Theorem 23).

Remark 2: We require $J = \mathcal{O}\big(\frac{1}{\lambda_1-\lambda_2}\log\frac{\tan^2\theta_0}{\delta(\lambda_2-\lambda_3)} + \frac{\lambda_2}{\lambda_2-\lambda_3}\log\frac{d\tau}{\rho}\big)$ to achieve ρ-accuracy and a further $K = \mathcal{O}\big(\frac{\beta_J}{\sqrt{\lambda_1^2-4\nu_J^2}}\log\frac{1}{\epsilon}\big)$ steps to achieve ϵ-accuracy. In the momentum phase, we absorb q_J as our initial vector, which has already made convergent progress towards v_1. In practice, we will not need all $J+K$ iterations.

Proof Sketch. We divide DMPower by its pre-momentum and momentum phases. The full proof is deferred to Section 2.7.3.

Pre-momentum phase: We regard the inexact deflation step as an exact deflation experiencing a perturbation every round. In Lemma 12, we show that this noise decays at every step and after $J_1 = \mathcal{O}\big(\frac{1}{\lambda_1-\lambda_2}\log\frac{\tan^2\theta_0}{\delta(\lambda_2-\lambda_3)}\big)$ steps, we achieve the Hardt-Price bounds [73] necessary for convergence of a noisy power method. The convergence rate for noisy power methods [73, Corollary 1.1] indicates that after an additional $J_2 = \mathcal{O}\big(\frac{\lambda_2}{\lambda_2-\lambda_3}\log\frac{d\tau}{\rho}\big)$ steps, we will reach our desired ρ-accuracy, so in total, we need $J = J_1 + J_2$ iterations to complete the pre-momentum phase. We set $\beta = \frac{\mu_J^2}{4}$ and proceed to the momentum phase.

Momentum phase: Now that our momentum $\widehat{\lambda}_2 = \mu_J^2/4$ coefficient is within the interval $[\lambda_2^2/4, \lambda_1^2/4)$, we may invoke Sa et al.'s Power+M convergence Theorem 4, which states that after $K = \mathcal{O}\Big(\frac{\beta_J}{\sqrt{\lambda_1^2-4\beta_J^2}}\log\big(\frac{1}{\epsilon}\big)\Big)$ steps of iteration on q_J (which we take to be our initial vector for Power+M) steps, we will have that $\sin^2\theta(q_{J+K}, v_1) < \epsilon$.

2.4.2 Delayed Momentum Streaming Power Method (DMStream)

Theorem 5 (Convergence of DMStream)**.** *Let* $\Sigma = \mathbb{E}[(\widehat{A}_t - A)\otimes(\widehat{A}_t - A)]$, *where* $\widehat{A}_j = \frac{1}{n}\sum_{i=1}^n x_i x_i^\top$ *represents any unbiased estimate of* A *in DMStream with fixed batch size* n *and*

$\otimes$ *denotes the Kronecker product. Assume we initialize with unit* $q_0 \in \mathbb{R}^d$ *where* $d \gg 0$ *and* $|v_1^\top q_0| \geq 1/2$. *Let* $\theta_0 = \arccos |q_0^\top v_1|$. *For any* $\delta < 1$, $\epsilon < 1$, *suppose*

$$||\Sigma|| \leq \frac{(\lambda_1^2 - 4\beta)\delta\epsilon}{256\sqrt{d}J} = \frac{(\lambda_1^2 - 4\beta)^{3/2}\delta\epsilon}{256\sqrt{d}\sqrt{\beta}} \log^{-1}\Big(\frac{32}{\delta\epsilon}\Big), \tag{2.8}$$

where J *is the total number of pre-momentum steps we have fixed at runtime. Furthermore, we let* $\rho < \min\{1/2, \sqrt{\frac{\lambda_1 - \lambda_2}{\lambda_2 - \lambda_d}}\}$ *represent the error threshold of our* λ_2 *estimates, i.e.,* $|\mu_k - \lambda_2| < \rho$. *Lastly, fix* $\tau > 1$ *and* $\delta = \min\{\rho, \frac{1}{\tau\sqrt{d}}\}$. *If our batch size* n *is chosen such that*

$$\frac{n}{\log^4 n} = \mathcal{O}\Big(\frac{1/\gamma^2 \log d}{(\lambda_2 - \lambda_3)^2 d}\Big). \tag{2.9}$$

where $\gamma = \frac{\rho(\lambda_2 - \lambda_3)}{10\tau\sqrt{d}}$, *then after*

$$J = \mathcal{O}\left(\frac{1}{\lambda_1 - \lambda_2} \log\left(\frac{\tan^2\theta_0 \tau\sqrt{d}}{\rho(\lambda_2 - \lambda_3)}\right) + \frac{\lambda_2}{\lambda_2 - \lambda_3} \log\frac{d\tau}{\rho}\right), \tag{2.10}$$

$$K = \frac{\sqrt{\beta}}{\sqrt{\lambda_1 - 4\beta}} \log\Big(\frac{32}{\delta\epsilon}\Big) \tag{2.11}$$

pre-momentum steps and momentum steps, respectively, with $(1 - \frac{1}{n^2})(1 - 2\delta)(1 - \tau^{-\Omega(1)} + e^{-\Omega(d)})$ *probability DMStream outputs a vector* q_K *such that* $\sin^2 \angle(q_K, v_1) < \epsilon$.

Remark 1: Both phases have a probabilistic guarantee, whereas in DMPower, the momentum phase was deterministically guaranteed. The probabilistic parameter for the pre-momentum phase τ while it is δ for the momentum phase.

Remark 2: We assume a variance condition on our estimates in equation 2.8. While there are many sophisticated methods designed to reduce variance [42, 66, 74] via introduction of step

sizes and anchor iterates, we do not explore these options in this paper. However, a simple strategy for reducing variance is to increase batch size n, since we have the relation $||\Sigma|| \leq \frac{\sigma^2}{s}$, where σ^2 is the variance of a single random sample.

Remark 3: The total sample complexity is $n(J + K)$.

Proof Sketch. We divide DMStream by its pre-momentum and momentum phases. The proof is deferred to Section 2.7.4. *Pre-momentum phase:* There are two sources of noise in every round of this phase: H_t the estimation error associated to $(A - \widehat{A}_j)w_j$, and G_t the estimation error associated to $(\lambda_1 v_1 v_1^\top - \nu_j q_j q_j^\top)w_j$. Proposition 18 shows us how to control $||H_t||$ through batch size and Proposition 21 details $||G_t||$. We require $J_1 := \mathcal{O}\left(\frac{1}{\lambda_1 - \lambda_2} \log\left(\frac{\tan^2\theta_0 \tau\sqrt{d}}{\rho(\lambda_2 - \lambda_3)}\right)\right)$ to achieve the Hardt-Price bounds and a further $J_2 := \mathcal{O}\big(\frac{\lambda_2}{\lambda_2 - \lambda_3} \log\frac{d\tau}{\rho}\big)$ to acquire the appropriate β, for a total of $J = J_1 + J_2$ pre-momentum rounds.

Momentum phase: Now that our momentum coefficient $\beta = \widehat{\lambda}_2 = \mu_J^2/4$ coefficient is within the interval $[\lambda_2^2/4, \lambda_1^2/4)$, we may invoke Sa et al.'s streaming power convergence Theorem 16, to conclude that we need $K = \frac{\sqrt{\beta}}{\sqrt{\lambda_1 - 4\beta}} \log\big(\frac{32}{\delta\epsilon}\big)$ steps to complete the momentum phase.

2.4.3 Precision of Inexact Deflation

In this section we refer to variables and notations as listed in Algorithm 1 and Algorithm 2. We let $\Delta_{1,2} := |\lambda_1 - \lambda_2|$. Although one sets the number of iterations J for the pre-momentum phase at run-time, we are also interested in the accuracy of our λ_2 estimates, that is, $|\lambda_2 - \mu_j|$, since this will determine how quickly our momentum phase converges. In Theorem 4 and Theorem 5, we discuss how many pre-momentum iterations J are needed to ensure $|\mu_j - \mu_{j-1}| \leq \rho$, where ρ is an error threshold which controls the accuracy of our final estimate $\widehat{\lambda}_2$. As such, for the

remainder of this section we will look at a modification of DMPower and DMStream where ρ is provided as a hyperparameter and how it affects our overall convergence.

Effects of inaccurate approximation of λ_2 In Theorem 4 and Theorem 5, the overall convergence rate is dependent on our pre-momentum phase error $|\lambda_2 - \widehat{\lambda}_2| < \rho$, where $\widehat{\lambda}_2 = \mu_J$. Ultimately, we need our approximated momentum coefficient $\beta = \widehat{\lambda}_2^2/4 \in [\lambda_2^2/4, \lambda_1^2/4)$. In fact, even if $\beta < \lambda_2^2/4$, the momentum phase will still converge, and will still experience similar momentum effects as long as $|\beta - \lambda_2^2/4| < |\lambda_1^2/4 - \lambda_2^2/4|$, which is a generalization of Theorem 2, provided in Theorem 23. We establish a proposition suggesting how accurately $\widehat{\lambda}_2$ must approximate λ_2:

Proposition 6. *The momentum phase of DMPower set with momentum coefficient* $\beta = \mu_J^2/4$ *converges if and only if* $|\lambda_2 - \widehat{\lambda}_2| \leq \Delta_{1,2}$.

The proof is deferred to Section 2.7.6. In lieu of fixed number of iterations for the pre-momentum phase, one may instead choose to exit the first for-loop if $|\mu_j - \mu_{j-1}| < \rho$, where we have now adopted ρ as a hyperparameter, which is a far less aggressive option than the guesswork required with randomly selecting a convergent β as in Power+M. We assume this termination condition for the remainder of our discussion.

By Theorem 4, the $\mu_j \to \lambda_2$, so we argue it is fair to assume $|\mu_j - \mu_{j-1}| \approx |\mu_j - \lambda_2|$. In which case, by the triangle inequality and Proposition 2.4.3, we have that $|\mu_j - \mu_{j-1}| < \rho$ if and only if $\rho \lesssim \frac{1}{2}\Delta_{1,2}$. Loosening ρ beyond this bound does not necessarily result in divergence, however, which we experimentally observe and discuss in the next section. Current state-of-the-art bounds do not say anything meaningful about how quickly we can expect to converge/diverge outside of this bound. According to Theorem 4, the looser we set ρ, the fewer iterations we can expect to run in the pre-momentum phase. However, the tighter we set ρ, the closer our β

approaches $\lambda_2^2/4$, which is the optimal assignment for the momentum phase.

Practical selection of ρ Instead of setting ρ to an exceedingly small value close to machine precision, we experimentally demonstrate in Figure 2.3 that DMPower is successful in the setting where $\Delta_{1,2} = \Delta_{2,3}$ for a variety of ρ selections. In Figure 2.3 we have set $\rho = \sqrt[k]{\epsilon}$ for $k = 1, 2, 3, 4$ to demonstrate flexibility, but they are independent precision bounds; ρ depends only on $\Delta_{1,2}$ by Proposition 6, not on ϵ. We further observe that our DMPower outperforms the vanilla power method at nearly all error thresholds, and converges at a rate similar to optimal Power+M for tighter eigengaps according to Figure 2.3. In Figure 2.2 and Table 2.2, we demonstrate that setting ρ looser causes us to non-negligibly lose precision in our approximation of λ_2. Setting $\rho = \epsilon$ tighter increases our accuracy, but our iteration complexity worsens by a noticeable amount, especially in the medium eigengap setting.

If one is certain about lower bounds α_1, α_2 of $\Delta_{1,2}$ and $\Delta_{2,3}$, respectively, then we have the following result for DMPower:

Proposition 7. *Assume $\alpha_1 \leq \Delta_{1,2}$ and $\alpha_2 \leq \Delta_{2,3}$. Fix $\rho < \min\{1/2, \sqrt{\alpha_1}\}$, $\tau > 1$ and let $\theta_0 = \arccos|q_0^\top v_1|$, $\delta = \min\{\rho, \frac{1}{\tau\sqrt{d}}\}$. Then after*

$$J = \mathcal{O}\Big(\frac{1}{\alpha_1}\log\frac{\tan^2\theta_0}{\delta\alpha_2} + \frac{1}{\alpha_2}\log\frac{d\tau}{\rho}\Big) \tag{2.12}$$

pre-momentum phase steps, we output a vector w_J such that if $\mu_J = w_J A w_J^\top$, then $|\lambda_2 - \mu_J| < \rho^2$. Since $\rho^2 < \alpha_1$, our momentum phase will converge with all but $\tau^{-\Omega(1)} + e^{-\Omega(d)}$ probability.

Proof. The size of J along with the probabilistic guarantee is a simple corollary of Theorem 4 in conjunction with Lemma 9 to relate $\sin(w_J, v_2)$ to $|\lambda_2 - \mu_J|$. □

It is important to note the usefulness and practicality of Proposition 7: in other momentum-based methods, having a lower bound on $\Delta_{1,2}$ was not sufficient to guarantee convergence – one would still need the actual location of $[\lambda_2, \lambda_1)$ over [0,1].

2.5 Experiments

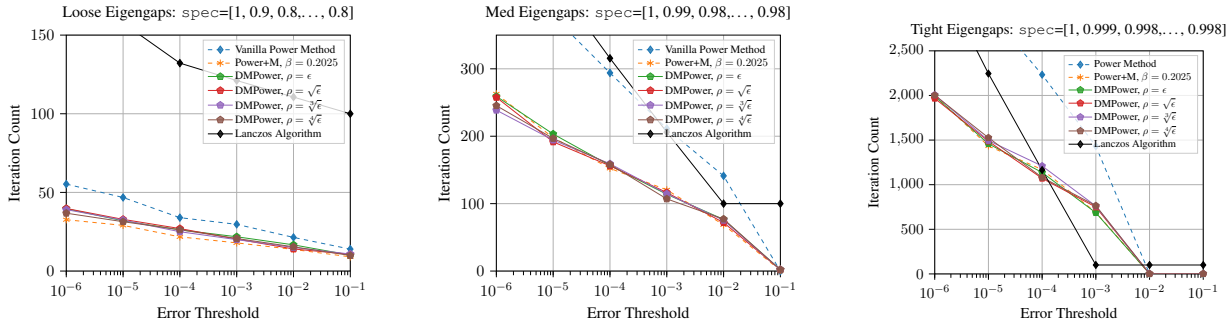
In this section, we discuss the set of experiments we conducted to measure the performance of DMPower and DMStream against a variety of common baseline algorithms. All the experiments were run on an Intel(R) Xeon(R) E5-1650 v4 machine with 32GiB RAM and Linux OS. The algorithms were implemented in Python using the Numpy and SciPy libraries. We refer the reader Section 2.7 for further experimental design and analyses, including the construction of the synthetic matrices and runtime (wall-time) comparisons.

2.5.1 DMPower Experiments

Experimental Setup. In these experiments, we compare DMPower against the vanilla power method, Power+M with optimal assignment of β, and the Lanczos algorithm. In each experiment, we generate a random symmetric PSD $A \in \mathbb{R}^{100\times 100}$ with a fixed spectrum and conduct PCA using the methods listed above. In Figures 2.1 and 2.3, we record the number of iterations that are required to achieve the desired error-tolerance ϵ between our dominant eigenvalue approximates and then take an average over 1000 runs, each time generating a new symmetric PSD A with a specified spectrum. The initial vector q_0 is uniformly set across all methods for every run. Since we know the spectrum of these synthetic matrices, we can initialize Power+M to run with $\beta = \lambda_2^2/4$, the optimal momentum coefficient. In Figure 2.2, we measure the accuracy of

Rayleigh quotient estimates to the Simultaneous Power Iteration rather than convergence speeds.

Iteration Complexity Results. As shown in Figure 2.3, DMPower outperforms the vanilla power method at nearly all ϵ thresholds, across a variety of ρ settings. We set $\rho = \epsilon^{1/k}$ for $k = 1, 2, 3, 4$, but we must stress that ρ and ϵ are independent precision bounds; ρ depends on $\Delta_{1,2}$ by Proposition 6. Furthermore, at medium and tight eigengaps, observe DMPower performs nearly as well as optimal Power+M. When factoring in tridiagonalization iterations, DMPower outperforms the Lanczos algorithm, especially at tighter precisions (the Lanczos is well-known to suffer from numerical instability without corrective measures such as re-orthogonalization [57]). See Table 2.3 for a full set of raw data. We also measured the proportion of pre-momentum versus momentum phase iterations in Table 2.4, noting that for a variety of ρ settings, most iterations of DMPower are spent in the momentum phase.



(a) $\Delta_{1,2} = \Delta_{2,3} = 0.1$ (b) $\Delta_{1,2} = \Delta_{2,3} = 0.01$ (c) $\Delta_{1,2} = \Delta_{2,3} = 0.001$

Figure 2.3: **Iteration Complexity Comparisons**. We compare DMPower (with different ρ settings) against the vanilla power method, Power+M with optimal β, and the Lanczos algorithm. The X-axis corresponds to ϵ, and the Y-axis measures performance by iteration count. Our algorithms demonstrate consistently favorable performance against the vanilla power method baseline and match optimal convergence speeds established by Power+M at medium and tight eigengap settings. *Note 1*: For the Lanczos algorithm, we start by taking 100 tri-diagonalization iterations, which is the recommended number for numerical stability. *Note 2:* We vary ρ according to ϵ for convenience, but they are independent precision thresholds. As stated in Proposition 6, ρ is dependent on $\Delta_{1,2}$.

Wall-time Performance. At loose and medium eigengaps, Power+M (optimal) registers

the lowest wall-time, owing to fewer matrix-vector computations. We remind the reader that this setting is only a baseline and not practically achievable. Of particular note is that at nearly all settings, DMPower runs noticeably faster than the Lanczos algorithm. We refer the reader to Table 2.5 for full data.

2.5.2 DMStream Experiments

Experimental Setup. We used a 50,000 sample subset of the MNIST dataset [75], which is represented as a matrix of size 50000×784. The dataset was first pre-processed by centering and dividing the entire matrix by $\sigma\sqrt{784}$. We compared DMStream against Oja's algorithm [76] with varying step sizes, stochastic power iteration (Algorithm 4), and Mini-Batch Power+M set with optimal $\beta = \lambda_2^2/4$, where λ_2 is the second eigenvalue of the (processed) covariance matrix of MNIST. We measured performance by a commonly-used metric $\log_{10}\left(1 - \frac{||X^\top q_K||}{||X^\top v_1||}\right)$. We tested over a variety of batch sizes with $\rho = 0.1$ for DMStream. For each batch size, we ran 50 iterations and averaged the results over 10 runs.

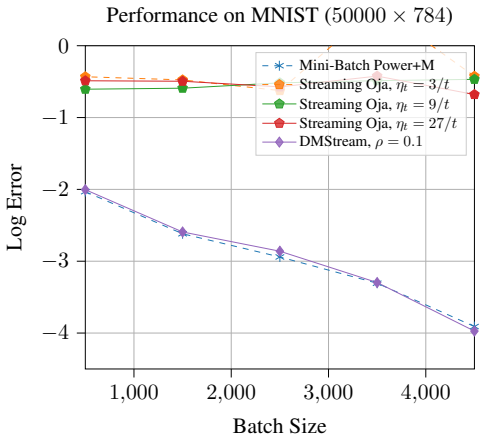


Figure 2.4: **Performance by batch size**. In this experiment we measure the performance of DMStream versus Oja's algorithm (with various step sizes) and Mini-Batch Power+M (optimal $\beta = \lambda_2^2/4$). We measure performance by the log error $\log_{10}(1 - \frac{||X^\top q_K||}{||X^\top v_1||})$. DMStream exhibits a consistent increase in accuracy as batch-size is increased and mimics the performance of optimal Mini-Batch Power+M.

Results. Our algorithm performs significantly better than Oja's algorithm for several step sizes and is as accurate as Mini-Batch Power+M initialized with optimal $\beta = \lambda_2^2/4$. As Table

2.6 demonstrates, our accuracy eventually ceases to improve for fixed batch size, but Figure 2.4 indicates noticeable improvement with increasing batch size, owing to reduced variance of our unbiased estimates $\widehat{A}_t$.

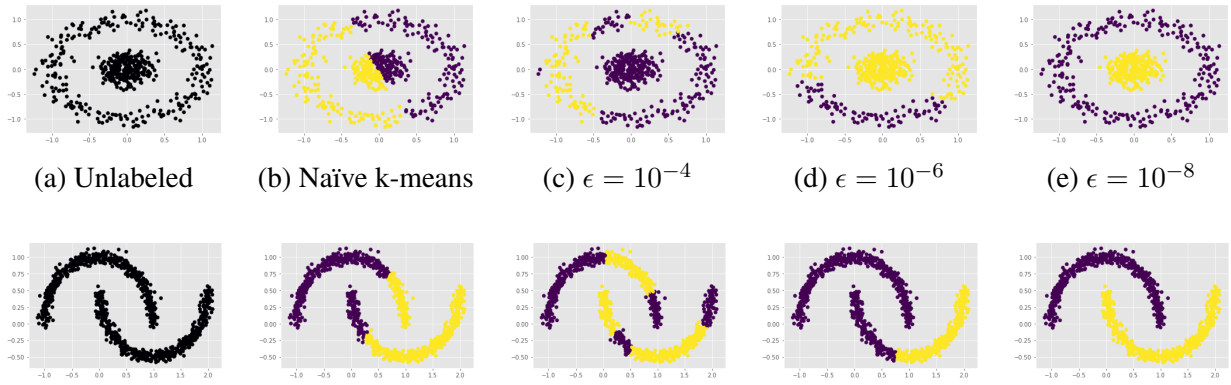
2.5.3 Application: Spectral Clustering

Overview. Clustering is the unsupervised learning task of dividing a collection of data points into distinct groups or "clusters." The k-means algorithm [77] is a popular clustering method which has found ubiquitous use in machine learning, including social network analysis [78], image processing [79], and other data mining tasks.

However, k-means is limited in effectiveness when applied to nonlinear data. Spectral clustering is an extension of k-means used to properly separate nonlinear data. Whereas k-means is applied directly on the data points $\{x_i\}_{i=1}^n$, spectral clustering first begins with a symmetric affinity matrix $A_{ij} = s(x_i, x_j)$ where s is a similarity function which could be Euclidean distance, for example. We define the diagonal matrix D where $D_{ii} = \sum_{j=1}^n A_{ij}$ and then form the normalized affinity matrix $W = D^{-1}A$. Spectral clustering then computes the top k components of W. These eigenvectors are supplied to the k-means algorithm to obtain the final separation result.

The power iteration may be used to find the eigenvectors of W, which is referred to as power iteration clustering (PIC). The deflation-based PIC algorithm [50] we use for our experiments is provided in Section 2.7.8. In this section, we compare variants of the power iteration in carrying out PIC and demonstrate that DMPower is capable of faster eigenvector computation and more accurate data separation when compared against the vanilla power iteration.

Experimental Setup. We used two popular toy datasets for clustering: half-moons and



(a) Unlabeled (b) Naïve k-means (c) $\epsilon = 10^{-4}$ (d) $\epsilon = 10^{-6}$ (e) $\epsilon = 10^{-8}$

(f) Unlabeled (g) Naïve k-means (h) $\epsilon = 10^{-4}$ (i) $\epsilon = 10^{-6}$ (j) $\epsilon = 10^{-8}$

Figure 2.5: **Performance of DMPM on Spectral Clustering.** We depict the performance of DMPM ($\rho = \sqrt[3]{\epsilon}$) on dividing a collection of data points into their natural, geometrically-partitioned clusters. The first series (a-e) is the concentric circles dataset while the second series (f-j) is the half-moons dataset. In both series, the first image depicts the unlabeled arrangement, the second depicts a naïve application of k-means without spectral clustering, while the last three images depict the performance of k-means assisted by spectral clustering over progressively tighter ϵ error thresholds (thus, requiring more accurate affinity matrix eigenvector approximations). As ϵ grows smaller, the data separation improves, eventually achieving perfect classification by $\epsilon = 10^{-8}$.

concentric circles. The half-moons dataset was generated with 500 samples while the concentric circles set was generated using 1000 samples. We compared DMPower (with various ρ settings) against the vanilla power method and Power+M with optimal assignment of β. We follow the deflation-based approach of spectral clustering, which is outlined in Algorithm 5. Each method was used to recover the top two eigenvectors and eigenvalues of the normalized affinity matrix W of each dataset (our similarity function was pairwise ℓ_2 distance).

We use a practical implementation of the DMPower Algorithm 1: we exit the pre-momentum phase when $||w_{j+1} - w_j|| \leq \rho$, where ρ is a function of ϵ, and exit the momentum phase when $||q_{k-1} - q_j|| \leq \epsilon$. Similarly, for the vanilla power iteration Algorithm 3 and the alternative Power+M update in equation 2.4, we end the procedure once $||q_{k+1} - q_k|| \leq \epsilon$. Therefore, termination of these algorithms is governed by the closeness of the approximates.

Results. DMPower, under all ρ settings, and for most error thresholds ϵ requires fewer iterations to recover the top two eigenvectors of the affinity matrices than the vanilla power method, see Table 2.6. Furthermore, DMPower with $\rho = \sqrt{\epsilon}, \sqrt[3]{\epsilon}$ performs similarly in both iteration complexity and accuracy when compared against Power+M with optimal $\beta = \lambda_2^2/4$ as reported in to Tables 2.7 and 2.8. Figure 2.5 depicts a progression of separation over the datasets, when DMPower ($\rho = \sqrt[3]{\epsilon}$) is used.

2.6 Conclusion

In summary, this paper introduces a new scheme for accelerating the vanilla and streaming power methods. The realization of this scheme is the delayed momentum power method (DMPower) and its streaming companion the delayed streaming momentum method (DMStream). DMPower is experimentally shown to outperform the vanilla power iteration and achieves iteration complexity similar to an existing accelerated method Power+M initialized with optimal hyperparameters. Empirically, it also outperforms the state-of-the-art Lanczos algorithm in both wall-time and iteration complexity under several regimes. DMStream is shown to vastly outperform all variations of Oja's algorithm and register close-to-optimal error when compared against the Mini-Batch Power+M initialized with an optimal momentum coefficient. We provide convergence guarantees for DMPower and DMStream using a mixture of perturbation theory and classical power iteration bounds. While other momentum-based methods rely on unrealistic spectral knowledge for acceleration, DMPower and DMStream are both practical and fast.

2.7 Supplemental Materials

2.7.1 Preliminaries and Facts

2.7.1.1 Vanilla Power Method

Unless noted otherwise, $||\cdot||$ refers to the 2-norm for vectors and the induced 2-norm for matrices. We recall the vanilla power method algorithm.

Algorithm 3 Vanilla Power Method

Require: $A \in \mathbb{R}^{d\times d}$ diagonalizable, initial vector $q_0 \in \mathbb{R}^d$, error threshold ϵ
Ensure: ϵ-accurate approximation of v_1, λ_1
1: **for** $k = 1, 2, \ldots$ **do** ▷ While $||q_k - v_1|| > \epsilon$
2: $q_k \leftarrow \frac{Aq_{k-1}}{\|Aq_{k-1}\|}$
3: $\nu_k \leftarrow q_k^\top A q_k$ ▷ Rayleigh Quotient
4: **end forreturn** q_k, ν_k

Although the termination condition relies on an ℓ_2 distance from v_1, this is often replaced with a sine squared error condition, that is, we exit the loop once $\sin^2(\theta_k) \triangleq 1 - (q_k^\top v_1)^2 > \epsilon$. We make use of both variations throughout this paper.

2.7.1.2 Power Iteration Bounds

The setting for the next two lemmas is the following: we let $A \in \mathbb{R}^{d\times d}$ be symmetric with spectrum $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \cdots \geq |\lambda_n|$ and associated unit eigenvectors $v_1, v_2, \ldots, v_n$. Note that we do not need the presence of the second eigengap $|\lambda_2| > |\lambda_3|$ for these classical results. We will now establish several well-known inequalities regarding the accuracy of the vanilla power iteration.

Lemma 8 ([80], p. 194)**.** *Let $q_0 \in \mathbb{R}^d$ such that $|{q_0}^\top v_1| \neq 0$. We may write $q_0 = \sum_{i=1}^d c_i v_i$ since A is diagonalizable (being symmetric). Let $C = \left(\sum_{i=1}^d \left(\frac{c_i}{c_1}\right)^2\right)^{1/2}$. We have that*

$$||\tilde{q_k} - v_1|| \leq C\left|\frac{\lambda_2}{\lambda_1}\right|^k \qquad k \geq 1 \tag{2.13}$$

where

$$\tilde{q_k} = \frac{q_k}{||A^k q_0||}\alpha_1 {\lambda_1}^k = v_1 + \sum_{i=2}^{d} \frac{c_i}{c_1}\left(\frac{\lambda_i}{\lambda_1}\right)^k v_i, \qquad k = 1, 2, \ldots \tag{2.14}$$

Remark. Since $\tilde{q_k}$ is nothing more than a scaled version of q_k convenient for analysis, as an abuse of notation, where 2-norm inequalites are invoked involving q_k and v_1, we assume we are working with $\tilde{q_k}$.

Lemma 9 ([58], p. 451)**.** *Let $q_0 \in \mathbb{R}^n$ such that $|{q_0}^\top v_1| \neq 0$. Let $q^k = \frac{A^k q_0}{||A^k q_0||}$ and $\nu_k = {q_k}^\top A q_k$, i.e., the basic power iteration approximation of v_1 and λ_1 after k steps. Define $\theta_k \in [0, \pi/2]$ by $\cos(\theta_k) = |{q_k}^\top v_1|$. For $k = 0, 1, 2, \ldots$, we have*

$$|\sin(\theta_k)| \leq \tan(\theta_0)\left|\frac{\lambda_2}{\lambda_1}\right|^k \tag{2.15}$$

$$|\lambda_1 - \nu_k| \leq \max_{2 \leq i \leq d} |\lambda_1 - \lambda_i| \tan(\theta_0)^2 \left|\frac{\lambda_2}{\lambda_1}\right|^{2k} \tag{2.16}$$

We now prove a more general, well-known fact: the Rayleigh quotient is a quadratically-accurate estimate when compared to the sine squared error.

Lemma 10. *Let $w_k \to v_j$ for $\{w_k\}_{i=1}^\infty$ a sequence of unit vectors and v_j a unit eigenvector of A associated to eigenvalue λ_j. Let η_k be the Rayleigh quotient of w_k and $\sin^2(\gamma_k) = 1 - (w_k^\top v_j)^2$.*

Then

$$|\eta_k - \lambda_i| = \mathcal{O}\big(\sin^2(\gamma_k)\big) \tag{2.17}$$

Proof. Express $w_k = \sum_{i=1}^{d} c_i v_i$, i.e., as a linear combination in the eigenbasis of A. We have then that

$$\eta_k - \lambda_j = w_k^\top A w_k - \lambda_j = \frac{\sum_{i=1}^{d} c_i^2 \lambda_i}{\sum_i^d c_i^2} - \lambda_j = \frac{\sum_{i=1}^{d} (\lambda_i - \lambda_j) c_i^2}{\sum_{i=1}^{d} c_i^2} \leq \max_{i \neq j} |\lambda_j - \lambda_i| \sum_{i \neq j}^{d} c_i^2 \tag{2.18}$$

Since $\sum_{i=1}^{d} c_i^2 = 1$ (w_k is unit), we have that $\sum\limits_{\substack{i=1 \\ i \neq j}}^{d} c_i^2 = 1 - c_j^2 = 1 - (w_k^\top v_j)^2 = \sin^2(\gamma_j)$. Therefore,

$$|\eta_k - \lambda_j| \leq \max_{i \neq j} |\lambda_j - \lambda_i| \sin^2(\gamma_k) \tag{2.19}$$

proving our claim. □

2.7.2 Noisy Deflation

The following results will help us determine how many steps of inexact deflation we need to complete before we achieve small enough noise at each iteration to permit convergence. As usual, we let ν_k and q_k reflect our eigenvalue and eigenvector approximation at step k of a vanilla power iteration with matrix A. We first recall the formulation of a *noisy power method* (NPM). We consider an alternative update step of Algorithm 3:

$$q_k = \frac{A q_{k-1} + G_{k-1}}{||A q_{k-1} + G_{k-1}||} \tag{2.20}$$

Here, G_{k-1} is noise or a perturbation added at each round. The following is a powerful result on the conditions under which NPM can successfully converge:

Corollary 11 (Noisy Power Method Convergence [73])**.** *Let $k \leq p$. Let $U \in \mathbb{R}^{d\times k}$ represent the top k singular vectors of $B \in \mathbb{R}^{d\times d}$ and let $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_d$ denote its singular values. Suppose X_0 is an orthonormal basis of a random p-dimensional subspace. Further, suppose at every step of NPM we have*

$$5||G_\ell|| \leq \epsilon(\lambda_k - \lambda_{k-1}) \;\; \textit{and} \;\; 5||U^\top G_\ell|| \leq (\lambda_k - \lambda_{k-1})\frac{\sqrt{p}-\sqrt{k-1}}{\tau\sqrt{d}} \tag{2.21}$$

for some fixed parameter τ and $\epsilon < 1/2$. Then with all but $\tau^{\Omega(p+1-k)} + e^{-\Omega(d)}$ probability, there exists an $L = \mathcal{O}\big(\frac{\lambda_k}{\lambda_k-\lambda_{k-1}} \log \frac{d\tau}{\epsilon}\big)$ so that after L steps we have $||(I - X_L X_L^\top)U|| \leq \epsilon$.

In the context of Theorem 4, we have that $B = A - \lambda_1 v_1 v_1^\top$, $X_0 = w_0$, $U = v_2$, and G_ℓ is the error $||\lambda_1 v_1 v_1^\top - \nu_\ell q_\ell q_\ell^\top||$, that is, the "inexactness" of our deflation, and d is the dimension. We also note that in relation to A, the deflated matrix B has spectrum $\lambda_2 > \lambda_3 \geq \cdots \lambda_{n-1} \geq \lambda_n = 0$. In this setting, we have that $||(I - X_L X_L^\top)U|| = ||(I - w_L w_L^\top)v_2)|| = \sqrt{1-(w_L v_2^\top)^2}$, i.e., the sine error between v_2 and w_L. We will now provide an upper bound on $||G_\ell||$ and show it decays with every round. For notational convenience, we denote $\theta_0 := \arccos|q_0^\top v_1|$.

Lemma 12. *If we express $A - \nu_k q_k q_k^\top$ as $A - \lambda_1 v_1 v_1^\top + G_k$, where $G_k = \lambda_1 v_1 v_1^\top - \nu_\ell q_\ell q_\ell^\top$ is a perturbation reflecting the error of our eigenpair approximation, then $||G_k|| = \mathcal{O}(\tan^2\theta_0 |\frac{\lambda_2}{\lambda_1}|^k)$.*

Proof. Let $q_k = v_1 + \xi_k$ and $\nu_k = \lambda_1 + \gamma_k$, that is, ξ_k and γ_k reflect the perturbations (error)

associated with our power iterates. We may express our inexact deflation matrix as follows:

$$A - \nu_k q_k {q_k}^\top = A - (\lambda_1 + \gamma_k)(v_1 + \xi_k)(v_1 + \xi_k)^\top. \tag{2.22}$$

Expanding, we have that our deviation from $A - \lambda_1 v_1 v_1^\top$ is expressible as

$$G_k = \lambda_1(v_1\xi_k^\top + \xi_k v_1^\top + \xi_k\xi_k^\top) + \gamma_k(v_1 v_1^\top + v_1\xi_k^\top + \xi_k v_1^\top + \xi_k\xi_k^\top). \tag{2.23}$$

$$||G_k|| \leq |\lambda_1|(||v_1\xi_k^\top|| + ||\xi_k v_1^\top|| + ||\xi_k\xi_k^\top||) + |\nu_k|(||v_1 v_1^\top|| + ||v_1\xi_k^\top|| + ||\xi_k v_1^\top|| + ||\xi_k\xi_k^\top||) \tag{2.24}$$

Express the initial vector q_0 of the power iteration in the eigenbasis of A: $v_0 = \sum_{i=1}^n c_i v_i$. For the next step, we recall by the previous two lemmas that

$$|\lambda_1 - \nu_k| \leq \max_{2\leq i\leq d}|\lambda_1 - \lambda_i| \tan^2(\theta_0) \left|\frac{\lambda_2}{\lambda_1}\right|^{2k}$$

and

$$||q_k - v_1|| \leq \left|\frac{\lambda_2}{\lambda_1}\right|^k \left(\sum_{i=1}^d (\frac{c_i}{c_1})^2\right)^{1/2}.$$

For brevity, we denote $C = \left(\sum_{i=1}^d [\frac{c_i}{c_1}]^2\right)^{1/2}$ and $D = \max_{2\leq i\leq d}|\lambda_1 - \lambda_i| \tan^2(\theta_0)$. Now, it also well known that for $u, v \in \mathbb{R}^l$ that $||uv^t|| = |u^\top v|$. Therefore, in conjunction with application of the Cauchy-Schwarz inequality, we have that

$$||v_1 v_1^\top|| = 1 \tag{2.25}$$

$$||v_1\xi_k^\top|| = ||\xi_k v_1^\top|| = |v_1^\top \xi_k| \leq |\xi_k| \leq C\left|\frac{\lambda_2}{\lambda_1}\right|^k \tag{2.26}$$

$$||\xi_k\xi_k^\top|| = |\xi_k^\top \xi_k| = |\xi_k|^2 \leq C^2\left|\frac{\lambda_2}{\lambda_1}\right|^{2k} \tag{2.27}$$

Noting that $\lambda_1 \leq 1$, we have then that

$$||G_k|| \leq 2C\left|\frac{\lambda_2}{\lambda_1}\right|^k + C^2\left|\frac{\lambda_2}{\lambda_1}\right|^{2k} + D\left|\frac{\lambda_2}{\lambda_1}\right|^{2k}(1 + 2C\left|\frac{\lambda_2}{\lambda_1}\right|^k + C^2\left|\frac{\lambda_2}{\lambda_1}\right|^{2k}). \tag{2.28}$$

Thus, $||G_k|| = \mathcal{O}(\tan^2(\theta_0)|\frac{\lambda_2}{\lambda_1}|^k)$. □

Lemma 13. *Let $v, w \in \mathbb{R}^d$ be of unit length, $A \in \mathbb{R}^{d\times d}$. We have then that*

$$|v^\top A w| \leq ||A||_F \tag{2.29}$$

Proof. Applying the Cauchy-Schwarz inequality and sub-multiplicativity of the induced 2-norm for matrices,

$$|v^\top A w| \leq ||v|| \cdot ||A|| \cdot ||w||_2 = ||A||_2 \leq ||A||_F.$$

□

We may establish how many initial steps of inexact deflation we need to run before it is set on a path towards convergence. Our setting is the same as in the previous lemmas.

Theorem 14. *Let $A \in \mathbb{R}^{d\times d}$ be symmetric PSD and w_0 be the initial unit vector for inexact deflation such that ${w_0}^\top x_2 \neq 0$. Fix $\tau > 1$, $\rho < 1/2$ and $\delta = \min\{\rho, \frac{1}{\tau\sqrt{d}}\}$. Then after $T = \mathcal{O}\big(\frac{1}{\lambda_1-\lambda_2}\log(\frac{\tan^2\theta_0}{\delta(\lambda_2-\lambda_3)})\big)$ steps, the perturbations G_k achieve the Hardt-Price bounds in Equation 2.21.*

Proof. Let $B = A - \lambda_1 v_1 {v_1}^\top$. For $k = 1, 2, \dots$ we have that step k in the pre-momentum phase involves computing $q_k = \frac{A^k q_{k-1}}{||A^k q_{k-1}||}$, $\nu_k = q_k^\top A q_k$ and then an inexact deflation $(A - \nu_k q_k^\top q_k) w_{k-1}$. We have then that inexact deflation at step k is representable as

$$(B + G_k) w_{k-1} = B w_{k-1} + G_k w_{k-1} \tag{2.30}$$

where G_k is associated with the error $||\lambda_1 v_1 {v_1}^\top - \nu_k q_k {q_k}^\top||$. By Lemma 12, the fact the w_k are unit, and sub-multiplicativity, we have that

$$||G_k w_{k-1}|| \leq ||G_k|| \cdot ||w_{k-1}|| = ||G_k|| = \mathcal{O}\Big(\tan^2 \theta_0 \left|\frac{\lambda_2}{\lambda_1}\right|^k\Big). \tag{2.31}$$

Reminding ourselves of the Hardt-Price bounds, we need any given perturbation to satisfy

$$5||G_k|| \leq \rho(\lambda_2 - \lambda_3) \quad \textit{and} \quad 5||G_k|| \leq (\lambda_2 - \lambda_3)\frac{1}{\tau\sqrt{d}} \tag{2.32}$$

Solving for k such that $\tan^2 \theta_0 \left|\frac{\lambda_2}{\lambda_1}\right|^k$ satisfies these bounds, we arrive at our claim.

□

2.7.3 Proof of Theorem 4

We first outline the setting before proving our main result. We let $A \in \mathbb{R}^{d \times d}$ be symmetric PSD with spectrum

$$1 \geq \lambda_1 > \lambda_2 > \lambda_3 \geq \cdots \geq \lambda_d \geq 0. \tag{2.33}$$

Distinct to our setting is the presence of a positive eigengap between λ_2 and λ_3. We select $q_0 = \sum_{i=1}^{n} c_i v_i$ such that $|v_1^\top q_0| \neq 0$, which will be used for the vanilla and momentum power iterations, and $w_0 = \sum_{i=1}^{d} b_i v_i$ such that $|{v_2}^\top w_0| \neq 0$, which will be used for inexact deflation. Let $\theta_0 = \arccos |q_0^\top v_1|$. Lastly, in the first for-loop we add the theoretical termination condition `while` $|\lambda_2 - \mu_k| > \rho$.

Theorem 15 (Restating of Theorem 4)**.** *Let J represent the number of steps in the pre-momentum phase and K the number of steps in the momentum phase as in Algorithm 1. Let $\epsilon < 1$ represent the desired error threshold of our v_1 estimates, i.e., $\sin^2\theta(q_t, v_1) < \epsilon$ and $\rho < \min\{1/2, \sqrt{\frac{\lambda_1-\lambda_2}{\lambda_2-\lambda_d}}\}$ represent the desired error threshold of our λ_2 estimates, i.e., $|\mu_k - \lambda_2| < \rho$. Select unit $q_0 \in \mathbb{R}^d$ and $w_0 \in \mathbb{R}^d$. Further fix $\tau > 1$ and $\delta = \min\{\rho, \frac{1}{\tau\sqrt{d}}\}$. Then after*

$$J = \mathcal{O}\Big(\frac{1}{\lambda_1 - \lambda_2} \log \frac{\tan^2\theta_0}{\delta(\lambda_2 - \lambda_3)} + \frac{\lambda_2}{\lambda_2 - \lambda_3} \log \frac{d\tau}{\rho}\Big), \tag{2.34}$$

$$K = \mathcal{O}\Big(\frac{\beta}{\sqrt{\lambda_1^2 - 4\beta^2}} \log \frac{1}{\epsilon}\Big) \tag{2.35}$$

pre-momentum and momentum steps, respectively, where $\beta = \widehat{\lambda}_2^2/4 = \mu_J^2/4$, with all but $\tau^{-\Omega(1)} + e^{-\Omega(d)}$ probability, DMPower *outputs a vector q_K with*

$$\sin^2\theta(q_K, v_1) < \epsilon. \tag{2.36}$$

Proof. We divide our proof into an analysis of the pre-momentum stage and then the momentum stage. Specifically, we will first establish how many iterations are needed to acquire a $\beta \in [\lambda_2^2/4, \lambda_1^2/4)$. Then, we will use this β to conduct Power+M updates and derive how many additional steps we will need to obtain a q_k with $1 - (q_k^\top v_1)^2 < \epsilon$.

Pre-momentum phase: By Theorem 14, after $J_1 = \mathcal{O}\big(\frac{1}{\lambda_1-\lambda_2}\log\frac{\tan^2\theta_0}{\delta(\lambda_2-\lambda_3)}\big)$ steps, we achieve the Hardt-Price bounds. Therefore, by Corollary 1.1 in [73], after a further $J_2 = \mathcal{O}\big(\frac{\lambda_2}{\lambda_2-\lambda_3}\log\frac{d\tau}{\rho}\big)$ iterations the w_t will converge to ρ-accuracy. That is, if we let $J = J_1 + J_2$ we have

$$||(I - w_J w_J^\top)v_2|| = 1 - (w_J^\top v_2) = \sin(\gamma_J) < \rho \tag{2.37}$$

with all but $\tau^{-\Omega(1)} + e^{-\Omega(n)}$ probability. Therefore in conjunction with Lemma 10 we have that

$$|\lambda_2 - \mu_J| \leq (\lambda_2 - \lambda_d)\sin^2(\gamma_J) < (\lambda_2 - \lambda_d)\rho^2 < \lambda_1 - \lambda_2 \tag{2.38}$$

so by Proposition 6 we obtain

$$\frac{1}{4}|\lambda_2^2 - \mu_J^2| < \frac{1}{4}|\lambda_1^2 - \lambda_2^2|. \tag{2.39}$$

We set $\beta = \frac{\mu_J^2}{4}$ as our momentum coefficient and proceed to the momentum phase.

Momentum phase: Our momentum coefficient is now within the interval of acceleration†, i.e., $\beta \in [\lambda_2^2/4, \lambda_1^2/4)$, so we may now invoke Theorem 2, which tell us that after

$$K = \mathcal{O}\left(\frac{\beta}{\sqrt{\lambda_1^2 - 4\beta^2}}\log\frac{1}{\epsilon}\right) \tag{2.40}$$

steps of Power+M iteration on q_J (which we now take to be our initial vector for Power+M), we will have that $1 - (q_{J+K}^\top v_1)^2 < \epsilon$, completing our proof.

† Subtly, we assume that $\mu_J \geq \lambda_2$. We discuss the case $\mu_J < \lambda_2$ in Theorem 23. □

Algorithm 4 Stochastic Power Method/Streaming PCA

Require: Streaming inputs $x_1, x_2, \cdots \in \mathbb{R}^d$, batch size n, unit $q_0 \in \mathbb{R}^d$, iterations T, unit $q_0 \in \mathbb{R}^d$

1: **for** $t = 1, 2, \ldots, T$ **do**
2: Generate unbiased estimate $\widehat{A}_t = \frac{1}{n}\sum_{i=(t-1)n+1}^{tn} x_i x_i^\top$
3: $q_t \leftarrow \widehat{A}_t q_{t-1}$
4: $q_t \leftarrow q_t / ||q_t||$
5: $\nu_t \leftarrow q_t^\top \widehat{A}_t q_t$ ▷ Inexact Rayleigh quotient estimate of λ_1
6: **end forreturn** q_T, ν_T

2.7.4 Proof of Theorem 5

We re-outline the typical streaming setting: we have d-dimensional data points $x_1, x_2, \cdots \sim \mathcal{D}$, with underlying covariance matrix $A \in \mathbb{R}^{d\times d}$ with eigenvalues $1 \geq \lambda_1 > \lambda_2 > \lambda_3 \geq \lambda_4 \geq \cdots \geq \lambda_n \geq 0$. Presumably, it is too costly to access and/or store A, but we have access to a stream of inputs $x_1, x_2, \ldots$. Algorithm 4 is a conventional streaming PCA method designed to recover the principal components of A using only a sample of inputs. As an abuse of notation, they are not indexed in any particular order – we receive the data points in a uniformly random manner. At each round, we form an unbiased estimate $\widehat{A} = \frac{1}{n}\sum_{i=1}^{n} x_i x_i^\top$, where n is a fixed batch size. We are interested in determining how many total samples are needed to output a vector q_t such that $\sin^2(q_t, v_1) < \epsilon$ for a fixed precision $\epsilon < 1$, which is also referred to as the sample complexity.

Similar to the vanilla power method, one may accelerate a conventional streaming PCA by attaching a momentum term. That is, our update step in Algorithm 4 would instead be

$$q_{j+1} \leftarrow \widehat{A}_j q_j - \beta q_{j-1} \tag{2.41}$$

where β is a momentum coefficient. De Sa et. al provide a guarantee for updates of this kind:

Theorem 16 ([42], Theorem 3)**.** *Suppose we run Algorithm 4 with momentum updates as in*

equation 2.41. Let $\Sigma = \mathbb{E}[(\widehat{A}_j - A) \otimes (\widehat{A}_j - A)]$. *Assume we initialize with unit* $q_0 \in \mathbb{R}^d$ *and with* $|v_1^\top q_0| \geq 1/2$. *For any* $\delta < 1$ *and* $\epsilon < 1$, *if* $2\sqrt{\beta} \in [\lambda_2, \lambda_1)$ *and*

$$J = \frac{\sqrt{\beta}}{\sqrt{\lambda_1 - 4\beta}} \log\Big(\frac{32}{\delta\epsilon}\Big) \;\; and \;\; ||\Sigma|| \leq \frac{(\lambda_1^2 - 4\beta)\delta\epsilon}{256\sqrt{d}J} = \frac{(\lambda_1^2 - 4\beta)^{3/2}\delta\epsilon}{256\sqrt{d}\sqrt{\beta}} \log^{-1}\Big(\frac{32}{\delta\epsilon}\Big), \tag{2.42}$$

then after J *updates with probability at least* $1 - 2\delta$, *we have that* $\sin^2(q_J, v_1) \leq \epsilon$.

Similar to Power+M, streaming PCA with momentum updates experiences noticeable acceleration. However, of particular note is the bounded variance condition and the initialization of β. The former has been addressed through further variance reduction techniques outlined in [42], but current literature does not suggest how to intelligently set β so that it lies in the convergence interval $[\lambda_2^2/4, \lambda_1^2/4)$. Similar to our design of DMPower, our Algorithm 2 successively approximates a convergent β in a pre-momentum phase before dropping into a momentum phase. The convergence analysis of DMStream is more challenging, however, since we have two sources of noise: the estimation error $||A - \widehat{A}||$ and the estimation error of $||v_1 - q_t||$. The remainder of this section is devoted to providing upper bounds on both sources of noise.

We have the following noisy representation of $\widehat{A}x$ for $x \in \mathbb{R}^d$:

$$\widehat{A}x = Ax + H \tag{2.43}$$

where $H = (A - \widehat{A})x$. In light of Theorem 14, if we can control $||H||$, then we can produce a convergent noisy power method. To this end, we provide several results to aid our analysis, some of which are not proven here. The first result bounds the error in a single step of a streaming PCA:

Lemma 17 ([73], Lemma 3.5)**.** *Let A be a covariance matrix as in the setting described above and $\widehat{A} = \frac{1}{n}\sum_{i=1}^{n} x_i x_i^\top$ an empirical estimate based off a streaming batch $x_1, x_2, \ldots, x_n$. Consider the noisy representation as in equation 2.43. Then with all but $\mathcal{O}(1/n^2)$ probability*

$$||H|| \leq \sqrt{\frac{\log^4 n \log d}{n}} + \frac{1}{n^2} \quad \textit{and} \quad ||v_1^\top H|| \leq \sqrt{\frac{\log^4 n \log d}{n} + \frac{1}{n^2}} \tag{2.44}$$

We now consider $H_t = (A - \widehat{A}_t)q_{t-1}$, where $\widehat{A}_t$ and q_t are the unbiased estimate of A and v_1 respectively in round j as in Algorithm 4.

Proposition 18 ([73], Theorem 3.2)**.** *Choose batch size n such that*

$$\frac{n}{\log^4 n} = \mathcal{O}\Big(\frac{1/\epsilon^2 \log d}{(\lambda_1 - \lambda_2)^2 d}\Big) \tag{2.45}$$

for $\epsilon < 1/2$ and we will have by Lemma 17 that

$$||H_t|| \leq \frac{\epsilon(\lambda_1 - \lambda_2)}{5} \quad \textit{and} \quad ||v_1^\top H_t|| \leq \frac{\lambda_1 - \lambda_2}{5\sqrt{d}}, \tag{2.46}$$

thereby satisfying the Hardt-Price bounds as in equation 2.21. Thus, by Theorem 14, after $T = \mathcal{O}(\log(d/\epsilon)/(1 - \lambda_2/\lambda_1))$ iterations, we have with all but $1 - \max\{1, T/n^2\}$ probability that Algorithm 2 outputs q_T such that $1 - (v_1^\top q_T)^2 < \epsilon$.

Now, consider the inexact update in Algorithm 2, $(\widehat{A}_t - \nu_t q_t q_t^\top)w_t$. For convenience, we let $B := A - \lambda_1 v_1 v_1^\top$, the exact deflation matrix. We may express this as

$$(\widehat{A}_t - \nu_t q_t q_t^\top)w_{t-1} = Bw_{t-1} + H_t + G_t \tag{2.47}$$

where $H_t = (A - \widehat{A}_t)w_{t-1}$ and $G_t = (\lambda_1 v_1 v_1^\top - \nu_t q_t q_t^\top)w_{t-1}$. By Proposition 18 we know how to control $||H_t||$, so we will now focus our attention on $||G_t||$. We will conduct analysis in the same spirit as Lemma 12, but first we will require a another pair of results. The first lemma demonstrates that if we satisfy the Hardt-Price bounds of equation 2.21, then $\tan\theta(v_1, q_t)$ decreases multiplicatively with each step of a noisy power method.

Lemma 19 ([73], Lemma 2.3 (modified))**.** *Let $v_1 \in \mathbb{R}^d$ be the dominant eigenvector with eigenvalue λ_1 of A, λ_2 the second dominant eigenvalue, $x \in \mathbb{R}^d$ a unit vector. Let $G \in \mathbb{R}^d$ and $\theta_0 = \arccos|v_1^\top x|$ satisfy*

$$4||v_1^\top G|| \leq (\lambda_1 - \lambda_2)\cos(\theta_0) \tag{2.48}$$

$$4||G|| \leq (\lambda_1 - \lambda_2)\epsilon \tag{2.49}$$

for some $\epsilon < 1$. Then

$$\tan\theta(v_1, Ax + G) \leq \max\{\epsilon, \max\{\epsilon, \left(\frac{\lambda_2}{\lambda_1}\right)^{1/4}\}\tan\theta_0\}. \tag{2.50}$$

We now provide a bound on the $\sin\theta(v_1, q_t)$ using Lemma 19.

Proposition 20 ([73], Theorem 2.4 (rephrased))**.** *Suppose q_0 is the initial vector supplied to a noisy power method, $\theta_0 = |v_1^\top q_0|$, and G_t is the noise experienced at round t. Further suppose that*

$$5||v_1^\top G_t|| \leq (\lambda_1 - \lambda_2)\cos\theta_0 \tag{2.51}$$

$$5||G_t|| \leq \epsilon(\lambda_1 - \lambda_2) \tag{2.52}$$

holds at every stage including and after round t for some $\epsilon < 1/2$. Then

$$\sin\theta(q_{t+k}, v_1) \leq \max\{\epsilon, (\lambda_2/\lambda_1)^{k/4}\tan\theta_0\} \tag{2.53}$$

Proof. Let $l \geq t$. Since $|G_l||$ satisfies the modified Hardt-Price bounds of Lemma 19, we have that

$$\tan\theta(v_1, q_l) \leq \max\{\epsilon, \max\{\epsilon, \tan\theta_0\}\}. \tag{2.54}$$

For $\epsilon < 1/2$ we have that,

$$\cos\theta(v_1, q_l) \geq \min\{1 - \epsilon^2/2, \cos\theta_0\} \geq \frac{7}{8}\cos\theta_0, \tag{2.55}$$

which means that we may invoke Lemma 19 at every step $l \geq t$. This gives us

$$\tan(v_1, q_{l+1}) = \tan\theta(v_1, Aq_l + G) \leq \max\{\epsilon, \max\{\epsilon, \left(\frac{\lambda_2}{\lambda_1}\right)^{1/4}\}\tan\theta(v_1, q_l)\}. \tag{2.56}$$

Extending this inequality recursively for $l + k$ and noting that $\sin\theta(q_l, v_1) \leq \tan\theta(q_l, v_1)$ gives us our claim. □

We are now prepared to conduct analysis on $||G_t||$ in equation 2.47.

Proposition 21. *Choose $\epsilon < 1/2$ and n such that*

$$\frac{n}{\log^4 n} = \mathcal{O}\Big(\frac{1/\epsilon^2 \log d}{(\lambda_1 - \lambda_2)^2 d}\Big). \tag{2.57}$$

. *Let*

$$\phi_t = |\lambda_1 - \lambda_d| \tan^2 \theta(q_0, v_1) \left|\frac{\lambda_2}{\lambda_1}\right|^{2t} + \min\{\frac{\epsilon(\lambda_1 - \lambda_2)}{5}, \frac{\lambda_1 - \lambda_2}{5\sqrt{d}}\} \tag{2.58}$$

$$\psi_t = \sqrt{2 - 2\sqrt{(1 - \min\{1, \max\{\epsilon^2, (\lambda_2/\lambda_1)^{t/2} \tan^2 \theta_0\})\}}}. \tag{2.59}$$

Then with all but $\mathcal{O}(1/n^2)$ *probability, we have that* $||G_t|| = \mathcal{O}(\max\{\phi_t, \psi_t\})$.

Proof. We first consider $|\lambda_1 - \nu_t| = |\lambda_1 - q_t^\top \widehat{A}_t q_t|$. Let $\theta_0 = q_0^\top v_1$. First observe that since $q_t \to v_1$, we have by Lemma 10 that $|\lambda_1 - q_t^\top A q_t| = \mathcal{O}(\sin^2(q_t, v_1))$. We have then that

$$|\lambda_1 - \nu_t| = |\lambda_1 - q_t^\top \widehat{A}_t q_1| = |\lambda_1 - q_t^\top A q_t + q_t^\top A q_t - q_t^\top \widehat{A}_t q_t| \tag{2.60}$$

$$\leq |\lambda_1 - q_t^\top A q_t| + |q_t^\top (A - \widehat{A}_t) q_t| \tag{2.61}$$

$$\leq |\lambda_1 - \lambda_d| \tan^2 \theta(q_0, v_1) \left|\frac{\lambda_2}{\lambda_1}\right|^{2t} + \min\{\frac{\epsilon(\lambda_1 - \lambda_2)}{10}, \frac{\lambda_1 - \lambda_2}{10\sqrt{d}}\} \tag{2.62}$$

where the last inequality follows from applying the result on Rayleigh quotient approximation in Lemma 10 to the left term and Proposition 18 along with the Cauchy-Schwarz inequality to the right term.

We now examine $||v_1 - q_t||$. We have chosen n in such a way that we satisfy the Hardt-Price bounds of equation 2.21, therefore, we may invoke Proposition 20 and conclude that

$$||v_1 - q_t|| \leq \sqrt{2 - 2\sqrt{(1 - \min\{1, \max\{\epsilon^2, (\lambda_2/\lambda_1)^{t/2} \tan^2 \theta_0\})\}}}, \tag{2.63}$$

where we have used the identity $||v_1 - q_t||^2 = 2 - 2\cos\theta(v_1, q_t)$ since unit v_1 and q_t are unit

vectors. Following the exact same analysis as in the proof of Lemma 12, we will arrive at

$$||G_t|| = \mathcal{O}(\max\{|\lambda_1 - \nu_t|, ||v_1 - q_t||\}). \tag{2.64}$$

Taking the upper bounds on $||\lambda_1 - \nu_t||$ and $||v_1 - q_t||$ which we derived above gives us our result. □

We are now prepared to prove our main convergence theorem for DMStream.

Theorem 22 (Restating of Theorem 5)**.** *Let* $\Sigma = \mathbb{E}[(\widehat{A}_t - A) \otimes (\widehat{A}_t - A)]$*, where* $\widehat{A}_j = \frac{1}{n}\sum_{i=1}^{n} x_i x_i^\top$ *represents any unbiased estimate of* A *in DMStream with fixed batch size* n*. Assume we initialize with a unit* $q_0 \in \mathbb{R}^d$ *where* $d \gg 0$ *and* $|v_1^\top q_0| \geq 1/2$*. Let* $\theta_0 = \arccos |q_0^\top v_1|$*. For any* $\delta < 1$*,* $\epsilon < 1$*, suppose*

$$||\Sigma|| \leq \frac{(\lambda_1^2 - 4\beta)\delta\epsilon}{256\sqrt{d}J} = \frac{(\lambda_1^2 - 4\beta)^{3/2}\delta\epsilon}{256\sqrt{d}\sqrt{\beta}} \log^{-1}\Big(\frac{32}{\delta\epsilon}\Big), \tag{2.65}$$

where J *is the total number of pre-momentum steps we have fixed at runtime. Furthermore, we let* $\rho < \min\{1/2, \sqrt{\frac{\lambda_1 - \lambda_2}{\lambda_2 - \lambda_d}}\}$ *represent the error threshold of our* λ_2 *estimates, i.e.,* $|\mu_k - \lambda_2| < \rho$*. Lastly, fix* $\tau > 1$ *and* $\delta = \min\{\rho, \frac{1}{\tau\sqrt{d}}\}$*. If the batch size* n *is chosen such that*

$$\frac{n}{\log^4 n} = \mathcal{O}\Big(\frac{1/\gamma^2 \log d}{(\lambda_2 - \lambda_3)^2 d}\Big). \tag{2.66}$$

where $\gamma = \frac{\rho(\lambda_2 - \lambda_3)}{10\tau\sqrt{d}}$*, then after*

$$J = \mathcal{O}\left(\frac{1}{\lambda_1 - \lambda_2} \log\left(\frac{\tan^2\theta_0 \tau\sqrt{d}}{\rho(\lambda_2 - \lambda_3)}\right) + \frac{\lambda_2}{\lambda_2 - \lambda_3} \log\frac{d\tau}{\rho}\right), \tag{2.67}$$

$$K = \frac{\sqrt{\beta}}{\sqrt{\lambda_1 - 4\beta}} \log\Big(\frac{32}{\delta\epsilon}\Big) \tag{2.68}$$

pre-momentum steps and momentum steps respectively, with $(1-\frac{1}{n^2})(1-2\delta)(1-\tau^{-\Omega(1)}+e^{-\Omega(d)})$ *probability DMStream outputs a vector* q_K *such that*

$$\sin^2 \angle(q_K, v_1) < \epsilon. \tag{2.69}$$

Proof. As a reminder, our exact deflation matrix B has spectrum $\lambda_2 > \lambda_3 \geq \lambda_4 \geq \cdots \lambda_n$ with respective eigenvectors $v_2, v_3, \ldots, v_n$. We will show that for our choice of n and J, that $||H_t||+||G_t||$ for all $t > J$ satisfy the Hardt-Price bounds, therefore allowing our inexact deflation to probabilistically succeed.

First we examine $||H_t||$. By our choice of n and since $d \gg 0$, we have by Proposition 18 that

$$||H_t|| \leq \frac{\gamma(\lambda_2 - \lambda_3)}{5} = \frac{\rho(\lambda_2 - \lambda_3)^2}{50\tau\sqrt{d}} < \frac{\rho(\lambda_2 - \lambda_3)}{10\tau\sqrt{d}}, \tag{2.70}$$

$$||v_2^\top H_t|| \leq ||H_t|| < \frac{\rho(\lambda_2 - \lambda_3)}{10\tau\sqrt{d}}. \tag{2.71}$$

Now, we will analyze $||G_t||$. By Proposition 21, we must consider both ϕ_t and ψ_t.

Case 1: $\phi_t > \psi_t$. For our choice of n we have that

$$\phi_t = |\lambda_1 - \lambda_d| \tan^2\theta(q_0, v_1) \left|\frac{\lambda_2}{\lambda_1}\right|^{2t} + \min\{\frac{\gamma(\lambda_1 - \lambda_2)}{5}, \frac{\lambda_1 - \lambda_2}{5\sqrt{d}}\} \tag{2.72}$$

$$= |\lambda_1 - \lambda_d| \tan^2\theta(q_0, v_1) \left|\frac{\lambda_2}{\lambda_1}\right|^{2t} + \frac{\rho(\lambda_2 - \lambda_3)}{50\tau\sqrt{d}} \tag{2.73}$$

$$< |\lambda_1 - \lambda_d| \tan^2 \theta(q_0, v_1) \left|\frac{\lambda_2}{\lambda_1}\right|^{2t} + \frac{\rho(\lambda_2 - \lambda_3)}{20\tau\sqrt{d}}. \tag{2.74}$$

Solving for $|\lambda_1 - \lambda_d| \tan^2 \theta_0 \left|\frac{\lambda_2}{\lambda_1}\right|^{2t} < \frac{\rho(\lambda_2-\lambda_3)}{20\tau\sqrt{d}}$ we get

$$t = \mathcal{O}\left(\frac{1}{\lambda_1 - \lambda_2} \log\left(\frac{\tan^2 \theta_0 \tau\sqrt{d}}{\rho(\lambda_2 - \lambda_3)}\right)\right), \tag{2.75}$$

therefore, in this many steps we will have that $\phi_t < \frac{\rho(\lambda_2-\lambda_3)}{10\tau\sqrt{d}}$.

Case 2: $\psi_t \geq \phi_t$. If $\min\{1, \max\{\gamma^2, (\lambda_2/\lambda_1)^{t/2} \tan^2 \theta_0\})\} = 1$ then $||G_t|| \leq \psi_t = 0$, i.e., we have 0 noise. So we consider the more interesting case where our minimum is $\max\{\gamma^2, (\lambda_2/\lambda_1)^{t/2} \tan^2 \theta_0\}$. We first note that for any $\alpha \leq 1$, we have that

$$\psi_t = \sqrt{2 - 2\sqrt{1-\alpha}} \leq \alpha. \tag{2.76}$$

If for all t our max is γ^2, we have that

$$||G_t|| \leq \psi_t \leq \gamma^2 \leq \frac{\rho^2(\lambda_2 - \lambda_3)^2}{100\tau^2 d} < \frac{\rho(\lambda_2 - \lambda_3)}{10\tau\sqrt{d}}. \tag{2.77}$$

Otherwise, solving for

$$\left|\frac{\lambda_2}{\lambda_1}\right|^{t/2} \tan^2 \theta(q_0, v_1) < \frac{\rho(\lambda_2 - \lambda_3)}{10\tau\sqrt{d}} \tag{2.78}$$

we get

$$t = \mathcal{O}\left(\frac{1}{\lambda_1 - \lambda_2} \log\left(\frac{\tan^2 \theta_0 \tau\sqrt{d}}{\rho(\lambda_2 - \lambda_3)}\right)\right). \tag{2.79}$$

Now, for our choice of n, after $J_1 := t = \mathcal{O}\left(\frac{1}{\lambda_1-\lambda_2} \log\left(\frac{\tan^2 \theta_0 \tau\sqrt{d}}{\rho(\lambda_2-\lambda_3)}\right)\right)$ steps, we have that for all

$t > J_1$,

$$||H_t + G_t|| \leq ||H_t|| + ||G_t|| < \frac{\rho(\lambda_2 - \lambda_3)}{10\tau\sqrt{d}} + \frac{\rho(\lambda_2 - \lambda_3)}{10\tau\sqrt{d}} < \frac{\rho(\lambda_2 - \lambda_3)}{5}. \tag{2.80}$$

and

$$||v_2(H_t + G_t)|| \leq ||H_t|| + ||G_t|| < \frac{\rho(\lambda_2 - \lambda_3)}{10\tau\sqrt{d}} + \frac{\rho(\lambda_2 - \lambda_3)}{10\tau\sqrt{d}} < \frac{\lambda_2 - \lambda_3}{5\tau\sqrt{d}}, \tag{2.81}$$

thereby satisfying the Hardt-Price bounds of equation 2.21, so after a further $J_2 := \mathcal{O}\big(\frac{\lambda_2}{\lambda_2 - \lambda_3} \log \frac{d\tau}{\rho}\big)$ steps, our pre-momentum phase, with a total of $J = J_1 + J_2$ steps, outputs a vector q_J such that $\sin\theta(q_J, v_1) < \rho$ with probability $(1 - \frac{1}{n^2})(1 - \tau^{-\Omega(1)} + e^{-\Omega(d)})$. Through dual application of Lemma 10 and Proposition 6, similar to the proof of Theorem 4, we have that $\beta = \mu_J^2/4 \in [\lambda_2^2/4, \lambda_2^2/4)$, therefore, we may proceed to the momentum phase.

By our assumptions on the variance of our unbiased estimates $\widehat{A}_t$ in relation to J, we have by Theorem 16 that after a further $K = \frac{\sqrt{\beta}}{\sqrt{\lambda_1 - 4\beta}} \log\Big(\frac{32}{\delta\epsilon}\Big)$ steps our entire algorithm outputs a vector q_{J+K} with $\sin^2\theta(q_{J+K}, v_1) < \epsilon$, with probability $(1 - 2\delta)$. By multiplying the probability of the pre-momentum phase succeeding with the probability of the momentum phase succeeding, our full claim follows. □

2.7.5 Experimental Data

We provide raw experimental data used to generate various plots displayed in this paper (mostly related to iteration complexity) and include comparative wall-time benchmarks for vanilla power method, Power+M, DMPower, and the Lanczos algorithm.

Table 2.1: Iterations required for Power+M to converge at various sub-optimal and optimal β assignments. $\beta = \lambda_2^2/4 = 0.2025$ is the optimal momentum coefficient at this setting, where $\lambda_2 = 0.9$.

Sub-optimal β Selection for Power+M Data with Loose Eigengaps: `spec`=[1,0.9,0.8,...,0.8]							
Error Threshold (ϵ)	10e-9	10e-8	10e-7	10e-6	10e-5	10e-4	10e-3
Vanilla Power Method	81.097	70.309	59.230	48.746	37.263	26.873	16.671
Power+M, $\beta = 0.1025$	60.463	52.565	44.420	36.745	28.272	20.561	12.859
Power+M, $\beta = 0.2025$	**34.986**	**30.954**	**26.764**	**22.879**	**18.497**	**14.472**	**10.205**
Power+M, $\beta = 0.3025$	43.579	38.605	32.439	27.265	20.991	16.182	10.981
Power+M, $\beta = 0.4025$	89.440	76.524	65.099	54.101	42.651	31.200	19.501
Power+M, $\beta = 0.4225$	111.972	94.846	81.729	68.635	50.530	36.275	23.144
Power+M, $\beta = 0.4525$	179.062	155.476	130.219	106.71	81.776	58.348	32.412

Table 2.2: Absolute difference between final approximation of λ_2 and true value of λ_2.

Accuracy vs Simultaneous Power Data with Loose Eigengaps: `spec`=[1,0.9,0.8,...,0.8]							
Error Threshold (ϵ)	10e-9	10e-8	10e-7	10e-6	10e-5	10e-4	10e-3
Simultaneous PM	0.1723	0.1721	0.1684	0.1628	0.1466	0.1107	0.1126
DMPower, $\rho = \epsilon$	**0.0000**	**0.0000**	**0.0003**	**0.0017**	**0.0054**	**0.0370**	0.0678
DMPower, $\rho = \sqrt{\epsilon}$	0.0065	0.0145	0.0316	0.0696	0.0672	0.0552	0.0574
DMPower, $\rho = \sqrt[3]{\epsilon}$	0.0570	0.0692	0.0648	0.0545	0.0570	0.0577	0.0505
DMPower, $\rho = \sqrt[4]{\epsilon}$	0.0667	0.0538	0.0537	0.0564	0.0587	0.0529	**0.0488**

Table 2.3: Iterations for Vanilla PM, Power+M, and DMPower at `spec`=[1, 0.99, 0.98,..., 0.98].

Iteration Complexity Data with $A \in \mathbb{R}^{10\times 10}$						
Error Threshold (ϵ)	10e-2	10e-3	10e-4	10e-5	10e-6	10e-7
Vanilla Power Method	**2.0**	77.18	185.4	285.4	385.42	474.62
Power+M, $\beta = \lambda_2^2/4$	2.0	40.5	98.86	143.42	199.82	**231.74**
DMPower, $\rho = \epsilon$	3.0	44.5	104.7	139.9	197.08	243.84
DMPower, $\rho = \sqrt{\epsilon}$	3.0	33.86	93.7	153.48	192.86	249.52
DMPower, $\rho = \sqrt[3]{\epsilon}$	3.0	36.8	102.36	**138.82**	200.48	234.48
DMPower, $\rho = \sqrt[4]{\epsilon}$	3.0	37.7	**81.28**	144.02	**192.72**	243.58
Lanczos Algorithm	10.0	**10.0**	89.08	184.66	284.78	388.56

Iteration Complexity Data with $A \in \mathbb{R}^{100\times 100}$						
Error Threshold (ϵ)	10e-2	10e-3	10e-4	10e-5	10e-6	10e-7
Vanilla Power Method	**1.0**	141.18	211.34	293.94	378.1	472.98
Power+M, $\beta = \lambda_2^2/4$	2.0	**68.66**	120.4	**152.32**	197.84	262.8
DMPower, $\rho = \epsilon$	3.0	76.54	113.9	156.6	203.26	259.2
DMPower, $\rho = \sqrt{\epsilon}$	3.0	71.92	116.5	156.08	**191.64**	257.66
DMPower, $\rho = \sqrt[3]{\epsilon}$	3.0	75.44	114.6	159.08	194.34	**238.66**
DMPower, $\rho = \sqrt[4]{\epsilon}$	3.0	77.1	**107.04**	158.2	196.74	245.38
Lanczos Algorithm	100.0	100.0	206.06	315.62	431.12	551.76

Iteration Complexity Data with $A \in \mathbb{R}^{500\times 500}$						
Error Threshold (ϵ)	10e-2	10e-3	10e-4	10e-5	10e-6	10e-7
Vanilla Power Method	**1.0**	185.98	259.22	360.12	429.16	489.4
Power+M, $\beta = \lambda_2^2/4$	1.0	93.16	133.48	163.18	214.64	259.0
DMPower, $\rho = \epsilon$	2.0	102.94	133.18	163.98	**203.88**	264.72
DMPower, $\rho = \sqrt{\epsilon}$	2.0	**93.06**	**131.76**	171.62	207.24	252.4
DMPower, $\rho = \sqrt[3]{\epsilon}$	2.0	97.56	143.66	170.06	212.36	**252.24**
DMPower, $\rho = \sqrt[4]{\epsilon}$	2.0	94.12	137.5	**161.2**	213.24	262.36
Lanczos Algorithm	500.0	500.0	607.24	717.84	833.56	964.18

Table 2.4: Percentage of iterations in the *momentum phase* for `spec`=[1, 0.99, 0.98,..., 0.98] with $A \in \mathbb{R}^{100\times 100}$. DMPower across a variety of ρ settings spends the vast majority of its time in the less computationally-intensive momentum phase.

Phases Iteration Complexity Data with $A \in \mathbb{R}^{100\times 100}$				
Error Threshold (ϵ)	10e-4	10e-5	10e-6	10e-7
DMPower, $\rho = \epsilon$	64.72%	88.15%	99.06%	98.71%
DMPower, $\rho = \sqrt{\epsilon}$	99.6%	99.49%	99.28%	98.72%
DMPower, $\rho = \sqrt[3]{\epsilon}$	99.6%	99.49%	99.29%	98.75%

Table 2.5: Recorded wall-time convergence speeds (in nanoseconds) for vanilla power method, Power+M with optimal β assignment, and various settings of DMPower, and the Lanczos algorithm. Performed 1000 calls using random PSD of various sizes with fixed spectrum $\lambda_1 = 1, \lambda_2 = 0.99, \lambda_3 = 0.98$, and remaining eigenvalues set to 0.98. In several instances DMPower exhibits faster wall-time speeds than Power+M with optimal β assignment, consistently outperforms Lanczos, and markedly accelerates the vanilla power method at all error thresholds (ϵ) tighter than 0.1.

Wall-Time Performance Data with $A \in \mathbb{R}^{10\times 10}$						
Error Threshold (ϵ)	10e-2	10e-3	10e-4	10e-5	10e-6	10e-7
Vanilla Power Method	**67.32**	3186.6	7440.56	11497.34	15388.04	18846.76
Power+M, $\beta = \lambda_2^2/4$	101.54	1941.46	4539.84	**6140.58**	**8514.94**	**9984.74**
DMPower, $\rho = \epsilon$	251.28	2117.2	4813.14	6436.1	8982.4	11550.66
DMPower, $\rho = \sqrt{\epsilon}$	262.54	1696.62	4383.94	7066.94	8869.68	11470.54
DMPower, $\rho = \sqrt[3]{\epsilon}$	260.84	1837.14	4732.32	6508.14	9440.62	16254.96
DMPower, $\rho = \sqrt[4]{\epsilon}$	256.94	1878.86	**3830.36**	6586.34	8938.38	11294.3
Lanczos Algorithm	1463.06	**1471.86**	5818.92	7836.14	11705.58	22829.88

Wall-Time Performance Data with $A \in \mathbb{R}^{100\times 100}$						
Error Threshold (ϵ)	10e-2	10e-3	10e-4	10e-5	10e-6	10e-7
Vanilla Power Method	**107.24**	10027.98	16213.82	21231.46	29691.06	31801.24
Power+M, $\beta = \lambda_2^2/4$	132.2	**4890.36**	9701.76	**10840.92**	**13930.12**	21100.88
DMPower, $\rho = \epsilon$	432.06	5969.96	8910.08	11985.2	15986.18	**19431.32**
DMPower, $\rho = \sqrt{\epsilon}$	457.82	5765.98	9028.7	12397.06	15048.34	19432.9
DMPower, $\rho = \sqrt[3]{\epsilon}$	435.48	6113.44	9393.82	12575.0	15028.84	20037.1
DMPower, $\rho = \sqrt[4]{\epsilon}$	432.32	6094.92	**8143.68**	12136.16	16063.48	20166.24
Lanczos Algorithm	7724.94	6672.18	13832.8	21457.2	29447.52	39152.46

Wall-Time Performance Data with $A \in \mathbb{R}^{500\times 500}$						
Error Threshold (ϵ)	10e-2	10e-3	10e-4	10e-5	10e-6	10e-7
Vanilla Power Method	**307.9**	27496.5	35858.14	49056.32	66559.52	70891.42
Power+M, $\beta = \lambda_2^2/4$	337.26	**12912.76**	**20098.66**	24924.28	31590.88	49575.94
DMPower, $\rho = \epsilon$	1423.64	15366.22	20172.32	**24663.36**	**30818.32**	38748.8
DMPower, $\rho = \sqrt{\epsilon}$	2094.86	15868.62	20382.9	27008.76	30883.86	38207.12
DMPower, $\rho = \sqrt[3]{\epsilon}$	1490.0	15209.92	22283.78	26917.66	31475.8	**36617.02**
DMPower, $\rho = \sqrt[4]{\epsilon}$	1623.94	17126.1	20743.88	25229.04	32106.58	39925.98
Lanczos Algorithm	58438.68	58700.02	80682.24	91623.18	106078.32	126685.16

Table 2.6: Averaged log error, batch size = 500, with performance measured by $\log_{10}\big(1 - \frac{||X^\top q_K||}{||X^\top v_1||}\big)$. DMStream registers much better accuracy than Oja's algorithm and emulates the performance of optimal Mini-Batch Power+M. We notice that accuracy does not improve as we increase the number of epochs, which is commonly observed for streaming algorithm running with small batch sizes. We demonstrate in Figure 2.4 that increasing batch size results in improved accuracy.

Log Error Performance Data with batch size = 500					
Epochs	10	20	30	40	50
DMStream, $\rho = 0.1$	-1.900	-1.894	-1.983	-1.969	-1.959
DMStream, $\rho = 0.01$	-1.992	-1.908	-1.882	-1.949	-1.905
DMSteam, $\rho = 0.001$	-1.929	-1.9585	-1.936	-1.963	-1.973
Oja $\eta_t = 3/t$	-0.588	-0.599	-0.625	-0.565	-0.549
Oja $\eta_t = 9/t$	-0.629	-0.592	-0.599	-0.531	-0.638
Oja $\eta_t = 27/t$	-0.680	-0.668	-0.584	-0.599	-0.647
Oja $\eta_t = 81/t$	-0.590	-0.676	-0.527	-0.604	-0.665
Mini-Batch Power+M (optimal $\beta = \lambda_2^2/4$)	-1.881	-1.860	-1.964	-1.996	-1.966

2.7.6 Precision Bounds for Momentum

In our main result Theorem 4, we assume that $\beta \in [\lambda_2^2/4, \lambda_1^2/4)$, while it is possible that we select $\beta < \lambda_2^2/4$. This poses no problem: as long as β is near $\lambda_2^2/4$ we will experience similar acceleration effects. We state the more general version of Theorem 2 which reflects this fact and add our own modified condition, $\Delta_{2,3} := \lambda_2 - \lambda_3 > 0$.

Theorem 23 (Generalized Convergence of Power+M [42]). *Given a PSD matrix $A \in \mathbb{R}^{d\times d}$ with eigenvalues $\lambda_1 > \lambda_2 > \lambda_3 \dots \lambda_d \geq 0$ with associated orthonormal eigenvectors $v_1, v_2, \dots, v_d$, for a unit $q_0 \in \mathbb{R}^n$ non-orthogonal to v_1, running Power+M with $\beta \leq \lambda_1$ results in q_k with*

$$\sin^2(\theta_k) = 1 - (q_k^\top v_1)^2 \leq \frac{1}{|q_0^\top v_1|^2} \cdot \begin{cases} 4\left(\frac{2\sqrt{\beta}}{\lambda_1+\sqrt{\lambda_1^2-4\beta}}\right)^{2k}, & \lambda_2 < 2\sqrt{\beta} \\ \left(\frac{\lambda_2+\sqrt{\lambda_2^2-4\beta}}{\lambda_1+\sqrt{\lambda_1^2-4\beta}}\right)^{2k}, & \lambda_2 \geq 2\sqrt{\beta} \end{cases} \tag{2.82}$$

Ultimately then, our β can land on either side of $\frac{\lambda_2^2}{4}$ and we we will experience acceleration as long as we are "close." The next result establishes tolerance for our estimations $\widehat{\lambda}_2$ of λ_2.

Proposition 24 (Proposition 6 re-stated). *The momentum phase of DMPower set with momentum coefficient $\beta = \widehat{\lambda}_2^2/4 = \mu_J^2/4$ converges if and only if*

$$|\lambda_2 - \widehat{\lambda}_2| \leq \Delta_{1,2}. \tag{2.83}$$

Proof. If $|\lambda_2 - \widehat{\lambda}_2| = |\lambda_2 - \mu_J| \leq \rho = \Delta_{1,2} = |\lambda_1 - \lambda_2|$, then

$$|\lambda_2 - \mu_J| \cdot |\lambda_2 + \mu_J| < |\lambda_1 - \lambda_2| \cdot |\lambda_1 + \lambda_2| \tag{2.84}$$

$$\Rightarrow |\lambda_1^2 - \mu_J^2| < |\lambda_1^2 - \lambda_2^2| \tag{2.85}$$

$$\Rightarrow \frac{1}{4}|\lambda_1^2 - \mu_J^2| < \frac{1}{4}|\lambda_1^2 - \lambda_2^2| \tag{2.86}$$

The first line of our inequality follows from the fact that $\mu_J > 0$ since A is PSD, therefore implicitly we have that $0 < \mu_J < \lambda_1$, implying $|\lambda_2 + \mu_J| < |\lambda_2 + \lambda_1|$. This shows we satisfy the constraint. To show that this is necessary and sufficient, we consider the case where $\rho > \Delta_{1,2}$. This allows for possible selection of $\mu_J > \lambda_1$, in which case we have that $\frac{1}{4}|{\lambda_2}^2 - \mu_k^2| > \frac{1}{4}|\lambda_1^2 - \lambda_2^2|$, which is outside of our guaranteed interval for convergence. □

2.7.7 Data Matrix Generation for Non-Streaming Experiments

For every experiment, we ran variations of the power method on a random covariance matrix A with a fixed spectrum. We constructed such matrices using a synthetic singular value decomposition (SVD). Specifically, we begin with a diagonal $d \times d$ matrix $\Sigma = \text{diag}\{1, \sqrt{\lambda_2}, \ldots, \sqrt{\lambda_d}\}$. Notice that by default we set $\lambda_1 = 1$. In practice, we set $\sqrt{\lambda_3} = \sqrt{\lambda_4} = \cdots = \sqrt{\lambda_d}$ for simplicity. We then drew two random orthogonal matrices $U \in \mathbb{R}^{1000 \times d}$ and $V \in \mathbb{R}^{d \times d}$ from the Haar distribution. We then form the data matrix $X = dU\Sigma V^\top \in \mathbb{R}^{1000 \times d}$, from which we acquire our covariance matrix $A = \frac{1}{1000} XX^\top$ with spectrum $\text{diag}\{\lambda_1, \lambda_2, \ldots, \lambda_d\}$. For every run in our non-streaming experiments, we would generate a new covariance matrix with our desired spectrum using this construction.

2.7.8 Spectral Clustering & Experimental Data

We first describe provide and discuss the algorithm used for deflation-based power iteration clustering [50]. We then provide the results of our spectral clustering experiments. We note that

Algorithm 5 Deflation-based Power Iteration Clustering (DPIC)

Require: normalized affinity matrix $W \in \mathbb{R}^{d \times d}$

1: $W_0 = W$
2: **for** $i = 1, 2$ **do** ▷ Top-2 component recovery
3: $v_i = \text{PowerIteration}(W_{i-1})$ ▷ Return the leading eigenvector
4: $W_i = W_{i-1} - \frac{W_{i-1} v_i v_i^\top W_{i-1}}{v_i^\top W_{i-1} v_i}$ ▷ Schur complement deflation
5: $i \leftarrow i + 1$
6: **end for**
7: Use k-means on eigenvector approximates v_1, v_2.
8: **return** C_1, C_2

PowerIteration in step 3 of Algorithm 5 may be replaced with DMPower, Power+M, or any other variant. To recover the second eigenvector, DPIC uses a Schur complement deflation [57] on W once the leading eigenvector is computed. Briefly, deflating a matrix shifts its spectrum so that the second leading eigenvalue/eigenvector now becomes the leading eigenvalue/eigenvector, upon which a power iteration may be used again. Successive deflations allows one to recover as many eigenvectors as desired, although numerical instability is increased with each deflation.

Table 2.7: Proportion of data points correctly classified by spectral clustering combined with k-means. As the error-threshold (and therefore, accuracy) of the eigenvector output is tightened, accuracy improves, eventually achieving perfect classification. See Figure 2.5 for a visual depiction of data separation using DMPower ($\rho = \sqrt[3]{\epsilon}$). We further note that although the vanilla power method achieves perfect classification at lower error-thresholds, it is at the cost of significantly more iterations as indicated in Table 2.8.

Spectral Clustering Accuracy Data on Concentric Circles Dataset					
Error Threshold (ϵ)	10e-2	10e-4	10e-6	10e-8	10e-10
Vanilla Power Method	0.6953	0.7854	1.0000	1.0000	1.0000
Power+M, $\beta = \lambda_2^2/4$	0.6679	0.7158	0.7810	0.9886	1.0000
DMPower, $\rho = \epsilon$	0.6997	0.7215	0.8191	0.9869	1.0000
DMPower, $\rho = \sqrt{\epsilon}$	0.6908	0.6927	0.7779	0.9881	1.0000
DMPower, $\rho = \sqrt[3]{\epsilon}$	0.6770	0.7056	0.7657	0.9872	1.0000

Spectral Clustering Accuracy Data on Half Moons Dataset					
Error Threshold (ϵ)	10e-2	10e-4	10e-6	10e-8	10e-10
Vanilla Power Method	0.6164	0.7793	0.9808	1.0000	1.0000
Power+M, $\beta = \lambda_2^2/4$	0.5966	0.6362	0.7560	1.0000	1.0000
DMPower, $\rho = \epsilon$	0.6196	0.6616	0.8185	0.9696	1.0000
DMPower, $\rho = \sqrt{\epsilon}$	0.6132	0.6112	0.7368	1.0000	1.0000
DMPower, $\rho = \sqrt[3]{\epsilon}$	0.6084	0.6283	0.8054	1.0000	1.0000

Table 2.8: Iteration complexity required to recover principal components. DMPower requires significantly fewer iterations for eigenvector recovery when compared to the vanilla power method, and closely mimics the performance of Power+M with $\beta = \lambda_2^2/4$.

Spectral Clustering Iterations Data on Concentric Circles Dataset					
Error Threshold (ϵ)	10e-2	10e-4	10e-6	10e-8	10e-10
Vanilla Power Method	7.72	59.16	605.36	1457.56	2426.36
Power+M, $\beta = \lambda_2^2/4$	**3.00**	**9.24**	85.12	**642.76**	**1321.00**
DMPower, $\rho = \epsilon$	6.00	11.16	92.00	751.68	1449.08
DMPower, $\rho = \sqrt{\epsilon}$	5.00	11.20	**77.68**	694.20	1452.84
DMPower, $\rho = \sqrt[3]{\epsilon}$	5.00	10.64	78.48	642.92	1505.88

Spectral Clustering Iterations Data on Half Moons Dataset					
Error Threshold (ϵ)	10e-2	10e-4	10e-6	10e-8	10e-10
Vanilla Power Method	7.12	58.72	506.00	1061.68	1929.52
Power+M, $\beta = \lambda_2^2/4$	**3.00**	**11.20**	**84.84**	**601.12**	1226.72
DMPower, $\rho = \epsilon$	6.00	12.08	89.16	629.96	1192.68
DMPower, $\rho = \sqrt{\epsilon}$	5.00	11.56	94.88	661.20	1225.08
DMPower, $\rho = \sqrt[3]{\epsilon}$	5.00	11.52	92.64	711.00	**1180.72**

Chapter 3: Representation and FLOPs-Minimal Evaluation of Multilinear Operations

3.1 Introduction

Modern neural networks are both expressive and accurate over a wide variety of learning and classification problems, but at the cost of increased width and depth. State-of-the-art convolutional models, for example, can contain up to several billion parameters [81, 82, 83]. The size and training costs of such networks are at odds with a rapidly emerging industrial and academic interest in performing learning tasks on low-fidelity hosts such as IoT and mobile devices [84, 85, 86]. An increasingly popular approach for generating compact yet expressive models is to use Tensorial Neural Networks (TNNs) [2, 87, 88, 89], which factorize each layer's weight tensor into several smaller factors. As a result, each TNN layer is a multilinear operation of its input and weight factors.

One method by which a TNN can arise is reshaping a network weight into a higher order tensor. These reshaped weights are then decomposed into factorized forms, such as canonical polyadic (CP), tensor train (TT), and Tucker (TK) decompositions. Factorization of a reshaped tensor is an efficient way of representing underlying properties such as periodicity and modularity invariances/similarities, which often exist in neural network models [2, 87, 90, 91], and preserved

in the factorization. In addition to their lighter weight, TNNs can preserve the same predictive power level over several popular backbone networks and tasks.

However, in contrast to the availability of high performance libraries for convolutional neural networks (CNNs), e.g., NVIDIA's cuDNN for GPUs or Intel's MKL for CPUs), solutions for efficient evaluation of multilinear operations (MLOs) through their tensor-decomposed counterparts are relatively unavailable. Emphasis in existing literature and implementations has been placed on speed-up via compression, i.e., reducing the number of MLOs required by tensorial convolutional kernels by reducing the number of their parameters via rank-reduction. Few works have focused on the reduction of convolutional MLOs when the size of the layer is fixed. Since the convolution operator is fundamental in many cutting-edge architectures and highly optimized by parallel computing libraries such as CUDA, support for this multilinear operation in TNN training, would benefit the tensor network community.

In this work, our contributions are summarized as follows:

1. We introduce a framework for representation and optimal evaluation order of multilinear floating point operations (FLOPs) through convolutional tensor layers.

2. Using our framework, one can represent a tensorial forward or backward computation as a *generalized* einsum graph/sequence and submit it to our novel meta-algorithm conv_einsum for FLOPs-minimized evaluation. Here, "generalized" means including convolutions, an operation not supported by any existing einsum implementation. To determine an optimal evaluation path, we develop an extension of the netcon algorithm [92] to support the expense of convolutions in its underlying cost model.

3. We present theory and a comprehensive set of experiments which demonstrate significant

improvements in computational and memory efficiency by using conv_einsum to train TNNs. **(4)** Our code is available as an open-source library for training tensorized ResNets under a variety of compression rates and factorizations.

3.2 Related works

Tensor networks. Tensor networks are widely used in quantum physics [93], numerical analysis [94], and machine learning [95, 96]. In neural networks, [97] and [98] use tensor networks to prove the expressive power of convolutional and recurrent neural networks. Recently, [99] combine tensor networks with genetic algorithms to search for efficient layer designs. Traditional tensor networks do not support *convolutions* in their operations, which is essential in convolutional neural networks. However, in recent years, so-called quantum convolutional neural networks (QCNNs) used for tasks such as quantum phase classification have been shown to share the same circuit structure as multiscale entanglement renormalization ansatzes (MERAs), which are classical tensor networks used to simulate many-body systems. Interestingly enough, [92] used netcon to optimize multilinear operations through MERAs, but did not consider convolutions.

Low-rank factorization. Various types of low-rank factorization have been proposed to reduce the number of parameters in linear layers. Pioneering works proposed to flatten/unfold the parameters in convolutional layers into matrices (known as *matricization*), followed by (sparse) dictionary learning or matrix decomposition [100, 101, 102]. Subsequently, [87] and [103] showed that it is possible to compress the parameters directly by standard tensor decompositions (in particular, CP or Tucker [104]). These decompositions, particularly of convolutional kernels,

can occasionally result in degenerate components with high intensity/Frobenius norm, so error-corrective mechanisms have since been introduced [105]. Further groundbreaking work [90, 106] demonstrated that the low-order weights can be efficiently compressed by the tensor-train decomposition [107] by first reshaping the parameters into higher-order tensors. This paradigm was later extended in two directions: (1) the tensor-train decomposition is used to compress LSTM/GRU layers in recurrent neural networks (RNNs) [108] and higher-order RNNs [109, 110]; and (2) other decompositions are explored for better compression, such as the tensor-ring decomposition [111] and block-term decomposition [112].

Existing Libraries/Algorithms. Various libraries support tensor operations. **(1) Pytorch** [113] supports specialized tensor operations commonly used in neural networks, including various convolutional layers. However, its `torch.einsum` cannot evaluate convolutional MLOs. **(2) TensorLy** [114] supports tensorization of common layer types across various platforms, including PyTorch and TensorFlow. **(3) NumPy** [115] is a general computation library that has an optimal sequencer in its einsum function, but it does not support convolutions. **(4) OptEinsum** [116] interfaces with most other einsum implementations and can further accelerate their computation via specialized BLAS routines. Due to its native implementation of netcon and path visualization capabilities, we develop conv_einsum inside this library. **(5) Einops** [117] extends the einsum functionality to GPUs. **(6) Gnetcon** [118] attempts to extend einsum to convolutions but does not support multi-way convolution. **(7) TedNet** [119] is a toolkit for computing low-rank tensor decompositions of common layer types. **(8) Tensor Comprehensions (TC)** [120] provides a language for expression for MLOs, including 2D convolutions, but it cannot easily process higher order convolutions.

3.3 Tensor Operations and einsum

In this section, we outline multi-linear operations (MLOs) common to TNNs. In brief, MLOs are nested summations over one or more indices/modes of a tensor and described in full generality in Section 3.7.1.1. Many of these operations are systematically expressible and computable via the popular notational framework and function einsum, first introduced by the Python library NumPy [115]. Therefore, we will first review the most important tensor network operations and their corresponding einsum representations. We then formally introduce TNNs.

Notations. We use lower case letters (e.g., $\boldsymbol{v}$) to denote vectors, upper case letters (e.g., $\boldsymbol{M}$) denote matrices, and curly letters (e.g., $\boldsymbol{\mathcal{T}}$) denote tensors. For a tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, we refer to a specific entry by the subscript notation $T_{i_1 i_2 \ldots i_N}$, where $1 \leq i_n \leq I_n$ for $1 \leq n \leq N$. We refer to N as the order, a single index n as a mode, and the magnitude of a mode I_n as dimension size. For example, a tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{3 \times 4 \times 5}$ is a 3^{rd} order tensor that has dimension size 4 at its second mode.

3.3.1 Representations of Multilinear Operations

The einsum function allows definitions of multilinear operations via string inputs. In this subsection, we highlight an example of a multilinear operation involving three primitive operations (*contraction*, *batch product*, *outer product*) in the language of einsum. Consider two 3^{rd}-order tensors $\boldsymbol{\mathcal{T}}^{(1)} \in \mathbb{R}^{B \times C \times I}$, $\boldsymbol{\mathcal{T}}^{(2)} \in \mathbb{R}^{A \times R \times T}$, and a multilinear operation between them: $\boldsymbol{\mathcal{T}}_{b,i,j} = \sum_{c=1}^{C} \boldsymbol{\mathcal{T}}^{(1)}_{b,c,i} \boldsymbol{\mathcal{T}}^{(2)}_{b,c,j}$.

We can denote the operation above in einsum as:

```
T = einsum('bci,bcj->bij', T1, T2)
```

where the string in the quotation mark precisely specifies the operation, which is known as an einsum string. In this string, the letter "c" indicates *contraction* since it appears in both inputs but not the output; the letter "b" symbolizes *batch product* since it appears in both inputs and the output; lastly, the letters "i" and "j" represent *outer product* as they each appears in one of the two inputs and they both appear in the output. Detailed description of einsum and these operations is provided in Section 3.7.1.1.

The einsum function allows definitions of multilinear operations via string inputs. In this subsection, we highlight an example of a multilinear operation involving three primitive operations (*contraction*, *batch product*, *outer product*) in the language of einsum. Consider two 3^{rd}-order tensors $\boldsymbol{\mathcal{T}}^{(1)} \in \mathbb{R}^{B\times C\times I}$, $\boldsymbol{\mathcal{T}}^{(2)} \in \mathbb{R}^{A\times R\times T}$, and a multilinear operation between them: $\boldsymbol{\mathcal{T}}_{b,i,j} = \sum_{c=1}^{C} \boldsymbol{\mathcal{T}}^{(1)}_{b,c,i} \boldsymbol{\mathcal{T}}^{(2)}_{b,c,j}$.

We can denote the operation above in einsum as:

```
T = einsum(bci,bcj->bij, T1, T2)
```

where the string in the quotation mark precisely specifies the operation, which is known as an einsum string. In this string, the letter "c" indicates *contraction* since it appears in both inputs but not the output; the letter "b" symbolizes *batch product* since it appears in both inputs and the output; lastly, the letters "i" and "j" represent *outer product* as they each appears in one of the two inputs and they both appear in the output. Detailed description of einsum and these operations is provided in Section 3.7.1.1.

3.3.2 From einsum to conv_einsum

While many popular libraries (e.g., NumPy, TensorFlow, PyTorch) implement einsum, none directly support convolutions, despite the ubiquity of convolutions in modern neural networks. We will generalize einsum to a meta-function conv_einsum, which can process convolutions.

Tensor convolution generalizes the convolution on vectors to higher-order tensors thereby extending the familiar linear convolution to a multi-linear operation. Given two tensors $\mathcal{T}^{(1)} \in \mathbb{R}^{X\times B\times C}$ and $\mathcal{T}^{(2)} \in \mathbb{R}^{L\times D\times E}$, we can define a convolution between the modes of dimensions X and L. The operation returns a 5^{th} order tensor $\mathcal{T} \in \mathbb{R}^{X'\times B\times C\times D\times E}$, with its entries as:

$$\mathcal{T}_{:,b,c,d,e} = \mathcal{T}^{(1)}_{:,b,c} * \mathcal{T}^{(2)}_{:,d,e}, \tag{3.1}$$

where $*$ denotes a convolution between two vectors. Note that the dimension sizes X and L can be different, and the output dimension size X' depends on the convolution type (e.g., a standard convolution yields $X' = X + L - 1$). We write Equation 3.1 in our proposed conv_einsum as

```
T = conv_einsum("xbc,ade->xbcde|x", T1, T2)
```

In this scheme, the same letter "x" is used for different modes, even if their dimension sizes may differ. Furthermore, the placement "x" to the right of the pipe-delimiter "|" indicates that conv_einsum performs convolution on the corresponding modes. We can use conv_einsum to represent multilinear operations on more than two inputs. Consider three tensors $\mathcal{X} \in \mathbb{R}^{B\times F\times S\times H\times W}$, $\mathcal{K}^{(1)} \in \mathbb{R}^{F\times G\times K\times K}$, $\mathcal{K}^{(2)} \in \mathbb{R}^{S\times T\times K\times K}$, and an operation

$$\mathcal{Y}_{b,g,t,:,:} = \sum_{f=1}^{F}\sum_{s=1}^{S} \mathcal{X}_{b,f,s,:,:} * \mathcal{K}^{(1)}_{f,g,:,:} * \mathcal{K}^{(2)}_{s,t,:,:} \tag{3.2}$$

which leads to an output tensor $\mathcal{Y} \in \mathbb{R}^{B \times G \times T \times H' \times W'}$. This is known as *interleaved group convolution* [121]. In conv_einsum, it writes as

```
T = conv_einsum(bfshw,fghw,sthw->bgthw|hw,X, K1, K2)
```

3.3.3 Compact Neural Networks via conv_einsum

In this section, we formulate various layers of compact neural networks in terms of conv_einsum. Expressions for many other layer types can be found in the Section 3.7.

Standard convolution layer. We review a popular layer type in neural networks — the standard 2D-convolutional layer. Such a layer is parameterized by a 4^{th} order tensor $\mathcal{W} \in \mathbb{R}^{T \times S \times H \times W}$, which maps a 3^{rd} order tensor $\mathcal{X} \in \mathbb{R}^{S \times H' \times W'}$ to a 3^{rd} order tensor $\mathcal{Y} \in \mathbb{R}^{T \times H' \times W'}$:

```
Y = conv_einsum(bshw,tshw->bthw|hw, X, W)
```

Note that a neural network typically computes its inputs in mini-batches, so the conv_einsum string contains an additional letter `"b"` to index examples in a mini-batch. Since convolutional layers include fully-connected layers as a special case when $H = W = 1$, we focus on designs of convolutional layers for the remainder of this subsection.

Tensorial Neural Networks. Convolutional layers motivate the importance and usage of TNNs since their structure has been shown to benefit from reshaping and tensorial decomposition. Numerous works propose to design *tensorial layers* where the (reshaped) convolution kernel $\mathcal{W}$ is factorized using tensor decompositions [2, 61, 87, 90, 103]. Our proposed conv_einsum can handle these types of designs. Here, we present two representatives of efficient tensorial convolutional layer designs based on the CP decomposition [104].

(1) In a *CP convolutional layer* [87], the kernel $\boldsymbol{\mathcal{W}}$ is factorized into 4 factors $\boldsymbol{W}^{(1)} \in \mathbb{R}^{R\times T}$, $\boldsymbol{W}^{(2)} \in \mathbb{R}^{R\times S}$, $\boldsymbol{W}^{(3)} \in \mathbb{R}^{R\times H}$, $\boldsymbol{W}^{(4)} \in \mathbb{R}^{R\times W}$ such that

```
W = conv_einsum(rt,rs,rh,rw->tshw, W1, W2, W3, W4)
```

Plugging this decomposition into the 2D-convolutional layer, we obtain the following conv_einsum output string:

```
Y = conv_einsum(bshw,rt,rs,rh,rw->bthw|hw, X, W1, W2, W3, W4)
```

(2) For a *reshaped CP convolutional layer* [2], the convolution kernel $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{T\times S\times H\times W}$ is first reshaped into a higher order tensor $\overline{\boldsymbol{\mathcal{W}}} \in \mathbb{R}^{T_1\cdots\times T_M\times S_1\cdots\times S_M\times H\times W}$ such that $T = \prod_{m=1}^{M} T_m$, $S = \prod_{m=1}^{M} S_m$, and then factorized into $(m+1)$ tensors $\boldsymbol{W}^{(m)} \in \mathbb{R}^{R\times T_m\times S_m}$ with $\boldsymbol{\mathcal{W}}^{(0)} \in \mathbb{R}^{R\times H\times W}$. For example, when $M = 3$:

```
W = conv_einsum(r(t1)(s1),r(t2)(s2),r(t3)(s3),rhw->
    (t3)(t2)(t1)(s3)(s2)(s1), W1, W2, W3, W0)
```

We can write the layer's conv_einsum string as:

```
Y = conv_einsum(b(s1)(s2)(s3)hw,r(t1)(s1),r(t2)(s2),r(t3)(s3),
    rhw->n(t1)(t2)(t3)hw, X, W1, W2, W3, W0)
```

For both layers, R is the *rank* of the CP decomposition, which controls the number of parameters (i.e., compression rate) of the layer.

A number of other works present alternative designs for efficient convolutional layers such as the *interleaved group convolution* [121] and *separable depth-wise convolution* [122]. As discussed in Section 3.7.2.1, our proposed conv_einsum also covers these specialized designs. We

also refer the reader to [2, 99] for more examples.

3.4 Algorithms

In the previous section, we presented how conv_einsum represents general multi-linear operations in (compact) neural networks. In this section, we develop a suite of algorithms to implement the conv_einsum function efficiently. We organize this section as follows: **(1)** First, we develop an algorithm to reduce a conv_einsum function with two inputs to a collection of atomic PyTorch operations, which allows us to reuse GPU-optimized functions in PyTorch to complete the computation. **(2)** Secondly, we derive an optimal sequencer which automatically decomposes a conv_einsum function with an arbitrary number of inputs into a FLOPs-minimal sequence of 2-input conv_einsum functions; **(3)** Lastly, we describe how gradient checkpointing [123] can be used to reduce the memory overhead in the backpropagation phase.

3.4.1 Atomic Operations

We can represent a conv_einsum function with two inputs via GPU-optimized PyTorch functions, specifically einsum and convNd (e.g., conv1d, conv2d). In particular, we will show that any conv_einsum function with convolution can be realized via convNd. To understand why such a transformation is possible, we first analyze the conv_einsum string for the conv1d function:

```
Y = conv_einsum(bsh,tsh->bth|h, X, W)
```

where "t" stands for the output channel, "s" the input channel, "h" the length of features/filters, and "b" the batch size. Now, we can categorize these letters in terms of primitive operations. **(1)**

The letter "h" is a convolution, appearing in both inputs and the output; **(2)** The letter "s" is a contraction, appearing in both inputs but not the output; **(3a)** The letter "t" is an outer product, appearing in the first input and the output; **(3b)** The letter "b" is another outer product, appearing in the second input and the output.

The conv1d function covers almost all mixtures of compound multi-linear operations in which each operation type appears at most once, which we refer to as an *atomic operation*, except two cases: **(4)** A batch product that appears in both inputs and the output; and **(5)** A self-contraction that appears in only one input. Fortunately, these two edge cases can be easily addressed. For **(4)**, the function conv1d supports a group-convolution option, which effectively extends to:

```
Y = conv_einsum(gtsh,bgsh->bgth|h, X, W)
```

where "g" stands for the filter group. In terms of tensor operations, it is a batch product, which appears in both inputs and the output. For **(5)**, such a letter can be eliminated by summing over the corresponding index in pre-processing.

In summary, **(1)-(5)** cover all possible types of tensor operations; the function conv1d covers **(1)-(4)** and **(5)** can be addressed via a pre-processing. Paired with index-swapping operations, conv1d can realize any conv_einsum string where each operation type appears only once.

Multiple letters with the same operation type. Now we will address the scenario where there are multiple letters with the same operation type. For example, if there are two different letters in a conv_einsum string designated for convolution, we can use conv2d instead of conv1d. Notice that conv2d realizes a conv_einsum such as:

```
Y = conv_einsum(gtshw,bgshw->bgthw|h,w, X, W)
```

where "g" stands for the filter group and "h"/"w" represent height/width of the filters/features respectively. In principle, we can use a convNd function to compute a conv_einsum function with N letters for convolution (though convNd for $N \geq 4$ requires custom implementation).

For non-convolution letters, all letters with the same type can be merged into one letter in the preprocessing (i.e., the corresponding modes are reshaped into one compound mode), and the letter is converted back to multiple letters in the post-processing step (i.e., the compound mode is reshaped into its corresponding modes).

```
import conv_einsum as cv
A=np.random.rand(4,7,9)
B=np.random.rand(10,5)
C=np.random.rand(5,4,2)
D=np.random.rand(6,8,9,2)
path_info = cv.contract_path(ijk,jl,lmq, njpq->ijknp|j,
    A, B, C, D)
print(path_info[1])
```

(a) **Tensor sequence generation.** We analyze a sequence over a collection of tensors $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}$, involving contraction, convolution, and batch product (Python). We store the optimal sequence in a string array `path_info`.

```
    Complete sequence:  ijk,jl,lmq,njpq->ijknp|j
     Naive FLOP count:  4.212e+05
 Optimized FLOP count:  2.056e+05
 Largest intermediate:  1.944e+05 elements
--------------------------------------------------------------------------------
                current                                remaining
--------------------------------------------------------------------------------
                lmq,jl->qj                             ijk,njpq,qj->ijknp|j
                qj,njpq->jnp|j                         ijk,jnp->ijknp|j
                jnp,ijk->ijknp|j                       ijknp->ijknp|j
```

(b) **An optimal sequence of paths.** Visualization (via opt-einsum) of the optimal sequence of paths for the conv_einsum string submitted in Figure 3.1a, with our modified support for convolution symbols. We are also presented with information about the naive left-to-right cost vs the cost of the suggested path.

Figure 3.1: **conv_einsum sample code.** The figure depicts the generation and analysis of a set of tensors coalesced into one tensor sequence (Python).

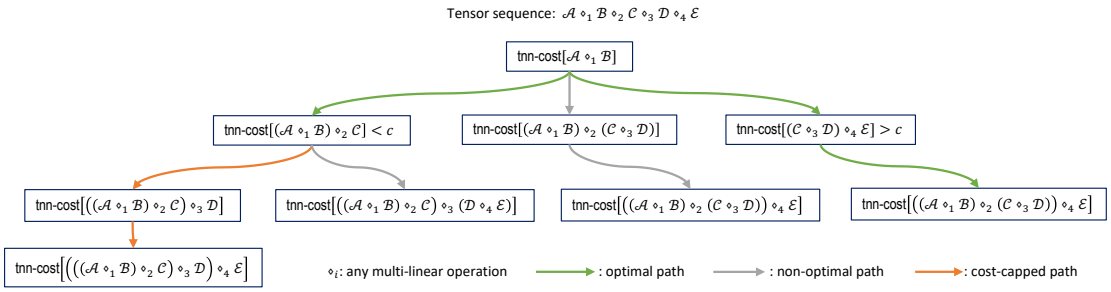


Figure 3.2: **Optimal sequencer example**. conv_einsum deploys the optimal sequencer to analyze the path tree of an abstract tensor sequence $\boldsymbol{\mathcal{A}} \circ_1 \boldsymbol{\mathcal{B}} \circ_2 \boldsymbol{\mathcal{C}} \circ_3 \boldsymbol{\mathcal{D}} \circ_4 \boldsymbol{\mathcal{E}}$, where $\circ_i$ for $1 \leq i \leq 4$ is any collection of multi-linear operations, including convolutions, batch products, contractions, and outer products. The green path indicates the optimal path and the orange path indicates a path which satisfies a user-specified cost cap c at each node.

3.4.2 Optimal Sequencer

The conv_einsum strategy for optimal sequence discovery will work with an extensive variety of decompositions and with any existing open-source einsum implementation such as `numpy.einsum` (Numpy), `tf.einsum` (TensorFlow), or opt-einsum [116], which sits on top of `NumPy`. These functions/libraries can natively handle determining the FLOPs-optimal evaluation order of tensor networks involving (non-convolutional) operations via the netcon algorithm [92]. We add support for convolutions inside the netcon algorithm; we refer to the extended algorithm as the *optimal sequencer* within our framework. We forked and modified the opt-einsum library to handle convolutions as a proof of concept. We present a high-level overview of the optimal sequencer which can be used for a `numpy.einsum` or `tf.einsum` implementation.

Better than left-to-right evaluation. netcon was designed to handle general contraction sequences in tensor networks. For example, for $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{I \times J \times K}, \boldsymbol{\mathcal{B}} \in \mathbb{R}^{J \times L}, \boldsymbol{\mathcal{C}} \in \mathbb{R}^{L \times M}$, one might be interested in the optimal order of evaluation of the tensor $\boldsymbol{\mathcal{T}} = \sum_{l=1}^{L} \sum_{j=1}^{J} \boldsymbol{\mathcal{A}}_{:,j,:} \otimes \boldsymbol{\mathcal{B}}_{j,l} \otimes \boldsymbol{\mathcal{C}}_{l,:}$.

Let us momentarily suppress the index and summation notation of this equation, i.e., let $(AB) \triangleq \sum_{j=0}^{J-1} \mathcal{A}_{:,j,:}\mathcal{B}_{j,l}$, $(BC) \triangleq \sum_{l=0}^{L-1} \mathcal{B}_{j,l}\mathcal{C}_{l,:}$. The possible paths we may take to arrive at $\boldsymbol{\mathcal{T}}$ include $(AB) \to (AB)C$ (the so-called **left-to-right** path), or $(BC) \to A(BC)$. Each of these paths has a predictable *contraction cost*, or number of multiplications/additions (FLOPs), dependent on the dimensions of the tensor modes involved in each intermediate product. netcon is designed to efficiently traverse such cost-path trees and determine the FLOPs-optimal path in a fast manner.

In particular, netcon is capable of handling all types of MLO sequences we have described except for those containing convolutions. For example, consider the tensor $\boldsymbol{\mathcal{T}}_{p,:,q,r,t} = \sum_{n=1}^{N} \boldsymbol{\mathcal{B}}_{n,p}\big(\boldsymbol{\mathcal{A}}_{n,:,r} * \boldsymbol{\mathcal{C}}_{:,q}\big)\boldsymbol{\mathcal{D}}_{r,t}$ which contains a convolution.

Our optimal sequencer extends the netcon paradigm to handle convolutions by replacing the contraction cost function with a more general `tnn-cost` function, which adds the cost (FLOPs-wise) of the convolutions (if present) within an intermediate product at each node. Further details of the logic is deferred to Section 3.7.3. Figure 3.2 depicts the optimal sequencer analyzing an abstract tensor sequence. Since einsum-like evaluation of tensorial convolutional layers was previously not possible, training speed up via optimal path evaluation of MLOs containing convolutions was difficult to explore. We now present a pair of theorems which assert the existence of cheaper-than-naive evaluation path for an RCP or RTK convolutional layer.

Theorem 25 (CP reduction)**.** *Let $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{B\times S\times H'\times W'}$ be the input to a reshaped CP (RCP) convolutional kernel $\overline{\boldsymbol{\mathcal{W}}} \in \mathbb{R}^{T_1\cdots\times T_M\times S_1\cdots\times S_M\times H\times W}$ such that $T = \prod_{m=1}^{M} T_m$, $S = \prod_{m=1}^{M} S_m$ are factored into $(m+1)$ rank-R CP factor tensors $\boldsymbol{W}^{(m)} \in \mathbb{R}^{R\times T_m\times S_m}$ with $\boldsymbol{\mathcal{W}}^{(0)} \in \mathbb{R}^{R\times H\times W}$. Assume $H' \gg H$ and $W' \gg W$ are large; in particular $SH'W' > aHW$ and $BH'W' > aS$ for some constant $a \geq 1$. Furthermore, let $R \geq S$. Then the forward pass through the RCP kernel*

(in the syntax of conv_einsum*),*

```
Y=conv_einsum(b(s1)(s2)(s3)···(sM)hw,r(t1)(s1),r(t2)(s2),
    r(t3)(s3),..., r(tM)(sM),rhw->r(t1)(t2)(t3)···(tM)hw|hw,
    X, W1, W2, W3, ..., WM, W0)
```

has a pairwise evaluation path which costs less FLOPs than the naive left-to-right evaluation,

```
Y=conv_einsum(br(t1)(t2)(t3)···(tM)hw, rhw -> br(t1)(t2)
    (t3)···(tM)hw|hw, YM, W0)
```

where

```
Ym=einsum(r(s1)(s2)···(s(m-1))(t1)(t2)···(t(m-1)),r(sm)(tm)
    ->r(s1)(s2)···(sm)(t1)(t2)···(tm), Y(m-1), Wm)
```

for $1 \leq m \leq M$*, noting that for tensor object* `YM`*, its mode symbols* `h`*,* `w` *correspond to dimensions* H', W'*, and* `b` *corresponds to an arbitrary batch size.*

Theorem 26 (Tucker reduction)**.** *Let* $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{B \times S \times H' \times W'}$ *be the input to a reshaped Tucker (RTK) convolutional kernel* $\overline{\boldsymbol{\mathcal{W}}} \in \mathbb{R}^{T_1 \cdots \times T_M \times S_1 \cdots \times S_M \times H \times W}$ *such that* $T = \prod_{m=1}^{M} T_m$*,* $S = \prod_{m=1}^{M} S_m$ *are factored into* $(m+1)$ *rank-*R_m *CP factor tensors* $\boldsymbol{W}^{(m)} \in \mathbb{R}^{R_m \times T_m \times S_m}$ *with* $\boldsymbol{\mathcal{W}}^{(0)} \in \mathbb{R}^{R \times H \times W}$ *and* $\boldsymbol{\mathcal{C}}^{(\in)}\mathbb{R}^{R_0 \times R_1 \times \cdots \times R_M}$*. Assume* $H' \gg H$ *and* $W' \gg W$ *are large; in particular* $SH'W' > aHW$ *and* $BH'W' > aS$ *for some constant* $a \geq 1$*. Furthermore, let* $\prod_{i=1}^{M} R_m \geq S$*. Then the forward pass through the RCT kernel (in the syntax of* conv_einsum*),*

```
Y=conv_einsum(b(s1)(s2)(s3)···(sM)hw,(r1)(t1)(s1),(r2)(t2)
    (s2),(r3)(t3)(s3),...,(rM)(tM)(sM),(r0)hw, (r0)(r1)(r2)···
    (rM)->b(t1)(t2)(t3)···(tM)hw|hw,X, W1, W2, W3, ..., WM, W0, C)
```

has a pairwise evaluation path which costs less FLOPs than the naive left-to-right evaluation,

```
Y=conv_einsum(br(t1)(t2)(t3)···(tM)hw, rhw -> br(t1)(t2)(t3)···
    (tM)hw|hw, YM, W0)
```

where

```
 Ym=einsum((r(m-1))(s1)(s2)···(s(m-1))(t1)(t2)···(t(m-1)),(rm)
    (sm)(tm)->r(s1)(s2)···(sm)(t1)(t2)···(tm), Y(m-1),Wm)
```

for $1 \leq m \leq M$*, noting that for tensor object* `YM`*, its mode symbols* `h`*,* `w` *correspond to dimensions* H', W'*, and* `b` *corresponds to an arbitrary batch size.*

Proof sketches. The full proof is deferred to Section 3.7. It uses an example path which sequentially reconstructs $\overline{\mathcal{W}}$ avoids the addition of a large $\mathcal{O}(H'W')$ cost to all intermediates, which occurs in the CP and Tucker naive evaluations.

3.4.3 Checkpointing

Since we evaluate a tensor network in a pairwise manner, computing a tensor network with N inputs leads to $(N-1)$ intermediate results. If we use an autograd-like function, we will need to save these intermediates in memory, causing high memory overhead. To avoid storing

intermediate products, one can use gradient checkpointing [123] which recomputes the gradient during the backward pass rather than saving all intermediate results in memory. Normally, in the forward pass, the model caches all values of the activation neurons and reuses them in the backwards pass calculation, which gradient checkpointing avoids.

3.5 Experiments

In this section, we demonstrate that conv_einsum greatly improves computational and memory efficiency of convolutional TNN training over a variety of tasks.

Tasks. **(1)** A classic two-stream convolutional neural network [124] is used for a video classification task, trained on the UCF-101 data set [125]. ResNet-101 [1] was chosen as the ConvNet for both the spatial and temporal streams, pre-trained on ImageNet [14]. The two-stream network is adapted from [126]. **(2)** An Automatic Speech Recognition task using the Conformer architecture [127], which incorporates convolution modules between the attention modules and the feed-forward modules of a Transformer model [128]. The model is trained from scratch with random normal initialization on the LibriSpeech dataset [129]. **(3)** Image classification tasks with model trained from scratch on CIFAR10 [3] and ImageNet datasets using ResNet-34.

Weight decay is set to 5×10^{-4}, momentum is set to 0.9, and learning rate is initialized as 0.05 with decay rate of 0.5 every 30 epochs. An NVIDIA GeForce RTX 2080Ti is used for all experiments. All training is implemented in PyTorch. For all RCP experiments, $M = 3$, and only the input and output channel modes are reshaped to a third order tensor.

TensorFlow vs PyTorch. Although all our training is done in PyTorch, we can reduce higher-order convolutions to atomic conv1d and conv2d operations in both PyTorch *and* Tensor-

Flow per Section 3.4. Computation of MLOs in both PyTorch and TensorFlow ultimately reduces to the same CUDA backend, so we expect performance comparisons to be similar. Furthermore, the number of FLOPs required to compute a fixed layer type is independent of the backend library and purely a function of the tensor dimensions and base types of MLOs involved.

Baselines. We compare conv_einsum-assisted training against two baselines across all tasks: **naive left-to-right** evaluation of tensorial forward passes with and without checkpointing. In particular, we compare the usage of conv_einsum to optimally evaluate tensorial forward passes against these naive implementations to demonstrate the benefits of evaluating MLOs in a FLOPs-minimal order. For all backpropagation, autograd is used.

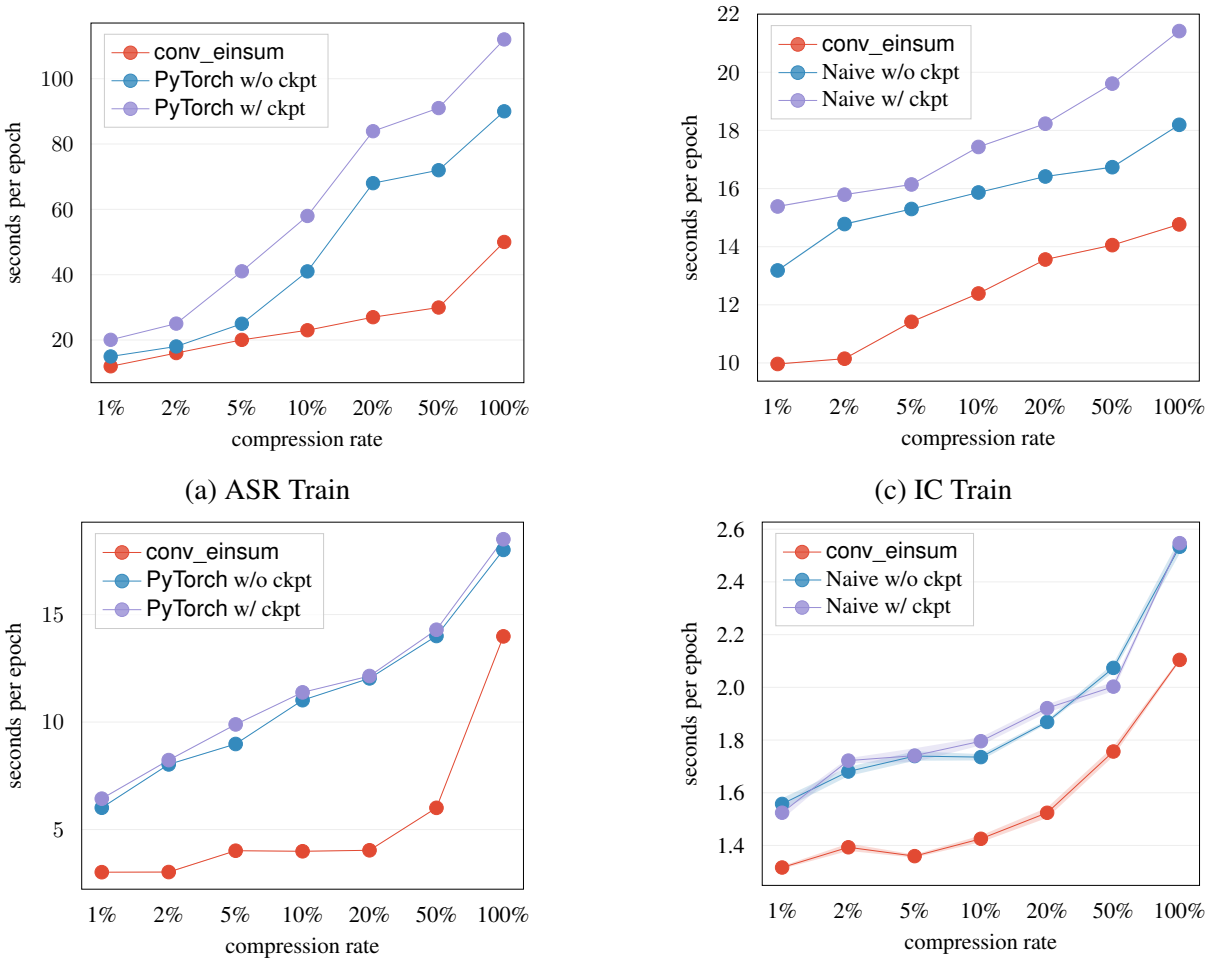
A compression rate (CR) of $x\%$ indicates that the size of each learnable layer of the TNN model $x\%$ the number of original layer. To achieve this, we first form the specified tensor decomposition of the learnable layer with rank R such that it is equal in size to the original layer. We then trim off $x\%$[1] of the least significant components, i.e., reduce the rank, until it contains $\leq x\%$ of the original parameters.

We stress that this work is not focused on improving the accuracy of TNNs, which is an orthogonal topic of research. Accuracy drop, especially due to degeneracy in tensor decompositions, is a well-known phenomenon independent of MLO evaluation. In any case, we report TNN accuracy for various tasks as supplementary experimental data in Section 3.7.5.

[1]In the case of 100% compression, the ranks of the reshaped tensors are selected so as to match the original backbone model size with no further reduction. The performance of such a TNN will not be equivalent to an untensorized counterpart despite containing the same number of parameters, since it is merely an approximation.

3.5.1 Runtime and FLOPs results

We used conv_einsum and its optimal sequencer to evaluate a tensorial forward pass in an order which incurs the minimum number of FLOPs. A naive implementation will evaluate a tensorial sequence from left to right as given. Additionally, conv_einsum uses gradient checkpointing by default to avoid memory overflow for backward passes.



(a) ASR Train

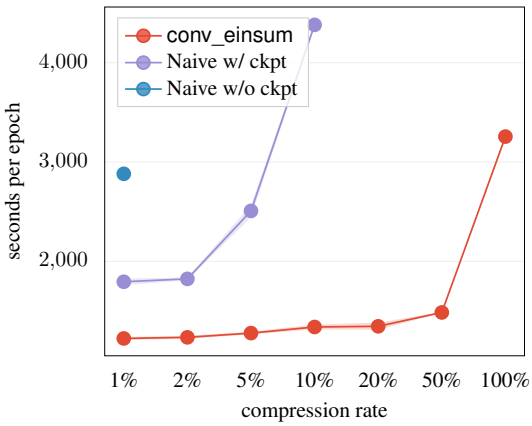
(c) IC Train

(b) ASR Test

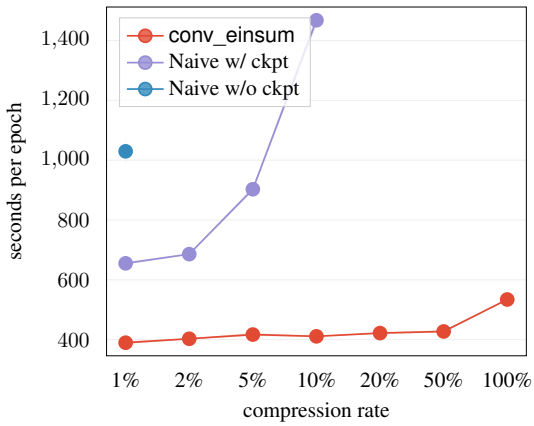
(d) IC Test

Figure 3.3: **Run-time comparison between conv_einsum and naive evaluation for image classification (IC) and automatic speech recognition (ASR) (CP-TNN) tasks**. A RCP-TNN ($M = 3$) and CIFAR-10 is used for IC while a CP-TNN and LibriSpeech dataset isused for ASR. 'ckpt' denotes checkpointing. Runtimes are averaged over 3 runs.

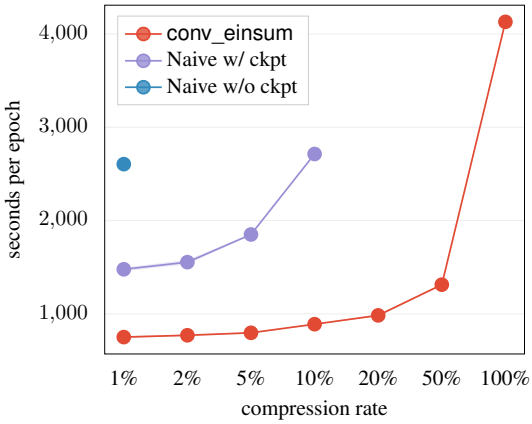
(1) The optimal sequencer used in conv_einsum significantly improves the runtime effi-



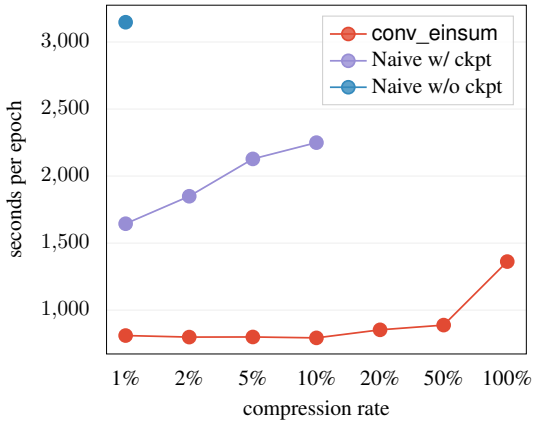
(a) Train (spatial)



(c) Train (temporal)



(b) Test (spatial)



(d) Test (temporal)

Figure 3.4: **Run-time comparison between conv_einsum and naive left-to-right PyTorch implementations for a video classification machine learning task** An RCP-TNN ($M = 3$) is trained on the UCF-101 dataset. All tests were run using the maximum allowable batch size. Naive w/ ckpt only ran without memory overflow for compression rates 1% - 10% and naive w/o ckpt was only able to run for compression rate 1%. 'ckpt' denotes checkpointing. Averaged over 3 runs.

Table 3.1: **Run-time (minutes per epoch) comparison between conv_einsum and PyTorch for ImageNet Classiciation w/ checkpointing.** We use a RCP ($M = 3$) ResNet-34 architecture and batch size of 256. Left-to-right evaluation runs into memory overflow without checkpointing. conv_einsum significantly improves training time. Runtimes are averaged over 3 runs.

	conv_einsum		Naive w/ ckpt	
CR	Train	Test	Train	Test
5%	21.8	1.06	32.4	1.42
10%	23.4	1.47	35.7	1.77
20%	28.9	1.95	39.9	2.35
50%	34.1	2.54	50.1	3.54
100%	41.6	3.08	65.6	4.08

ciency of training and test in TNNs. In Figures 3.3,3.4 and Table 3.1, we compare the training and test times between conv_einsum and naive left-to-right (with and w/o checkpointing) implementations over a range of model scales and decompositions. The IC and VC tasks use RCP ($M = 3$) decompositions, while the ASR task uses a standard CP decomposition. In the VC task, for each model size, we use the maximal allowable batch size, while in the ASR task, we use the same batch size across scales. We observe that when memory is the bottleneck of a task (such as VC), checkpointing helps accelerate the runtime by allowing more batches. On the other hand, when the batch sizes are the same (as in ASR), checkpointing trades computational complexity for space, thus increasing the overall runtime. In either scenario, conv_einsum achieves the fastest runtimes.

(2) conv_einsum improves computation through weight tensors of differing sizes and decompositions. conv_einsum serves as a general solution for improving efficiency and is tensor-structure-agnostic. The networks we have experimented with contain weights of vastly differing sizes. Figures 3.3 and 3.4 demonstrate that conv_einsum exhibits competitive results against standard left-to-right implementation (with or without) checkpointing over various tensor scales.

Furthermore, Tables 3.5 and 3.6 in Section 3.7 shows the results of our conv_einsum training using different forms of tensor decomposition in both a high-performance (GPU) and low-performance (CPU) environment. We observe that conv_einsum outperforms naive implementation with/without checkpointing in all cases.

(3) conv_einsum greatly reduces the number of FLOPs needed to process tensorial convolutional layers. Given a forward pass through a tensorial convolutional layer expressible as a conv_einsum string, an optimal evaluation path can be determined. Evaluation of an MLO sequence from left-to-right will result in an excessive amount of FLOPs as predicted by Theorem 25 and shown in Table 3.2, which lists the number FLOPs saved by optimal path evaluation through CP convolutional layer blocks of ResNet-34.

Table 3.2: **FLOPs per CP convolutional layer in ResNet-34.** We calculate the FLOPs incurred by forward passes through CP convolutional layers (CR$=100\%$), using a batch size of 128. Layer nomenclature and associated kernel structure correspond with architecture description in [1].

Layer	Left-to-Right	conv_einsum	Speedup$\times$
conv1	1.51×10^{15}	3.87×10^{13}	3.90
conv2_x	1.38×10^{14}	3.09×10^{13}	4.47
conv3_x	5.85×10^{13}	9.67×10^{12}	6.05
conv4_x	6.03×10^{13}	3.71×10^{12}	16.25
conv5_x	4.25×10^{13}	4.72×10^{11}	90.04

3.5.2 Memory Results

Naive evaluation of tensorial forward passes suffers from high intermediary memory costs during training. conv_einsum significantly reduces these costs. In TNNs, during training computation of some intermediate data objects can be prohibitively large and thus difficult to fit into

Table 3.3: **Maximum batch size for a speech and video task.** The maximal batch size allowed for data under varying compression rates and using different libraries on **(1)** an ASR on LibriSpeech and **(2)** a VC task for spatial (S) and temporal (T) streams of a two-stream network on UCF-101. conv_einsum permits larger batch sizes.

Automatic speech recognition task

CR	conv_einsum	Naive w/ ckpt	Naive w/o ckpt
1%	14	8	6
2%	14	8	6
5%	12	6	4
10%	10	4	2
20%	8	2	0
50%	6	2	0
100%	4	1	0

Video classification task

	conv_einsum		Naive w/ ckpt		Naive w/o ckpt	
CR	S	T	S	T	S	T
1%	20	30	2	4	1	2
2%	20	30	2	4	0	0
5%	20	30	2	4	0	0
10%	18	30	1	2	0	0
20%	16	27	0	0	0	0
50%	12	22	0	0	0	0
100%	4	14	0	0	0	0

memory. In Table 3.3, we present the maximal batch size allowed for two large-scale tasks under different compression rates: video classification (VC) and automatic speech recognition (ASR). We observe that if the size of a TNN matches the original backbone model size (i.e., CR=100%), the maximal allowed batch size in a standard PyTorch implementation is 0 without checkpointing. Even if we compress the model to 1% of the original number of parameters, the maximal allowed batch size is still limited, making computation infeasible or slow. We theorize that optimal evaluation order results in uniformly smaller intermediate products.

3.6 Conclusions and Discussions

In this work, we introduce an einsum-like notational framework and meta-algorithm conv_einsum, which is capable of expressing and efficiently computing passes through convolutional TNNs. conv_einsum-assisted training is competitive against, and in many cases, superior to standard PyTorch TNN training. For future work, we plan to accelerate training and test times through

incorporation of parallel computation libraries such as TensorRT. Additionally, our experiments do not make use of low-rank decomposition stabilizing algorithms [100, 105] which improve the quality of factorized components. Combining conv_einsum with such methods could result in exceptionally light, accurate, and fast TNNs.

3.7 Supplemental Material

3.7.1 Examples of primitive operations

Contraction. *Tensor contraction* generalizes matrix multiplication to higher-order tensors. For instance, given two 3^{rd} order tensors $\mathcal{T}^{(1)} \in \mathbb{R}^{A\times B\times C}, \mathcal{T}^{(2)} \in \mathbb{R}^{A\times D\times E}$, we can define a contraction between the modes with shared dimension size A. The operation returns a 4^{th} order tensor $\mathcal{T} \in \mathbb{R}^{B\times C\times D\times E}$ with its entries calculated as: $\mathcal{T}_{b,c,d,e} = \sum_{a=1}^{A} \mathcal{T}^{(1)}_{a,b,c} \cdot \mathcal{T}^{(2)}_{a,d,e}$. This contraction would be submitted to einsum as:

```
T = einsum(abc,ade->bcde, T1, T2)
```

Here, the modes are ordered and represented by concatenated letters (which can be case-sensitive). These sub-strings, corresponding to each input tensor, are separated by commas and lie to the left of the arrow. The output, which lies to the right of the arrow, concatenates all mode letters in a single string sans the letter "a" to indicate that a contraction must occur over these modes. The remaining parameters correspond to the ordered set of tensors. This method will fail if the modes denoted by the same letters do not share dimension sizes. So a contraction corresponds to a letter that appears in all inputs, but not the output.

Outer product. *Tensor outer product* generalizes outer product to higher-order tensors.

For instance, an outer product of two 3rd order tensors $\mathcal{T}^{(1)} \in \mathbb{R}^{A\times B\times C}, \mathcal{T}^{(2)} \in \mathbb{R}^{D\times E\times F}$ returns a 6th order tensor $\mathcal{T} \in \mathbb{R}^{A\times B\times C\times D\times E\times F}$. The entries of $\mathcal{T}$ are calculated as: $\mathcal{T}_{a,b,c,d,e,f} = \mathcal{T}^{(1)}_{a,b,c} \cdot \mathcal{T}^{(2)}_{d,e,f}$. In the language of einsum, we write this as:

```
T = einsum(abc,def->abcdef, T1, T2)
```

Notice that a letter for the outer product only appears in one of the inputs and in the output.

Batch product. *Tensor batch product* is a variation of the outer product. Given tensors $\mathcal{T}^{(1)} \in \mathbb{R}^{A\times B\times C}, \mathcal{T}^{(2)} \in \mathbb{R}^{C\times E\times D}$, the batch product over the dimensional-shared mode C returns a 5th order tensor $\mathcal{T} \in \mathbb{R}^{A\times B\times C\times D\times E}$. The entries of $\mathcal{T}$ are calculated as: $\mathcal{T}_{a,b,c,d,e} = \mathcal{T}^{(1)}_{a,b,c} \cdot \mathcal{T}^{(2)}_{a,d,e}$. The einsum implementation is

```
T = einsum(abc,cde->abcde, T1, T2)
```

Notice that the letter corresponding to the batch product appears in both inputs and the output.

In this section, we will outline the multi-linear operations covered in preliminaries in full generality along with their conv_einsum representations and other tensor decompositions of TNNs. Much of this content adapts and follows the notation of Su et al. [2].

3.7.1.1 Fully General Multilinear operations

Multi-operations among multiple tensors. We can simultaneously perform a series of multi-linear operations among a group of tensors. Let $\mathcal{T}^{(1)} \in \mathbb{R}^{I\times T\times S}$, $\mathcal{T}^{(2)} \in \mathbb{R}^{J\times R\times T}$, and $\mathcal{T}^{(3)} \in \mathbb{R}^{K\times S\times R}$. We can define a simultaneous contraction on modes with dimension sizes R, S, and T. This simultaneous contraction returns a 3rd order tensor $\mathcal{T} \in \mathbb{R}^{I\times J\times K}$, with its entries

computed as

$$\boldsymbol{\mathcal{T}}_{i,j,k} = \sum_{r=1}^{R}\sum_{s=1}^{S}\sum_{t=1}^{T} \boldsymbol{\mathcal{T}}^{(1)}_{i,t,s}\boldsymbol{\mathcal{T}}^{(2)}_{j,r,t}\boldsymbol{\mathcal{T}}^{(3)}_{k,s,r} \tag{3.3}$$

Equivalently, via conv_einsum, we can write it as:

```
T = conv_einsum(its,jrt,ksr->ijk, T1, T2, T3)
```

Below we outline a simpler example that performs multiple operations between two tensors.

Operation	Definition	conv_einsum
mode-(k,l) Tensor Contraction	$\boldsymbol{\mathcal{T}}_{i_0,\cdots,i_{k-1},i_{k+1},\cdots,i_{m-1},j_0,\cdots,j_{l-1},j_{l+1},\cdots,j_{n-1}}$ $= \langle \boldsymbol{\mathcal{T}}^{(0)}_{i_0,\cdots,i_{k-1},:,i_{k+1},\cdots,i_{m-1}}, \boldsymbol{\mathcal{T}}^{(1)}_{j_0,\cdots,j_{l-1},:,j_{l+1},\cdots,j_{n-1}} \rangle$	$I_0\cdots I_{k-1}I_k\cdots I_{m-1}, J_0\cdots J_{l-1}J_l\cdots J_{n-1}$ $\rightarrow I_1 I_{k-1} I_{k+1} I_{m-1} J_0\cdots J_{l-1}J_{l+1}\cdots J_{n-1}$
mode-(k,l) Tensor Convolution	$\boldsymbol{\mathcal{T}}^{(i_0,\cdots,i_{k-1},:,i_{k+1},\cdots,i_{m-1},j_0,\cdots,j_{l-1},j_{l+1},\cdots,j_{n-1})}_{2}$ $= \boldsymbol{\mathcal{T}}^{(0)}_{i_0,\cdots,i_{k-1},:,i_{k+1},\cdots,i_{m-1}} * \boldsymbol{\mathcal{T}}^{(1)}_{j_0,\cdots,j_{l-1},:,j_{l+1},\cdots,j_{n-1}}$	$I_0\cdots I_{k-1}C\cdots I_{m-1}, J_0\cdots J_{l-1}C\cdots J_{n-1}$ $\rightarrow I_1 I_{k-1} C I_{k+1} I_{m-1} J_0\cdots J_{l-1}J_{l+1}\cdots J_{n-1} \mid C$
mode-(k,l) Tensor Batch Product	$\boldsymbol{\mathcal{T}}_{i_0,\cdots,i_{k-1},r,i_{k+1},\cdots,i_{m-1},j_0,\cdots,j_{n-1}}$ $= \boldsymbol{\mathcal{T}}^{(0)}_{i_0,\cdots,i_{k-1},r,i_{k+1},\cdots,i_{m-1}} \boldsymbol{\mathcal{T}}^{(1)}_{j_0,\cdots,j_{l-1},r,j_{l+1},\cdots,j_{n-1}}$	$I_0\cdots I_{k-1}C\cdots I_{m-1}, J_0\cdots J_{l-1}J_l\cdots J_{n-1}$ $\rightarrow I_1 I_{k-1} I_k I_{k+1} I_{m-1} J_0\cdots J_{l-1}J_{l+1}\cdots J_{n-1}$
Tensor Outer Product	$\boldsymbol{\mathcal{T}}^{(0)}_{i_0,\cdots,i_{k-1},i_k,i_{k+1},\cdots,i_{m-1},j_0,\cdots,j_{l-1},j_l,j_{l+1},\cdots,j_{n-1}}$ $= \boldsymbol{\mathcal{T}}^{(1)}_{i_0,\cdots,i_{m-1}} * \boldsymbol{\mathcal{Y}}_{j_0,\cdots,j_{l+1},\cdots,j_{n-1}}$	$I_0\cdots I_{m-1}, J_0\cdots J_{n-1}$ $\rightarrow I_1\cdots I_{m-1}J_0\cdots J_{n-1}$

Table 3.4: **Primitive tensor operations**. $\boldsymbol{\mathcal{T}}^{(0)} \in \mathbb{R}^{I_0\times\cdots\times I_{m-1}}$, $\boldsymbol{\mathcal{T}}^{(1)} \in \mathbb{R}^{J_0\times\cdots\times J_{n-1}}$ are the input tensors and $\boldsymbol{\mathcal{T}}$ is the output tensor. Note that both mode-(I_k, J_l) tensor contraction, mode-(I_k, J_l) tensor batch product are legal only if $I_k = J_l$. The outer product can be performed with any two tensors. Red highlighted modes are owned by $\boldsymbol{\mathcal{T}}^{(0)}$ while blue highlighted modes belong to $\boldsymbol{\mathcal{T}}^{(1)}$.

Tensor contraction Given tensors $\boldsymbol{\mathcal{T}}^{(0)} \in \mathbb{R}^{I_0\times\cdots I_k\times I_{m-1}}$, $\boldsymbol{\mathcal{T}}^{(1)} \in \mathbb{R}^{J_0\times\cdots J_l\cdots\times J_{n-1}}$, the mode-$(k,l)$ contraction returns an order $(m+n-2)$ tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{I_0\times\cdots I_{k-1}\times I_{k+1}\times\cdots\times I_{m-1}\times J_0\times\cdots\times J_{l-1}\times J_{l+1}\times\cdots\times J_{n-1}}$. The entries of $\boldsymbol{\mathcal{T}}$ are calculated as follows:

$$\begin{aligned}
&\boldsymbol{\mathcal{T}}_{i_0,\cdots,i_{k-1},i_{k+1},\cdots,i_{m-1},j_0,\cdots,j_{l-1},j_{l+1},\cdots,j_{n-1}} \\
&= \sum_{r=1}^{I_k-1} \boldsymbol{\mathcal{T}}^{(0)}_{i_0,\cdots,i_{k-1},r,i_{k+1},\cdots,i_{m-1}} \cdot \boldsymbol{\mathcal{T}}^{(1)}_{j_0,\cdots,j_{l-1},r,j_{l+1},\cdots,j_{n-1}} \\
&= \langle \boldsymbol{\mathcal{T}}^{(0)}_{i_0,\cdots,i_{k-1},:,i_{k+1},\cdots,i_{m-1}}, \boldsymbol{\mathcal{T}}^{(1)}_{j_0,\cdots,j_{l-1},:,j_{l+1},\cdots,j_{n-1}} \rangle.
\end{aligned}$$

Tensor Convolution Given tensors $\boldsymbol{\mathcal{T}}^{(0)} \in \mathbb{R}^{I_0 \times \cdots \times I_{m-1}}, \boldsymbol{\mathcal{T}}^{(1)} \in \mathbb{R}^{J_0 \times \cdots \times J_{n-1}}$, the mode-$(k,l)$ convolution returns an order $(m+n-1)$ tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{I_0 \times \cdots I'_k \times \cdots \times I_{m-1} \times J_0 \times \cdots \times J_{l-1} \times J_{l+1} \times \cdots \times J_{n-1}}$. For any convolution operator * the entries of $\boldsymbol{\mathcal{T}}$ are calculated as follows:

$$
\begin{aligned}
&\boldsymbol{\mathcal{T}}_{i_0,\cdots,i_{k-1},:,i_{k+1},\cdots,i_{m-1},j_0,\cdots,j_{l-1},j_{l+1},\cdots,j_{n-1}} \\
&= \boldsymbol{\mathcal{T}}^{(0)}_{i_0,\cdots,i_{k-1},:,i_{k+1},\cdots,i_{m-1}} * \boldsymbol{\mathcal{T}}^{(1)}_{j_0,\cdots,j_{l-1},:,j_{l+1},\cdots,j_{n-1}} \\
&= \boldsymbol{\mathcal{T}}^{(1)}_{j_0,\cdots,j_{l-1},:,j_{l+1},\cdots,j_{n-1}} \bar{*} \boldsymbol{\mathcal{T}}^{(0)}_{i_0,\cdots,i_{k-1},:,i_{k+1},\cdots,i_{m-1}}
\end{aligned}
$$

Here we have intentionally left $*$ and the dimension of the k-th mode I'_k ambiguous, as it will vary depending on the type of convolution specified by the user. For example, with max padding, we have that $I'_k = \max\{I_k, J_l\}$, and with same-padding we have $I'_k = I_k$.

Tensor Batch Product Given tensors $\boldsymbol{\mathcal{T}}^{(0)} \in \mathbb{R}^{I_0 \times \cdots I_k \cdots \times I_{m-1}}, \boldsymbol{\mathcal{T}}^{(1)} \in \mathbb{R}^{J_0 \times \cdots I_l \cdots \times J_{n-1}}$, the mode-$(k,l)$ batch product returns an order $(m+n-1)$ tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{I_0 \times \cdots I_k \times \cdots \times I_{m-1} \times J_0 \times \cdots \times \cdots \times J_{n-1}}$. The entries of $\boldsymbol{\mathcal{T}}$ are calculated as follows:

$$
\begin{aligned}
&\boldsymbol{\mathcal{T}}_{i_0,\cdots,i_{k-1},r,i_{k+1},\cdots,i_{m-1},j_0,\cdots,j_{l-1},j_{l+1},\cdots,j_{n-1}} \\
&= \boldsymbol{\mathcal{T}}^{(0)}_{i_0,\cdots,i_{k-1},r,i_{k+1},\cdots,i_{m-1}} \boldsymbol{\mathcal{T}}^{(1)}_{j_0,\cdots,j_{l-1},r,j_{l+1},\cdots,j_{n-1}}
\end{aligned}
$$

Tensor Outer Product Given tensors $\boldsymbol{\mathcal{T}}^{(0)} \in \mathbb{R}^{I_0 \times \cdots \times I_{m-1}}, \boldsymbol{\mathcal{T}}^{(1)} \in \mathbb{R}^{J_0 \times \cdots \times J_{n-1}}$, the outer product returns an order $(m+n)$ tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{I_0 \times \cdots I_k \times \cdots \times I_{m-1} \times J_0 \times \cdots \times J_{n-1}}$. The entries of $\boldsymbol{\mathcal{T}}$ are calculated as follows:

$$
\boldsymbol{\mathcal{T}}_{i_0,\cdots,i_{m-1},j_0,\cdots,j_{n-1}} = \boldsymbol{\mathcal{T}}^{(0)}_{i_0,\cdots,i_{m-1}} \boldsymbol{\mathcal{T}}^{(1)}_{j_0,\cdots,j_{n-1}}. \tag{3.4}
$$

Table 3.4 summarizes these multi-linear operations along with their conv_einsum input representations.

3.7.2 Tensorial Neural Networks

CP decomposition [104]. (1a) In a *CP convolutional layer* [87], the original kernel $\mathcal{W} \in \mathbb{R}^{T\times S\times H\times W}$ is CP factorized as:

```
W = conv_einsum(rt,rs,rh,rw->tshw, W1, W2, W3, W4).
```

As a result, the layer is parameterized by 4 weight matrices $\boldsymbol{W}^{(1)} \in \mathbb{R}^{R\times T}$, $\boldsymbol{W}^{(2)} \in \mathbb{R}^{R\times S}$, $\boldsymbol{W}^{(3)} \in \mathbb{R}^{R\times H}$, $\boldsymbol{W}^{(4)} \in \mathbb{R}^{R\times W}$. We can write this layer in conv_einsum as

```
Y = conv_einsum(bshw,rt,ts,rh,rw->bthw|hw, X, W1)
```

(1b) In a *reshaped CP convolutional layer* [2], the original kernel $\mathcal{W}$ is first reshaped as $\overline{\mathcal{W}} \in \mathbb{R}^{T_1\cdots\times T_M\times S_1\cdots\times S_M\times H\times W}$ such that $T = \prod_{m=1}^{M} T_m$ and $S = \prod_{m=1}^{M} S_m$. Suppose $M = 3$, then the reshaped kernel is factorized by a CP decomposition as

```
W = conv_einsum(r(t1)(s1),r(t2)(s2),r(t3)(s3),rhw->(t1)(t2)(t3)
    (s1)(s2)(s3)hw, W1, W2, W3, W0).
```

As a result, the layer is parameterized by $(M+1)$ weight tensors $\mathcal{W}^{(m)} \in \mathbb{R}^{R\times T_m\times S_m}$ and $\mathcal{W}^{(0)} \in \mathbb{R}^{R\times H\times W}$. We can write the layer in conv_einsum as

```
Y = conv_einsum(b(s1)(s2)(s3)hw,r(t1)(s1),r(t2)(s2),r(t3)(s3),
    rhw->b(t1)(t2)(t3)hw|hw, X, W1, W2, W3, W0)
```

Tucker (TK) decomposition [104]. (2a) In a *TK convolutional layer* [87], the original kernel $\mathcal{W} \in \mathbb{R}^{T \times S \times H \times W}$ is factorized by TK as:

```
W = conv_einsum((r1)t,(r2)s,(r1)(r2)hw->tshw, W1, W2, W0)
```

Consequently, the layer has 3 weight tensors $\boldsymbol{W}^{(1)} \in \mathbb{R}^{R_1 \times T}$, $\boldsymbol{W}^{(2)} \in \mathbb{R}^{R_2 \times S}$, $\mathcal{W}^{(0)} \in \mathbb{R}^{R_1 \times R_2 \times H \times W}$ as parameters. We can write the layer in conv_einsum as

```
Y = conv_einsum(bshw,(r1)t,(r2)s,(r1)(r2)hw->bthw|hw,
    X, W1, W2, W0)
```

(2b) In a *reshaped TK convolutional layer* [2], the original kernel $\mathcal{W}$ is first reshaped as $\overline{\mathcal{W}} \in \mathbb{R}^{T_1 \cdots \times T_M \times S_1 \cdots \times S_M \times H \times W}$ such that $T = \prod_{m=1}^{M} T_m$ and $S = \prod_{m=1}^{M} S_m$. Suppose $M = 3$, the reshaped kernel is then factorized by a TK decomposition as

```
W = conv_einsum((r1)(t1)(s1),(r2)(t2)(s2),(r3)(t3)(s3),(r0)hw,
    (r0)(r1)(r2)(r3)->(t1)(t2)(t3)(s1)(s2)(s3)hw,
    W1, W2, W3, W0, C)
```

Therefore, the layer has $(M+2)$ weight tensors $\mathcal{W}^{(m)} \in \mathbb{R}^{R \times T_m \times S_m}$, $\mathcal{W}^{(0)} \in \mathbb{R}^{R \times H \times W}$, $\mathcal{C} \in \mathbb{R}^{R_0 \times R_1 \times R_2 \times R_3}$. We can write the layer in conv_einsum as

```
Y = conv_einsum(b(s1)(s2)(s3)hw,(r1)(t1)(s1),(r2)(t2)(s2),(r3)
(t3)(s3),(r0)hw,(r0)(r1)(r2)(r3)->b(t1)(t2)(t3)hw|hw,
X, W1, W2, W3, W0, C)
```

Tensor-Train (TT) decomposition [107]. (3a) In a *TT convolutional layer*, the original kernel $\mathcal{W} \in \mathbb{R}^{T \times S \times H \times W}$ is factorized by TT as:

```
W = conv_einsum((r1)t,(r1)(r2)h,(r2)(r3)w,(r3)s->tshw,
    W1, W2, W3, W4)
```

Consequently, the layer has 4 weight tensors $\boldsymbol{W}^{(1)} \in \mathbb{R}^{R_1 \times T}$, $\boldsymbol{W}^{(2)} \in \mathbb{R}^{R_1 \times R_2 \times H}$, $\boldsymbol{\mathcal{W}}^{(3)} \in \mathbb{R}^{R_2 \times R_3 \times W}$, and $\boldsymbol{\mathcal{W}}^{(4)} \in \mathbb{R}^{R_3 \times S}$ as parameters. We can write the layer in conv_einsum as

```
Y = conv_einsum(bshw,(r1)t,(r1)(r2)h,(r2)(r3)w,(r3)s->bthw|hw,
    X, W1, W2, W3, W4).
```

(3b) In a *reshaped TT convolutional layer* [90], the original kernel $\boldsymbol{\mathcal{W}}$ is first reshaped as $\overline{\boldsymbol{\mathcal{W}}} \in \mathbb{R}^{T_1 \cdots \times T_M \times S_1 \cdots \times S_M \times H \times W}$ such that $T = \prod_{m=1}^{M} T_m$ and $S = \prod_{m=1}^{M} S_m$. Suppose $M = 3$, the reshaped kernel is then factorized by a TT decomposition as

```
W = conv_einsum((r1)(t1)(s1),(r1)(r2)(t2)(s2),(r2)(r3)(t3)(s3),
    (r3)hw->(t1)(t2)(t3)(s1)(s2)(s3)hw,
    W1, W2, W3, W0).
```

Therefore, the layer has $(M+1)$ weight tensors $\boldsymbol{\mathcal{W}}^{(1)} \in \mathbb{R}^{R_1 \times T_1 \times S_1}$, $\boldsymbol{\mathcal{W}}^{(m)} \in \mathbb{R}^{R_{m-1} \times R_m \times T_m \times S_m}$, and $\boldsymbol{\mathcal{W}}^{(0)} \in \mathbb{R}^{R_3 \times H \times W}$. The layer in conv_einsum is

```
Y = conv_einsum(b(s1)(s2)(s3)hw,(r1)(t1)(s1),(r1)(r2)(t2)(s2),
    (r2)(r3)(t3)(s3),(r3)hw->b(t1)(t2)(t3)hw|hw,
    X, W1, W2, W3, W0).
```

Tensor-Ring (TR) decomposition [111]. (4a) In a *TR convolutional layer*, the original kernel $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{T \times S \times H \times W}$ is factorized by TR as:

```
W = conv_einsum((r0)(r1)t,(r1)(r2)h,(r2)(r3)w,(r3)(r0)s->tshw,
    W1, W2, W3, W4)
```

Consequently, the layer has 4 weight tensors $\boldsymbol{W}^{(1)} \in \mathbb{R}^{R_1 \times T}$, $\boldsymbol{W}^{(2)} \in \mathbb{R}^{R_0 \times R_1 \times R_2 \times H}$, $\boldsymbol{\mathcal{W}}^{(3)} \in \mathbb{R}^{R_2 \times R_3 \times W}$, and $\boldsymbol{\mathcal{W}}^{(4)} \in \mathbb{R}^{R_3 \times R_0 \times S}$ as parameters. We can write the layer as

```
Y = conv_einsum(bshw,(r0)(r1)t,(r1)(r2)h,(r2)(r3)w,
    (r3)(r0)s->bthw|hw, X, W1, W2, W3, W4)
```

(4b) In a *reshaped TR convolutional layer* [2], the original kernel $\boldsymbol{\mathcal{W}}$ is first reshaped as $\overline{\boldsymbol{\mathcal{W}}} \in \mathbb{R}^{T_1 \cdots \times T_M \times S_1 \cdots \times S_M \times H \times W}$ such that $T = \prod_{m=1}^{M} T_m$ and $S = \prod_{m=1}^{M} S_m$. Suppose $M = 3$, the reshaped kernel is then factorized by a TR decomposition as

```
W = conv_einsum((r0)(r1)(t1)(s1),(r1)(r2)(t2)(s2),
        (r2)(r3)(t3)(s3),(r3)(r0)hw->(t1)(t2)(t3)(s1)(s2)(s3)hw,
        W1, W2, W3, W0)
```

Therefore, the layer has $(M+1)$ weight tensors $\boldsymbol{\mathcal{W}}^{(m)} \in \mathbb{R}^{R_{m-1} \times R_m \times T_m \times S_m}$, and $\boldsymbol{\mathcal{W}}^{(0)} \in \mathbb{R}^{R_3 \times R_0 \times H \times W}$. The layer in conv_einsum is

```
Y = conv_einsum(b(s1)(s2)(s3)hw,(r0)(r1)(t1)(s1),
    (r1)(r2)(t2)(s2),(r2)(r3)(t3)(s3),(r3)(r0)hw
    ->b(t1)(t2)(t3)hw|hw, X, W1, W2, W3, W0)
```

Block-Term (BT) decomposition [112]. In a *reshaped BT convolutional layer*, $\overline{\boldsymbol{\mathcal{W}}} \in \mathbb{R}^{T_1 \cdots \times T_M \times S_1 \cdots \times S_M \times H \times W}$ such that $T = \prod_{m=1}^{M} T_m$ and $S = \prod_{m=1}^{M} S_m$. Suppose $M = 3$, the reshaped kernel is then factorized by a BT decomposition as

```
\textcolor{blue}{W = conv_einsum(r(r1)(t1)(s1),r(r2)(t2)(s2),
    r(r3)(t3)(s3),r(r0)hw,r(r1)(r2)(r3)(r0)->
        (t1)(t2)(t3)(s1)(s2)(s3)hw, W1, W2, W3, W0, C)}
```

Consequently, the layer has $(m+1)$ weight tensors $\mathcal{W}^{(1)} \in \mathbb{R}^{R\times R_1\times T_1\times S_1}$, $\mathcal{W}^{(2)} \in \mathbb{R}^{R\times R_2\times T_2\times S_2}$, $\mathcal{W}^{(3)} \in \mathbb{R}^{R\times R_3\times T_3\times S_3}$, $\mathcal{W}^{(0)} \in \mathbb{R}^{R\times R_0\times H\times W}$, and $\mathcal{C} \in \mathbb{R}^{R\times R_1\times R_2\times R_3\times R_0}$. The layer in conv_einsum is

```
Y = conv_einsum(b(s1)(s2)(s3)hw,r(r1)(t1)(s1),r(r2)(t2)(s2),
    r(r3)(t3)(s3),r(r0)hw,r(r1)(r2)(r3)(r0)->
    b(t1)(t2)(t3)hw|hw, X, W1, W2, W3, W0, C)
```

Hierarchical-Tucker (HT) decomposition [130]. In a *reshaped HT convolutional layer*, $\overline{\mathcal{W}} \in \mathbb{R}^{T_1\cdots\times T_M\times S_1\cdots\times S_M\times H\times W}$ such that $T = \prod_{m=1}^{M} T_m$ and $S = \prod_{m=1}^{M} S_m$. Suppose $M = 3$, the reshaped kernel is then factorized by a HT decomposition as

```
W = conv_einsum((r1)(t1)(s1),(r2)(t2)(s2),(r3)(t3)(s3),(r0)hw,
    (r1)(r2)(r4),(r3)(r0)(r5),(r4)(r5)->
    (t1)(t2)(t3)(s1)(s2)(s3)hw, W1, W2, W3, W0, C1, C2, C3)
```

Consequently, the layer has $(2m+1)$ weight tensors $\mathcal{W}^{(1)} \in \mathbb{R}^{R_1\times T_1\times S_1}$, $\mathcal{W}^{(2)} \in \mathbb{R}^{R_2\times T_2\times S_2}$, $\mathcal{W}^{(3)} \in \mathbb{R}^{R_3\times T_3\times S_3}$, $\mathcal{W}^{(0)} \in \mathbb{R}^{R_0\times H\times W}$, $\mathcal{C}^{(1)} \in \mathbb{R}^{R_1\times R_2\times R_4}$, $\mathcal{C}^{(2)} \in \mathbb{R}^{R_3\times R_0\times R_5}$, and $\mathcal{C}^{(3)} \in \mathbb{R}^{R_4\times R_5}$. The layer in conv_einsum is

```
Y = conv_einsum(b(s1)(s2)(s3)hw,(r1)(t1)(s1),(r2)(t2)(s2),
    (r3)(t3)(s3),(r0)hw,(r1)(r2)(r4),(r3)(r0)(r5),(r4)(r5)
    ->b(t1)(t2)(t3)hw|hw, X, W1, W2, W3, W0, C1, C2, C3)
```

3.7.2.1 Efficient Convolutional Layers

A number of works design efficient convolutional layers by modifying the linear operations in deep networks. These types of convolutional layers can be thought as special cases of TNNs. Our proposed conv_einsum can cover these alternative efficient designs as well. Here, we review two representative designs, namely the *interleaved group convolution* [121] and *separable depth-wise convolution* [122], in the language of conv_einsum.

(1) In an *interleaved group convolution*, the layer partitions the input channels S into two modes M and S' such that $S = MS'$, and the output channel T into two modes N and T' such that $T = NT'$. As a result, the layer has two 4^{th} order tensors $\mathcal{W}^{(1)} \in \mathbb{R}^{N\times M\times H\times W}$, $\mathcal{W}^{(2)} \in \mathbb{R}^{T'\times S'\times H\times W}$ as parameters and computes its output as:

```
Y = conv_einsum(bmshw,nmhw,tshw->bnthw|hw,X,W1,W2).
```

(2) In a *separable depth-wise convolution*, we assume the input and output channels are the same, i.e., $T = S$. The layer is parameterized by two matrices $\boldsymbol{W}^{(1)} \in \mathbb{R}^{S\times H}$ and $\boldsymbol{W}_2 \in \mathbb{R}^{S\times W}$ such that

```
Y = conv_einsum(bshw,sh,sw->bshw|hw, X, W1, W2)
```

We refer to [99] for more examples.

3.7.3 Optimal Sequencer

opt-einsum [116] determines the optimal evaluation path of a tensor contraction sequence via the netcon algorithm [92]. The algorithm considers the cost of evaluating intermediate

products as it explores the path tree. We do not cover the tree traversal strategy here (we refer the reader to [92]), but instead discuss how our conv_einsum generalizes netcon by calculating the cost of a tensorial intermediate product. We first review the cost (i.e., number of additions/multiplications) of each primitive operation in FLOPs. Let $\mathcal{T}^{(0)} \in \mathbb{R}^{I_0 \times \cdots I_k \times I_{m-1}}$, $\mathcal{T}^{(1)} \in \mathbb{R}^{J_0 \times \cdots J_l \cdots \times J_{n-1}}$:

1. The mode-(k,l) contraction cost is

$$\mathcal{O}\Big((\prod_{p=0}^{m-1} I_p)(\prod_{q=0,q\neq l}^{n-1} J_q)\Big). \tag{3.5}$$

2. The mode-(k,l) batch product cost is

$$\mathcal{O}\Big((\prod_{p=0}^{m-1} I_p)(\prod_{q=0,q\neq l}^{n-1} J_q)\Big). \tag{3.6}$$

3. The outer product cost is

$$\mathcal{O}\Big((\prod_{p=0}^{m-1} I_p)(\prod_{q=0}^{n-1} J_q)\Big). \tag{3.7}$$

4. The mode-(k,l) convolution cost (without a Fast Fourier Transform) is

$$\mathcal{O}\Big((\prod_{p=0}^{m-1} I_p)(\prod_{q=0}^{n-1} J_q)\Big). \tag{3.8}$$

Now, as netcon explores the path tree associated to contraction sequence (which includes batch products, outer products as special cases of contractions), it invokes a cost method which relies on cost Equations 2.5 and 2.7 to analyze an intermediate tensor product along a path The tnn-cost method replaces the standard cost function of netcon to fully realize the optimal sequencer by adding in the convolution cost model of Equation 2.7 (in addition to complex string handling to accommodate one letter of convolution type being associated to several dimensional sizes).

Convolution Varieties. In our preceding discussions, we subtly assume commutative property of the convolution operation for optimal order evaluation. However, in practice, the convolutions used in neural networks are not necessarily commutative, since one input corresponds to features and another corresponds to filters. Specifically, if the convolution is not standard (e.g., dilated or strided) or not circularly padded, the convolution operation will not be communicative. To make our OpenTNN compatible with neural network practice, we support non-communicative convolutions if a letter for convolution only appears in two inputs. When non-communicative convolution is used, we assume the input with larger dimension size at the specified mode as features and another input as filters. However, if a letter for convolution appear more than twice in the inputs (i.e., the convolution is multi-way), we will only support communicative convolution with circular padding for now.

Modification of the cost model for training. Existing einsum implementations only consider forward computation in tensor networks. However, in a neural network setting, we also need to consider the backpropagation computation. Further modification of tnn-cost is needed to incorporate backpropagation costs, which are once again tensorial sequences dictated by the same cost equations. In practice, we submit a flag to conv_einsum to indicate that the optimal sequencer is being used for training.

Specifically, given two inputs $\mathcal{T}^{(1)}$, $\mathcal{T}^{(2)}$, which interact through an atomic operation f resulting in an output tensor $\mathcal{T} = f(\mathcal{T}^{(1)}, \mathcal{T}^{(2)})$, a standard netcon-driven einsum sequencer will calculate the cost of computing $\mathcal{T}$ without any concern for associated backpropagation calculations. However, the backpropagation algorithm needs to compute $\partial\mathcal{L}/\partial\mathcal{T}^{(1)} = g_1(\partial\mathcal{L}/\partial\mathcal{T}, \mathcal{T}^{(2)})$ and $\partial\mathcal{L}/\partial\mathcal{T}^{(2)} = g_2(\mathcal{T}^{(1)}, \partial\mathcal{L}/\partial\mathcal{T})$, where g_1 and g_2 are gradient calculations dependent on f. Therefore, we modify the cost from $\mathsf{cost}(f)$ to $\mathsf{cost}(f) + \mathsf{cost}(g_1) + \mathsf{cost}(g_2)$. For in-

stance, consider f as a standard 2D-convolution, where the operation between the input $\boldsymbol{\mathcal{T}}^{(1)} \in \mathbb{R}^{B\times S\times X\times Y}$ and the weight $\boldsymbol{\mathcal{T}}^{(2)} \in \mathbb{R}^{T\times S\times H\times W}$ leads to the output $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{B\times T\times X'\times Y'}$. We have $\mathsf{cost}(f) = O(BHWXYTS)$ for the forward pass, and $\mathsf{cost}(g_1) = O(BHWX'Y'TS)$, $\mathsf{cost}(g_2) = O(BXYX'Y'TS)$ for the backward pass. In order to achieve optimal scheduling, we further modify the cost function of netcon to consider all three such costs.

3.7.4 Proof of Theorem 25 and 26

Theorem (CP reduction)**.** *Let $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{B\times S\times H'\times W'}$ be the input to a reshaped CP (RCP) convolutional kernel $\overline{\boldsymbol{\mathcal{W}}} \in \mathbb{R}^{T_1\cdots\times T_M\times S_1\cdots\times S_M\times H\times W}$ such that $T = \prod_{m=1}^{M} T_m$, $S = \prod_{m=1}^{M} S_m$ are factored into $(m+1)$ rank-R CP factor tensors $\boldsymbol{W}^{(m)} \in \mathbb{R}^{R\times T_m\times S_m}$ with $\boldsymbol{\mathcal{W}}^{(0)} \in \mathbb{R}^{R\times H\times W}$. Assume $H' \gg H$ and $W' \gg W$ are large; in particular $SH'W' > aHW$ and $BH'W' > aS$ for some constant $a \geq 1$. Furthermore, let $R \geq S$. Then the forward pass through the RCP kernel (in the syntax of conv_einsum),*

```
Y=conv_einsum(b(s1)(s2)(s3)···(sM)hw,r(t1)(s1),
    r(t2)(s2),r(t3)(s3),...,  r(tM)(sM),rhw->r(t1)(t2)(t3)
        ···(tM)hw,X,  W1,  W2,  W3,  ...,  WM,  W0)
```

has a pairwise evaluation path which costs less FLOPs than the following naive left-to-right evaluation,

```
Y=conv_einsum(br(t1)(t2)(t3)···(tM)hw,  rhw ->
    br(t1)(t2)(t3)···(tM)hw  hw,  YM,  W0)
```

where

```
Ym=einsum(r(s1)(s2)···(s(m−1))(t1)(t2)···(t(m−1)),
    r(sm)(tm)->r(s1)(s2)···(sm)(t1)(t2)···(tm), Y(m−1), Wm)
```

for $1 \leq m \leq M$*, noting that for tensor object* `YM`*, its mode symbols* `h`*,* `w` *correspond to dimensions* H', W'*, and* `b` *corresponds to an arbitrary batch size.*

Proof. We first evaluate the number of multiplications and additions of the left-to-right evaluation. Let $U_k = \prod_{i=k}^{M} S_i \prod_{i=1}^{k} T_i$ for $1 \leq k \leq M-1$ and $U_M = T$. According to the cost formulae and general definitions of the multilinear operations involved, the number of multiplications for the left-to-right evaluation is

$$\begin{aligned} M_{\text{naive}} &= BRH'W' \sum_{i=1}^{M} U_i + BRTHWH'W' \\ &= BRH'W'(\sum_{i=1}^{M} U_i + THW). \end{aligned}$$

Additions are incurred by contractions over the $\mathtt{s}_i$, and by the formula for a general multilinear contraction, in less frequency than the resulting multiplications, so to prove our claim, we only concern ourselves with demonstrating there is a path with less multiplications. Now, we will choose the following path,

```
Ym=conv_einsum(r(s1)(s2)···(s(m−1))(t1)(t2)···(t(m−1)),
    r(tm)(sm)->r(s1)(s2)···(sm)(t1)(t2)···(tm), Y(m−1), Wm),
```

for $2 \leq m \leq M$. We then perform

```
    Y0=conv_einsum(r(s1)(s2)···(sm)(t1)(t2)···(tm),
        rhw->(s1)(s2)···(sm)(t1)(t2)···(tm)hw, YM, W0)
```

followed lastly by

```
Y=conv_einsum((s1)(s2)···(sm)(t1)(t2)···(tm)hw,

    b(s1)(s2)···(sm)hw->br(t1)(t2)···(tm)hw|hw, Y0, X).
```

Let $V_k = \prod_{i=1}^{k} S_i T_i$. The number of multiplications of this path is

$$M_{\text{reduced}} = R\sum_{i=1}^{M} V_i + RSTHW + BSTHWH'W'.$$

Now, for the first term of M_{reduced},

$$R\sum_{i=1}^{M} V_i < RS\sum_{i=1}^{M}\prod_{j=1}^{i} T_i < \frac{BRH'W'}{a}\sum_{i=1}^{M} U_i.$$

with the last inequality following from our assumption that and $BH'W' > aS$. For the second term we have that

$$\begin{aligned} RSTHW &< \frac{RTH'W'}{a} \\ &< \frac{(a-1)RTH'W'}{a} \\ &< \frac{(a-1)RH'W'\sum_{i=1}^{M} U_i}{a} \\ &< \frac{(a-1)BRH'W'\sum_{i=1}^{M} U_i}{a}, \end{aligned}$$

where the second inequality follows from the fact $T < U_M$ (since U_M contains T), therefore the entire series is larger than T. By the preceding two final inequalities we have then that

$$R\sum_{i=1}^{M} V_1 + RSTHW < BRH'W'\sum_{i=1}^{M} U_i \tag{3.9}$$

For the last term, by our assumption $R \geq S$, we have that

$$BSTHWH'W' < BRTHWH'W'. \tag{3.10}$$

Combining the upper bounds given by equations 3.9 and 3.10, we establish our result, $M_{\text{reduced}} <$

M_{naive}.

Remarks. (1) Our proof demonstrates the existence of a cheaper path. The FLOPs-minimal path is more complicated and depends on the sizes of the factor tensors S_i, T_i. *(2)* Our assumption $H', W' \gg H, W$ is common in image processing.natural one – image size often greatly exceed filter size. □

Theorem (Tucker reduction)**.** *Let $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{B\times S\times H'\times W'}$ be the input to a reshaped Tucker (RTK) convolutional kernel $\overline{\boldsymbol{\mathcal{W}}} \in \mathbb{R}^{T_1\cdots\times T_M\times S_1\cdots\times S_M\times H\times W}$ such that $T = \prod_{m=1}^{M} T_m$, $S = \prod_{m=1}^{M} S_m$ are factored into $(m+1)$ rank-R_m CP factor tensors $\boldsymbol{W}^{(m)} \in \mathbb{R}^{R_m\times T_m\times S_m}$ with $\boldsymbol{\mathcal{W}}^{(0)} \in \mathbb{R}^{R\times H\times W}$ and $\boldsymbol{\mathcal{C}}^{(\in)}\mathbb{R}^{R_0\times R_1\times\cdots\times R_M}$. Assume $H' \gg H$ and $W' \gg W$ are large; in particular $SH'W' > aHW$ and $BH'W' > aS$ for some constant $a \geq 1$. Furthermore, let $\prod_{i=1}^{M} R_m \geq S$. Then the forward pass through the RCT kernel (in the syntax of* conv_einsum*),*

```
Y=conv_einsum(b(s1)(s2)(s3)···(sM)hw,(r1)(t1)(s1),(r2)(t2)(s2),
    (r3)(t3)(s3),...,(rM)(tM)(sM),(r0)hw, (r0)(r1)(r2)···(rM)->
    b(t1)(t2)(t3)···(tM)hw|hw,X, W1, W2, W3, ..., WM, W0, C)
```

has a pairwise evaluation path which costs less FLOPs than the naive left-to-right evaluation,

```
Y=conv_einsum(br(t1)(t2)(t3)···(tM)hw, rhw ->
    br(t1)(t2)(t3)···(tM)hw|hw, YM, W0)
```

where

```
 Ym=einsum(r(s1)(s2)···(s(m-1))(t1)(t2)···(t(m-1)),
    r(sm)(tm)->r(s1)(s2)···(sm)(t1)(t2)···(tm), Y(m-1), Wm)
```

for $1 \leq m \leq M$*, noting that for tensor object* `YM`*, its mode symbols* `h`*,* `w` *correspond to dimensions* H', W'*, and* `b` *corresponds to an arbitrary batch size.*

Proof. The proof follows a nearly identical argument as in the proof of previous Theorem with instead with the only difference being slightly different intermediate osts according to the R_m modes of the $\boldsymbol{\mathcal{W}}^{()}m$, which may all be upper-bounded by R. □

3.7.5 Supplementary Experimental Data

In this section, we report on additional data relating to not included in the manuscript.

Table 3.5: **Run-time (seconds per epoch) comparison between conv_einsum and PyTorch implementation of TNNs using different tensor decomposition forms in the image classification task on the CIFAR-10 dataset**. The base architecture used is ResNet-34. We observe that conv_einsum outperforms PyTorch under different tensor decompositions. Here, RXX denotes the reshaped XX decomposition, with reshaping factor M=3.

	conv-einsum		Naive w/o ckpt		Naive w/ ckpt	
Tensor Form	Train	Test	Train	Test	Train	Test
RCP	14	2.14	22	2.57	29	2.61
RTR	8	1.41	16	1.86	16	1.86
RTT	8	1.37	16	1.61	16	1.68
RTK	6	1.34	17	1.47	17	1.54

Table 3.6: **TNN performance under various model scales for image classification under low resources (4-core CPU).** CIFAR10 classification using RCP ($M = 3$) ResNet-34 and a batch size of 128. Result is seconds per epoch.

Compression Rate (CR)	RCP-train	RCP-test	TK-train	TK-test
100%	2030	132	187	12.9
50%	1378	97	151	10.6
20%	1054	76	150	10.1
10%	872	65	160	9.5
5%	785	58	147	9.0

Table 3.7: **TNN performance under various model scales for diverse machine learning tasks.** Automatic Speech Recognition (ASR) on LibriSpeech is measured by Word Error Rate (WER) (the lower, the better). Image classification (IC) on CIFAR10 (results from [2]) is measured by top-1 precision. Video classification (VC) on UCF-101 is measured by top-1 accuracy. Averaged over 3 runs.

Compression Rate (CR)	IC	ASR	VC
Original	93.2	2.1	88.98
100%	-	2.08	89.00
50%	-	2.29	88.61
20%	-	2.36	88.10
10%	91.28	2.43	87.63
5%	89.86	3.01	86.62
2%	85.70	3.76	86.41

Chapter 4: Federated Compression of Large Networks via Sketching

4.1 Introduction

Federated learning (FL) is an emerging setting of machine learning that has gained considerable interest within the last few years [86]. In centralized federated learning, a set of clients, such as mobile devices, collaboratively solve a machine learning problem under the coordination of a central server without revealing any local data. This private paradigm has found use in a wide breadth of tasks such as speech prediction, document classification, computer vision, healthcare, and finance.

Constrained Clients. At each iteration of federated training, clients download and train a global model on privately-held local data. Model updates (in the form of weights or gradients) across all participating clients are then communicated to the central server, where they are aggregated (averaged, for example) and used to update the global model. However, an oft-neglected caveat to this procedure is that constrained clients such as mobile devices could face difficulty downloading larger models such as deep convolutional networks [3, 131], transformers [128], and LSTMs [132, 133], which can contain prohibitive numbers of parameters. Furthermore, constrained clients may not be able to store or compute on fully-sized architectures. Consequently, deep learning towards edge-based applications face a resource challenge as state-of-the-art models continue to grow without bound [24, 134].

There has been an abundance of progress towards improving communication-efficiency via reduced complexity of outgoing model updates [135, 136, 137, 138, 139, 140, 141, 142]. In all of these instances, gradients are quantized/compressed to reduce uplink complexity. However, very few works have addressed the cost of downloading and hosting large models in local client memory (downlink complexity) which is at least equally responsible for communication bottlenecks.

In this paper, we propose a novel federated learning algorithm, `Comfetch`, which allows memory-constrained clients to train a large global model and compute its gradients with respect to sketch-based representations of large weights. `Comfetch` parameterizes each target weight W in the global architecture as a *sketch key pair* which contains a count sketch of the weight $H(W)$ and an unsketching map $\mathcal{U}(\cdot)$. Count sketches are data structures commonly used for lower-dimensional projections with desirable ℓ_2-recovery guarantees [143]. The central server first transmits sketch key pairs of target layer weights to the client. The client then uses the key to pass each layer input x first through the count sketched weight as $H(W)(x)$ and then unsketches the result using $\mathcal{U}(\cdot)$ so the output-dimensionality is retained. Since these sketched key pairs are cheaper to transmit and feed inputs through, we not only improve local memory costs but also improve communication and computational efficiency *for free*.

Our contributions.

1. We develop bi-directional compression FL algorithm `Comfetch` which allows memory-constrained clients to train a large network. In comparison to FL with full architecture downloads, `Comfetch` *greatly reduces memory overhead and communication costs.* Additionally, unlike other bi-directional works, *fully-sized weights are never seen in local memory.*

2. We experimentally demonstrate that `Comfetch` training converges to global models which are *competitive against uncompressed training* and other popular model reduction strategies such as random dropout and magnitude-based pruning while only using 10-25% of the full model size.

3. We provide a probabilistic *non-convex convergence guarantee* for `Comfetch`. In particular, our theory must contend with accumulated gradient approximation error resulting from successive inexact weight approximations. However, we prove under modest assumptions that these sketched architectures are guaranteed to converge to a stationary point.

4.2 Preliminaries and Problem Setup

In this section, we outline the objectives and assumptions of our federated learning setting. `Comfetch` is compatible (but not exclusively) with fully-connected networks and convolutional networks, so we also review notations common to these types of models. We end with a formal description of the Count Sketch data structure.

4.2.1 Federated Learning Setup

Let $\mathcal{D} = \mathcal{X} \times \mathcal{Y}$ be a global data set, where $\mathcal{X}$ and $\mathcal{Y}$ are the feature space and label space, respectively. Let $\{\mathcal{D}_i\}_{i=1}^N$ be a (possibly non-iid) collection of N local client data distributions over $\mathcal{D}$. Given a loss function $\mathcal{L} : \mathcal{W} \times \mathcal{D} \to \mathbb{R}$, where $\mathcal{W}$ is a hypothesis class parameterized by weight matrices, we will solve the optimization problem,

$$\min_W f(W) = \frac{1}{N} \sum_{i=1}^{N} f_i(W), \tag{4.1}$$

where W describes the set of model parameters and $f_i(W) = \mathbb{E}_{z\sim\mathcal{D}_i}\ell(W;\xi)$ is client i's loss function ℓ (we assume homogeneous loss type) and local data distribution $\mathcal{D}_i$. The central server and clients will collaboratively solve this optimization problem in an iterative manner, so we let W_t represent the global model weights at time t. At the beginning of each round, the server selects N clients uniformly at random from a large cluster to participate in training. Each client c_i for $1 \leq i \leq N$ downloads W_t and minimizes f_i using a preferred optimizer (SGD, Adam, etc.) and sends their locally-updated model parameters (or gradients) to the central server. The server aggregates all the local models to update the global weights. A typical scheme for aggregation is averaging over models/gradients (as implied in equation 4.1), which is referred to as FedAvg [84].

4.2.2 Network Architectures

To facilitate later descriptions of how to parametrize layer weights via the count sketch, we review the forward pass of two popular architectures: fully-connected networks and convolutional ResNets. Our description follows the notations of [144].

Multilayer fully-connected networks: Let $\{W_t^\ell\}_{\ell=1}^L$ represent the weights of our layers at time t, where L is the depth of the network. Let $x \in \mathbb{R}^d$ be the input. We define the network prediction recursively. Let $\sigma : \mathbb{R}^d \to \mathbb{R}^d$ represent a nonlinear activation function, and denote $x^0 := x$. We have that $x^\ell = \sigma(W_t^\ell x^{\ell-1})$ for $1 \leq \ell \leq L-1$ and the final prediction is $\hat{y} = \mathbf{a}^\top x^L$, where $\mathbf{a} \in \mathbb{R}^d$ is the output layer. Here, we have omitted bias and regularization terms. The output $\hat{y}$ is fed into $\mathcal{L}$ where it is compared against the true label.

Convolutional ResNet: We will now describe the output of a canonical ResNet architecture. We

will intentionally avoid using any convolutional operators $(*)$ since we will be sketching along the input or output channel mode. Let $x^0 \in \mathbb{R}^{s_0 \times p}$, where s_0 is the number of input channels and p is the number of pixels. We denote $s^\ell = m$ as the number of channels and p as the number of pixels for all $\ell \in [L]$. For $x^{\ell-1} \in \mathbb{R}^{s_{\ell-1} \times p}$, we use an operator $\phi_\ell(\cdot)$ to divide $x^{\ell-1}$ into a stack of p patches. Each patch will have size $qs_{\ell-1}$ which implies that $\phi_\ell(x^{\ell-1}) \in \mathbb{R}^{qs_{\ell-1} \times p}$. Let $W_t^\ell \in \mathbb{R}^{s_\ell \times qs_{\ell-1}}$. Similar to the fully-connected case, we define the layers recursively: $x^1 = \sqrt{\frac{c_\sigma}{m}}\sigma\Big(W_t^1 \phi_0(x^0)\Big), x^\ell = x^{\ell-1} + \frac{c_{res}}{L\sqrt{m}}\sigma\Big(W_t^\ell \phi_\ell(x^{\ell-1})\Big)$, for $2 \leq \ell \leq L$ and $0 < c_{res} < 1$. The output is $\hat{y} = \langle W_t^L, x^L \rangle$, where $W_t^L \in \mathbb{R}^{m \times p}$ and $\langle \, , \rangle$ is the Frobenius product.

4.2.3 Count Sketch

The crux of `Comfetch` is using the count sketch to compress layer weights. Briefly, the count sketch data structure contains a collection of k pairwise-independent hashing maps $h_i : [d] \to [c]$ for $i \in [k]$, each of which is paired with a sign map $s_i : [d] \to \{\pm 1\}$, for $i \in [k]$. Each hash function/sign map pair is used to project a d-dimensional vector x into a smaller c-dimensional space, which we refer to as a sketch. Through an "unsketch" procedure (detailed in Section 4.8.2), we use the k sketches to create a d-dimensional approximation $\hat{x}$ of x. To develop a sketch of a weight matrix, we apply a count sketch to each row of the weight matrix. Our method is generalizable to tensorial weights, but these require higher-order count sketches (HCS) (see 4.8.2).

4.3 Comfetch

We propose Algorithm 6, `Comfetch`, to minimize the aggregated loss function (equation 4.1) of a large network on memory-constrained devices. Algorithm 6 uses only a single sketch, but it is possible to utilize multiple sketches of weights. We assume a single count sketch is used for all weight compressions for the remainder of this section.

We use a sketch-unsketch paradigm $W_t^\ell \to \mathcal{U}(H(W_t^\ell)) = {H^\ell}^\top H^\ell W_t^\ell$, where $H^\ell \in \mathbb{R}^{d\times c}$ is a randomly drawn count sketch matrix to approximate each weight $W_t^\ell \in \mathbb{R}^{d\times d}$ at iteration t and layer ℓ with $c << d$. We require the client to store only a hash-function $h^\ell \in \mathbb{R}^d$ associated with H^ℓ and the sketched weight $H^\ell W_t^\ell$, thus reducing the memory footprint of storing each layer from $\mathcal{O}(d^2)$ to $\mathcal{O}((c+1)d)$. (As an abuse of notation, we use h^ℓ and H^ℓ interchangeably, since one may infer H^ℓ from h^ℓ, the latter of which we will transmit in implementation since it is cheaper to do so.) We design a mechanism for the central controller to aggregate model updates from the clients and backpropagate the sketched model parameters. The algorithm follows the typical structure of a federated learning algorithm: *model transmission and download*, *client update*, and *model update*. This process is repeated over T iterations. We will describe each phase in detail for a fixed weight. For simplicity of notations, we assume that $W_t^\ell \in \mathbb{R}^{d\times d}$.

4.3.1 Model Transmission and Download

At iteration t, the central server first prepares the global model for the transmission by sketching down all the current weights $\{W_t^\ell\}_{\ell=1}^L$, via Count Sketch matrices $H^\ell \in \mathbb{R}^{c\times d}$, where $c << d$ is referred to as the *sketching length* or *sketch dimension*. We assume that our layers are either convolutional or fully-connected as described in Section 4.2. For each weight W_t^ℓ, the

Algorithm 6 `Comfetch` (Single Sketch)

Require: initial weights $\{W_0^\ell\}_{\ell=1}^L$, learning rate η, number of iterations T, momentum parameter ρ, batch size M of data, batch size N of clients

Init momentum term $\{u_0^\ell = 0\}_{\ell=1}^L$

Init error accumulation term $e_0 = 0$

for $t = 1, 2, \ldots, T$ **do**

  Init sketch key pairs $\{H^{\ell^T}, H^\ell W_t^\ell\}_{\ell=1}^L$

  Uniformly select at random N clients $c_1, c_2, \ldots, c_N$

  loop {in parallel on clients $\{c_i\}_{i=1}^N$}

  for $\ell = 1, 2, \ldots, L$ **do**

    Download weight sketches $\{H^{\ell^T}, H^\ell W_t^\ell\}_{\ell=1}^L$

    Compute grads $g_i^\ell = \nabla_{H^\ell w_t^\ell} \mathcal{L}(\hat{W}_t^\ell, z \sim \mathcal{D}_i)$ ▷ $\hat{W}_t^\ell = H^{\ell^\top} H^\ell W_t^\ell$

  end for

  Send $\{g_i^\ell\}_{\ell=1}^L$ to Central Server

  end loop

  for $\ell = 1, 2, \ldots, L$ **do**

    Aggregate restored gradients: $g^\ell = \frac{1}{N} \sum_{i=1}^N g_i^\ell$

    Update momentum: $u_t^\ell = \rho u_{t-1}^\ell + g^\ell$

    Update error feedback: $e_t^\ell = \eta u_t^\ell + e_t^\ell$

    Approximate gradient: $\Delta_t = \text{Top-}k(e_t^\ell)$

    Error accumulation: $e_{t+1}^\ell = e_t^\ell - \Delta_t$

    Update weight: $W_{t+1}^\ell = W_t^\ell - \Delta_t$

  end for

end forreturn $\{W_T^\ell\}_{\ell=1}^L$

server randomly draws a count sketch matrix H^ℓ. The server transmits $\{(H^{\ell^\top}, H^\ell W_t^\ell)\}_{\ell=1}^L$ to N clients selected uniformly at random from a large cluster, who then download the sketched parameters.

Cost Complexity. Note that any H^ℓ bijectively corresponds to a hash function $h^\ell : d \to c$ which is representable as a length d vector, so it is cheaper to store/transmit h^ℓ. Hence, in practice, the central server will transmit $\{(h^\ell, H_i^\ell W_t^\ell)\}_{\ell=1}^L$, for a total local memory and transmission cost of $\mathcal{O}\big((c+1)d\big)$, which is far less than the usual $O(d^2)$ cost of transmission and storage.

4.3.2 Client Update

The client C_i will now conduct a single round of training on the sketched network parameters using local data. In practice, the client distributions $\mathcal{D}_i$ will be finite and small [86], so we assume that the client is taking the full gradient with respect to the weights, but the algorithm generalizes to stochastic gradients as well.

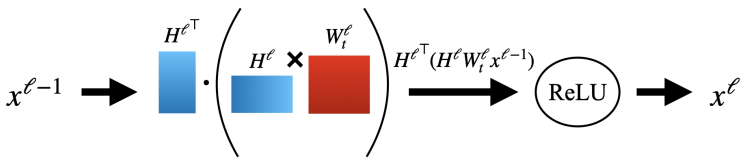


Figure 4.1: The forward pass of a fully-connected layer.

Forward pass. Assume there are L weights. Following the notations described in Section 4.2, the forward pass of a fully-connected (FC) layer is $x^\ell = \sigma(H^{\ell^\top}(H^\ell W_t^\ell x^{\ell-1}))$, for $1 \le \ell \le L-1$ and the final output is $\hat{y} = \mathbf{a}^\top x^L$. Similarly, for a convolutional ResNet, we have that $x^\ell = x^{\ell-1} + \frac{c_{res}}{L\sqrt{m}}\sigma\Big(H^{\ell^\top}\big(H^\ell W^\ell \phi_\ell(x^{\ell-1})\big)\Big)$, for $2 \le \ell \le L$ and the final output is $\hat{y} = \langle W^L, x^L \rangle$, where $W^L \in \mathbb{R}^{m \times p}$.

Cost Complexity. Each sketched weight costs the client $\mathcal{O}(cd)$ to locally store which is

an improvement over storing the original weight which costs $\mathcal{O}(d^2)$.

Remark 1: The client never sees or directly compute a $d \times d$ weight matrix at any stage. As emphasized by parenthetical grouping, for example in the case of a fully-connected layer, we compute $H^\ell W_t^\ell x^{\ell-1}$, followed by multiplication on the left by $H^{\ell^\top}$.

Remark 2: In the convolutional case, where the kernel can be interpreted as a tensor of filter weights, $H^\ell W^\ell \phi_\ell(x^{\ell-1})$ can be regarded as a higher-order count sketch (HCS) of $W^\ell \phi_\ell(x^{\ell-1})$ (Section 4.8.2.2, or we may matricize the kernel and apply the count sketch in a stacked manner.

4.3.3 Backwards pass and uplink

To compute the gradient, we must first clearly define the weights of the client networks. We have $H^{\ell^\top} H^\ell W_t^\ell x^{\ell-1}$ and $H^\ell W^\ell \phi_\ell(x^{\ell-1})$ in the fully-connected and convolutional layers, respectively. Therefore, we can represent each sketched weight as $R^\ell W_t^\ell$ where $R^\ell = H^{\ell^\top} H^\top W_t^\ell$, so our sketches are simply dimension-preserving linear transformations of the original weights. For further notational convenience, we denote $\hat{W}_t^\ell \triangleq R^\ell W_t^\ell$. Now that we have defined the weights of our client models, we may now take gradients. The server will want to receive $\frac{\partial \mathcal{L}(\hat{W}_t^\ell, z)}{\partial W_t^\ell}$ as an approximation of $\frac{\partial \mathcal{L}(W_t^\ell, z)}{\partial W_t^\ell}$, but the client will not want to store or compute $\frac{\partial \mathcal{L}(\hat{W}_t^\ell, z)}{\partial W_t^\ell}$, since it is of size $d \times d$. Instead, the client will transmit a $\mathcal{O}(c \times d)$ packet of data which will allow the server to compute $\frac{\partial \mathcal{L}(\hat{W}_t^\ell, z)}{\partial W_t^\ell}$. Using the chain rule we have that:

$$\frac{\partial \mathcal{L}(\hat{W}_t^\ell, z)}{\partial W_t^\ell} = \frac{\partial \mathcal{L}(\hat{W}_t^\ell, z)}{\partial H^\ell W_t^\ell} \frac{\partial H^\ell W_t^\ell}{\partial W_t^\ell}. \tag{4.2}$$

Since the server has knowledge of $\frac{\partial H^\ell W_t^\ell}{\partial W_t^\ell} = H^\ell$, the client only needs to upload $g_i^\ell \triangleq \nabla_{H^\ell W_t^\ell} \mathcal{L}(\hat{W}_t^\ell, z) \in \mathbb{R}^{c \times d}$. One might ask: *how does* $\nabla_{H^\ell W_t^\ell} \mathcal{L}(\hat{W}_t^\ell, z)$ *relate to* $\nabla \mathcal{L}(\hat{W}_t^\ell, z)$*?*

Using the chain rule again (and dropping z for notational convenience), we have that

$$\nabla_{W_t^\ell}\mathcal{L}(\hat{W}_t^\ell, z) = \nabla\mathcal{L}(\hat{W}_t^\ell)\nabla_{W_t^\ell}\hat{W}_t^\ell = \nabla\mathcal{L}(\hat{W}_t^\ell)R^\ell, \tag{4.3}$$

which indicates that $\nabla_{W_t^\ell}\mathcal{L}(\hat{W}_t^\ell)R^\ell = \nabla\mathcal{L}(\hat{W}_t^\ell)$, i.e., the gradients the client is submitting to the server are count sketch approximations of the true gradient of our sketched network. Thus, we are performing uplink compression of our gradients using count sketches as our compression operator.

Cost Complexity. Each g_t^ℓ costs $\mathcal{O}(cd)$ to store and transmit, which is a strong improvement over the usual uplink complexity of $\mathcal{O}(d^2)$ and even cheaper than the sketched storage cost of $\mathcal{O}((c+1)d)$.

Remark. The client will never directly compute $g_i^\ell \in \mathbb{R}^{d\times d}$. The purpose of equation (4.2) is to illustrate the gradient calculation, but we first take care to show this derivative is well-defined. Assuming $\mathcal{L}$ is differentiable with respect to any weight, we only need to prove that $\frac{\partial R^\ell W}{\partial H^\ell W}$ is computable, which is indeed the case due to the structure H^ℓ. For any count sketch matrix $H\mathbb{R}$ and vector x, if x_i is bucketed by hash function h_j, we have that $[H^\top Hx]_i = H_{ji}\cdot[Hx]_j$, therefore, the partial derivative $\frac{\partial R^\ell W}{\partial H^\ell W}$ is well-defined. In practice, the client will use an autograd-like library.

4.3.4 Model Update

The Central Server aggregates the $\{g_i^\ell\}_{\ell=1}^L$ across all clients c_i for $i \in [N]$ and computes a de-compressed average over the gradients: $g^\ell = \frac{1}{N}H^{\ell^\top}\sum_{i=1}^N g_i^{\ell^\top}$.

The remainder of the model update is an SGD (or Adam)-like procedure that follows the error-feedback and momentum scheme similar to other compression-correcting literature [137,

138]. The error-feedback term e_t allows for the correction of error associated with our gradient approximations g^ℓ. Specifically, we are correcting the error associated with using $\nabla_{W_t^\ell} f(R^\ell W_t^\ell)$ as an approximation of $\nabla f(W_t^\ell)$. Once we form the full error term e_t, we take the Top-k components (in absolute magnitude) of it, which we expect to be relatively undiluted by the approximation error, to form Δ_t. We have that Δ_t is our error-corrected gradient approximation with a momentum term already baked into it. (The momentum term u_t is common to SGD-variants in the non-federated setting, the benefits of which are discussed by Sutskever et al. [30].) This Δ_t will help us mimic stochastic gradient descent, as shown in Line 18 of Algorithm 6.

4.4 Related Work

A practical assumption of centralized federated learning is that clients will be physically removed from the central server and communicate over unreliable wireless channels [86, 145, 146]. Therefore, there has been significant interest in reducing the size of data communicated between the server and federated agents [135]. [137] suggest taking count sketches of the local gradients to reduce client update costs to great effect, but their algorithm `Sketched SGD` requires an extra round of communication with the server, which although appropriate for their distributed single-machine setting, would fail in the general federated setting due to a lack of persistent clients. [138] successfully eliminates the extra round with their `FetchSGD` by taking several independent sketches of the gradients coupled with error feedback [147] in the server update phase. `FetchSGD` only extracts the Top-k components of the gradients to mitigate count sketch recovery error, similar to the `MISSION` algorithm [148]. Low-precision quantization of gradients has also been proposed [139, 149] to great effect.

While the aforementioned methods successfully decrease upload communication costs, they do not address downlink complexity, which could be a bottleneck for memory-constrained clients such as mobile devices. [150] and [151] propose distributing subnetworks of the global models to clients based on computational and memory constraints, but we are interested in implicitly preserving the full-dimensionality of the model to maximize performance. [152] sparsifies global models at the server, which once again, reduces the overall capacity of the model, and is better suited for pruning; pre-trained models. *Bi-directional compression works are still uncommon*: to the best of our knowledge, [153], is the only other work to consider compression of model weights while retaining dimensionality, but convergence guarantees are only provided under aggressive assumptions of lossless decompression and experiments are only performed on relatively small models such as ResNet-9. Other bi-directional works transmit compressed gradients [154, 155, 156, 157] and require restoration inside local memory which is equivalent to storing a fully-sized architecture, which `Comfetch` avoids.

4.5 Convergence Guarantee

In this section, we provide a non-convex convergence result for `Comfetch` For all results, $||\cdot||$ refers to the ℓ_2 norm. We begin by outlining our assumptions. Without loss of generality, we will denote our weights as $w \in \mathbb{R}^d$ (through vectorization, for example).

Assumption 1 (L-Smooth)**.** *The objective function $f(W)$ in equation 4.1 is L-smooth. That is, for all $x, y \in \mathbb{R}^d$ we have that,*

$$||\nabla f(x) - \nabla f(y)|| \leq L||x - y||. \tag{4.4}$$

Assumption 2 (Unbiased and Bounded)**.** *All stochastic gradients g of $f(w)$ are unbiased and bounded,*

$$\mathbb{E}g = \nabla f(w) \;\; and \;\; \mathbb{E}||g||^2 \leq G^2. \tag{4.5}$$

Assumptions 1-2 are standard to convergence proofs of SGD-like algorithms, including those of a federated nature [137, 138, 147, 158, 159].

Assumption 3 (Heavy Hitters)**.** *In the notation of Algorithm 6, let $\{W_t^\ell\}_{t=1}^T$ be the sequence of model weights of the ℓ^{th} layer generated by* `Comfetch`*. There exists a constant ϵ such that for all $t \in [T]$, the approximated gradient of our sketch network with momentum $z_t^\ell := \eta(\rho u_{t-1}^\ell + g_{t-1}^\ell) + e_{t-1}^\ell$ contains at least one coordinate i such that $(z_t)_i^2 \geq \epsilon||z_t||^2$. Furthermore, there exists a constant c such that for any given weight W_t^ℓ, there exists a coordinate j with $(W_t^\ell)_j^2 \geq (1/c)||W_t^\ell||^2$. These coordinates are referred to as heavy hitters [160].*

Assumption 3 is a variant on a common heavy-hitter assumption suggested in the convergence theorem of `FetchSGD` to ensure successful error-feedback [138]. Heavy hitters are also required in the convergence analysis of `Sketched-SGD` [137]. In this version, we are requiring $\nabla_w \mathcal{L}(H^\top H w)$ to contain a heavy hitter.

Theorem 27. *Let $w_0 \in \mathbb{R}^d$ denote an initialized model weight and consider a sketch of size $\mathcal{O}\big(\frac{1}{\epsilon^2}\log\frac{d}{\delta}\big)$ where sketch dimension $c = 1/\epsilon^2$. Define $\tilde{f}(x) = f(\mathcal{U}_H(H(x))$. Under Assumptions 1-3 and with step size $\gamma = \frac{c(1-\rho)}{2Ld\sqrt{T}}$, we have that Comfetch returns $\{w_t^\ell\}_{i=1}^T$ such that*

$$\min_{t=1\cdots T}||\nabla \tilde{f}(w_t)||^2 = \mathcal{O}\left(\frac{4Ld(f(w_0)-f^*)+G^2}{c\sqrt{T}} + \frac{2d^2(1+\epsilon)^2G^2}{c^2(1-\epsilon)^2\epsilon^2 T}\right), \tag{4.6}$$

with probability $1-\delta$ over the sketching randomness.

Our analysis critically relies on the fact that if our original network is L-smooth, the

sketched architecture is $\frac{Ld}{c}$-smooth. We defer the proof to Section 4.8.3

4.6 Experiments

In this section, we investigate the performance of `Comfetch` models.

Vision Task. We perform image classification tasks over CIFAR-10 and CIFAR-100 [3] using a ResNet-18 architecture, which contains roughly 11M parameters. Both CIFAR-10/100 are benchmark computer vision datasets containing with 60K 32×32 color images labeled with 10 or 100 possible labels, respectively. Vision experiments were run on a computing cluster using mixtures of NVIDIA Tesla T4 and RTX A4000 GPUs each equipped with 16 GB RAM. We used the PyTorch library for training our models and MPI for distributed averaging.

Training Setup. The number of federated clients is always either 4 or 10, which we specify within the captions. At each round, all clients download the current model and each weight W is passed through a count sketch recovery operation $H^\top HW$, where H is a count sketch matrix of size $d_2 \times cr \cdot d_1$, where d_2 is the output dimensionality, cr is the compression rate, and d_1 is the input dimensionality. This simulates a layer input passing through a sketch key pair (see Figure 4.1).

Clients will locally train their model for $E = 1$ epoch with batch size $B = 128$ before averaging and the clients use an Adam optimizer with learning rate $lr = 0.001$, $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1e{-}8$. After completing the training epoch, the local models are averaged to update the global model. The sketch used is homogeneous across all clients and we do not use error feedback (which has been observed to have an insignificant effect on performance and only needed for theoretical analysis).

Data Splits. For iid training, the training dataset is shuffled and then each client is given $|\mathcal{D}|/N$ samples selected uniformly at random, where $|\mathcal{D}|$ is the total dataset size. For non-iid training, the size of local training sets are the same size, but samples are selected in a label-skewed manner according to a Dirichlet allocation.

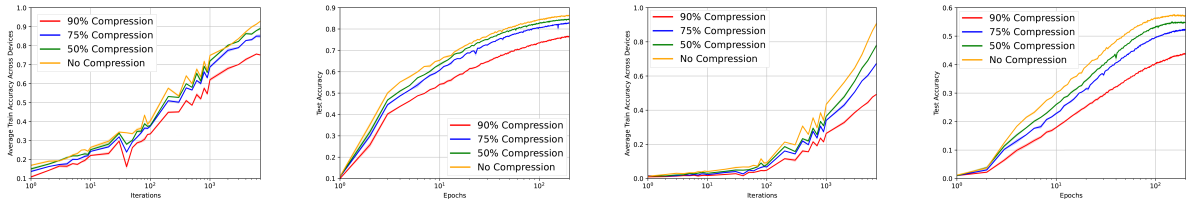
4.6.1 CIFAR10/100 Experiments

In general, `Comfetch` *models are competitive against uncompressed FedAvg training in CIFAR10/100 training.* CIFAR-10 model performance is presented in Table 4.1 and and Figures 4.2 and 4.3. For iid settings, 50% sketched compression results in a test accuracy drop off of $< 2\%$, while 75% compression results in decreased accuracy of $< 4\%$. We begin to notice a significant decline at 90% compression, which appears to be a general threshold across all experiments. Non-iid training appears to be slightly more challenging, but nonetheless, our `Comfetch` models mimic the performance of un-sketched models with minimal dropoff for compression rates up to 75%.

Method	Compression Rate	IID Test Accuracy (%)	Non-IID Test Accuracy (%)
`Comfetch`	90%	76.53	75.29
`Comfetch`	75%	82.85	81.89
`Comfetch`	50%	84.42	83.59
`No Compression`	0%	86.37	84.99

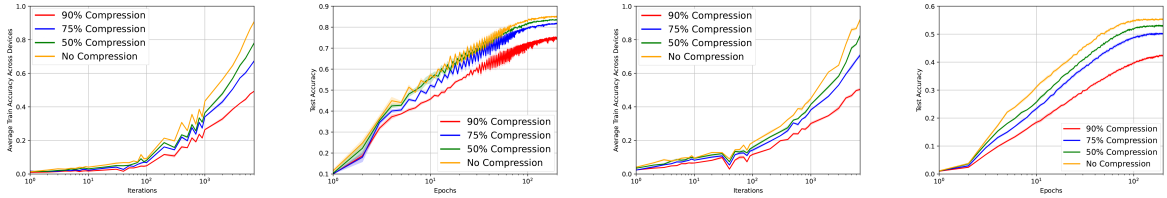
Table 4.1: **IID CIFAR-10 Top accuracy.** Average top test accuracy over three runs for 10 client `Comfetch` classifying CIFAR-10 [3] using ResNet-18. We analyze how `Comfetch` performs over a range of compression rates as well as IID and non-IID ($\alpha = 1$) dataset splits.

CIFAR-100 training, displayed in Table 4.2 and the bottom series of Figures 4.2 and 4.3, is far more challenging. However, our models still display competitive performance against non-sketched training for models with up to 75%, regardless of iid or non-iid training. Again, we observe a noticeable decline at 90% compression, which we is due to sketch recovery becoming



(a) Train Accuracy (b) Test Accuracy (c) Train Accuracy (d) Test Accuracy

Figure 4.2: **IID CIFAR-10/100 Training/Test Curves.** Accuracy convergence of `Comfetch` under varying compression rates with $N = 10$ clients. **(a)-(b)** corresponds to CIFAR-10 [3] image classification, while **(c)-(d)** correspond to CIFAR-100 image classification. In these experiments, only a single sketch is used and the datasets are IID. Similar accuracy with different `Comfetch` compression rates suggests that our method retains the expressive power of non-sketched models, while simultaneously reducing their storage size.

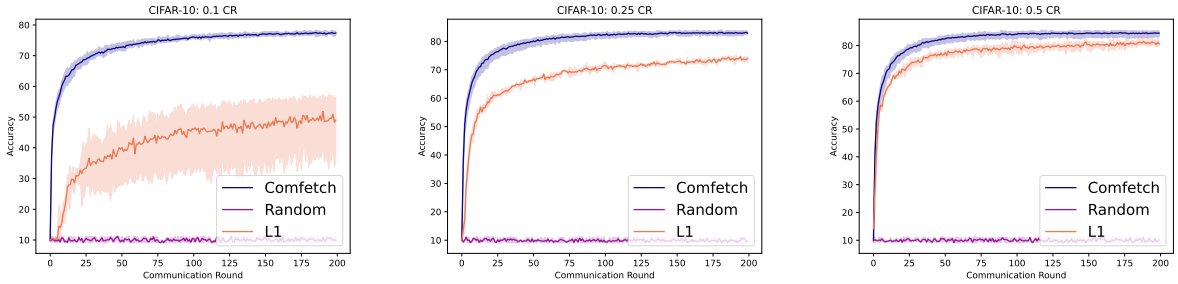


(a) Train Accuracy (b) Test Accuracy (c) Train Accuracy (d) Test Accuracy

Figure 4.3: **Non-IID CIFAR-10/100 Training/Test Curves.** Accuracy convergence of `Comfetch` under varying compression rates in non-IID settings with $N = 10$ clients. **(a)-(b)** corresponds to CIFAR-10 image classification, while **(c)-(d)** corresponds to CIFAR-100 image classification. Datasets are non-IID with a Dirichlet split ($\alpha = 1$). Only a single sketch is used. Similar accuracy with different `Comfetch` compression rates suggests that our method retains the expressive power of the model while reducing the parameter counts even in the non-IID domain.

Method	Compression Rate	IID Test Accuracy (%)	Non-IID Test Accuracy (%)
`Comfetch`	90%	43.80	42.52
`Comfetch`	75%	52.25	50.21
`Comfetch`	50%	54.76	52.89
`No Compression`	0%	57.16	55.21

Table 4.2: **Non-IID CIFAR-100 Top Accuracy.** Average test accuracy over three random runs for 10 clients classifying CIFAR-100 [3] using ResNet18. We analyze how `Comfetch` performs over a range of compression rates as well as IID and non-IID ($\alpha = 1$) dataset splits.



(a) L1 Pruning (b) Random Pruning (c) Random Pruning

Figure 4.4: **Pruned Models.** $N = 4$ clients are given iid distributions of CIFAR-10 (12500 samples each) split across classes. Client model weights are then pruned randomly or based on ℓ_1 magnitude at compression rates of 0.1, 0.25 and 0.5.

too lossy at this degree.

4.6.2 FedAvg Pruning

For multi-client federated averaging, we investigate baseline model compressions in Figure 4.4 where a non-sketched ResNet-18 is pruned according to ℓ_1-magnitude or random dropout prior to training to simulate compression. There are 4 clients in this setting, splitting CIFAR-10 in an iid manner. While such strategies have been shown to perform well for pre-trained models, we demonstrate in Figure 4.4, that such naive model compression is less performant than Comfetch. *No matter the pruning strategy or degree of compression, FedAvg models compressed in this manner underperform sketch-based weight compression.* Random dropout, in particular, prior to training is especially destructive. We observe that at 90% compression with random weight pruning, the models essentially classify close to random. Magnitude-based pruning appears to be far superior to random pruning.

4.7 Conclusion

In this work, we present a federated learning algorithm `Comfetch` for training large networks on memory-constrained clients. In our scheme, the server parameterizes the global model weights via sketch key pairs, significantly reducing storage costs. These sketched architectures greatly reduce bi-directional communication, memory, and computational costs while retaining high performance.

The limitations of `Comfetch` motivate future directions. We note that the theory developed in Section 4.5 does not predict the success of single-sketch `Comfetch` which we observe to be very effective in our experiments. Structurally-aware sketches [161, 162] or oblivious sketches [163] may provide insight into one-sketch guarantees.

4.8 Supplemental Material

4.8.1 Further Related Work

Network Compression. A popular approach to network compression involves taking low-rank factorizations of the weight tensors [106, 107, 164, 165]. Oftentimes, this will require learning the factors, which increase training overhead [107, 164] or increases the depth of the network [165], and computing exact tensor factorizations is known to be NP-hard [166]. `Comfetch` avoids these issues by relying on simple linear transformations of the original weights. [167] replaces fully-connected and convolutional layers in a non-federated setting via sign sketches (Hadamard matrices). Their compressed CNNs perform worse than our federated models on CIFAR-10 [3] classification, cost more memory than our `Comfetch` to store, and do not have

convergence guarantees. Knowledge distillation [30, 168], quantization [169], binarization [170], Huffman coding [171], and other similar techniques seek to gradually reduce the number of the parameters during training or compress the model post-training. In our scenario, the memory-constrained clients can never store the original architecture due to its size, and in most centralized FL settings, the server does not have access to the local client data [86], therefore rendering these techniques inapplicable. Hence, we opt for the popular count sketch compression scheme, which is data-oblivious.

4.8.2 Sketches

In this section, we review sketching algorithms and data structures which are the most relevant to the implementation of `Comfetch`.

4.8.2.1 Count Sketch

Sketching algorithms seek to compress high-dimensional data structures into lower-dimensional spaces via entry hashing. Sketching was initially proposed by Muthukrishnan et al. [172] as a solution to estimating the frequency of items in a stream [172]. Alon et al. [160] then proposed the eponymous AMS sketch to formalize this setting along with ℓ_2-guarantees of the decompression of hashed data. The *Count Sketch* is the successor of the AMS sketch, first proposed by Charikar et al. [143] and outlined in Algorithm 7. The Count Sketch is desired in settings where one is interested in estimating the heavy hitters of a high-dimensional vector, which are the entries with large absolute magnitude relative to other entries.

The Count Sketch data structure contains a collection of k pairwise-independent hashing

maps $h_i : [d] \to [c]$ for $i \in [k]$, each of which is paired with a sign map $s_i : [d] \to \{\pm 1\}$, for $i \in [k]$. Each hash function/sign map pair is used to project a d-dimensional vector x into c-dimensional space, which we refer to as a sketch. Furthermore, we refer to c as the sketching length and k as the number of the sketches. Each of the k sketches is representable as a linear transformation $H_i x \in \mathbb{R}^c$ for $i \in [k]$, where $H_i \in \mathbb{R}^{c \times d}$ is referred to as a Count Sketch matrix. To "unsketch" or restore a sketch back to d-dimensional space, we simply multiply by the transpose of H_i, i.e., $\hat{x}_i = H_i^\top H_i x \in \mathbb{R}^d$, where we use the hat notation to specifically indicate that $\hat{x}_i$ is merely an approximation of x. The final approximation $\hat{x}$ of x is determined coordinate-wise as $(\hat{x})_j = \underset{1 \le i \le k}{\text{median}} \{\hat{x}_i\}_{i=1}^k$ for $j \in [d]$.

Count sketch matrices As detailed in the previous section, each sketch has a matrix representation which we will now make explicit. Let $h : [d] \to [c]$ and $s : [d] \to \{\pm 1\}$ be a hash function/sign pair. The associated count sketch matrix H is representable as the following $c \times d$ matrix transformation: fixing $i \in [c]$ and $j \in [d]$, we have that $(H)_{ij} = s(i)$ if $h(j) = i$ for $j \in [d]$ and 0 otherwise.

4.8.2.2 Higher-order Sketches

While the count sketch is traditionally used to project vectors into lower-dimensional space, it is also possible to sketch higher order tensors. The advantage to sketching higher order tensors is two-fold for us: (1) The ℓ_2 guarantees associated with vectorizing a tensor and then using a standard count sketch does not scale well with increased modes and does not take advantage of the compact representations tensors offer. (2) In the forward pass discussion of Section 4.3, for multi-layer perceptrons, our parameterized weights imitate the count sketch of vectors (namely, the last

Algorithm 7 `Count Sketch` [143]

Require: vector $x \in \mathbb{R}^d$, number of hash functions k, sketch length c

procedure INIT(c, k)
  Init sign hashes $\{s_j\}_{j=1}^k$ and hash functions $\{h_j\}_{j=1}^k$ ▷ Must be 2-wise independent
end procedure
  Init $k \times c$ table of counters S
procedure SKETCH(i, x_i)
  for $j = 1, \ldots, k$ **do**
    $S[j, h_j(i)] += s_j(i)x_i$
  end for
end procedure
procedure UNSKETCH(k)
  Init length k array estimates
  for $j = 1, \ldots, k$ **do**
    estimates$[k] = s_j(i)S[j, h_j(i)]$
  end forreturn
end procedure

layer's output), but for CNN layers, our parametrized weights imitate multiplying matrices by count sketch matrices (since the last layer's output is a matrix), which are not modeled under the usual count sketch model. A variety of tensor sketching algorithms exist [173, 174, 175], but we elect to use the higher-order sketch (HCS) of Shi et al. [176] due to its ease of implementation and resemblance to the count sketch.

We leave the details of the HCS algorithm for tensors with order 3 or greater to [176], but for matrices, the method is straightforward. Let $W \in \mathbb{R}^{d \times d}$, for simplicity (but one may use rectangular matrices as well). First, in the same manner as the count sketch, draw k pairwise-independent hash functions h_i and sign maps s_i of sketching length c. To sketch W, simply compute and maintain the collection of products $\hat{W}_i = H_i^\top H_i W$. The decompression (recovery) of W is denoted $\hat{W}$, where coordinate-wise, we have that $(\hat{W})_{mn} = \text{median}\{\hat{W}_{i_{mn}}\}_{i=1}^k$, for $m, n \in [d]$.

4.8.2.3 Two-Sided Sketching

Our discussion of the HCS in Section 4.8.2.2 only allows compression along a single mode of a matrix $W \in \mathbb{R}^{d\times d}$, but it is possible to sketch along both modes of the matrix, allowing for further compression. In architectural terms, this allows us to decrease the width of each layer and decrease the density of inter-layer connectivity. The procedure is a simple extension of the one-sided HCS. Draw k pairwise-independent hash functions h_i and sign maps s_i . To sketch W, maintain the collection of products $\hat{W}_i = (H_{i_1}^\top H_{i_1})W(H_{i_2}^\top H_{i_2})$, where we note that H_{i_1} and H_{i_2} are independently drawn. The decompression (recovery) of W is denoted $\hat{W}$, where coordinate-wise, we have that $(\hat{W})_{mn} = \text{median}\{\hat{W}_{i_{mn}}\}_{i=1}^k$, for $m, n \in [d]$.

Backpropagation Rule Following the single-sketch model of Section 4.3, we derive the back-propagation rule for a two-sided sketch. We first require a lemma, which is a basic result of matrix calculus:

Lemma 28. *Let $X \in \mathbb{R}^{m\times n}$, $A \in \mathbb{R}^{p\times m}$, $B \in \mathbb{R}^{m\times q}$, then*

$$\frac{\partial AXB}{\partial X} = B \otimes A^\top \tag{4.7}$$

where $\otimes$ is the Kronecker product.

We may now use the above result for calculating the gradient of our network after sketching our layers $\{W_t^\ell\}_{\ell=1}^L$ from both sides, in the same notation as the federated learning setup discussed in Section 4.2.

Proposition 29. *Let $W_t^\ell \in \mathbb{R}^{d\times d}$, $H_1^\ell \in \mathbb{R}^{p\times d}$, $H_2^\ell \in \mathbb{R}^{d\times q}$. Denote $\tilde{W}_t^\ell = H_1^\ell W_\ell H_2^\ell$, i.e., a weight*

matrix sketched from both sides. Then,

$$\nabla_{W_t^\ell}\mathcal{L}(W_t^\ell, z) = (H_2^\ell \otimes H_1^{\ell\top})\nabla_{\tilde{W}_t^\ell}\mathcal{L} \tag{4.8}$$

Proof. By the chain rule,

$$\frac{\partial \mathcal{L}}{\partial W_t^\ell} = \frac{\partial \mathcal{L}}{\partial \tilde{W}_t^\ell}\frac{\partial \tilde{W}_t^\ell}{W_t^\ell} \tag{4.9}$$

$$= \frac{\partial \mathcal{L}}{\partial \tilde{W}_t^\ell}(H_1^\ell \otimes H_2^{\ell\top}), \tag{4.10}$$

where the last line follows by Lemma 28. By transposing both sides of the equation to obtain gradient equations and noting that $(A \otimes B)^\top = B^\top \otimes A^\top$, the result follows. □

4.8.3 Proof of Theorem 27

Before proving our main convergence result, we define a recursive sequence crucial to error-feedback analysis following [138, 147].

Let $C(x) = \text{Top-}k(\mathcal{U}(H(x))$. We define the temporal sequence $\tilde{W}_t = W_t - e_t - \frac{\eta\rho}{1-\rho}u_{t-1}$. This sequence is recursive:

$$\begin{aligned}
\tilde{W}_t &= W_{t-1} - C(\eta u_{t-2} + g_{t-1}) \\
&\quad + C(\eta(\rho u_{t-2} + g_{t-1}) + e_{t-1}) \\
&\quad - \eta(\rho u_{t-2} + g_{t-1}) - e_{t-1} - \frac{\eta\rho}{1-\rho}z_{t-1} \\
&= W_{t-1} - e_{t-1} - \eta g_{t-1} - \eta\rho u_{t-2} \\
&\quad - \frac{\eta\rho}{1-\rho}(\rho u_{t-2} + g_{t-1}) \\
&= W_{t-1} - e_{t-1} - \frac{\eta\rho}{1-\rho}u_{t-2} - \frac{\eta}{1-\rho}g_{t-1}
\end{aligned}$$

$$= \tilde{W}_{t-1} - \frac{\eta}{1-\rho} g_{t-1}$$

This is an almost-stochastic SGD update, but we must prove $\nabla f(\tilde{W}_t) \approx \nabla f(W_t)$. To do so, we will provide two results which bound $\mathbb{E}||u_t||^2$ and $\mathbb{E}||e_t||^2$, respectively.

Lemma 30. $\mathbb{E}||u_{t-1}||^2 \leq \left(\frac{G}{1-\rho}\right)^2$.

Proof.

$$\mathbb{E}||u_t||^2 = \mathbb{E}||\sum_{i=1}^{t} \rho^i g_i||^2 \leq \mathbb{E}||\sum_{i=1}^{\infty} \rho^i g_i||^2 \leq \left(\frac{G}{(1-\rho)}\right)^2. \tag{4.11}$$

□

Proposition 31 ([147], Lemma 3)**.** $\mathbb{E}||e_{t-1}||^2 \leq \frac{4(1-\epsilon^2)\eta^2 G^2}{\epsilon^2(1-\rho)^2}$.

Proof.

$$\begin{aligned}
\mathbb{E}||e_{t+1}||^2 &= \mathbb{E}||\eta(\rho u_t + g_t) + e_t - C(\eta(\rho u_t + g_t) + e_t)||^2 \\
&\leq (1-\epsilon)\mathbb{E}||\eta(\rho u_t + g_t) + e_t||^2 \\
&\leq (1-\epsilon)\Big((1+\gamma)||e_t||^2 + (1+1/\gamma)\eta^2||u_t||^2\Big) \\
&\leq (1-\epsilon)\left(||e_{t-1}||^2 + \frac{(1+1/\gamma)\eta^2 G^2}{(1-\rho)^2}\right) \\
&\leq \sum_{i=0}^{\infty} \frac{((1-\epsilon)(1+\gamma))^i (1+1/\gamma)\eta^2 G^2}{(1-\rho)^2} \\
&\leq \frac{(1-\epsilon)(1+1/\gamma)\eta^2 G^2)}{1-((1-\epsilon)(1+\gamma))} \\
&\leq \frac{4(1-\epsilon)\eta^2 G^2}{\epsilon^2(1-\rho)^2},
\end{aligned}$$

where in the third inequality, we use Young's inequality; in the fourth inequality, we invoke Lemma 30; and in the last line, we bound everything by choosing $\gamma = \frac{\epsilon}{2(1-\gamma)}$. □

Proof of Theorem 27. Consider a Comfetch model weight: w is sent down by the server as a sketch key pair $\{\mathcal{U}_H(\cdot), H(w)\}$ where H is a count sketch structure of size $\mathcal{O}\big(\frac{1}{\epsilon}\log\frac{d}{\delta}\big)\big)$ (following notation as described in Section 4.2.3). The objective function of our sketch network is $\tilde{f}(w) \triangleq f(\mathcal{U}_H(H(w)))$.

Although $\mathcal{U}_H(Hw)$ is not a linear transformation, we note that for any fixed single count sketch matrix H^*, $||H^\top Hw|| \leq \frac{d}{c}||w||$, therefore, under the count sketch median recovery scheme, $||\mathcal{U}_H(H(w))|| \leq \frac{d}{c}||w||$. Now, invoking Assumption 1, we have that

$$\begin{aligned}||\nabla\hat{f}(x) - \nabla\tilde{f}(y)|| &= ||\nabla f(\mathcal{U}_H(H(x))) - \nabla f(\mathcal{U}_H(H(w)))|| \\ &\leq L||\mathcal{U}_H(H(x)) - \mathcal{U}_H(H(y))|| \\ &\leq \frac{Ld}{c}||x-y||.\end{aligned}$$

This establishes that $\tilde{f}$ is $\frac{Ld}{c}$-smooth. Furthermore, Assumption 2 trivially holds for the stochastic gradients of $\hat{f}$ as well. Denote $\tilde{w}_t = \mathcal{U}_H(H(w))$. We follow analysis in [138]; by $\frac{Ld}{c}$-smoothness of $\tilde{f}$,

$$\begin{aligned}\mathbb{E}\tilde{f}(w_{t+1}) &\leq \tilde{f}(\tilde{w}_t) + \Big\langle\nabla\tilde{f}(\tilde{w}_t), \mathbb{E}[\tilde{w}_{t+1} - \tilde{w}_t]\Big\rangle + \frac{Ld}{2c}\mathbb{E}\Big|\Big|\tilde{w}_{t+1} - \tilde{w}_t\Big|\Big|^2 \\ &\leq \tilde{f}(\tilde{w}_t) + \Big\langle\nabla\tilde{f}(\tilde{w}_t), \mathbb{E}[\tilde{w}_{t+1} - \tilde{w}_t]\Big\rangle + \frac{Ld\eta^2}{2c(1-\rho)^2}\mathbb{E}||\tilde{g}_t||^2 \\ &\leq \tilde{f}(\tilde{w}_t) - \frac{\eta}{(1-\rho)}\Big\langle\nabla\tilde{f}(\tilde{w}_t), \mathbb{E}[g_t]\Big\rangle + \frac{Ld\eta^2}{2c(1-\rho)^2}\mathbb{E}||\tilde{g}_t||^2 \\ &\leq \tilde{f}(\tilde{w}_t) - \frac{\eta}{1-\rho}\langle\nabla\tilde{f}(\tilde{w}_t), \nabla\tilde{f}(w_t)\rangle + \frac{Ld\eta^2G^2}{2c(1-\rho)^2} \\ &= \tilde{f}(\tilde{w}_t) - \frac{\eta}{(1-\rho)^2}\mathbb{E}||\nabla\tilde{f}(w_t)||^2 + \frac{\eta}{2(1-\rho)^2}\mathbb{E}||\nabla\tilde{f}(w_t)||^2 \\ &\qquad + \frac{\eta}{2(1-\rho)^2}\mathbb{E}||\nabla\tilde{f}(\tilde{w}_t) - \nabla\tilde{f}(w_t)||^2 + \frac{Ld\eta^2G^2}{2c(1-\rho)^2} \\ &\leq f(\tilde{w}_t) - \frac{\eta}{2(1-\rho)}\mathbb{E}||\nabla\tilde{f}(w_t)||^2 + \frac{\eta L^2d^2}{2(1-\rho)c^2}||\tilde{w}_t - w_t||^2 + \frac{Ld\eta^2G^2}{2c(1-\rho)^2}\end{aligned}$$

$$\leq f(\tilde{w}_t) - \frac{\eta}{2(1-\rho)}\mathbb{E}||\nabla \tilde{f}(w_t)||^2 + \frac{\eta L^2 d^2}{2(1-\rho)c^2}||e_t + \frac{\eta\rho}{1-\rho}u_{t-1}||^2 + \frac{Ld\eta^2 G^2}{2c(1-\rho)^2}$$

.

We must bound $||e_t + \frac{\eta\rho}{1-\rho}u_{t-1}||^2$ now. However, since we only maintain $H(e_t)$ and $H(u_{t-1})$, we may instead consider the sketched norm,

$$||H(e_t) + \frac{\eta\rho}{1-\rho}H(u_{t-1})||^2.$$

By size of $H(\cdot)$ and Assumption 3, we have with probability $1-\delta$ that our sketch will recover all (ℓ_2, ϵ)-heavy hitters of w and that $||H(w)|| \leq (1+\epsilon)||w||$ [172]. Invoking Lemma 30,

$$||H(u_{t-1})||^2 \leq \big(\sum_{i=1}^{t-1}\rho^i||H(\tilde{g}_i)||^2\big) \leq \big(\sum_{i=1}^{t-1}\rho^i(1+\epsilon)G\big)^2 \leq \big(\frac{(1+\epsilon)G}{1-\rho}\big)^2. \tag{4.12}$$

We also have by Proposition 31 that

$$||H(e_t)||^2 \leq \frac{(1+\epsilon)^2(1-\epsilon)(1+1/\gamma)\eta^2 G^2}{1-((1-\epsilon)(1+\gamma))}. \tag{4.13}$$

By choosing $\gamma = \frac{\epsilon}{2(1-\epsilon)}$ in Equation 4.13, we have that

$$||H(e_t)||^2 \leq \frac{4(1+\epsilon)^2(1-\epsilon)\eta^2 G^2}{\epsilon^2(1-\rho)^2}. \tag{4.14}$$

Using Equations 4.14 and 4.12 to upper bound $||e_t + \frac{\eta\rho}{1-\rho}u_{t-1}||^2$, we conclude that

$$\begin{aligned}\mathbb{E}||\nabla \tilde{f}(w_t)||^2 \leq{}& \frac{2(1-\rho)}{\eta}\big(\tilde{f}(\tilde{w}_t) - \mathbb{E}\tilde{f}(\tilde{w}_{t+1})\big) \\ &+ \frac{2(1-\rho)}{\eta}\big(\frac{4\eta L^2 d^2(1+\epsilon)^2\eta^2 G^2}{2(1-\rho)c^2(1-\epsilon)\epsilon^2(1-\rho)^2}\big) \\ &+ \frac{2(1-\rho)}{\eta}\left(\frac{Ld\eta^2 G^2}{2c(1-\rho)^2}\right).\end{aligned}$$

Averaging over T and setting $\eta = \frac{c(1-\rho)}{2Ld\sqrt{T}}$ gives us our result. □

Remark. The theory is more restrictive than our empirical results, which is often the case

with sketching compression schemes, as the classical sketching concentration bounds are ensured via multiple sketches. In particular, using only a single sketch works very well as shown in Section 4.6.

4.8.4 Multi-Sketch `Comfetch`

In this section, we describe how to incorporate the usage of multiple sketches per layer for `Comfetch`, which is described by Algorithm 8.

Algorithm 8 Multi-sketch `Comfetch`

Require: initial weights $\{W_0^\ell\}_{\ell=1}^L$, learning rate η, number of iterations T, momentum parameter ρ, batch size M of data, batch size N of clients

Init momentum term $\{u_0^\ell = 0\}_{\ell=1}^L$

Init error accumulation term $e_0 = 0$

for $t = 1, 2, \ldots, T$ **do**

  Init sketching and unsketching procedures $\{\mathcal{U}^\ell, \mathcal{S}^\ell\}_{\ell=1}^L$

  Uniformly select at random N clients $c_1, c_2, \ldots, c_N$

  loop {in parallel on clients $\{c_i\}_{i=1}^N$}

  Download parameterized weight pairs $\{(\mathcal{U}^\ell, \mathcal{S}^\ell(W_t^\ell)\}_{\ell=1}^L$

  for $\ell = 1, 2, \ldots, L$ **do**

    Compute grads $g_i^\ell = \{\nabla_{H_j W_t} \mathcal{L}(R_i^\ell W_t^\ell, z \sim \mathcal{D}_i)\}_{j=1}^k$ ▷ See equation 4.22

  end for

  Send $\{g_t^\ell\}_{\ell=1}^L$ to Central Server

  end loop

  for $\ell = 1, 2, \ldots, L$ **do**

    Aggregate restored gradients: $g_t^\ell = \frac{1}{N} \sum_{i=1}^N \mathcal{G}(g_i^\ell)$ ▷ See equation 4.23

    Update sketched momentum: $u_t^\ell = \rho u_{t-1}^\ell + g_t^\ell$

    Update error feedback: $e_t = \eta u_t + e_t$

    Approximate gradient with feedback: $\Delta^t = \text{Top-k}(e_t)$

    Error accumulation: $e^{t+1} = e_t - \Delta_t$

    Update weights: $W_{t+1} = W_t - \Delta_t$

  end for

end forreturn $\{w_T^\ell\}_{\ell=1}^L$

4.8.5 Model Transmission and Download

At iteration t, the central server first prepares the global model for the transmission by sketching down all the current weights $\{W_t^\ell\}_{\ell=1}^L$, . We assume that our layers are either convolutional or fully-connected as described in Section 4.2, and for simplicity, that all our W_t^ℓ are of size $d \times d$, but they can be rectangular in practice. For each weight W_t^ℓ, the central server randomly draws count sketch matrices $\{H_i^\ell\}_{i=1}^k$, where $H_i^\ell \in \mathbb{R}^{c \times d}$, $c << d$ is the sketching length, and transmits $\{({H_i^\ell}^\top, H_i^\ell W_t^\ell)\}_{i=1}^k$ to a selection of N uniformly randomly drawn clients, who then download these sketched parameters into local memory. Henceforth, will denote $\{(\mathcal{U}^\ell, S^\ell(W_t^\ell)\} \triangleq \{({H_i^\ell}^\top, H_i^\ell W_t^\ell)\}_{i=1}^k$ for $\ell \in [L]$.

Cost Complexity We note that k corresponds to the number of independent sketches, which is required theoretically to achieve a guarantee on approximating W_t^ℓ, but in practice, we observe in Section 4.6 that one sketch is sufficient for model convergence. Also note that any H_i^ℓ bijectively corresponds to a hash function $h_i^\ell : d \to c$ which is representable as a length d vector. Hence, in practice, the central server will transmit $\{(h_i^\ell, H_i^\ell W_t^\ell)\}_{i=1}^k$, for a total local memory and transmission cost of $\mathcal{O}\big((kc+1)d\big)$, which in the one-sketch ($k = 1$) case is less than the $O(d^2)$ cost of transmitting the full weight.

4.8.5.1 Client Update

The client C_i will now conduct a single round of training on the sketched network parameters using their local data. Often in practice, the client distributions $\mathcal{D}_i$ will be finite and small [86], so we can assume that the client is always taking the full gradient with respect to the

weights, but the algorithm generalizes to stochastic gradients as well.

Forward pass Let $x \in \mathcal{D}_i$ and let $\{(\mathcal{U}^\ell, S^\ell(W_t^\ell)\} \triangleq \{(H_i^{\ell\top}, H_i^\ell W_t^\ell)\}_{i=1}^k$ for $l \in [L]$, where L is the depth of our network. Following the notations described in Section 4.2, the forward pass of a fully-connected layer is

$$x^\ell = \sigma(\mathcal{U}^\ell(S^i(W_t^\ell x^{i-1})), 1 \leq \ell \leq L-1 \tag{4.15}$$

$$\hat{y} = \mathbf{a}^\top x^L, \tag{4.16}$$

where $(\mathcal{U}^\ell(S^\ell(W_t^\ell))x)_i \triangleq \underset{1\leq j\leq k}{\text{median}}\{(H_j^{\ell\top} H_j^\ell W_t^\ell x\}$ for any $x \in \mathbb{R}^d$, in agreement with the count sketch procedure of Algorithm 7. Similarly, for a convolutional ResNet, we have that

$$x^1 = \sqrt{\frac{c_\sigma}{m}}\sigma\Big(\mathcal{U}^1(\mathcal{S}^1(W_t^1))\phi(x^0)\Big) \tag{4.17}$$

$$x^\ell = x^{\ell-1} + \frac{c_{res}}{L\sqrt{m}}\sigma\Big(\mathcal{U}^\ell(\mathcal{S}^\ell(W_t^\ell))\phi_\ell(x^{\ell-1})\Big),$$

$$2 \leq \ell \leq L \tag{4.18}$$

$$\hat{y} = \langle W_t^L, x^L\rangle, \text{ where } W^L \in \mathbb{R}^{m\times p}. \tag{4.19}$$

Remark 1: We never directly compute a $d \times d$ weight matrix at any stage.

Remark 2: In the convolutional case, where the inputs between layers are matrices, $H_i W x = H_i(Wx)$ can be regarded as a higher-order count sketch (HCS) of Wx as described in Section 4.8.2.

Backward pass. In order to compute the gradient, we must first clearly define the weights of the modified network. Let $\{H_i\}_{i=1}^k$ be a random set of $c \times d$ count sketch matrices and let $x \in \mathbb{R}^d$. We have that $\hat{x}_i := \mathcal{U}(\mathcal{S}(x))_i = \underset{1\leq j\leq k}{\text{median}}\{(H_j^\top H_j x)_i\}$. If sketch H_{j_i} results in the median

recovery of the ith coordinate of x, then we have the following representation:

$$\hat{x} = \sum_{i=1}^{d} E_i H_{j_i}^\top H_{j_i} x, \tag{4.20}$$

where $E_i \in \mathbb{R}^{d\times d}$ is a matrix with 1 at entry i, i and 0 everywhere else. Therefore, it possible to define $\hat{x} = \mathcal{U}(\mathcal{S}(x)) = Ax$ where $A \in \mathbb{R}^{d\times d}$. That is, we can represent the count sketch recovery of x as a matrix transformation.

By the above discussion, we can represent $\mathcal{U}^\ell(\mathcal{S}^\ell(W_t^\ell x^{\ell-1})) = R_i^\ell W_t^\ell x^{\ell-1}$, where $R_i^\ell \in \mathbb{R}^{d\times d}$ is referred to as the *recovery matrix*. Hence, the weights of the client network are $R_i^\ell W_t^\ell$. Note that we specifically indicate the client index i, since the recovery matrix will vary depending on the local data.

Now that we have have defined the weights of our client models, we may now take gradients. The server will want to receive $\frac{\partial \mathcal{L}(R_i^\ell W_t^\ell, z)}{\partial W_t^\ell}$ as an approximation of $\frac{\partial \mathcal{L}(W_t^\ell, z)}{\partial W_t^\ell}$, but the client will not want to compute $\frac{\partial \mathcal{L}(R_i^\ell \hat{W}_t^\ell, z)}{\partial W_t^\ell}$, since it is of size $d \times d$. We will want to transmit a $\mathcal{O}(kc \times d)$ packet of data which will allow the server to compute $\frac{\partial \mathcal{L}(R_i^\ell W_t^\ell, z)}{\partial W_t^\ell}$. To this end, let $\{H_i^\ell\}_{i=1}^k$ be the set of count sketch matrices associated with layer ℓ. Let $E_{H_i} \in \mathbb{R}^{d\times d}$ denote the matrix which has a 1 at entry j, j for $1 \le j \le d$ if H_i contains the median recovery of $(W_t^\ell x^{\ell-1})_j$ and 0 everywhere else. We have then that,

$$R_i^\ell = \sum_{i=1}^{k} E_{H_i} H_i W_t^\ell x^{\ell-1}. \tag{4.21}$$

Therefore, by the chain rule of the total derivative,

$$\frac{\partial \mathcal{L}(R_i^\ell W_t^\ell, z)}{\partial W_t^\ell} = \sum_{i=1}^{k} \frac{\partial \mathcal{L}(R_i^\ell W_t^\ell, z)}{\partial H_i^\ell W_t^\ell} \frac{\partial H_i^\ell}{\partial W_t^\ell}. \tag{4.22}$$

Thus, the client will upload $g_t^\ell \triangleq \{\nabla_{H_i W_t^\ell} \mathcal{L}(R_i^\ell W_t^\ell, z)\}_{i=1}^k$ for all $\ell \in [L]$.

Remark. We subtly avoided the point that the recovery matrix R_i^ℓ as described is recursively de-

termined by the initial input. Therefore, our sketched weights will not be represented as simple linear transformations of the original weights. However, since the client will clearly be determining gradients through an `autograd`-like library, this will not pose an issue anyways.

Cost Complexity. If the client chooses to upload the gradients contained within g_t^ℓ in a predefined manner (for example, in the order the sketches were transmitted), then the central server will know which gradients correspond to which sketch, and thus will be able to compute equation without any additional information. Thus, the total communication cost is $\mathcal{O}(kcd)$, which in the single-sketch $k = 1$ scenario, is a strong improvement over the usual uplink cost of $\mathcal{O}(d^2)$ and even cheaper than the download cost.

Remark. The complexity of computing $R_i^\ell W_t^\ell x^{\ell-1}$ is $\mathcal{O}(2kcd + k)$, since the client must individually compute $H_i^\top H_i W_t x_t^{\ell-1}$ for all $i \in [k]$ to determine the median coordinates. This may not be cheaper than the usual $\mathcal{O}(d^2)$ matrix-vector multiplication of the original layer if too many sketches are used, but as we demonstrate in Section 4.6, a single sketch is sufficient.

4.8.6 Model Update

The Central Server aggregates the $\{g_i^\ell\}_{\ell=1}^L$ across all $i \in [N]$. We describe the procedure for updating the weight of a fixed layer ℓ. For each $i \in [N]$, the server computes

$$\mathcal{G}(g_i^\ell) = \sum_{j=1}^{k} {H_j^\ell}^\top \nabla_{H_j^\ell W_t^\ell} \mathcal{L}(R_i^\ell W_t^\ell, z). \tag{4.23}$$

The server takes an average over the $\mathcal{G}(g_\ell)$ to compute a stochastic gradient of $f(R^\ell W_t^\ell)$ with respect to W_t^ℓ. That is,

$$\nabla_{W_t^\ell} f(R_i^\ell W_t^\ell) = \mathbb{E} \frac{1}{N} \sum_{i=1}^{N} \mathcal{G}(g_i^\ell). \tag{4.24}$$

The remainder of the model update follows the error-feedback and momentum scheme of `FetchSGD` [138]. Error-feedback allows for the correction of error associated with gradient approximations. In the case of `FetchSGD`, error-feedback corrects the error associated with a taking a sketch and unsketch of the gradients. In the case of `Comfetch`, we are correcting the error associated with using $\nabla_{W_t^\ell} f(R^\ell W_t^\ell)$ as an approximation of $\nabla f(W_t^\ell)$. The reader is encourage to consult the work of Karimireddy et al. and Stitch et al. for further details on error-feedback for SGD-like methods [147, 177].

Remark. We subtly avoided the point that the recovery matrix R_i^ℓ as described is recursively determined by the initial input. Therefore, our sketched weights will not be represented as simple linear transformations of the original weights. However, since in practice the client will clearly be determining gradients through an `autograd`-like library, this will not pose an issue.

4.8.7 Additional Language Task Data

In this section, we present additional data in Table 4.3 and Figure 4.5 for language tasks not included in the main paper. The results demonstrate that our `Comfetch` preserves accuracy while reducing the size of weights in gates of the LSTM layer by different ratios. We maintain a high test accuracy while reducing the number of parameters from 9607 to 2439. We exclude embedding parameters from the overall parameter counts as they serve as inputs and can be pre-trained.

Method	Bandwidth Compression	Memory Compression	Number of Params	Test Acc (%)
FedAvg	1	1	9607	47.58
FedAvg-1/2	1	1/2	4423	49.78
FedAvg-1/4	1	1/4	2599	44.01
FedAvg-1/8	1	1/8	1879	25.63
FetchSGD	1	1	9607	86.81
Comfetch	1	1	9607	80.44
Comfetch-1/2	1/2	1/2	5511	81.50
Comfetch-1/4	1/4	1/4	3463	80.57
Comfetch-1/8	1/8	1/8	2439	78.61

Table 4.3: Model accuracies under different memory footprints in clients, for predicting part of speech taggings for MNLI [4] sentences using an LSTM. We exclude embeddings from parameter counts as they serve as inputs and can be pretrained.

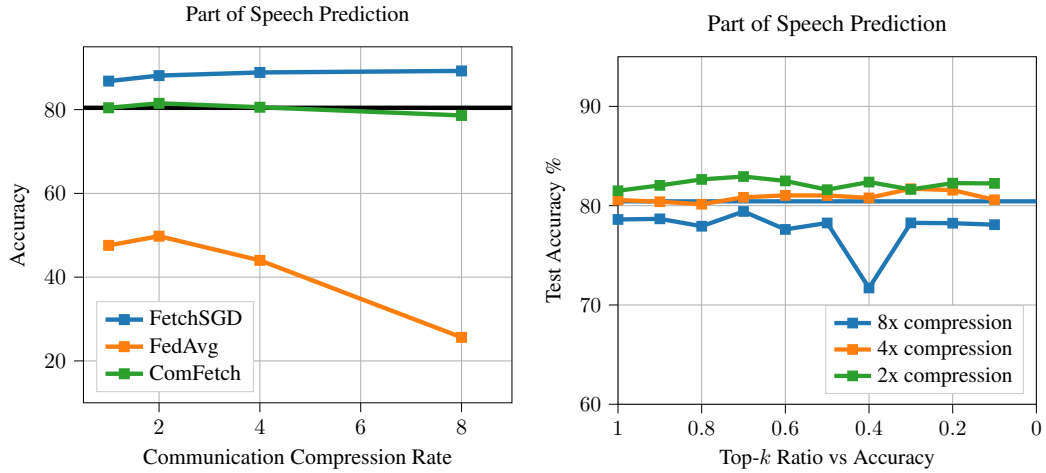


Figure 4.5: Test accuracy achieved on predicting part of speech taggings for MNLI [4] sentences. On the left, we present quantitative comparisons to other methods. The horizontal line reflects the simple baseline where no compression is applied during training. On the right, we report the test accuracy with different compression rate while varying the K ratio. FetchSGD only compresses the network weights during communication, but each client still needs to decompress the entire network locally to perform training.

4.8.8 Prediction Error Bound

In this section, we provide a bound on the error between the prediction of a fully-connected multi-layer network and its sketched counterpart. We denote by $\phi : \mathbb{R}^d \to \mathbb{R}^d$, the ReLU (rectified linear unit), where for any $x \in \mathbb{R}^d$, we have that $(\phi(x))_i = \max((x)_i, 0)$. For ease (and an abuse) of notation, we let $\psi \circ W_L \circ \psi \circ W_{L-1} \circ \cdots \circ \psi \circ W_1 x = \phi(W_L(\phi(W_{L-1}(\cdots(\phi(W_1 x)\cdots)$, where $W_\ell \in \mathbb{R}^{d\times d}$ for $1 \leq \ell \leq L$, $x \in \mathbb{R}^d$, and we are freely allowing W_1 to act as both a linear

transformation and a matrix multiplication (i.e., $W_1 \circ x = W_1 x$).

Theorem 32. *Let*

$$\hat{y}_L = \psi \circ W_L \circ \psi \circ W_{L-1} \circ \cdots \circ \psi \circ W_1 \circ x, \tag{4.25}$$

$W_\ell \in \mathbb{R}^{d \times d}$ *for* $1 \leq \ell \leq L$, $x \in \mathbb{R}^d$,, *and* ψ *is the ReLU activation function. Now let*

$$\tilde{y}_L = \psi \circ H_L^{-1} H_L W_L \circ \psi \circ H_{L-1}^{-1} H_{L-1} W_{L-1} \circ \tag{4.26}$$

$$\cdots \circ \psi \circ H_1^{-1} H_1 W_1 x, \tag{4.27}$$

where $H_\ell^{-1} H_\ell W_i$ *reflects a count sketch recovery of* W_ℓ. *If for each* $H_\ell^{-1} H_\ell W_\ell$ *we have chosen each sketching length of* W_ℓ *as* $c = \Omega(||W_\ell||_F^2/\epsilon^2)$ *and the number of independent sketches as* $\mathcal{O}(\log \frac{d}{\delta})$, *for* $0 < \delta < 1$, *then with* $(1-\delta)^L$ *probability we have that*

$$||\tilde{y}_L - \hat{y}_L|| \leq \sum_{j=1}^{L} g_j(x), \tag{4.28}$$

where $g_j(x) = \lambda_j \lambda_{j+1} \cdots \lambda_L ||x|| d^2 \epsilon^2 \prod_{n=1}^{j-1} \hat{\lambda_n}$, λ_i *is the maximum singular value of* W_i, *and* $\hat{\lambda}_i$ *is the maximum singular value of* $H_\ell^{-1} H_\ell$. *We let* $g_0 = d^2 \epsilon^2 ||x||$.

Proof. We proceed by induction on the number of layers L. For simplicity, we denote $\hat{W}_i := H_i^{-1} \circ H_i \circ W_i$. For $L = 1$, we have that

$$||\psi \circ \hat{W}_i x - \psi W_i x|| \leq ||\hat{W}_i - W_i|| ||x|| \leq d^2 \epsilon^2 ||x|| = g_0, \tag{4.29}$$

where the first inequality follows by the fact the ReLU ψ is 1-Lipschitz and the second inequality follows by the conventional HCS guarantee [176] and our prescribed width and depth of sketches, $\mathcal{O}(\frac{1}{\epsilon^2} \log \frac{d}{\delta})$. Assume the hypothesis holds for $L = k$ layers, then for $L = k + 1$ layers we have

that

$$||\tilde{y}_{k+1} - \hat{y}_{k+1}|| = ||\psi \circ \hat{W_{k+1}} \circ \tilde{y}_k - \psi \circ W_{k+1} \circ \hat{y}_k|| \tag{4.30}$$

$$\leq ||\hat{W}_{k+1} \circ \tilde{y}_k - W_{k+1} \circ \hat{y}_k|| \tag{4.31}$$

$$\leq ||\hat{W}_{k+1} \circ \tilde{y}_k - W_{k+1} \circ \tilde{y}_k + W_{k+1} \circ \tilde{y}_k - W_{k+1} \circ \hat{y}_k|| \tag{4.32}$$

$$\leq ||\hat{W}_{k+1} \circ \tilde{y}_k - W_{k+1} \circ \tilde{y}_k|| + ||W_{k+1} \circ \tilde{y}_k - W_{k+1} \circ \hat{y}_k|| \tag{4.33}$$

$$\leq ||\hat{W}_{k+1} - W_{k+1}||||\tilde{y}_k|| + ||W_{k+1}||||\tilde{y}_k - \hat{y}_k|| \tag{4.34}$$

$$\leq d\epsilon^2 ||x|| \prod_{n=1}^{k} \hat{\lambda}_n + \lambda_{k+1} \sum_{j=1}^{k} g_j \tag{4.35}$$

$$= \sum_{j=1}^{k+1} g_j, \tag{4.36}$$

where the second to last inequality follows by applying the inductive hypothesis to the right term $||\tilde{y}_k - \hat{y}_k||$ and noting that for the left term,

$$||\tilde{y}_k|| = ||\psi \circ \hat{W}_k \circ \psi \hat{W}_{k-1} \circ \cdots \circ \hat{W}_i x|| \tag{4.37}$$

$$\leq ||x|| \prod_{i=1}^{k} ||\hat{W}_i|| \leq \hat{\lambda}_i ||x||. \tag{4.38}$$

The probabilistic guarantee of $(1 - \delta)^L$ follows by the independence of each individual layer sketching. □

The above result theoretically demonstrates that the noise is controllable via increased sketched length and number of independent sketches, and in general, requires increased space complexity as L increases.

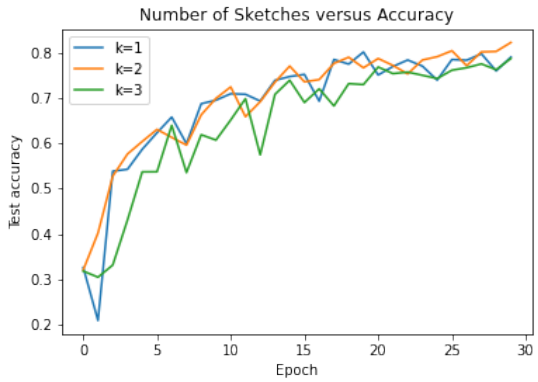


Figure 4.6: **Multi-Sketch Comfetch.** We assess the affect of using multiple sketches during training, for a single client training of CIFAR-10 with 87.5% compression. The use of multiple sketches has no noticeable effect on model performance.

4.8.9 Multi-Sketch Ablation & ResNet-9 Experiments

Theorem 27 and nearly all of count sketch theory use multiple sketches to obtain convergence guarantees. To assess the effects of using using multiple sketches on model performance, in Figure 4.6, we train a single client on CIFAR-10 using varying number of sketches between layers for weight recovery. We set our compression rate to 87.5% since Comfetch models at this level of compression experience noticeable performance decline. We find that using multiple sketches does not improve performance, justifying usage of a single sketch in our experiments. We believe that one-sketch guarantees would be valuable for uplink/downlink compression literature since the single-sketch compression strategy is successful in practice.

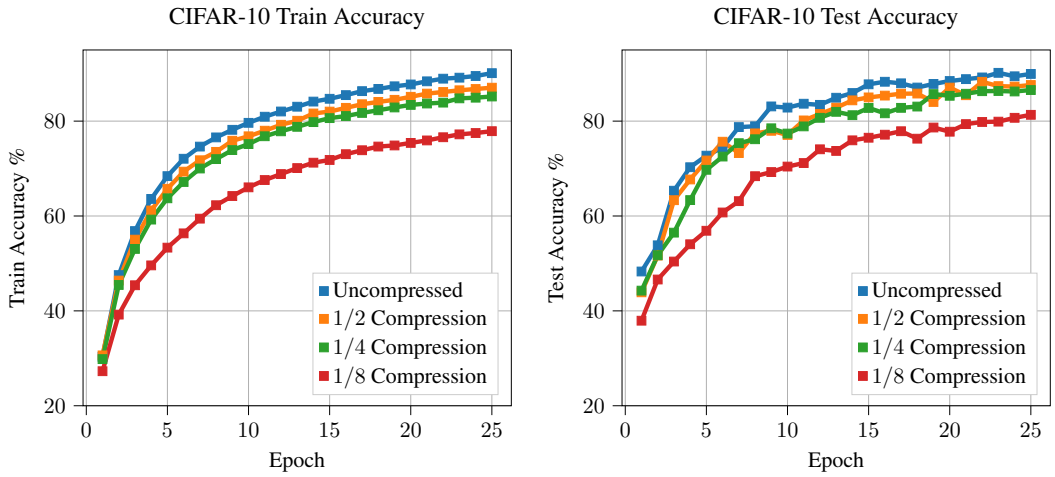
4.8.9.1 1 Client, ResNet-9

In Table 4.4, we conduct a simple study examining how Comfetch compression affects single (unfederated) client training when ResNet-9 is used to train over CIFAR-10 over 25 epochs. We additionally conduct random pruning of a non-sketched model to mimic compression. Fixed

random pruning to achieve the same compression amount outputs worse models. It is important to note that pruning works well for pre-trained models, but this study demonstrates the ineffectiveness of one-time pruning prior to training. Figure 4.2 demonstrates the training and test curves.

Method	Model Size	Test Acc (%)
`No sketch`	1	86.04
`No sketch`	1/2	75.38
`Comfetch`	1/2	**87.89**
`No sketch`	1/4	73.34
`Comfetch`	1/4	**86.12**
`No sketch`	1/8	72.40
`Comfetch`	1/8	**81.18**

Table 4.4: Test accuracy under different memory footprints in clients for the CIFAR-10 [3] image classification task. `Comfetch` maintains a large and powerful global model under communication compression and client memory compression. The no sketch model is compressed via random pruning, which performs much worse than our sketch-compressed models.



(a) Train Accuracy (b) Test Accuracy

Figure 4.7: Test accuracy convergence of 1-client ResNet-9 `Comfetch` under varying compression rates. **(a)-(b)** correspond to the CIFAR-10 [3] image classification. Similar accuracy with 1) different `Comfetch` compression rates suggests that our method retains the expressive power of the model while reducing the parameter sizes.

Chapter 5: Decentralized Federated Learning via Wait-Free Model Communication

5.1 Introduction

Federated Learning (FL) is an increasingly popular setting to train powerful deep neural networks with data derived from an assortment of clients. Recent research [6, 46, 178] has focused on constructing decentralized FL algorithms that overcome speed and scalability issues found within classical centralized FL [84, 179]. While decentralized algorithms have eliminated a major bottleneck in the distributed setting, the central server, their scalability potential is still largely untapped. Many are plagued by high communication time per round [180]. Shortening the communication time per round allows more clients to connect and then communicate with one another, thereby increasing scalability.

Due to the synchronous nature of current decentralized FL algorithms, communication time per round, and consequently run-time, is amplified by parallelization delays. These delays are caused by the slowest client in the network. To circumvent these issues, asynchronous decentralized FL algorithms have been proposed [181, 182, 183, 184]. However, these algorithms still suffer from high communication time per round. Furthermore, their communication protocols either do not propagate models well throughout the network (via gossip algorithms) or require

partial synchronization. Finally, these asynchronous algorithms rely on a deterministic bounded-delay assumption, which ensures that the slowest client in the network updates at least every τ iterations. This assumption is satisfied only under certain conditions [185], and worsens the convergence rate by adding a sub-optimal reliance on τ.

To remedy these drawbacks, we propose the **S**hared **W**a**I**t-**F**ree **T**ransmission (SWIFT) algorithm: an efficient, scalable, and high-performing decentralized FL algorithm. Unlike other decentralized FL algorithms, SWIFT obtains minimal communication time per round due to its wait-free structure. Furthermore, SWIFT is the first asynchronous decentralized FL algorithm to obtain an optimal $\mathcal{O}(1/\sqrt{T})$ convergence rate (aligning with stochastic gradient descent) *without a bounded-delay assumption*. Instead, SWIFT leverages the expected delay of each client (detailed in our remarks within Section 4.5). Experiments validate SWIFT's efficiency, showcasing a reduction in communication time by nearly an order of magnitude and run-times by upwards of 50%. All the while, SWIFT remains at state-of-the-art (SOTA) global test/train loss for image classification compared to other decentralized FL algorithms. We summarize our ***main contributions*** as follows.

1. Propose a novel wait-free decentralized FL algorithm (called SWIFT) and prove its theoretical convergence without a bounded-delay assumption.

2. Implement a novel pre-processing algorithm to ensure non-symmetric and non-doubly stochastic communication matrices are symmetric and doubly-stochastic under expectation.

3. Provide the first theoretical client-communication error bound for non-symmetric and non-doubly stochastic communication matrices in the asynchronous setting.

Algorithm	Iteration Convergence Rate	Client (i) Comm-Time Complexity	Neighborhood Avg.	Asynchronous	Private Memory
D-SGD	$\mathcal{O}(1/\sqrt{T})$	$\mathcal{O}(T \max_{j\in\mathcal{N}_i} \mathscr{C}_j)$	✓	✗	✓
PA-SGD	$\mathcal{O}(1/\sqrt{T})$	$\mathcal{O}(\lvert\mathcal{C}_s\rvert \max_{j\in\mathcal{N}_i} \mathscr{C}_j)$	✓	✗	✓
LD-SGD	$\mathcal{O}(1/\sqrt{T})$	$\mathcal{O}(\lvert\mathcal{C}_s\rvert \max_{j\in\mathcal{N}_i} \mathscr{C}_j)$	✓	✗	✓
AD-PSGD	$\mathcal{O}(\tau/\sqrt{T})$	$\mathcal{O}(T\mathscr{C}_i)$	✗	✓	✗
SWIFT	$\mathcal{O}(1/\sqrt{T})$	$\mathcal{O}(\lvert\mathcal{C}_s\rvert\mathscr{C}_i)$	✓	✓	✓

(1) Notation: total iterations T, communication set $\mathcal{C}_s$ ($\lvert\mathcal{C}_s\rvert < T$), client i's neighborhood $\mathcal{N}_i$, maximal bounded delay τ, and client i's communication time per round $\mathscr{C}_i$. **(2)** As compared to AD-PSGD, SWIFT does not have a τ convergence rate term due to using an expected client delay in analysis.

Table 5.1: Rate and complexity comparisons for decentralized FL algorithms.

4. Demonstrate experimentally a significant reduction in communication time and run-time per epoch for CIFAR-10 classification in both IID and non-IID settings compared to synchronous decentralized FL algorithms.

5.2 Related Works

Asynchronous Learning. HOGWILD! [186], AsySG-Con [45], and AD-PSGD [46] are seminal examples of asynchronous algorithms that allow clients to proceed at their own pace. However, these methods require a shared memory/oracle from which clients grab the most up-to-date global parameters (e.g. the current graph-averaged gradient). By contrast, SWIFT relies on a message passing interface (MPI) to exchange parameters between neighbors, rather than interfacing with a shared memory structure. Algorithmically, each client relies on local memory to store current neighbor weights. To circumvent local memory overload, common in IoT clusters [85], clients maintain a mailbox containing neighbor models: each client pulls out neighbor models one at a time to sequentially compute the desired aggregated statistics.

Decentralized Stochastic Gradient Descent (SGD). The predecessor to decentralized FL is gossip learning [187, 188]. Gossip learning was first introduced by the control community to assist with mean estimation of decentrally-hosted data distributions [189, 190]. Now, SGD-based gossip algorithms are used to solve large-scale machine learning tasks [45, 181,

186, 191, 192, 193]. A key feature of gossip learning is the presence of a globally shared oracle/memory with whom clients exchange parameters at the end of training rounds [187]. While read/write-accessible shared memory is well-suited for a single-organization ecosystem (i.e. all clients are controllable and trusted), this is unrealistic for more general edge-based paradigms. Neighborhood-based communication and aggregation algorithms, such as D-SGD [46] and PA-SGD [6], can theoretically and empirically outperform their centralized counterparts, especially under heterogeneous client data distributions. Unfortunately, these algorithms suffer from synchronization slowdowns. SWIFT is asynchronous (avoiding slowdowns), utilizes neighborhood averaging, and does not require shared memory.

Communication Under Expectation. Few works in FL center on communication uncertainty. In [194], a lightweight, yet unreliable, transmission protocol is constructed in lieu of slow heavyweight protocols. A synchronous algorithm is developed to converge under expectation of an unreliable communication matrix (probabilistic link reliability). SWIFT also convergences under expectation of a communication matrix, yet in a different and asynchronous setting. SWIFT is already lightweight and reliable, and our use of expectation does not regard link reliability.

Communication Efficiency. Minimizing each client i's communication time per round $\mathscr{C}_i$ is a challenge in FL, as the radius of information exchange can be large [195]. MATCHA [180] decomposes the base network into m disjoint matchings. Every epoch, a random sub-graph is generated from a combination of matchings, each having an activation probability p_k. Clients then exchange parameters along this sub-graph. This requires a total communication-time complexity of $\mathcal{O}(T \sum_{k=1}^{m} p_k \max_{j \in \mathcal{N}_i} \mathscr{C}_j)$, where $\mathcal{N}_i$ are client i's neighbors. LD-SGD [6] and PA-SGD [178] explore how reducing the number of neighborhood parameter exchanges affects convergence. Both algorithms create a communication set $\mathcal{C}_s$ (defined in Appendix 5.8.3) that dictate

when clients communicate with one another. The communication-time complexities are listed in Table 5.1. These methods, however, are synchronous and their communication-time complexities depend upon the slowest neighbor $\max_{j \in \mathcal{N}_i} \mathscr{C}_j$. SWIFT improves upon this, achieving a communication-time complexity depending on a client's own communication-time per round. Unlike AD-PSGD [181], which achieves a similar communication-time complexity, SWIFT allows for periodic communication, uses only local memory, and does not require a bounded-delay assumption.

5.3 Problem Formulation

Decentralized FL. In the FL setting, we have n clients represented as vertices of an arbitrary communication graph $\mathcal{G}$ with vertex set $\mathcal{V} = \{1, \ldots, n\}$ and edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. Each client i communicates with one-hop neighboring clients j such that $(i, j) \in \mathcal{E}$. We denote the neighborhood for client i as $\mathcal{N}_i$, and clients work in tandem to find the global model parameters x by solving:

$$\min_{x \in \mathbb{R}^d} f(x) := \sum_{i=1}^{n} p_i f_i(x), \quad f_i(x) := \mathbb{E}_{\xi_i \sim \mathcal{D}_i}\big[\ell(x, \xi)\big], \quad \sum_{i=1}^{n} p_i = 1, \quad p_i \geq 0. \tag{5.1}$$

The global objective function $f(x)$ is the weighted average of all local objective functions $f_i(x)$. In Equation 5.1, $p_i, \forall i \in [n]$ denotes the client influence score. This term controls the influence of client i on the global consensus model, forming the *client influence vector* $p = \{p_i\}_{i=1}^n$. These scores also reflect the sampling probability of each client. We note that each local objective function $f_i(x)$ is the expectation of loss function ℓ with respect to potentially different local data $\xi_i = \{\xi_{i,j}\}_{j=1}^M$ from each client i's distribution $\mathcal{D}_i$, i.e., $\xi_{i,j} \sim \mathcal{D}_i$. The total number of iterations is denoted as T.

Definition	Notation	Definition	Notation
Global Iteration	t	Number of Clients	n
Active Client at t	i_t	Parameter Dimension	d
Client (i,j) Communication Coefficient at t	$w_{i,j}^t$	Data Dimension	f
Local Model of Client i at t	$x_i^t \in \mathbb{R}^d$	Step-Size	γ
Local Model of Active Client at t	$x_{i_t}^t \in \mathbb{R}^d$	Total SWIFT Iterations	T
Mini-Batch Data from Active Client at t	$\xi_{i_t}^t \in \mathbb{R}^f$	Mini-Batch Size (Uniform)	M
Gradient of the Active Model at t	$g(x_{i_t}^t, \xi_{i_t}^t) \in \mathbb{R}^d$	Communication Set	$\mathcal{C}_s$
Client i Communication Vector at t	$w_i^t \in \mathbb{R}^n$	Global Data Distribution	$\mathcal{D}$
Local Model Matrix at t	$X^t \in \mathbb{R}^{d\times n}$	Global Objective	$f(x)$
Zero-Padded Gradient of the Active Model at t	$G(x_{i_t}^t, \xi_{i_t}^t) \in \mathbb{R}^{d\times n}$	Client Influence Score	p_i
Expected Local Model Gradients at t	$\bar{G}(X^t, \xi^t) \in \mathbb{R}^{d\times n}$	Client Influence Vector	$p \in \mathbb{R}^n$
Active Client Communication Matrix at t	$W_{i_t}^t \in \mathbb{R}^{n\times n}$	One-Hot Vector	$\boldsymbol{e_i} \in \mathbb{R}^n$
Expected Client Communication Matrix at t	$\bar{W}^t \in \mathbb{R}^{n\times n}$	Identity Matrix	$I_n \in \mathbb{R}^{n\times n}$

Existing Inter-Client Communication in Decentralized FL. All clients balance their individual training with inter-client communications in order to achieve consensus while operating in a decentralized manner. The core idea of decentralized FL is that each client communicates with its neighbors (connected clients) and shares local information. Balancing individual training with inter-client communication ensures individual client models are well-tailored to personal data while remaining (i) robust to other client data, and (ii) able to converge to an optimal consensus model.

Periodic Averaging. Algorithms such as Periodic Averaging SGD (PA-SGD) [178] and Local Decentralized SGD (LD-SGD) reduce communication time by performing *multiple* local updates before synchronizing. This process is accomplished through the use of a communication set $\mathcal{C}_s$, which defines the set of iterations a client must perform synchronization,

$$\mathcal{C}_s = \{t \in \mathbb{N} \mid t \bmod (s+1) = 0,\ t \leq T\}. \tag{5.2}$$

We adopt this communication set notation, although synchronization is unneeded in our algorithm.

5.4 Shared Wait-Free Transmission (SWIFT) Federated Learning

In this section, we present the **S**hared **W**a**I**t-**F**ree **T**ransmission (SWIFT) Algorithm. SWIFT is an asynchronous algorithm that allows clients to work at their own speed. Therefore, it removes the dependency on the slowest client which is the major drawback of synchronous settings. Moreover, unlike other asynchronous algorithms, SWIFT does not require a bound on the speed of the slowest client in the network and allows for neighborhood averaging and periodic communication.

A SWIFT Overview. Each client i runs SWIFT in parallel, first receiving an initial model x_i, communication set $\mathcal{C}_s$, and counter $c_i \leftarrow 1$. SWIFT is concisely summarized in the following steps:

(0) Determine client-communication weights w_i via Algorithm 10.

(1) Broadcast the local model to all neighboring clients.

(2) Sample a random local data batch of size M.

(3) Compute the gradient update of the loss function ℓ with the sampled local data.

(4) Fetch and store neighboring local models, and average them with one's own local model if $c_i \in \mathcal{C}_s$.

(5) Update the local model with the computed gradient update, as well as the counter $c_i \leftarrow c_i + 1$.

(6) Repeat steps **(1)**-**(5)** until convergence.

A diagram and algorithmic block of SWIFT are depicted in Figure 5.1 and Algorithm 9 respectively.

Active Clients, Asynchronous Iterations, and the Local-Model Matrix. Each time a client finishes a pass through steps **(1)**-**(5)**, one global iteration is performed. Thus, the global

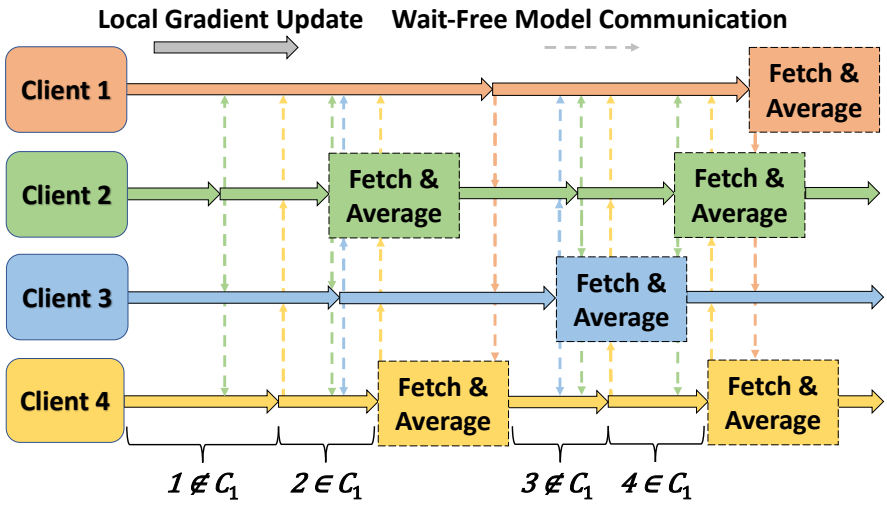


Figure 5.1: SWIFT schematic with $\mathcal{C}_s = \mathcal{C}_1$ (i.e., clients communicate every two local update steps).

iteration t is increased after the completion of *any* client's averaging and local gradient update. The client that performs the t-th iteration is called the active client, and is designated as i_t (Line 6 of Algorithm 9). *There is only one active client per global iteration.* All other client models remain unchanged during the t-th iteration (Line 16 of Algorithm 9). In synchronous algorithms, the global iteration t increases *only after all clients finish an update*. SWIFT, which is asynchronous, increases the global iteration t after *any* client finishes an update. In our analysis, we define local-model matrix $X^t \in \mathbb{R}^{d\times n}$ as the concatenation of all local client models at iteration t for ease of notation,

$$X^t := [x_1^t, \ldots, x_n^t] \in \mathbb{R}^{d\times n}. \tag{5.3}$$

Inspired by PA-SGD [178], SWIFT handles multiple local gradient steps before averaging models amongst neighboring clients (Line 10 of Algorithm 9). Periodic averaging for SWIFT, governed by a dynamic client-communication matrix, is detailed below.

Wait Free. The backbone of SWIFT is its wait-free structure. Unlike any other decen-

Algorithm 9 **S**hared **Wa**I**t**-**F**ree **T**ransmission (SWIFT)

Require: Vertex set $\mathcal{V}$, Total steps T, Step-size γ, Client Influence Vector p, Distributions of client data $\mathcal{D}_i$, Communication set $\mathcal{C}_s$, Batch size M, Loss function ℓ, and Initial model x^0

Initialize each client's local update counter $c_i \leftarrow 1, \forall i \in \mathcal{V}$

Obtain each client's new communication vector w_i^t using Algorithm 10

for $t = 1, \ldots, T$ **do**

  if network topology changes **then**

    Renew each client's communication vector w_i^t using Algorithm 10

  end if

  Randomly **select an active client** i_t according to Client Influence Probability Vector p

  Broadcast active client's model $x_{i_t}^t$ to all its neighbors $\{k \mid w_{i_t,k}^t \neq 0, k \neq i_t\}$

  Sample a batch of active client i_t's local data $\xi_{i_t}^t := \{\xi_{i_t,m}^t\}_{m=1}^M$ from distribution $\mathcal{D}_{i_t}$

  Compute the gradient update: $g(x_{i_t}^t, \xi_{i_t}^t) \leftarrow \frac{1}{M}\sum_{m=1}^M \nabla \ell(x_{i_t}^t, \xi_{i_t,m}^t)$

  if current step falls in the predefined communication set, i.e., $c_{i_t} \in \mathcal{C}_s$ **then**

    Fetch and store the latest models $\{x_k^t\}$ from i_t's neighbors $\{k \mid w_{i_t,k}^t \neq 0, k \neq i_t\}$

    Model average for the active client: $x_{i_t}^{t+1/2} \leftarrow \sum_k w_{i_t,k}^t x_k^t + w_{i_t,i_t}^t x_{i_t}^t$

  else

    Active client model remains the same: $x_{i_t}^{t+1/2} \leftarrow x_{i_t}^t$

  end if

  Model update for the active client: $x_{i_t}^{t+1} \leftarrow x_{i_t}^{t+1/2} - \gamma g(x_{i_t}^t; \xi_{i_t}^t)$

  Update other clients: $x_j^{t+1} \leftarrow x_j^t, \quad \forall\, j \neq i_t$

  Update the active client's counter $c_{i_t} \leftarrow c_{i_t} + 1$

end for

return Consensus model $\frac{1}{n}\sum_{i=1}^n x_i^T$

tralized FL algorithms, SWIFT does not require simultaneous averaging between two clients or a neighborhood of clients. Instead, each client fetches the latest models its neighbors have sent it and performs averaging with those available (Lines 11-12 of Algorithm 9). There is no pause in local training waiting for a neighboring client to finish computations or average with, making SWIFT wait-free.

Update Rule. SWIFT runs in parallel with all clients performing local gradient updates and model communication simultaneously. Collectively, the update rule can be written in matrix form as

$$X^{t+1} = X^t W_{i_t}^t - \gamma G(x_{i_t}^t, \xi_{i_t}^t), \tag{5.4}$$

where γ denotes the step size parameter and the matrix $G(x_{i_t}^t, \xi_{i_t}^t) \in \mathbb{R}^{d \times n}$ is the zero-padded gradient of the active model $x_{i_t}^t$. The entries of $G(x_{i_t}^t, \xi_{i_t}^t)$ are zero except for the i_t-th column, which contains the active gradient $g(x_{i_t}^t, \xi_{i_t}^t)$. Next, we describe the client-communication matrix $W_{i_t}^t$.

Client-Communication Matrix. The backbone of decentralized FL algorithms is the client-communication matrix W (also known as the weighting matrix). To remove all forms of synchronization and to become wait-free, SWIFT relies upon a novel client-communication matrix $W_{i_t}^t$ that is neither symmetric nor doubly-stochastic, unlike other algorithms in FL [6, 178, 181, 196]. The result of a non-symmetric and non-doubly stochastic client-communication matrix, is that averaging occurs for a single active client i_t and not over a pair or neighborhood of clients. This curbs superfluous communication time.

Within SWIFT, a dynamic client-communication matrix is implemented to allow for periodic averaging. We will now define the *active client-communication matrix* $W_{i_t}^t$ in SWIFT, where

i_t is the active client which performs the t-th global iteration. $W_{i_t}^t$ can be one of two forms: (1) an identity matrix $W_{i_t}^t = I_n$ if $c_{i_t} \notin \mathcal{C}_s$ or (2) a communication matrix if $c_{i_t} \in \mathcal{C}_s$ with structure,

$$W_{i_t}^t := I_n + (w_{i_t}^t - \boldsymbol{e}_{i_t})\boldsymbol{e}_{i_t}^\intercal,\ w_{i_t}^t := [w_{1,i_t}^t, \ldots, w_{n,i_t}^t]^\intercal \in \mathbb{R}^n,\ \sum_{j=1}^{n} w_{j,i} = 1,\ w_{i,i} \geq 1/n\ \forall i. \quad (5.5)$$

The vector $w_{i_t}^t \in \mathbb{R}^n$ denotes *the active client-communication vector* at iteration t, which contains the communication coefficients between client i_t and all clients (including itself). The client-communication coefficients induce a weighted average of local neighboring models. We note that $w_{i_t}^t$ is often sparse because clients are connected to few other clients only in most decentralized settings.

Novel Client-Communication Weight Selection. While utilizing a non-symmetric and non-doubly-stochastic client-communication matrix decreases communication time, there are technical difficulties when it comes to guaranteeing the convergence. One of the novelties of our work is that we carefully design a client-communication matrix $W_{i_t}^t$ such that it is symmetric and doubly-stochastic *under expectation of all potential active clients* i_t and has diagonal values greater than or equal to $1/n$. Specifically, we can write

$$\mathbb{E}_{i_t}\left[W_{i_t}^t\right] = \sum_{i=1}^{n} p_i \left[I_n + (w_i^t - \boldsymbol{e}_i)\boldsymbol{e}_i^\intercal\right] = I_n + \sum_{i=1}^{n} p_i (w_i^t - \boldsymbol{e}_i)\boldsymbol{e}_i^\intercal =: \bar{W}^t, \quad (5.6)$$

where, we denote $\bar{W}^t$ as the *expected client-communication matrix* with the following form,

$$[\bar{W}^t]_{i,i} = 1 + p_i(w_{i,i}^t - 1), \text{ and } [\bar{W}^t]_{i,j} = p_j w_{i,j}^t, \text{for } i \neq j. \quad (5.7)$$

Note that $\bar{W}^t$ is column stochastic as the entries of any column sum to one. If we ensure that $\bar{W}^t$ is symmetric, then it will become doubly-stochastic. By Equation 5.7, $\bar{W}^t$ becomes symmetric

Algorithm 10 Communication Coefficient Selection (CCS)

Require: Client Influence Score (CIS) $p_i \in \mathbb{R}$, Client Degree $d_i \in \mathbb{R}$, Client Neighbor Set $J_i = \{\forall j : \text{client } j \text{ is a one-hop neighbor of client } i\}, \forall i$

for $i = 1 : n$ in parallel **do**

if CIS are non-uniform **then**

Initialize Client-Communication Vector $w_i = [w_{1,i}, w_{2,i}, \cdots, w_{n,i}] \leftarrow (1/n)\boldsymbol{e}_i$

else

Initialize Client-Communication Vector $w_i = [w_{1,i}, w_{2,i}, \cdots, w_{n,i}] \leftarrow \mathbf{0}$

end if

Exchange CIS and degree with all neighbors

Store

Neighbor CIS Vector $P^J \leftarrow [\{p_j\}_{j \in J_i}] \in \mathbb{R}^{d_i}$

Construct Neighbor Subsets J^{L}, J^{SE}, $J^{\mathsf{E}} \subset J_i$ as subsets of i's neighbors with degree larger than, no larger than and equal to d_i respectively

for $\forall j \in J^{\mathsf{L}}$ **do**

Wait to fetch $w_{j,i}$ from neighbor client j with a degree larger than d_i

end for

Determine the sum of the *total coefficients assigned (TCA)* $s_w^i \leftarrow \sum_{m=1}^n w_{m,i}$

if $|J^{\mathsf{SE}}| > 0$ **then**

Determine the sum of all remaining neighbors' CIS $s_p^i \leftarrow \sum_{j \in J^{\mathsf{SE}}} P_j^J$

if $|J^{\mathsf{E}}| > 0$ **then**

Exchange s_w^i and s_p^i with all neighbors $j \in J^{\mathsf{E}}$, storing all exchanged s_w^j, s_p^j

Store $s_w^* \leftarrow \max\{s_w^i, s_w^j\}$ and $s_p^* \leftarrow \max\{s_p^i, s_p^j\} \ \ \forall j \in J^{\mathsf{E}}$

Set $w_{j,i} \leftarrow \frac{(1-s_w^*)P_j^J}{s_p^*} \ \forall j \in J^{\mathsf{E}}$

Recompute $s_w^i = \sum_{m=1}^n w_{m,i}$ and $s_p^i = \sum_{j \in \{J^{\mathsf{SE}} \cup i\} \setminus J^{\mathsf{E}}} P_j^J$

end if

Set $w_{j,i} \leftarrow w_{j,i} + \frac{(1-s_w^i)P_j^J}{s_p^i} \ \forall j \in \{J^{\mathsf{SE}} \cup i\} \setminus J^{\mathsf{E}}$ for all remaining neighbors

Send $w_{i,j} = \frac{(1-s_w^i)P_i^J}{s_p^i}$ to all waiting neighbors $j \in J^{\mathsf{SE}} \setminus J^{\mathsf{E}}$

else

$w_{i,i} = 1 - s_w^i$

end if

end for

return Client-Communication Vector $w_i \in \mathbb{R}^n, \forall i \in [n]$

if,

$$p_j w_{i,j}^t = p_i w_{j,i}^t \; \forall i, j \in \mathcal{V}. \tag{5.8}$$

To achieve the symmetry of Equation 5.8, SWIFT deploys a novel pre-processing algorithm: the Communication Coefficient Selection (CCS) Algorithm. Given any client-influence vector p_i, CCS determines all client-communication coefficients such that Equations 5.5 and 5.8 hold for every global iteration t. Unlike other algorithms, CCS focuses on the *expected* client-communication matrix, ensuring its symmetry. CCS only needs to run once, before running SWIFT. In the event that the underlying network topology changes, CCS can be run again during the middle of training. Below, we detail how CCS guarantees Equations 5.5 and 5.8 to hold.

The CCS Algorithm, presented in Algorithm 10, is a waterfall method: clients receive coefficients from their larger-degree neighbors. Every client runs CCS concurrently, with the following steps:

(1) Receive coefficients from larger-degree neighbors. If the largest, or tied, skip to **(2)**.

(2) Calculate the total coefficients already assigned s_w as well as the sum of the client influence scores for the unassigned clients s_p.

(3) Assign the leftover coefficients $1 - s_w$ to the remaining unassigned neighbors (and self) in a manner proportional to each unassigned client i's percentage of leftover influence scores p_i/s_p.

(4) If tied with neighbors in degree, ensure assigned coefficients do not sum to larger than one.

(5) Send coefficients to smaller-degree neighbors.

In Algorithm 10, the terms s_w and s_p play a pivotal role in satisfying Equations 5.5 and 5.8 respectively. Equation 5.8 is satisfied by assigning weights amongst client i and its neighbors j

as follows

$$p_j \frac{(1-s_w)p_i}{s_p} = p_i \frac{(1-s_w)p_j}{s_p}. \tag{5.9}$$

Assigning weights proportionally with respect to neighboring client influence scores and total neighbors ensures higher influence clients receive higher weighting during averaging.

5.5 SWIFT Theoretical Analysis

Major Message from Theoretical Analysis. As summarized in Table 5.1, the efficiency and effectiveness of decentralized FL algorithms depend on both the iteration convergence rate and communication-time complexity; their product roughly approximates the total time for convergence. In this section, we will prove that SWIFT improves the SOTA convergence time of decentralized FL as it obtains SOTA iteration convergence rate (Theorem 34) and outperforms SOTA communication-time complexity.

Before presenting our theoretical results, we first detail standard assumptions [195] required for the analysis.

Assumption 4 (L-smooth global and local objective functions)**.** $||\nabla f(x) - \nabla f(y)|| \leq L||x-y||$.

Assumption 5 (Unbiased stochastic gradient)**.** $\mathbb{E}_{\xi \sim \mathcal{D}_i}\left[\nabla \ell(x;\xi)\right] = \nabla f_i(x)$ *for each* $i \in \mathcal{V}$.

Assumption 6 (Bounded inter-client gradient variance)**.** *The variance of the stochastic gradient is bounded for any* x *with client* i *sampled from probability vector* p *and local client data* ξ *sampled from* $\mathcal{D}_i$*. This implies there exist constants* $\sigma, \zeta \geq 0$ *(where* $\zeta = 0$ *in IID settings) such that:* $\mathbb{E}_i||\nabla f(x) - \nabla f_i(x)||^2 \leq \zeta^2 \; \forall i, \forall x$, $\mathbb{E}_{\xi \sim \mathcal{D}_i}||\nabla f_i(x) - \nabla \ell(x;\xi)||^2 \leq \sigma^2, \forall x$.

As mentioned in Section 5.4, the use of a non-symmetric, non-doubly-stochastic matrix

$W_{i_t}^t$ causes issues in analysis. In Section 5.8.2, we discuss properties of stochastic matrices, including symmetric and doubly-stochastic matrices, and formally define ρ_ν, a constant related to the connectivity of the network. Utilizing our symmetric and doubly-stochastic expected client-communication matrix (constructed via Algorithm 10), we reformulate Equation 5.4 by adding and subtracting out $\bar{W}^t$,

$$X^{t+1} = X^t \bar{W}^t + X^t(W_{i_t}^t - \bar{W}^t) - \gamma G(x_{i_t}^t, \xi_{i_t}^t). \tag{5.10}$$

Next, we present our first main result in Lemma 33, establishing a client-communication error bound.

Lemma 33 (Client-Communication Error Bound)**.** *Following Algorithm 10, the product of the difference between the expected and actual client communication matrices is bounded as follows:*

$$\mathbb{E}\sum_{j=0}^{t}\Big\|G(x_{i_j}^j, \xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q - \bar{W}^q)(\frac{\mathbf{1_n}}{n} - \boldsymbol{e}_i)\Big\|^2 = \mathcal{O}\Big(\frac{\sigma^2}{M} + \mathbb{E}\sum_{j=0}^{t}\|\nabla f_{i_j}(x_{i_j}^j)\|^2\Big). \tag{5.11}$$

Remark. One novelty of our work is that we are the first to bound the client-communication error in the asynchronous decentralized FL setting. The upper bound in Lemma 33 is unique to our analysis because other decentralized works do not incorporate wait-free communication [6, 46, 178, 181]. Now, we are ready to present our main theorem, which establishes the convergence rate of SWIFT:

Theorem 34 (Convergence Rate of SWIFT)**.** *Under assumptions 4, 5 and 6 (with Algorithm 10), let $\Delta_f := f(\bar{x}^0) - f(\bar{x}^*)$, step-size γ, total iteration T, and average model $\bar{x}^t$ be defined as*

$$\gamma := \frac{\sqrt{Mn^2\Delta_f}}{\sqrt{TL} + \sqrt{M}} \leq \sqrt{\frac{Mn^2\Delta_f}{TL}}, \quad T \geq 193^2 LM\Delta_f\rho_\nu^2 n^4 p_{max}^2, \quad \bar{x}^t := \frac{1}{n}\sum_{i=1}^{n} x_i^t.$$

Then, for the output of Algorithm 9, it holds that

$$\frac{1}{T}\sum_{t=0}^{T-1}\mathbb{E}||\nabla f(\bar{x}^t)||^2 \leq \frac{2\sqrt{\Delta_f}}{T} + \frac{2\sqrt{L\Delta_f}\left(1+\frac{1921}{20}\rho_\nu\left(\sigma^2+6\zeta^2\sqrt{M}\right)\right)}{\sqrt{TM}}. \tag{5.12}$$

Iteration Convergence Rate Remarks. **(1)** We prove that SWIFT obtains a $\mathcal{O}(1/\sqrt{T})$ iteration convergence rate, matching the optimal rate for SGD [46, 181, 197, 198]. **(2)** Unlike existing asynchronous decentralized SGD algorithms, SWIFT's iteration convergence rate does not depend on the maximal bounded delay. Instead, we bound any delays by taking the *expectation* over the active client. The probability of each client i being the active client is simply its sampling probability p_i. We therefore assume that each client i is *expected* to perform updates at its prescribed sampling probability rate p_i. Clients which are often delayed in practice can be dealt with by lowering their inputted sampling probability. **(3)** SWIFT converges in fewer total iterations T with respect to n total clients compared to other asynchronous methods [181] ($T = \Omega(n^4 p_{max}^2)$ in SWIFT versus $T = \Omega(n^4)$ in AD-PSGD).

Communication-Time Complexity Remarks. **(1)** Due to its asynchronous nature, SWIFT achieves a communication-time complexity that relies only on each client's own communication time per round $\mathscr{C}_i$. This improves upon synchronous decentralized SGD algorithms, which rely upon the communication time per round of the slowest neighboring client $\max_{j\in\mathcal{N}_i}\mathscr{C}_j$. **(2)** Unlike AD-PSGD [181], which also achieves a communication-time complexity reliant on $\mathscr{C}_i$, SWIFT incorporates periodic averaging which further reduces the communication complexity from T rounds of communication to $|\mathcal{C}_s|$. Furthermore, SWIFT allows for entire neighborhood averaging, and not just one-to-one gossip averaging. This increases neighborhood information sharing, improving model robustness and reducing model divergence.

Corollary 35 (Convergence under Uniform Client Influence)**.** *In the common scenario where client influences are uniform,* $p_i = 1/n \ \forall i \implies p_{max} = 1/n$*,* SWIFT *obtains convergence improvements: total iterations* T *with respect to the number of total clients* n *improves to* $T = \Omega(n^2)$ *as compared to* $T = \Omega(n^4)$ *for AD-PSGD under the same conditions.*

5.6 Experiments

Below, we perform image classification experiments for a range of decentralized FL algorithms [3]. We compare the results of SWIFT to the following decentralized baselines:

• The most common synchronous decentralized FL algorithm: D-SGD [46].

• Synchronous decentralized FL communication reduction algorithms: PA-SGD [178] and LD-SGD [6].

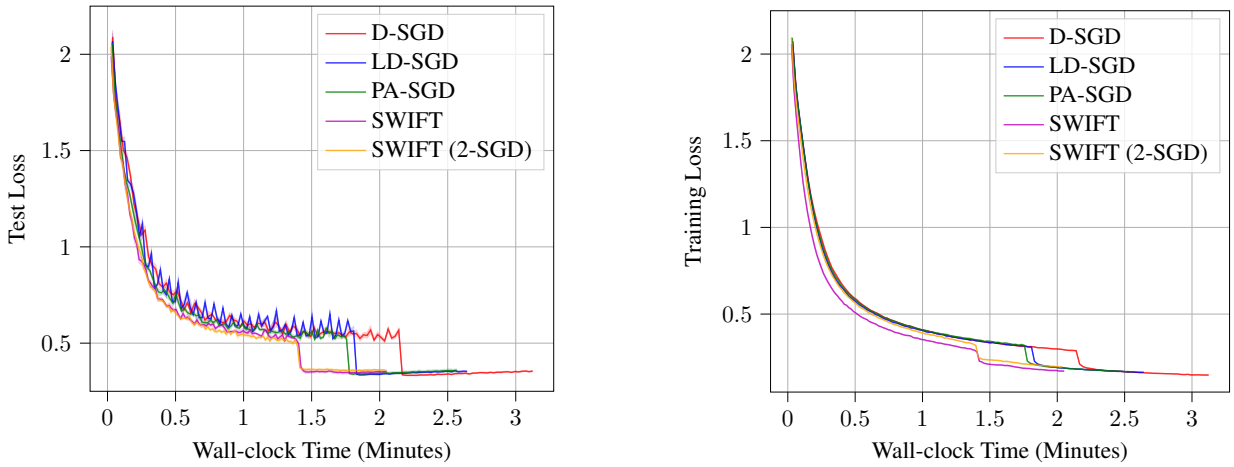
• The most prominent asynchronous decentralized FL algorithm: AD-PSGD [181].

Finer details of the experimental setup are in Section 5.8.1. Throughout our experiments we use two network topologies: standard ring and ring of cliques (ROC). ROC-xC signifies a ring of cliques with x clusters. The ROC topology is more reflective of a realistic network, as networks usually have pockets of connected clients. These topologies are visualized in Figures 5.7 and 5.8 respectively.

5.6.1 Baseline Comparison

To compare the performance of SWIFT to all other algorithms listed above, we reproduce an experiment within [181]. With no working code for AD-PSGD to run on anything but an extreme supercomputing cluster (Section 5.1.2 of [181]), reproducing this experiment allows us

to compare the relative performance of SWIFT to AD-PSGD.



(a) Average test loss.

(b) Average train loss.

Figure 5.2: Baseline performance comparison on CIFAR-10 for 16 client ring.

Table in Figure 5.2 showcases that SWIFT reduces the average epoch time, relative to D-SGD, by 35% ($\mathcal{C}_0$ and $\mathcal{C}_1$). This far outpaces AD-PSGD (as well as the other synchronous algorithms), with AD-PSGD only reducing the average epoch time by 16% relative to D-SGD. Finally, Figure 5.2 displays how much faster SWIFT achieves optimal train and test loss values compared to other decentralized baseline algorithms. SWIFT outperforms all other baseline algorithms even without any slow-down (which we examine in Section 5.6.2), where wait-free algorithms like SWIFT especially shine.

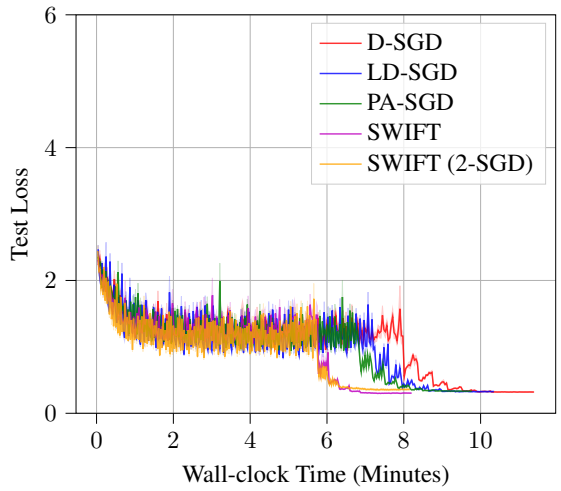
Decentralized FL Algorithms	16 Client Ring			
	Epoch (s)	% Change	Comm. (s)	% Change
SWIFT ($\mathcal{C}_0$)	**1.019**	-34.60	**0.086**	-86.28
D-SGD ($\mathcal{C}_0$)	1.558	—	0.627	—-
AD-PSGD* ($\mathcal{C}_0$)	—	-15.86	—	—
SWIFT ($\mathcal{C}_1$)	**1.016**	-34.79	**0.064**	-89.79
LD-SGD ($\mathcal{C}_1$)	1.320	-15.28	0.428	-31.74
PA-SGD ($\mathcal{C}_1$)	1.281	-17.78	0.358	-42.90

* AD-PSGD results come from Table 4 in [181].

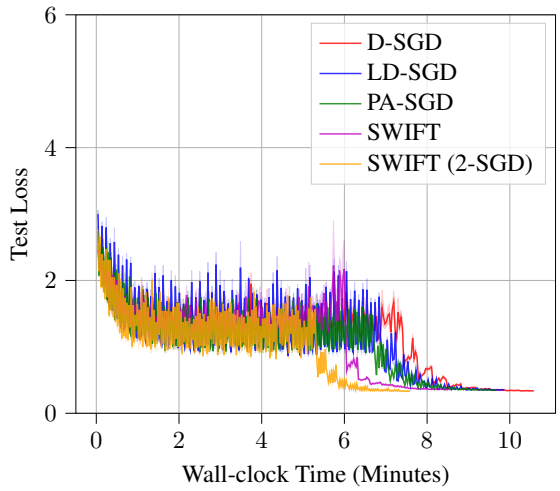
Table 5.2: Baseline average epoch and communication times on CIFAR-10 for 16 client ring.

5.6.2 Varying Heterogeneities

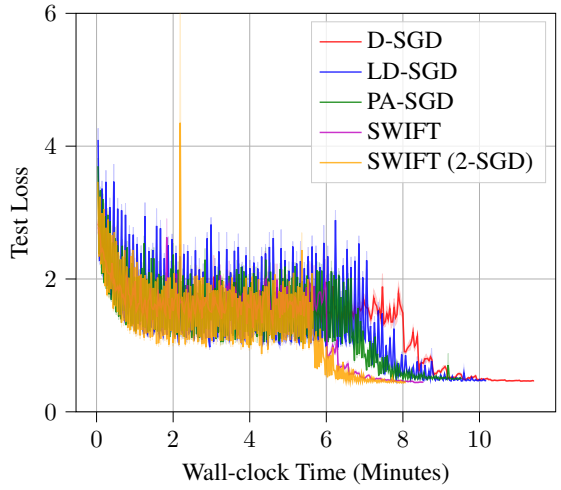
Varying Degrees of Non-IIDness Our second experiment evaluates SWIFT's efficacy at converging to a well-performing optima under varying degrees of non-IIDness. We vary the degree (percentage) of each client's data coming from one label. The remaining percentage of data is randomly sampled (IID) data over all labels. A ResNet-18 model is trained by 10 clients in a 3-cluster ROC network topology. We chose 10 clients to make the label distribution process easier: CIFAR-10 has 10 labels. As expected, when data becomes more non-IID, the test loss becomes higher and the overall accuracy lower (Table 5.3). We do see, however, that SWIFT converges faster, and to a lower average loss, than all other synchronous baselines (Figure 5.3). In fact, SWIFT with $\mathcal{C}_1$ converges *much* quicker than the synchronous algorithms. This is an important result: SWIFT converges both quicker and to a smaller loss than synchronous algorithms in the non-IID setting.



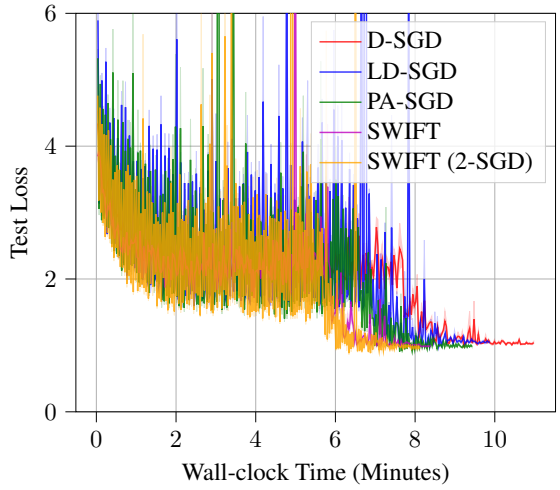
(a) 1/4 degree non-IID data.



(b) 1/2 degree non-IID data.



(c) 7/10 degree non-IID data.



(d) 9/10 degree non-IID data.

Figure 5.3: Average test loss for varying degrees of non-IIDness on CIFAR-10, 10 client ROC-3C.

Varying Heterogeneity of Clients In this experiment, we investigate the performance of SWIFT under varying heterogeneity, or speed, of our clients (causing different delays). This is done with 16 clients in a ring topology. We add an artificial slowdown, suspending execution of one of the clients such that it takes a certain amount of time longer (slowdown) to perform the computational portions of the training process. We perform tests in the case where a client is two times (2x) and four times (4x) as slow as usual. We then compare how the decentralized algorithms fare under these circumstances compared to the normal setting with no added slowdown.

In Table 5.4, the average epoch, communication, and total time is displayed. Average total time includes computations, communication, and any added slowdown (wait time). SWIFT avoids large average total times as the slowdown grows larger. The wait-free structure of SWIFT allows all non-slowed clients to finish their work at their own speed. All other algorithms require clients to wait for the slowest client to finish a mini-batch before proceeding. At large slowdowns

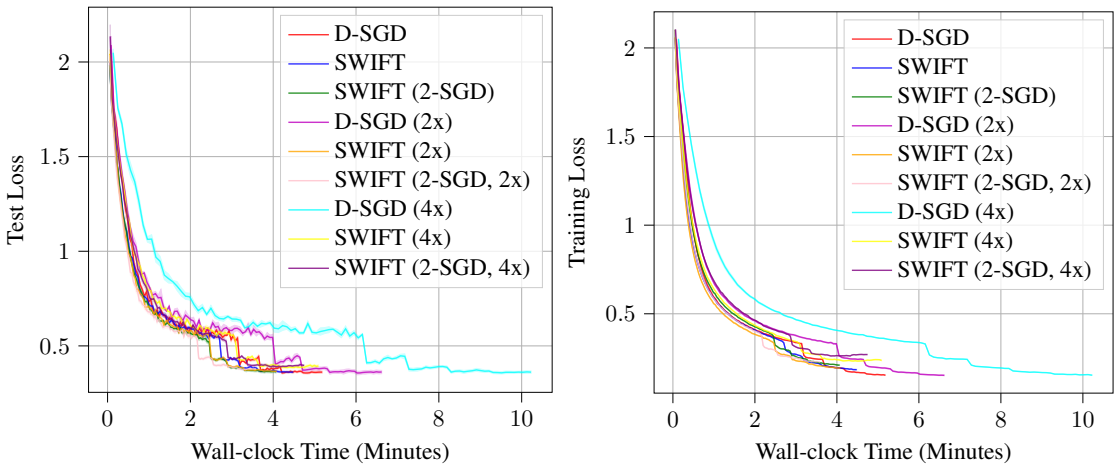
Decentralized FL Algorithms	10 Client Ring of Cliques - 3 Cluster Topology	
	Epoch Time (s)	Communication Time (s)
SWIFT ($\mathcal{C}_0$)	**1.709**	**0.197**
D-SGD ($\mathcal{C}_0$)	2.116	0.705
SWIFT ($\mathcal{C}_1$)	**1.517**	**0.110**
LD-SGD ($\mathcal{C}_1$)	1.973	0.575
PA-SGD ($\mathcal{C}_1$)	1.929	0.421

Table 5.3: Average epoch and communication times for non-IID setting on CIFAR-10.

Decentralized FL Algorithms	No Slowdown			2x Slowdown			4x Slowdown		
	Total (s)	Epoch (s)	Comm. (s)	Total (s)	Epoch (s)	Comm. (s)	Total (s)	Epoch (s)	Comm. (s)
SWIFT ($\mathcal{C}_0$)	**2.69**	**1.033**	**0.087**	**2.451**	**0.964**	**0.064**	**3.054**	**1.117**	**0.091**
D-SGD ($\mathcal{C}_0$)	3.110	1.45	0.564	3.972	1.571	0.616	6.137	1.666	0.651
SWIFT ($\mathcal{C}_1$)	**2.44**	**0.996**	**0.061**	**2.152**	**0.928**	**0.040**	**2.847**	**1.074**	**0.065**
LD-SGD ($\mathcal{C}_1$)	3.323	1.353	0.413	3.762	1.389	0.428	5.917	1.412	0.448
PA-SGD ($\mathcal{C}_1$)	3.183	1.270	0.331	3.557	1.262	0.309	5.743	1.270	0.318

Table 5.4: Average epoch and communication times on CIFAR-10 for 16 client ring with slowdown.

(4x), the average total time for SWIFT is nearly *half* of those for synchronous algorithms. Thus, SWIFT is very effective at reducing the run-time when clients are slow within the network.



(a) Test loss slowdown.

(b) Train loss slowdown.

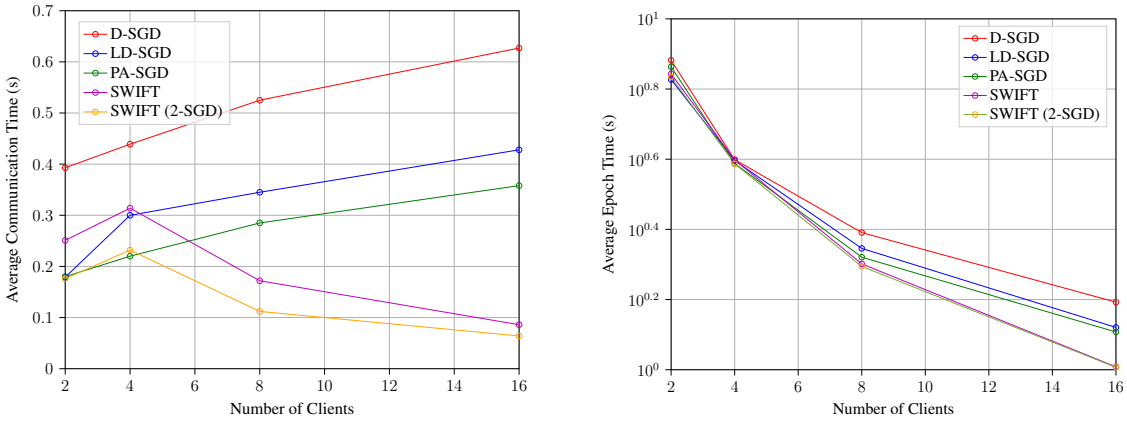
Figure 5.4: SWIFT vs. D-SGD for CIFAR-10 in 16 client ring with varying slowdown.

Figure 5.4 shows how SWIFT is able to converge faster to an equivalent, or smaller, test loss than D-SGD for all slowdowns. In the case of large slowdowns (4x), SWIFT significantly outperforms D-SGD, finishing in better than half the run-time. We do not include the other baseline algorithms to avoid overcrowding of the plotting space. However, SWIFT also performs much better than PA-SGD and LD-SGD as shown in Table 5.4. These results show that the wait-free structure of SWIFT allows it to be efficient under client slowdown.

5.6.3 Varying Numbers of Clients & Network Topologies

Varying Numbers of Clients In our fourth experiment, we determine how SWIFT performs versus other baseline algorithms as we vary the number of clients. In Table 5.5, the time per epoch for SWIFT drops by nearly the optimal factor of 2 as the number of clients is doubled. For all algorithms, there is a bit of parallel overhead when the number of clients is small, however

this becomes minimal as the number of clients grow to be large (greater than 4 clients). In comparison to the synchronous algorithms, SWIFT actually *decreases* its communication time as the number of clients increases. This allows the parallel performance to be quite efficient, as shown in Figure 5.5.



(a) Communication time.

(b) Epoch time.

Figure 5.5: Average communication and epoch times for increasing numbers of clients.

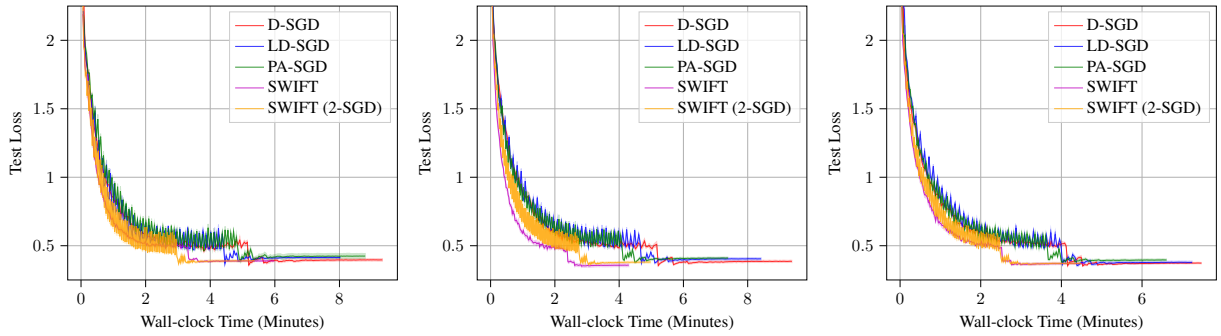
Decentralized FL Algorithms	16 Client Ring		8 Client Ring		4 Client Ring		2 Client Ring	
	Epoch (s)	Comm. (s)	Epoch (s)	Comm. (s)	Epoch (s)	Comm. (s)	Epoch (s)	Comm. (s)
SWIFT ($\mathcal{C}_0$)	**1.019**	**0.086**	**2.003**	**0.172**	**3.964**	**0.314**	**6.971**	**0.251**
D-SGD ($\mathcal{C}_0$)	1.558	0.627	2.459	0.525	3.970	0.439	7.62	0.393
SWIFT ($\mathcal{C}_1$)	**1.016**	**0.064**	**1.970**	**0.112**	**3.862**	0.232	6.83	**0.176**
LD-SGD ($\mathcal{C}_1$)	1.320	0.428	2.217	0.345	3.946	0.300	**6.712**	0.179
PA-SGD ($\mathcal{C}_1$)	1.281	0.358	2.093	0.285	3.871	**0.220**	7.303	0.180

Table 5.5: Average epoch and communication times on CIFAR-10 with varying clients in ring topology.

Varying Topologies Our fifth experiment analyzes the effectiveness of SWIFT versus other baseline decentralized algorithms under different, and more realistic, network topologies. In this experiment setting, we train 16 clients on varying network topologies (Table 5.6 and Figure 5.6).

Decentralized FL	16 Client ROC-2C		16 Client ROC-4C		16 Client Ring	
Algorithms	Epoch (s)	Comm. (s)	Epoch (s)	Comm. (s)	Epoch (s)	Comm. (s)
SWIFT ($\mathcal{C}_0$)	**1.793**	**0.416**	**1.291**	**0.124**	**1.367**	**0.121**
D-SGD ($\mathcal{C}_0$)	2.799	1.479	2.813	1.464	2.241	0.962
SWIFT ($\mathcal{C}_1$)	**1.611**	**0.295**	**1.494**	**0.174**	**1.348**	**0.085**
LD-SGD ($\mathcal{C}_1$)	2.408	0.987	2.525	1.105	2.172	0.517
PA-SGD ($\mathcal{C}_1$)	2.639	0.765	2.216	0.708	1.982	0.500

Table 5.6: Average epoch and communication times on CIFAR-10 for varying network topologies.



(a) ROC-2C ($n = 16$). (b) ROC-4C ($n = 16$). (c) Ring ($n = 16$).

Figure 5.6: Average test loss for varying network topologies on CIFAR-10.

5.7 Conclusion

SWIFT delivers on the promises of decentralized FL: a low communication and run-time algorithm (**scalable**) which attains SOTA loss (**high-performing**). As a wait-free algorithm, SWIFT is well-suited to rapidly solve large-scale distributed optimization problems. Empirically, SWIFT reduces communication time by almost 90% compared to baseline decentralized FL algorithms. In future work, we aim to add protocols for selecting optimal client sampling probabilities. We would like to show how varying these values can: (i) boost convergence both theoretically and empirically, and (ii) improve robustness under local client data distribution shift.

5.8 Supplemental Material

5.8.1 Additional Experimental Details and Setup

5.8.1.1 Computational Specifications

We train our consensus model on a network of nodes. Each node has an NVIDIA GeForce RTX 2080 Ti GPU. All algorithms are built in Python and communicate via Open MPI, using MPI4Py (Python bindings for MPI). The training is also done in Python, leveraging Pytorch.

5.8.1.2 Data Partitioning

For all experiments, the training set is evenly partitioned amongst the number of clients training the consensus model. While the size of each client's partition is equal, we perform testing with data that is both (1) independent and identically distributed (iid) among all clients and (2) sorted by class and is thus non-iid. For the iid setting, each client is assigned data uniformly at random over all classes. In the non-iid setting, each client is assigned a subset of classes from which it will receive data exclusively. The c classes are assigned in the following steps:

(1) The class subset size n_c, for all clients, is determined as the ceiling of the number of classes per client $n_c = \lceil \frac{c}{n} \rceil$. Each class k within the class subset will take up $1/n_c$ of the client's total data partition if possible.

(2) The classes within each client's class subset are assigned cyclically, starting with the first client. The first client selects the first n_c classes, the second client selects the next n_c classes, and so on. Classes can, in some cases, be assigned to multiple clients. If the final class has been assigned, and more clients have yet to be assigned any classes, classes can be re-assigned starting

at the first class.

(3) Each client is assigned data from the classes in their class subset cyclically ($1/n_c$ of its partition for each class), starting with the first client. If no more data is available from a specific class, the required data to fill its fraction of the partition is replaced by data from the next class.

Since we follow this data partitioning process within our experiments, each client is assigned equal partitions of data. Therefore, following the works [6, 180, 181, 194], we set the client influence scores to be uniform for all clients $p_i = 1/n \; \forall i \in \mathcal{V}$.

5.8.1.3 Experimental Setup

Below we provide information into the hyperparameters we select for our experiments in Section 5.6.

Experiment Type	Model	Epochs E	γ	γ Decay (Rate, E, Freq.)	M	Weight Decay	Momentum
Baseline	ResNet-18	200	0.1	(1/10, 81 & 122, Single)	32	10^{-4}	0.9
Vary non-IIDness	ResNet-18	300	0.8	(1/2, 200, 10)	32	10^{-4}	0.9
Vary Heterogeneity	ResNet-18	100	0.1	(1/2, 50, 10)	32	10^{-4}	0.9
Vary # of Clients	ResNet-18	200	0.1	(1/10, 81 & 122, Single)	32	10^{-4}	0.9
Vary Topology	ResNet-50	200	0.1	(1/2, 100, 10)	64	10^{-4}	0.9

Table 5.7: Hyperparameters for all experiments.

In Table 5.7, one can see that the step-size decay column is split into the following sections: rate, E, and frequency. The rate is the decay rate for the step-size. For example, in the Baseline row, the step-size decays by 1/10. The term E is the epoch at which decay begins during training. For example, in the Baseline row, the step-size decays at $E = 81$ and 122. Frequency simply is how often the step-size decays. For example, in the Vary Topology row, the step-size decays every 10 epochs.

5.8.1.4 Network Topologies

Ring Topology. Please refer to Figure 5.7.

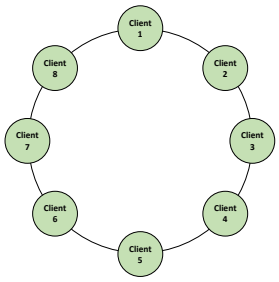


Figure 5.7: An 8 Client Ring.

Ring of Cliques Topology. Please refer to Figure 5.8.

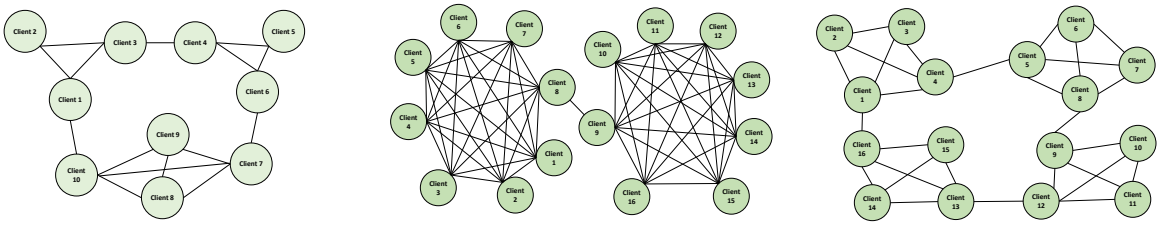


Figure 5.8: **(Left)** 10 Client, 3-Cluster. **(Middle)** 16 Client, 2-Cluster. **(Right)** 16 Client, 4-Cluster.

5.8.2 Additional Properties and Algorithm Details

Stochastic Matrices. Within our work, we use a non-symmetric, non-doubly-stochastic matrix $W_{i_t}^t$ for client averaging. Utilizing $W_{i_t}^t$ comes with some analysis issues (it is non-symmetric), however it provides the wait-free nature of SWIFT. Interestingly, $W_{i_t}^t$ does have some unique properties: it is column-stochastic. Lemma 37 proves that the product of stochastic matrices converges exponentially to a stochastic vector with common ratio $\nu \in [0, 1)$.

Symmetric and Doubly-Stochastcic Matrices. As mentioned in Section 4.5, we utilize

Algorithm 10 to select client weights such that we have a symmetric and doubly-stochastic communication matrix $\bar{W}^t$ under expectation. By Lemma 36, there exists a scalar $\rho \in [0, 1)$ such that $\max\{|\lambda_2\big((\bar{W}^t)^\intercal \bar{W}^t\big)|, |\lambda_n\big((\bar{W}^t)^\intercal \bar{W}^t\big)|\} \leq \rho, \ \forall t$. This parameter ρ reflects the connectivity of the underlying graph topology. The value of ρ is inversely proportionate to how fast information spreads in the client network. A small value of ρ results in information spreading faster ($\rho = 0$ in centralized settings).

Within our analysis, we denote the parameter ρ_ν as a combination of ρ and ν:

$$\rho_\nu := \frac{n-1}{n}\left(\frac{7}{2(1-\rho)} + \frac{\sqrt{\rho}}{(1-\sqrt{\rho})^2} + \frac{384}{(1-\nu^2)}\right) \tag{5.13}$$

Optimal Step-Size Under Uniform Client Influence. The defined step-size γ and total iterations T for SWIFT is

$$\gamma := \frac{\sqrt{Mn^2\Delta_f}}{\sqrt{TL} + \sqrt{M}} \leq \sqrt{\frac{Mn^2\Delta_f}{TL}}, \quad T \geq 193^2 LM\Delta_f \rho_\nu^2 n^4 p_{max}^2.$$

Therefore, γ can be rewritten as

$$\gamma \leq \sqrt{\frac{Mn^2\Delta_f}{(193^2 LM\Delta_f \rho_\nu^2 n^4 p_{max}^2)L}} = \sqrt{\frac{1}{193^2 L^2 \rho_\nu^2 n^2 p_{max}^2}}.$$

When the client influence scores are uniform (i.e, $p_i = 1/n \ \forall i \in \mathcal{V}$), one can see that our step-size becomes

$$\gamma = \mathcal{O}\left(\frac{1}{L}\right).$$

This mirrors the optimal step-size in analysis of gradient descent convergence to first-order stationary points $\mathcal{O}(1/L)$ [199].

Communication Coefficient Selection (CCS) Initialization. We include different client-communication vector initializations if the client influence scores are uniform versus non-uniform.

The reason for this is to ensure that the self-weight for each client i, $w_{i,i}$, has a value greater than $1/n$. This naturally occurs when the CIS are uniform, however is not so when they are non-uniform.

5.8.3 Review of Existing Inter-Client Communication in Decentralized FL

Decentralized SGD (D-SGD) [46] One of the foundational decentralized Federated Learning algorithms is Decentralized SGD. In order to minimize Equation 5.1, D-SGD orchestrates a local gradient step for all clients before performing synchronous neighborhood averaging. The D-SGD process for a single client i is defined as:

$$x_i^{t+1} = \sum_{j=1}^{n} W_{ij}\left[x_j^t - g(x_j^t, \xi_j^t)\right]. \tag{5.14}$$

The term $g(x_j^t, \xi_j^t)$ denotes the stochastic gradient of x_j^t with mini-batch data ξ_j^t sampled from the local data distribution of client j. The matrix W is a weighting matrix, where W_{ij} is the amount of x_i^{t+1} which will be made up of client j's local model after one local gradient step (e.g. if $W_{ij} = 1/2$, then half of x_i^{t+1} will have been composed of client j's model after its local gradient step). The weighting matrix only has a zero value $W_{ij} = 0$ if clients i and j are not connected (they are not within the same neighborhood). The values of W_{ij} are generally selected ahead of time by a central host, with the usual weighting scheme being uniform. In D-SGD, model communication occurs only after all local gradient updates are finished. These gradient updates are computed in parallel.

Periodic Averaging SGD (PA-SGD) [178] The Periodic Averaging SGD algorithm is an extension of D-SGD. In order to save communication costs when the number of clients grows to be large, PA-SGD performs model averaging after an additional I_1 local gradient steps. Thus,

the communication set for PA-SGD is defined as:

$$\mathcal{C}_{I_1} = \{t \in \mathbb{N} | \; t \mod (I_1 + 1) = 0\}.$$

The special case of $I_1 = 0$ reduces to D-SGD. The PA-SGD process for a single client i is defined as:

$$x_i^{t+1} = \begin{cases} \sum_{j=1}^{w} W_{ij} \left[x_j^t - g(x_j^t, \xi_j^t) \right], & t \in \mathcal{C}_{I_1} \\ x_i^t - g(x_i^t, \xi_i^t), & \text{otherwise.} \end{cases} \tag{5.15}$$

Compared with D-SGD, PA-SGD still suffers from the inefficiency of having to wait for the slowest client for each update. However, PA-SGD saves communication costs by reducing the frequency of communication.

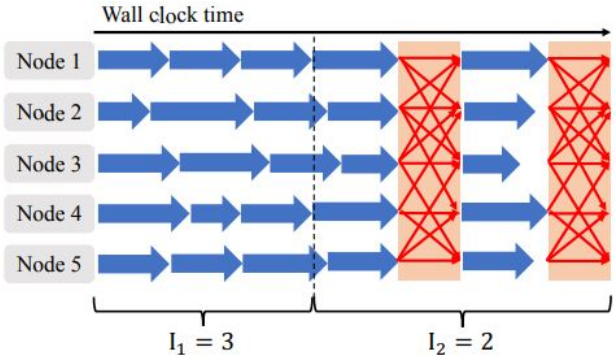


Figure 5.9: I_1, I_2 Depiction (from [6]).

Local Decentralized SGD (LD-SGD) [6] Continuing to generalize the foundational decentralized Federated Learning algorithms is Local Decentralized SGD. LD-SGD generalizes PA-SGD by allowing multiple chunks of singular D-SGD updates, as described in Equation 5.14, in between the increased local gradient steps seen in PA-SGD. The number of D-SGD chunks is dictated by a new parameter I_2.

In this case, the communication set for LD-SGD is defined as

$$\mathcal{C}_{I_1,I_2} = \begin{cases} \bigcup \sum_{i=I_1}^{I_1+I_2} \{t \in \mathbb{N} |\ t \mod (i+1) = 0\} & \text{if } I_1 > 0, \\ \{t \in \mathbb{N}\} & \text{if } I_1 = 0. \end{cases}$$

For example, in the case $I_1 = 3, I_2 = 2$, LD-SGD will take three local gradient steps and then perform two D-SGD updates (which consists of a local gradient step and then averaging) as shown in Figure 5.9. The special case of $I_2 = 1$ reduces to PA-SGD. The LD-SGD process for a single client i is defined as:

$$x_i^{t+1} = \begin{cases} \text{Perform } \sum_{j=1}^{w} W_{ij}\left[x_j^t - g(x_j^t, \xi_j^t)\right], & t \in \mathcal{C}_{I_1,I_2} \\ x_i^t - g(x_i^t, \xi_i^t), & \text{otherwise} \end{cases} \tag{5.16}$$

5.8.4 Proof of the Main Theorem

Before beginning, we quickly define the expected gradient $\mathbb{E}_{i_t}\left[G(x_{i_t}^t, \xi_{*,t}^{i_t})\right]$ as

$$\bar{G}(X^t, \xi^t) := \mathbb{E}_{i_t}\left[G(x_{i_t}^t, \xi_{i_t}^t)\right] = \sum_{i=1}^{n} p_i G(x_{i_t}^t, \xi_{i_t}^t). \tag{5.17}$$

Proof of Theorem 1. In this theorem, we characterize the convergence of the average of *all* local models. Using the Gradient Lipschitz assumption with Equation 5.10 yields

$$\begin{aligned} f\Big(\frac{X^{t+1}\mathbf{1_n}}{n}\Big) \leq & f\Big(\frac{X^t\mathbf{1_n}}{n}\Big) + \Big\langle \nabla f\Big(\frac{X^t\mathbf{1_n}}{n}\Big), \frac{X^t(W_{i_t}^t - \bar{W}^t)\mathbf{1_n}}{n} - \gamma\frac{G(x_{i_t}^t, \xi_{*,t}^{i_t})\mathbf{1_n}}{n}\Big\rangle \\ & + \frac{L}{2}||\frac{X^t(W_{i_t}^t - \bar{W}^t)\mathbf{1_n}}{n} - \gamma\frac{G(x_{i_t}^t, \xi_{*,t}^{i_t})\mathbf{1_n}}{n}||^2. \end{aligned} \tag{5.18}$$

We first denote the average over all local models as $\bar{x}^t := \frac{X^t\mathbf{1_n}}{n}$. Taking the expectation with

respect to the updating client i_t yields

$$
\begin{aligned}
f(\bar{x}^{t+1}) \leq & f(\bar{x}^t) + \left\langle \nabla f(\bar{x}^t), X^t(\bar{W}^t - \bar{W}^t)\frac{\mathbf{1_n}}{n} - \gamma \bar{G}(X^t, \xi_{*,t})\frac{\mathbf{1_n}}{n} \right\rangle \\
& + \frac{L}{2}\mathbb{E}_{i_t}||X^t(W_{i_t}^t - \bar{W}^t)\frac{\mathbf{1_n}}{n} - \gamma G(x_{i_t}^t, \xi_{*,t}^{i_t})\frac{\mathbf{1_n}}{n}||^2 \qquad (5.19) \\
= & f(\bar{x}^t) + \left\langle \nabla f(\bar{x}^t), -\gamma \bar{G}(X^t, \xi_{*,t})\frac{\mathbf{1_n}}{n} \right\rangle \\
& + \frac{L}{2}\mathbb{E}_{i_t}||X^t(W_{i_t}^t - \bar{W}^t)\frac{\mathbf{1_n}}{n} - \gamma G(x_{i_t}^t, \xi_{*,t}^{i_t})\frac{\mathbf{1_n}}{n}||^2 \qquad (5.20) \\
= & f(\bar{x}^t) + \left\langle \nabla f(\bar{x}^t), -\frac{\gamma}{Mn}\sum_{i=1}^{n}\sum_{m=1}^{M} p_i \nabla \ell(x_i^t, \xi_{m,t}^i) \right\rangle \\
& + \frac{L}{2}\mathbb{E}_{i_t}||X^t(W_{i_t}^t - \bar{W}^t)\frac{\mathbf{1_n}}{n} - \gamma G(x_{i_t}^t, \xi_{*,t}^{i_t})\frac{\mathbf{1_n}}{n}||^2 \qquad (5.21) \\
= & f(\bar{x}^t) - \gamma\left\langle \nabla f(\bar{x}^t), \frac{1}{Mn}\sum_{i=1}^{n}\sum_{m=1}^{M} p_i \nabla \ell(x_i^t, \xi_{m,t}^i) \right\rangle \\
& + \frac{L}{2}\mathbb{E}_{i_t}||X^t(W_{i_t}^t - \bar{W}^t)\frac{\mathbf{1_n}}{n} - \gamma G(x_{i_t}^t, \xi_{*,t}^{i_t})\frac{\mathbf{1_n}}{n}||^2. \qquad (5.22)
\end{aligned}
$$

Taking the expectation over all local data $\mathbb{E}_{\xi\sim\mathcal{D}_i}$ yields

$$
\begin{aligned}
f(\bar{x}^{t+1}) - f(\bar{x}^t) \leq & -\frac{\gamma}{n}\left\langle \nabla f(\bar{x}^t), \sum_{i=1}^{n} p_i \nabla f_i(x_i^t) \right\rangle \\
& + \frac{L}{2}\mathbb{E}_{\xi\sim\mathcal{D}_i, i_t}||X^t(W_{i_t}^t - \bar{W}^t)\frac{\mathbf{1_n}}{n} - \gamma G(x_{i_t}^t, \xi_{*,t}^{i_t})\frac{\mathbf{1_n}}{n}||^2. \qquad (5.23)
\end{aligned}
$$

By properties of the inner product

$$
\begin{aligned}
f(\bar{x}^{t+1}) - f(\bar{x}^t) \leq & -\frac{\gamma}{2n}\left(||\nabla f(\bar{x}^t)||^2 + ||\sum_{i=1}^{n} p_i \nabla f_i(x_i^t)||^2 - ||\nabla f(\bar{x}^t) - \sum_{i=1}^{n} p_i \nabla f_i(x_i^t)||^2 \right) \\
& + \frac{L}{2}\underbrace{\mathbb{E}_{\xi\sim\mathcal{D}_i, i_t}||X^t(W_{i_t}^t - \bar{W}^t)\frac{\mathbf{1_n}}{n} - \gamma G(x_{i_t}^t, \xi_{*,t}^{i_t})\frac{\mathbf{1_n}}{n}||^2}_{:=A}. \qquad (5.24)
\end{aligned}
$$

Bounding Term A: Given the update equation, term A can be transformed into

$$
\mathbb{E}_{\xi\sim\mathcal{D}_i, i_t}||\left(X^t(W_{i_t}^t - \bar{W}^t) - \gamma G(x_{i_t}^t, \xi_{*,t}^{i_t}) \right)\frac{\mathbf{1_n}}{n}||^2 = \mathbb{E}_{\xi\sim\mathcal{D}_i, i_t}||\left(X^{t+1} - X^t\bar{W}^t \right)\frac{\mathbf{1_n}}{n}||^2. \quad (5.25)
$$

Due to the symmetric and doubly stochastic property of $\bar{W}^t$ this reduces to

$$\mathbb{E}_{\xi\sim\mathcal{D}_i,i_t}||\bar{x}^{t+1}-\bar{x}^t||^2 = \mathbb{E}_{\xi\sim\mathcal{D}_i,i_t}||\frac{1}{n}\sum_{i=1}^{n}\left(x_i^{t+1}-x_i^t\right)||^2 = \mathbb{E}_{\xi\sim\mathcal{D}_i,i_t}||\frac{1}{n}\left(x_{i_t}^{t+1}-x_{i_t}^t\right)||^2 \tag{5.26}$$

$$= \frac{1}{n^2}\mathbb{E}_{\xi\sim\mathcal{D}_i,i_t}||x_{i_t}^{t+1}-x_{i_t}^t||^2 \tag{5.27}$$

$$\leq \frac{3}{n^2}\mathbb{E}_{\xi\sim\mathcal{D}_i,i_t}\Big(||x_{i_t}^{t+1}-\bar{x}^{t+1}||^2 + ||\bar{x}^t - x_{i_t}^t||^2 + ||\bar{x}^{t+1}-\bar{x}^t||^2\Big). \tag{5.28}$$

Combining like terms yields

$$(1-\frac{3}{n^2})\mathbb{E}_{\xi\sim\mathcal{D}_i,i_t}||\bar{x}^{t+1}-\bar{x}^t||^2 \leq \frac{3}{n^2}\mathbb{E}_{\xi\sim\mathcal{D}_i,i_t}\Big(||x_{i_t}^{t+1}-\bar{x}^{t+1}||^2 + ||\bar{x}^t - x_{i_t}^t||^2\Big). \tag{5.29}$$

Since $n \geq 2$ (we assume at least 2 devices are running the algorithm) we find the following result

$$\begin{aligned}\mathbb{E}_{\xi\sim\mathcal{D}_i,i_t}||\bar{x}^{t+1}-\bar{x}^t||^2 &\leq \frac{3}{(n^2-3)}\mathbb{E}_{\xi\sim\mathcal{D}_i,i_t}\Big(||x_{i_t}^{t+1}-\bar{x}^{t+1}||^2 + ||\bar{x}^t - x_{i_t}^t||^2\Big) \\ &= \frac{3}{(n^2-3)}\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{i=1}^{n}p_i\Big(||\bar{x}^{t+1}-x_i^{t+1}||^2 + ||\bar{x}^t - x_i^t||^2\Big)\end{aligned} \tag{5.30}$$

Thus, we have bounded Term A. Substituting this back into Equation 5.24 results in

$$\begin{aligned}&\leq -\frac{\gamma}{2n}\Big(||\nabla f(\bar{x}^t)||^2 + ||\sum_{i=1}^{n}p_i\nabla f_i(x_i^t)||^2 - ||\nabla f(\bar{x}^t) - \sum_{i=1}^{n}p_i\nabla f_i(x_i^t)||^2\Big)] \\ &+ \frac{3L}{2(n^2-3)}\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{i=1}^{n}p_i\Big(||\bar{x}^{t+1}-x_i^{t+1}||^2 + ||\bar{x}^t - x_i^t||^2\Big).\end{aligned} \tag{5.31}$$

Taking the sum from $t=0$ to $t=T-1$ yields

$$\begin{aligned}f(\bar{x}^T) - f(\bar{x}^0) \leq -\frac{\gamma}{2n}\Big(&\sum_{t=0}^{T-1}||\nabla f(\bar{x}^t)||^2 - \sum_{t=0}^{T-1}||\nabla f(\bar{x}^t) - \sum_{i=1}^{n}p_i\nabla f_i(x_i^t)||^2 \\ &+ \sum_{t=0}^{T-1}||\sum_{i=1}^{n}p_i\nabla f_i(x_i^t)||^2\Big) \\ &+ \frac{3L}{2(n^2-3)}\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{t=0}^{T-1}\sum_{i=1}^{n}p_i\Big(||\bar{x}^{t+1}-x_i^{t+1}||^2 + ||\bar{x}^t - x_i^t||^2\Big).\end{aligned} \tag{5.32}$$

Using the Lipschitz Gradient assumption, the following term is bounded as

$$\sum_{t=0}^{T-1} ||\nabla f(\bar{x}^t) - \sum_{i=1}^{n} p_i \nabla f_i(x_i^t)||^2 = \sum_{t=0}^{T-1} || \sum_{i=1}^{n} p_i \nabla f_i(\bar{x}^t) - \sum_{i=1}^{n} p_i \nabla f_i(x_i^t)||^2 \tag{5.33}$$

$$\leq \sum_{t=0}^{T-1} \sum_{i=1}^{n} p_i^2 ||\nabla f_i(\bar{x}^t) - \nabla f_i(x_i^t)||^2 \tag{5.34}$$

$$\leq L^2 \sum_{t=0}^{T-1} \sum_{i=1}^{n} p_i^2 ||\bar{x}^t - x_i^t||^2 \tag{5.35}$$

$$\leq L^2 p_{max} \sum_{t=0}^{T-1} \sum_{i=1}^{n} p_i ||\bar{x}^t - x_i^t||^2 \tag{5.36}$$

Placing this back into Equation 5.32, and rearranging, yields

$$\begin{aligned} f(\bar{x}^T) - f(\bar{x}^0) \leq & - \frac{\gamma}{2n} \sum_{t=0}^{T-1} ||\nabla f(\bar{x}^t)||^2 - \frac{\gamma}{2n} \sum_{t=0}^{T-1} || \sum_{i=1}^{n} p_i \nabla f_i(x_i^t)||^2 \\ & + \frac{3L}{2(n^2-3)} \mathbb{E}_{\xi \sim \mathcal{D}_i} \sum_{t=0}^{T-1} \sum_{i=1}^{n} p_i \left(||\bar{x}^{t+1} - x_i^{t+1}||^2 + ||\bar{x}^t - x_i^t||^2 \right) \\ & + \frac{\gamma L^2 p_{max}}{2n} \sum_{t=0}^{T-1} \sum_{i=1}^{n} p_i ||\bar{x}^t - x_i^t||^2 \end{aligned} \tag{5.37}$$

Given that $\bar{x}^0 = x_i^0$ for all clients i, one can see that

$$\sum_{t=0}^{T-1} \sum_{i=1}^{n} p_i ||\bar{x}^{t+1} - x_i^{t+1}||^2 = \sum_{t=0}^{T-1} \sum_{i=1}^{n} p_i ||\bar{x}^{t+1} - x_i^{t+1}||^2 + \sum_{i=1}^{n} p_i ||\bar{x}^0 - x_i^0||^2 \tag{5.38}$$

$$= \sum_{t=0}^{T-1} \sum_{i=1}^{n} p_i ||\bar{x}^t - x_i^t||^2 + \sum_{i=1}^{n} p_i ||\bar{x}^T - x_i^T||^2 \tag{5.39}$$

$$\geq \sum_{t=0}^{T-1} \sum_{i=1}^{n} p_i ||\bar{x}^t - x_i^t||^2. \tag{5.40}$$

Using the result of Equation 5.38 condenses Equation 5.37 into

$$f(\bar{x}^T) - f(\bar{x}^0) \leq - \frac{\gamma}{2n} \sum_{t=0}^{T-1} ||\nabla f(\bar{x}^t)||^2 - \frac{\gamma}{2n} \sum_{t=0}^{T-1} || \sum_{i=1}^{n} p_i \nabla f_i(x_i^t)||^2$$

$$+\left(\frac{3L}{(n^2-3)}+\frac{\gamma L^2 p_{max}}{2n}\right)\underbrace{\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{t=0}^{T-1}\sum_{i=1}^{n}p_i||\bar{x}^{t+1}-x_i^{t+1}||^2}_{:=B} \tag{5.41}$$

Bounding Term B. The recursion of our update rule can be written as

$$X^{t+1}=X^0-\gamma\sum_{j=0}^{t}G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}\bar{W}^q-\gamma\sum_{j=0}^{t}G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q-\bar{W}^q) \tag{5.42}$$

The recursion equation for the expected consensus model $\bar{x}^t$ and expected local model for client i can be computed by multiplying by $\frac{\mathbf{1_n}}{n}$ and $\boldsymbol{e}_i$ respectively

$$\bar{x}^{t+1}=\bar{x}^0-\gamma\sum_{j=0}^{t}G(x_{i_j}^j,\xi_{*,j}^{i_j})\frac{\mathbf{1_n}}{n}-\gamma\sum_{j=0}^{t}G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q-\bar{W}^q)\frac{\mathbf{1_n}}{n} \tag{5.43}$$

$$x_i^{t+1}=x_i^0-\gamma\sum_{j=0}^{t}G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}\bar{W}^q\boldsymbol{e}_i-\gamma\sum_{j=0}^{t}G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q-\bar{W}^q)\boldsymbol{e}_i \tag{5.44}$$

Using the recursive equations above transforms the bound on term B

$$\begin{aligned}&\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{t=0}^{T-1}\sum_{i=1}^{n}p_i||\bar{x}^{t+1}-x_i^{t+1}||^2\\&=\gamma^2\mathbb{E}_{\xi\sim\mathcal{D}_i}\textstyle\sum_{t=0}^{T-1}\sum_{i=1}^{n}p_i\Big|\Big|-\sum_{j=0}^{t}G(x_{i_j}^j,\xi_{*,j}^{i_j})\Big(\frac{\mathbf{1_n}}{n}-\prod_{q=j+1}^{t}\bar{W}^q\boldsymbol{e}_i\Big)-\sum_{j=0}^{t}G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q-\bar{W}^q)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)\Big|\Big|^2\end{aligned} \tag{5.45}$$

$$\begin{aligned}\leq&2\gamma^2\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{t=0}^{T-1}\sum_{i=1}^{n}p_i\Bigg(||\sum_{j=0}^{t}G(x_{i_j}^j,\xi_{*,j}^{i_j})\Big(\frac{\mathbf{1_n}}{n}-\prod_{q=j+1}^{t}\bar{W}^q\boldsymbol{e}_i\Big)||^2\\&+||\sum_{j=0}^{t}G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q-\bar{W}^q)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)||^2\Bigg)\end{aligned} \tag{5.46}$$

$$\begin{aligned}=&2\gamma^2\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{t=0}^{T-1}\sum_{i=1}^{n}p_i\Bigg(\underbrace{\sum_{j=0}^{t}||G(x_{i_j}^j,\xi_{*,j}^{i_j})\Big(\frac{\mathbf{1_n}}{n}-\prod_{q=j+1}^{t}\bar{W}^q\boldsymbol{e}_i\Big)||^2}_{:=B_1}\\&+\underbrace{\sum_{j=0}^{t}||G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q-\bar{W}^q)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)||^2}_{:=B_3}\end{aligned}$$

$$+\underbrace{2\sum_{j=0}^{t}\sum_{j'=j+1}^{t}\langle G(x_{i_j}^{j},\xi_{*,j}^{i_j})(\frac{\mathbf{1_n}}{n}-\prod_{q=j+1}^{t}\bar{W}^q\boldsymbol{e}_i), G(x_{i_{j'}}^{j'},\xi_{*,j'}^{i_{j'}})(\frac{\mathbf{1_n}}{n}-\prod_{q=j'+1}^{t}\bar{W}^q\boldsymbol{e}_i)\rangle}_{:=B_2}$$

$$+\underbrace{2\sum_{j=0}^{t}\sum_{j'=j+1}^{t}\langle G(x_{i_j}^{j},\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q-\bar{W}^q)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i), G(x_{i_{j'}}^{j'},\xi_{*,j'}^{i_{j'}})\prod_{q=j'+1}^{t}(W_{i_q}^q-\bar{W}^q)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)\rangle}_{:=B_4}\Bigg) \quad (5.47)$$

Bounding Term B_1. Using Lemma 36,

$$\begin{aligned}
&\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{j=0}^{t}||G(x_{i_j}^{j},\xi_{*,j}^{i_j})\left(\frac{\mathbf{1_n}}{n}-\prod_{q=j+1}^{t}\bar{W}^q\boldsymbol{e}_i\right)||^2\\
&\leq \mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{j=0}^{t}||G(x_{i_j}^{j},\xi_{*,j}^{i_j})||^2||\left(\frac{\mathbf{1_n}}{n}-\prod_{q=j+1}^{t}\bar{W}^q\boldsymbol{e}_i\right)||^2 \quad (5.48)\\
&= \mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{j=0}^{t}||\frac{1}{M}\sum_{m=1}^{M}\nabla\ell(x_{i_j}^{j};\xi_{m,j}^{i_j})||^2||\left(\frac{\mathbf{1_n}}{n}-\prod_{q=j+1}^{t}\bar{W}^q\boldsymbol{e}_i\right)||^2 \quad (5.49)\\
&\leq \mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{j=0}^{t}||\frac{1}{M}\sum_{m=1}^{M}\nabla\ell(x_{i_j}^{j};\xi_{m,j}^{i_j})||^2(\frac{n-1}{n})\rho^{t-j} \quad (5.50)
\end{aligned}$$

Using Lemma 38, the equation above becomes

$$\begin{aligned}
&= 2\sum_{j=0}^{t}\left(\frac{\sigma^2}{M}+||\nabla f_{i_j}(x_{i_j}^{j})||^2\right)(\frac{n-1}{n})\rho^{t-j} \quad (5.51)\\
&= 2\left(\sum_{j=0}^{t}\frac{\sigma^2}{M}(\frac{n-1}{n})\rho^{t-j}+\sum_{j=0}^{t}||\nabla f_{i_j}(x_{i_j}^{j})||^2(\frac{n-1}{n})\rho^{t-j}\right) \quad (5.52)\\
&\leq \frac{2(n-1)\sigma^2}{(1-\rho)Mn}+2\sum_{j=0}^{t}||\nabla f_{i_j}(x_{i_j}^{j})||^2(\frac{n-1}{n})\rho^{t-j} \quad (5.53)
\end{aligned}$$

Taking the expectation over worker i_j yields the desired bound

$$\begin{aligned}
&\mathbb{E}_{i_j}\left[\frac{2(n-1)\sigma^2}{(1-\rho)Mn}+2\sum_{j=0}^{t}||\nabla f_{i_j}(x_{i_j}^{j})||^2(\frac{n-1}{n})\rho^{t-j}\right]\\
&= \frac{2(n-1)\sigma^2}{(1-\rho)Mn}+\frac{2(n-1)}{n}\sum_{j=0}^{t}\mathbb{E}_{i_j}||\nabla f_{i_j}(x_{i_j}^{j})||^2\rho^{t-j} \quad (5.54)
\end{aligned}$$

Bounding Term B_2. Using Lemma 36,

$$2\sum_{j=0}^{t}\sum_{j'=j+1}^{t}\langle G(x_{i_j}^{j},\xi_{*,j}^{i_j})\big(\frac{\mathbf{1_n}}{n}-\prod_{q=j+1}^{t}\bar{W}^q\boldsymbol{e}_i\big),G(x_{i_{j'}}^{j'},\xi_{*,j'}^{i_{j'}})\big(\frac{\mathbf{1_n}}{n}-\prod_{q=j'+1}^{t}\bar{W}^q\boldsymbol{e}_i\big)\rangle$$

$$=2\sum_{j=0}^{t}\sum_{j'=j+1}^{t}||G(x_{i_j}^{j},\xi_{*,j}^{i_j})||||\big(\tfrac{\mathbf{1_n}}{n}-\textstyle\prod_{q=j+1}^{t}\bar{W}^q\boldsymbol{e}_i\big)||||G(x_{i_{j'}}^{j'},\xi_{*,j'}^{i_{j'}})||||\big(\tfrac{\mathbf{1_n}}{n}-\textstyle\prod_{q=j'+1}^{t}\bar{W}^q\boldsymbol{e}_i\big)|| \tag{5.55}$$

For any $\alpha_{j,j'}>0$ we find

$$\leq 2\sum_{j=0}^{t}\sum_{j'=j+1}^{t}\Bigg(\frac{||G(x_{i_j}^{j},\xi_{*,j}^{i_j})||^2||G(x_{i_{j'}}^{j'},\xi_{*,j'}^{i_{j'}})||^2}{2\alpha_{j,j'}} + \frac{\alpha_{j,j'}||\big(\frac{\mathbf{1_n}}{n}-\prod_{q=j+1}^{t}\bar{W}^q\boldsymbol{e}_i\big)||^2||\big(\frac{\mathbf{1_n}}{n}-\prod_{q=j'+1}^{t}\bar{W}^q\boldsymbol{e}_i\big)||^2}{2}\Bigg) \tag{5.56}$$

$$\leq \sum_{j\neq j'}^{t}\Bigg(\frac{||G(x_{i_j}^{j},\xi_{*,j}^{i_j})||^2||G(x_{i_{j'}}^{j'},\xi_{*,j'}^{i_{j'}})||^2}{2\alpha_{j,j'}}+\frac{\alpha_{j,j'}\rho^{t-\min\{j,j'\}}}{2}(\frac{n-1}{n})^2\Bigg),\alpha_{j,j'}=\alpha_{j',j} \tag{5.57}$$

By applying inequality of arithmetic and geometric means to the term in the last step, we can choose $\alpha_{j,j'}>0$ s.t.

$$\leq \frac{n-1}{n}\sum_{j\neq j'}^{t}\Bigg(||G(x_{i_j}^{j},\xi_{*,j}^{i_j})||||G(x_{i_{j'}}^{j'},\xi_{*,j'}^{i_{j'}})||\rho^{\frac{t-\min\{j,j'\}}{2}}\Bigg) \tag{5.58}$$

$$\leq \frac{n-1}{n}\sum_{j\neq j'}^{t}\Bigg(\frac{||G(x_{i_j}^{j},\xi_{*,j}^{i_j})||^2+||G(x_{i_{j'}}^{j'},\xi_{*,j'}^{i_{j'}})||^2}{2}\rho^{\frac{t-\min\{j,j'\}}{2}}\Bigg) \tag{5.59}$$

$$=\frac{n-1}{n}\sum_{j\neq j'}^{t}||G(x_{i_j}^{j},\xi_{*,j}^{i_j})||^2\rho^{\frac{t-\min\{j,j'\}}{2}} \tag{5.60}$$

$$=\frac{n-1}{n}\sum_{j=0}^{t}\sum_{j'=j+1}^{t}||G(x_{i_j}^{j},\xi_{*,j}^{i_j})||^2\rho^{\frac{t-j}{2}} \tag{5.61}$$

$$=\frac{n-1}{n}\sum_{j=0}^{t}||G(x_{i_j}^{j},\xi_{*,j}^{i_j})||^2 2(t-j)\rho^{\frac{t-j}{2}} \tag{5.62}$$

$$=\frac{n-1}{n}\sum_{j=0}^{t}2(t-j)\rho^{\frac{t-j}{2}}||\frac{1}{M}\sum_{m=1}^{M}\nabla\ell(x_{i_j}^{j};\xi_{m,j}^{i_j})||^2 \tag{5.63}$$

Using Lemma 38 (and the expectation $\mathbb{E}_{\xi\sim\mathcal{D}_i}$ that was omitted above but is present) yields

$$= \frac{2(n-1)}{n}\sum_{j=0}^{t} 2(t-j)\rho^{\frac{t-j}{2}}\left(\frac{\sigma^2}{M} + ||\nabla f_{i_j}(x_{i_j}^j)||^2\right) \tag{5.64}$$

$$\leq \frac{4(n-1)\sqrt{\rho}\sigma^2}{Mn(1-\sqrt{\rho})^2} + \frac{2(n-1)}{n}\sum_{j=0}^{t} ||\nabla f_{i_j}(x_{i_j}^j)||^2 2(t-j)\rho^{\frac{t-j}{2}} \tag{5.65}$$

Taking the expectation over worker i_j yields the desired bound

$$\mathbb{E}_{i_j}\left[\frac{4(n-1)\sqrt{\rho}\sigma^2}{Mn(1-\sqrt{\rho})^2} + \frac{2(n-1)}{n}\sum_{j=0}^{t} ||\nabla f_{i_j}(x_{i_j}^j)||^2 2(t-j)\rho^{\frac{t-j}{2}}\right]$$

$$= \frac{4(n-1)\sqrt{\rho}\sigma^2}{Mn(1-\sqrt{\rho})^2} + \frac{2(n-1)}{n}\sum_{j=0}^{t} \mathbb{E}_{i_j}||\nabla f_{i_j}(x_{i_j}^j)||^2 2(t-j)\rho^{\frac{t-j}{2}} \tag{5.66}$$

Bounding Term B_3.

$$\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{j=0}^{t} ||G(x_{i_j}^j, \xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q - \bar{W}^q)(\frac{\mathbf{1_n}}{n} - \boldsymbol{e}_i)||^2$$

$$=\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{j=0}^{t} ||G(x_{i_j}^j, \xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q - \phi 1^\intercal + \phi 1^\intercal - \bar{W}^q)(\frac{\mathbf{1_n}}{n} - \boldsymbol{e}_i)||^2 \tag{5.67}$$

$$=\mathbb{E}_{\xi\sim\mathcal{D}_i}\textstyle\sum_{j=0}^{t} ||G(x_{i_j}^j, \xi_{*,j}^{i_j})\left[\prod_{q=j+1}^{t}(W_{i_q}^q - \phi 1^\intercal)(\frac{\mathbf{1_n}}{n} - \boldsymbol{e}_i) - \prod_{q=j+1}^{t}(\bar{W}^q - \phi 1^\intercal)(\frac{\mathbf{1_n}}{n} - \boldsymbol{e}_i)\right]||^2 \tag{5.68}$$

$$\leq 2\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{j=0}^{t} ||G(x_{i_j}^j, \xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q - \phi 1^\intercal)(\frac{\mathbf{1_n}}{n} - \boldsymbol{e}_i)||^2$$

$$+ 2\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{j=0}^{t} ||G(x_{i_j}^j, \xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(\bar{W}^q - \phi 1^\intercal)(\frac{\mathbf{1_n}}{n} - \boldsymbol{e}_i)||^2 \tag{5.69}$$

Due to the structure of $\phi 1^\intercal$, multiplying this matrix by $\frac{\mathbf{1_n}}{n}$ or $\boldsymbol{e}_i$ yields the same result. Using this, as well as the double stochasticity of $\bar{W}$, we find

$$= 2\mathbb{E}_{\xi\sim\mathcal{D}_i}\textstyle\sum_{j=0}^{t}\left[||G(x_{i_j}^j, \xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q - \phi 1^\intercal)(\frac{\mathbf{1_n}}{n} - \boldsymbol{e}_i)||^2 + ||G(x_{i_j}^j, \xi_{*,j}^{i_j})\prod_{q=j+1}^{t}\bar{W}^q(\frac{\mathbf{1_n}}{n} - \boldsymbol{e}_i)||^2\right] \tag{5.70}$$

$$= 2\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{j=0}^{t}\Big[||G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q-\phi 1^\intercal)(\tfrac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)||^2+\underbrace{||G(x_{i_j}^j,\xi_{*,j}^{i_j})(\frac{\mathbf{1_n}}{n}-\prod_{q=j+1}^{t}\bar{W}^q\boldsymbol{e}_i)||^2}_{=B_1}\Big] \tag{5.71}$$

Using the result of Lemma 37, as our communication graph $\mathcal{G}$ is uniformly strongly connected and $[W_{i_t}^t]_{i,i}\geq 1/n$ by construction, we see that

$$|\big[\prod_{q=j+1}^{t}\bar{W}^q-\phi 1^\intercal\big]_{h,k}|\leq 4\nu^{t-j-1}\;\forall\; h,k. \tag{5.72}$$

Using this result, we find that

$$|\big[\prod_{q=j+1}^{t}(W_{i_q}^q-\phi 1^\intercal)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)\big]_h|\leq 8(\frac{n-1}{n})\nu^{t-j-1}\;\forall\; h. \tag{5.73}$$

We can remove the -1 exponent by doubling the constant out front

$$|\big[\prod_{q=j+1}^{t}(W_{i_q}^q-\phi 1^\intercal)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)\big]_h|\leq 16(\frac{n-1}{n})\nu^{t-j}\;\forall\; h. \tag{5.74}$$

Finally, using the fact that $G(x_{i_j}^j,\xi_{*,j}^{i_j})$ is all zeros except for one column, the i_j-th column, yields the desired result

$$\begin{aligned}&\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{j=0}^{t}||G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q-\phi 1^\intercal)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)||^2\\&\leq\sum_{j=0}^{t}256(\frac{n-1}{n})^2\nu^{2(t-j)}\mathbb{E}_{\xi\sim\mathcal{D}_i}||\frac{1}{M}\sum_{m=1}^{M}\nabla\ell(x_{i_j}^j,\xi_{*,j}^{i_j})||^2.\end{aligned} \tag{5.75}$$

Utilizing Lemma 38 yields

$$\leq 256(\frac{n-1}{n})^2\sum_{j=0}^{t}\nu^{2(t-j)}\left(\frac{\sigma^2}{M}+||\nabla f_{i_j}(x_{i_j}^t)||^2\right). \tag{5.76}$$

By properties of geometric series, and taking the expectation over worker i_j, we find

$$\leq\frac{256\sigma^2}{(1-\nu^2)M}(\frac{n-1}{n})^2+256(\frac{n-1}{n})^2\sum_{j=0}^{t}\mathbb{E}_{i_j}||\nabla f_{i_j}(x_{i_j}^t)||^2\nu^{2(t-j)}. \tag{5.77}$$

Using the bound of B_1 in the main proof above, we arrive at the final bound of B_3

$$\begin{aligned}
&\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{j=0}^{t}||G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q-\bar{W}^q)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)||^2\\
&\leq\frac{512\sigma^2}{(1-\nu^2)M}(\frac{n-1}{n})^2+\frac{4(n-1)\sigma^2}{(1-\rho)Mn}+512(\frac{n-1}{n})^2\sum_{j=0}^{t}\mathbb{E}_{i_j}||\nabla f_{i_j}(x_{i_j}^t)||^2\nu^{2(t-j)}\\
&\quad+\frac{4(n-1)}{n}\sum_{j=0}^{t}\mathbb{E}_{i_j}||\nabla f_{i_j}(x_{i_j}^j)||^2\rho^{t-j} \qquad (5.78)\\
&\leq\frac{4(n-1)\sigma^2}{Mn}\left(\frac{1}{(1-\rho)}+\frac{128}{(1-\nu^2)}\right)\\
&\quad+\frac{4(n-1)}{n}\sum_{j=0}^{t}\mathbb{E}_{i_j}||\nabla f_{i_j}(x_{i_j}^j)||^2\left(\rho^{t-j}+128\nu^{2(t-j)}\right) \qquad (5.79)
\end{aligned}$$

Bounding Term B_4. Following similar steps as bounding Term B_2 we find

$$\begin{aligned}
&2\mathbb{E}_{\xi\sim\mathcal{D}_i}\textstyle\sum_{j=0}^{t}\sum_{j'=j+1}^{t}\langle G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q-\bar{W}^q)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i),G(x_{i_{j'}}^{j'},\xi_{*,j'}^{i_{j'}})\prod_{q=j'+1}^{t}(W_{i_q}^q-\bar{W}^q)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)\rangle\\
&\leq 2\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{j=0}^{t}\sum_{j'=j+1}^{t}\left(\frac{||G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q-\bar{W}^q)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)||^2}{2}\right.\\
&\quad\left.+\frac{||G(x_{i_{j'}}^{j'},\xi_{*,j'}^{i_{j'}})\prod_{q=j'+1}^{t}(W_{i_q}^q-\bar{W}^q)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)||^2}{2}\right) \qquad (5.80)\\
&\leq\underbrace{2\,\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{j=0}^{t}||G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q-\bar{W}^q)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)||^2}_{=B_3} \qquad (5.81)
\end{aligned}$$

Once again, we can use the proof of Lemma 33 to bound this result.

Finishing Bound of Term B. Putting all terms together, we find that Term B is bounded above by

$$\begin{aligned}
&\sum_{t=0}^{T-1}\sum_{i=1}^{n}p_i||\bar{x}^{t+1}-x_i^{t+1}||^2\\
&\leq 2\gamma^2\sum_{t=0}^{T-1}\sum_{i=1}^{n}p_i\left(\frac{2(n-1)\sigma^2}{(1-\rho)Mn}+\frac{2(n-1)}{n}\sum_{j=0}^{t}\mathbb{E}_{i_j}||\nabla f_{i_j}(x_{i_j}^j)||^2\rho^{t-j}\right.\\
&\quad\left.+\frac{4(n-1)\sqrt{\rho}\sigma^2}{Mn(1-\sqrt{\rho})^2}+\frac{2(n-1)}{n}\sum_{j=0}^{t}\mathbb{E}_{i_j}||\nabla f_{i_j}(x_{i_j}^j)||^2 2(t-j)\rho^{\frac{t-j}{2}}\right.
\end{aligned}$$

$$+\frac{12(n-1)\sigma^2}{Mn}\left(\frac{1}{(1-\rho)}+\frac{128}{(1-\nu^2)}\right)$$
$$+\frac{12(n-1)}{n}\sum_{j=0}^{t}\mathbb{E}_{i_j}||\nabla f_{i_j}(x_{i_j}^j)||^2\left(\rho^{t-j}+128\nu^{2(t-j)}\right)\Bigg) \tag{5.82}$$

Simplifying results in

$$\leq 2\gamma^2\sum_{t=0}^{T-1}\sum_{i=1}^{n}p_i\Bigg(\frac{4(n-1)\sigma^2}{Mn}\left(\frac{7}{2(1-\rho)}+\frac{\sqrt{\rho}}{(1-\sqrt{\rho})^2}+\frac{384}{(1-\nu^2)}\right)$$
$$+\frac{4(n-1)}{n}\sum_{j=0}^{t}\mathbb{E}_{i_j}||\nabla f_{i_j}(x_{i_j}^j)||^2\left(\frac{7}{2}\rho^{t-j}+(t-j)\rho^{\frac{t-j}{2}}+384\nu^{2(t-j)}\right)\Bigg) \tag{5.83}$$
$$\leq\frac{8\gamma^2\sigma^2T}{M}\Bigg(\underbrace{\frac{n-1}{n}(\frac{7}{2(1-\rho)}+\frac{\sqrt{\rho}}{(1-\sqrt{\rho})^2}+\frac{384}{(1-\nu^2)})}_{:=\rho_\nu}\Bigg)$$
$$+\frac{8(n-1)\gamma^2}{n}\sum_{t=0}^{T-1}\sum_{i=1}^{n}p_i\sum_{j=0}^{t}\mathbb{E}_{i_j}||\nabla f_{i_j}(x_{i_j}^j)||^2\left(\frac{7}{2}\rho^{t-j}+(t-j)\rho^{\frac{t-j}{2}}+384\nu^{2(t-j)}\right) \tag{5.84}$$

Using Lemma 39 results in

$$\leq\frac{8\gamma^2\sigma^2T\rho_\nu}{M}+\frac{8(n-1)\gamma^2}{n}\sum_{t=0}^{T-1}\sum_{i=1}^{n}p_i\sum_{j=0}^{t}\Bigg(2||\sum_{i=1}^{n}p_i\nabla f_i(x_i^j)||^2$$
$$+12L^2\sum_{i=1}^{n}p_i||\bar{x}^j-x_i^j||^2+6\zeta^2\Bigg)\left(\frac{7}{2}\rho^{t-j}+(t-j)\rho^{\frac{t-j}{2}}+384\nu^{2(t-j)}\right) \tag{5.85}$$

$$\leq\frac{8\gamma^2\sigma^2T\rho_\nu}{M}+\frac{16(n-1)\gamma^2}{n}\sum_{t=0}^{T-1}\sum_{j=0}^{t}\Bigg(||\sum_{i=1}^{n}p_i\nabla f_i(x_i^j)||^2$$
$$+6L^2\sum_{i=1}^{n}p_i||\bar{x}^j-x_i^j||^2+3\zeta^2\Bigg)\left(\frac{7}{2}\rho^{t-j}+(t-j)\rho^{\frac{t-j}{2}}+384\nu^{2(t-j)}\right) \tag{5.86}$$
$$\leq\frac{8\gamma^2\sigma^2T\rho_\nu}{M}+\frac{16(n-1)\gamma^2}{n}\sum_{j=0}^{T-1}\sum_{t=j+1}^{\infty}\Bigg(||\sum_{i=1}^{n}p_i\nabla f_i(x_i^j)||^2$$
$$+6L^2\sum_{i=1}^{n}p_i||\bar{x}^j-x_i^j||^2+3\zeta^2\Bigg)\left(\frac{7}{2}\rho^{t-j}+(t-j)\rho^{\frac{t-j}{2}}+384\nu^{2(t-j)}\right) \tag{5.87}$$
$$\leq\frac{8\gamma^2\sigma^2T\rho_\nu}{M}+\frac{16(n-1)\gamma^2}{n}\sum_{j=0}^{T-1}\Bigg(||\sum_{i=1}^{n}p_i\nabla f_i(x_i^j)||^2$$

$$+ 6L^2 \sum_{i=1}^{n} p_i ||\bar{x}^j - x_i^j||^2 + 3\zeta^2 \Bigg) \Bigg(\sum_{h=0}^{\infty} \frac{7}{2} \rho^h + h\rho^{\frac{h}{2}} + 384\nu^{2h} \Bigg) \tag{5.88}$$

$$\leq \frac{8\gamma^2\sigma^2 T\rho_\nu}{M} + 16\rho_\nu\gamma^2 \sum_{j=0}^{T-1} \Bigg(|| \sum_{i=1}^{n} p_i \nabla f_i(x_i^j)||^2 + 6L^2 \sum_{i=1}^{n} p_i ||\bar{x}^j - x_i^j||^2 + 3\zeta^2 \Bigg) \tag{5.89}$$

Using Equation 5.38 ends with

$$\leq \frac{8\gamma^2\sigma^2 T\rho_\nu}{M} + 48T\rho_\nu\gamma^2\zeta^2 + 16\rho_\nu\gamma^2 \sum_{t=0}^{T-1} || \sum_{i=1}^{n} p_i \nabla f_i(x_i^t)||^2$$
$$+ 96L^2\rho_\nu\gamma^2 \sum_{t=0}^{T-1} \sum_{i=1}^{n} p_i ||\bar{x}^{t+1} - x_i^{t+1}||^2 \tag{5.90}$$

Now subtract the final term on the right hand side from both sides

$$\sum_{t=0}^{T-1} \sum_{i=1}^{n} p_i ||\bar{x}^{t+1} - x_i^{t+1}||^2 \Bigg(\underbrace{1 - 96L^2\rho_\nu\gamma^2}_{:=z} \Bigg) \leq \frac{8\gamma^2\sigma^2 T\rho_\nu}{M} + 48T\rho_\nu\gamma^2\zeta^2$$
$$+ 16\rho_\nu\gamma^2 \sum_{t=0}^{T-1} || \sum_{i=1}^{n} p_i \nabla f_i(x_i^t)||^2 \tag{5.91}$$

By Lemma 40, we can divide z from both sides

$$\sum_{t=0}^{T-1} \sum_{i=1}^{n} p_i ||\bar{x}^{t+1} - x_i^{t+1}||^2 \leq \frac{8\gamma^2\sigma^2 T\rho_\nu}{Mz} + \frac{48T\rho_\nu\gamma^2\zeta^2}{z} + \frac{16\rho_\nu\gamma^2}{z} \sum_{t=0}^{T-1} || \sum_{i=1}^{n} p_i \nabla f_i(x_i^t)||^2 \tag{5.92}$$

Finishing Bound of Term A. Substituting the bound of B above into Equation 5.41 yields

$$f(\bar{x}^T) - f(\bar{x}^0) \leq -\frac{\gamma}{2n} \sum_{t=0}^{T-1} ||\nabla f(\bar{x}^t)||^2 - \frac{\gamma}{2n} \sum_{t=0}^{T-1} || \sum_{i=1}^{n} p_i \nabla f_i(x_i^t)||^2$$
$$+ \Bigg(\underbrace{\frac{3L}{(n^2-3)} + \frac{\gamma L^2 p_{max}}{2n}}_{:=\phi} \Bigg) \Bigg(\frac{8\gamma^2\sigma^2 T\rho_\nu}{Mz} + \frac{48T\rho_\nu\gamma^2\zeta^2}{z}$$
$$+ \frac{16\rho_\nu\gamma^2}{z} \sum_{t=0}^{T-1} || \sum_{i=1}^{n} p_i \nabla f_i(x_i^t)||^2 \Bigg) \tag{5.93}$$

Rearranging terms simplifies the inequality above to

$$f(\bar{x}^T) - f(\bar{x}^0) \leq -\frac{\gamma}{2n} \sum_{t=0}^{T-1} ||\nabla f(\bar{x}^t)||^2 + \frac{\gamma}{2n} \Bigg(\frac{32\rho_\nu\phi\gamma n}{z} - 1 \Bigg) \sum_{t=0}^{T-1} || \sum_{i=1}^{n} p_i \nabla f_i(x_i^t)||^2$$

$$+\frac{8\phi\gamma^2\sigma^2T\rho_\nu}{Mz}+\frac{48\phi T\rho_\nu\gamma^2\zeta^2}{z} \tag{5.94}$$

From Lemma 42, we find that $(1-\frac{32\rho_\nu\phi\gamma n}{z})\geq 0$. Therefore, the second term of the right hand side above can be removed.

$$f(\bar{x}^T)-f(\bar{x}^0)\leq -\frac{\gamma}{2n}\sum_{t=0}^{T-1}||\nabla f(\bar{x}^t)||^2+\frac{8\phi\gamma^2\sigma^2T\rho_\nu}{Mz}+\frac{48\phi T\rho_\nu\gamma^2\zeta^2}{z} \tag{5.95}$$

Rearranging the inequality above and dividing by T yields

$$\frac{1}{T}\sum_{t=0}^{T-1}||\nabla f(\bar{x}^t)||^2\leq \frac{2n\big(f(\bar{x}^0)-f(\bar{x}^T)\big)}{T\gamma}+\frac{16n\phi\gamma\sigma^2\rho_\nu}{Mz}+\frac{96n\phi\rho_\nu\gamma\zeta^2}{z} \tag{5.96}$$

$$=\frac{2n\big(f(\bar{x}^0)-f(\bar{x}^T)\big)}{T\gamma}+\frac{16\gamma\phi n\rho_\nu}{z}\big(\frac{\sigma^2}{M}+6\zeta^2\big) \tag{5.97}$$

From Lemmas 40 and 41, the inequality above becomes

$$\frac{1}{T}\sum_{t=0}^{T-1}||\nabla f(\bar{x}^t)||^2\leq \frac{2n\big(f(\bar{x}^0)-f(\bar{x}^T)\big)}{T\gamma}+\frac{\frac{1921}{10}L\gamma\rho_\nu}{n}\big(\frac{\sigma^2}{M}+6\zeta^2\big) \tag{5.98}$$

Substituting in the defined step-size γ (as well as its bound) yields

$$\frac{1}{T}\sum_{t=0}^{T-1}||\nabla f(\bar{x}^t)||^2\leq\frac{2n\big(f(\bar{x}^0)-f(\bar{x}^T)\big)}{T}\left(\frac{\sqrt{TL}+\sqrt{M}}{\sqrt{Mn^2\Delta_f}}\right)$$
$$+\frac{\frac{1921}{10}L}{n}\left(\frac{\sqrt{Mn^2\Delta_f}}{\sqrt{TL}}\right)\rho_\nu\big(\frac{\sigma^2}{M}+6\zeta^2\big) \tag{5.99}$$

$$=\frac{2\sqrt{\Delta_f}}{T}+\frac{2\sqrt{L\Delta_f}}{\sqrt{TM}}+\frac{\frac{1921}{10}\sqrt{L\Delta_f}\rho_\nu\big(\sigma^2+6\zeta^2\sqrt{M}\big)}{\sqrt{TM}} \tag{5.100}$$

The final desired result is shown as

$$\frac{1}{T}\sum_{t=0}^{T-1}||\nabla f(\bar{x}^t)||^2\leq\frac{2\sqrt{\Delta_f}}{T}+\frac{2\sqrt{L\Delta_f}\left(1+\frac{1921}{20}\rho_\nu\big(\sigma^2+6\zeta^2\sqrt{M}\big)\right)}{\sqrt{TM}} \tag{5.101}$$

□

5.8.5 Additional Lemmas

Lemma 36 (From Lemma 3 in Lian et al. 2018 [181])**.** *Let W^t be a symmetric doubly stochastic matrix for each iteration t. Then*

$$||\frac{\mathbf{1}_n}{n} - \prod_{t=1}^{T} W^t e_i||^2 \leq \frac{n-1}{n}\rho^T, \ \forall T \geq 0.$$

Lemma 37 (From Corollary 2 in Nedic and Olshevsky 2014 [200])**.** *Let the communication graph $\mathcal{G}$ be uniformly strongly connected (otherwise known as B-strongly-connected for some integer $B > 0$), and $A(t) \in \mathbb{R}^{n\times n}$ be a column stochastic matrix with $[A(t)]_{i,i} \geq 1/n \ \forall i, t$. Define the product of matrices $A(t)$ through $A(s)$ (for $t \geq s \geq 0$) as $A(t:s) := A(t)\dots A(s)$. Then, there exists a stochastic vector $\phi(t) \in \mathbb{R}^n$ such that*

$$|[A(t:s)]_{i,j} - \phi_i(t)| \leq C\nu^{t-s}$$

will always hold for the following values of C and ν,

$$C = 4, \quad \nu = (1 - 1/n^{nB})^{1/B} < 1.$$

Lemma 38. *Under Assumption 1, the following inequality holds*

$$\mathbb{E}_{\xi\sim\mathcal{D}_i}||\frac{1}{M}\sum_{m=1}^{M}\nabla\ell(x_i^t, \xi_{m,t}^i)||^2 \leq \frac{\sigma^2}{M} + ||\nabla f_i(x_i^t)||^2.$$

Lemma 39. *Under Assumption 1, the following inequality holds*

$$\mathbb{E}_i||\nabla f_i(x_i^t)||^2 \leq 2||\sum_{i=1}^{n} p_i\nabla f_i(x_i^t)||^2 + 12L^2\sum_{i=1}^{n} p_i||\bar{x}^t - x_i^t||^2 + 6\zeta^2.$$

Lemma 40. *Given the defined step-size γ and total iterations T in Theorem 34, the term z is*

bounded

$$1 > z := 1 - 96L^2\rho_\nu\gamma^2 \geq 1 - \frac{384}{193^2(775)}.$$

Lemma 41. *Given the defined step-size γ and total iterations T in Theorem 34, the term ϕ is bounded*

$$\phi := \frac{3L}{(n^2-3)} + \frac{L^2\gamma p_{max}}{2n} \leq \frac{L}{n^2}(12 + \frac{2}{775(193)}).$$

Lemma 42. *Given the defined step-size γ and total iterations T in Theorem 34, we find the following bound*

$$\big(1 - \frac{32\rho_\nu\phi\gamma n}{z}\big) \geq 0$$

5.8.5.1 Lemma Proofs

Proof of Lemma 33. These steps are shown in the Main Theorem proof. Due to the symetry and double stochasticity of $\bar{W}$ we find

$$\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{j=0}^{t}||G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q-\bar{W}^q)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)||^2 \tag{5.102}$$

$$=\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{j=0}^{t}||G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q-\phi 1^\intercal+\phi 1^\intercal-\bar{W}^q)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)||^2 \tag{5.103}$$

$$=\mathbb{E}_{\xi\sim\mathcal{D}_i}\textstyle\sum_{j=0}^{t}||G(x_{i_j}^j,\xi_{*,j}^{i_j})\Big[\prod_{q=j+1}^{t}(W_{i_q}^q-\phi 1^\intercal)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)-\prod_{q=j+1}^{t}(\bar{W}^q-\phi 1^\intercal)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)\Big]||^2 \tag{5.104}$$

$$\leq 2\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{j=0}^{t}||G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q-\phi 1^\intercal)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)||^2$$
$$+2\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{j=0}^{t}||G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(\bar{W}^q-\phi 1^\intercal)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)||^2 \tag{5.105}$$

Due to the structure of $\phi 1^\intercal$, multiplying this matrix by $\frac{\mathbf{1_n}}{n}$ or $\boldsymbol{e}_i$ yields the same result. Using this, as well as the double stochasticity of $\bar{W}$, we find

$$= 2\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{j=0}^{t}\left[||G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q-\phi 1^\intercal)(\tfrac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)||^2+||G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}\bar{W}^q(\tfrac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)||^2\right] \tag{5.106}$$

$$= 2\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{j=0}^{t}\Bigg[||G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q-\phi 1^\intercal)(\tfrac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)||^2+\underbrace{||G(x_{i_j}^j,\xi_{*,j}^{i_j})(\frac{\mathbf{1_n}}{n}-\prod_{q=j+1}^{t}\bar{W}^q\boldsymbol{e}_i)||^2}_{=B_1}\Bigg] \tag{5.107}$$

Using the result of Lemma 37, as our communication graph $\mathcal{G}$ is uniformly strongly connected and $[W_{i_t}^t]_{i,i}\geq 1/n$ by construction, we see that

$$|\big[\prod_{q=j+1}^{t}\bar{W}^q-\phi 1^\intercal\big]_{h,k}|\leq 4\nu^{t-j-1}\;\forall\; h,k. \tag{5.108}$$

Using this result, we find that

$$|\big[\prod_{q=j+1}^{t}(W_{i_q}^q-\phi 1^\intercal)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)\big]_h|\leq 8(\frac{n-1}{n})\nu^{t-j-1}\;\forall\; h. \tag{5.109}$$

We can remove the -1 exponent by doubling the constant out front

$$|\big[\prod_{q=j+1}^{t}(W_{i_q}^q-\phi 1^\intercal)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)\big]_h|\leq 16(\frac{n-1}{n})\nu^{t-j}\;\forall\; h. \tag{5.110}$$

Since $G(x_{i_j}^j,\xi_{*,j}^{i_j})$ is all zeros except for one column, the i_j-th column, we find

$$\begin{aligned}&\mathbb{E}_{\xi\sim\mathcal{D}_i}\sum_{j=0}^{t}||G(x_{i_j}^j,\xi_{*,j}^{i_j})\prod_{q=j+1}^{t}(W_{i_q}^q-\phi 1^\intercal)(\frac{\mathbf{1_n}}{n}-\boldsymbol{e}_i)||^2\\&\leq\sum_{j=0}^{t}256(\frac{n-1}{n})^2\nu^{2(t-j)}\mathbb{E}_{\xi\sim\mathcal{D}_i}||\frac{1}{M}\sum_{m=1}^{M}\nabla\ell(x_{i_j}^j,\xi_{*,j}^{i_j})||^2.\end{aligned} \tag{5.111}$$

Utilizing Lemma 38 yields

$$\leq 256(\frac{n-1}{n})^2\sum_{j=0}^{t}\nu^{2(t-j)}\left(\frac{\sigma^2}{M}+||\nabla f_{i_j}(x_{i_j}^t)||^2\right). \tag{5.112}$$

By properties of geometric series, and taking the expectation over worker i_j, we find

$$\leq \frac{256\sigma^2}{(1-\nu^2)M}(\frac{n-1}{n})^2 + 256(\frac{n-1}{n})^2 \sum_{j=0}^{t} \mathbb{E}_{i_j} ||\nabla f_{i_j}(x_{i_j}^t)||^2 \nu^{2(t-j)}. \tag{5.113}$$

Using the bound Equation 5.54 (term B_1) in the main proof above, we arrive at the final bound

$$\begin{aligned}
&\mathbb{E}_{\xi\sim\mathcal{D}_i} \sum_{j=0}^{t} ||G(x_{i_j}^j, \xi_{*,j}^{i_j}) \prod_{q=j+1}^{t} (W_{i_q}^q - \bar{W}^q)(\frac{\mathbf{1_n}}{n} - \boldsymbol{e}_i)||^2 \\
\leq &\frac{512\sigma^2}{(1-\nu^2)M}(\frac{n-1}{n})^2 + \frac{4(n-1)\sigma^2}{(1-\rho)Mn} + 512(\frac{n-1}{n})^2 \sum_{j=0}^{t} \mathbb{E}_{i_j} ||\nabla f_{i_j}(x_{i_j}^t)||^2 \nu^{2(t-j)} \\
&+ \frac{4(n-1)}{n} \sum_{j=0}^{t} \mathbb{E}_{i_j} ||\nabla f_{i_j}(x_{i_j}^j)||^2 \rho^{t-j} \qquad (5.114) \\
\leq &\frac{4(n-1)\sigma^2}{Mn}\left(\frac{1}{(1-\rho)} + \frac{128}{(1-\nu^2)}\right) \\
&+ \frac{4(n-1)}{n} \sum_{j=0}^{t} \mathbb{E}_{i_j} ||\nabla f_{i_j}(x_{i_j}^j)||^2 \left(\rho^{t-j} + 128\nu^{2(t-j)}\right) \qquad (5.115)
\end{aligned}$$

Thus, we have our desired result

$$\mathbb{E} \sum_{j=0}^{t} ||G(x_{i_j}^j, \xi_{*,j}^{i_j}) \prod_{q=j+1}^{t} (W_{i_q}^q - \bar{W}^q)(\frac{\mathbf{1_n}}{n} - \boldsymbol{e}_i)||^2 \leq \mathcal{O}(\frac{\sigma^2}{M} + \mathbb{E} \sum_{j=0}^{t} ||\nabla f_{i_j}(x_{i_j}^j)||^2). \tag{5.116}$$

□

Proof of Lemma 38.

$$\begin{aligned}
&\mathbb{E}_{\xi\sim\mathcal{D}_i} ||\frac{1}{M} \sum_{m=1}^{M} \nabla \ell(x_i^t, \xi_{m,t}^i)||^2 \qquad (5.117) \\
=&\frac{1}{M^2} \mathbb{E}_{\xi\sim\mathcal{D}_i} ||\sum_{m=1}^{M} \nabla \ell(x_i^t, \xi_{m,t}^i) - \nabla f_i(x_i^t) + \nabla f_i(x_i^t)||^2 \qquad (5.118) \\
=&\frac{1}{M^2} \mathbb{E}_{\xi\sim\mathcal{D}_i} ||\sum_{m=1}^{M} \nabla \ell(x_i^t, \xi_{m,t}^i) - \nabla f_i(x_i^t)|| + \mathbb{E}_{\xi\sim\mathcal{D}_i} ||\sum_{m=1}^{M} \nabla f_i(x_i^t)||^2 \\
&+ \frac{2}{M^2} \mathbb{E}_{\xi\sim\mathcal{D}_i} \langle \sum_{m=1}^{M} \left(\nabla \ell(x_i^t, \xi_{m,t}^i) - \nabla f_i(x_i^t)\right), \sum_{m=1}^{M} \nabla f_i(x_i^t) \rangle \qquad (5.119)
\end{aligned}$$

$$=\frac{1}{M^2}\sum_{m=1}^{M}\mathbb{E}_{\xi\sim\mathcal{D}_i}||\nabla\ell(x_i^t,\xi_{m,t}^i)-\nabla f_i(x_i^t)||+M^2||\nabla f_i(x_i^t)||^2 \tag{5.120}$$

$$\leq\frac{\sigma^2}{M}+||\nabla f_i(x_i^t)||^2 \tag{5.121}$$

□

Proof of Lemma 39.

$$\mathbb{E}_i||\nabla f_i(x_i^t)||^2=\mathbb{E}_i||\nabla f_i(x_i^t)-\sum_{i'=1}^{n}p_{i'}\nabla f_i(x_{i'}^t)+\sum_{i'=1}^{n}p_{i'}\nabla f_i(x_{i'}^t)||^2 \tag{5.122}$$

$$\leq 2\mathbb{E}_i\bigg(||\nabla f_i(x_i^t)-\sum_{j=1}^{n}p_j\nabla f_j(x_j^t)||^2+||\sum_{j=1}^{n}p_j\nabla f_j(x_j^t)||^2\bigg) \tag{5.123}$$

The first term on the right hand side can be bounded by

$$\begin{aligned}
&\mathbb{E}_i||\nabla f_i(x_i^t)-\sum_{j=1}^{n}p_j\nabla f_j(x_j^t)||^2\\
=&3\mathbb{E}_i\bigg(||\nabla f_i(x_i^t)-\nabla f_i(\frac{X^t\mathbf{1_n}}{n})||^2+||\sum_{j=1}^{n}p_j\nabla f_j(\frac{X^t\mathbf{1_n}}{n})-\sum_{j=1}^{n}p_j\nabla f_j(x_j^t)||^2\\
&+||\nabla f_i(\frac{X^t\mathbf{1_n}}{n})-\sum_{j=1}^{n}p_j\nabla f_j(\frac{X^t\mathbf{1_n}}{n})||^2\bigg) &(5.124)\\
\leq&3\mathbb{E}_i\bigg(L^2||x_i^t-\bar{x}^t||^2+\sum_{j=1}^{n}p_j||\nabla f_j(\bar{x}^t)-\nabla f_j(x_j^t)||^2+||\nabla f_i(\bar{x}^t)-\nabla f(\bar{x}^t)||^2\bigg) &(5.125)\\
\leq&3\mathbb{E}_i\bigg(L^2||x_i^t-\bar{x}^t||^2+L^2\sum_{j=1}^{n}p_j||\bar{x}^t-x_j^t||^2\bigg)+3\zeta^2 &(5.126)\\
=&6L^2\sum_{i=1}^{n}p_i||\bar{x}^t-x_i^t||^2+3\zeta^2 &(5.127)
\end{aligned}$$

Combining all terms yields the final result

$$\mathbb{E}_i||\nabla f_i(x_i^t)||^2\leq 2||\sum_{i=1}^{n}p_i\nabla f_i(x_i^t)||^2+12L^2\sum_{i=1}^{n}p_i||\bar{x}^t-x_i^t||^2+6\zeta^2 \tag{5.128}$$

□

Proof of Lemma 40. It is trivial to see that $z<1$. We now determine the lower bound of z given

$n \geq 2$, $p_{max} \geq 1/n$, and $\rho_\nu \geq 775/4$

$$z = 1 - 96L^2\rho_\nu\gamma^2 = 1 - 96L^2\rho_\nu(\frac{Mn^2\Delta_f}{TL}) = 1 - 96L^2\rho_\nu(\frac{1}{193^2L^2\rho_\nu^2n^2p_{max}^2}) \tag{5.129}$$

$$= 1 - \frac{96}{193^2\rho_\nu n^2 p_{max}^2} \leq 1 - \frac{384}{193^2(775)} \tag{5.130}$$

□

Proof of Lemma 41. Given $n \geq 2$ and $\rho_\nu \geq 775/4$, and the definition of γ, one can see

$$\phi = (\frac{3L}{(n^2-3)} + \frac{L^2\gamma p_{max}}{2n}) = \frac{L}{n^2}(\frac{3}{(1-3/n^2)} + \frac{L\gamma np_{max}}{2}) \tag{5.131}$$

$$\leq \frac{L}{n^2}(\frac{3}{(1-3/n^2)} + \frac{Lnp_{max}}{2}(\frac{\sqrt{Mn^2\Delta_f}}{\sqrt{TL}})) \tag{5.132}$$

Using the definition of T it follows that

$$\phi \leq \frac{L}{n^2}(\frac{3}{(1-3/n^2)} + \frac{Lnp_{max}}{2}(\frac{1}{193L\rho_\nu np_{max}})) \leq \frac{L}{n^2}(\frac{3}{(1-3/4)} + \frac{1}{386\rho_\nu}) \tag{5.133}$$

$$= \frac{L}{n^2}(12 + \frac{2}{775(193)}) \tag{5.134}$$

□

Proof of Lemma 42. Given $n \geq 2$, $p_{max} \geq 1/n$ Lemma 40, and Lemma 41, one can see

$$1 - 1 - \frac{32\rho_\nu\phi\gamma n}{z} = 1 - \frac{32\rho_\nu\phi n}{z}(\frac{1}{193L\rho_\nu np_{max}}) = 1 - \frac{32n\phi}{193Lz} \tag{5.135}$$

$$\geq 1 - \frac{32(12 + \frac{2}{775(193)})}{193zn} \geq 1 - \frac{16(12 + \frac{2}{775(193)})}{193(1 - \frac{384}{193^2(775)})} \geq 0 \tag{5.136}$$

□

Chapter 6: Large-Scale Distributed Learning via Private On-Device LSH

6.1 Introduction

Locality-sensitive hashing (LSH) has proven to be a remarkably effective tool for memory- and computationally-efficient data clustering and nearest neighbor search [201, 202, 203]. LSH algorithms such as SimHash [201] can be used to search for vectors in collection $W \subset \mathbb{R}^d$ of massive cardinality which will form a large inner product with a reference vector $x \in \mathbb{R}^d$. This procedure, known as maximum inner product search (MIPS) [204], has been applied to neural network (NN) training. In NN training, the weights of a dense layer that are estimated to produce a large inner product with the input (thereby, a large softmax, for example) are activated while the remainder are dropped out.

While LSH-based pruning greatly reduces training costs associated with large-scale models, popular frameworks such as SLIDE [20] and Mongoose [205] cannot be deployed in distributed settings over memory-constrained devices such as GPUs or mobile phones for the following reasons: **(a)** required maintenance of a large target layer in memory and **(b)** access to the input is needed to conduct LSH.

With many modern NN architectures reaching billions of parameters in size, requiring resource-constrained devices to conduct LSH analysis over even part of such a large model is infeasible as it requires many linear projections of massive weights. The hope of offloading this

memory- and computationally-intensive task to a central host in the distributed setting is equally fruitless. LSH-based pruning cannot be conducted by a central host as it requires access to either local client data or hashed mappings of such data. Both of these violate the fundamental host-client privacy contract especially in a federated setting [195]. Therefore, in order to maintain privacy, devices are forced to conduct LSH themselves, returning us back to our original drawback in a vicious circle. We raise the following question then:

Can a resource-constrained device conduct LSH-like pruning of a large dense layer without ever needing to see the entirety of its underlying weight?

This work makes the following contributions to positively resolve this question:

1. Introduce a novel family of hash functions, PGHash, for detection of high cosine similarity amongst vectors. PGHash improves upon the efficiency of SimHash by comparing binarized random projections of *folded* vectors. We prove several statistical properties about PGHash, including angle/norm distortion bounds and that it is an LSH family.

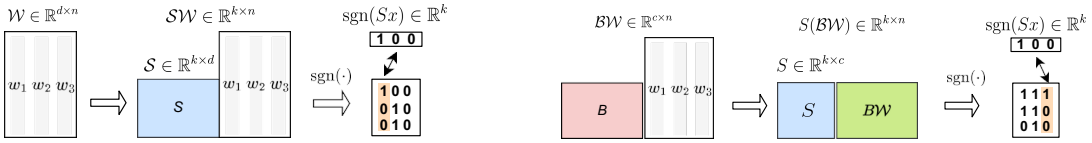
2. Present an algorithmic LSH framework, leveraging our hash functions, which allows for private, personalized, and memory-efficient distributed/federated training of large scale recommender networks via dynamic pruning.

3. Showcase experimentally that our PGHash-based framework is able to efficiently train large-scale recommender networks. Our approach is competitive against a distributed implementation of SLIDE [20] using full-scale LSH. Furthermore, where entry-magnitude similarity is desired over angular similarity (training over Amazon-670K, for example), we empirically demonstrate that using our DWTA [206] variant of PGHash, PGHash-D, matches the performance of using full-scale DWTA.

6.2 Related work

LSH Families. Locality-sensitive hashing families have been used to efficiently solve the approximate nearest neighbors problem [202, 203, 207]. SimHash [201], based on randomized hyperplane projections, is used to estimate cosine similarity. Each SimHash function requires a significant number of random bits if the dimensionality of each target point is large. However, bit reduction using Nisan's pseudorandom generator [208] is often suggested [209, 210]. MinHash [211], a competitor to SimHash [212], measures Jaccard similarity between binary vectors and has been used for document classification. The Winner Take All (WTA) hash [213] compares the ordering of entries by magnitude (corresponding to Kendall-Tau similarity); such comparative reasoning has proven popular in vision applications [214]. However, it was observed that WTA was ineffective at differentiating highly-sparse vectors leading to the development of Densified WTA (DWTA) [206]. Since MinHash, WTA, and DWTA are better suited for binary vector comparison, and we require comparison over real-valued vectors, PGHash is founded on SimHash.

Hash-based Pruning. One of the earlier proposals of pruning based on input-neuron angular similarity via LSH tables is in [10], where a scheme for asynchronous gradient updates amongst multiple threads training over a batch along with hashed backpropagation are also outlined. These principles are executed to great effect in both the SLIDE [20] and Mongoose [205] frameworks for training extreme scale recommender networks. Mongoose improves SLIDE by using an adaptive scheduler to determine when to re-run LSH over a layer weight, and by utilizing learnable hash functions. Both works demonstrated that a CPU using LSH-based dynamic dropout could achieve competitive training complexity against a GPU conducting fully-dense training. Reformer [215] uses LSH to reduce the memory complexity of self-attention layers.



(a) SimHash (b) PGHash

Figure 6.1: **Hash-based dropout.** Two procedures for conducting cosine similarity estimation between full weight matrix $W \in \mathbb{R}^{d\times n}$ (column w_i denotes the weight of neurons i) and an input $x \in \mathbb{R}^d$. **(a)** SimHash generates a hash table via left-multiplication by a randomized rectangular Gaussian matrix $S \sim \mathcal{N}(0, I_d)$ onto the *fully-sized* W. **(b)** PGHash generates a hash table via left-multiplication by a randomized square Gaussian matrix $S \sim \mathcal{N}(0, I_c)$ onto a base projection $BW \in \mathbb{R}^{c\times n}$ of W. In both procedures, weight w_i is selected for activation if its signed hash matches the signed hash of x.

Distributed Recommender Networks. Several works which prune according to input-neuron angular similarity estimations via LSH utilize multiple workers on a single machine [10, 20, 205, 216]. Federated training of recommender networks is an emerging topic of interest, with particular interest in personalized training [217, 218] malicious clients [219, 220], and wireless unreliability [221]. D-SLIDE [222], which is the federated version of SLIDE, eases local on-device memory and computational requirements by sharding the network across clients. However, in the presence of low client numbers, the proportion of the model owned per device can still be taxing, whereas our compression is independent of the number of federated agents. In [223], clients query the server for weights based off the results of LSH conducted using server-provided hash functions. We regard this complete server control over the hashing family, and therefore access to hash-encoding of local client data, as non-private and potentially open to honest-but-curious attacks.

6.3 Preliminaries

Let $W = \{w_1, w_2, \ldots, w_n\} \subset \mathbb{R}^d$ be the weights of a dense hidden layer and $x \in \mathbb{R}^d$ be the input. For brevity, we refer to $W \in \mathbb{R}^{d \times n}$ as the *weight* of the layer. In particular, $w_i \in \mathbb{R}^d$ corresponds to the i^{th} neuron of the layer. We assume that the layer contains dn parameters. Within our work we perform MIPS, as we select weights which produce large inner products with x. Mathematically, we can begin to define this by first letting $p = \max w_i^\top x$ for $1 \leq i \leq n$. For $0 < \epsilon < 1$ we are interested in selecting $\mathcal{S} \subset W$, such that for $\forall w_i \in S$, $w_i^\top x > \epsilon p$. The weights of $\mathcal{S}$ will pass through activation while the rest are dropped out, reducing the computational complexity of the forward and backward pass through this layer. As detailed in Section 6.4 and illustrated in Figure 6.1, we will determine $\mathcal{S}$ by estimating angles with a projection $BW = \{Bw_i\}_{i=1}^n$, with $B \in \mathbb{R}^{c \times d}$ such that $c << d$.

Locality-sensitive Hashing (LSH). LSH [203] is an efficient framework for solving the ϵ-approximate nearest neighbor search (NNS) problem:

Definition 43 (ϵ-NNS). *Given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ in a metric space $\mathcal{M}$ and similarity funcion $Sim : \mathcal{M} \times \mathcal{M} \to \mathbb{R}$ over this space, find a point $p \in P$, such that for a query point $q \in X$ and all $p' \in P$, we have that $Sim(p, q) \leq (1 + \epsilon)Sim(p', q)$.*

It is important to note that the similarity function $Sim(\cdot)$ need not be a distance metric, but rather any general comparison mapping. Popular choices include Euclidean distance and cosine similarity, the latter of which is the primary focus of this paper. The cosine similarity for $x, y \in \mathbb{R}^d$ is defined as $\cos(x, y) \triangleq x^\top y / \big(||x||_2 ||y||_2\big)$. We can frame the MIPS problem described previously as an ϵ-NNS one if we assume that the weights $w_i \in W$ are of unit length.

Thus, we are searching for ϵ-nearest neighbors in W of the query x according to cosine similarity.

Consider a family $\mathcal{H}$ containing hash functions of the form $h : \mathcal{M} \to \mathcal{S}$, where $\mathcal{S}$ is a co-domain with significantly lower feature dimensionality than $\mathcal{M}$. We say that $\mathcal{H}$ is locality-sensitive if the hashes of a pair of points x, y in $\mathcal{M}$, computed by an $h \in \mathcal{H}$ (selected uniformly at random), have a higher collision (matching) probability in S the more similar x and y are according to Sim. We now formally define this notion following [202].

Definition 44 (Locality-sensitive Hashing)**.** *A family $\mathcal{H}$ is called $(S_0, \epsilon S_0, p_1, p_2)$-**sensitive** if for any two points $x, y \in \mathbb{R}^d$ and h chosen uniformly at random from $\mathcal{H}$ satisfies the following,*

- *if* $Sim(x, y) \geq S_0 \Rightarrow Pr(h(x) = h(y)) \geq p_1$,
- *if* $Sim(x, y) \leq \epsilon S_0 \Rightarrow Pr(h(x) = h(y)) \leq p_2$.

For an effective LSH, $p_1 < p_2$ and $\epsilon < 1$ is required. An LSH family allows us to conduct a similarity search over a collection of vectors through comparison of their hashed mappings. Of course, locality loss is inevitable if Sim is a dimension-lowering projection. Through a mixture of increased precision (raising the output dimension of $\mathcal{H}$) and repeated trials (running several trials over independently chosen h), we may tighten the correspondence between Sim and matches over $\mathcal{H}$, following the spirit of the Johnson-Lindenstrauss Lemma [224].

SimHash. A popular LSH algorithm for estimating cosine similarity is SimHash, which uses signed random projections [201] as its hash functions. Specifically, for a collection of vectors $W \subset R^d$, the SimHash family $\mathcal{H}^{Sim}$ consists of hash functions h_v, each indexed by a random Gaussian vector $v \sim \mathcal{N}(0, I_n)$, i.e., an n-dimensional vector with iid entries drawn from $\mathcal{N}(0, 1)$. For $x \in R^n$, we define the hash mapping $h_v(x) := \text{sgn}(v^\top x)$. Here, we modify $\text{sgn}(x)$ to return

1 if $x > 0$, else it returns 0. For Gaussian v chosen uniformly at random and fixed $x, y \in \mathbb{R}^d$, we have $Pr(h_v(x) = h_v(y)) = 1 - \frac{\arccos(\frac{x^\top y}{||x|| \cdot ||y||})}{\pi}$. This hashing scheme was popularized in [225] as part of a randomized approximation algorithm for solving MAX-CUT. Notice that the probability of a hashed pair matching is monotonically increasing with respect to the cosine similarity of x and y, satisfying Definition 44. More precisely, if we set $S_0 = \cos(x, y)$ then $\mathcal{H}^{Sim}$ is $\left(S_0, \epsilon S_0, \left(1 - \arccos(\frac{S_0}{\pi})\right), \left(1 - \arccos(\frac{\epsilon S_0}{\pi})\right)\right)$-sensitive [201, 212]. The above discussion considers the sign of a single random projection, but in practice we will perform multiple projections.

Definition 45 (Hash table)**.** *Let $X = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$ and $V = \{v_1^\top, \ldots, v_k^\top\} \subset \mathcal{N}(0, I_d)$, where k is the **hash length**. Define $h_V : \mathbb{R}^d \to \mathbb{R}^k$ by $[h_V(x)]_i = h_{v_i}(x)$ where $h_{v_i} \in \mathcal{H}^{Sim}$ for $1 \leq i \leq k$. For fixed V, the **hash table** $h_V(X) \in \mathbb{R}^{k \times n}$ is a binary matrix with columns $h_V(x_j)$ for $1 \leq j \leq n$.*

Following the notation above, we may estimate similarity between an input $q \in \mathbb{R}$ and a collection of vectors $X \subset \mathbb{R}^d$ by measuring the Hamming distances (or exact sign matches) between $h_V(q)$ and columns of $h_V(X)$. SimHash is now more discriminatory, as $\mathcal{H}^{Sim}$ can separate R^d into 2^k buckets corresponding to all possible length k binary vectors (which we refer to as **hash codes**). Finally, counting the frequency of exact matches or computing the average Hamming distance over several independently generated hash tables further improves our estimation of closeness. Implementations of the well-known SLIDE framework [20, 216], which utilize SimHash for LSH-based weight pruning, require upwards of 50 tables.

DWTA. Another popular similarity metric is to measure how often high-magnitude entries between two vectors occur at the exact same positions. The densified winner-take-all (DWTA) LSH family [226] estimates this similarity by uniformly drawing k random coordinates

over W and recording the position of the highest-magnitude entry. Similar to SimHash, this process is repeated several times, and vectors with the highest frequency of matches are expected to have similar magnitude ordering. This type of comparative reasoning is useful for computer vision applications [227].

6.4 PGHash

In this section, we develop a family of hash functions $\mathcal{H}^{PG}$ which allow for memory-efficient serial generation of hash tables using a single dimensionally-reduced sketch of W. This is in contrast to traditional LSH frameworks, which produce hash tables via randomized projections over the entirety of W. We first present an overview of the Periodic Gaussian Hash (PGHash) followed by its algorithm for distributed settings, an exploration of several statistical properties regarding the local sensitivity of $\mathcal{H}^{PG}$.

PGHash Motivation. As detailed in Section 6.3, our goal is to efficiently estimate cosine similarity between an input to a layer $x \in \mathbb{R}^d$ and the columns of a large weight matrix $W \in \mathbb{R}^{d \times n}$. SimHash performs hash table generation by first multiplying a matrix of uniformly drawn Gaussian hyperplanes $T_V \in \mathbb{R}^{c \times d}$ with W. The full hash table is computed as $h_V(W) = \text{sgn}(T_V W)$. Then, the j^{th} neuron is activated if $\text{sgn}(T_V x) = h_V(w_j)$ for a layer input x.

One can immediately notice that generation of a new hash table $h_{\tilde{V}}(W)$ requires both **(i)** computation of $T_{\tilde{V}} W$ which requires access to the fully-sized weights and **(ii)** the storage of $T_{\tilde{V}}$ to compute $\text{sgn}(T_{\tilde{V}} x')$ for further inputs x'. *This is problematic for a memory-constrained device, as it would need to maintain both W and $T_{\tilde{V}}$ to generate further tables and perform dynamic*

pruning. To solve this issue, we introduce a family of hash functions generated from a single projection BW of W.

6.4.1 PGHash theory

Definition 46 (Periodic Gaussian Hash)**.** *Assume* ***sketch dimension*** $c << d$ *divides* d *for simplicity. Let* $B = [I_c|I_c|\cdots|I_c] \in \mathbb{R}^{c\times d}$*, where* I_c *is the* $c \times c$ *identity matrix and* $|$ *denotes* $\frac{d}{c}$ *concatenations. Let* $S \in \mathbb{R}^{k\times c}$ *be a random Gaussian matrix with iid entries drawn from* $\mathcal{N}(0,1)$*. We may define a* ***Periodic Gaussian Hash (PGHash)*** *function* $h_S^{PG} : \mathbb{R}^d \to \mathbb{R}^k$ *by* $[h_S^{PG}(x)]_i = [sgn(SBx)]_i$ *for* $1 \leq i \leq k$*. We denote the family of all such hash functions as* $\mathcal{H}^{PG}(c,d)$.

We use the term "periodic" to describe the hash functions described in Definition 46, since unlike SimHash which projects a point via a fully random Gaussian vector as in SimHash, our projection is accomplished using a repeating concatenation of a length c Gaussian vector. Furthermore, for $h_S^{PG} \subset H^{PG}(c,d)$, the matrix representation SB is a tiling of a single Gaussian matrix. Notice that we may easily extend the notion of a PGHash of one vector to an entire hash table over multiple vectors following Definition 45. In this manner, we may generate a sequence of hash tables $\{h_{S_i}^{PG}(W)\}_{i=1}^{\tau}$ over a weight matrix W simply by drawing random Gaussian matrices $S_i \in \mathbb{R}^{k\times c}$ for $1 \leq i \leq \tau$ (where k is the hash length) and computing $\text{sgn}(S_iBW)$.

Extension to DWTA (PGHash-D) Remark. When DWTA (described in Section 6.3) is preferred over SimHash for similarity estimation, we may modify Definition 46 as follows: $B = D_{\mathbb{1}}P$ where P is a $d \times d$ random permutation matrix, and $D_{\mathbb{1}}$ is a $c \times d$ rectangular diagonal matrix with $D_{ii} = 1$ for $1 \leq i \leq c$. We denote our hash functions as $h_S^{PG-D} : \mathbb{R}^d \to \mathbb{R}^k$ by

$h_S^{PG-D}(x) = \max_i [SBx]_i$ for $1 \leq i \leq k$, with $k \leq c$, where S is now a rectangular permutation matrix which selects k rows of Bx at random. We refer to this scheme as PGHash-D, whereas PGHash refers to Definition 46.

Local Memory Complexity Remark. When generating a new table using PGHash, a device maintains S and needs access to just BW, which cost $\mathcal{O}(kc)$ and $\mathcal{O}(cn)$ space complexity respectively. This is much smaller than the $\mathcal{O}(kd)$ and $\mathcal{O}(dn)$ local memory requirements of SimHash.

Sensitivity of $\mathcal{H}^{PG}$. In this section, we will explore the sensitivity of $\mathcal{H}^{PG}(c, d)$.

Definition 47. *Let $x \in \mathbb{R}^d$ and $c \in \mathbb{R}$ such that $c|d$. Define the (d, c)-**folding** $x_c \in \mathbb{R}^c$ of x as $[x_c]_i = \sum_{j=1}^{\frac{d}{c}} [x]_{i+j \cdot \frac{d}{c}}$. Equivalently, $x_c = Bx$, with B as specified in Definition 46.*

Theorem 48. *Let $x, y \in R^d$. Define the following similarity function $Sim_c^d(x, y) \triangleq \cos(x_c, y_c)$, where x_c, y_c are (d, c)-foldings of x, y. $\mathcal{H}^{PG}(c, d)$ is an LSH family with respect to Sim_c^d.*

Proof. Let $h_v \in \mathcal{H}^{PG}$. This means that for a randomly chosen $v' \sim \mathcal{N}(0, I_{\frac{d}{c}})$, v is a c-times concatenation of v. We see that $\text{sgn}(v^\top x) = \text{sgn}\left(\frac{v^\top x}{||v|| \cdot ||x||}\right) = \text{sgn}\left(\sqrt{\frac{c}{d}} \frac{||x_c||}{||x||} \frac{v'^\top x_c}{||v'|| ||x_c||}\right)$. Since sgn is unaffected by the positive multiplicative factors, we conclude that $\text{sgn}(v^\top x) = \text{sgn}(v'^\top x_c)$. Through symmetric argument, we find $\text{sgn}(v^\top y) = \text{sgn}(v'^\top y_c)$. Since $v' \sim \mathcal{N}(0, I_c)$, comparing the sign of $v^\top x$ to $v^\top y$ is equivalent to a standard SimHash over x_c and y_c, i.e., estimation of $\cos(x_c, y_c)$. □

Corollary 49. *Let $x, y \in \mathbb{R}^d$, then $\mathcal{H}^{PG}(c, d)$ is $\left(S_c, \epsilon S_c, \left(1 - \arccos(\frac{S_c}{\pi})\right), \left(1 - \arccos(\frac{\epsilon S_c}{\pi})\right)\right)$-sensitive where $S_c = \cos(x_c, y_c)$,.*

Proof. This follows directly from the well-known sensitivity of SimHash [201]. □

We see that $\mathcal{H}^{PG}$ is LSH with respect to the angle between (d, c)-foldings of vectors. The use of periodic Gaussian vectors restricts the degrees of freedom (from d to d/c) of our projections. However, the usage of pseudo-random and/or non-iid hash tables has been observed to perform well in certain regimes[228, 229]. *Although $\mathcal{H}^{PG}(c, d)$ is LSH, is $\cos(x_c, y_c)$ necessarily an acceptable proxy for $\cos(x, y)$, in particular, for high angular similarity?* Heuristically, yes, for highly-cosine similar vectors: assuming x and y are both unit (since scaling does not affect angle) then we have that $||x - y||^2 = 2 - 2\cos(x, y)$. If ℓ_2 similarity between x and y is already high, then the ℓ_2 similarity of their (normalized) (d, c)-foldings will also be high, and thus their cosine similarity as well. We now provide a characterization on the angle distortion of a (d, c)-folding.

Theorem 50. *Let $x, y \in \mathbb{S}^{d-1}$. Assume that neither x nor y vanish under multiplication by B and that the set $S_{x,y} = \{v \in \text{span}(x, y) \ : ||v|| = 1\}$ does not contain a 0-eigenvector of $B^\top B$. We denote the following quantities: $\lambda = d/c$, $\theta := \frac{1}{2}\arccos(\cos(x, y))$, $\alpha = \inf\{c > 0 \mid |||Bv|| \geq c, \forall v \in S_{x,y}\}$, and $\beta = \alpha/\lambda$. ($\alpha > 0$ since $S_{x,y}$ does not contain any 0-eigenvectors.) Then $\cos(x_c, y_c)$ lives between $\frac{1-\beta^2\tan^2\theta}{1+\beta^2\tan^2\theta}$ and $-\frac{\tan^2\theta^2-\beta^2}{\tan^2\theta^2+\beta^2}$.*

Proof sketch. Consider the unit circle $S_{x,y}$ contained in span(x, y) (Let us assume x and y are unit, WLOG). The linear distortion $BS_{x,y}$ is an ellipse containing $x_c = Bx$ and $y_c = By$. The length of the axes of this ellipse are determined by the eigenvalues of $B^\top B$. The bounds follow from further trigonometric arguments, by considering when the axes of $BS_{x,y}$ are maximally stretched and shrunk respectively. These distortions are strongly related to λ and β.

We can see that as $\cos(x, y) \to 0$ we have $\cos(x_c, y_c) \to 0$. It is natural to consider the distribution of α in Theorem 50 as how extreme shrinking by B (the folding matrix) can greatly

Algorithm 11 `Distributed PGHash`

Require: weights $\{W_0^\ell\}_{\ell=1}^L$, objective $\mathcal{F}$, N devices, target layer P, T iterations, folding matrix B, τ hash tables, hash length k, sketch dim c, comp. ratio CR.

Server Executes:
for $t = 1, 2, \ldots, T$ **do**
  Compile pre-target weights $W_A = \{W_t^\ell\}_{\ell=1}^{P-1}$
  Compile post-target weights $W_B = \{W_t^\ell\}_{\ell=P+1}^{L}$
  for each device i **in parallel do**
    $\Theta_i \leftarrow$ **DeviceLSH**(i, W_A, BW_t^P, CR, τ, k, c)
    $W_{t+1,i} \leftarrow$ **DeviceUpdate**(i, $W_t^P(\Theta_i) \bigcup W_B$)
  end for
  $W_{t+1} \leftarrow$ Average the active weights $W_{t+1,i}$ across all devices
end forreturn $\{W_T^\ell\}_{\ell=1}^L$

DeviceLSH$(i, W_A, BW, CR, \tau, k, c)$:
Sample input ξ from local data distribution $\mathcal{D}_i$
$\xi^{P-1} = \mathcal{F}(W_A, \xi)$ $\overline{W_A \text{ is stored locally}}$
$\Theta_i \leftarrow$ `PGHash`$(\xi^{P-1}, BW, CR, \tau, k, c)$
return Θ_i

DeviceUpdate(i, W_B):
$W \leftarrow W_A \bigcup W_B$
return $W - \eta \nabla_W \mathcal{F}(W; \xi)$

Algorithm 12 `PGHash`

Require: batched input $X \in \mathbb{R}^{d\times M}$, projected weight $BW \in \mathbb{R}^{c\times n}$, compression rate CR, τ hash tables, hash length k, sketch dim c.

```
Set Θ = [ ]
for i = 1, 2, ..., τ do
    Draw random Gaussian S ∈ R^{k×c}
    for each sample x in X do
        for j = 1, 2, ..., n do
            if sgn(SBx) = [sgn(SBW)]_{:,j} then
                Θ.add(j)
            end if
        end for
    end for
    if |Θ| > ⌊CR · n⌋ then
        Randomly remove selected neurons
    from table-i until |Θ| = ⌊CR · n⌋ return Θ
    end if
end for
return Θ
```

distort the angle. We can characterize this statistical distribution exactly.

Proposition 51. *Let $u \in S^{d-1}$, drawn uniformly at random. Then $||Bu||^2 \sim \text{Beta}(\frac{c}{2}, \frac{d-c}{2}, 0, \frac{d}{c})$, the four parameter Beta distribution with pdf $f(x) = \frac{(2x/c)^{c/2-1}(1-2x/c)^{(d-c)/2-1}}{(d/c)\beta(c/2,(d-c)/2}$ and $\mathbb{E}||Bu||^2 = 1$.*

We defer proof of Proposition 51 to Section 6.7.5. Since the folded magnitude $||Bu||^2$ is unit in expectation, the distortion term α^2/λ^2 in Theorem 50 will often be close to 1, greatly tightening the angle distortion bounds.

6.4.2 PGHash algorithm

Below, we detail our protocol for deploying PGHash in a centralized distributed setting (presented algorithmically in Algorithm 11). Over a network of N devices, the central host identifies a target layer P whose weight $W_t^P \in \mathbb{R}^{d\times n}$ (at iteration t) is too expensive for memory-

constrained devices to fully train or host in local memory. Neurons (columns of W_t^P) are pruned by devices according to its estimated cosine similarity to the output x^{P-1} of the previous layer.

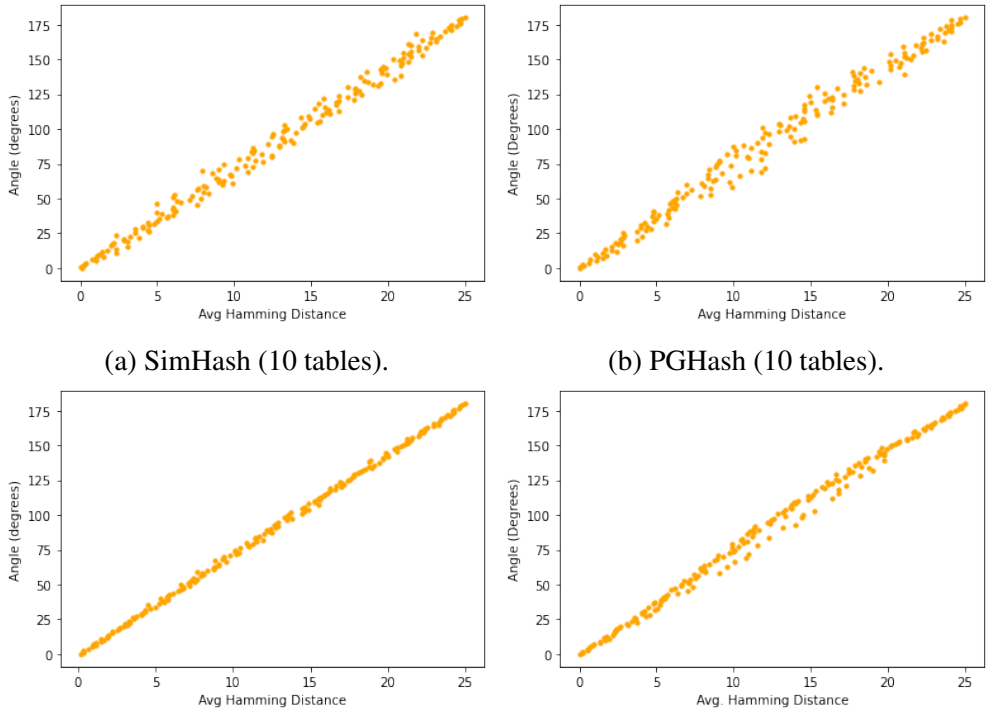
The central host begins each round of distributed training t by sending each device **(1)** all weights $\{W_t^\ell\}_{\ell=1}^{P-1}$ required to generate the input x^{P-1} and **(2)** the compressed target layer BW_t^P. Using these weights, each device conducts PGHash analysis (via Algorithm 12) using its current batch of local data to determine its activated neurons. The central host sends each device i their set of activated neurons $W_t^P(\Theta_i)$, and each device performs a standard gradient update on their new model $\{W_t^\ell\}_{\ell=1}^{P-1} \bigcup W_t^P(\Theta_i) \bigcup \{W_t^\ell\}_{\ell=P+1}^{L}$. Finally, the central host receives updated models from each device and averages only the weights which are activated during training.

The on-device PGHash analysis (Algorithm 12) consists of first running a forward pass up to the target layer to generate the input x^{P-1}. Devices generate personal hash tables by performing left-multiplication of BW_t^P and x^{P-1} by a random Gaussian matrix $S \sim \mathcal{N}(0, I_c)$, as described in Section 6.3. The number of tables τ is specified by the user. A neuron j is marked as active if the input hash code $\text{sgn}(Sx^{P-1})$ is identical to j-th weight's hash code $\text{sgn}(SB[W_t^P]_{:,j})$. In Section 6.7.1.2, we detail how Hamming distance can also be used for neuron selection.

Computational Complexity Remark. Through the use of dynamic pruning, PGHash significantly reduces both the forward and backward training computational complexities. PGHash activates *at most* $CR \cdot n$ neurons per sample as opposed to n for full training. In practice, PGHash activates only a fraction of the $CR \cdot n$ neurons (as shown in Figure 6.6a). Therefore, the number of floating point operations within forward and backward training is dramatically reduced.

Communication Complexity Remark. By reducing the size of the model needed in local memory and subsequently requesting a pruned version of the architecture we improve communication efficiency. For a fixed number of rounds T and target weight size dn, the total communi-

cation complexity, with respect to this data structure, is $\mathcal{O}(T \cdot CR \cdot dn)$, which significantly less bits than the vanilla $\mathcal{O}(Tdn)$ communication cost of vanilla federated training. In Section 4.6, we show that PGHash achieves near state-of-the-art results with only $CR = 0.1$ (10% of a massive weight matrix).



(a) SimHash (10 tables).

(b) PGHash (10 tables).

(c) SimHash (100 tables).

(d) PGHash (100 tables).

Figure 6.2: **Correlation between angle and Hamming distance.** We plot the average Hamming distance (x-axis) between a PG/SimHash hashed fixed vector x and collection of vectors $\mathcal{V}$ versus their true angles (y-axis). Vectors are unit, length 100, and hashed down to dimension $k = 25$ binary vectors according to $\tau = 10$ or 100 hash tables. PGHash has a sketch dimension of $c = 25$. Both PGHash and SimHash show strong correlation between Hamming distance and angular similarity.

6.5 Experiments

In this section, we **(1)** gauge the sensitivity of PGHash and **(2)** analyze the performance of PGHash and our own DWTA variant (PGHash-D) in training large-scale recommender networks.

PGHash and PGHash-D require only 6.25% ($c = 8$) of the final layer sent by the server to perform on-device LSH in our experiments. In PGHash, devices receive the compressed matrix $BW \in \mathbb{R}^{c \times n}$ via the procedure outlined in Section 6.4. In PGHash-D, devices receive c out of d randomly selected coordinates for all n neurons in the final layer weight. Using k of the c coordinates (ensuring privacy since the server is unaware of the coordinates used for LSH), PGHash-D selects neurons which, within the k coordinates, share the same index of highest-magnitude entry between the input and weight. We employ PGHash for Delicious-200K and PGHash-D for Amazon-670K and WikiLSHTC-325K.

PGHash Sensitivity Analysis. Our first experiment measures the ability of $\mathcal{H}^{PG}(c, d)$ to estimate cosine similarity. We produce a fixed unit vector $x \in \mathbb{R}^{100}$ and set of 180 vectors $\{v_i\}_{i=1}^{180}$ of the same dimension. Both the Gaussian vector x and collection of vectors V are fed through varying numbers of SimHash and PGHash tables. We produce a scatter plot measuring the correlation between angle and average Hamming distance. PGHash, as seen in Figure 6.2, is an effective estimator of cosine similarity. We observe that PGHash, like SimHash, successfully produces low average Hamming distances for vectors that are indeed close in angle. This provides evidence that selecting neurons with exact hash code matches (vanilla sampling) is effective for choosing neurons which are close in angle to the input vector. Finally, we find increasing the number of hash tables helps reduce variance.

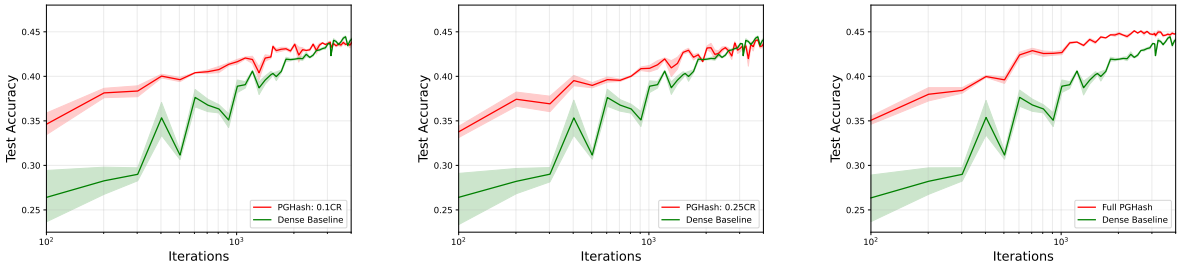
Large-Scale Recommender Network Training. Our second experiment tests how well PGHash(-D) can train large-scale recommender networks. We train these networks efficiently by utilizing dynamic neuronal dropout as done in [20]. We use three extreme multi-label datasets for training recommender networks: Delicious-200K, Amazon-670K, and WikiLSHTC-325K. These datasets come from the Extreme Classification Repository [19]. The dimensionality of

these datasets is large: 782,585/205,443 (Delicious-200K), 135,909/670,091 (Amazon-670K), and 1,617,899/325,056 (WikiLSHTC-325K) features/labels. Due to space, Wiki results are found in Section 6.7.1.3.

The feature and label sets of these datasets are extremely sparse. Akin to [20, 205, 230], we train a recommender network using a fully-connected neural network with a single hidden layer of size 128. Therefore, for Amazon-670K, our two dense layers have weight matrices of size $(135,909 \times 128)$ and $(128 \times 670,091)$. The final layer weights output logits for label prediction, and we use PGHash(-D) to prune its size to improve computational efficiency during training.

Unlike [20, 205, 230], PGHash(-D) can be deployed in a federated setting. Within our experiments, we show the efficacy of PGHash for both single- and multi-device settings. Training in the federated setting (following the protocols of Algorithm 11) allows each device to rapidly train portions of the entire neural network in tandem. We partition data evenly (in an IID manner) amongst devices. Finally, we train our neural network using TensorFlow. We use the Adam [231] optimizer with an initial learning rate of 1e-4. A detailed list of the hyper-parameters we use in our experiments can be found in Section 6.7.1.1. Accuracy in our figures refers to the $P@1$ metric, which measures whether the predicted label with the highest probability is within the true list of labels. These experiments are run on a cloud cluster using Intel Xeon Silver 4216 processors with 128GB of total memory.

Sampling Strategy. One important aspect of training is how we select activated neurons for each sample through LSH. Like [20], we utilize vanilla sampling. In our vanilla sampling protocol, a total of $CR \cdot n$ neurons are selected across the entire sampled batch of data. As detailed in Section 6.4 and Algorithm 12, a neuron is selected when its hash code exactly matches the hash code of the input. We retrieve neurons until either $CR \cdot n$ are selected or all τ tables



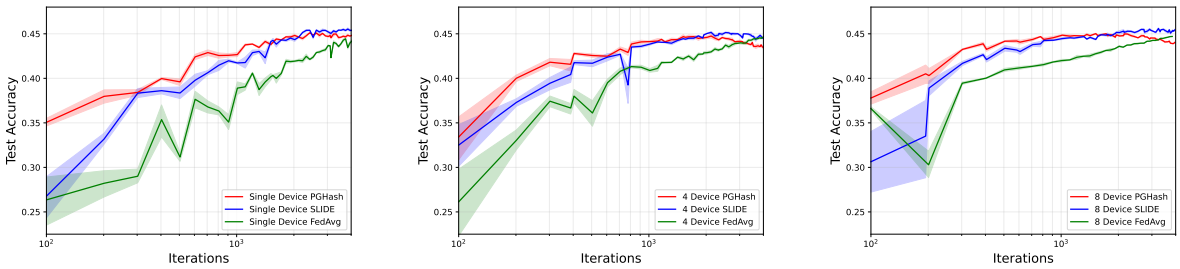
(a) 90% Compression. (b) 75% Compression. (c) No Compression.

Figure 6.3: **Compressed PGHash.** We record model accuracy of a large recommendation network on an extreme classification task (Delicious-200K) using PGHash for varying compression rates (CR). Compressed PGHash, even at 90% compression, is competitive with full training (*without even including effects of sparsity-induced neuronal drop out*). Hyperparameters are in Appendix 6.7.1.1.

have been looked up.

Compression Efficacy. We begin by analyzing how PGHash performs when varying the compression rate CR. Figure 6.3 showcases how PGHash performs for compression rates of 75% and 90% as well as no compression. Interestingly, PGHash reaches near-optimal accuracy even when compressed. This shows the effectiveness of PGHash at accurately selecting fruitful active neurons given a batch of data. The difference between the convergence of PGHash for varying compression rates lies within the volatility of training. As expected, PGHash experiences more volatile training (Figures 6.3a and 6.3b) when undergoing compression as compared to non-compressed training (Figure 6.3c).

Distributed Efficacy. In Figures 6.4 and 6.5, we analyze how well PGHash(-D) performs in a federated setting. We compare PGHash(-D) to a federated version of SLIDE [20] that we implemented (using respectively, a full SimHash or DWTA), as well as fully-dense Federated Averaging (FedAvg) for Delicious-200K. One can immediately see in Figures 6.4 and 6.5 that PGHash(-D) performs identically to, or better than, Federated SLIDE. In fact, for Delicious-



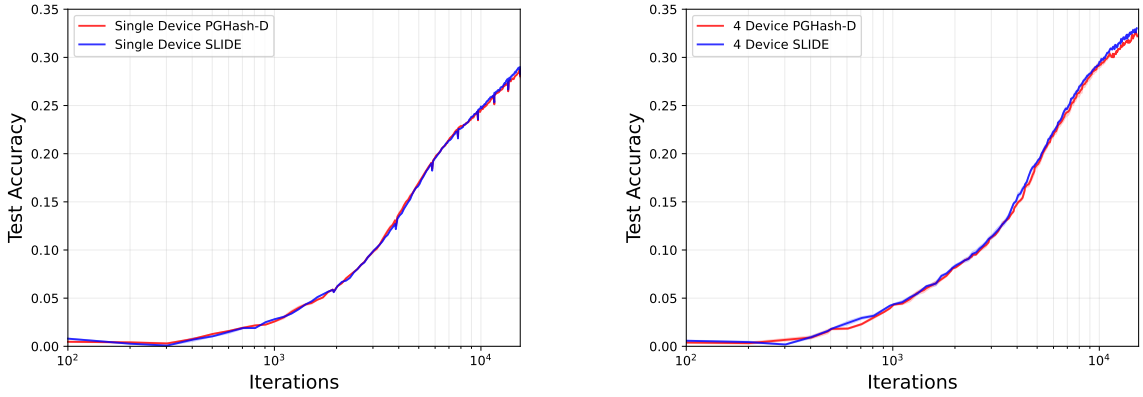
(a) 1 Device Results. (b) 4 Device Results. (c) 8 Device Results.

Figure 6.4: **Federated Delicious-200K PGHash.** We record model accuracy of a large recommendation network on an extreme classification task (Delicious-200K) trained in a federated setting. PGHash achieves competitive accuracies compared with Federated SLIDE and FedAvg. In fact, PGHash converges quicker to near-optimal accuracy. Hyperparameters are in Appendix 6.7.1.1.

200K, PGHash and Federated SLIDE outperform the dense baseline (FedAvg). In Section 6.7.1.3 we detail the difficulties of PGHash-D and SLIDE in matching the dense baseline as well as the failure of SimHash to achieve performance akin to DWTA for Amazon-670K.

PGHash(-D) and Federated SLIDE smartly train portions of the network related to each batch of local device data, via LSH, in order to make up for the lack of a full output layer. However, unlike Federated SLIDE, PGHash(-D) can perform on-device LSH *using as little as 6.25% of the full weight* W ($c = 8$) for both Delicious-200K and Amazon-670K experiments. Furthermore, for Delicious-200K, PGHash generates a dense Gaussian that is only 6.25% ($c = 8$) the size of that for Federated SLIDE. In summary, PGHash(-D) attains similar performance to Federated SLIDE while storing less than a tenth of the parameters.

Induced Sparsity. PGHash(-D) induces a large amount of sparsity through its LSH process. This is especially prevalent in large-scale recommender networks, where the number of labels for each data point is a miniscule fraction of the total output layer size (*e.g.* Delicious-200K has on average only 75.54 labels per point). PGHash(-D) performs well at identifying this



(a) 1 Device Results.

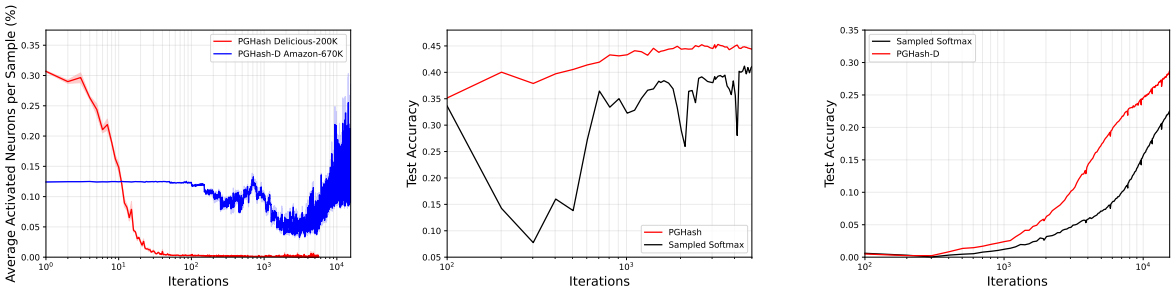
(b) 4 Device Results.

Figure 6.5: **Federated Amazon-670K PGHash-D.** We record model accuracy of a large recommendation network on Amazon-670K trained in a federated setting. PGHash-D matches the convergence of Federated SLIDE without requiring LSH to be performed by the central server.

small subset of neurons as training progresses. As one can see in Figure 6.6a, even when PGHash is allowed to select all possible neurons (*i.e.,* no compression $CR = 1$), it still manages to *select fewer than 1% of the total neurons after only 50 iterations of training over Delicious-200K*. For Amazon-670K, PGHash-D requires less than 30% of the total neurons for the majority of training even. Therefore, PGHash(-D) greatly increases sparsity within the NN, improving the computational efficiency of the algorithm by reducing the number of floating point operations required in the forward and backward training.

6.6 Conclusion

In this work, we present a new hashing family, PGHash, which enables the generation of multiple LSH hash tables using a single base projection of a massive target weight. These hash tables can be used to dynamically select for neurons which are similar to the layer input. This alleviates memory, communication, and privacy costs associated with conventional LSH-training



(a) Average Activated Neurons. (b) Sampled Softmax (Delicious). (c) Sampled Softmax (Amazon).

Figure 6.6: **PGHash(-D) Computational Efficiency.** In Figure **6.6a**, we showcase that PGHash(-D) activates only a fraction of the total final layer neurons *even without compression*. Through this induced sparsity, PGHash(-D) greatly reduces the computational complexity of forward and backward training compared to full training. In Figures **6.6b** and **6.6c**, we compare PGHash(-D) with the Sampled Softmax heuristic [7] (randomly sampling 10% of the total neurons) for efficiently training recommender networks. PGHash(-D) outperforms the Sampled Softmax baseline, as it selects a better set of activated neurons via LSH to more efficiently train the recommender network.

approaches. As a proof of concept, we demonstrate that (i) the PGHash family is effective at mimicking SimHash and (ii) our framework is competitive against other, memory-inefficient, LSH-based federated training baselines of large-scale recommender networks. For future work, we intend to explore how multi-layer PGHash pruning affects model performance and incorporate learnable hashes as in the Mongoose [205] pipeline.

Limitations. Our theory indicates that PGHash is useful for detecting high angular similarity, but could prove unreliable for differentiating between intermediately dissimilar vectors. Additionally, LSH-based pruning has only shown success on large classification layers or attention layers in transformers [215]. When considering broader impacts, large-scale recommender networks, and any subsequent improvements to their design, can be used for strategically negative advertising purposes.

6.7 Supplemental Material

6.7.1 Experiment details

In this section, we provide deeper background into how our experiments were run as well as some additional results and observations. We first detail the hyper-parameters we used in order to reproduce our results. Then, we provide additional comments and details into our sampling approach. Finally, we describe some of the interesting observations we encountered while training the Amazon-670K and Wiki-325K recommender networks.

6.7.1.1 Experiment hyper-parameters

Below, we detail the hyper-parameters we used when running our federated experiments.

Table 6.1: Hyper-parameters for Federated Experiments (PGHash and Federated SLIDE).

Dataset	**Algorithm**	**Hash Type**	**LR**	**Batch Size**	**Steps per LSH**	k	c	**Tables**	CR
Delicious-200K	PGHash	PGHash	1e-4	128	1	8	8	50	1
Delicious-200K	SLIDE	SimHash	1e-4	128	1	8	N/A	50	1
Amazon-670K	PGHash	PGHash-D	1e-4	256	50	8	8	50	1
Amazon-670K	SLIDE	DWTA	1e-4	256	50	8	N/A	50	1
Wiki-325K	PGHash	PGHash-D	1e-4	256	50	5	16	50	1
Wiki-325K	SLIDE	DWTA	1e-4	256	50	5	N/A	50	1

What one can immediately see from Table 6.1, is that we use a Densified Winner Take All (DWTA) variant of PGHash for the larger output datasets Amazon-670K and Wiki-325K. As experienced in [20, 205, 216], SimHash fails to perform well on these larger datasets. We surmise that SimHash fails due in part to its inability to select a large enough number of neurons per sample (we observed this dearth of activated neurons empirically). Reducing the hash length k does increase the number of neurons selected, however this decreases the accuracy. Therefore, DWTA is used because it utilizes more neurons per sample on these larger problems and also still

achieves good accuracy.

Table 6.2: **Hyper-parameters for Compression Experiments (PGHash).**

Dataset	Algorithm	Hash Type	LR	Batch Size	Steps per LSH	k	c	Tables	CR
Delicious-200K	PGHash	PGHash	1e-4	128	1	8	8	50	0.1/0.25/1

As a quick note, we record test accuracy every so often (around 100 iterations for Delicious-200K and Amazon-670K). Similar to [20], to reduce the test accuracy computations (as the test sets are very large) we compute the test accuracy of 30 randomly sampled large batches of test data.

6.7.1.2 Neuron sampling

Speed of Neuron Sampling. In Table 6.3 we display the time it takes to perform LSH for PGHash given a set number of tables. These times were collected locally during training. The entries in Table 6.3 denote the time it takes to compute hashing of the final layer weights w_i and each sample x in batch M as well as vanilla-style matching (neuron selection) for each sample.

Table 6.3: **Average LSH time for PGHash over a range of tables.** We compute the average μ time (and standard deviation σ) it takes for PGHash to perform *vanilla sampling* (exact matches) between the hash codes of sample x and each weight w_i in the final dense layer. Times are sampled for PGHash on Delicious-200k for batch size $M = 128$, $k = 9$, and $c = 8$ for one device.

Method	1 table (seconds)	50 tables (seconds)	100 tables (seconds)
PGHash	$\mu = 0.0807, \sigma = 0.0076$	$\mu = 3.1113, \sigma = 0.0555$	$\mu = 6.2091, \sigma = 0.1642$
SLIDE	$\mu = 0.0825, \sigma = 0.0099$	$\mu = 3.2443, \sigma = 0.1671$	$\mu = 6.2944, \sigma = 0.0689$

We find in Table 6.3 that PGHash achieves near sub-linear speed with respect to the number of tables τ and slightly outperforms SLIDE. PGHash edges out SLIDE due to the smaller matrix multiplication cost, as PGHash utilizes a smaller random Gaussian matrix (size $c \times c$). The speed-up over SLIDE will become more significant when the input layer is larger (as $d = 128$ in our

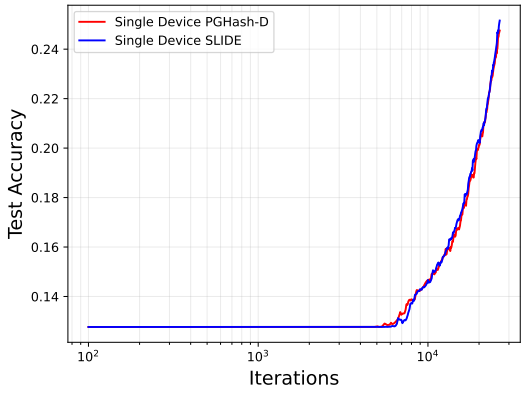
experiments). Therefore, PGHash obtains superior sampling performance to SLIDE.

Hamming Distance Sampling. An alternative method to vanilla sampling is to instead select final layer weights (neurons) w_i which have a small Hamming distance relative to a given sample x. As a refresher, the Hamming distance simply computes the number of non-matching entries between two binary codes (strings). If two binary codes match exactly, then the Hamming distance is zero. In this sampling routine, either **(i)** the top-k weights w_i with the smallest Hamming distance to sample x are selected to be activated or **(ii)** all weights w_i with a Hamming distance of β or smaller to sample x are selected to be activated. Interestingly, the vanilla-sampling approach we use in our work is equivalent to using $\beta = 0$ in **(ii)**.

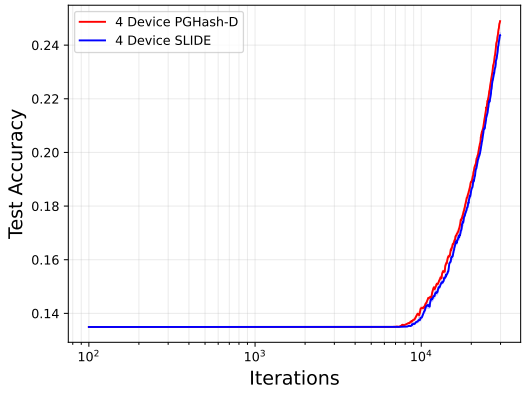
In either of the scenarios listed above, hash codes for w_i and x are computed as done in PGHash(-D). From there, however, the hash code for x is compared to the hash codes for all final layer weights in order to compute the Hamming distance for each w_i. The process of computing n Hamming distances for each sample x is very expensive (much harder than just finding exact matches). That is why our work, as well as [20, 205], use vanilla sampling instead of other methods.

6.7.1.3 Amazon-670K and Wiki-325K experiment analysis

Sub-par SimHash Performance. SimHash is known to perform worse than DWTA on Amazon-670K and Wiki-325K. Utilizing SimHash for these experiments is unfair as it is shown by [20, 205], for example, that DWTA achieves much higher performance on Amazon-670K. For this reason, DWTA is the chosen hash function in [20] for Amazon-670K experiments. To verify this observation, we performed experiments on Amazon-670K with PGHash (not PGHash-D) and



(a) 1 Device Results.



(b) 4 Device Results.

Figure 6.7: **Wiki-325K PGHash-D.** We record model accuracy of a large recommendation network on Wiki-325K trained in a federated setting. PGHash-D matches or outperforms the convergence of SLIDE without requiring LSH to be performed by the central server.

SLIDE (with a SimHash hash function). Table 6.4 displays the SimHash approach for Amazon-670K.

Table 6.4: **PGHash and SLIDE performance on Amazon-670K using SimHash.** Accuracy across the first 5,000 iterations for a single device. Batch size $M = 1024$, $k = 8$, and $c = 8$.

Iteration	SLIDE	PGHash
1,000	10.82%	10.04%
2,000	18.27%	15.99%
3,000	21.83%	19.51%
4,000	23.72%	21.65%
5,000	25.08%	23.38%

As shown in Table 6.4, even with a much larger batch size, SLIDE and PGHash are unable to crack 30% on Amazon-670K. We would like to note that using a smaller batch size (like the $M = 256$ value we use in our Amazon-670K experiments) resulted in an even further drop in accuracy. These empirical results back-up the notion that SimHash is ill-fit for Amazon-670K.

Wiki-325K Performance. In Figure 6.7, we showcase how PGHash-D performs on Wiki-325K. Quite similar to the Amazon-670K results (shown in Figure 6.5), PGHash-D almost exactly matches up with SLIDE. In order to map how well our training progresses, we periodically

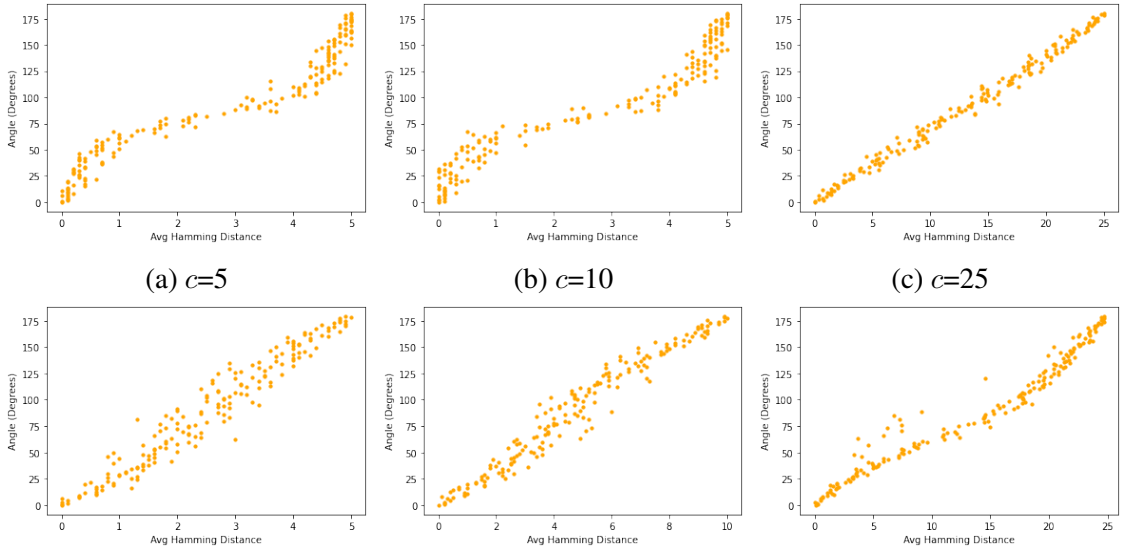
check test accuracies. However, since the test set is very large, determining test accuracies over the entire test set is infeasible due to time constraints on the cluster. Therefore, we determine test accuracies over 30 batches of test data as a substitute as is done in [20, 205].

Matching Full-Training Performance. Along with the failure for SimHash to perform well on Amazon-670K and Wiki-325K, SLIDE and PGHash(-D) are unable to match the performance of full-training on these data-sets. This is observed empirically for Amazon-670K by GResearch in the following article `https://www.gresearch.co.uk/blog/article/implementing-slide/`. We surmise that the failure of SLIDE and PGHash(-D) to match full-training performance on Amazon-670K and Wiki-325K arises due to the small average labels per point in these two data-sets (5.45 and 3.19 respectively). Early on in training, SLIDE and PGHash(-D) do not utilize enough activated neurons. This is detrimental to performance when there are only a few labels per sample, as the neurons corresponding to the true label are rarely selected at the beginning of training (and these final layer weights are tuned much slower). In full-training, the true neurons are always selected and therefore the final layer weights are better adjusted from the beginning. We also note that [222] requires a hidden layer size of 1024 for a distributed version of SLIDE to achieve improved test accuracies for Amazon-670K. Thus, increasing the hidden layer size may have improved our performance (we kept it as 128 to match the original SLIDE paper [20]).

6.7.2 PGHash: angle versus Hamming distance

In this section, we visually explore the degree to which PGHash is a consistent estimator of angular similarity. Specifically, let $x, y \in \mathbb{R}^d$: then we know by Theorem 48 that $\mathcal{H}^{PG}(c, d)$ is

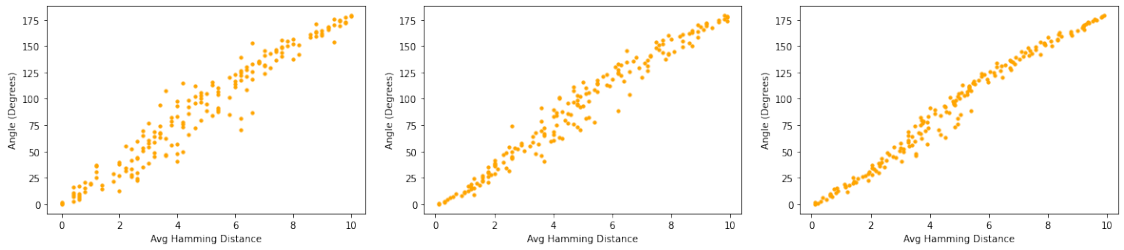
an LSH for $\cos(x_c, y_c)$. We demonstrate that in the unit vector regime, $\theta_c = \arccos(\cos(x_c, Y_c))$ is an acceptable surrogate for $\theta = \arccos(x, Y)$, where $Y = \{y^i\}_{i=1}^N$ and $Y_c = \{y_c^i\}_{i=1}^N$.



(a) c=5 (b) c=10 (c) c=25

(d) c=5 (e) c=10 (f) c=25

Figure 6.8: **Angle/Hamming Distance as a function of sketch dimension.** The average Hamming distance between a PGHashed fixed unit vector $x \in \mathbb{R}^{100}$ and a collection of vectors $y_i \in \mathbb{R}^{100}$ which form different angles with x. Increasing sketch dimension c smooths and reduces the variance of the scatter towards linear correlation. Furthermore, the Hamming scales linearly with c, improving discernibility. (a)-(c) & (d)-(f) are independent series.



(a) 5 tables (b) 10 tables (c) 50 tables

Figure 6.9: **Angle/Hamming Distance as a function of tables** The average Hamming distance between a PGHashed fixed unit vector $x \in \mathbb{R}^{1}00$ and a collection of vectors $y_i \in \mathbb{R}^{100}$ which form different angles with x and fixed sketch dimension $c = 10$. Increasing the number of tables reduces variance.

6.7.3 Additional proofs

Fact 52. *Let x, y, e_1, e_2 be d-dimensional unit vectors such that that the e_i lie on the unit circle contained with the plane spanned by x and y (denoted as $S_{x,y}$) and $e_1 \perp e_2$. Consider the point v on $S_{x,y}$ such that the line through it bisects the angle of the lines passing through x and y. Let $\eta = \arccos(\cos(v, e_1)$. Denote $\theta = \frac{1}{2}\arccos(\cos(x, y))$. Then we may write $x = \cos(\eta + \theta)e_1 + \sin(\eta + \theta)e_2$ and $y = \cos(\eta - \theta)e_1 + \sin(\eta - \theta)e_2$.*

6.7.4 Proof of Theorem 50

Proof. Let $\theta = \frac{1}{2}\arccos(\cos(x, y))$ where $x, y \in \mathbb{S}^{d-1}$ and $A = B^\top B$. The cosine similarity between $x_c = Bx$ and $y_c = By$ (for B correspondent to a (d, c)-folding), is expressible as

$$\cos(x_c, y_c) = \frac{x^\top B^\top By}{(x^\top B^\top Bx)(y^\top B^\top By)} = \frac{x^\top Ay}{\sqrt{(x^\top Ax)(y^\top Ay)}}. \tag{6.1}$$

Consider the SVD $B = UDV^\top$ where U and V are orthogonal and D is $c \times d$ rectangular diagonal matrix. We have then that $A = B^\top B = V\hat{D}^2V^\top$. (Here $\hat{D}$ is now a square diagonal matrix containing squared D_{ii} along the diagonal and 0 everywhere else.) Notice that choice of U nor the ordering of columns v_i of V affects the angle calculation in Equation 6.1. First, we re-order the columns of V so as to order the diagonal entries d_i of D (i.e., the squared singular values) in decreasing order, and as an abuse of notation set $B = \frac{1}{d_1}DV^\top$. Denoting $\hat{\lambda}_i = d_i/d_1$ for $1 \leq i \leq n$, we have that $Bv_i = \hat{\lambda}_i e_i$. (By construction of B we have that $d_i \in \{\frac{d}{c}, 0\}$, therefore, $\hat{\lambda}_i \in \{1, 0\}$)

Consider B acting on S^{d-1}: it scales each dimension by $\hat{\lambda}_i$, thus (as with any linear transformation of a sphere), transforms it into an ellipsoid, with c principal axes determined by the

v_i. The greatest possible distance from the origin to the ellipsoid BS^{d-1} is 1 while the shortest possible distance is 0. Now consider the unit circle $S_{x,y} = \{v \in \text{span}(x, y) \ : ||v|| = 1\}$. We have that $BS_{x,y} \subset BS^{d-1} \cap BU$ is an ellipse (since the intersection of an ellipsoid and plane is always an ellipse).

Choose unit w_1 and w_2 belonging to $S_{x,y}$ such that $w_1 \perp w_2$. by By Fact 52, we may parameterize our vectors as $x = \cos(\eta-\theta)w_1+\sin(\eta-\theta)w_2$ and $y = \sin(\eta+\theta)w_1+\sin(\eta+\theta)w_2$, where η is the angle made with w_1 with the bisector of x and y. By assumption, $||Bw||^2 \geq \alpha$ (the minimal shrinking factor of B on $S_{x,y}$) for some positive α. Denoting $\lambda = \frac{d}{c}$ (the maximal stretching factor of B on $S_{x,y}$), we have that the angle between Bx and By is upper-bounded by

$$f(\eta) = \arctan(\frac{\alpha}{\lambda}\tan(\eta+\theta)) - \arctan(\frac{\alpha}{\lambda}\tan(\eta-\theta)) \tag{6.2}$$

.

The numerator of $\frac{df}{d\eta}$ is $\beta(1-\beta)(1+\beta)\sin(2\theta)\sin(2\eta)$ where $\beta = \alpha/\lambda$. The derivative is trivially 0 if (1) $\beta = 0$, (2) $\beta = 1$, or (3) $\theta = 0$. (1) will not occur as we assume that $S_{x,y}$ does not contain a 0-eigenvector of $A = B^\top B$. (2) can only occur if A is a multiple of the identity matrix (which it is not by construction), and (3) implies that x and y are parallel, in which case their angle will not be distorted. Aside from these pathological cases, the critical points occur at $\eta = 0, \pi/2$. We have then that $\cos(Bx, By)$ lives between $\cos(f(0)) = \frac{1-\beta^2\tan^2\theta}{1+\beta^2\tan^2\theta}$ and $\cos(f(\pi/2)) = -\frac{\tan^2\theta^2-\beta^2}{\tan^2\theta^2+\beta^2}$.

□

Remark. The constant β has an enormous influence on the bounds in Theorem 50. The smaller the α (i.e., shrinking of $||w||$), the greater the bounds on distortion. Although we have imposed constraints on x, y, if we treat them as any possible pair of random unit vectors, then the w in $S_{x,y}$

effectively becomes a random unit vector as well. We can exactly characterize the distribution of $||BX||$ where X denotes a random variable which selects a d-dimensional unit vector uniformly at random.

6.7.4.1 Proof of Proposition 51

Proof. We can sample a d-dimensional vector uniformly at random from the unit sphere S^{d-1} by drawing a d-dimensional Gaussian vector with iid entries and normalizing. Let us represent this as the random variable $X = Z'/||Z'||$ where $Z' \sim \mathcal{N}(0, I_d)$. Consider a (c, d)-folding matrix B, i.e., a d/c horizontal stack of $c \times c$ identity matrices (let us assume $c|d$). We are interested in determining the distribution of $||BX||^2$. For ease of notation, consider the permutation Z of Z' where $Z_i = Z'_{(\lfloor \frac{d/c}{i} \rfloor - 1)*(d/c)+i \pmod{d/c}}$. Since this permutation is representable as an orthogonal matrix P (and multi-variate Gaussians are invariant in distribution under orthogonal transformations), we may instead consider $X := P(Z'/||Z'||)^2 = Z/||Z||^2$. We may write the norm-squared as

$$||BX||^2 = \frac{(Z_1 + \cdots + Z_{d/c})^2}{||Z||^2} + \frac{(Z_{d/c+1} + \cdots + Z_{2d/c})^2}{||Z||^2} + \cdots + \frac{(Z_{(c-1)(d/c)+1} + \cdots + Z_d)^2}{||Z||^2}. \tag{6.3}$$

Consider the first term $\frac{(Z_1+\cdots+Z_{d/c})^2}{||Z||^2}$. First note that for any unit vector u, the distribution of $\frac{(u^\top Z)^2}{||Z||^2}$ does not depend on choice of u. Consider the unit vector u' then which contains $\sqrt{d/c}$ in the first d/c entries and 0 otherwise. Then $\frac{(u'^\top Z)^2}{||Z||^2}$ is equivalent to d/c times our first term. Of course, since $\frac{(e_1^\top Z)^2}{||Z||^2}$ has the same distribution as $\frac{(u'^\top Z)^2}{||Z||^2}$, we have by transitivity that $\frac{Z_1^2}{||Z||^2} \overset{d}{=} (n/q)\frac{(Z_1+\cdots+Z_{d/c})^2}{||Z||^2}$.

By extending the discussion above to the other terms, and by their independence with respect to rotation of Z (since their numerators contain squared sums of mutually disjoint Z

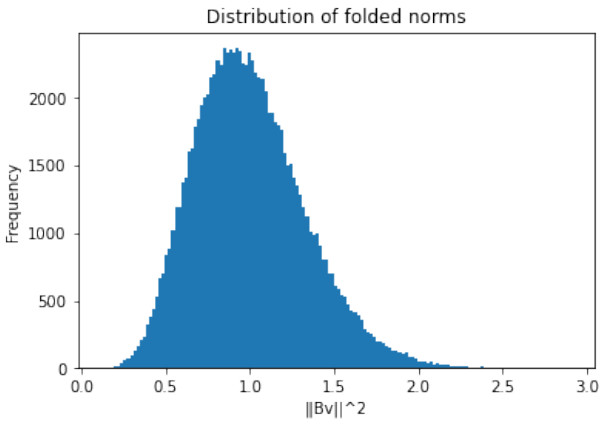


Figure 6.10: **Distribution of folded norms.** 100k randomly drawn unit vectors ($d = 128$) are folded down to length 16 by are usual (d, c)-folding procedure. Depicted is a binned histogram of the norms. As predicted by the statistical description of $||BX||^2$, where X is a randomly drawn unit vector, the mass is centered at 1, i.e., most norms are preserved. Empirically, we observe that the folded norms rarely exceed $\sqrt{128}16$, although the theoretical support is $[0, 8]$: this concurs with the pdf.

coordinates), we have that

$$||BX||^2 \stackrel{d}{=} \frac{d}{c} \cdot \frac{Z_1^2 + Z_{d/c}^2 + Z_{2d/c}^2 + \cdots + Z_d^2}{||Z||^2}. \tag{6.4}$$

The distribution of $\frac{Z_1^2+Z_{d/c}^2+Z_{2d/c}^2+\cdots+Z_d^2}{||Z||^2}$ is well-known to follow a $\mathrm{Beta}(\frac{c}{2}, \frac{d-c}{2})$ distribution [232]. In totality, $||BX||^2 \stackrel{d}{=} \frac{d}{c}\mathrm{Beta}(\frac{c}{2}, \frac{d-c}{2})$. However, we will move to the four parameter description of this scaled Beta distribution which is $\mathrm{Beta}(\frac{c}{2}, \frac{d-c}{2}, 0, \frac{d}{c})$. The pdf and expected value follows by the usual statistical descriptions of this distribution, which can also be found in [232]. □

Figure 6.10 depicts how (d, c)-foldings affect the norms of unit vectors.

6.7.5 Additional theory

In this section, we provide additional theory relevant to SimHash.

We present several well-known results regarding SimHash.

Proposition 53 (SimHash estimation)**.** *Let $x, y \in \mathbb{S}$, i.e., unit d-dimensional vectors. Denote $\theta = \arccos(|\cos(x, y)|)$. Let $v \in S^d$ be a unit vector drawn uniformly at random (according to the Haar measure, for example). Then,*

$$Pr[sgn(v^\top x) \neq sgn(v^\top y)] = \frac{\theta}{\pi}. \tag{6.5}$$

Proof. We reproduce the argument of [225]. We have by symmetry that $Pr[\text{sgn}(v^\top x) \neq \text{sgn}(v^\top y)] = 2Pr[v^\top x > 0, v^\top y < 0]$. The set $\mathcal{U} = \{v \in S^d : v^\top x > 0, v^\top y \leq 0\}$ corresponds to the intersection of two half-spaces whose dihedral angle (i.e., angle between the normals of both spaces) is exactly θ. Intersecting with the d-dimensional unit sphere produces gives a subspace of measure $\frac{\theta}{2\pi}$, therefore, $2Pr[v^\top x > 0, v^\top y < 0] = \frac{\theta}{\pi}$, completing the argument. □

Corollary 54. *Let v instead be a d-dimensional random Gaussian vector with iid entries $\sim \mathcal{N}(0, 1)$. Then for $x, y \in \mathbb{R}^d$,*

$$Pr[\text{sgn}(v^\top x) \neq \text{sgn}(v^\top y)] = \frac{\theta}{\pi} \tag{6.6}$$

Proof. Randomly drawn, normalized Gaussian vectors are well-known to be uniformly distributed on the unit sphere. □

In the setup as above, let the X be a random variable which returns 1 if x and y have differing signs when taking the standard inner product with a randomly drawn Gaussian v. Let $X_1, X_2, \ldots, X_n$ represent a sequence of independent X events. Then,

Proposition 55. $\mathbb{E}[\frac{1}{n}\sum_{i=1}^n X_i] = 1 - \frac{\theta}{\pi}$ *and* $\mathbb{V}[X] = \frac{1}{N}\frac{\theta}{\pi}(1 - \frac{\theta}{\pi})$.

Given that PGHash is equivalent to a SimHash over (d, c)-foldings of R^d, the variance reduction we observe by using multiple tables (Figure 6.9 is explainable by Proposition 55.

Chapter 7: Benchmarking the Robustness of Image Watermarks

7.1 Introduction

Recent and pivotal advancements in text-to-image diffusion models [233, 234, 235] have garnered the attention of the AI community and the general public. Open-source models such as Stable Diffusion and proprietary models such as the Dall·E family and Midjourney have enabled users to produce images that are of human-produced quality. Consequently, there has been a strong push in the AI/ML community to develop reliable algorithms for detecting AI-generated content and determining its source [236]. One avenue for maintaining the provenance of generative content is by embedding *watermarks*. A watermark is a signal encoded onto an image to signify its source or ownership [237, 238, 239, 240, 241, 242]. To avoid degradation of image quality, an invisible watermark is desired. Many such watermarks are robust to common image manipulations [241, 242, 243, 244], and adversarial efforts to remove the watermark are complicated by the difficulty of decoding/extracting the message without private knowledge of the watermarking scheme [240, 241]. Despite this difficulty, various watermark removal schemes can still be effective [244, 245]. However, a lack of standardized evaluations in existing literature (i.e., inconsistent image quality measures, statistical parameters, and types of attacks) has resulted in an incomplete picture of the vulnerabilities and robustness of these algorithms in the real world.

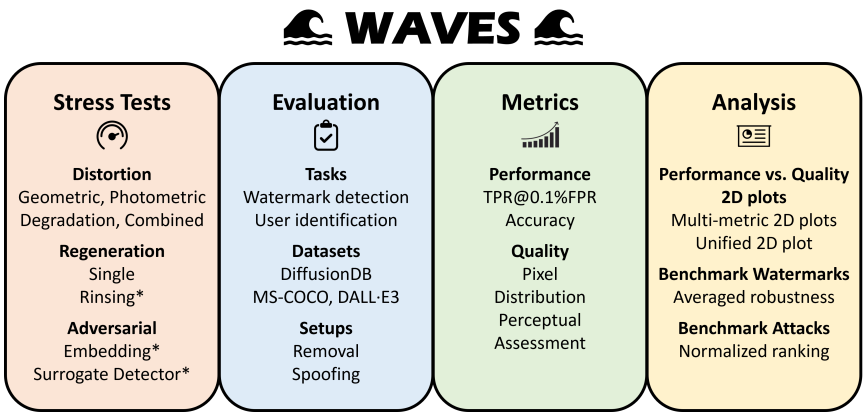


Figure 7.1: **WAVES** establishes a standardized evaluation framework that encompasses a comprehensive suite of stress tests including both existing and newly proposed stronger attacks (denoted by *).

We present WAVES (**W**atermark **A**nalysis **v**ia **E**nhanced **S**tress-testing), a benchmark for assessing watermark robustness, overcoming the limitations of current evaluation methods. WAVES consists of a comprehensive variety of novel & realistic attacks, including classical image distortions, image regeneration, and adversarial attacks. In an effort to stress-test existing/future watermarks, we propose several new attacks such as adversarial embedding attacks, and new variants of existing attacks such as multi-regeneration attacks.

WAVES focuses on the sensitivity and robustness of watermark detection, measured by the true positive rate (TPR) at 0.1% false positive rate (FPR), and in the meantime, studies the severity of image degradations needed to decrease this sensitivity with multiple quality metrics. WAVES develops a series of Performance vs. Quality 2D plots varying over several prominent image similarity metrics, which are then aggregated in a heuristically novel manner to paint an overall picture of watermark robustness and attack potency.

We extensively evaluate the security of three prominent watermarking algorithms, Stable Signature, Tree-Ring, and StegStamp, respectively representing three major techniques for

Table 7.1: **Comparison of robustness evaluations with existing works.** For *categories of attacks*, D denotes distortions, R denotes image regeneration attacks, and A denotes adversarial attacks. *Reference image source* shows whether to use non-watermarked images in the evaluation and what is the source of those images. For the *number of quality metrics*, we use 8 in the paper but test 10 in total. *Joint test* means whether the performance and quality are jointly tested under a range of attack strengths. Our benchmark is the most comprehensive one, with a large scale of attacks, data, metrics, and more realistic evaluation setups.

Research Work	Num. of Attacks	Categories of Attacks	Num. of Datasets	Sample Size per Dataset	Non-watermarked Image Source	Performance Metric	Num. of Quality Metrics	Joint Test
StegaStamp Watermark [240]	5	D	1	1000	-	bit accuracy	3	✗
Stable Signature Watermark [241]	12	D, R	1	5000	-	bit accuracy	3	✗
TreeRing Watermark [242]	6	D	2	1000	generate by same model	TPR@1%FPR	2	✗
Regeneration Attack [244]	10	D, R	2	500	-	bit accuracy	3	✗
Surrogate Model Attack [245]	2	R, A	1	2500	real images	AUROC	0	✗
Adaptive Attack [243]	10	D, A	1	1000	real images	TPR@1%FPR	3	✗
WAVES (ours)	26	D, R, A	3	5000	real images	TPR@0.1%FPR	8 (10)	✓

embedding an invisible signature. WAVES effectively reveals weaknesses in them and discovers previously undetected vulnerabilities. For example, watermarking algorithms using publicly available VAEs can have their watermarks effectively removed with minimal image manipulation. DALL·E3's usage of an open-source KL-VAE underscores the need for unique VAEs in such systems.

Our **contributions** are summarized as follows:

1. In practical scenarios where false alarms incur high costs, our evaluation metric for watermark detection prioritizes the True Positive Rate (TPR) at a stringent False Positive Rate (FPR) threshold, specifically 0.1%. This focus addresses the inadequacies of alternative metrics such as the p-value and Area Under the Receiver Operating Characteristic (AUROC).

2. Additionally, our metric incorporates image quality alongside TPR@0.1% FPR. This inte-

gration acknowledges the necessity of maintaining a balance between reducing the accuracy of watermark detection and the practical utility of the image in practical scenarios.

3. We introduce a comprehensive taxonomy of attacks that encompasses classical distortions (blurring, rotation, cropping, etc.) and powerful, novel variations of regeneration and adversarial attacks, against watermarks.

4. We standardize the evaluation of watermark robustness, allowing us to rank attacks and watermarks. We formalize the watermark *detection* and user *identification* problems and evaluate the robustness under both scenarios.

5. Our benchmark uncovers several especially harmful attacks for popular watermarks, some of which are first introduced in this work, underscoring the need for refinement of existing watermarking algorithms and systems. WAVES contributes as a toolkit to examine the watermark robustness and helps future development of robust watermarks.

7.2 Image Watermarks

We briefly review invisible watermarks and defer detailed discussions to Section 7.6.1. Generally, there are two types of watermarking methods. **(1) Post-processing watermarks** embed watermarks after image generation. *(1a) Frequency-domain methods* like DWT, DCT [246], and DWTDCT [237] modify images in transform domains. *(1b) Deep encoder-decoder methods* such as HiDDeN [238], RivaGAN [239], and StegaStamp [240] use trained neural networks for embedding and decoding watermarks. Post-processing watermarks are model-agnostic but can introduce human-visible artifacts, compromising image quality. **(2) In-processing water-**

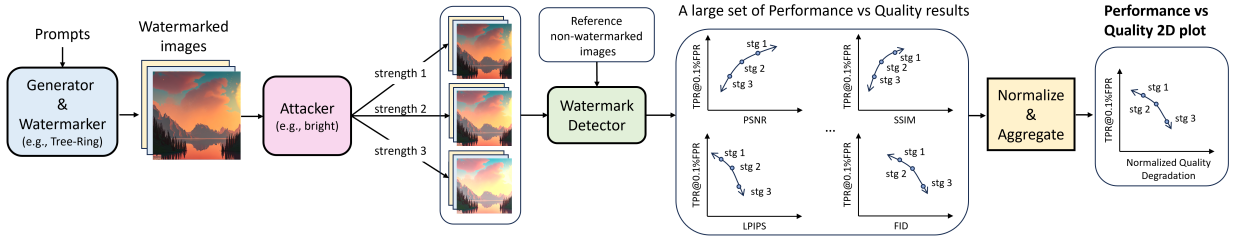
marks integrate watermarking into the image generation process, substantially eliminating visible artifacts. *(2a) Whole model modifications* embed watermarks by training the entire generative models on watermarked images [247, 248, 249]. *(2b) Partial model modifications* such as Stable Signature [241] only fine-tune the decoder of the latent-diffusion model. *(2c) Random seed modification* watermarks like Tree-Ring [242] embed watermarks into the initial noise vector of diffusion models which can be retrieved at detection time.

Robustness is an essential property of watermarks especially since there is an incentive to remove watermarks. Besides natural image distortions, some watermarks are shown to be vulnerable to regeneration through diffusion models or VAEs Zhao et al. [244], Saberi et al. [245], and adversarial attacks Lukas et al. [243], Saberi et al. [245]. However, some unrealistic attacks and inconsistent robustness evaluations across different studies have muddled the understanding of watermark robustness, obscuring the true vulnerabilities of these methods. Therefore, this paper provides a standardized and comprehensive benchmark, encompassing a set of realistic and strong attacks. Our benchmark enables apple-to-apple comparison of watermarks as well as attacks, which helps standardize and accelerate the studies of robust watermarks.

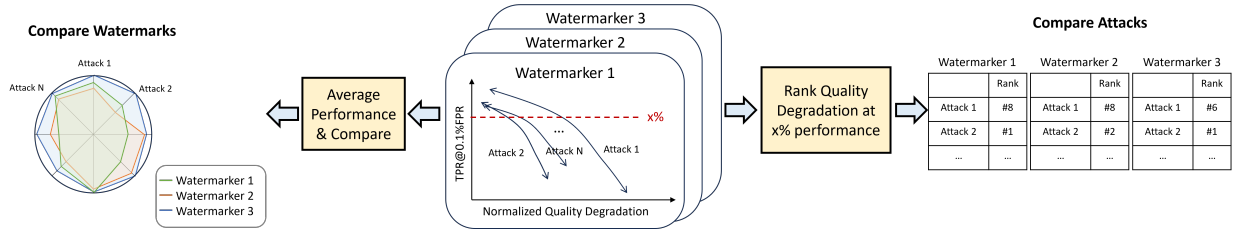
7.3 Standardized Evaluation through WAVES

7.3.1 Standardized Evaluation Workflow and Metrics

As shown in Table 7.1, our benchmark, WAVES, stands out by considering three diverse datasets, incorporating 26 diverse attacks across three categories, and employing 8 quality metrics. These distinguish our work as the most extensive and realistic setup to date for watermark robustness evaluation. For more details on evaluation workflow, setups, metrics, and more anal-



(a) Evaluation of a single attack on a watermarking method. We first attack watermarked images over a variety of strengths (also labeled 'stg'). Then, we evaluate the detection performance (TPR@0.1%FPR) and a collection of image quality metrics such as PSNR and plot a set of performance vs. quality plots. By normalizing and aggregating these quality metrics, we derive a consolidated 2D plot that represents the overall performance vs. quality for the evaluation.



(b) Benchmarking watermarks and attacks. For each watermark, we plot all attacks on a unified performance vs. quality 2D plot to facilitate a detailed comparison. Based on this, we provide two additional analytical perspectives. We compare watermarks' robustness through the averaged performance under different attacks. We evaluate attacks' potency by ranking the quality at a specific performance threshold.

Figure 7.2: Evaluation workflow.

yses, see Section 7.6.5.

Applications and formulation of invisible image watermarks. Invisible image watermarks, originally for protecting creators' intellectual property, have expanded into broader applications like **AI Detection** — identifying AI-generated images [245], and **User Identification** — tracking the source of an image to its creator [241]. We are interested in message-based approaches, where a unique, invisible identifier is embedded into an image. which may be recovered by the content creator at any time to establish provenance. The choice of message varies across methods, with Tree-Ring using random complex Gaussians and others like Stable Signature employing binary strings.

Evaluation Workflow. The trade-off between watermark performance and image quality, especially when watermark attacks lead to image distortions, is critical. We introduce *Perfor-*

Table 7.2: A taxonomy of all the attacks in our stress-testing set. Novel attacks proposed by WAVES are marked with *.

Category	Subcategory (prefix)	Description	Attack Names (suffix)
Distortion	Single (Dist-)	Single distortion	-Rotation, -RCrop, -Erase, -Bright, -Contrast, -Blur, -Noise, -JPEG
	Combination (DistCom-)	Combination of a type of distortions	-Geo, -Photo, -Deg, -All
Regeneration	Single (Regen-)	A single VAE or diffusion regeneration	-Diff, -DiffP[1], -VAE, -KLVAE[2]
	Rinsing* (Rinse-)	A multi-diffusion regeneration	-2xDiff, -4xDiff
Adversarial	Embedding (grey-box)* (AdvEmbG-)[3]	Use the same VAE	-KLVAE8
	Embedding (black-box)* (AdvEmbB-)	Use other encoders	-RN18, -CLIP, -KLVAE16, -SdxlVAE
	Surrogate detector attack* (AdvCLS-)[4]	Train a watermark detector	-UnWM&WM, -Real&WM, -WM1&WM2

[1] DiffP requires user prompts.
[2] KLVAE with bottleneck size 8 is grey-box.
[3] AdvEmbG is grey-box.
[4] AdvCLS needs data and training.

mance vs. Quality 2D plots for a comprehensive comparison, a novel perspective over the typical performance-centric analyses. The evaluation process involves comparing watermarked images with a diverse set of real and AI-generated reference images to produce the performance vs. quality 2D plots, and processing or aggregating the 2D plots to compare attacks and watermarks, as depicted in Figure 7.2.

Performance Metrics in AI Detection and User Identification. WAVES prioritizes fairness and comprehensiveness by using evaluation metrics that are independent of the choice of statistical tests and p-value thresholds, in contrast to some prior practices such as [241]. AI detection in WAVES is akin to binary classification, utilizing ROC curve-based metrics. Given the significant impact of false positives in mislabeling non-watermarked images, strict control over the false positive rate (FPR) is crucial. Therefore, rather than AUROC (since a high AUROC score does not necessarily imply a high true positive rate (TPR) at low FPR levels), WAVES focuses on TPR@$x\%$FPR, specifically at a challenging low FPR threshold of 0.1%, extending recent studies such as [242] with a larger dataset and a more stringent FPR criterion. User identification is approached as multi-class classification, and we measure performance by the accuracy of correct image assignments to users.

Implementing Diverse Image Quality Metrics: Recognizing that no single metric can fully capture the aspects of generated images, we use a range of image quality metrics and propose a normalized, aggregated metric for evaluating watermark and attack methods. WAVES integrates over 8 metrics in 4 categories: *(1)* **Image similarities**, including Peak Signal-to-Noise Ratio (PSNR), Structural Similarity Index (SSIM), and Normalized Mutual Information (NMI), which assess the pixel-wise accuracy after attacks; *(2)* **Distribution distances** such as Frechet Inception Distance (FID) [250] and a variant based on CLIP feature space (CLIP-FID) [251]; *(3)* **Perception-based metrics** like Learned Perceptual Image Patch Similarity (LPIPS) [252]; *(4)* **Image quality assessments** including aesthetics and artifacts scores [253], which quantify the changes in aesthetic and artifact features.

Normalization and Aggregation of Image Quality Metrics: Addressing the distinct characteristics of various image quality metrics, WAVES proposes *a normalized and aggregated quality metric* for a unified measure of image quality degradation and comprehensive scoring of attack or watermark methods. We define the normalized scale for each metric by assigning the 10% quantile value over all attacked images (across 26 attack methods, three watermark methods, and three datasets) as the 0.1 point, and the 90% quantile as the 0.9 point. *Normalized quality metrics are always ranked in ascending order of image degradation.* This normalization ensures equivalent significance across different metrics, defined by their quantiles in a large set of attacked watermarked images. Normalized metrics are aggregated and extensively utilized in Section 7.4 for Performance vs Quality plots, watermark radar plots, and attack leaderboards.

7.3.2 Stress-testing Watermarks

We evaluate the robustness of watermarks with a wide range of attacks detailed in this section and summarized in Table 7.2. Figure 7.24 demonstrates the visual effects.

Distortion Attacks. Watermarked images often face distortions such as compression and cropping during internet transmission, necessitating watermarks that can endure common alterations. However, most studies only test resilience against singular or extreme distortions. In WAVES, we establish the following distortions within an acceptable quality threshold as our baselines. **Geometric distortions**: rotation, resized-crop, and erasing; **Photometric distortions**: adjustments in brightness and contrast; **Degradation distortions**: Gaussian blur, Gaussian noise, and JPEG compression; **Combo distortions**: combinations of geometric, photometric, and degradation distortions, both individually and collectively. Detailed setups for each are provided in the Section 7.7.1.

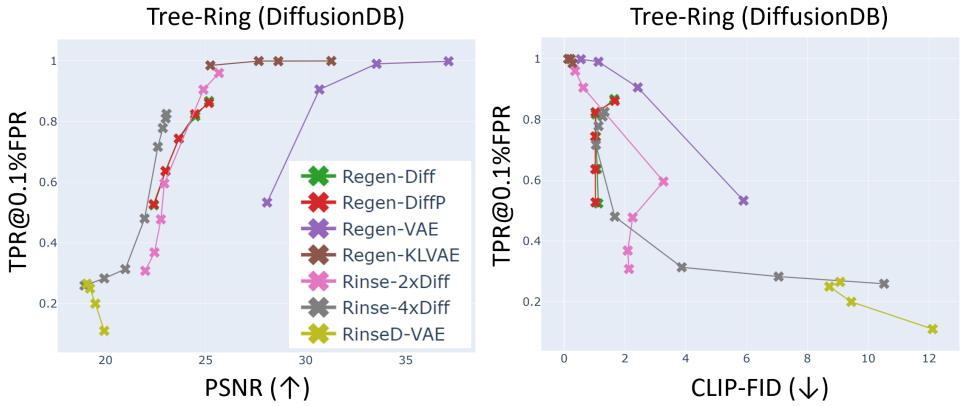


Figure 7.3: Regeneration attacks on Tree-Ringk. Regen-Diff is a single diffusive regeneration and Rinse-[N]xDiff is a rinsing one with N repeated diffusions, with the number of noising steps as attack strength. Regen-VAE uses a pre-trained VAE with quality factor as strength and Regen-KLVAE uses pre-trained KL-VAEs with bottleneck size as strength. RinseD-VAE applies a VAE as a denoiser after Rinse-4xDiff.

Regeneration Attacks, employing diffusion models or VAEs [244, 245], aim at altering

an image's latent representation by noising and then denoising an image. Different from existing works that only perform a **Single regeneration**, we also investigate **Rinsing regenerations**, where an image undergoes multiple cycles of noising and denoising through a pre-trained diffusion model. Furthermore, we introduce two additional variations: prompted regeneration and mixed regeneration (rinse + VAE denoising). To simulate a realistic attack, we use a lower version diffusion model than the one used to generate watermarked images. All such attacks are detailed in Section 7.7.2. As shown in Figure 7.3, in contrast with the conclusions of Zhao et al. [244], the Tree-Ring watermark is not robust against regeneration attacks. In particular, a single regeneration such as Regen-Diff and Regen-VAE can significantly harm the TPR@0.1%FPR while maintaining reasonable CLIP-FID. Rinsing regenerations significantly lower the TPR@0.1%FPR at the cost of markedly decreased image quality. A 2x rinsing regeneration (Regen-2xDiff) strikes a balance between both low-TPR@0.1%FPR and high image quality. In regards to the Stable Signature, Figure 7.7 and Table 7.3 concur with the analysis of Zhao et al. [244] – regeneration attacks are completely destructive and rinsing regenerations reiterate this phenomenon. The StegaStamp is mildly affected by regenerations, and only by diffusive attacks, including our novel rinsing and prompted regenerations.

Adversarial Attacks. Deep neural networks are vulnerable to adversarial examples, [254, 255]. In WAVES, we explore watermark robustness against two types of adversarial attacks.

(A) Embedding Attacks. Watermark detection can be thwarted by perturbations on image embedding. Such attacks have been used against Multimodal Large Language Models like GPT-4V [256] and shown good transferability [257]. We examine if attacks on off-the-shelf embedding models can transfer to watermark detectors. Given an encoder $f : \mathcal{X} \to \mathcal{Z}$ mapping images to latent features, we craft an adversarial image x_{adv} to diverge its embedding from

the original watermarked image x, within an l_∞ perturbation ball limit: $\max_{x_{adv}} \|f(x_{adv}) - f(x)\|_2$, s.t. $\|x_{adv} - x\|_\infty \leq \epsilon$. We approximately solve this using the PGD [258] algorithm (see details in Section 7.7.3.1), and see if the adversarial image transfers to real watermark detectors.

We evaluate five off-the-shelf encoders. **AdvEmbB-RN18** uses a pre-trained ResNet18 [1], targeting the pre-logit feature layer. **AdvEmbB-CLIP** employs CLIP's [259] image encoder. **AdvEmbG-KLVAE8** utilizes the encoder of KL-VAE (f8) which is used in the victim latent diffusion model. This is a grey-box setting but reflects the use of public VAEs in proprietary models (for example, DALLE·3 uses a public KL-VAE according to `https://cdn.openai.com/papers/dall-e-3.pdf`). Further, we do ablation studies on KL-VAE (f16), which has a different architecture but is trained on the same data, and on SDXL-VAE [260], an enhanced version of KL-VAE (f8). They are black-box attacks and are labeled **AdvEmbB-KLVAE16** and **AdvEmbB-SdxlVAE**.

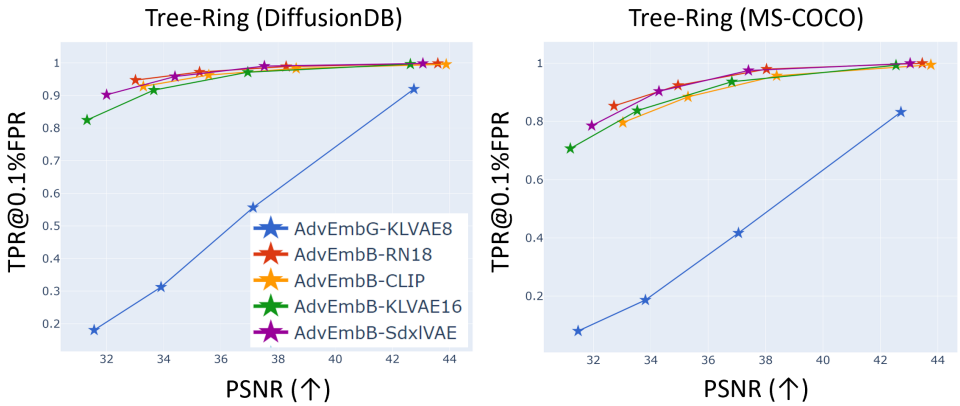


Figure 7.4: Adversarial embedding attacks target Tree-Ring at strengths of {2/255, 4/255, 6/255, 8/255}. Tree-Ring shows vulnerability to embedding attacks, especially when the adversary can access the VAE being used.

As shown in Figure 7.4, Tree-Ring is vulnerable to embedding attacks, particularly under the grey-box condition where TPR@0.1%FPR can drop to nearly zero, effectively removing most watermarks. This is because the detection process of Tree-Ring first maps the image to the latent

representation through the encoder of KL-VAE (f8), then conducts inverse DDIM to retrieve the watermark. The embedding attack changes the latent representation severely; therefore, watermark retrieval becomes very difficult. Using similar yet distinct VAEs, attack effectiveness diminishes but still manages to remove some watermarks, with KL-VAE (f16), trained on the same images, demonstrating the highest transferability. CLIP-based attacks also achieve some success, especially on natural images like MS-COCO, likely due to CLIP being trained on natural images akin to those in MS-COCO, enhancing the transferability. Conversely, Stable Signature and StegaStamp demonstrate robustness against embedding attacks (Figure 7.7), likely because their detectors are trained independently from generative models, differing significantly from standard classifiers and VAEs. Hence, our attacks fail to effectively transfer to their detectors.

(B) Surrogate Detector Attacks. Watermark detection hinges on a detector that decodes and verifies messages from watermarked images. Adversaries might acquire numerous watermarked and non-watermarked images to train a surrogate detector, and transfer attacks on it to the actual watermark detector. Figure 7.5 explores our various settings.

AdvCls-UnWM&WM trains a surrogate detector with both watermarked and non-watermarked images from the victim generative model, as per Saberi et al. [245]. Note that this is an unrealistic setting for proprietary models since all their outputs are assumed to be watermarked. **AdvCls-Real&WM** trains the surrogate watermark detector with watermarked and non-watermarked images, where non-watermarked images are sampled from the ImageNet dataset (not from the generative model). This approach is more applicable to proprietary models. **AdvCls-WM1&WM2** only uses watermarked images. It actually trains a surrogate watermark message classifier to distinguish two users. Suppose the system assigns a particular message to each user for identification purposes, the adversary can collect the training data from two users' outputs, with an identical

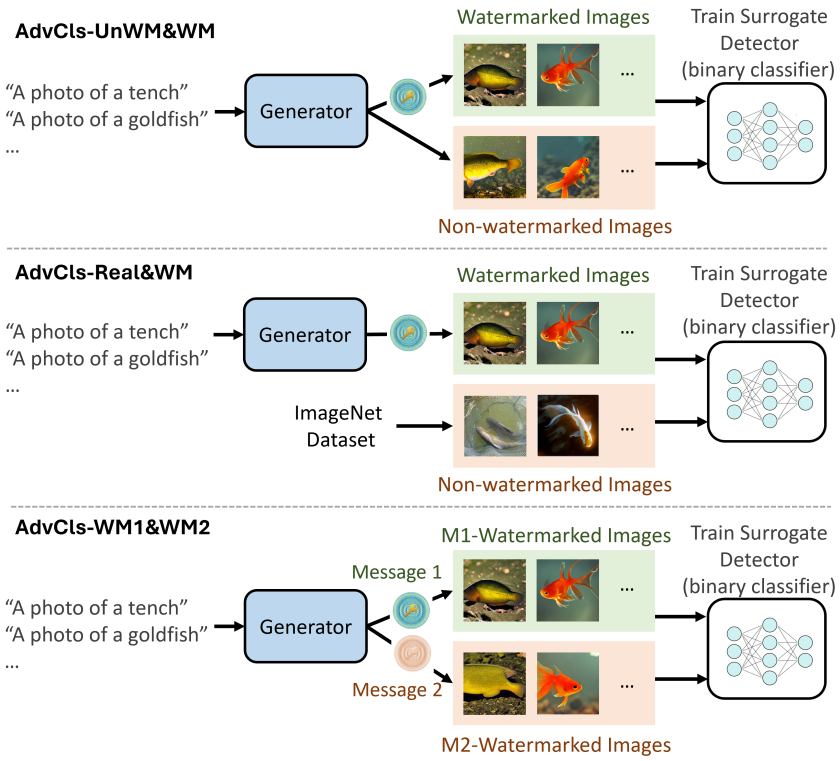


Figure 7.5: Three settings for training the surrogate detector. The Generator is the victim generator under attack. We externalize the watermarking process for simplicity, but it could be in-processing watermarks. After training the surrogate detectors, the adversary performs PGD attacks on them to flip the labels.

set of prompts. Adversarial attacks on this surrogate model aim at user misidentification. All surrogate detectors are fine-tuned on ResNet18. We use ImageNet text prompts "A photo of a $\{$*class name*$\}$" to generate training images (see details in Section 7.7.3.2).

With the trained surrogate detector $f : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{Y} = \{0, 1\}$, adversaries launch targeted attacks. The goal is to craft an adversarial image x_{adv} from an original image x so that f incorrectly predicts the target label y_{target} (i.e., wrong label), minimizing the following with cross-entropy loss: $\min_{x_{adv}} L(f(x_{adv}), y_{target})$, s.t. $\|x_{adv} - x\|_\infty \leq \epsilon$. It enables adversaries to erase watermarks from marked images or implant them into clean images in the first two settings, and to disrupt user identification as well as watermark detection in the third setting. We solve it with the PGD algorithm.

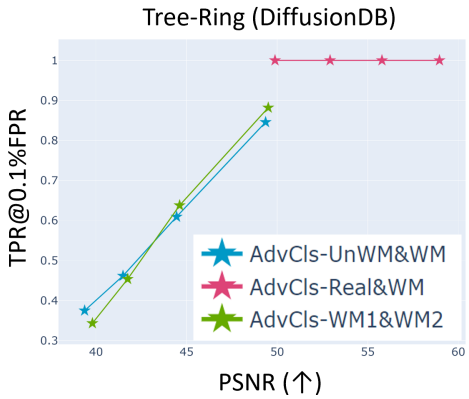


Figure 7.6: Adversarial surrogate detector attacks on Tree-Ring.

Figure 7.6 shows Tree-Ring's vulnerability to surrogate detector-based attacks. In **AdvCls-UnWM&WM**, the adversary accessing non-watermarked images has good transferability and removes watermarks effectively. However, it fails to add watermarks to clean images (spoofing attack), as detailed in Figure 7.20. The reason behind this is explored in Section 7.8.2, where we find the attacker disrupts the entire latent space, not just the watermark (as shown in Figure 7.21).

Conversely, the spoofing attack fails to embed the precise watermark. **AdvCls-Real&WM** attack fails entirely, likely due to the surrogate model appearing to differentiate real from generated images, using broader features than the watermark. The newly proposed **AdvCls-WM1&WM2** successfully attacks Tree-Ring using only watermarked images. Like the first scenario, the surrogate model fails to precisely locate watermarks but learns the mapping to the latent feature space, allowing a PGD attack to remove the watermark by disturbing the entire latent space (see Figure 7.22). In user identification tasks (Figure 7.23), the attack doesn't consistently mislead the detector into misidentifying User1's watermarked images as User2's (targeted misidentification). Instead, imprecise perturbations often lead to incorrect attribution of User1's images to others.

Figure 7.7 shows that Stable Signature and StegaStamp are robust to these attacks. Even with high surrogate classifier accuracy in AdvCls-UnWM&WM, adversarial examples fail to transfer to the true detector, possibly due to reliance on different features than those used by the true detector.

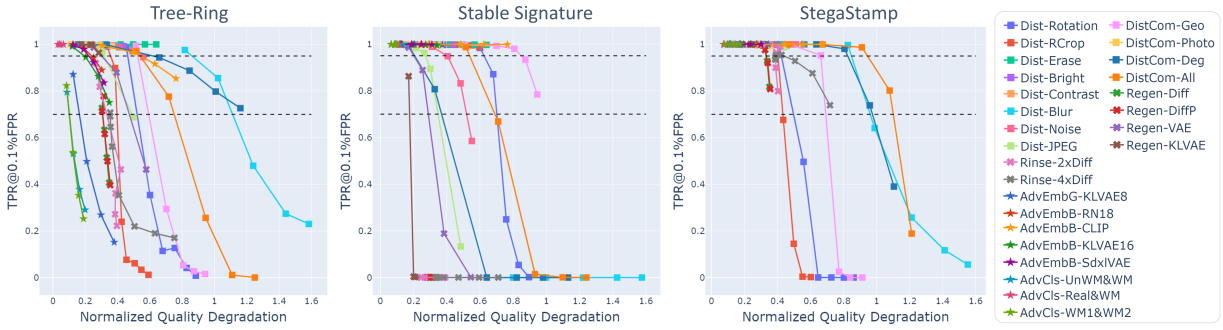


Figure 7.7: Unified performance vs. quality degradation 2D plots under detection setup. We evaluate each watermarking method under various attacks. Two dashed lines show the thresholds used for ranking attacks.

7.4 Benchmarking Results and Analysis

We extensively evaluate the security of three prominent watermarking algorithms (according to Section 7.6.4.2), Stable Signature, Tree-Ring, and StegaStamp, respectively representing three major watermarking types: in-processing via model modification, in-processing via random seed modification, and post-processing. We conduct thorough evaluations with images from DiffusionDB [261], MS-COCO [262], and the DALL·E3 datasets; see Section 7.6.4.1 for details.

Performance vs. Quality 2D plots. We evaluate 3 watermarking methods under 26 attacks, and report results across 3 datasets in Figure 7.25 to Figure 7.30. The quality of images post-attack is evaluated using 8 metrics and the detection performance is measured by TPR@0.1%FPR. Figure 7.13 shows that different quality metrics yield a similar ranking of attacks. Consequently, we aggregate these metrics into a single, unified quality metric — *Normalized Quality Degradation*, with lower scores indicating lesser quality degradation caused by attacks. Furthermore, we aggregate the results across three distinct datasets, and derive the unified Performance vs. Quality degradation 2D plots in Figure 7.7, visualizing the unified evaluation results for each watermarking method against each attack. We defer the aggregation details to Section 7.6.5. Based on these unified 2D plots, we benchmark watermarks and attacks in the following sections.

7.4.1 Benchmarking Watermark Robustness

Figure 7.8 provides a high-level overview of watermarks' robustness. We categorize effective attacks into seven types (same as categories in Table 7.2): *Distortion Single*, *Distortions Combination*, *Regeneration Single*, *Regeneration Rinsing*, *Adv Embedding Grey-box*, *Adv Em-*

bedding Black-box, and *Adv Surrogate Detector*. Attacks considered are detailed in Section 7.6.6. The Average TPR@0.1%FPR, calculated for each category across strength levels, assesses watermarking method robustness. Figure 7.8 shows the robustness of three watermarking methods where the area covered indicates the overall robustness. Figure 7.8 shows the distribution of quality degradation for each type of attack to illustrate the potential trade-off between attack effectiveness and image quality.

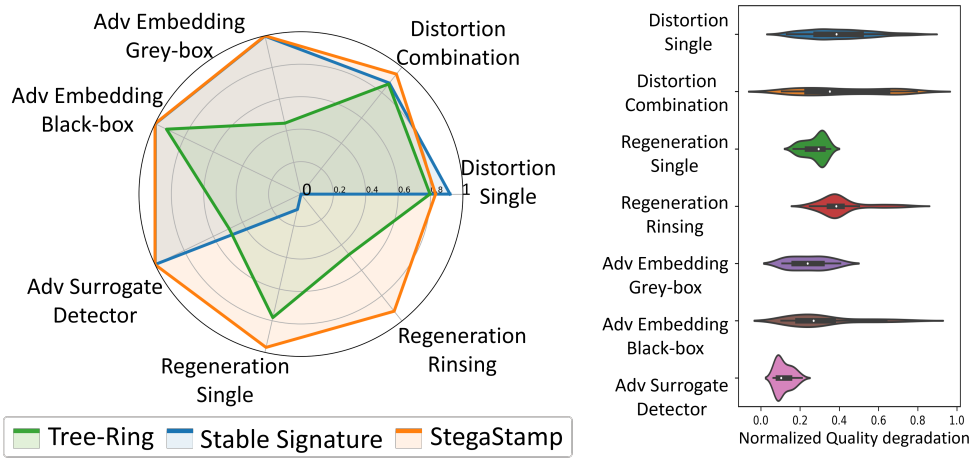


Figure 7.8: (left) Detection performance of three watermarks after attacks, measured by Average TPR@0.1%FPR with lower values (near center) indicating higher vulnerabilities. (right) The distribution of quality degradation. The lower the better.

WAVES provides a clear comparison of watermarks' robustness and reveals undiscovered vulnerabilities. Figure 7.8 reveals that StegaStamp occupies the largest area, signaling its exceptional robustness. Tree-Ring follows suit with a smaller area, and Stable Signature occupies the least space. Interestingly, different watermarking methods exhibit vulnerabilities to different types of attacks. Tree-Ring is particularly vulnerable to adversarial attacks introduced in this paper, with a significant vulnerability to grey-box embedding and surrogate detector attacks. It is also vulnerable to regeneration rinsing attacks. Stable Signature is vulnerable to almost all regeneration attacks. All three watermarks maintain a relative robustness against dis-

tortions. Furthermore, as observed in Figure 7.8, adversarial attacks generally cause less quality degradation, highlighting their potency against Tree-Ring watermarks. WAVES offers an apple-to-apple comparison of watermarks through a multi-dimensional stress test of their robustness, enabling a nuanced and comprehensive understanding of their security in various scenarios.

7.4.2 Benchmarking Attacks

Table 7.3 features a leaderboard ranking attacks based on their impact on detection performance and image quality. We assess attacks using performance thresholds (TPR@0.1%FPR=0.95 and TPR@0.1%FPR=0.7) and quality degradation at these thresholds (Q@0.95P and Q@0.7P). Additionally, we evaluate average performance (Avg P) and quality degradation (Avg Q) across all strengths. These metrics are used to rank 26 attacks for each watermarking method, with details deferred to Section 7.6.7.

Attack effectiveness varies among watermarks. Table 7.3 shows variability in attack efficiency across watermarking methods. Metrics like Q@0.95P and Q@0.7P provide nuanced comparisons, while Avg P and Avg Q offer insights into overall attack potency and image quality impact. Our analysis identifies each watermark's specific weaknesses to certain attacks. For instance, AdvCls-UnWM&WM, AdvCls-WM1&WM2, and AdvEmbG-KLVAE8 are notably effective against Tree-Ring, whereas Regen-Diff and Regen-DiffP are more potent against Stable Signature. Regeneration attacks impact StegaStamp but do not greatly affect its average detection performance; in contrast, certain distortion attacks significantly lower detection performance, at the cost of quality degradation. No single attack excels across all watermarking methods, yet regeneration attacks exhibit some level of consistent effectiveness. This significant variation

in attack effectiveness emphasizes the imperative for diverse and watermark-tailored defensive strategies.

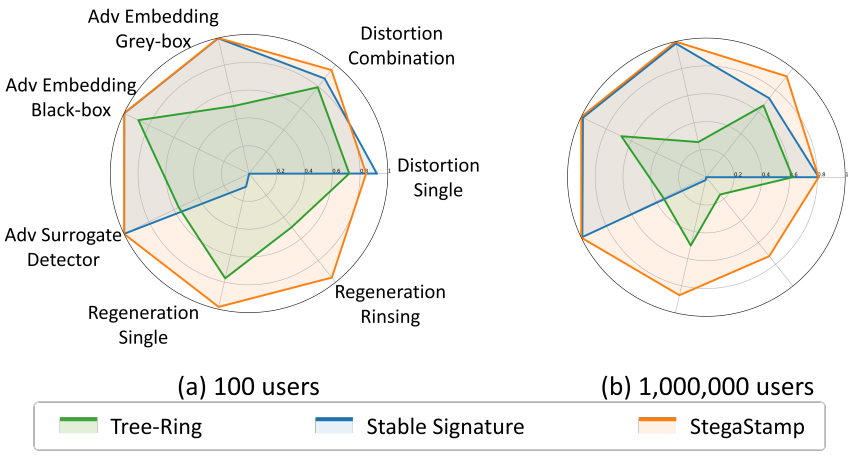


Figure 7.9: Identification accuracy of three watermarks after attacks.

7.4.3 Benchmarking Results for User Identification

We detail the user identification results, following the evaluation method from Section 7.3.1. The key distinction here is the use of identification accuracy as the performance metric. Our study includes scenarios with 100, and 1 million users, reflecting a range of real-world conditions. Utilizing the same evaluation approach, we generate unified Performance vs. Quality degradation 2D plots (Figure 7.19), radar plots for watermark comparison (Figure 7.9), and an attack leaderboard in the identification context (Table 7.5).

Identification results mirror findings from detection, showing similar trends in watermark robustness and attack effectiveness. Figure 7.9 and Table 7.5 reveal that trends in watermark robustness and attack potency closely match those in detection, largely because both rely on precise watermark decoding. Notably, watermarks become more vulnerable as user numbers increase, a trend particularly evident in attacks that already strongly affect detection. Since iden-

tification demands more accurate decoding, its vulnerability amplifies with user growth. Thus, insights gained from detection scenarios generally apply to identification, especially when attacks are not identification-specific. However, novel attacks such as our AdvCls-WM1&WM2, may target user identification. Watermarking strategies should evolve to address emerging challenges in both detection and identification.

7.4.4 Summary of Takeaway Messages

WAVES provides a standardized framework for benchmarking watermark robustness and attack potency. WAVES evaluates both detection and identification tasks. It unifies the quality metrics and assesses attack potency against both performance degradation and quality degradation. The Performance vs. Quality 2D plots allow for a comprehensive analysis of various watermarks in one unified framework. With over twenty attacks tested, WAVES exposes new vulnerabilities in popular watermarking techniques.

Different watermarking methods have different vulnerabilities. Our analysis reveals significant differences in watermark vulnerabilities against attacks. Specifically, Tree-Ring is more vulnerable to adversarial attacks, which generally cause less quality degradation, while Stable Signature is susceptible to most regeneration attacks. This diversity in vulnerabilities highlights the imperative for watermarking methods to identify and strengthen their specific weak areas.

Avoid using publicly available VAEs. WAVES demonstrates the risks of using publicly available VAEs in watermarked diffusion models. An adversarial embedding attack using the same VAE easily compromises Tree-Ring by altering latent features with little visual change.

Stable Signature's design renders it vulnerable to regeneration attacks that use a VAE with an encoder identical to the victim model's VAE encoder, while coupled with a different decoder. Today's proprietary generators, like DALL·3, typically train the latent diffusion model themselves but use a publicly available VAE. This practice, especially with Tree-Ring or Stable Signature watermarking, increases vulnerability, pointing to a critical security concern in those popular AI services.

The robustness of StegaStamp potentially illuminates a path for future robust watermarks. The StegaStamp watermark [240] stands out in our evaluation for its robustness. Designed for physical-world use which requires high robustness, StegaStamp is trained with a series of distortions that mimic real-world scenarios, significantly enhancing its robustness. However, it's important to recognize the potential trade-off between watermark robustness and quality. As a post-processing method, the original paper finds that StegaStamp may introduce artifacts. In contrast, this might not pose a problem for in-processing watermarks. Therefore, in-processing watermarks could still benefit from incorporating augmentation or adversarial training.

7.5 Conclusion

In this work, we comprehensively assess the robustness of several popular watermarking techniques. Our framework, WAVES, uses a battery of attacks to degrade a watermark. The potency of an attack is jointly judged against two metrics: detection/identification probability of the watermark and quality of the image after the attack. We provide a taxonomy, categorization, and ranking of over 20+ styles of attack on the DiffusionDB, MS-COCO, and DALLE·3 datasets. In particular, our normalized ranking of an attack's viability aggregates many popular perceptual

metrics. Our analysis highlights the vulnerabilities of existing watermarking strategies. WAVES can serve as a practical field guide to the design and evaluation of any watermarking algorithm.

7.6 Supplementary Material

7.6.1 A Mini Survey of Image Watermarks

In this section, we detail the existing landscape of watermarking approaches in the era of AI-Generated Content (AIGC) everywhere. Figure 7.10 depicts our scenario of interest. First, an AI company/owner embeds a watermark into its generated images. Then, if the owner is shown one of their watermarked images at a later point in time, they can identify ownership of it by recovering the watermark message. Commonly, users might modify watermarked images for legitimate personal purposes. There are also instances where users attempt to erase a watermark for malicious reasons, such as disseminating fake information or infringing upon copyright. For simplicity, we term any image manipulation as an "attack."

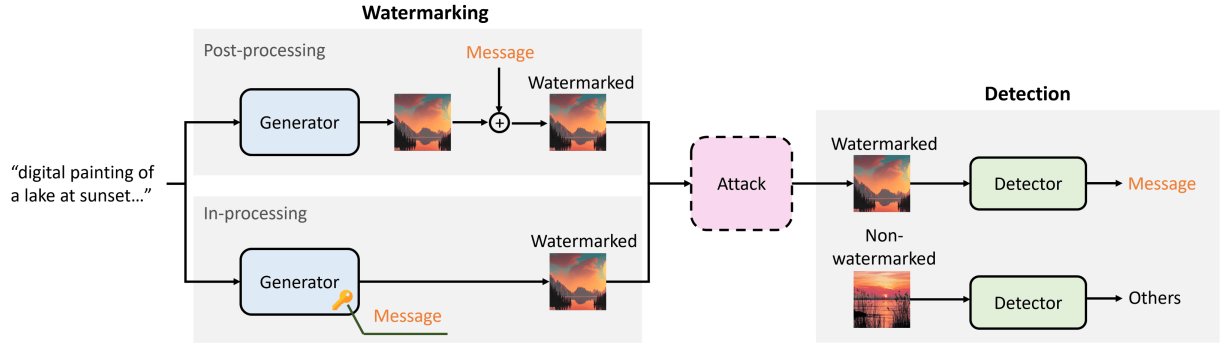


Figure 7.10: **An illustration of a robust watermarking workflow.** An AI company provides two services: (1) generate watermarked images, i.e., embed invisible messages, and (2) detect these messages when shown any of their watermarked images. There is an attack stage between the watermarking and detection stages. The watermarked images may experience natural distortions (e.g., compression, re-scaling) or manipulated by malicious users attempting to remove the watermarks. A robust watermarking method should still be able to detect the original message after an attack.

Watermarking AI-generated Images. Imprinting invisible watermarks into digital images has a long and rich history. From conventional steganography to recent generative model-based methods, we categorize popular watermarking techniques into two categories: post-processing methods and in-processing methods.

Post-processing approaches embed post-hoc watermarks into images. When watermarking AI-generated images, we apply such methods *after* the generation process. Post-processing watermarks are model-agnostic and applicable to any image. However, they sometimes introduce human-visible artifacts, compromising image quality. We review popular post-processing methods.

P1) Frequency-domain methods. These methods manipulate the representation of an image in some transform domain [263, 264, 265]. The image transform can be a Discrete Wavelet Transform (DWT), Discrete Cosine Transform (DCT) [246], or SVD decomposition [266]. These transformations have a range of invariance properties that make them robust to translation and resizing. The commercial implementation of Stable Diffusion [235] uses DWTDCT [237] to watermark its generated images. However, many studies have shown that these watermarks are vulnerable to common image manipulations [244].

P2) Deep encoder-decoder methods. These methods rely on trained networks for embedding and decoding the watermark [267]. Methods such as HiDDeN [238] and RivaGAN [239] learn an encoder to imprint a hidden message inside an image and a decoder (also called a detector) to extract the message. To train robust watermarks, RedMark [268] integrates differentiable attack layers between the encoder and decoder in the end-to-end training process; RivaGAN [239] employs an adversarial network to remove the watermark during training; StegaStamp [240] adds a series of strong image perturbations between the encoder and decoder during training, resulting

in watermarks which are robust to real-world distortions caused by photographing an image as it appears on a display.

P3) Others. There are other varieties of post-processing methods that do not fall into **P1** or **P2**. SSL [269] embeds watermarks in self-supervised-latent spaces by shifting the image's features into a designated region. DeepSigns [270] and DeepMarks [271] embed target watermarks into the probability density functions of weights and activation maps. Entangled watermarks [272] designs a reinforced watermark based on a target watermark and the task data.

In-processing methods adapt generative models to directly embed watermarks as part of the image generation process, substantially reducing or eliminating visible artifacts. With diffusion models presently dominating the field of image generation, a surge of in-processing approaches specific to these models has recently emerged. We categorize current work into three categories.

I1) Model modification. *The entire model.* This line of work inherits the encoder-decoder idea and bakes the encoder into the entire generative model. This is usually accomplished by watermarking training images with a pre-trained watermark encoder and decoder, then training or fine-tuning the generative model on these watermarked images [247, 248, 249]. This type of method has been shown to work well on small models like guided diffusion, but suffers from the expensive training of large text-to-image generation models [273], making it inapplicable in practice.

Parts of the model. Stable Signature [241] follows the above two-stage training pipeline while only fine-tuning the decoder of the latent-diffusion model (LDM) [235], leaving the diffusion component unchanged. This type of watermarker is much more efficient to train. By fine-tuning multiple latent decoders, the model can embed different messages into images.

The robustness of these two types of model modification critically relies on the robustness of the pre-trained encoder and decoder.

I2) Modification of a random seed. Tree-Ring [242], different from all the above methods, embeds a pattern into the initial noise vector used by a diffusion model for sampling. The pattern can be retrieved at detection time by inverting the diffusion process using DDIM [274] as the sampler. This method does not require any training, can easily embed different watermarks, and is robust to many simple distortions and attacks. The robustness of Tree-Ring relies on the accuracy of the DDIM inversion.

Removing Watermarks. Robustness is an essential property of watermarks. Evaluations of robustness in existing literature focus on simple image distortions like rotation, Gaussian blur, etc. Recently, inspired by adversarial purification [275], Zhao et al. [244] and Saberi et al. [245] both find that regenerating images by noising and denoising images through a diffusion model or a VAE can effectively remove some watermarks. Saberi et al. [245] propose adversarial attacks based on a trained surrogate watermark detector. Lukas et al. [243] also introduces adversarial attacks but requires the knowledge of the watermarking algorithm and a similar surrogate generative model. Jiang et al. [276] studies white-box attacks and black-box query-based attacks. Some attacks are not possible in realistic scenarios where the attacker has only API access. Furthermore, existing evaluations use differing quality/performance metrics, making it difficult to compare the effectiveness between watermarking methods and between attacks.

Benchmarks for Image Watermarks. Before the advent of AIGC, there were significant benchmarks introduced that greatly accelerated the progress of watermark standardization [277, 278, 279]. However, with the development of AIGC, the need to watermark images generated by AI

has become urgent, as previous methods were weak in robustness and could not meet current requirements. Nowadays, more and more methods for watermarking images generated by AI have been proposed, but they all use different methods to evaluate robustness. Therefore, this paper proposes a benchmark for the AIGC era.

7.6.2 Formalism of Watermark Detection and Identification

Invisible image watermarks, which are inspired by classical watermarks to protect the intellectual properties of creators, are now applied for a wider range of application scenarios. With the vast development of AI generative models, most current research focuses on applying invisible watermarks to (1) identify AI-generated images (AI Detection) [245], and (2) identify the user who generated the image for source tracking (User Identification) [241].

To fairly evaluate the different watermark methods for different applications, we start from formulating a general, message-based watermarking protocol, partially adopting the notation of [243], which generalizes most of the existing setups. Let θ_G denote an image generator, $\mathcal{M}$ the space of watermark messages, and $\mathcal{X}$ the domain of images. We assume $\mathcal{M}$ is a metric space with distance function $D(\cdot, \cdot)$. The choice of message space $\mathcal{M}$ can be very different depending on the watermarking algorithm: for Tree-Ring, messages are random complex Gaussians, while for the Stable Signature and StegaStamp, each message is a length-d binary string, where d denotes the length of the message. For watermarking algorithms following the encoder-decoder training approach, like Stable Signature and StegaStamp, the choice of message length d is fixed after training. Some methods, such as Tree-Ring, enjoy flexible message length at the time of injecting watermarks.

In addition to classifying images as watermarked or non-watermarked, a good detector will often provide a *p-value* for the watermark detection, which measures the probability that the level of watermark strength observed in an image could occur by random chance. The Tree-Ring watermark also includes an image location parameter τ to embed a message $m \in \mathcal{M}$, but we subsume this under the parameters of θ_G. We now introduce several important watermarking operations:

- $\mathsf{EMBED} : \theta_G \times \mathcal{M} \to \mathcal{X}$ is the generative procedure that creates a watermarked image given user-defined parameters of θ_G (such as prompt, guidance scale, etc. for a diffusion model) and a target message $m \in \mathcal{M}$.

- $\mathsf{DECODE} : \mathcal{X} \to \mathcal{M}$ is a recovery procedure of a message m embedded within a watermarked image $x = \mathrm{EMBED}(\theta_G, m)$. In particular, the recovery $m' = \mathrm{DECODE}(x)$ may be imperfect, i.e., $m' \neq m$.

- $\mathsf{VERIFY}_\alpha : \mathcal{M} \times \mathcal{M} \to \{0, 1\}$ is conducted by the model owner to decide whether x was watermarked by inspecting $m' = \mathsf{DECODE}(x)$, where $x = \mathsf{EMBED}(\theta_G, m)$. For a decoded message m', we consider the following p-value (further discussed in Section 7.6.3) for evaluating whether the image could have been watermarked using m. which is defined as

$$p = \mathrm{P}_m\big(D(\omega, m') < D(m, m') \mid H_0\big),$$

where, $D(\omega, m')$ is the similarity between an arbitrary message $\omega \sim \mathcal{M}$ (drawn uniformly at random) and m', and $D(m, m')$ is the similarity between the ground truth message m and the recovered message m'. H_0 denotes the null hypothesis that the image was generated

without knowledge of the watermark (and therefore the recovered message is random). $\mathsf{VERIFY}_\alpha(m', m)$ returns 1 if $p < \alpha$, and 0 otherwise. In our experiments, we set $\alpha = 0.001$.

To establish a comprehensive evaluation toolbox, we consider two distinct problems that naturally arise during watermark analysis: detection and identification. Let $\mathcal{A} : \mathcal{X} \to \mathcal{X}$ represent an image attack function and denote by Q a fixed subset of messages independently drawn from $\mathcal{M}$ used by θ_G. Further, assume that the owner of θ_G will only embed messages contained within a finite subset Q drawn randomly from $\mathcal{M}$.

7.6.2.1 Detection

In the *watermark detection problem*, given $x = \text{EMBED}(\theta_G, m)$, and an attack $x' = \mathcal{A}(x)$, the model owner is tasked with producing EMBED and DECODE protocols which satisfy the following,

(1) If $x = \mathsf{EMBED}(\theta_G, m)$ is a watermarked image, then $\mathsf{VERIFY}_\alpha(\mathsf{DECODE}(x')) = 1$. *(2)* If $x = \mathsf{EMBED}(\theta_G, \mathsf{NULL})$ is an unwatermarked image, then $\text{VERIFY}_\alpha(\mathsf{DECODE}(x')) = 0$.

For both conditions, a comparison of the extracted message $m' = \mathsf{DECODE}(x)$ is performed against all messages in Q. Failure of the above conditions is referred to as Type II and Type I errors, respectively. Exploration of the tradeoff between minimization of both error types is an interesting research topic in its own right [244, 245].

7.6.2.2 Identification

While watermark detection requires only that $\mathsf{VERIFY}(\theta_G, x') = 1$, the *watermark identification problem* further requires that one can accurately determine which message from Q is embedded in the image. Rigorously, given $x = \mathsf{EMBED}(\theta_G, m)$, an attack $x' = \mathcal{A}(x)$, and $m' = \mathsf{DECODE}(\theta_G, x')$, the user requires the EMBED and DECODE to satisfy

$$\arg\min_{m' \in Q} \ \mathrm{P}\big(D(\omega, m) < D(m', m) \mid H_0\big) = m,$$

for randomly drawn $\omega \sim \mathcal{M}$ if x.

The identification problem is useful in the scenario where the model owner wishes to identify the user who created an image (e.g., a user of DALL·E). Note that as $|Q| \to \infty$, the identification problem becomes difficult as Q will resemble $\mathcal{M}$ in distribution.

7.6.3 Details on Performance Metrics

7.6.3.1 Clarifications on p-Value

Here, we clarify the definition of the p-value as follows. Watermark injection and evaluation are often done by encoding a message m into the image, and later recovering the message m', which may be an imperfect recovery. In addition to classifying images as watermarked or non-watermarked, a good detector will often provide a *p-value* for the watermark detection, which measures the probability that the level of watermark strength observed in an image could happen

by random chance. Rigorously, we have

$$p = \mathrm{P}_m\big(D(\omega, m') < D(m, m') \mid H_0\big),$$

where $D(\omega, m')$ is a dissimilarity metric between an arbitrary message $\omega \sim \mathcal{M}$ (selected uniformly at random) and recovered message m' from the image by the detector, and $D(m, m')$ denotes dissimilarity between the ground truth message m and the recovered message m'. H_0 denotes the null hypothesis that the image was generated without knowledge of the watermark (and therefore, the recovered message is random). The same hypothesis testing can also be applied to user identification.

As in some prior work [241], one may set a threshold on the estimated p-value to determine the detection result. However, this approach makes it difficult to compare different watermark methods fairly. Even if we set the same p-value threshold on all watermark methods, the distinct choice of message space $\mathcal{M}$, message distribution P_m, and hypothesis test may differ. Therefore, we seek to evaluate watermark methods mainly using metrics that are independent of the choice of p-value threshold and statistical test.

7.6.3.2 Performance Metrics for User Identification

For user identification, we also focus on metrics that do not depend on statistical testing and hyperparameters like p-value thresholds.

The user detection issue involving K users is aptly conceptualized as a K-way classification task. This can be reframed into a binary classification problem by designating the positive class as the correct user and the negative class as all other users. From this perspective, the TPR@x%FPR metric becomes applicable, defined for a specific FPR threshold and user count.

In our study, we focus on TPR@0.1%FPR for a scenario involving 1,000 users. The identification performance results are shown in Section 7.4.3.

7.6.3.3 Other Performance Metrics

While this paper primarily focuses on the TPR@0.1%FPR metric, it's important to acknowledge other common metrics such as p-values, AUROC scores, mean accuracies, and bit accuracies.

However, we do not report p-values since their absolute values depend heavily on the chosen statistical test, making them less comparable across different watermark methods.

AUROC scores, although independent of the choice of p-value threshold and statistical test, have limitations used as a metric for evaluating watermark detection. In AI-generated image applications, labeling non-watermarked images as watermarked (false positive) are particularly detrimental. As a result, strict control of false positive rate (FPR) is crucial. However, a high AUROC does not guarantee a high true positive rate (TPR) at low false positive rate (FPR) levels.

Using message distances such as bit accuracy as a metric for evaluating watermarks' performance has several limitations:

(1) Insensitivity to error distribution: bit accuracy measures the proportion of correctly identified bits in the watermark but does not account for the distribution of errors. This means it treats all errors equally, regardless of their impact or pattern. In watermarking, certain types of errors (like clustered errors) might be more detrimental than others.

(2) Lack of contextual insight: bit accuracy alone doesn't provide insights into the types of errors (false positives or false negatives). In watermark detection, understanding the nature of errors

is crucial, especially in differentiating between missing a watermark and incorrectly identifying one.

(3) Threshold dependency: the effectiveness of bit accuracy is dependent on the threshold chosen for determining a bit's value. Different thresholds can yield significantly different bit accuracies, making the metric somewhat arbitrary and less reliable for comparing different watermarking schemes.

(4) Non-representation of overall system performance: bit accuracy focuses narrowly on the correctness of individual bits, neglecting the broader context of the watermarking system's performance, such as its robustness against attacks, computational efficiency, or impact on image quality.

(5) Potential misleading results in imbalanced cases: in scenarios where the watermark bits are not evenly distributed (e.g., more 0s than 1s or vice versa), bit accuracy might give a skewed view of the system's performance. It could show high accuracy even if the system is only good at detecting the majority class. For these reasons, it's often more effective to use a combination of metrics that can provide a holistic view of the watermarking system's performance, considering aspects like error distribution, false positives/negatives, and overall impact on the media.

Although these metrics are not included in the paper, they are incorporated in the benchmark software and available for future research use.

7.6.4 Design Choices of WAVES

7.6.4.1 Dataset Preparation

We utilize three datasets for the non-watermarked reference images in our evaluation: **DiffusionDB**, **MS-COCO**, and **DALL·E3**, each comprising 5000 reference images and prompts. **DiffusionDB** represents a diverse collection from the DiffusionDB dataset [261], focusing on images generated from the Stable Diffusion [235] models. **MS-COCO** is derived from the well-known Microsoft COCO detection challenge [262], featuring a wide range of everyday scenes and objects. **DALL·E3**[1] includes images from the DALL·E3 model, showcasing another popular diffusion model trained on substantially different data. These datasets provide a comprehensive range of image types and contexts, ideal for robust watermark evaluation.

The three datasets are filtered subsets of the corresponding source dataset using the same filtering algorithm. The source dataset information is listed below.

- *DiffusionDB*: the 2m_random_100k split of DiffusionDB dataset [261], link.
- *MS-COCO*: the validation split of the 2017 Microsoft COCO detection challenge [262], link.
- *DALL·E3*: the train split of the *dalle-3-dataset* repository on HuggingFace, collected from the LAION share-dalle-3 discord channel, link.

The filtering algorithm considers the following rules to subsample the 5,000 image subset:

[1]The DALL·E3 dataset is hosted at `https://huggingface.co/datasets/laion/dalle-3-dataset`.

- *Remove columns*: Remove irrelevant columns and only keep the reference images and prompt strings.

- *Filter prompts*: Tokenize the prompt strings by the Open Clip's tokenizer, and filter out samples with no tokens and more than 75 tokens. This is because Stable Diffusion [235] truncates prompts at 75 tokens [261].

- *Rank images*: Rank the images by their aesthetics score, as defined by [253], in descending order. We then select the top 5,000 images, along with their corresponding prompt strings. This approach is adopted because the DiffusionDB and DALL·E3 datasets, sourced from chat-bots, contain some lower-quality images. We posit that watermarking holds greater utility for high-quality AI-generated images, as the copyright protection of low-quality generated images is less meaningful and practical.

In our study, we examined three distinct datasets—DiffusionDB, MS-COCO, and DALL·E3—each characterized by a unique distribution of prompt words. As illustrated in the word-cloud plots (Figure 7.11), we observe notable differences. DiffusionDB predominantly features prompt words that emphasize the desired quality of the generated images, such as "beautiful" and "highly detailed." In contrast, MS-COCO's prompts mainly focus on describing the objects within the images. Meanwhile, DALL·E3's prompts show a tendency towards describing aspects of fine arts.

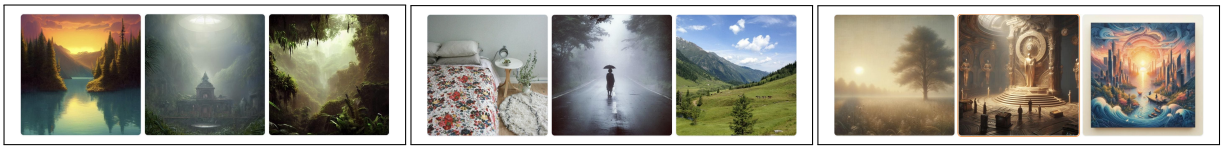
Image examples from the three datasets are illustrated in Figure 7.12. The reference images for DiffusionDB are produced by Stable Diffusion, MS-COCO includes real-world photographs, and DALL·E3 contains images generated by the DALL·E3 model. This choice of datasets effectively covers two popular generative models and the real-world scenario, highlighting their



(a) DiffusionDB prompts (b) MS-COCO prompts (c) DALL·E3 prompts

Figure 7.11: Word clouds of DiffusionDB, MS-COCO, and DALL·E3 prompts.

relevance in practical watermarking applications.



(a) DiffusionDB (b) MS-COCO (c) DALL·E3

Figure 7.12: Image examples of DiffusionDB, MS-COCO, and DALL·E3.

7.6.4.2 Selection of Watermark Representatives

Our WAVES framework can be used to stress-test the robustness of any watermark. In this work, however, we focus on three methods: the *Stable Signature*, *Tree-Ring*, and *Stegastamp*. This is due to existing and extensive studies [242, 244, 245] indicating these three methods are far more robust to simple off-the-shelf attacks than alternative watermarking algorithms listed in Section 7.6.1. We list these competitors along with their documented vulnerabilities in Table 7.4.

7.6.5 Evaluation Details

In this section, we provide more details on the evaluation scheme of WAVES.

7.6.5.1 Watermarking Protocol and Evaluation Workflow.

In-depth information on the applications of invisible image watermarks is provided, focusing on AI detection and user identification. We delve into the evolution of watermarks from classical copyright protection tools to their modern uses in AI scenarios. Section 7.6 discusses the specific roles of AI detection in distinguishing AI-created images and user identification in tracing image origins, citing studies like [241, 245].

The formulation of our watermarking protocol is detailed, explaining the use of an image generator θ_G, a metric space of watermark messages $\mathcal{M}$, and an image domain $\mathcal{X}$. We elaborate on the variations in the choice of message space $\mathcal{M}$ across different watermark methods. For example, Tree-Ring uses random complex Gaussians, whereas Stable Signature and StegaStamp use binary strings. The implications of these choices on the flexibility and effectiveness of watermark methods are discussed.

An extensive analysis of the trade-off between watermark performance and image quality in the context of watermark attacks is provided. This includes the rationale for using Performance vs. Quality 2D plots for attack comparisons, highlighting the comprehensive perspective this offers over traditional performance-focused analyses. The methodology of our evaluation process is laid out in detail, describing how we compare watermarked images from model θ_G with a mixed set of real and AI-generated images to achieve a robust and unbiased assessment. This section also covers the specific metrics used, including TPR@0.1%FPR and various image quality metrics, and how they are integrated into a consolidated performance vs. quality analysis.

7.6.5.2 Performance Evaluation Metrics

The evaluation approach in WAVES addresses the challenges of using p-values for fair watermark method comparison. The diversity in message spaces $\mathcal{M}$, distributions P_m, and hypothesis tests can lead to biased results when traditional p-value thresholds are used. Our metrics, designed to be independent of these thresholds and tests, offer a balanced and thorough evaluation of watermark methods, focusing on their inherent strengths in encoding and recovering messages.

Emphasizing TPR@$x\%$FPR, particularly at the low FPR of 0.1%, sets WAVES apart in evaluating watermark methods. This novel approach, inspired by studies like Fernandez et al. [241], Wen et al. [242], challenges watermark methods beyond typical benchmarks such as TPR@1%FPR. Applied to a broader image dataset, it provides a more comprehensive evaluation of their effectiveness. In user identification, WAVES's multi-class classification approach assesses watermark methods' efficacy in correctly attributing users. The appendices detail the methodology's implementation and present additional results, demonstrating the effectiveness and accuracy of our approach in various user identification scenarios.

We treat the user identification problem as a multi-class classification task, as outlined in Section 7.3.1. This involves defining a set of ground-truth messages, each corresponding to a unique user. To avoid the exhaustive evaluation process (watermark encoding, attacking, and decoding) for varying numbers of users, we consistently watermark images with the same message, the ground-truth message of the first user, and generate a random set of ground-truth messages for the remaining users at the time of evaluation. This approach is feasible since the ground-truth messages for users other than the first do not influence the watermarking or attack phases. We conduct the identification assessment ten times with ten distinct random sets of ground-truth

messages for the other users, and we report the mean multi-class classification accuracy.

7.6.5.3 Processing Results

A set of Performance vs. Quality 2D plots show the detailed evaluation results. We evaluate 3 watermarking methods under the 26 attacks, and report results across 3 datasets in Figure 7.25 to Figure 7.30. The quality of images post-attack is evaluated using 8 metrics and the detection performance of 3 methods is measured by TPR@0.1%FPR.

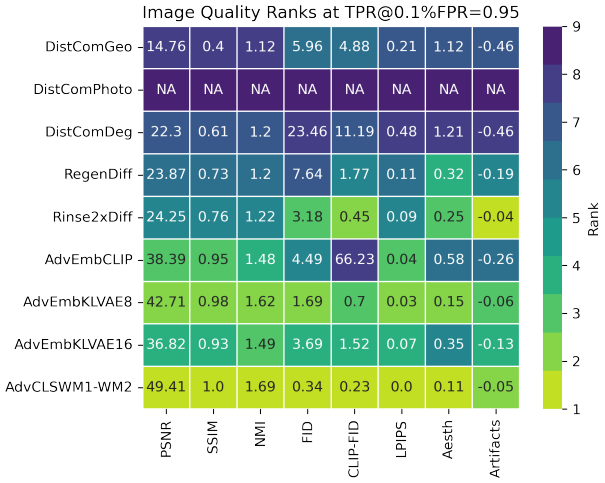


Figure 7.13: **Ranking attacks with different quality metrics** on DiffusionDB images watermarked by Tree-Ring. Attack potency is ranked by image quality at 0.95 TPR@0.1%FPR. Colors indicate the ranks (1=best, 9=worst), and values show the measured quality. We use 'NA' to label an attack if its attack curve lies entirely above TPR=0.95; the attack is automatically ranked last.

Different quality metrics yield similar ranking of attacks. Despite measuring different aspects of image quality, we observe that eight quality metrics consistently produce similar rankings for attacks, as illustrated in Figure 7.13. Since a strong attack should remove the watermark without sacrificing the image quality, we rank attack potency by ranking the post-attack quality, from best to worst, at a frozen performance threshold (e.g., TPR@0.1%FPR=0.95). Upon comparing the rankings derived from different quality metrics, we find that the variations in rank

order are minimal. Consequently, we aggregate these metrics into a single, unified quality metric.

Unified Performance vs. Quality degradation 2D plots. We first set the "standardized" 0.1 and 0.9 points for each metric according to the distribution of measured values (as depicted in Figure 7.14). Subsequently, every metric's value is normalized to predominantly fall within the $[0.1, 0.9]$ range of the normalized quality metric (the detailed methodology is provided in Section 7.6.5.4). We average these normalized quality scores to derive the *Normalized Quality Degradation*, with lower scores indicating lesser quality degradation caused by attacks, which is preferred. Furthermore, we aggregate the results across three distinct datasets. The Performance vs. Quality degradation 2D plots, as shown in Figure 7.7, visualize the unified evaluation results for each watermarking method. We use unified Performance vs. Quality degradation 2D plots to benchmark watermarks and attacks in the following sections.

7.6.5.4 Normalization and Aggregation of Quality Metrics

The eight quality metrics in WAVES exhibit unique range characteristics. To synthesize these into a single metric, we normalize each metric into a common interval, assigning the 10% quantile of all attacked images as the 0.1 point, and the 90% quantile as the 0.9 point. This normalization is based on a comprehensive dataset covering 26 attack methods, three watermark methods, and three datasets. Our focus is on specific applications, particularly attacking invisible image watermarks. The normalization process is informed by the cumulative distribution functions (CDFs) of these metrics, which exhibit a roughly linear distribution between the 10% and 90% quantiles, but a non-linear pattern outside this range. This observation is particularly evident in metrics like PSNR. The normalization method ensures values carry equivalent signif-

icance across different metrics. Figure 7.14 in this section provides a visual representation of the CDFs across all metrics. After normalization, metrics are aggregated by averaging to form the comprehensive quality metric, utilized in Section 7.4 for Performance vs Quality plots, watermark radar plots, and attack leaderboards. This section elaborates on the normalization and aggregation process, providing a foundation for understanding the metric's application and significance.

In Figure 7.14, the cumulative distribution functions (CDFs) for eight image quality metrics over all attacked watermarked images are presented. This illustration includes the metric values at the 10% and 90% quantiles, which are used as the boundaries for normalizing the metric values within the range of $[0.1, 0.9]$. Such normalization ensures that all normalized metrics exhibit a comparable statistical distribution over attacked watermarked images, facilitating an unbiased aggregated evaluation. To consolidate these normalized metrics, we first calculate the average within each of the four defined categories (image similarities, distribution distances, perception-based metrics, and image quality assessments) as delineated in Section 7.3.1. Subsequently, the average of these category averages is calculated to yield a single, consolidated normalized, and aggregated quality metric.

7.6.6 Details of Benchmarking Watermarks

When benchmarking watermark robustness in Figure 7.8 and Figure 7.9, we consider the following effective attacks. We select 21 attacks from 26 attacks. We include all distortion attacks. We select the two most effective single regeneration attacks and two rinsing attacks. For adversarial attacks, we do not include AdvEmbB-RN18, and AdvCls-Real&WM since they basi-

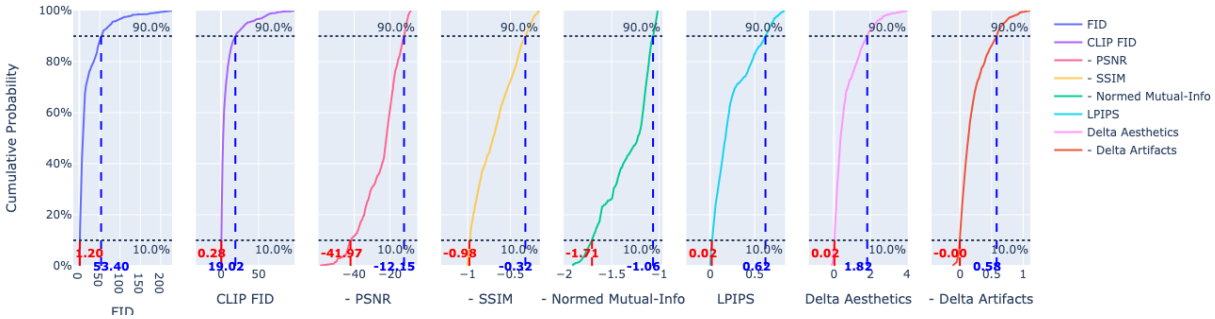


Figure 7.14: Cumulative distribution functions (CDFs) for eight image quality metrics across all attacked watermarked images. The horizontal dashed lines mark the 10% and 90% quantiles, and the intersecting vertical dashed lines delineate the bounds of the normalization intervals. Values at the lower bound are normalized to 0.1, and those at the upper bound to 0.9.

cally do not work. We also eliminate AdvCls-UnWM&WM and only use AdvCls-WM1&WM2 to represent surrogate detector attacks since AdvCls-UnWM&WM is based on an unrealistic assumption. For each type of attack, we compute Average TPR@0.1%FPR across all practical strength levels that cause quality degradation less than 0.8, and across all attacks in each category.

- *Distortion Single*: Dist-Rotation, Dist-RCrop, Dist-Erase, Dist-Bright, Dist-Contrast, Dist-Blur, Dist-Noise, Dist-JPEG.
- *Distortions Combination*: DistCom-Geo, DistCom-Photo, DistCom-Deg, DistCom-All.
- *Regeneration Single*: Regen-Diff, Regen-KLVAE.
- *Regeneration Rinsing*: Regen-2xDiff, Regen-4xDiff.
- *Adv Embedding Grey-box*: AdvEmbG-KLVAE8.
- *Adv Embedding Black-box*: AdvEmbB-CLIP, AdvEmbB-SdxlVAE, AdvEmbB-KLVAE16.
- *Adv Surrogate Detector*: AdvCls-WM1&WM2.

7.6.7 Details of Benchmarking Attacks

In addition to benchmarking watermarks, WAVES also facilitates the analysis from the perspective of attacks. Table 7.3 provides a leaderboard of individual attacks. A strong attack should result in low post-attack detection performance while simultaneously preserving image quality for practical uses. Therefore, we benchmark attacks according to both performance and quality degradation. Based on three Performance vs. Quality 2D plots in Figure 7.7, we first select two performance thresholds, TPR@0.1%FPR=0.95 and TPR@0.1%FPR=0.7, ensuring intersections with most attack curves. Then, we calculate the quality degradation for each attack at these two performance thresholds, denoted as Q@0.95P and Q@0.7P. Given that some attack curves do not intersect with either threshold, we also compute each attack's average performance and quality degradation across all strengths, termed as Avg P and Avg Q. We report these metrics — Q@0.95P, Q@0.7P, Avg P, and Avg Q — for attack comparison. Based on them, we also provide a ranking of 26 attacks for each watermarking method for reference. During this ranking process, we incorporate a 0.01 buffer for both P and Q, meaning that if the difference between any two values is less than 0.01, they are considered a tie in terms of ranking.

7.7 Details of Attacks

7.7.1 Distortion Attacks

For single distortions, we consider, as described in Section 7.3.2, eight types: rotation, resized-crop, random erasing, brightness adjustment, contrast adjustment, Gaussian blur, Gaussian noise, and JPEG compression. For each distortion, we consider five evenly distributed dis-

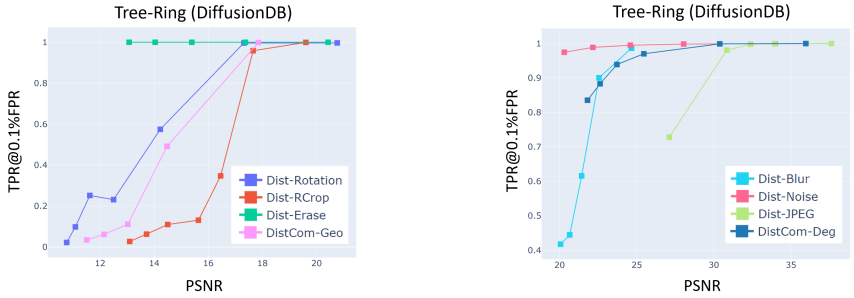
tortion strengths between minimum and maximum; the minimums and maximums are listed as follows.

- *Rotation*: rotate 9° to 45° clock-wise.
- *Resized-crop*: crop 10% to 50% of the image area.
- *Random erasing*: erase 5% to 25% of the image area and fill with gray color.
- *Brightness adjustment*: increase image brightness by 20% to 100%.
- *Contrast adjustment*: increase image contrast by 20% to 100%.
- *Gaussian blur*: blur with kernel size from 4 to 20 pixels.
- *Gaussian noise*: add Gaussian random noise with standard deviation from 0.02 to 0.1 (when pixel values normalized to [0, 1]).
- *JPEG compression*: compress with JPEG quality score from 90 to 10.

It is worth noting that our strength selections are more conservative than most of the watermark papers, such as [241, 242]. This is because we want to keep the image quality after distortion within a reasonable interval compared to the other attacks. While some watermark papers intentionally select unreasonably large distortion strength (for example, cropping 90% of image area in [241], or Gaussian blurring with kernel size 40 [242]) to demonstrate their robustness under some distortions. We implement the distortions following the standard image augmentations in the *torchvision* library.

For combinations of distortions (also called combo distortions in paper for short), we apply each single distortion with the same relative strength, where the relative strength is between 0

and 1, normalized with respect to the minimum and maximum strengths above. For combinations of geometric, photometric, and degradation distortions, we consider five evenly distributed normalized strengths from 0.05 to 0.45. For combinations of all distortions, we consider five evenly distributed normalized strengths from 0.05 to 0.20. The relative strengths are selected for reasonable image qualities after distortions again.



(a) Geometric distortions (PSNR ↑)

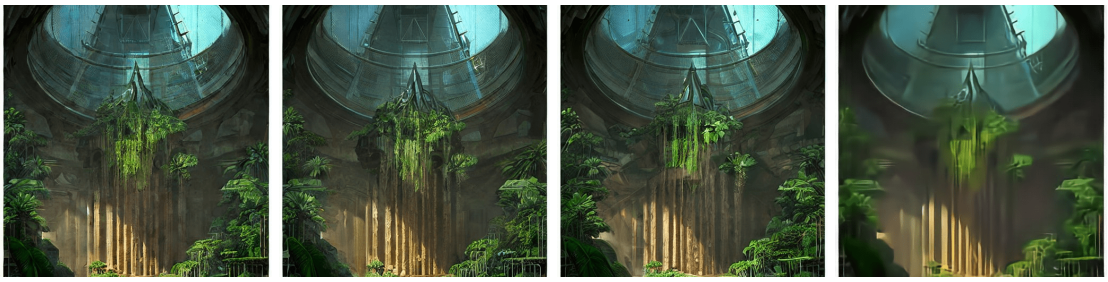
(b) Degradation distortions (PSNR ↑)

Figure 7.15: Distortions and their combinations. We combine three types of distortions: geometric, photometric, and degradation, both individually and collectively. By comparing quality-performance plots, we see combinations of distortions do not necessarily lead to better attacks.

7.7.2 Regeneration Attacks

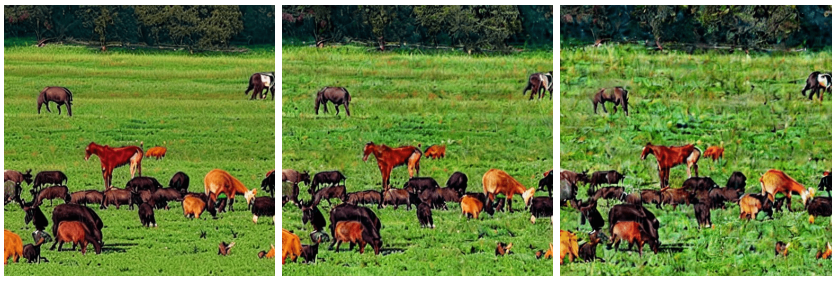
Following the language of Section 7.3, regeneration attacks [244] use off-the-shelf VAEs and diffusion models to transfer a target image $x \in \mathcal{X}$ to a latent representation followed by a restoration to $x' \in \mathcal{X}$ that is faithful to its original representation, i.e., $x' \approx x$. Since the chosen VAE or diffusion model will not be contained by the attacker's model of interest, the entire regeneration is likely to disrupt the latent representation of x, thereby damaging an embedded watermark. However, since the capacity of the attacker's regenerative model is inferior to the target model, x' will likely be of reduced quality. In this work, the target model is Stable Diffusion v2.1 while the surrogate model used for regeneration is Stable Diffusion v1.4.

Figure 7.3 demonstrates that a long diffusion or low-quality VAE attack will significantly reduce watermark detectability but at the expense of reduced image quality, which is clear by visual inspection of the sequence of images in Figure 7.16. Rising regenerations achieve similar reductions in detection, although too deep of rinsing regenerations (> 30 noising steps) significantly alter image quality as evidenced by Figure 7.17.



(a) Regen-Diff-40 (b) Regen-Diff-120 (c) Regen-Diff-200 (d) Regen-VAE-1

Figure 7.16: Regenerative diffusion with varying depth of noising steps and a VAE regeneration with a low quality factor.



(a) Rinse-4xDiff-10 (b) Regen-4xDiff-30 (c) Regen-4xDiff-50

Figure 7.17: 4x rinsing regeneration with varying depth of noising steps per diffusion.

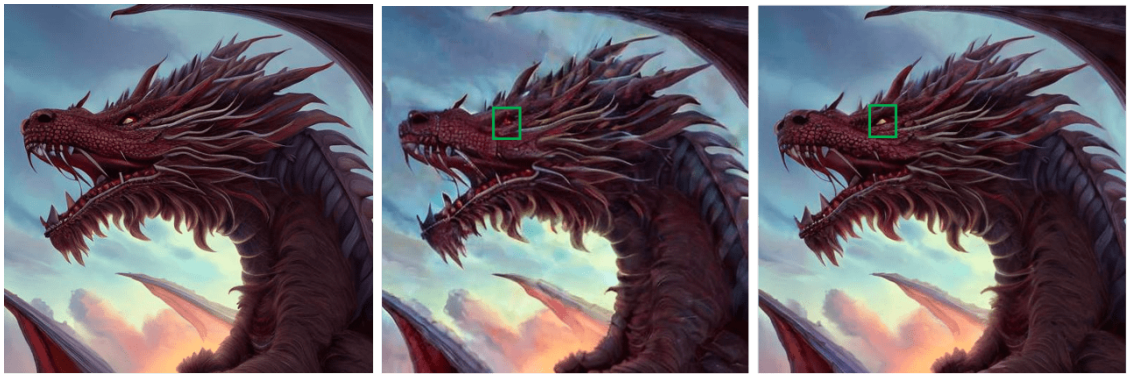
7.7.2.1 Prompted Regeneration

We propose a simple variation on a regenerative diffusion attack: if an image is produced via a known prompt, then an attacker uses the prompt to guide the diffusion of their surrogate model. This type of attack is reasonable and realistic for users of online generative models such

as DALL·E or Midjourney. Figure 7.3 and Tables 7.5 & 7.3 indicate that this type of attack, labeled Regen-DiffP is slightly stronger than conventional Regen-Diff.

7.7.2.2 Mixed Regeneration

Mixed regeneration refers to any style of attack that uses a regenerative diffusion on an image followed by VAE-style regeneration for the purposes of denoising. In Figure 7.3, we label examples of such attacks as RinseD-VAE and RegenD-KLVAE, which respectively denote VAE and KLVAE denoising following a 4x rinsing regeneration with 50 steps (Rinse-4xDiff-50). According to Figure 7.3, such a combination improves PSNR and CLIP-FID, as opposed to a Rinse-4xDiff alone. The restorative effects of mixed regeneration are visually observable for shallower (i.e., 2x or 3x) rinsing regenerations, as depicted in Figure 7.18. We do not extensively study or rank such attacks in this work, but include them as a future topic of research.



(a) Unattacked (b) Rinse-3xDiff (c) Rinse-3xDiff+VAE

Figure 7.18: An image of a dragon attacked using a 3x rinsing regeneration. Pushing the image through a VAE restores image quality, noticeable in the eye color of the dragon (indicated by the green box). Image is drawn from the Gustavosta Stable Diffusion dataset available.

All tested regeneration attacks are summarized as follows, with five evenly divided strengths between the listed minimum and maximum unless specified otherwise:

- *Regeneration via diffusion*: passes an image through Stable Diffusion v1.4 with strength as

the number of noise/de-noising steps timesteps, 40 to 200.

- *Regeneration via prompted diffusion*: passes an image through Stable Diffusion v1.4 conditioned on its generative prompt with strength as the number of noise/de-noising steps timesteps, 40 to 200.

- *Regeneration via VAE*: Image is encoded then decoded by a pre-trained VAE (bmshj2018) [280] with strength as quality level from 1 to 7.

- *Regeneration via KL-VAE*: Image is encoded and then decoded by a pre-trained KL-regularized autoencoder with strength as bottleneck sizes 4, 8, 16, or 32.

- *Rinsing generation 2x*: an image is noised then de-noised by Stable Diffusion v1.4 two times with strength as number of timesteps, 20-100 (per diffusion).

- *Rinsing generation 4x*: an image is noised then de-noised by Stable Diffusion v1.4 two times with strength as number of timesteps, 10-50 (per diffusion).

- *Mixed Regeneration via VAE*: an image passed through a rinsing regeneration 4x (for 50 timesteps each) and then a VAE with strength as quality level from 1-7.

- *Mixed Regeneration via KL-VAE*: an image passed through a rinsing regeneration 4x (for 50 timesteps each) and then a KL-VAE with strength as bottleneck sizes 4, 8, 16, or 32.

7.7.3 Adversarial Attacks

7.7.3.1 Embedding Attack

The embedding attacks use off-the-shelf encoders and perform untargeted attacks. We use the Projected Gradient Descent (PGD) algorithm [258] to optimize the adversarial examples. We conduct the attack using a range of perturbation budgets ϵ, specifically $\{$2/255, 4/255, 6/255, 8/255$\}$. All the attacks are configured with a step size of $\alpha = 0.05 * \epsilon$ and the number of total iterations of 200. The attacks are on the watermarked images, aiming to remove the watermarks by perturbing their latent representations.

7.7.3.2 Surrogate Detector Attack

Figure 7.5 illustrates the three settings of training the surrogate detectors. In all three settings, we train the surrogate detectors by fine-tuning the ResNet18[2] for 10 epochs with a learning rate of 0.001 and a batch size of 128. The training images are either generated by the victim generator with the ImageNet text prompts "A photo of a $\{$*ImageNet class name*$\}$," or real ImageNet images. We randomly shuffle those images and build the binary training set according to each setting. In the AdvCls-UnWM&WM setting, we train the surrogate detector with 3000 images (1500 images per class) since we find a larger training set might have the overfitting problem. In the AdvCls-Real&WM and AdvCls-WM1&WM2 settings, we train the surrogate detector with 15000 images (7500 images per class). The watermarked images in AdvCls-WM1&WM2 are embedded with two distinct messages. One message is the one used in the test watermarked images. The other one is randomly generated. In all three settings, we use

[2]https://pytorch.org/vision/main/models/generated/torchvision.models.resnet18.html

5000 images (2500 images per class) for validation (derived from the same source as the training set), and the training yields nearly 100% validation accuracy in all cases.

After completing the training phase, the adversary executes a Projected Gradient Descent (PGD) attack on the surrogate detector using the testing data (DiffusionDB, MS-COCO, DALL·E3). In all three settings, we conduct the attack using a range of perturbation budgets ϵ, specifically $\{2/255, 4/255, 6/255, 8/255\}$. The attack is configured with a step size of $\alpha = 0.01 * \epsilon$ and the number of total iterations of 50. By flipping the label, the adversary can either try to remove the watermarks or add the watermarks. The analyses of results appear in Section 7.8.2.

7.8 Additional Results

7.8.1 More Results for Identification

Figure 7.19 shows the Performance vs. Quality degradation plots under the user identification setting. Table 7.5 presents the ranking of attacks in the identification setup.

7.8.2 More Analyses on Surrogate Detector Attacks

The AdvCls-UnWM&WM attack leverages a surrogate model to distinguish between images that are watermarked and those that are not. As demonstrated in Figure 7.6, the PGD attack is effective in removing watermarks by flipping the label of watermarked images. This raises a question: Is it possible to similarly 'add' watermarks to clean images by flipping their labels? This process, commonly referred to as a spoofing attack, which

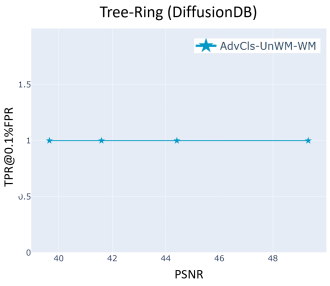


Figure 7.20: The spoofing attack fails for AdvCls-UnWM&WM.

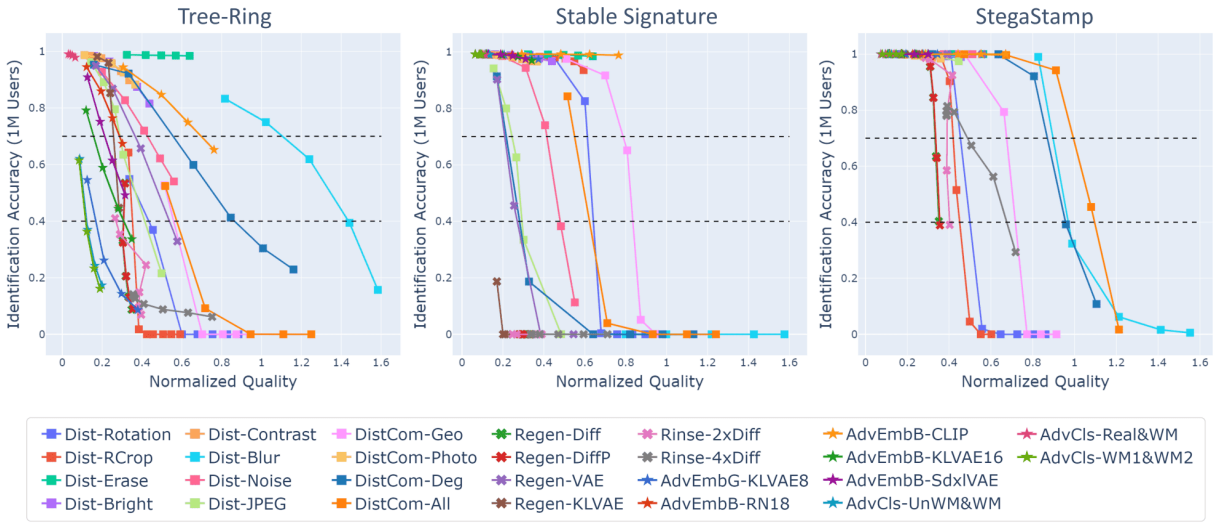


Figure 7.19: **Aggregated performance vs. quality degradation 2D plots under identification setup (one million users).** We evaluate each watermarking method under various attacks. Two dashed lines show to thresholds used for ranking attacks.

demonstrates a false detection of watermarks in clean images, is

explored in our study.

However, as illustrated in Figure 7.20, our attempts to add watermarks to clean images by simply flipping the labels were unsuccessful. In this experiment, detailed in Figure 7.20, we focus exclusively on unwatermarked images, aiming to introduce watermarks, while leaving already watermarked images untouched. Despite employing the most intensive perturbations, we were unable to artificially add watermarks to these images. This outcome leads to an intriguing inquiry: Why is the technique effective in removing watermarks but not in adding them? We delve into the underlying reasons for this asymmetry in Figure 7.21.

The insights from Figure 7.21 reveal that the surrogate model does not exactly remove the watermark. Instead, it perturbs the watermark along with other features within the latent space. The disturbance alone is sufficient to confuse the detector, making it challenging to recognize the watermark. In contrast, successfully adding watermarks requires precise modifications in the

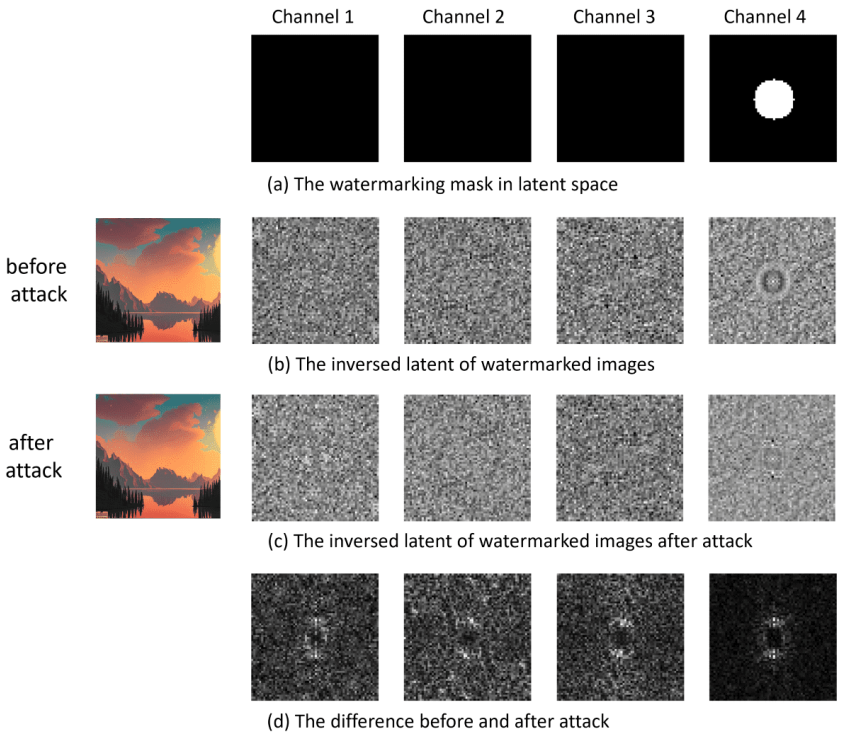


Figure 7.21: Visualization of AdvCls-UnWM&WM attack. (a) shows the watermarking mask of Tree-Ring where there are four channels, and we only watermark the last channel. The watermark message is the rings, which contain ten complex numbers that are not shown in the figure. (b) and (c) show the inversed latent before and after the attack in the Fourier space. We only show the real part of the latent. Clearly, the rings exist before the attack and vanish after the attack. (d) shows the magnitude of the element-wise difference before and after the attack. The attack not only perturbs the watermark part but also other features. The average magnitude change of the watermark-part and non-watermark-part is around 2:1. The attack successfully disturbs the watermark, albeit in an imprecise manner.

latent space, rather than mere perturbations, which proves to be a more challenging task. The relative imprecision of this attack may stem from the 'transferable gap' between the surrogate model and the ground-truth detector. Notably, for the purpose of watermark removal, perturbing the latent space proves to be adequately effective.

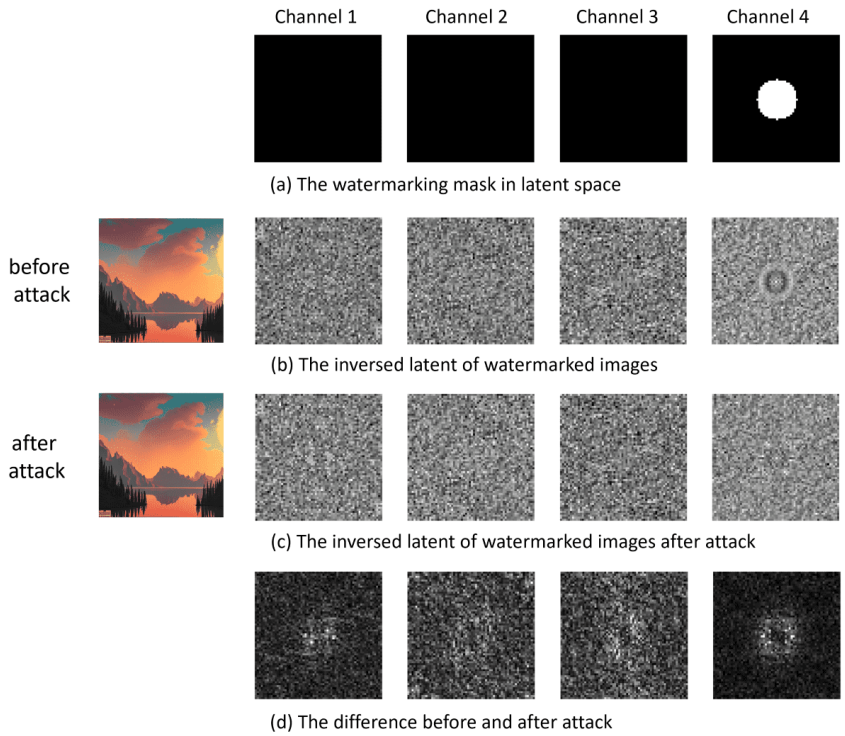


Figure 7.22: Visualization of AdvCls-WM1&WM2 attack. (a) and (b) are the same as that in Figure 7.21. (c) shows the inversed latent after the attack, where the watermark vanishes instead of changing to another watermark. (d) shows the magnitude of the element-wise difference before and after the attack. The attack not only perturbs the watermark part but also other features. The average magnitude change of the watermark-part and non-watermark-part is also around 2:1. Although the surrogate detector is trained to classify two different watermark messages. The attack based on it cannot change the watermark message from one to another but can effectively disturb the watermark.

These findings have led to the development of our proposed AdvCls-WM1&WM2 attack, which utilizes images watermarked with different messages (e.g., collected from two users, User1 and User2). The essential requirement for this approach is the surrogate model's ability to map images to the generator's latent space. This mapping allows the attacker to perturb the latent

space, removing the watermark. In contrast to the AdvCls-UnWM&WM approach, which uses both watermarked and non-watermarked images for training (differing only in the latent space), AdvCls-WM1&WM2 uses two sets of images, each embedded with a distinct watermark message (differing only in the latent space as well). Figure 7.22 shows that AdvCls-WM1&WM2 attack effectively disrupts the latent features of the images, including the watermarks. However, it lacks the precision to interchange the embedded watermark message. Consequently, while this attack can remove watermarks and mislead user identification—mistaking an image originally generated by User1 as belonging to another user—it cannot accurately manipulate the identification to frame User2 as desired by the attacker. The identification results in Figure 7.23 also support this finding. Although AdvCls-WM1&WM2 aims to misidentify images as belonging to User2, it often leads to misidentification as users other than User2. However, in a system with fewer users, like 100 users, and under intense attack conditions (e.g., strength=8), AdvCls-WM1&WM2 demonstrates a targeted identification success rate of 0.7%, showing a potential direction for attacks aimed at targeted user identification.

7.8.3 Visualization of Attacks

In Section 7.24, we present visualizations of several attacks included in the WAVES benchmark. Prefix indicates the attack strategy, while suffix indicates the strength.

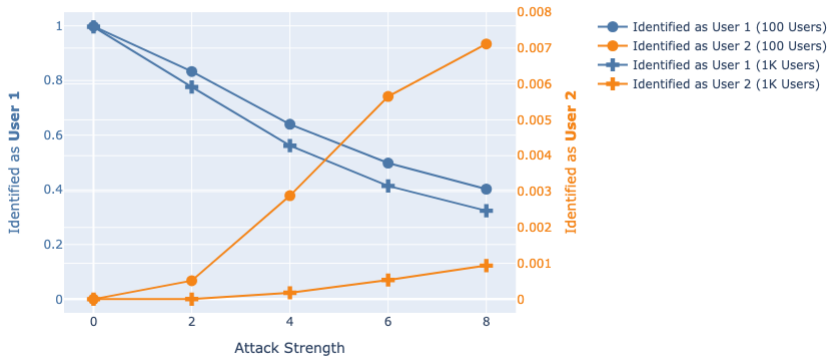
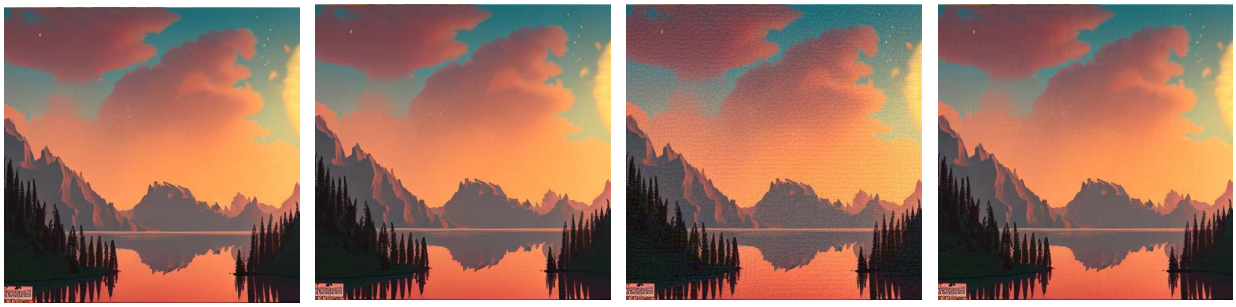
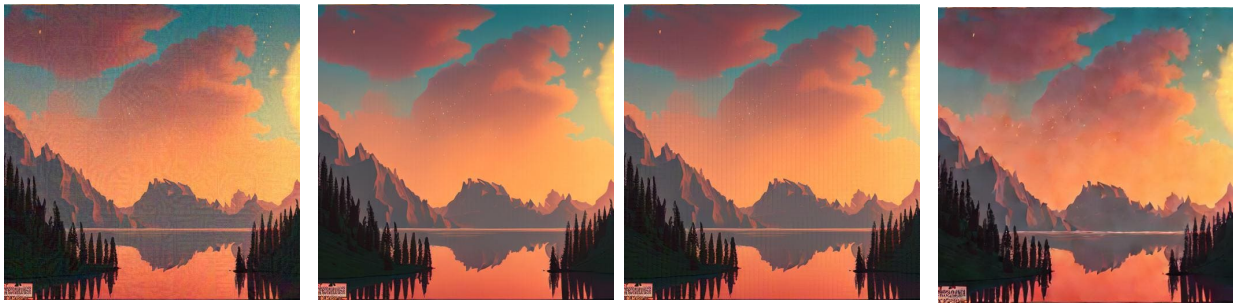


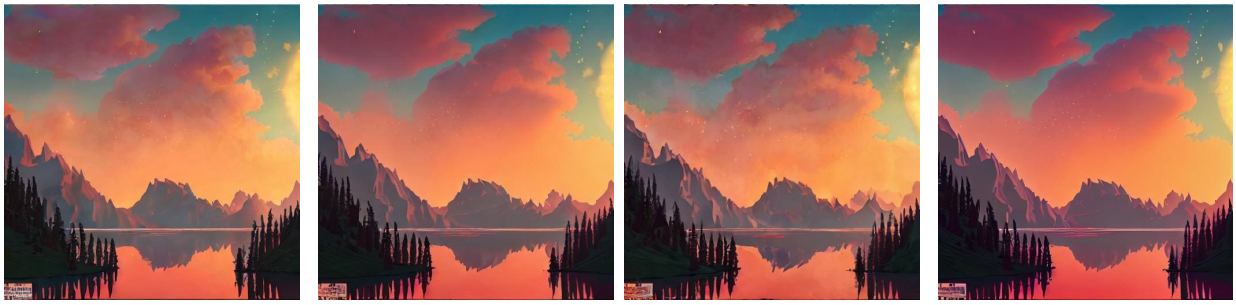
Figure 7.23: The user identification results for Tree-Ring under AdvCls-WM1&WM2 attacks. The original watermarked images are embedded with User1's message. AdvCls-WM1&WM2 tries to disrupt the latent feature of those images so that they can be misidentified as User2 generated. We simulate two settings: 100 users and 1000 users in total. The blue curves represent the proportion of images correctly identified as belonging to User1, while the orange curves show those misidentified as User2's. Note that, the axes for blue and orange curves have different ranges in the figure. With increasing attack strengths, the likelihood of correctly identifying them as User1's decreases significantly under both 100 and 1K user scenarios. However, misidentification as User2's images occurs notably only when the total number of users is small (e.g., 100 users).



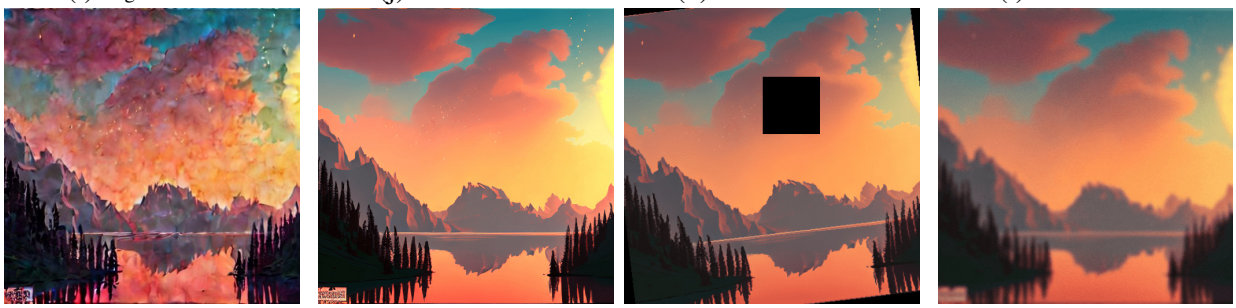
(a) Tree-Ring Unattacked (b) AdvEmbG-KLVAE8-2/255 (c) AdvEmbG-KLVAE8-8/255 (d) AdvEmbB-CLIP-2/255



(e) AdvEmbB-CLIP-8/255 (f) AdvClsWM1WM2-2/255 (g) AdvClsWM1WM28/255 (h) Regen-Diff-40



(i) Regen-Diff-200 (j) Rinse-2xDiff-20 (k) Rinse-2xDiff-100 (l) Rinse-4xDiff-10



(m) Rinse-4xDiff-50 (n) DistCom-Photo-0.15 (o) DistCom-Geo-0.15 (p) DistCom-Deg-0.15

Figure 7.24: A visual demonstration of various adversarial, regeneration, and distortion attacks on a Tree-Ring watermarked image. **Figure (a)** is the base unattacked image. The base prompt, drawn from DiffusionDB, is "digital painting of a lake at sunset surrounded by forests and mountains," along with further styling details.

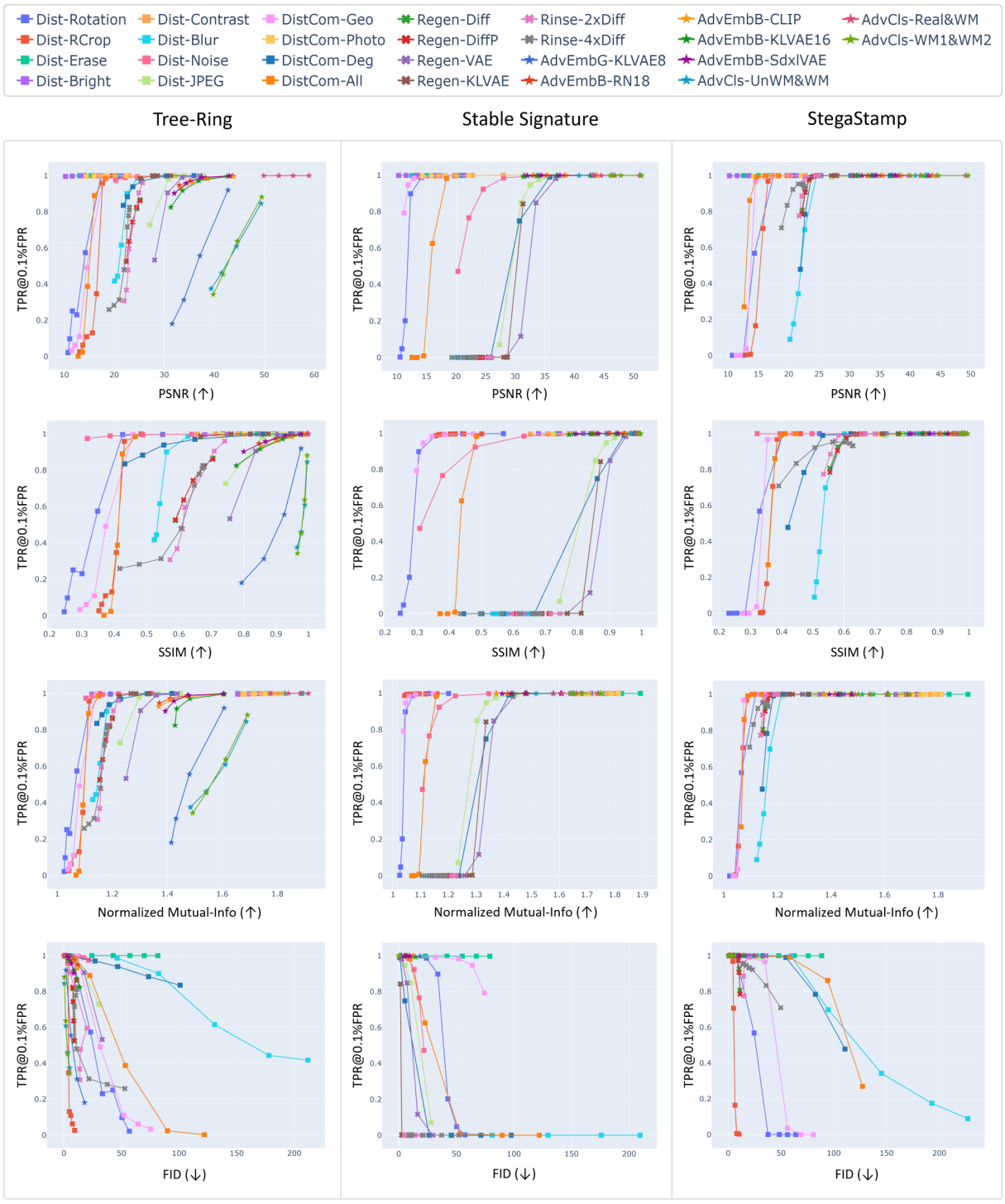


Figure 7.25: Evaluation on DiffusionDB dataset under the detection setup (part 1).

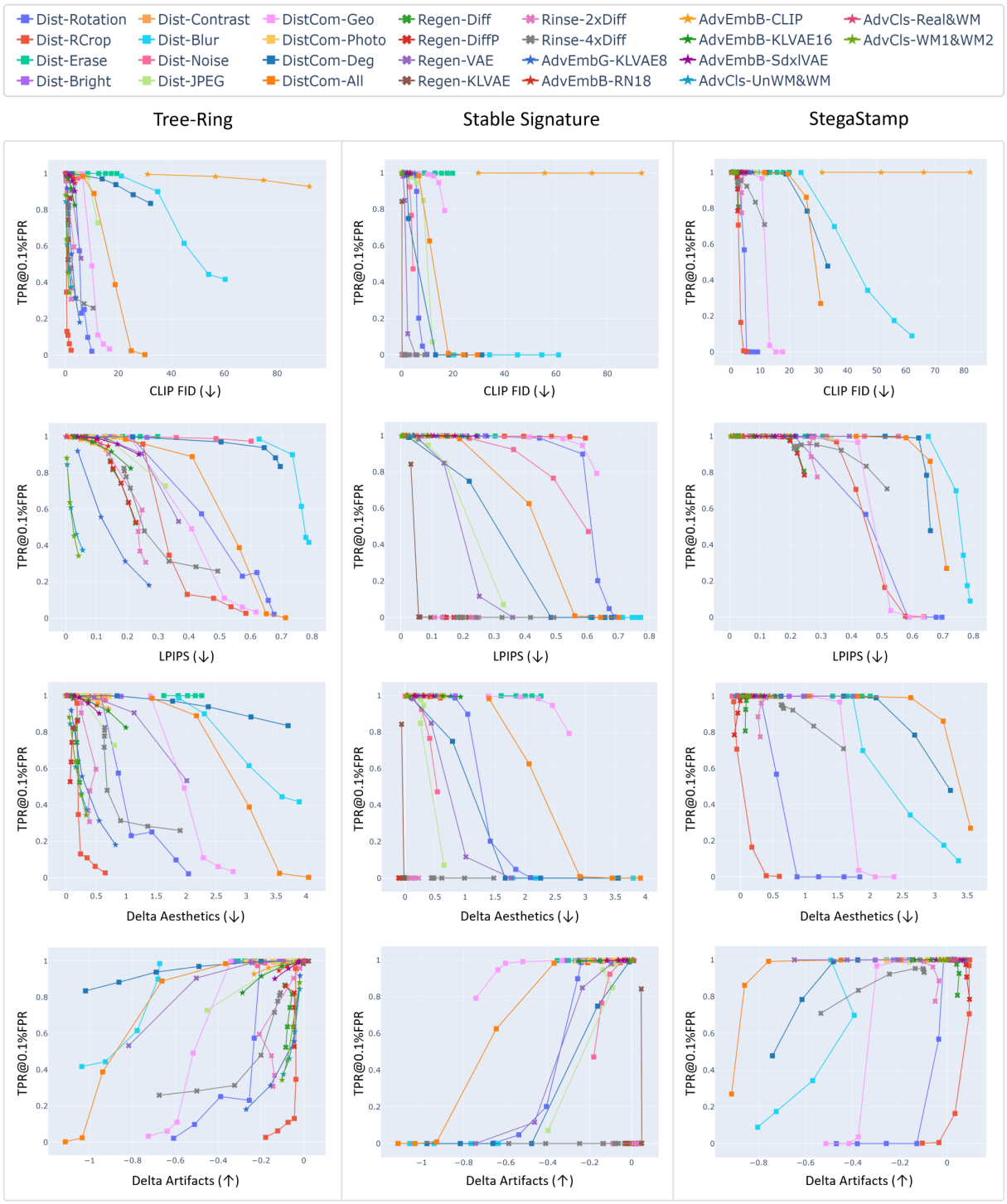


Figure 7.26: Evaluation on DiffusionDB dataset under the detection setup (part 2).

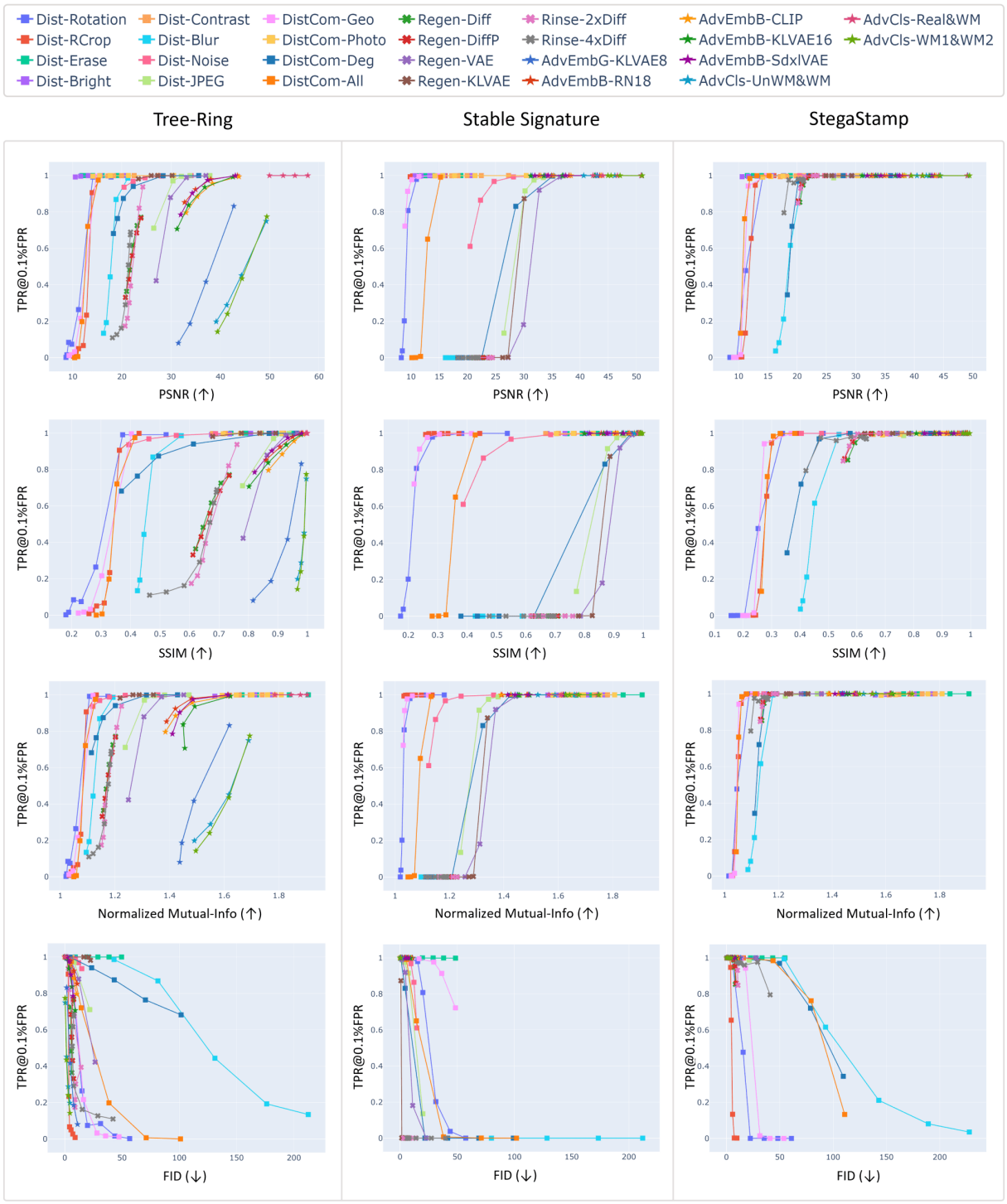


Figure 7.27: Evaluation on MS-COCO dataset under the detection setup (part 1).

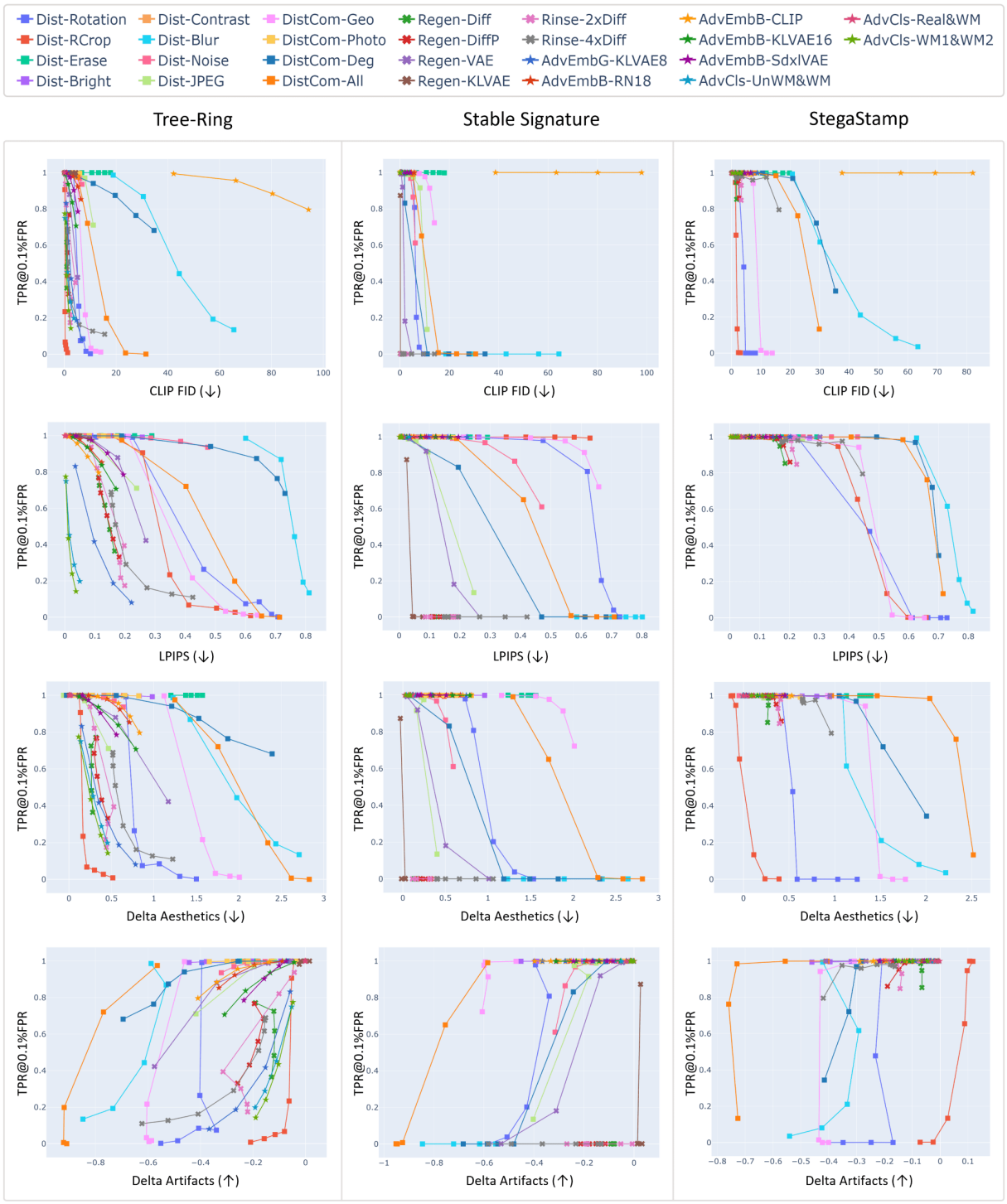


Figure 7.28: Evaluation on MS-COCO dataset under the detection setup (part 2).

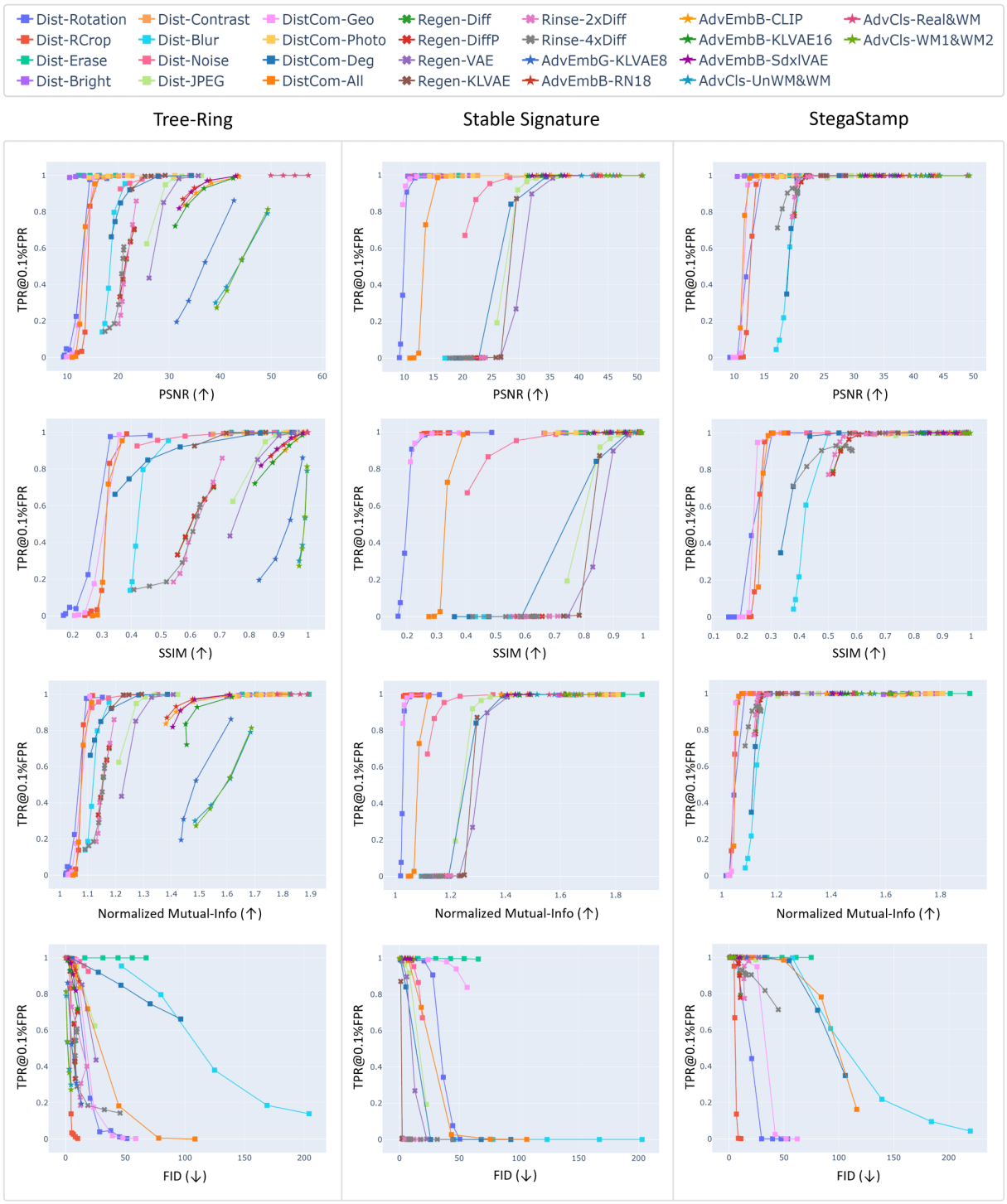


Figure 7.29: Evaluation on DALL·E3 dataset under the detection setup (part 1).

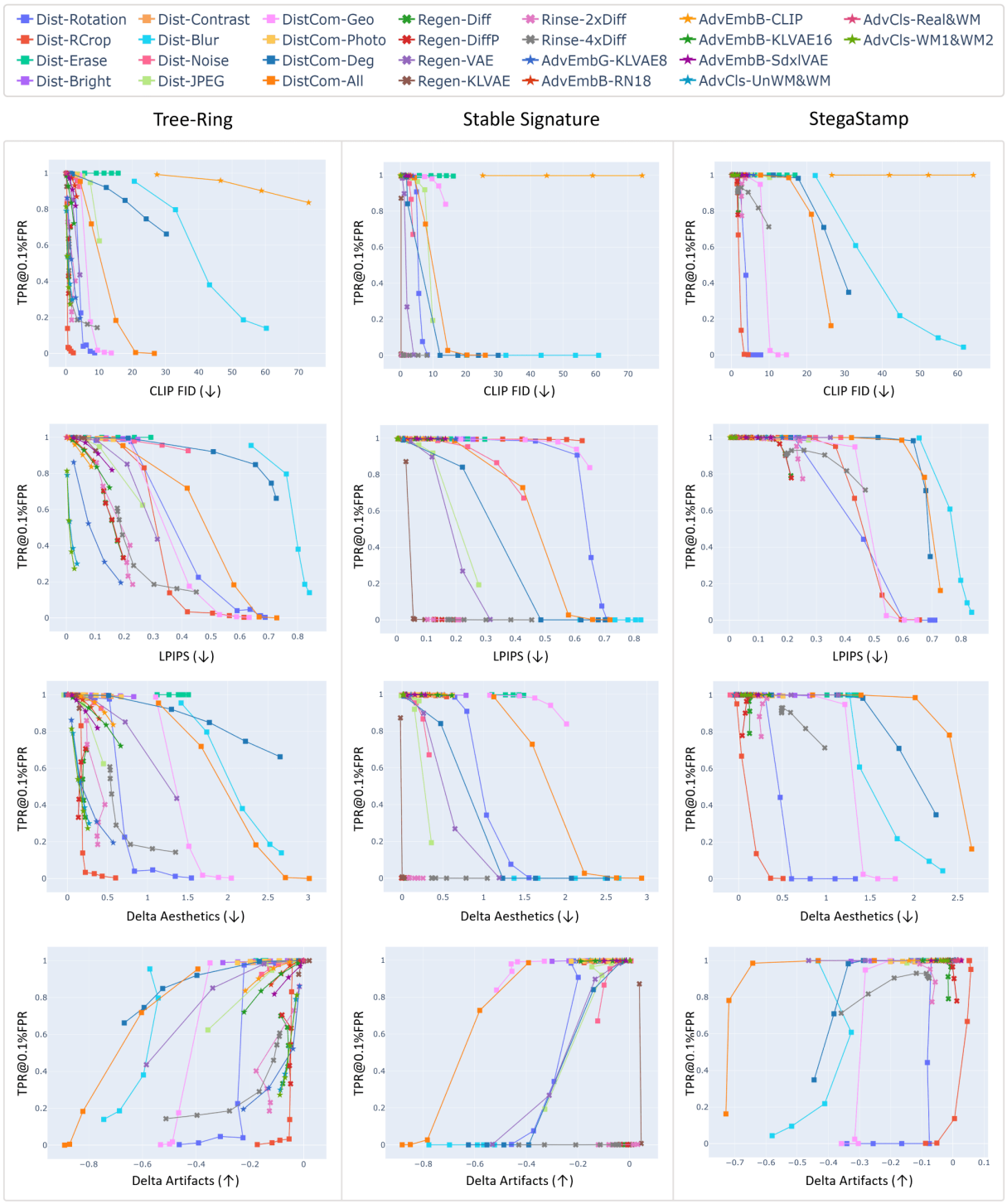


Figure 7.30: Evaluation on DALL·E3 dataset under the detection setup (part 2).

7.8.4 Full Results on DiffusionDB, MS-COCO and DALL·E3

7.9 Limitations

We only stress-test the Tree-Ring, Stable Signature, and Stegastamp watermarking algorithms. We curated these watermarks for WAVES after an extensive literature review indicated these three techniques to be the most powerful and practical candidates for deployment in the wild. However, we emphasize our framework is extensible to any watermarking method. Additionally, our attack ranking method relies on author-selected TPR thresholds and image quality metrics that we believe will fairly capture attack potency based on existing literature and experimental studies. The use of other quality metrics (MSE, Watson-DFT, etc.) and differing TPR thresholds may affect attack rankings.

Table 7.3: **Comparison of attacks across three watermarking methods under the detection setup.** Q denotes the normalized quality degradation, and P denotes the performance as derived from Figure 7.7. Q@0.95P measures quality degradation at a 0.95 performance threshold where "inf" denotes cases where all tested attack strengths yield performance above 0.95, and "-inf" where all are below. A similar notation applies to Q@0.7P. Avg P and Avg Q are the average performance and quality over all the attack strengths. The lower the performance and the smaller the quality degradation, the stronger the attack is. For each watermarking method, we rank attacks by Q@0.95P, Q@0.7P, Avg P, Avg Q, in that order, with lower values (↓) indicating stronger attacks. The top 5 attacks of each watermarking method are highlighted in red.

Attack	Tree-Ring					Stable Signature					StegaStamp				
	Rank	Q@0.95P	Q@0.7P	Avg P	Avg Q	Rank	Q@0.95P	Q@0.7P	Avg P	Avg Q	Rank	Q@0.95P	Q@0.7P	Avg P	Avg Q
Dist-Rotation	11	0.464	0.521	0.375	0.648	12	0.624	0.702	0.594	0.650	5	0.423	0.498	0.357	0.616
Dist-RCrop	18	0.592	0.592	0.332	0.463	24	inf	inf	0.995	0.461	6	0.602	0.602	0.540	0.451
Dist-Erase	26	inf	inf	1.000	0.490	25	inf	inf	0.998	0.489	25	inf	inf	1.000	0.483
Dist-Bright	25	inf	inf	0.997	0.304	23	inf	inf	0.998	0.305	22	inf	inf	0.998	0.317
Dist-Contrast	22	inf	inf	0.998	0.243	20	inf	inf	0.998	0.243	17	inf	inf	0.998	0.231
Dist-Blur	20	0.861	1.112	0.563	1.221	5	-inf	-inf	0.000	1.204	9	0.848	0.962	0.414	1.198
Dist-Noise	16	0.548	inf	0.980	0.395	8	0.402	0.520	0.870	0.390	24	inf	inf	1.000	0.360
Dist-JPEG	12	0.499	0.499	0.929	0.284	9	0.485	0.485	0.793	0.284	21	inf	inf	0.998	0.263
DistCom-Geo	13	0.525	0.593	0.277	0.768	13	0.850	inf	0.937	0.767	7	0.663	0.693	0.396	0.733
DistCom-Photo	22	inf	inf	0.998	0.242	20	inf	inf	0.998	0.243	17	inf	inf	0.998	0.239
DistCom-Deg	19	0.620	inf	0.892	0.694	7	0.206	0.369	0.300	0.679	8	0.826	0.975	0.852	0.664
DistCom-All	14	0.539	0.751	0.403	0.908	11	0.538	0.691	0.334	0.900	10	0.945	1.101	0.795	0.870
Regen-Diff	5	-inf	0.307	0.612	0.323	1	-inf	-inf	0.001	0.300	1	0.331	inf	0.943	0.327
Regen-DiffP	4	-inf	0.307	0.601	0.327	1	-inf	-inf	0.001	0.303	1	0.333	inf	0.940	0.329
Regen-VAE	17	0.578	0.578	0.832	0.348	10	0.545	0.545	0.516	0.339	23	inf	inf	1.000	0.343
Regen-KLVAE	22	inf	inf	0.990	0.233	6	-inf	0.176	0.217	0.206	17	inf	inf	1.000	0.240
Rinse-2xDiff	6	-inf	0.333	0.510	0.357	3	-inf	-inf	0.001	0.332	4	0.391	inf	0.941	0.366
Rinse-4xDiff	7	-inf	0.355	0.443	0.466	4	-inf	-inf	0.000	0.438	3	0.388	inf	0.909	0.477
AdvEmbG-KLVAE8	3	-inf	0.164	0.448	0.253	20	inf	inf	0.998	0.249	17	inf	inf	1.000	0.232
AdvEmbB-RN18	10	0.241	inf	0.953	0.218	17	inf	inf	0.999	0.212	14	inf	inf	1.000	0.196
AdvEmbB-CLIP	15	0.541	inf	0.932	0.549	26	inf	inf	0.999	0.541	25	inf	inf	1.000	0.488
AdvEmbB-KLVAE16	8	0.195	inf	0.888	0.238	19	inf	inf	0.997	0.233	14	inf	inf	1.000	0.206
AdvEmbB-SdxlVAE	9	0.222	inf	0.934	0.221	17	inf	inf	0.998	0.219	14	inf	inf	1.000	0.204
AdvCls-UnWM&WM	1	-inf	0.102	0.499	0.145	14	inf	inf	0.999	0.101	11	inf	inf	1.000	0.101
AdvCls-Real&WM	21	inf	inf	1.000	0.047	14	inf	inf	0.998	0.092	11	inf	inf	1.000	0.106
AdvCls-WM1&WM2	1	-inf	0.101	0.492	0.139	14	inf	inf	0.999	0.084	13	inf	inf	1.000	0.129

Table 7.4: A list of alternative watermarking algorithms not tested by WAVES in this work.

Method	**Known Weakness(es)**
DwtDct [237]	Distortion [242], Purification [245]
DwtDctSvd [237]	Distortion [242, 244], Purification [245], Regeneration [244]
RivaGan [256]	Regeneration [244], Purification [245]
SSL [269]	Distortion[244], Regeneration [244]
WatermarkDM [273]	Purification [245]

Table 7.5: **Comparison of attacks across three watermarking methods under the identification setup (one million users).** Q denotes the normalized quality degradation and P denotes the performance as derived from aggregated 2D plots. Q@0.7P measures quality degradation at a 0.7 performance threshold where "inf" denotes cases where all tested attack strengths yield performance above 0.7, and "-inf" where all are below. Q@0.4P is defined analogously. Avg P and Avg Q are the average performance and quality over all the attack strengths. The lower the performance and the smaller the quality degradation, the stronger the attack. For each watermarking method, we rank attacks by Q@0.7P, Q@0.4P, Avg P, Avg Q, in that order, with lower values (↓) indicating stronger attacks. The top 5 attack of each watermarking method are highlighted in red.

Attack	Tree-Ring					Stable Signature					StegaStamp				
	Rank	Q@0.7P	Q@0.4P	Avg P	Avg Q	Rank	Q@0.7P	Q@0.4P	Avg P	Avg Q	Rank	Q@0.7P	Q@0.4P	Avg P	Avg Q
Dist-Rotation	8	-inf	0.434	0.131	0.648	12	0.613	0.642	0.400	0.650	4	0.454	0.500	0.288	0.616
Dist-RCrop	11	-inf	0.592	0.094	0.463	24	inf	inf	0.972	0.461	6	0.602	0.602	0.494	0.451
Dist-Erase	26	inf	inf	0.986	0.490	25	inf	inf	0.988	0.489	25	inf	inf	1.000	0.483
Dist-Bright	22	inf	inf	0.913	0.304	23	inf	inf	0.982	0.305	22	inf	inf	0.995	0.317
Dist-Contrast	23	inf	inf	0.949	0.243	20	inf	inf	0.979	0.243	17	inf	inf	0.994	0.231
Dist-Blur	21	1.105	1.437	0.551	1.221	5	-inf	-inf	0.000	1.204	9	0.897	0.970	0.280	1.198
Dist-Noise	16	0.427	inf	0.728	0.395	8	0.415	0.480	0.633	0.390	24	inf	inf	1.000	0.360
Dist-JPEG	17	0.499	0.499	0.700	0.284	9	0.485	0.485	0.540	0.284	21	inf	inf	0.995	0.263
DistCom-Geo	9	-inf	0.559	0.105	0.768	13	0.788	0.835	0.519	0.767	7	0.676	0.717	0.359	0.733
DistCom-Photo	23	inf	inf	0.947	0.242	20	inf	inf	0.981	0.243	17	inf	inf	0.994	0.239
DistCom-Deg	18	0.556	0.864	0.570	0.694	7	0.216	0.281	0.183	0.679	8	0.870	0.957	0.737	0.664
DistCom-All	10	-inf	0.575	0.123	0.908	11	0.550	0.623	0.176	0.900	10	0.995	1.096	0.682	0.870
Regen-Diff	6	-inf	0.307	0.258	0.323	1	-inf	-inf	0.000	0.300	2	0.333	inf	0.766	0.327
Regen-DiffP	6	-inf	0.308	0.256	0.327	1	-inf	-inf	0.000	0.303	1	0.336	0.356	0.763	0.329
Regen-VAE	19	0.578	0.578	0.701	0.348	10	0.545	0.545	0.340	0.339	23	inf	inf	1.000	0.343
Regen-KLVAE	14	0.257	inf	0.810	0.233	6	-inf	-inf	0.047	0.206	17	inf	inf	0.999	0.240
Rinse-2xDiff	5	-inf	0.270	0.220	0.357	3	-inf	-inf	0.000	0.332	3	0.390	0.402	0.778	0.366
Rinse-4xDiff	1	-inf	-inf	0.110	0.466	4	-inf	-inf	0.000	0.438	5	0.488	0.676	0.687	0.477
AdvEmbG-KLVAE8	4	-inf	0.168	0.259	0.253	20	inf	inf	0.985	0.249	17	inf	inf	1.000	0.232
AdvEmbB-RN18	15	0.288	inf	0.811	0.218	17	inf	inf	0.990	0.212	14	inf	inf	1.000	0.196
AdvEmbB-CLIP	20	0.697	inf	0.798	0.549	26	inf	inf	0.991	0.541	25	inf	inf	1.000	0.488
AdvEmbB-KLVAE16	12	0.158	0.309	0.540	0.238	19	inf	inf	0.983	0.233	14	inf	inf	1.000	0.206
AdvEmbB-SdxlVAE	13	0.214	inf	0.692	0.221	17	inf	inf	0.986	0.219	14	inf	inf	1.000	0.204
AdvCls-UnWM&WM	2	-inf	0.123	0.352	0.145	14	inf	inf	0.991	0.101	11	inf	inf	1.000	0.101
AdvCls-Real&WM	25	inf	inf	0.986	0.047	14	inf	inf	0.990	0.092	11	inf	inf	1.000	0.106
AdvCls-WM1&WM2	2	-inf	0.118	0.343	0.139	14	inf	inf	0.991	0.084	13	inf	inf	1.000	0.129

Chapter 8: Conclusion

This dissertation investigates strategies for model compression and accelerated training. This manuscript can be divided into three modules: (1) general systems, (2) distributed systems, and (3) watermarking generative imagery.

In the first module, we are focused on ad hoc efficiency solutions. In Chapter 2, we discuss an approach for accelerating the eigenvector discovery of a large diagonalizable matrix using imperfect spectral knowledge. In Chapter 3, we develop a library and framework, conv_einsum for efficient evaluation and representation of multilinear operations in tensorized DNNs. A potential future direction for conv_einsum would include full integration with cuDNN and AutoML-like recommendations for compression ratio, tensor factorization type (CP, Tucker, Tensor-Train, etc.), and auto-tensorization for architectures beyond the ResNet family.

In the second module, we focus on efficiency in the distributed setting. Chapter 4 uses Count Sketch parametrizations of large model weights to reduce communication costs between clients and a centralized server. Chapter 5 leverages ADPSGD and model caching to enable wait-free model training for peer-to-peer learning (i.e., DFL) which preserves model performance and converges much faster than synchronized analogues. In Chapter 6, we use a novel family of LSH-like functions to allow devices to dynamically select tiny subarchitectures of large-scale recommender networks from round to round. For extreme multi-label datasets, future efforts should

focus on the direct compression of the massive labels themselves. Preliminary studies that use signed Gaussian projections on these labels show promising results on datasets such as Delicious-200K. General alleviation of label heterogeneity should also reduce iteration complexity. [281] suggests that training-free generation of images via generative models such as StyleGAN mimic the power spectra frequency found in natural scenes. We have found that encoding such natural noise with deficient labels to artificially balance label skew can drastically improve training over noniid MNIST, CIFAR-10, SVHN, and CIFAR-100.

In the third module (Chapter 7), we examine the robustness of watermarks for images, especially of a generative nature. We propose WAVES, a benchmark consisting of a myriad of realistic image manipulations to systematically break popular modern watermarking algorithms without severely compromising image quality. It is important to follow up on this work with proposals for defense. In particular, the usage of a U-Net or image "purifier" to restore an underlying watermark (following the StegaStamp decoding approach), maybe help ward off several of our most potent attacks.

This work demonstrates the potential for deploying and training large models onto resource-constrained systems. Research on the efficiency of AI/ML will only continue to grow in importance as private and public entities alike seek to to harness the impressive capacity of large models sans reliance on supercomputing clusters or expensive hardware. Viable solutions in this field will have profound implications for technological equity, IoT, and the environment.

Bibliography

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

[2] Jiahao Su, Jingling Li, Bobby Bhattacharjee, and Furong Huang. Tensorial neural networks: Generalization of neural networks and application to model compression. *arXiv preprint arXiv:1805.10352*, 2018.

[3] A Krizhevsky. Learning multiple layers of features from tiny images. *Master's thesis, University of Tront*, 2009.

[4] Adina Williams, Nikita Nangia, and Samuel R Bowman. A broad-coverage challenge corpus for sentence understanding through inference. *arXiv preprint arXiv:1704.05426*, 2017.

[5] Pablo Villalobos, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, Anson Ho, and Marius Hobbhahn. Machine learning model sizes and the parameter gap, 2022.

[6] Xiang Li, Wenhao Yang, Shusen Wang, and Zhihua Zhang. Communication-efficient local decentralized sgd methods. *arXiv preprint arXiv:1910.09126*, 2019.

[7] Sébastien Jean, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. On using very large target vocabulary for neural machine translation. *arXiv preprint arXiv:1412.2007*, 2014.

[8] Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. Scaling vision transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12104–12113, 2022.

[9] 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. *arXiv preprint arXiv:2001.08361*, 2020.

[10] Ryan Spring and Anshumali Shrivastava. Scalable and sustainable deep learning via randomized hashing. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 445–454, 2017.

[11] Tom Henighan, Jared Kaplan, Mor Katz, Mark Chen, Christopher Hesse, Jacob Jackson, Heewoo Jun, Tom B Brown, Prafulla Dhariwal, Scott Gray, et al. Scaling laws for autoregressive generative modeling. *arXiv preprint arXiv:2010.14701*, 2020.

[12] Ibrahim M Alabdulmohsin, Behnam Neyshabur, and Xiaohua Zhai. Revisiting neural scaling laws in language and vision. *Advances in Neural Information Processing Systems*, 35:22300–22312, 2022.

[13] Dionisis Kandris, Christos Nakas, Dimitrios Vomvas, and Grigorios Koulouras. Applications of wireless sensor networks: an up-to-date survey. *Applied system innovation*, 3(1): 14, 2020.

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

[15] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[16] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[17] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. Imagenet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*, pages 1–10, 2018.

[18] Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. Compute trends across three eras of machine learning. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2022.

[19] Kush Bhatia, Kunal Dahiya, Himanshu Jain, Anshul Mittal, Yashoteja Prabhu, and Manik Varma. The extreme classification repository: Multi-label datasets and code. *URL http://manikvarma. org/downloads/XC/XMLRepository. html*, 2016.

[20] Beidi Chen, Tharun Medini, James Farwell, Charlie Tai, Anshumali Shrivastava, et al. Slide: In defense of smart algorithms over hardware acceleration for large-scale deep learning systems. *Proceedings of Machine Learning and Systems*, 2:291–306, 2020.

[21] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[22] Alon Brutzkus and Amir Globerson. Why do larger models generalize better? a theoretical perspective via the xor problem. In *International Conference on Machine Learning*, pages 822–830. PMLR, 2019.

[23] Sébastien Bubeck and Mark Sellke. A universal law of robustness via isoperimetry. *Advances in Neural Information Processing Systems*, 34:28811–28822, 2021.

[24] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[25] Simon S Du, Xiyu Zhai, Barnabas Poczos, and Aarti Singh. Gradient descent provably optimizes over-parameterized neural networks. *arXiv preprint arXiv:1810.02054*, 2018.

[26] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

[27] Eran Malach, Gilad Yehudai, Shai Shalev-Schwartz, and Ohad Shamir. Proving the lottery ticket hypothesis: Pruning is all you need. In *International Conference on Machine Learning*, pages 6682–6691. PMLR, 2020.

[28] Shiwei Liu and Zhangyang Wang. Ten lessons we have learned in the new" sparseland": A short handbook for sparse neural network researchers. *arXiv preprint arXiv:2302.02596*, 2023.

[29] Guillaume Bellec, David Kappel, Wolfgang Maass, and Robert Legenstein. Deep rewiring: Training very sparse deep networks. *arXiv preprint arXiv:1711.05136*, 2017.

[30] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[31] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36, 2024.

[32] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. In *Low-Power Computer Vision*, pages 291–326. Chapman and Hall/CRC, 2022.

[33] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. In *International Conference on Machine Learning*, pages 2498–2507. PMLR, 2017.

[34] Christos Louizos, Karen Ullrich, and Max Welling. Bayesian compression for deep learning. *Advances in neural information processing systems*, 30, 2017.

[35] Yarin Gal, Jiri Hron, and Alex Kendall. Concrete dropout. *Advances in neural information processing systems*, 30, 2017.

[36] Durk P Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick. *Advances in neural information processing systems*, 28, 2015.

[37] Kirill Neklyudov, Dmitry Molchanov, Arsenii Ashukha, and Dmitry P Vetrov. Structured bayesian pruning via log-normal multiplicative noise. *Advances in Neural Information Processing Systems*, 30, 2017.

[38] Tahseen Rabbani, Marco Bornstein, and Furong Huang. Large-scale distributed learning via private on-device lsh. *Advances in Neural Information Processing Systems*, 36, 2024.

[39] Tahseen Rabbani, Brandon Feng, Yifan Yang, Arjun Rajkumar, Amitabh Varshney, and Furong Huang. Comfetch: Federated learning of large networks on memory-constrained clients via sketching. *arXiv preprint arXiv:2109.08346*, 2021.

[40] Tahseen Rabbani, Jiahao Su, Xiaoyu Liu, David Chan, Geoffrey Sangston, and Furong Huang. conv_einsum: A framework for representation and fast evaluation of multilinear operations in convolutional tensorial neural networks. *arXiv preprint arXiv:2401.03384*, 2024.

[41] Sebastian Raschka, Joshua Patterson, and Corey Nolet. Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information*, 11(4):193, 2020.

[42] C De Sa, Bryan He, Ioannis Mitliagkas, Christopher Ré, and Peng Xu. Accelerated stochastic power iteration. *Proceedings of machine learning research*, 84:58–67, 2018.

[43] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[44] Tahseen Rabbani, Apollo Jain, Arjun Rajkumar, and Furong Huang. Practical and fast momentum-based power methods. In *Mathematical and Scientific Machine Learning*, pages 721–756. PMLR, 2022.

[45] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. *Advances in Neural Information Processing Systems*, 28, 2015.

[46] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. *Advances in Neural Information Processing Systems*, 30, 2017.

[47] Marco Bornstein, Tahseen Rabbani, Evan Wang, Amrit Singh Bedi, and Furong Huang. Swift: Rapid decentralized federated learning via wait-free model communication. *arXiv preprint arXiv:2210.14026*, 2022.

[48] Frank Lin and William W Cohen. Power iteration clustering. In *ICML*, 2010.

[49] Frank Lin and William W Cohen. A very fast method for clustering big text datasets. In *ECAI*, pages 303–308, 2010.

[50] Nguyen Duc Thang, Young-Koo Lee, Sungyoung Lee, et al. Deflation-based power iteration clustering. *Applied intelligence*, 39(2):367–385, 2013.

[51] Michel Journée, Yurii Nesterov, Peter Richtárik, and Rodolphe Sepulchre. Generalized power method for sparse principal component analysis. *Journal of Machine Learning Research*, 11(2), 2010.

[52] Xiao-Tong Yuan and Tong Zhang. Truncated power method for sparse eigenvalue problems. *Journal of Machine Learning Research*, 14(4), 2013.

[53] Brian Gawalt, Youwei Zhang, and Laurent El Ghaoui. Sparse pca for text corpus summarization and exploration. In *NIPS 2010 Workshop on Low-Rank Matrix Approximation*. Citeseer, 2010.

[54] Reza Drikvandi and Olamide Lawal. Sparse principal component analysis for natural language processing. *Annals of Data Science*, pages 1–17, 2020.

[55] Cheolmin Kim and Diego Klabjan. Stochastic variance-reduced algorithms for pca with arbitrary mini-batch sizes. *Proceedings of the 23rdInternational Conference on Artificial Intelligence and Statistics (AISTATS)*, 2020.

[56] Vien V Mai and Mikael Johansson. Noisy accelerated power method for eigenproblems with applications. *IEEE Transactions on Signal Processing*, 67(12):3287–3299, 2019.

[57] Yousef Saad. *Numerical methods for large eigenvalue problems: revised edition*, volume 66. Siam, 2011.

[58] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU press, 2012.

[59] Qi Lei, Kai Zhong, and Inderjit S Dhillon. Coordinate-wise power method. In *Advances in Neural Information Processing Systems*, pages 2064–2072, 2016.

[60] Lloyd N Trefethen and David Bau III. *Numerical linear algebra*, volume 50. Siam, 1997.

[61] Po-An Wang and Chi-Jen Lu. Tensor decomposition via simultaneous power iteration. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3665–3673. JMLR. org, 2017.

[62] Steffen Börm and Christian Mehl. *Numerical methods for eigenvalue problems*. Walter de Gruyter, 2012.

[63] Moritz Hardt and Aaron Roth. Beyond worst-case analysis in private singular vector computation. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 331–340, 2013.

[64] Michael Kapralov and Kunal Talwar. On differentially private low rank approximation. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 1395–1414. SIAM, 2013.

[65] Ioannis Mitliagkas, Constantine Caramanis, and Prateek Jain. Memory limited, streaming pca. In *Advances in neural information processing systems*, pages 2886–2894, 2013.

[66] Ohad Shamir. A stochastic pca and svd algorithm with an exponential convergence rate. In *International Conference on Machine Learning*, pages 144–152, 2015.

[67] Prateek Jain, Chi Jin, Sham M Kakade, Praneeth Netrapalli, and Aaron Sidford. Streaming pca: Matching matrix bernstein and near-optimal finite sample guarantees for oja's algorithm. In *Conference on learning theory*, pages 1147–1164, 2016.

[68] P-A Absil, Robert Mahony, and Rodolphe Sepulchre. *Optimization algorithms on matrix manifolds*. Princeton University Press, 2009.

[69] Zaiwen Wen and Wotao Yin. A feasible method for optimization with orthogonality constraints. *Mathematical Programming*, 142(1):397–434, 2013.

[70] Renaud-Alexandre Pitaval, Wei Dai, and Olav Tirkkonen. Convergence of gradient descent for low-rank matrix approximation. *IEEE Transactions on Information Theory*, 61(8): 4451–4457, 2015.

[71] Zhiqiang Xu, Xin Cao, and Xin Gao. Convergence analysis of gradient descent for eigenvector computation. International Joint Conferences on Artificial Intelligence, 2018.

[72] Bao Wang and Qiang Ye. Stochastic gradient descent with nonlinear conjugate gradient-style adaptive momentum, 2020.

[73] Moritz Hardt and Eric Price. The noisy power method: A meta algorithm with applications. In *Advances in Neural Information Processing Systems*, pages 2861–2869, 2014.

[74] Matthew Partridge and Rafael A Calvo. Fast dimensionality reduction and simple pca. *Intelligent data analysis*, 2(3):203–214, 1998.

[75] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[76] Erkki Oja. Simplified neuron model as a principal component analyzer. *Journal of mathematical biology*, 15(3):267–273, 1982.

[77] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[78] Nina Mishra, Robert Schreiber, Isabelle Stanton, and Robert E Tarjan. Clustering social networks. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 56–67. Springer, 2007.

[79] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905, 2000.

[80] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Numerical mathematics*, volume 37. Springer Science & Business Media, 2010.

[81] Asifullah Khan, Anabia Sohail, Umme Zahoora, and Aqsa Saeed Qureshi. A survey of the recent architectures of deep convolutional neural networks. *Artificial Intelligence Review*, 53(8):5455–5516, 2020.

[82] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

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

[84] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.

[85] He Li, Kaoru Ota, and Mianxiong Dong. Learning iot in edge: Deep learning for the internet of things with edge computing. *IEEE network*, 32(1):96–101, 2018.

[86] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *arXiv preprint arXiv:1912.04977*, 2019.

[87] V Lebedev, Y Ganin, M Rakhuba, I Oseledets, and V Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. In *3rd International Conference on Learning Representations, ICLR 2015-Conference Track Proceedings*, 2015.

[88] Jean Kossaifi, Aran Khanna, Zachary Lipton, Tommaso Furlanello, and Anima Anandkumar. Tensor contraction layers for parsimonious deep nets. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 26–32, 2017.

[89] Jean Kossaifi, Zachary C Lipton, Arinbjorn Kolbeinsson, Aran Khanna, Tommaso Furlanello, and Anima Anandkumar. Tensor regression networks. *Journal of Machine Learning Research*, 21(123):1–21, 2020.

[90] Timur Garipov, Dmitry Podoprikhin, Alexander Novikov, and Dmitry Vetrov. Ultimate tensorization: compressing convolutional and fc layers alike. *arXiv preprint arXiv:1611.03214*, 2016.

[91] Wenqi Wang, Yifan Sun, Brian Eriksson, Wenlin Wang, and Vaneet Aggarwal. Wide compression: Tensor ring nets. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9329–9338, 2018.

[92] Robert NC Pfeifer, Jutho Haegeman, and Frank Verstraete. Faster identification of optimal contraction sequences for tensor networks. *Physical Review E*, 90(3):033315, 2014.

[93] Román Orús. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics*, 349:117–158, 2014.

[94] Lars Grasedyck, Daniel Kressner, and Christine Tobler. A literature survey of low-rank tensor approximation techniques. *GAMM-Mitteilungen*, 36(1):53–78, 2013.

[95] Andrzej Cichocki, Namgil Lee, Ivan Oseledets, Anh-Huy Phan, Qibin Zhao, Danilo P Mandic, et al. Tensor networks for dimensionality reduction and large-scale optimization: Part 1 low-rank tensor decompositions. *Foundations and Trends® in Machine Learning*, 9 (4-5):249–429, 2016.

[96] Andrzej Cichocki, Anh-Huy Phan, Qibin Zhao, Namgil Lee, Ivan Oseledets, Masashi Sugiyama, Danilo P Mandic, et al. Tensor networks for dimensionality reduction and large-scale optimization: Part 2 applications and future perspectives. *Foundations and Trends® in Machine Learning*, 9(6):431–673, 2017.

[97] Nadav Cohen and Amnon Shashua. Convolutional rectifier networks as generalized tensor decompositions. In *International Conference on Machine Learning*, pages 955–963, 2016.

[98] Valentin Khrulkov, Alexander Novikov, and Ivan Oseledets. Expressive power of recurrent neural networks. In *International Conference on Learning Representations*, 2018.

[99] Kohei Hayashi, Taiki Yamaguchi, Yohei Sugawara, and Shin-ichi Maeda. Exploring unexplored tensor network decompositions for convolutional neural networks. *Advances in Neural Information Processing Systems*, 32:5552–5562, 2019.

[100] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. In *Proceedings of the British Machine Vision Conference. BMVA Press*, 2014.

[101] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, pages 1269–1277, 2014.

[102] Xiangyu Zhang, Jianhua Zou, Xiang Ming, Kaiming He, and Jian Sun. Efficient and accurate approximations of nonlinear convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1984–1992, 2015.

[103] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

[104] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[105] Anh-Huy Phan, Konstantin Sobolev, Konstantin Sozykin, Dmitry Ermilov, Julia Gusak, Petr Tichavskỳ, Valeriy Glukhov, Ivan Oseledets, and Andrzej Cichocki. Stable low-rank tensor decomposition for compression of convolutional neural network. In *European Conference on Computer Vision*, pages 522–539. Springer, 2020.

[106] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. Tensorizing neural networks. In *Advances in Neural Information Processing Systems*, pages 442–450, 2015.

[107] Ivan V Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33 (5):2295–2317, 2011.

[108] Yinchong Yang, Denis Krompass, and Volker Tresp. Tensor-train recurrent neural networks for video classification. In *International Conference on Machine Learning*, pages 3891–3900. PMLR, 2017.

[109] Rose Yu, Stephan Zheng, Anima Anandkumar, and Yisong Yue. Long-term forecasting using tensor-train rnns. *arXiv preprint arXiv:1711.00073*, 2017.

[110] Jiahao Su, Wonmin Byeon, Furong Huang, Jan Kautz, and Animashree Anandkumar. Convolutional tensor-train lstm for spatio-temporal learning. *arXiv preprint arXiv:2002.09131*, 2020.

[111] Qibin Zhao, Guoxu Zhou, Shengli Xie, Liqing Zhang, and Andrzej Cichocki. Tensor ring decomposition. *arXiv preprint arXiv:1606.05535*, 2016.

[112] Jinmian Ye, Guangxi Li, Di Chen, Haiqin Yang, Shandian Zhe, and Zenglin Xu. Block-term tensor neural networks. *Neural Networks*, 2020.

[113] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

[114] Jean Kossaifi, Yannis Panagakis, Anima Anandkumar, and Maja Pantic. Tensorly: Tensor learning in python. *Journal of Machine Learning Research*, 20(26):1–6, 2019.

[115] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

[116] G Daniel, Johnnie Gray, et al. Opt_einsum-a python package for optimizing contraction order for einsum-like expressions. *Journal of Open Source Software*, 3(26):753, 2018.

[117] Alex Rogozhnikov and Cristian Garcia. Einops. https://github.com/arogozhnikov/einops, 2020.

[118] Alexander Reustle, Tahseen Rabbani, and Furong Huang. Fast gpu convolution for cp-decomposed tensorial neural networks. In *Proceedings of SAI Intelligent Systems Conference*, pages 468–487. Springer, 2020.

[119] Yu Pan, Maolin Wang, and Zenglin Xu. Tednet: A pytorch toolkit for tensor decomposition networks. *Neurocomputing*, 469:234–238, 2022.

[120] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[121] Ting Zhang, Guo-Jun Qi, Bin Xiao, and Jingdong Wang. Interleaved group convolutions. In *Proceedings of the IEEE international conference on computer vision*, pages 4373–4382, 2017.

[122] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[123] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[124] Karen Simonyan and Andrew Zisserman. Two-stream convolutional networks for action recognition in videos. In *Advances in Neural Information Processing Systems*, pages 568–576, 2014.

[125] Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. Ucf101: A dataset of 101 human actions classes from videos in the wild. *ArXiv*, abs/1212.0402, 2012.

[126] Yi Huang and Rajat Shrivastava. Two-stream action recognition. https://github.com/jeffreyyihuang/two-stream-action-recognition, 2019.

[127] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, et al. Conformer: Convolution-augmented transformer for speech recognition. *arXiv preprint arXiv:2005.08100*, 2020.

[128] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[129] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: an asr corpus based on public domain audio books. In *2015 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5206–5210. IEEE, 2015.

[130] Bijiao Wu, Dingheng Wang, Guangshe Zhao, Lei Deng, and Guoqi Li. Hybrid tensor decomposition in neural network compression. *Neural Networks*, 132:309–320, 2020.

[131] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[132] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[133] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, 2020.

[134] Yunbin Deng. Deep learning on mobile devices: a review. In *Mobile Multimedia/Image Processing, Security, and Applications 2019*, volume 10993, page 109930A. International Society for Optics and Photonics, 2019.

[135] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

[136] Farzin Haddadpour, Mohammad Mahdi Kamani, Aryan Mokhtari, and Mehrdad Mahdavi. Federated learning with compression: Unified analysis and sharp guarantees. In *International Conference on Artificial Intelligence and Statistics*, pages 2350–2358. PMLR, 2021.

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

[138] 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 *International Conference on Machine Learning*, pages 8253–8265. PMLR, 2020.

[139] Amirhossein Reisizadeh, Aryan Mokhtari, Hamed Hassani, Ali Jadbabaie, and Ramtin Pedarsani. Fedpaq: A communication-efficient federated learning method with periodic averaging and quantization. In *International Conference on Artificial Intelligence and Statistics*, pages 2021–2031. PMLR, 2020.

[140] Samuel Horvóth, Chen-Yu Ho, Ludovit Horvath, Atal Narayan Sahu, Marco Canini, and Peter Richtárik. Natural compression for distributed deep learning. In *Mathematical and Scientific Machine Learning*, pages 129–141. PMLR, 2022.

[141] Mher Safaryan, Egor Shulgin, and Peter Richtárik. Uncertainty principle for communication compression in distributed and federated learning and the search for an optimal compressor. *Information and Inference: A Journal of the IMA*, 11(2):557–580, 2022.

[142] Sarit Khirirat, Hamid Reza Feyzmahdavian, and Mikael Johansson. Distributed learning with compressed gradients. *arXiv preprint arXiv:1806.06573*, 2018.

[143] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.

[144] Simon Du, Jason Lee, Haochuan Li, Liwei Wang, and Xiyu Zhai. Gradient descent finds global minima of deep neural networks. In *International Conference on Machine Learning*, pages 1675–1685. PMLR, 2019.

[145] Timothy Yang, Galen Andrew, Hubert Eichner, Haicheng Sun, Wei Li, Nicholas Kong, Daniel Ramage, and Françoise Beaufays. Applied federated learning: Improving google keyboard query suggestions. *arXiv preprint arXiv:1812.02903*, 2018.

[146] Mohammad Mohammadi Amiri and Deniz Gündüz. Federated learning over wireless fading channels. *IEEE Transactions on Wireless Communications*, 19(5):3546–3557, 2020.

[147] Sai Praneeth Karimireddy, Quentin Rebjock, Sebastian Stich, and Martin Jaggi. Error feedback fixes signsgd and other gradient compression schemes. In *International Conference on Machine Learning*, pages 3252–3261. PMLR, 2019.

[148] Amirali Aghazadeh, Ryan Spring, Daniel Lejeune, Gautam Dasarathy, Anshumali Shrivastava, et al. Mission: Ultra large-scale feature selection using count-sketches. In *International Conference on Machine Learning*, pages 80–88. PMLR, 2018.

[149] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. QSGD: communication-efficient SGD via gradient quantization and encoding. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 1709–1720, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/6c340f25839e6acdc73414517203f5f0-Abstract.html.

[150] Chaoyue Niu, Fan Wu, Shaojie Tang, Lifeng Hua, Rongfei Jia, Chengfei Lv, Zhihua Wu, and Guihai Chen. Billion-scale federated learning on mobile clients: a submodel design with tunable privacy. In *MobiCom '20: The 26th Annual International Conference on Mobile Computing and Networking, London, United Kingdom, September 21-25, 2020*, pages 31:1–31:14. ACM, 2020. doi: 10.1145/3372224.3419188. URL https://doi.org/10.1145/3372224.3419188.

[151] Enmao Diao, Jie Ding, and Vahid Tarokh. Heterofl: Computation and communication efficient federated learning for heterogeneous clients. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=TNkPBBYFkXg.

[152] Suhail Mohmad Shah and Vincent KN Lau. Model compression for communication efficient federated learning. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[153] Ron Dorfman, Shay Vargaftik, Yaniv Ben-Itzhak, and Kfir Yehuda Levy. Docofl: Downlink compression for cross-device federated learning. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 8356–8388. PMLR, 2023. URL https://proceedings.mlr.press/v202/dorfman23a.html.

[154] Constantin Philippenko and Aymeric Dieuleveut. Bidirectional compression in heterogeneous settings for distributed or federated learning with partial participation: tight convergence guarantees. *arXiv preprint arXiv:2006.14591*, 2020.

[155] Hanlin Tang, Chen Yu, Xiangru Lian, Tong Zhang, and Ji Liu. Doublesqueeze: Parallel stochastic gradient descent with double-pass error-compensated compression. In *International Conference on Machine Learning*, pages 6155–6165. PMLR, 2019.

[156] Kaja Gruntkowska, Alexander Tyurin, and Peter Richtárik. Ef21-p and friends: Improved theoretical communication complexity for distributed optimization with bidirectional compression. In *International Conference on Machine Learning*, pages 11761–11807. PMLR, 2023.

[157] Shuai Zheng, Ziyue Huang, and James Kwok. Communication-efficient distributed blockwise momentum sgd with error-feedback. *Advances in Neural Information Processing Systems*, 32, 2019.

[158] Arkadi Nemirovski, Anatoli Juditsky, Guanghui Lan, and Alexander Shapiro. Robust stochastic approximation approach to stochastic programming. *SIAM Journal on optimization*, 19(4):1574–1609, 2009.

[159] Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, and Andrew Cotter. Pegasos: Primal estimated sub-gradient solver for svm. *Mathematical programming*, 127(1):3–30, 2011.

[160] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and system sciences*, 58(1):137–147, 1999.

[161] Anru R Zhang, Yuetian Luo, Garvesh Raskutti, and Ming Yuan. Islet: Fast and optimal low-rank tensor regression via importance sketching. *SIAM journal on mathematics of data science*, 2(2):444–479, 2020.

[162] Jiecao Chen and Qin Zhang. Bias-aware sketches. *arXiv preprint arXiv:1610.07718*, 2016.

[163] Thomas D Ahle, Michael Kapralov, Jakob BT Knudsen, Rasmus Pagh, Ameya Velingker, David P Woodruff, and Amir Zandieh. Oblivious sketching of high-degree polynomial kernels. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 141–160. SIAM, 2020.

[164] Misha Denil, Babak Shakibi, Laurent Dinh, Marc'Aurelio Ranzato, and Nando De Freitas. Predicting parameters in deep learning. *arXiv preprint arXiv:1306.0543*, 2013.

[165] Cheng Tai, Tong Xiao, Yi Zhang, Xiaogang Wang, et al. Convolutional neural networks with low-rank regularization. *arXiv preprint arXiv:1511.06067*, 2015.

[166] Nicolas Gillis and François Glineur. Low-rank matrix approximation with weights or missing data is np-hard. *SIAM Journal on Matrix Analysis and Applications*, 32(4):1149–1165, 2011.

[167] Shiva Prasad Kasiviswanathan, Nina Narodytska, and Hongxia Jin. Deep neural network approximation using tensor sketching. *arXiv preprint arXiv:1710.07850*, 2017.

[168] Lei Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? *arXiv preprint arXiv:1312.6184*, 2013.

[169] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[170] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pages 4114–4122, 2016.

[171] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[172] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[173] Ninh Pham and Rasmus Pagh. Fast and scalable polynomial kernels via explicit feature maps. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 239–247, 2013.

[174] Ruhui Jin, Tamara G Kolda, and Rachel Ward. Faster johnson-lindenstrauss transforms via kronecker products. *arXiv preprint arXiv:1909.04801*, 2019.

[175] Yining Wang, Hsiao-Yu Tung, Alexander Smola, and Animashree Anandkumar. Fast and guaranteed tensor decomposition via sketching. *arXiv preprint arXiv:1506.04448*, 2015.

[176] Yang Shi and Animashree Anandkumar. Higher-order count sketch: Dimensionality reduction that retains efficient tensor operations. *arXiv preprint arXiv:1901.11261*, 2019.

[177] Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. Sparsified sgd with memory. *arXiv preprint arXiv:1809.07599*, 2018.

[178] Jianyu Wang and Gauri Joshi. Cooperative sgd: A unified framework for the design and analysis of communication-efficient sgd algorithms. *arXiv preprint arXiv:1808.07576*, 2018.

[179] Stefano Savazzi, Monica Nicoli, and Vittorio Rampa. Federated learning with cooperating devices: A consensus approach for massive iot networks. *IEEE Internet of Things Journal*, 7(5):4641–4654, 2020.

[180] Jianyu Wang, Anit Kumar Sahu, Zhouyi Yang, Gauri Joshi, and Soummya Kar. Matcha: Speeding up decentralized sgd via matching decomposition sampling. In *2019 Sixth Indian Control Conference (ICC)*, pages 299–300. IEEE, 2019.

[181] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous decentralized parallel stochastic gradient descent. In *International Conference on Machine Learning*, pages 3043–3052. PMLR, 2018.

[182] Qinyi Luo, Jiaao He, Youwei Zhuo, and Xuehai Qian. Prague: High-performance heterogeneity-aware asynchronous decentralized training. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 401–416, 2020.

[183] Qi Liu, Bo Yang, Zhaojian Wang, Dafeng Zhu, Xinyi Wang, Kai Ma, and Xinping Guan. Asynchronous decentralized federated learning for collaborative fault diagnosis of pv stations. *IEEE Transactions on Network Science and Engineering*, 2022.

[184] Giorgi Nadiradze, Amirmojtaba Sabour, Peter Davies, Shigang Li, and Dan Alistarh. Asynchronous decentralized sgd with quantized and local updates. *Advances in Neural Information Processing Systems*, 34:6829–6842, 2021.

[185] Soheil Abbasloo and H Jonathan Chao. Sharpedge: An asynchronous and core-agnostic solution to guarantee bounded-delays. *CCF Transactions on Networking*, 3(1):35–50, 2020.

[186] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in neural information processing systems*, 24, 2011.

[187] Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Randomized gossip algorithms. *IEEE transactions on information theory*, 52(6):2508–2530, 2006.

[188] István Hegedűs, Gábor Danner, and Márk Jelasity. Decentralized learning works: An empirical comparison of gossip learning and federated learning. *Journal of Parallel and Distributed Computing*, 148:109–124, 2021.

[189] Tuncer Can Aysal, Mehmet Ercan Yildiz, Anand D Sarwate, and Anna Scaglione. Broadcast gossip algorithms for consensus. *IEEE Transactions on Signal processing*, 57(7): 2748–2761, 2009.

[190] Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Gossip algorithms: Design, analysis and applications. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 3, pages 1653–1664. IEEE, 2005.

[191] Saeed Ghadimi, Guanghui Lan, and Hongchao Zhang. Mini-batch stochastic approximation methods for nonconvex stochastic composite optimization. *Mathematical Programming*, 155(1):267–305, 2016.

[192] Angelia Nedic and Asuman Ozdaglar. Distributed subgradient methods for multi-agent optimization. *IEEE Transactions on Automatic Control*, 54(1):48–61, 2009.

[193] Alekh Agarwal and John C Duchi. Distributed delayed stochastic optimization. *Advances in neural information processing systems*, 24, 2011.

[194] Hao Ye, Le Liang, and Geoffrey Ye Li. Decentralized federated learning with unreliable communications. *IEEE Journal of Selected Topics in Signal Processing*, 2022.

[195] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *Foundations and Trends® in Machine Learning*, 14(1–2):1–210, 2021.

[196] Anastasia Koloskova, Nicolas Loizou, Sadra Boreiri, Martin Jaggi, and Sebastian Stich. A unified theory of decentralized sgd with changing topology and local updates. In *International Conference on Machine Learning*, pages 5381–5393. PMLR, 2020.

[197] Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research*, 13(1), 2012.

[198] Saeed Ghadimi and Guanghui Lan. Stochastic first-and zeroth-order methods for nonconvex stochastic programming. *SIAM Journal on Optimization*, 23(4):2341–2368, 2013.

[199] Yurii Nesterov. Introductory lectures on convex programming volume i: Basic course, 1998.

[200] Angelia Nedić and Alex Olshevsky. Distributed optimization over time-varying directed graphs. *IEEE Transactions on Automatic Control*, 60(3):601–615, 2014.

[201] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, 2002.

[202] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

[203] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 51(1):117–122, 2008.

[204] Anshumali Shrivastava and Ping Li. Asymmetric lsh (alsh) for sublinear time maximum inner product search (mips). *Advances in neural information processing systems*, 27, 2014.

[205] Beidi Chen, Zichang Liu, Binghui Peng, Zhaozhuo Xu, Jonathan Lingjie Li, Tri Dao, Zhao Song, Anshumali Shrivastava, and Christopher Re. Mongoose: A learnable lsh framework for efficient neural network training. In *International Conference on Learning Representations*, 2020.

[206] Beidi Chen and Anshumali Shrivastava. Revisiting winner take all (wta) hashing for sparse datasets. *arXiv preprint arXiv:1612.01834*, 2016.

[207] Piotr Indyk, Rajeev Motwani, Prabhakar Raghavan, and Santosh Vempala. Locality-preserving hashing in multidimensional spaces. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 618–625, 1997.

[208] Noam Nisan. Pseudorandom generators for space-bounded computations. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 204–212, 1990.

[209] Lars Engebretsen, Piotr Indyk, and Ryan O'Donnell. Derandomized dimensionality reduction with applications. *Symposium on Discrete Algorithms*, 2002.

[210] Piotr Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *Journal of the ACM (JACM)*, 53(3):307–323, 2006.

[211] Andrei Z Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pages 21–29. IEEE, 1997.

[212] Anshumali Shrivastava and Ping Li. In defense of minhash over simhash. In *Artificial Intelligence and Statistics*, pages 886–894. PMLR, 2014.

[213] Jay Yagnik, Dennis Strelow, David A Ross, and Ruei-sung Lin. The power of comparative reasoning. In *2011 International Conference on Computer Vision*, pages 2431–2438. IEEE, 2011.

[214] Devi Parikh and Kristen Grauman. Relative attributes. In *2011 International Conference on Computer Vision*, pages 503–510. IEEE, 2011.

[215] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020.

[216] Zaifeng Pan, Feng Zhang, Hourun Li, Chenyang Zhang, Xiaoyong Du, and Dong Deng. G-slide: A gpu-based sub-linear deep learning engine via lsh sparsification. *IEEE Transactions on Parallel and Distributed Systems*, 33(11):3015–3027, 2021.

[217] Junjie Jia and Zhipeng Lei. Personalized recommendation algorithm for mobile based on federated matrix factorization. In *Journal of Physics: Conference Series*, volume 1802, page 032021. IOP Publishing, 2021.

[218] Chengshuai Shi, Cong Shen, and Jing Yang. Federated multi-armed bandits with personalization. In *International Conference on Artificial Intelligence and Statistics*, pages 2917–2925. PMLR, 2021.

[219] Chuhan Wu, Fangzhao Wu, Tao Qi, Yongfeng Huang, and Xing Xie. Fedattack: Effective and covert poisoning attack on federated recommendation via hard sampling. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 4164–4172, 2022.

[220] Shijie Zhang, Hongzhi Yin, Tong Chen, Zi Huang, Quoc Viet Hung Nguyen, and Lizhen Cui. Pipattack: Poisoning federated recommender systems formanipulating item promotion. *arXiv preprint arXiv:2110.10926*, 2021.

[221] Zareen Alamgir, Farwa K Khan, and Saira Karim. Federated recommenders: methods, challenges and future. *Cluster Computing*, 25(6):4075–4096, 2022.

[222] Minghao Yan, Nicholas Meisburger, Tharun Medini, and Anshumali Shrivastava. Distributed slide: Enabling training large neural networks on low bandwidth and simple cpu-clusters via model parallelism and sparsity. *arXiv preprint arXiv:2201.12667*, 2022.

[223] Zhaozhuo Xu, Luyang Liu, Zheng Xu, and Anshumali Shrivastava. Adaptive sparse federated learning in large output spaces via hashing. In *Workshop on Federated Learning: Recent Advances and New Challenges (in Conjunction with NeurIPS 2022)*, 2022.

[224] William B Johnson. Extensions of lipschitz mappings into a hilbert space. *Contemp. Math.*, 26:189–206, 1984.

[225] Michel X Goemans and David P Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM (JACM)*, 42(6):1115–1145, 1995.

[226] Beidi Chen and Anshumali Shrivastava. Densified winner take all (wta) hashing for sparse datasets. In *Uncertainty in artificial intelligence*, 2018.

[227] Andrew Ziegler, Eric Christiansen, David Kriegman, and Serge Belongie. Locally uniform comparison image descriptor. *Advances in Neural Information Processing Systems*, 25, 2012.

[228] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal lsh for angular distance. *Advances in neural information processing systems*, 28, 2015.

[229] Felix Yu, Sanjiv Kumar, Yunchao Gong, and Shih-Fu Chang. Circulant binary embedding. In *International conference on machine learning*, pages 946–954. PMLR, 2014.

[230] Ian En-Hsu Yen, Satyen Kale, Felix Yu, Daniel Holtmann-Rice, Sanjiv Kumar, and Pradeep Ravikumar. Loss decomposition for fast learning in large output spaces. In *International Conference on Machine Learning*, pages 5640–5649. PMLR, 2018.

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

[232] Arjun K Gupta and Saralees Nadarajah. *Handbook of beta distribution and its applications*. CRC press, 2004.

[233] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

[234] Prafulla Dhariwal and Alexander Nichol. Diffusion models beat gans on image synthesis. *Advances in neural information processing systems*, 34:8780–8794, 2021.

[235] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10684–10695, 2022.

[236] Executive Office of the President. Executive order 14110: Safe, secure, and trustworthy development and use of artificial intelligence. *Federal Register*, 88:75191–75226, 2023.

[237] Ali Al-Haj. Combined dwt-dct digital image watermarking. *Journal of computer science*, 3(9):740–746, 2007.

[238] Jiren Zhu, Russell Kaplan, Justin Johnson, and Li Fei-Fei. Hidden: Hiding data with deep networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 657–672, 2018.

[239] Kevin Alex Zhang, Lei Xu, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. Robust invisible video watermarking with attention. *arXiv preprint arXiv:1909.01285*, 2019.

[240] Matthew Tancik, Ben Mildenhall, and Ren Ng. Stegastamp: Invisible hyperlinks in physical photographs. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2117–2126, 2020.

[241] Pierre Fernandez, Guillaume Couairon, Hervé Jégou, Matthijs Douze, and Teddy Furon. The stable signature: Rooting watermarks in latent diffusion models. *arXiv preprint arXiv:2303.15435*, 2023.

[242] Yuxin Wen, John Kirchenbauer, Jonas Geiping, and Tom Goldstein. Tree-ring watermarks: Fingerprints for diffusion images that are invisible and robust. *arXiv preprint arXiv:2305.20030*, 2023.

[243] Nils Lukas, Abdulrahman Diaa, Lucas Fenaux, and Florian Kerschbaum. Leveraging optimization for adaptive attacks on image watermarks. *arXiv preprint arXiv:2309.16952*, 2023.

[244] Xuandong Zhao, Kexun Zhang, Zihao Su, Saastha Vasan, Ilya Grishchenko, Christopher Kruegel, Giovanni Vigna, Yu-Xiang Wang, and Lei Li. Invisible image watermarks are provably removable using generative ai, 2023.

[245] Mehrdad Saberi, Vinu Sankar Sadasivan, Keivan Rezaei, Aounon Kumar, Atoosa Chegini, Wenxiao Wang, and Soheil Feizi. Robustness of ai-image detectors: Fundamental limits and practical attacks. *arXiv preprint arXiv:2310.00076*, 2023.

[246] Ingemar Cox, Matthew Miller, Jeffrey Bloom, Jessica Fridrich, and Ton Kalker. *Digital watermarking and steganography*. Morgan kaufmann, 2007.

[247] Ning Yu, Vladislav Skripniuk, Sahar Abdelnabi, and Mario Fritz. Artificial fingerprinting for generative models: Rooting deepfake attribution in training data. In *Proceedings of the IEEE/CVF International conference on computer vision*, pages 14448–14457, 2021.

[248] Yu Zeng, Mo Zhou, Yuan Xue, and Vishal M Patel. Securing deep generative models with universal adversarial signature. *arXiv preprint arXiv:2305.16310*, 2023.

[249] Nils Lukas and Florian Kerschbaum. Ptw: Pivotal tuning watermarking for pre-trained image generators. *arXiv preprint arXiv:2304.07361*, 2023.

[250] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems*, 30, 2017.

[251] Tuomas Kynkäänniemi, Tero Karras, Miika Aittala, Timo Aila, and Jaakko Lehtinen. The role of imagenet classes in fr\'echet inception distance. *arXiv preprint arXiv:2203.06026*, 2022.

[252] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *CVPR*, 2018.

[253] Jiazheng Xu, Xiao Liu, Yuchen Wu, Yuxuan Tong, Qinkai Li, Ming Ding, Jie Tang, and Yuxiao Dong. Imagereward: Learning and evaluating human preferences for text-to-image generation. *arXiv preprint arXiv:2304.05977*, 2023.

[254] Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Logan Engstrom, Brandon Tran, and Aleksander Madry. Adversarial examples are not bugs, they are features. *Advances in neural information processing systems*, 32, 2019.

[255] Anirban Chakraborty, Manaar Alam, Vishal Dey, Anupam Chattopadhyay, and Debdeep Mukhopadhyay. Adversarial attacks and defences: A survey. *arXiv preprint arXiv:1810.00069*, 2018.

[256] Yinpeng Dong, Huanran Chen, Jiawei Chen, Zhengwei Fang, Xiao Yang, Yichi Zhang, Yu Tian, Hang Su, and Jun Zhu. How robust is google's bard to adversarial image attacks? *arXiv preprint arXiv:2309.11751*, 2023.

[257] Nathan Inkawhich, Wei Wen, Hai Helen Li, and Yiran Chen. Feature space perturbations yield more transferable adversarial examples. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7066–7074, 2019.

[258] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[259] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021.

[260] Dustin Podell, Zion English, Kyle Lacey, Andreas Blattmann, Tim Dockhorn, Jonas Müller, Joe Penna, and Robin Rombach. Sdxl: Improving latent diffusion models for high-resolution image synthesis. *arXiv preprint arXiv:2307.01952*, 2023.

[261] Zijie J Wang, Evan Montoya, David Munechika, Haoyang Yang, Benjamin Hoover, and Duen Horng Chau. Diffusiondb: A large-scale prompt gallery dataset for text-to-image generative models. *arXiv preprint arXiv:2210.14896*, 2022.

[262] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pages 740–755. Springer, 2014.

[263] JJK ó Ruanaidh, WJ Dowling, and FM Boland. Watermarking digital images for copyright protection. *IEE PROCEEDINGS VISION IMAGE AND SIGNAL PROCESSING*, 143:250–256, 1996.

[264] Ingemar J Cox, Joe Kilian, Tom Leighton, and Talal Shamoon. Secure spread spectrum watermarking for images, audio and video. In *Proceedings of 3rd IEEE international conference on image processing*, volume 3, pages 243–246. IEEE, 1996.

[265] Joseph JK O'Ruanaidh and Thierry Pun. Rotation, scale and translation invariant digital image watermarking. In *Proceedings of International Conference on Image Processing*, volume 1, pages 536–539. IEEE, 1997.

[266] Chin-Chen Chang, Piyu Tsai, and Chia-Chen Lin. Svd-based digital image watermarking scheme. *Pattern Recognition Letters*, 26(10):1577–1586, 2005.

[267] Jamie Hayes and George Danezis. Generating steganographic images via adversarial training. *Advances in neural information processing systems*, 30, 2017.

[268] Mahdi Ahmadi, Alireza Norouzi, Nader Karimi, Shadrokh Samavi, and Ali Emami. Redmark: Framework for residual diffusion watermarking based on deep networks. *Expert Systems with Applications*, 146:113157, 2020.

[269] Pierre Fernandez, Alexandre Sablayrolles, Teddy Furon, Hervé Jégou, and Matthijs Douze. Watermarking images in self-supervised latent spaces. In *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3054–3058. IEEE, 2022.

[270] Bita Darvish Rouhani, Huili Chen, and Farinaz Koushanfar. Deepsigns: A generic watermarking framework for ip protection of deep learning models. *arXiv preprint arXiv:1804.00750*, 2018.

[271] Huili Chen, Bita Darvish Rouhani, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar. Deepmarks: A secure fingerprinting framework for digital rights management of deep learning models. In *Proceedings of the 2019 on International Conference on Multimedia Retrieval*, pages 105–113, 2019.

[272] Hengrui Jia, Christopher A Choquette-Choo, Varun Chandrasekaran, and Nicolas Papernot. Entangled watermarks as a defense against model extraction. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1937–1954, 2021.

[273] Yunqing Zhao, Tianyu Pang, Chao Du, Xiao Yang, Ngai-Man Cheung, and Min Lin. A recipe for watermarking diffusion models. *arXiv preprint arXiv:2303.10137*, 2023.

[274] Jiaming Song, Chenlin Meng, and Stefano Ermon. Denoising diffusion implicit models. *arXiv preprint arXiv:2010.02502*, 2020.

[275] Weili Nie, Brandon Guo, Yujia Huang, Chaowei Xiao, Arash Vahdat, and Anima Anandkumar. Diffusion models for adversarial purification. *arXiv preprint arXiv:2205.07460*, 2022.

[276] Zhengyuan Jiang, Jinghuai Zhang, and Neil Zhenqiang Gong. Evading watermark based detection of ai-generated content. *arXiv preprint arXiv:2305.03807*, 2023.

[277] Martin Kutter and Fabien AP Petitcolas. Fair benchmark for image watermarking systems. In *Security and watermarking of multimedia contents*, volume 3657, pages 226–239. SPIE, 1999.

[278] Hai Tao, Li Chongmin, Jasni Mohamad Zain, and Ahmed N Abdalla. Robust image watermarking theories and techniques: A review. *Journal of applied research and technology*, 12(1):122–138, 2014.

[279] Fabien AP Petitcolas. Watermarking schemes evaluation. *IEEE signal processing magazine*, 17(5):58–64, 2000.

[280] Johannes Ballé, David Minnen, Saurabh Singh, Sung Jin Hwang, and Nick Johnston. Variational image compression with a scale hyperprior. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL `https://openreview.net/forum?id=rkcQFMZRb`.

[281] Manel Baradad Jurjo, Jonas Wulff, Tongzhou Wang, Phillip Isola, and Antonio Torralba. Learning to see by looking at noise. *Advances in Neural Information Processing Systems*, 34:2556–2569, 2021.