Communication-Efficient Second-Order Optimization Methods on Modern Parallel Systems

by

Saeed Soori

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy

Department of Computer Science
University of Toronto

Communication-Efficient Second-Order Optimization Methods on Modern Parallel Systems

Saeed Soori
Doctor of Philosophy

Department of Computer Science
University of Toronto
2022

Abstract

The cost of an algorithm on modern computing platforms depends both on the computation and "communication". Communication in a general sense is defined as the data movement among levels of a memory hierarchy or between processors. On modern computer architectures, the communication cost is often orders of magnitude larger than the computation cost and this gap is rapidly increasing. This suggests that for the best performance the algorithms should reduce the communication even at the cost of increased computations.

The need to automatically process big data has driven the rapid development and deployment of machine learning algorithms. Machine learning applications require some form of mathematical optimization to find the best parameters obtained by optimizing some objective function representing the machine learning problem. Second-order optimization methods accelerate the optimization process by capturing the geometry of the optimization landscape. However, they can be very time-consuming as they suffer from significant computation and communication overheads. This work presents techniques and algorithms to accelerate second-order optimization methods on modern computing platforms.

The performance of second-order methods such as quasi-Newtons and natural gradient descent is improved on modern computing systems. The sub-sampled quasi-Newton method is accelerated on distributed memory systems and demonstrates speedups of up to 12× compared to current efficient implementations. We develop, for the first time, a distributed quasi-Newton algorithm that can achieve local superlinear convergence that uses asynchronous communications to reduce communication overheads. We develop a novel framework for implementing asynchronous communication models on cloud systems that allows practitioners to implement and dispatch asynchronous optimization methods. A scalable natural gradient descent method is developed that computes the curvature information efficiently and reduces the training of deep neural networks up to 2.1× on modern graphical processing units.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The volume of data currently generated by sensor networks, social media, and computational science has made manual data processing nearly impossible. The need to automatically process and interpret this data has driven the rapid development and deployment of machine learning algorithms and tools which have enabled progress in many critical fields [1–4]. Machine learning encompasses many different problems like regression, classification, clustering, and dimensionality reduction. All of these problems have the common task of creating models from an input dataset that can subsequently be used as predictors (i.e., accurately classify, cluster, etc.) on new data. All of these applications require some form of mathematical optimization to find the best predictor (i.e., one that minimizes the prediction error) obtained by maximizing or minimizing some objective function representing the machine learning problem being solved. Many optimization problems in machine learning including empirical risk minimization are based on processing large amounts of data as input. Due to the advances in sensing technologies and storage capabilities the size of the data we can collect and store increases in an exponential manner [5]. As a consequence, a single processor is typically not capable of processing and storing all the samples of a dataset. To solve such "big data" problems, we typically rely on modern computing resources where the data is distributed over a cluster of computing units. Due to the distributed nature of these systems, the traditional optimization methods are not suitable to solve such problems. This has made imperative the use of high-performance optimization to solve these problems.

Second-order optimization methods have gained significant traction in recent years as they accelerate the optimization process by capturing the geometry of the optimization landscape. However, their execution can be very time-consuming as they suffer from significant computation and communication overheads when running on modern computing systems [6–8]. In this thesis, we tackle the challenges of reducing the communication bottlenecks in second-order optimization methods. We investigate the communication cost of different second-order methods depending on the modern parallel platforms they are executed on and employ novel techniques to mitigate this cost. The reported performance and speedups are compared to the fastest available implementations on modern parallel systems. The first chapter is organized as follows:

Section 1.1 discusses parallel optimization methods in machine learning and briefly reviews the first-order and second-order optimization methods. Section 1.2 discusses the techniques to develop high-performance optimization methods by reducing the communication cost on modern parallel

systems. Major contributions of this thesis are discussed in section 1.3.

1.1 Parallel Optimization in Machine Learning

Optimization is one of the main pillars of machine learning where parameters of an optimization problem are computed based on observed data, i.e., an objective function is optimized by iteratively updating the model parameters until convergence. However, the speed at which datasets grow in size is strongly outpacing the evolution of the computational power of single devices, as well as their memory capacity. Therefore, parallel approaches for training machine learning models have become tremendously important [9]. In the modern parallel platforms, there are m computing resources such as processors that are connected via a bus or a network. Typically, each processor has access to a chunk of data and can communicate with some or all of the processors depending on the underlying communication. In this case, the objective function is typically defined as:

$$\min_x F(x) = \frac{1}{m} \sum_{i=1}^{m} f_i(x) \tag{1.1}$$

where $x \in \mathbb{R}^n$ is the unknown optimization parameters and f_i denotes the i-th local objective functions. When operating on modern parallel platforms, processors have to communicate with each other or through memory hierarchy to collectively converge to the optimum of the objective function Equation 1.1. As the number of processing units increases, the communication cost also increases which depends on factors such as the memory hierarchy and the underlying communication network. Because communication is often more expensive than computation, increasing the number of processing units does not always result in improved performance [10, 11]. Therefore, there exists a trade-off between computation and communication costs for high-performance optimization methods. Thus adapting such algorithms to run on modern parallel architectures in a reasonable time frame is fundamental in machine learning. Optimization methods can be categorized based on their use of curvature information of the objective function to solve Equation 1.1. We briefly review these methods in the following section.

1.1.1 Optimization Methods

Popular optimization methods can be divided into two major categories based on their use of curvature information: first-order optimization methods, which are represented by the widely used stochastic gradient methods; second-order optimization methods, in which Newton's method is a typical example.

First-order methods. First-order methods rely solely on gradient information. A typical example is the gradient descent method:

$$x \longleftarrow x - \gamma \nabla F(x) \tag{1.2}$$

where $\nabla F(x)$ is the gradient of the objective function and γ is the step size. Stochastic gradient descent is widely used in practice and has been improved in many ways. Examples of such methods include accelerated SGD [12, 13], variance reduction SGD [14], stochastic coordinate descent methods [15, 16], distributed variants of stochastic gradient descent (SGD) [17, 18], and dual coordinate ascent

algorithms [19]. The common denominator in all of these methods is that they significantly reduce the amount of local computation. But this blessing comes with an inevitable curse that they, in turn, may require a far greater number of iterations to converge. Indeed, as a result of their highly iterative nature, many of these first-order methods require synchronizations in every iteration when executing on modern parallel systems, and they must do so for many iterations.

Second-order methods. Second-order methods are among the most powerful algorithms in mathematical optimization. These methods often use a preconditioning matrix to transform the gradient before applying each step. Classically, the preconditioner is the matrix of second-order derivatives (the Hessian) in the context of exact deterministic optimization [20]. Second-order methods often have significantly better convergence properties than first-order methods. Newton method is a typical example with the following update rule:

$$x \longleftarrow x - \gamma H_x^{-1} \nabla F(x) \tag{1.3}$$

Where $H_x \in \mathbb{R}^{n \times n}$ is the Hessian of the objective function which determines the curvature information and γ is the step size. Typically computing the Hessian and its inverse is impractical due to its large size, i.e., $n \times n$, where n is the number of parameters and can be millions. Specifically, the size of typical problems prohibits their use in practice, as they require quadratic storage, i.e., $\mathcal{O}(n^2)$, and cubic computation time, i.e., $\mathcal{O}(n^3 + mn^2)$, for each gradient update. In order to reduce the overhead of computation and storage requirements, numerous works have been proposed in the prior work, some of which include Hessian-free, quasi-Newton, Gauss-Newton, and natural gradient descent methods.

Hessian-free methods use Hessian-vector products to estimate the updates in 1.3. These methods avoid computing the Hessian and its inverse directly and instead use a linear solver such as conjugate gradients [21, 22] to obtain hessian-vector products, hence, referred to as Hessian-free methods [23]. Despite making good progress on a per-iteration basis, having to run a conjugate gradient descent optimization at every iteration proved to be too slow to compete with first-order methods [24]. In quasi-Newton methods, the Hessian or its inverse is estimated with approaches such as low-rank approximations [25] or subsampling methods [26]. The most well-known method in this family is Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm which approximates the Hessian using changes in the gradients and optimization variable [25, 27]. The more recent work in this area has focused on stochastic quasi-Newton methods which were proposed and analyzed in various settings [28–30]. In particular, the limited memory variant of BFGS, L-BFGS, reduces the storage requirement and is well suited for optimization problems with a large number of variables. Gauss-Newton methods approximate the Hessian with a positive semidefinite matrix [31]. In contrast to the Newton method, the Gauss-Newton approach does not require computation or estimation of the second derivatives of the objective function, however, its implementation would impose infeasible memory requirements [32]. Hence, approximate Gauss-Newton approaches such as block-diagonal methods or low-rank approximations have been proposed in recent work [32]. The Natural Gradient Descent (NGD) method approximates the Hessian with the Fisher information matrix [33]. NGD is closely related to the Gauss-Newton method and it has been shown that the Fisher coincides with a generalized Gauss-Newton for a class of optimization problems [33, 34]. Numerous studies have proposed approximation methods to reduce the computational cost of NGD methods so that NGD can be used in large-scale models with many parameters, especially, in deep neural networks (DNNs).

Quasi-Newton and natural gradient descent methods are the main focus of this thesis. In the following section, the performance bottleneck of optimization methods on modern computing systems is discussed and the techniques to mitigate this bottleneck are reviewed.

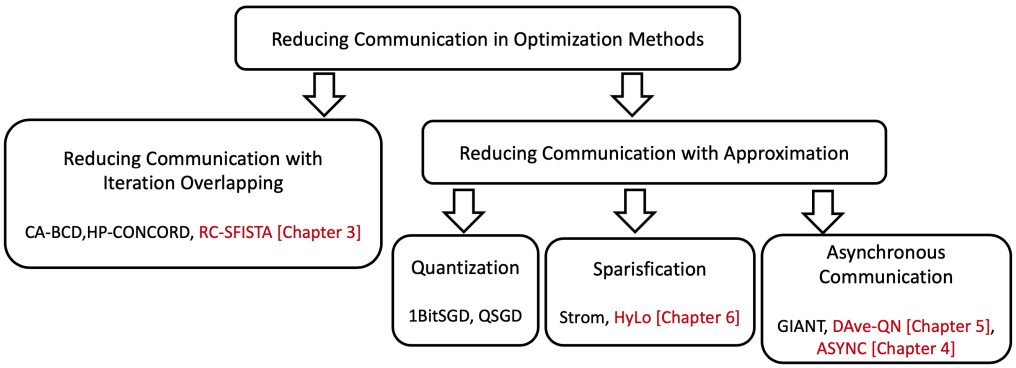


Figure 1.1: Different techniques for reducing communication in optimization methods.

1.2 High-Performance Optimization Methods

Due to the increase in size and complexity of modern datasets, the distributed storage of these datasets as well as accompanying parallel solution methods are either necessary or at least highly desirable [35]. The main drawback of big data optimization algorithms comes from the potentially high communication cost. Communication is defined as data movements in the memory hierarchy of a single processor or between different processors. On modern computer architectures, the communication cost is often orders of magnitude larger than the computation cost, i.e., the cost of floating-point operations (flops), and this gap is increasing [36]. Therefore, there is a trade-off between communication and computation cost which varies for different optimization methods. Recent works aim to leverage this trade-off and develop new methods that run faster on modern computing platforms. This work, as shown in Figure 1.1, can be categorized into two groups: *i)* iteration overlapping methods that reformulate the classical optimization methods in exact arithmetic without changing the convergence behavior and *ii)* approximate methods that reduce the communication overhead by introducing some type of error in the computations.

1.2.1 Reducing Communication with Iteration Overlapping Methods

Iteration overlapping methods reformulate the existing algorithms by overlapping a set of iterations to reduce the communication overheads. These techniques reduce the number of communication rounds, i.e., the number of messages sent among processors, at the expense of extra computations. These methods do not involve any approximations if one neglects the numerical round-offs. These methods, which are also known as communication-avoiding (CA) methods, were first introduced in numerical linear algebra for solving linear systems [37]. In these works, CA techniques reduce communication by taking k-steps of the iterative solvers at the same time; data will be on fast memory while the k-steps of the iterative solver are taken at the same time, reducing memory references considerably.

Recently, numerous works have adopted overlapping techniques to reduce communication overheads in optimization methods. These methods attempt to compute multiple iterations of the optimization problem on each processor before any communication happens. In the first line of work, coordinate descent methods [38] are accelerated by computing multiple blocks of updates for regularized least square problems. These methods communicate every s iteration, where s is a tuning parameter that controls the trade-off between computation and communication cost. Similar attempts have been made for kernel methods and support vector machines problems [39]. In the second line of work, the data is partially replicated on different processors and the method is reformulated to cut down on communication at the expense of redundant computations and extra space, such as sparse inverse covariance estimation [40]. The amount of replication can change the performance of these methods and it governs the communication-computation trade-off. To the best of our knowledge, there has not been any work on iteration overlapping methods for second-order optimization methods.

1.2.2 Reducing Communication with Approximation

Approximate optimization methods attempt to mitigate the communication overheads in two ways: *i)* reducing the amount of data sent either between levels of a memory hierarchy or between processors over a network. These works can be categorized as quantization and sparsification methods. *ii)* reducing processors' idle times by allowing asynchronous communications. We briefly review these methods in the following.

Quantization. Quantization methods can reduce the amount of data communicated by quantizing the gradient with a fixed number of bits per problem dimension [41–43]. These methods typically quantize each coordinate of the input vector separately. For each coordinate, the quantization levels are distributed either uniformly [41] or non-uniformly [44] within the range of the input vector. In the extreme case, each coordinate of the gradient vector is reduced to its sign which has been experimentally observed to preserve convergence under certain conditions [45] while in general, it can fail due to biased quantization error. Stochastic quantization methods guarantee convergence by allowing for unbiased quantization error. The convergence of quantized methods is usually analyzed by relating the variance of the quantized (thus noisy) gradient to the bit rate. [41] showed that processors can adjust the number of bits sent per iteration, at the cost of possibly higher variance. Moreover, the communication cost is further reduced using an encoding technique such as Elias encoding [41]. These works are primarily targeted toward first-order methods and to our knowledge, there is limited work that applies quantization to second-order optimization methods [46].

Sparsification. Sparsification methods reduce communication by only selecting an "important" sparse subset of the components to broadcast at each iteration and accumulating the rest locally [47,48]. These methods often produce non-structured vectors after sparsifying the input vector. For example, top-k sparsification only preserves the k coordinates of the largest magnitude and sends them with full precision in which the error is controlled by a user-specific accuracy parameter, e.g., k. However, similar to quantization techniques, the convergence analysis of these methods is challenging. Recent works try to remedy the convergence failure of sparsification methods by appropriately amplifying the remaining coordinates to ensure the unbiasedness of the sparsified stochastic gradient [49].

Sparsification techniques can also induce structured sparsity on the gradient or curvature matrix with methods such as row-wise sparsification [50], group sparsification [51], and block-diagonal approximations [52]. Structured sparsification appears in the context of neural networks where

gradient and curvature are represented with weight matrices or high-order tensors. For instance, row-wise sparsification only selects one row of the weight matrix where the max gradient magnitude lies [50]. Block-diagonal approximations are mainly used for second-order methods to approximate the Hessian or Fisher information matrix with a block-diagonal matrix where each block corresponds to a specific set of coordinates or weight matrices [52, 53]. Many works leverage the communication-computation trade-off and further approximate each block at the expense of accuracy loss. For instance, KFAC [54] first applies a block-diagonal structure to the Fisher matrix and then approximates every block with Kronecker factorization.

Asynchronous communication. Asynchronous updates provide an alternative solution to reduce the communication overhead to a certain amount [55–57]. In synchronous optimization methods, processors compute the local gradients (or preconditioned gradients) of the objective function which is then aggregated based on a synchronization mechanism. Therefore, in the presence of slow processors, i.e., stragglers, all processors have to wait for the slowest one at the synchronization point. In contrast, the asynchronous optimization methods relax the communication constraints by allowing the processors to operate independently, which results in better resource usage and faster implementations. However, the processors can submit a low-quality update of parameters and, thereby, it can lead to poor convergence properties [58, 59]. Therefore, the careful convergence analysis of asynchronous parallel methods is arduous, due to the specific difficulties associated with this setup. For instance, while iterates can usually be clearly defined as a function of the previous iterates, this is no longer the case when the method is asynchronous because several processors are writing updates concurrently.

Much work has been devoted recently to proposing and analyzing asynchronous parallel variants of synchronous optimization methods, both first-order [55, 60] and second-order [56, 61], which mainly focus on improving their convergence. These methods can be categorized based on the distributed platform: shared memory systems, i.e., multiple cores of a processor can write to the parameters independently, and distributed memory systems, i.e., processors' have their copy of the parameters. One of the earliest examples of first-order methods developed on shared memory systems is HogWild, an asynchronous variant of stochastic gradient descent with constant step size [62], later improved by approaches such as variance reduction techniques [63]. Recent works also propose several asynchronous algorithms with linear convergence properties on distributed memory systems [64, 65]. Among second-order optimization algorithms, asynchronous quasi-Newton methods have been recently proposed for both shared memory [66] and distributed memory [56, 66, 67] systems with linear convergence guarantees.

Recent advancements in distributed asynchronous optimization demand a wider range of asynchronous communication models [68–71], often defined by the user. These models adaptively adjust their parameters during the optimization process to optimize their resource usage. Moreover, distributed optimization methods operate on batches of data and thus have to be implemented in cluster-computing engines with a bulk (coarse-grained) computation model. Recently, numerous distributed frameworks have been developed to simplify implementing and deploying asynchronous optimization methods. Some works have modified existing coarse-grained synchronous frameworks such as Spark [72] to support asynchronous optimization methods by adding an extra communication layer [73]. In the second line of work, several coarse-grained machine learning engines have adopted the parameter server [74] architecture to implement asynchronous communication between nodes with

push-pull operations [75, 76]. Other distributed parameter server frameworks such as PyTorch [77] and TensorFlow [78] are specialized for deep learning applications.

1.3 Thesis Contributions

In this thesis, the second-order optimization methods are accelerated on modern computing platforms by reducing their communication overheads. We focus on two categories of second-order methods, quasi-Newton approaches, and natural gradient descent. The quasi-Newton methods and in particular, subsampled and BFGS types, are introduced in chapters 3 and 4 respectively. In chapter 3, we reduce the communication overhead of a subsampled quasi-Newton method using iteration overlapping techniques resulting in a scalable optimization method that can achieve significant speedups. In chapter 4, a BFGS type quasi-Newton method is accelerated using asynchronous communications with local superlinear convergence guarantees. In order to support different asynchronous executions on modern parallel platforms and specifically cloud environments, we develop a robust and easy-to-use framework to implement asynchronous optimization algorithms, studied in chapter 5. Finally, chapter 6 proposes a novel natural gradient method that uses structured sparsification to approximate the Fisher information matrix in deep neural networks. The major contributions of this thesis are as follows:

- A novel stochastic variance-reduced algorithm with low communication overheads, called RC-SFISTA, is introduced that reduces the communication and computation cost of proximal Newton methods. The convergence analysis of the proposed method shows the same convergence rate as its deterministic formulation. Other techniques such as *Hessian-reuse* are presented which control the trade-off between communication and computation cost.

- The first distributed asynchronous algorithm with superlinear convergence guarantees for master-slave architectures, called DAve-QN, is presented. The proposed algorithm is communication-efficient in the sense that at every iteration the master node and slaves communicate small size vectors.

- A novel framework for machine learning practitioners to implement and dispatch asynchronous machine learning applications with custom communication models and a robust programming interface is designed and developed on cloud and distributed platforms. The proposed framework supports optimization methods that operate on *history* by using an efficient history recovery strategy.

- An efficient natural gradient descent method called HyLo is presented which speeds up the training of neural networks on distributed platforms. The communication and computation cost of training is reduced by factorizing the Fisher information matrix using novel algorithms such as Khatri-Rao based interpolative decomposition and sampling. As a result, the training time of large-scale deep neural networks is significantly accelerated.

1.4 Summary

The main objective of this work and the importance of second-order optimization for machine learning applications were discussed in this chapter. The performance bottleneck of parallel optimization methods was introduced and the existing techniques to tackle this problem were discussed.

1.5 Thesis Outline

The rest of this thesis is organized as follows:

- Chapter 2: gives the background on the notation, empirical risk minimization problem in machine learning, different regularizations, and least-squares problem. Optimization methods and, in particular, second-order algorithms used throughout this thesis are also detailed in this chapter. Afterward, the distributed platforms used in this work are classified and explained.
- Chapter 3: accelerates second-order proximal Newton methods used for solving the regularized least-squares problem on distributed memory systems by employing iteration overlapping techniques. The performance of the proposed approach is compared against optimized implementations for proximal Newton methods.
- Chapter 4: introduces a novel second-order quasi-Newton optimization method for solving the empirical risk minimization problem that uses asynchronous communications. The proposed method improves current asynchronous second-order methods in convergence rate and communication cost. It is the first distributed asynchronous method that can achieve local superlinear convergence rate and is also communication-efficient in the sense that it only requires communicating a few vectors per iteration.
- Chapter 5: introduces a framework to implement a wide range of asynchronous optimization methods on distributed cloud systems. The proposed framework supports first and second-order methods that require a custom user-defined communication model among processors. The robust programming model and ease-of-implementation on distributed platforms are also described in this chapter.
- Chapter 6: accelerates the natural gradient descent method for training deep neural networks by sparsifying the computations. The proposed NGD method uses a computationally-efficient low-rank factorization to approximate the Fisher information matrix. Finally, its convergence properties and performance are compared against optimized implementations of other second-order methods on image classification and segmentation tasks.

Chapter 2

Background

In this chapter, the expected risk minimization problem as the standard framework for many machine learning applications is introduced. Linear and logistic regression are given as popular examples for analyzing optimization method performances. Afterward, this chapter reviews the first and second-order optimization methods that are used throughout this thesis to derive efficient optimization methods. Finally, this chapter ends with the classification of the modern computing platforms used in this work and the practical challenges in deriving communication-efficient methods on each platform. The bold letters are used to denote a vector and $\|.\|$ refers to a vector norm. When writing optimization problems, argmin is used to show that the solution to the optimization problem is the argument that minimizes the optimization problem. log refers to logarithms with base 2.

2.1 Empirical Risk Minimization

A large fraction of machine learning applications require the solution of an empirical risk minimization problem (ERM) which is expressed as the minimization of a sum of individual costs associated with individual elements of a training set. The setting is as follows: Given n machines, each machine has access to m_i samples $\{\xi_{i,j}, y_{i,j}\}_{j=1}^{m_i}$ for $i = 1, 2, \ldots, n$. The samples $\xi_{i,j}$ are random variables supported on a set $\mathcal{P} \subset \mathbb{R}^d$ and $y_{i,j}$ is the corresponding response or label. Each machine has a loss function that is averaged over the local dataset:

$$f_i(\boldsymbol{x}) = \frac{1}{m_i} \sum_{j=1}^{m_i} \phi(\boldsymbol{x}, \xi_{i,j}, y_{i,j}) \tag{2.1}$$

where the function $\phi : \mathbb{R}^p \times \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}$ is convex in $\boldsymbol{x}$ for each $\xi \in \mathbb{R}^d$ and $y \in \mathbb{R}$ fixed. The goal is to develop communication-efficient distributed algorithms to minimize the overall empirical loss defined by

$$\boldsymbol{x}^* \coloneqq \operatorname*{argmin}_{\boldsymbol{x} \in \mathbb{R}^p} f(\boldsymbol{x}) \coloneqq \operatorname*{argmin}_{\boldsymbol{x} \in \mathbb{R}^p} \frac{1}{n} \sum_{i=1}^{n} f_i(\boldsymbol{x}). \tag{2.2}$$

The examples of least-squares or binary logistic regression are both widely useful and provide plenty of intuition for this setting, so they often serve as the working illustration. Still, more complex problems fall into the same category of ERM. These include multiclass classification and structured

prediction problems that arise, for instance in image classification and natural language processing, or applications in computational biology. Without loss of generality, throughout this section, we assume each machine has access to one sample to simplify the notations.

2.1.1 Regularization

Machine learning models such as linear regression can learn complicated relationships between their inputs and outputs. With limited training data, however, many of these complicated relationships will be the result of sampling noise, so they will exist in the training set but not in real test data even if it is drawn from the same distribution [79]. This leads to overfitting and many methods have been developed for reducing it. A typical approach is introducing penalties of various kinds, such as L1 and L2 regularization.

L2-regularization. In this approach, we use the Euclidean distance as the penalty term. This involves adding the term $\frac{\lambda}{2}\|\boldsymbol{x}\|_2^2$ to the loss function, where λ is a hyperparameter. In some cases, the theoretical motivation for using this type of regularization is clear. For example, in the context of linear regression, L2 regularization increases the bias of the learned parameters while reducing their variance across instantiations of the training data; in other words, it is a manifestation of the bias-variance trade-off [80].

L1-regularization. While L2 regularization is an effective means of achieving numerical stability and increasing predictive performance, it does not address another problem with least-squares estimates, the parsimony of the model, and the interpretability of the coefficient values. While the size of the coefficient values is bounded, minimizing the ERM with a penalty on the L2-norm does not encourage sparsity and the resulting models typically have non-zero values associated with all coefficients. It has been proposed that, rather than simply achieving the goal of 'shrinking' the coefficients, higher λ values for the L2 penalty force the coefficients to be more similar to each other in order to minimize their joint 2-norm [81]. A recent trend has been to replace the L2-norm with an L1-norm. In this case, the term $\lambda\|\boldsymbol{x}\|_1$ is added to the objective function. This L1 regularization has many of the beneficial properties of L2 regularization, but yields sparse models that are more easily interpreted [82].

2.1.2 Least-Squares Regression

The least-squares problem is widely used in deriving and experimenting with optimization methods. The empirical loss is defined as:

$$f(\boldsymbol{x}) \coloneqq \frac{1}{2n}\|A\boldsymbol{x} - \boldsymbol{b}\|_2^2 \tag{2.3}$$

where $A \in \mathbb{R}^{n\times p}$ is the matrix containing the training samples where each row represents a sample and $\boldsymbol{b} \in \mathbb{R}^n$ is the vector of labels.

2.1.3 Logistic Regression

Logistic regression (LR) is a standard probabilistic statistical classification model that has been extensively used across disciplines such as computer vision, marketing, and social sciences, to name a few [83]. Different from linear regression, the outcome of LR on one sample is the probability that it

is positive or negative, where the probability depends on a linear measure of the sample. Therefore, LR is widely used for classification [84]. The empirical loss for the binary logistic regression is defined as:

$$f(\boldsymbol{x}) \coloneqq \frac{1}{n}\sum_{j=1}^{n}\log(1+\exp(-b_j a_j^\top \boldsymbol{x})) \tag{2.4}$$

where $a_j \in \mathbb{R}^p$ are the feature vectors and $b_j \in \{-1,+1\}$ are the labels. The multi-class logistic regression can be stated using 2.4 where the labels are $b_j \in \mathbb{R}$.

2.2 First-Order Optimization Algorithms

Over the past decades, several first-order methods have been brought up and widely used in practice [85, 86]. The primitive method is simply gradient descent, which updates the parameters of the model in the opposite direction of the gradient of the objective function:

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \gamma \nabla f(\boldsymbol{x}_k) \tag{2.5}$$

where γ is the learning rate. Gradient descent (GD) has been one of the most commonly used first-order methods. The most computationally taxing part of 2.5 is calculating the gradient which is done in an exact matter using the entire training data set. However, the gradient can instead be approximated using a smaller sample (mini-batch) of the data set and thus reduce the computation needed. This leads to stochastic gradient methods.

Stochastic gradient descent. Since gradient descent has high computational complexity in each iteration for large-scale data and does not allow the online update, stochastic gradient descent (SGD) was proposed [85, 87]. The idea of stochastic gradient descent is using one sample randomly to update the gradient per iteration, instead of directly calculating the exact value of the gradient. The stochastic gradient is an unbiased estimate of the real gradient [85]. The cost of the stochastic gradient descent algorithm is independent of sample numbers and can achieve sublinear convergence speed [87]. In this work, we consider the mini-batch version of SGD in which m samples are used to estimate the gradient. Therefore, the update for SGD is defined as:

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \gamma \hat{\nabla} f(\boldsymbol{x}_k) \tag{2.6}$$

where $\hat{\nabla} f(\boldsymbol{x}_k)$ is the estimated gradient using m random samples.

Although practical and effective, GD converges slowly in many applications. To accelerate its convergence, there has been a surge of interest in accelerated gradient methods, where "accelerated" means that the convergence rate can be improved without much stronger assumptions or significant additional computational burden [88]. Nesterov-accelerated methods such as FISTA and Variance-reduced methods such as SVRG, SAG, and, SAGA are such examples.

2.3 Second-Order Optimization Algorithms

Second-order optimization methods use second-order information to construct updates that account for the curvature of the objective function [89]. Both the first-order derivative (gradient) and

second-order derivative (Hessian matrix) are used to approximate the objective function with a quadratic function, and then minimize it [87]. The general update rule for second-order methods can be written as :

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \gamma {H_k}^{-1} \nabla f(\boldsymbol{x}_k) \tag{2.7}$$

where $H(x_k) \in \mathbb{R}^{p \times p}$ is the Hessian matrix at iteration k. Hessian matrix is very large in practice and therefore it is infeasible to directly compute and inverse it. Various approximations to the Hessian matrix have been proposed to help alleviate this problem. We briefly review the most related methods in this section.

2.3.1 Quasi-Newton Methods

The basic idea of the quasi-Newton method is to use a positive definite matrix to approximate the inverse of the Hessian to reduce its costly computations.

Sub-Sampled Quasi-Newton Methods

One approach is to use sub-sampling techniques, where Hessian matrix H is constructed based on a randomly selected set of data points:

$$H_k = \frac{1}{|S|} \sum_{i \in S} \nabla^2 f_i(\boldsymbol{x}_k) \tag{2.8}$$

where S contains the randomly sampled data points. These methods have been shown effective in accelerating some machine learning applications [90] and have been further studied in recent work [91]. In this work, a subsampled quasi-Newton approach is used in the derivation of communication-efficient second-order methods in chapter 3.

BFGS Methods

The most popular quasi-Newton method is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm which is well-liked for its robustness and its self-correcting properties, as well as for the superlinear convergence it achieves [92, 93]. In general, BFGS methods approximate the Hessian matrix using the following equation:

$$H_{k+1} = H_k - \frac{H_k s_k s_k^\top H_k}{s_k^\top H_k s_k} + \frac{y_k y_k^\top}{y_k^\top s_k} \tag{2.9}$$

where s_k and y_k are vectors that are obtained from changes in gradient and optimization variables respectively.

One of the disadvantages of the BFGS algorithm is the high memory requirements. Limited Memory BFGS (L-BFGS) is a variant of BFGS, which uses only the recent iterates and gradients to construct the approximate Hessian, providing improvement in terms of memory usage. The L-BFGS algorithm avoids storing the sequential approximations of the Hessian matrix which allows it to generalize well to the high-dimensional setting. More specifically, the algorithm stores information about the spatial displacement and the change in gradient and uses them to estimate a search direction without storing or computing the Hessian explicitly [94]. Chapter 4 provides additional

details on the BFGS algorithm and develops a distributed asynchronous quasi-Newton algorithm that can achieve superlinear convergence.

2.3.2 Natural Gradient Descent Methods

Natural gradient descent (NGD), pioneered by Amari [95], is an optimization method traditionally motivated from the perspective of information geometry and works well for many applications as an alternative to stochastic gradient descent [52]. The update rule for NGD is defined as:

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \gamma F_k^{-1} \nabla f(\boldsymbol{x}_k) \tag{2.10}$$

where F_k is the Fisher information matrix at iteration k. Fisher matrix is defined as:

$$F = \mathbb{E}_{P_{\xi,y}}[\nabla \log(p_{\xi,y}|\boldsymbol{x}) \nabla \log(p_{\xi,y}|\boldsymbol{x})^\top] \tag{2.11}$$

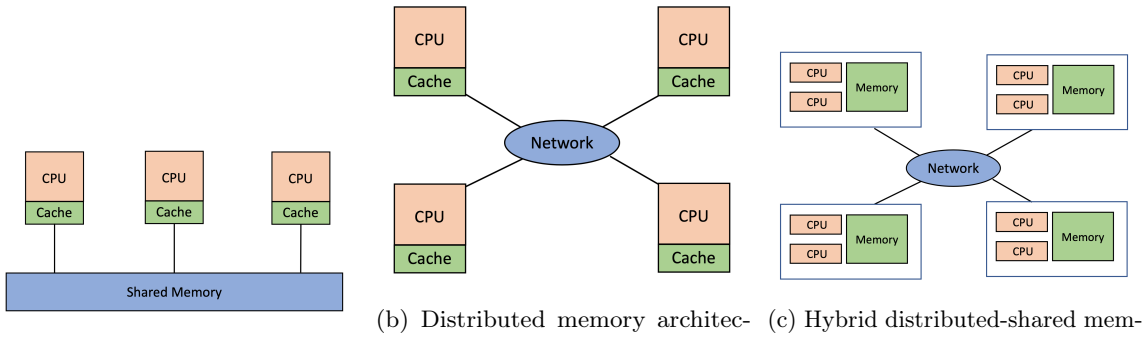
where gradients are taken w.r.t $\boldsymbol{x}$ and $P_{\xi,y}(\boldsymbol{x})$ is the learned distribution whose density is $p_{\xi,y}(\boldsymbol{x})$. Fisher and Hessian are closely related and in fact, Fisher can be cast as an approximation of the Hessian in different ways [52]. Specifically, the Fisher information matrix describes the local metric of the objective function surface concerning the KL-divergence function [96] and in many important cases, it is shown to be equivalent to the generalized Gauss-Newton approach, but with certain properties that favor its use over Gauss-Newton methods [97]. Since it is expensive to estimate the Fisher information matrix and calculate its inverse, recent work has proposed different approximations such as block-diagonal [98, 99] for the Fisher matrix, especially for overparametrized neural networks. Chapter 6 provides additional details on NGD and Fisher matrix and proposes a communication-efficient method for training neural networks.

2.3.3 Gauss-Newton Methods

The classical Gauss-Newton matrix (or more simply the Gauss-Newton matrix) is the curvature matrix G which arises in the Gauss-Newton method for non-linear least-squares problems [52]. The Gauss-Newton matrix is an approximation of the Hessian matrix that avoids the trouble of negative curvature [100]. Typically, the Gauss-Newton method requires $f(\boldsymbol{x})$ to be expressed as a composition of two functions written as $f(\boldsymbol{x}) = Q(F(\boldsymbol{x}))$ where Q is convex. Then the generalized Gauss-Newton matrix can be written as:

$$G = J^\top H_Q J \tag{2.12}$$

where J is the Jacobian of the function F and H_Q is the Hessian of the function Q. Gauss-Newton method is closely related to natural gradient descent and it has been shown that the generalized Gauss-Newton matrix is equivalent to the Fisher information matrix if the predictive distribution is in the exponential family, such as categorical distribution (for classification) or Gaussian distribution (for regression) [101].



(a) Shared memory architecture. (b) Distributed memory architecture. (c) Hybrid distributed-shared memory architecture.

Figure 2.1: Classification of the modern parallel systems based on the memory architecture.

2.4 Modern Parallel Architectures

Based on the memory configuration, modern parallel systems are classified into three main categories: shared memory, distributed memory, and hybrid models which are shown in Figure 2.1.

Shared memory architectures. A shared-memory system, shown in Figure 2.1a, makes a global physical memory equally accessible to all processors. These systems offer a general and convenient programming model that enables simple data sharing through a uniform mechanism of reading and writing shared structures in the common memory [102]. The shared memory space can either be accessed uniformly by all processors (Uniform Memory Access-UMA) or have a non-uniform access pattern (Non-Uniform Memory Access-NUMA). A common example of shared memory systems is Graphical Processing Units (GPUs). GPUs have hundreds of processor cores and thousands of threads running concurrently on these cores that all have access to the shared memory, also known as global memory, in the GPU. However, shared-memory multiprocessors typically suffer from increased contention and longer latencies in accessing the shared memory, which degrades peak performance and limits scalability compared to distributed systems [102].

Distributed memory architectures. In distributed memory systems, as shown in Figure 2.1b, inter-processor memories are connected using a network. Each processor has a separate memory space that is not mapped to others. If a processor requires data located in another memory space, the programmer has to explicitly manage how data is transferred. Transferring data over the network is costly and depends on the type of network. The communication cost includes the "bandwidth cost", i.e., the number of words sent between processors over a network, and the "latency cost", i.e., the number of messages sent, where a message either consists of a group of contiguous words being sent. Distributed memory systems are more popular in machine learning as they scale up easily and can be used in applications with very large datasets. These applications often process the data with the "data-parallel" approach which is based on partitioning the data into several blocks and running multiple copies of the same program concurrently, each running on a different data block, thus the name of the paradigm [103].

Hybrid distributed-shared memory architectures. Hybrid memory models are a combination of shared and distributed memory architectures, as shown in Figure 2.1c. By connecting shared memory processors through a network, a hybrid memory model is constructed. The communication cost in these systems involves both the cost of accessing the shared memory and transferring data via

the network such as Multi-GPUs systems where multiple GPUs are connected via a network. These systems are used when the data to be processed does not fit in the global memory of a single GPU.

Chapter 3

Reducing Communication in Proximal Newton Methods

In this chapter, the communication cost of a Newton method is reduced by reformulating the optimization algorithm using iteration overlapping techniques. Our proposed method belongs to the class of *Proximal Newton (PN)* methods which are used in solving optimization methods with sparsity constraints such as L1 regularized least-squares. PN methods are a class of second-order optimization methods that use a first-order inner solver to solve a subproblem iteratively. Afterward, the solution of the subproblem is aggregated by communicating data amongst processors. The performance of PN methods used for solving an optimization problem is limited by the performance of the inner solver used in these algorithms. Our proposed method uses overlapping techniques to reduce the total number of communication rounds at the cost of computing multiple iterations of the optimization problem. The detailed contributions of this chapter are as follows:

- A novel stochastic variance-reduced formulation of the FISTA algorithm, called SFISTA, is introduced that reduces the computation complexity of PN methods. A convergence theorem is provided for the stochastic variance-reduced method which shows the same convergence rate as its deterministic formulation.
- Iterations in the proposed SFISTA method are overlapped to reduce communication rounds and latency costs without increasing the number of messages.
- A Hessian-reuse algorithm is developed that enables the reuse of data when solving a subproblem in SFISTA. The parameter S in the Hessian-reuse methods enables finding an efficient trade-off between computational complexity and communication costs in the algorithm.
- The upper bounds for parameters k in iteration-overlapping and S in the Hessian-reuse method are provided with respect to machine and algorithm specifications.
- RC-SFISTA is implemented on both MPI and Spark and is compared to the state-of-the-art framework ProxCoCoA. RC-SFISTA outperforms the classical method up to 12× with MPI on 256 processors. We also demonstrate that RC-SFISTA performs better than ProxCoCoA up to 12× on 256 workers for the tested datasets.

3.1 Introduction

A popular approach to estimating parameters in convex optimization problems is solving a regularized least-squares problem [104–106] using proximal methods [107,108]. A common approach to improving the performance of optimization methods on distributed platforms is to reformulate the algorithm to reduce the number of iterations needed to reach the optimal solution.Works such as [109, 110] attempt to improve the convergence properties by applying different solvers to locally stored data. CoCoA [109] uses a local solver on each machine and shares information between solvers with highly flexible communication schemes. ProxCoCoA [106], GLMNET [111], and BLITZ [112] propose communication-efficient algorithms for proximal methods, however, the methods do not necessarily preserve the exact arithmetic of the conventional algorithm.

Iteration-overlapping techniques [113, 114] reduce the overall communication cost in optimization methods by unrolling iterations in the algorithm to break the dependencies between vector updates. These works produce a solution identical to that of the conventional algorithm in exact arithmetic. k-step Krylov solvers [115, 116] compute k basis vectors at once by unrolling k iterations of the standard algorithm. P-packSVM [117] proposes an SGD-based algorithm for support vector machines that communicates every k iterations. CA-BCD [38] reduces communication for the class of l2-regularization least-squares problems by rearranging the computations to execute k iterations per *communication round*, i.e., the total number of times that processors exchange *messages* over a network. SA-accBCD [118] applies a similar approach to proximal least-squares problems. While these works reduce communication costs by reducing the number of communication rounds, they increase the amount of communicated data at each round, i.e., *message size*. Also, since these methods are deterministic, every iteration operates on all the data which leads to high computational complexity for *overdetermined* problems.

Overdetermined problems involving a large number of data points are often solved with random sampling, i.e., a stochastic approach, to reduce the overall computational complexity of the algorithm by reducing the size of data that the algorithm operates on. Examples of such work include stochastic formulations of gradient descent [119, 120] and proximal gradient methods [121]. However, the rate of convergence of a basic stochastic method is slower than that of the deterministic algorithm due to the variance introduced by random sampling [122]. Stochastic proximal methods such as Acc-Prox-SVRG [123] and Prox-SVRG [124] propose variance-reduction techniques to overcome this challenge. Even though the computational complexity of stochastic optimization methods is lower than deterministic formulations, the performance of stochastic methods is often bound by data communication required at each iteration of the algorithm.

3.2 Background

This section introduces PN methods for solving a class of optimization problems that arise frequently in machine learning applications. The performance model used in this work to evaluate the effectiveness of the proposed reformulations will also be discussed.

3.2.1 The Proximal Newton Method

Composite optimization problems arise frequently in machine learning and data analytics applications. For example, consider:

$$\min_{w\in\mathbb{R}^d} \; F(w) \equiv f(w) + g(w) \tag{3.1}$$

where $g : \mathbb{R}^d \to \mathbb{R}$ is a continuous, convex, possibly non-smooth function and $f : \mathbb{R}^d \to \mathbb{R}$ is a convex function, twice continuously differentiable, with an L-Lipschitz gradient, expressed as:

$$f(w) = \frac{1}{m}\sum_{i=1}^{m} f_i(w) \tag{3.2}$$

This is a general class of problems that includes numerous machine learning problems including logistic regression and regularized least-squares problems, or more general *empirical risk minimization* problems.

PN methods shown in Algorithm 3.1 are used to solve the optimization problem in (3.1). These methods could be seen as a generalization of the classical proximal gradient methods where the curvature of the function, i.e., Hessian, is used to select a search direction [125]. PN methods define a subproblem at each iteration which can be minimized using a first-order inner solver shown in line 4 of Algorithm 3.1. First-order methods that use *proximal mapping* to handle the non-smooth segment of the objective function in (3.1) are very popular and perform well in practice [126, 127].

Algorithm 3.1 Proximal Newton Method

1 **Input:** w_0, $\{\gamma_n\}$
2 **repeat**
3 Update H_n, an approximation of Hessian
4 $z_n = \underset{y}{\operatorname{argmin}} \; \frac{1}{2}(y - w_n)^T \boldsymbol{H_n}(y - w_n) + \nabla f(w_n)^T(y - w_n) + g(y)$ } Inner Solver
5 $\Delta w_n = z_n - w_n$
6 $w_{n+1} = w_n + \gamma_n \Delta w_n$
until *Stopping conditions are satisfied*

L1-regularized least-squares problem. The general problem in (3.1) can represent a large class of regression problems. In particular, we focus on the l1-regularized least-squares problem:

$$f(w) = \frac{1}{2m}\sum_{i=1}^{m}(x_i^T w - y_i)^2, \quad g(w) = \lambda\|w\|_1 \tag{3.3}$$

where $x_i \in \mathbb{R}^d$ is the i-th data point and $y_i \in \mathbb{R}$ is the corresponding label. We set $X = [x_1, ..., x_m] \in \mathbb{R}^{d\times m}$ as the input data matrix, where rows are the features and columns are the samples, and $y \in \mathbb{R}^m$ holds the labels. $w \in \mathbb{R}^d$ is the optimization variable, and $\lambda \in \mathbb{R}$ is the regularization (penalty) parameter. In this case, the gradient and Hessian of $f(w)$ is given by:

$$\nabla f(w) = \frac{1}{m}(XX^T w - Xy) \quad H = \frac{1}{m}XX^T \tag{3.4}$$

where $H \in \mathbb{R}^{d\times d}$ is the Hessian and with defining $R \in \mathbb{R}^d$ as $R = \frac{1}{m}Xy$, for the l1-regularized

least-squares problem the gradient of f will be

$$\nabla f(w) = Hw - R \tag{3.5}$$

3.2.2 The Inner Solver

The subproblem in PN methods can be solved using a first-order method. While inner solvers like coordinate descent [128] are used in PN methods, this work uses FISTA which is the most popular method in the family of accelerated proximal methods. FISTA has the same convergence rate as the accelerated coordinate descent methods [125]. The FISTA algorithm is shown in Algorithm 3.2 where the proximal mapping is defined as:

Algorithm 3.2 FISTA

Input: $w_0 = w_{-1} = \mathbf{0} \in \mathbb{R}^d$, $t_0 = 1$, γ.
for $n = 1, ..., N$ **do**
  $t_n = \frac{1+\sqrt{1+t_{n-1}^2}}{2}$
  $v_n = w_{n-1} + \frac{t_{n-1}-1}{t_n}(w_{n-1} - w_{n-2})$
  $w_n = \text{Prox}_\gamma(v_n - \gamma \nabla f(v_n))$
output w_N

$$\text{Prox}_\gamma(w) = \underset{x}{\text{argmin}}\{\frac{1}{2\gamma}\|x - w\|^2 + g(x)\} \tag{3.6}$$

In this work, we use FISTA as an inner solver for PN methods. We focus on optimizing the performance of FISTA for l1-regularized least-squares problems and demonstrate that improving the inner solver performance improves the performance of the PN method by reducing its overall computation and communication cost.

3.2.3 Performance Model

This work proposes novel formulations of the PN solvers to improve their performance when executed on distributed hardware platforms. The following elaborates on the performance model used to demonstrate the effectiveness of the proposed method. The cost of an algorithm includes arithmetic and communication. Traditionally, algorithms have been analyzed with floating-point operation costs. However, communication costs are essential in analyzing algorithms in large-scale simulations [129]. The cost of floating-point operations and communication, including bandwidth and latency, can be combined to obtain the performance model. In this work, we use a simplified model known as the $\alpha-\beta$ model [130]. In distributed-memory settings, communication cost includes the "bandwidth cost", i.e., the number of words sent either between levels of a memory hierarchy or between processors over a network, and the "latency cost", i.e., the number of messages sent, where a message either consists of a group of contiguous words being sent or is used for interprocess synchronization. The total execution time of an algorithm is:

$$T = \gamma F + \alpha L + \beta W \tag{3.7}$$

where T is the overall execution time and γ, α, and β are machine-specific parameters that represent the cost of one floating-point operation, the cost of sending a message, and the cost of moving a word. Parameters F, L, and W represent the number of flops, the number of messages communicated between processors, and the number of words moved respectively.

3.3 The Reduced-Communication Stochastic FISTA (RC-SFISTA)

We propose RC-SFISTA that improves the performance of the FISTA algorithm as well as PN methods on distributed platforms. RC-SFISTA is developed to maintain an efficient trade-off between computation and communication costs in PN methods. We first introduce a novel variance-reduced stochastic formulation of FISTA (SFISTA) to reduce its computational complexity with random sampling. The communication cost of the formulated SFISTA is then reduced in a subsequent step with iteration-overlapping. A Hessian-reuse method is also proposed that provides an effective trade-off between computational complexity and communication costs for the inner solvers in SFISTA. The extension and applicability of the proposed RC-SFISTA method for second-order optimization methods, specifically PN methods, are elaborated at the end.

3.3.1 Reducing Computational Complexity with a Stochastic Formulation

We reduce the computational complexity of FISTA with random sampling at each iteration to significantly decrease the number of floating-point operations in the algorithm. The stochastic variance-reduced FISTA (SFISTA) algorithm is developed for the general optimization problem in (3.1) and afterward for the l1-regularized least-squares problem. FISTA is made stochastic by estimating the gradient in line 5 of Algorithm 3.2 with:

$$\nabla \hat{f}(v_n) = \frac{1}{\bar{m}} \sum_{i \in \mathbb{I}_n} \nabla f_i(v_n) \tag{3.8}$$

where $\mathbb{I}_n$ is a subset of size $\bar{m} = \lfloor bm \rfloor$ from $1, ..., m$ chosen uniformly at random and $0 < b < 1$ is the sampling rate. Even though estimating the gradient in (3.8) reduces the computational complexity of FISTA, the convergence rate of the stochastic method is slower due to the variance introduced by sampling. Therefore, we use a variance-reduction method where the gradient is estimated by:

$$\nabla \hat{f}(v_n) = \frac{1}{\bar{m}} \left(\sum_{i \in \mathbb{I}_n} \nabla f_i(v_n) - \sum_{i \in \mathbb{I}_n} \nabla f_i(\hat{w}_s) \right) + \nabla f(\hat{w}_s) \tag{3.9}$$

and $\hat{w}_s$ is the value of w in every N iteration. The last term in (3.9) computes the full gradient using all data samples at every N iteration which reduces the variance and allows us to preserve the convergence rate of FISTA. The SFISTA algorithm is shown in Algorithm 3.3. An approach similar to [131] can be used to prove the convergence of SFISTA. The main theorem, proven in Appendix A.1, which guarantees the convergence is as follows:

Theorem 1. Consider Algorithm 3.3 with a mini-batch size of $\bar{m}$ and the Lipschitz constant L

where the step size γ is a positive number such that:

$$\gamma^{-1} \geq \max\left(\frac{L}{2} + \sqrt{\frac{1}{4} + \frac{4L^2(m-\bar{m})}{\bar{m}(m-1)}}, L\right) \tag{3.10}$$

and

$$\gamma < \left(1 - \frac{t_{N-1}^2}{t_N^2}\right) \frac{\bar{m}(m-1)}{8L(m-\bar{m})} \tag{3.11}$$

then,

$$\mathbb{E}[F(w_N)] - F(w^*) \leq \frac{\mathbb{E}[F(\hat{w}_s)] - F(w^*)}{t_N^2(1-\eta)^N} + \frac{C^2}{2\gamma t_N^2(1-\eta)^N} \tag{3.12}$$

for some $C \geq 0$ and $0 < \eta < 1 - \frac{t_{N-1}^2}{t_N^2}$. $\mathbb{E}[\cdot]$ denotes the expectation with respect to $\mathbb{I}_n$ and w^* is the optimal solution to problem (3.1). Since $t_N = O(N)$, η can be chosen to be close enough to 0 so that SFISTA converges to the optimal solution with a rate of $O(1/N^2)$:

$$\mathbb{E}[F(w_N)] - F(w^*) \leq \frac{C_1}{N^2}\left(\mathbb{E}[F(\hat{w}_s)] - F(w^*)\right) + \frac{C_2^2}{\gamma N^2} \tag{3.13}$$

for positive constants C_1 and C_2. Also, with additional constraints, such as strong convexity of the objective function, a better rate for stochastic PN methods with variance-reduction is obtainable [124].

Algorithm 3.3 SFISTA

1 **Input:** $\hat{w}_0 = w_{-1} = \mathbf{0} \in \mathbb{R}^d$, $t_0 = 1$, γ
2 **for** $s = 0, \ldots$ **do**
3 $w_0 = \hat{w}_s$
4 **for** $n = 1, \ldots, N$ **do**
5 $t_n = \frac{1+\sqrt{1+t_{n-1}^2}}{2}$
6 $v_n = w_{n-1} + \frac{t_{n-1}-1}{t_n}(w_{n-1} - w_{n-2})$
7 $w_n = \text{Prox}_\gamma(v_n - \gamma\nabla\hat{f}(v_n))$
8 $\hat{w}_{s+1} = w_N$
9 **output** w_N

The total number of floating-point operations for the proposed stochastic formulation of FISTA with a sampling rate of b is reduced by a factor of $1/b$ which reduces the computational complexity without changing the convergence rate.

SFISTA for the l1-regularized least-squares problem. This work focuses on the problem of l1-regularized least-squares where the gradient is computed based on (3.4) and the gradient and the Hessian are related by (3.5). As seen in (3.9), the gradient is estimated with a random sampling of the input data at each iteration. For l1-regularized least-squares, since the non-smooth function is an l1-norm operator, we write the update in line 7 of Algorithm 3.3 as:

$$w_n = \mathbb{S}_{\lambda\gamma}(v_n - \gamma\nabla\hat{f}(v_n)) \tag{3.14}$$

where $\mathbb{S}_\alpha(\beta) = sign(\beta)\max(|\beta| - \alpha, 0)$. We also use the same symbol $\mathbb{I}_n$ to represent the sampling matrix $\mathbb{I}_n = [e_{i_1}, e_{i_2}, ..., e_{i_{\bar{m}}}] \in \mathbb{R}^{m\times\bar{m}}$ where $\{i_h \in [m] | h = 1, ..., \bar{m}\}$ is chosen uniformly at random

and $e_i \in \mathbb{R}^m$ is all zeros except entry i. Finally, to simplify SFISTA, we rewrite the update for v_n as:

$$v_n = w_{n-1} + \mu_n(w_{n-1} - w_{n-2}) \tag{3.15}$$

where $\mu_n = \frac{t_{n-1}-1}{t_n}$. The SFISTA algorithm for l1-regularized least-squares problem is shown in Algorithm 3.4.

Algorithm 3.4 SFISTA for L1-regularized Least-Squares Problem

Input: $\hat{w}_0 = w_{-1} = \mathbf{0} \in \mathbb{R}^d$, $t_0 = 1$, γ
for $s = 0, ...$ **do**
  $w_0 = \hat{w}_s$
  for $n = 1, ..., N$ **do**
    $t_n = \frac{1+\sqrt{1+t_{n-1}^2}}{2}$
    $\mu_n = \frac{t_{n-1}-1}{t_n}$
    $v_n = w_{n-1} + \mu_n(w_{n-1} - w_{n-2})$
    $g_n = \frac{1}{\tilde{m}}\left(X\mathbb{I}_n\mathbb{I}_n^T X^T v_n - X\mathbb{I}_n\mathbb{I}_n^T y\right)$
    $\theta_n = v_n - \gamma g_n$
    $w_n = \mathbb{S}_{\lambda\gamma}(\theta_n)$
  $\hat{w}_{s+1} = w_N$
output w_N

As demonstrated, SFISTA for l1-regularized least-squares converges to the optimal solution with a rate of $O(1/N^2)$. Therefore, it keeps the convergence rate of FISTA and reduces the number of required floating-point operations at each iteration.

3.3.2 Reducing Communication Costs with Overlapping Iterations and Hessian-Reuse

SFISTA solves the l1-regularized least-squares problem by iteratively computing a Hessian matrix and thereafter updating local variables. Since SFISTA communicates data at each iteration, it becomes communication-bound when executed on distributed memory systems for large datasets. We reduce the communication overhead of SFISTA for the l1-regularized least-squares problem by *(1)* proposing an iteration-overlapping technique that reduces latency costs in SFISTA by $O(k)$ without altering bandwidth costs and convergence behavior; *(2)* and introducing a subproblem that reduces the total number of iterations N needed for SFISTA to convergence which results in lower bandwidth and latency costs. The combination of these techniques, which we call the *Reduced-Communication SFISTA (RC-SFISTA)* algorithm, reduces the number of iterations in the algorithm that require data communication over the network.

Latency, arithmetic, and bandwidth costs in SFISTA. We use the model discussed in (3.7) to analyze the performance of SFISTA. Table 3.1 summarizes the latency, bandwidth, and flop cost of SFISTA. As demonstrated, the performance and scalability of SFISTA are limited by latency and bandwidth costs which both increase with the number of iterations N and number of processors P. The RC-SFISTA formulation proposed in this section reduces the latency cost by a factor of $O(k)$ while preserving the bandwidth cost.

Overlapping iterations in SFISTA to reduce latency costs. We propose a novel formulation of SFISTA that allows for iterations of the algorithm to be overlapped to reduce communication

Table 3.1: Latency, flops, and bandwidth costs for N iterations of RC-SFISTA and SFISTA. Parameters d, $\bar{m}$, k, and S represent # columns, # sampled rows, the iteration-overlapping parameter, and the inner loop parameter; f is the matrix non-zero fill-in.

Algorithm	Latency cost (L)	Flops cost (F)	Bandwidth cost (W)
SFISTA	$O\left(Nlog(P)\right)$	$O\left(\frac{Nd^2\bar{m}f}{P}\right)$	$O(Nd^2log(P))$
RC-SFISTA	$O\left(\frac{Nlog(P)}{k}\right)$	$O\left(\frac{Nd^2\bar{m}f}{P} + Sd^2\right)$	$O\left(Nd^2log(P)\right)$

costs. Available works that overlap iterations in some optimization methods, do not support FISTA, and reduce latency costs at the expense of increasing the amount of data moved among processors over a network, i.e., message cost. However, our reformulation of SFISTA does not alter the message costs and provides an $O(k)$ reduction in latency costs of the algorithm with overlapping iterations. By leveraging the fact that multiple instances of the Hessian could be generated at once, we unroll k-consecutive iterations in SFISTA. The recurrence updates in SFISTA should be unrolled for k iterations so that updates to the optimization variable can be postponed for k iterations. Lets define $\Delta w_n = \mathbb{S}_{\lambda\gamma}(\theta_n) - w_{n-1}$. We start by changing the loop index in Algorithm 3.4 from n to $nk + j$ where n is the outer loop index, k is the recurrence unrolling parameter, and j is inner loop index. Assuming we are at iteration $nk + 1$ and v_{nk}, g_{nk}, θ_{nk} and Δw_{nk} have been computed and if $\Delta v_{nk} = v_{nk+1} - v_{nk}$, then the updates for the next iteration are:

$$\begin{aligned}
\Delta w_{nk+1} =& \mathbb{S}_{\lambda\gamma}(\theta_{nk+1}) - w_{nk} \\
g_{nk+1} =& \frac{1}{\bar{m}}\left[X\mathbb{I}_{nk+1}\mathbb{I}_{nk+1}^T X^T\left(v_{nk} + \Delta v_{nk}\right) - X\mathbb{I}_{nk+1}\mathbb{I}_{nk+1}^T y\right] \\
\theta_{nk+1} =& v_{nk} + \Delta v_{nk} - \gamma g_{nk+1}
\end{aligned}$$

thus, for iteration $nk + 2$ the updates are computed as follows:

$$\begin{aligned}
\Delta w_{nk+2} =& \mathbb{S}_{\lambda\gamma}(\theta_{nk+2}) - (w_{nk} + \Delta w_{nk+1}) \\
g_{nk+2} =& \frac{1}{\bar{m}}\left[X\mathbb{I}_{nk+2}\mathbb{I}_{nk+2}^T X^T\left(v_{nk} + \Delta v_{nk} + \Delta v_{nk+1}\right)\right. \\
& \left. - X\mathbb{I}_{nk+2}\mathbb{I}_{nk+2}^T y\right] \\
\theta_{nk+2} =& v_{nk} + \Delta v_{nk} + \Delta v_{nk+1} - \gamma g_{nk+2}
\end{aligned}$$

Therefore, by induction we will have

$$\begin{aligned}
g_{nk+j} =& \frac{1}{\bar{m}}\left[X\mathbb{I}_{nk+j}\mathbb{I}_{nk+j}^T X^T\left(v_{nk} + \sum_{i=0}^{j-1}\Delta v_{nk+i}\right) - X\mathbb{I}_{nk+j}\mathbb{I}_{nk+j}^T y\right] \\
\theta_{nk+j} =& v_{nk} + \sum_{i=0}^{j-1}\Delta v_{nk+i} - \gamma g_{nk+j} \\
\Delta w_{nk+j} =& \mathbb{S}_{\lambda\gamma}(\theta_{nk+j}) - (w_{nk} + \sum_{i=1}^{j-1}\Delta w_{nk+i})
\end{aligned} \tag{3.16}$$

and Δv_{n+k} is obtained via

$$\begin{aligned}\Delta v_{nk+j} =&(1+\mu_{nk+j+1})\left[w_{nk+j}-w_{nk+j-1}\right]\\ &-\mu_{nk+j}\left[w_{nk+j-1}-w_{nk+j-2}\right]\\ =&(1+\mu_{nk+j+1})\Delta w_{nk+j}-\mu_{nk+j}\Delta w_{nk+j-1}\end{aligned} \tag{3.17}$$

With this approach, updates are postponed for k consecutive iterations. Communication in (3.16) is avoided by computing the following matrices for k iterations and storing them on all processors:

$$H_{nk+j} =\frac{1}{\bar{m}}X\mathbb{I}_{nk+j}\mathbb{I}_{nk+j}^T X^T, \quad R_{nk+j} = \frac{1}{\bar{m}}X\mathbb{I}_{nk+j}\mathbb{I}_{nk+j}^T y \tag{3.18}$$

H_n is the approximated Hessian at iteration n for the l1-regularized least-squares problem and the processors over the network communicate only every k iterations. This will reduce the number of messages transferred by a factor of *O(k)*. The resulting method does not change the convergence of SFISTA and keeps the same number of arithmetic operations.

The RC-SFISTA algorithm is shown in Algorithm 3.5. Algorithm parts that relate to implementing iteration-overlapping are highlighted in red. As shown, the number of iterations in line 2 is reduced by a factor of k. The inner loop in line 3 computes local matrices H and R and the inner loop in line 7 updates local variables without communication. Table 3.1 shows the latency, bandwidth, and flop cost of RC-SFISTA. With iteration-overlapping, the latency of SFISTA is reduced by a factor of k while bandwidth costs do not change which can potentially lead to a k-fold speedup.

Algorithm 3.5 RC-SFISTA for the L1-regularized Least-Squares Problem

1 **Input:** $X \in \mathbb{R}^{d\times m}$, $y \in \mathbb{R}^m$, $w_0 = w_{-1} = \mathbf{0} \in \mathbb{R}^d$, $k \in \mathbb{N}$, $b \in (0,1]$, $t_0 = 1$, $\bar{m} = \lfloor bm \rfloor$,$\gamma$
2 **for** $n = 0, ..., \frac{N}{k}$ **do**
3 **for** $j = 1, ..., k$ **do**
4 Generate $I_{nk+j} = [e_{i_1}, e_{i_2}, ..., e_{i_{\bar{m}}}] \in \mathbb{R}^{m\times\bar{m}}$ where $\{i_h \in [m] | h = 1, ..., \bar{m}\}$ is chosen uniformly at random
5 $H_{nk+j} = \frac{1}{\bar{m}}XI_{nk+j}I_{nk+j}^T X^T, R_{nk+j} = \frac{1}{\bar{m}}XI_{nk+j}I_{nk+j}^T y$
6 set $G = [H_{nk+1}|H_{nk+2}|...|H_{(n+1)k}]$ and $R = [R_{nk+1}|R_{nk+2}|...|R_{(n+1)k}]$ and send them to all processors.
7 **for** $j = 1, ..., k$ **do**
8 H_{nk+j} and R_{nk+j} are $d \times d$ and $d \times 1$ blocks of G and R respectively
9 **for** $s = 1, ..., S$ **do**
10 update Δv_{nk+j-1}^s based on (3.20)
11 update g_{nk+j}^s based on (3.21)
12 update θ_{nk+j}^s based on (3.22)
13 update Δw_{nk+j}^s based on (3.23)
14 $w_{nk+j}^s = \Delta w_{nk+j}^s + w_{nk+j}^{s-1}$
15 $w_{nk+j} = w_{nk+j}^S$
16 **output** w_N

The Hessian-reuse method. We propose a novel technique called *Hessian-reuse* to further reduce the number of outer iterations in RC-SFISTA. The objective of Hessian-reuse, which solves a local subproblem, is to find an efficient trade-off between data communication and *local operations*, operations that are computed on the same processor and do not lead to inter-processor data communication. By solving a subproblem local to each processor, shown in lines 9-15 of Algorithm 3.5, we expect the overall problem to converge faster, i.e., the number of iterations corresponding to

the for-loop in line 2 which require inter-processor data communication to reduce. The following discusses the Hessian-reuse method and analyzes why a better convergence is expected with this formulation.

Every update in SFISTA requires matrices H_n and R_n which can be reused repeatedly to update local variables. Reusing the Hessian could contribute more to minimizing the objective function and in particular, it will reduce the total number of iterations N in SFISTA. Since both latency and bandwidth costs increase with the number of iterations, this approach can reduce communication overhead. We first consider the subproblem in PN methods which is provided by:

$$\begin{aligned} z_k =& \operatorname*{argmin}_y \frac{1}{2}(y-w_n)^T \boldsymbol{H_n}(y-w_n) + \nabla f(w_n)^T(y-w_n) + g(y) \\ =& \operatorname*{argmin}_y \Phi(y) + g(y) \end{aligned} \tag{3.19}$$

where $\Phi(y)$ is the smooth segment of the objective function. To solve this minimization problem, RC-SFISTA requires the gradient of the smooth segment of this objective problem given by $\nabla\Phi(y) = H_n y - R_n$ which has the same equation for the gradient in the l1-regularized least-squares problem in (3.5). This means applying SFISTA to solve the subproblem at (3.19) is identical to applying the SFISTA recurrence updates while using the same H_n and R_n to compute the gradient.

While we apply iteration-overlapping, the single update rules for SFISTA (lines 6-10 Algorithm 3.4) are changed into a new inner loop update. In other words, the update rules are changed to:

$$\Delta v^s_{nk+j} = (1+\mu_{s+1})\Delta w^s_{nk+j} - \mu_s \Delta w^s_{nk+j-1} \tag{3.20}$$

$$g^s_{nk+j} = H_{nk+j}\left(v_{nk} + \sum_{i=0}^{j-1} \Delta v^s_{nk+i}\right) - R_{nk+j} \tag{3.21}$$

$$\theta^s_{nk+j} = v_{nk} + \sum_{i=0}^{j-1} \Delta v^s_{nk+i} - \gamma g^s_{nk+j} \tag{3.22}$$

$$\Delta w^s_{nk+j} = \mathbb{S}_{\lambda\gamma}(\theta^s_{nk+j}) - (w_{nk} + \sum_{i=1}^{j-1} \Delta w^s_{nk+i}) \tag{3.23}$$

where $\Delta w^s_n = \mathbb{S}_{\lambda\gamma}(\theta^s_n) - w^s_{n-1}$ and $\Delta w^1_n = \Delta w_n$. Using the same Hessian for multiple iterations could be seen as sampling the same data points from matrices X and y. This is doable as long as inner loop parameter S is small compared to the total iterations N. Otherwise, the subproblem is over-solved and executes redundant flops which could increase the overall runtime of the algorithm. The applied Hessian-reuse method is shown in blue in Algorithm 3.5. The inner loop at Line 9 updates the local variables for S iterations redundantly on all processors. The parameter S is tuned to find an efficient trade-off between computation and communication.

As discussed, with iteration-overlapping, RC-SFISTA reduces the latency costs by a factor of $O(k)$ without increasing bandwidth costs. With the Hessian-reuse method, RC-SFISTA increases the algorithm's computation complexity to further reduce communication rounds. Thus, the parameter S should be tuned to find an efficient trade-off between computation complexity and data communication. The algorithm's cost is shown in Table 3.1.

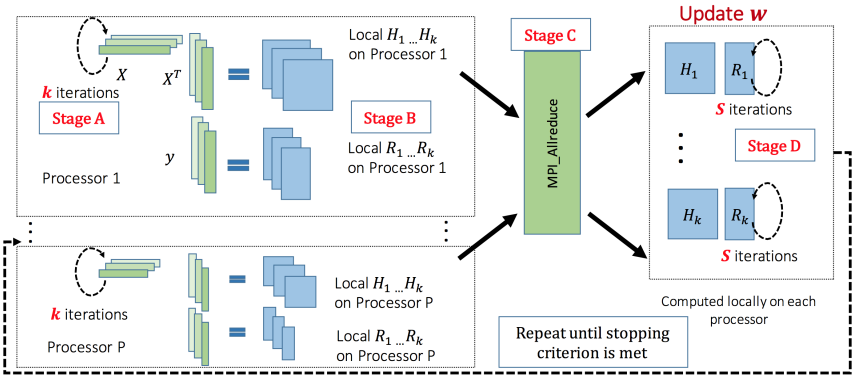


Figure 3.1: A high-level description of RC-SFISTA implemented on a distributed memory system of P processors.

3.3.3 Extension to Proximal Newton Methods

RC-SFISTA could be used both as an inner solver inside PN methods or as an independent solver for the l1-regularized least-squares problem. RC-SFISTA approximates the Hessian with random sampling which follows the same logic in line 3 of Algorithm 3.1. Then both methods minimize a quadratic subproblem that follows (3.19). Therefore, RC-SFISTA could also be used as an inner solver for PN methods. In this case, the iteration-overlapping approach reduces latency costs by a factor of $O(k)$ and if used independently it benefits from both iteration-overlapping and Hessian-reuse.

3.4 Implementation on Distributed Architectures

This section presents the implementation of RC-SFISTA on a distributed architecture. The theoretical upper bounds for parameters related to iteration overlapping and the Hessian-reuse implementations are also provided based on machine specification.

3.4.1 Distributed Implementation

An overview of implementing RC-SFISTA on distributed architectures is shown in Figure 3.1. We assume that the data matrix X is sparse with "fdm" non-zeros that are uniformly distributed, where $0 < f < 1$ represents the percentage of non-zero fill-in in the data matrix X. Also, X and y are partitioned column- and row-wise, respectively, on P processors. In *stage A*, every processor randomly samples columns from X and the corresponding rows from y for k iterations. Using the sampled data, k instances of matrices H and R are computed locally on each processor in *stage B*. Processor contributions are combined using MPI_Allreduce during *stage C*. Since the result from MPI_Allreduce is stored on all processors, RC-SFISTA will compute k iterations of solution updates without any communication between processors and each processor solves k subproblems for S iterations at *stage D*. This process is repeated until a stopping criterion is met.

3.4.2 RC-SFISTA Parameter Bounds

The iteration-overlapping parameter k significantly improves the performance of RC-SFISTA if latency costs dominate the overall runtime of the algorithm. Also, the inner loop parameter S reduces

the total number of iterations in RC-SFISTA at the expense of more flops, often leading to better performance. In this section, the bounds for parameters k and S are derived based on algorithm and machine specifications.

RC-SFISTA runtime. We use the model in (3.7) to analyze the performance of RC-SFISTA. The total number of flops for RC-SFISTA is dominated by the matrix-matrix multiplication in line 5 and the Hessian-reuse in lines 10-14 in Algorithm 3.5. Also, RC-SFISTA only communicates a matrix of size d^2 every k iterations in line 6 requiring $O(d^2k)$ words to be moved between processors with $\frac{N}{k}$ messages. Thus, the total runtime of Algorithm 3.5 is:

$$T = \gamma\left(\frac{Nd^2\bar{m}f}{P} + Sd^2\right) + \alpha\left(\frac{Nlog(P)}{k}\right) + \beta\left(Nd^2log(P)\right) \tag{3.24}$$

The iteration-overlapping parameter *k*. The theoretical upper bound for k depends on machine specifications and dataset dimensions. Since k only appears in latency costs in (3.24), increasing k will always lead to a lower running time. However, the performance obtained from iteration-overlapping is more significant if the latency cost dominates the total runtime. By considering latency and bandwidth, the following upper bound is achieved:

$$k \leq \frac{\alpha}{\beta d^2} \tag{3.25}$$

This upper bound shows that RC-SFISTA leads to better performance on distributed platforms with a higher rate of latency to bandwidth ratio ($\frac{\alpha}{\beta}$). Comparing the first two terms in (3.24), i.e., flops and latency costs, we have:

$$k \leq \frac{\alpha NPlog(P)}{\gamma[Nd^2\bar{m}f + Sd^2P]}. \tag{3.26}$$

Equation 3.26 shows that for matrices with a lower sparsity degree f, k can be larger. In particular, if the dataset is very sparse ($f \sim 0$) we can write:

$$kS \leq \frac{\alpha Nlog(P)}{\gamma d^2}. \tag{3.27}$$

The upper bound in (3.27) provides a trade-off between the computational complexity and communication cost of RC-SFISTA where increasing the value of the iteration-overlapping parameter k results in a tighter bound for S.

The inner loop parameter *S*. Equation (3.27) shows that for higher values of k, a lower value for S is expected. Specifically, if the upper bound (3.25) is used, then:

$$S \leq \frac{\beta Nlog(P)}{\gamma}. \tag{3.28}$$

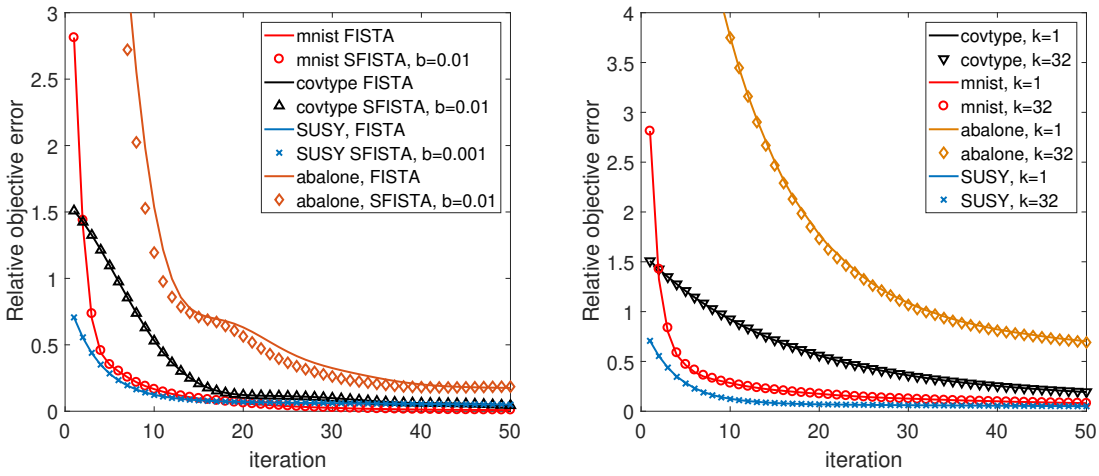
Equation (3.28) states that the upper bound for S depends on machine-specific parameters β and γ. Therefore, RC-SFISTA achieves a better performance on distributed architectures with a higher ratio of $\frac{\beta}{\gamma}$.

3.5 Results

This section presents the experimental setup and performance results. We show that k does not change the convergence behavior of SFISTA. Experimental results are provided that show the inner loop parameter S improves the convergence of RC-SFISTA. Afterward, speedup results of the proposed method as an independent solver and inner solver for PN methods are discussed. Finally, we demonstrate that RC-SFISTA performs better than the state-of-the-art framework ProxCoCoA up to 12×.

Table 3.2: The datasets for experimental study.

Dataset	Row numbers	Column numbers	Percentage of nnz (f)	Size (nnz)
abalone	4177	8	100%	258.7KB
SUSY	5M	18	25.39%	2.47GB
covtype	581,012	54	22.12%	71.2MB
mnist	60,000	780	19.22%	114.8MB
epsilon	400,000	2000	100%	12.16GB



(a) The effect of sampling rate.

(b) The effect of parameter k.

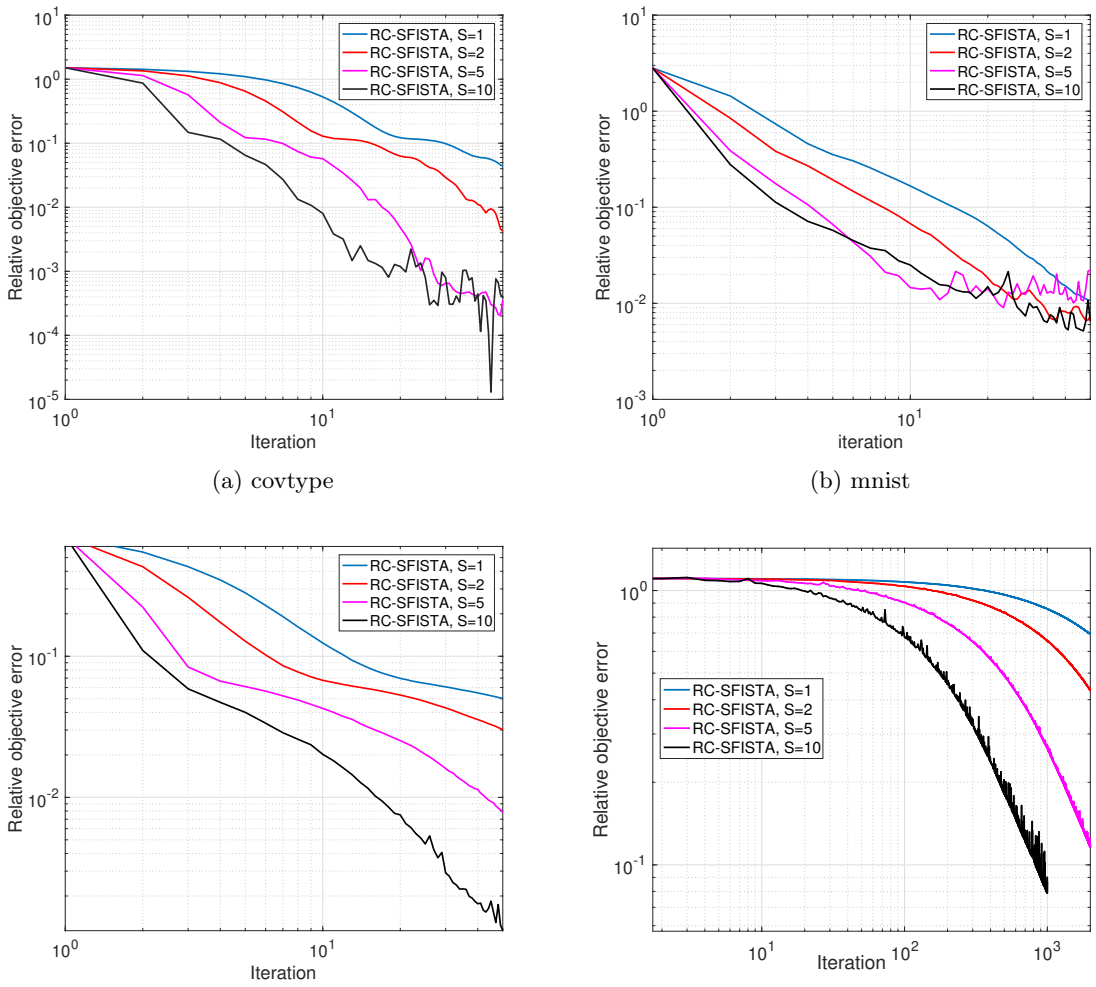
Figure 3.2: Convergence of RC-SFISTA for different b and k.

3.5.1 Experimental Setup

Table 3.2 shows the datasets used for our experiments [132]. The datasets are from dense and sparse machine learning applications and vary in size and sparsity [132]. RC-SFISTA is implemented in C/C++ using Intel MKL 11.1 for (sparse/dense) BLAS routines and MPI 2.1 for parallel processing. We use the compressed sparse row format to store the data of sparse datasets; abalone is stored as dense. Our experiments are conducted on the XSEDE Comet CPU nodes [133]. Because of resource constraints on Comet, for experiments with less than 64 nodes, we use one processor per node, while for larger runs multiple processors per node were used. For example, to execute RC-SFISTA on 256 processors, we use 64 nodes and 4 processors per node.

Regularization parameter λ. The parameter λ should be chosen based on the prediction accuracy of the dataset and can affect convergence rates. We tune λ so that our experiments have reasonable running time. The final tuned value for λ is 0.0001 for *epsilon* and 0.1 for all other benchmarks.

Stopping criteria. The relative objective error e_n is used as the stopping criteria where $e_n = |\frac{F(w_n)-F(w^*)}{F(w^*)}|$. Algorithm 3.5 returns when the relative objective error achieves a value less than a user-specified tolerance *tol* which here is chosen to provide a reasonable execution time. The optimal solution, w^* is computed using *Templates for First-Order Conic Solvers* (TFOCS) which is competitive with state-of-the-art methods [134]. TFOCS uses a first-order method where the tolerance for its stopping criteria is 10^{-8}.



(a) covtype (b) mnist

(c) SUSY (d) epsilon

Figure 3.3: Convergence of RC-SFISTA for different values of inner loop parameter S.

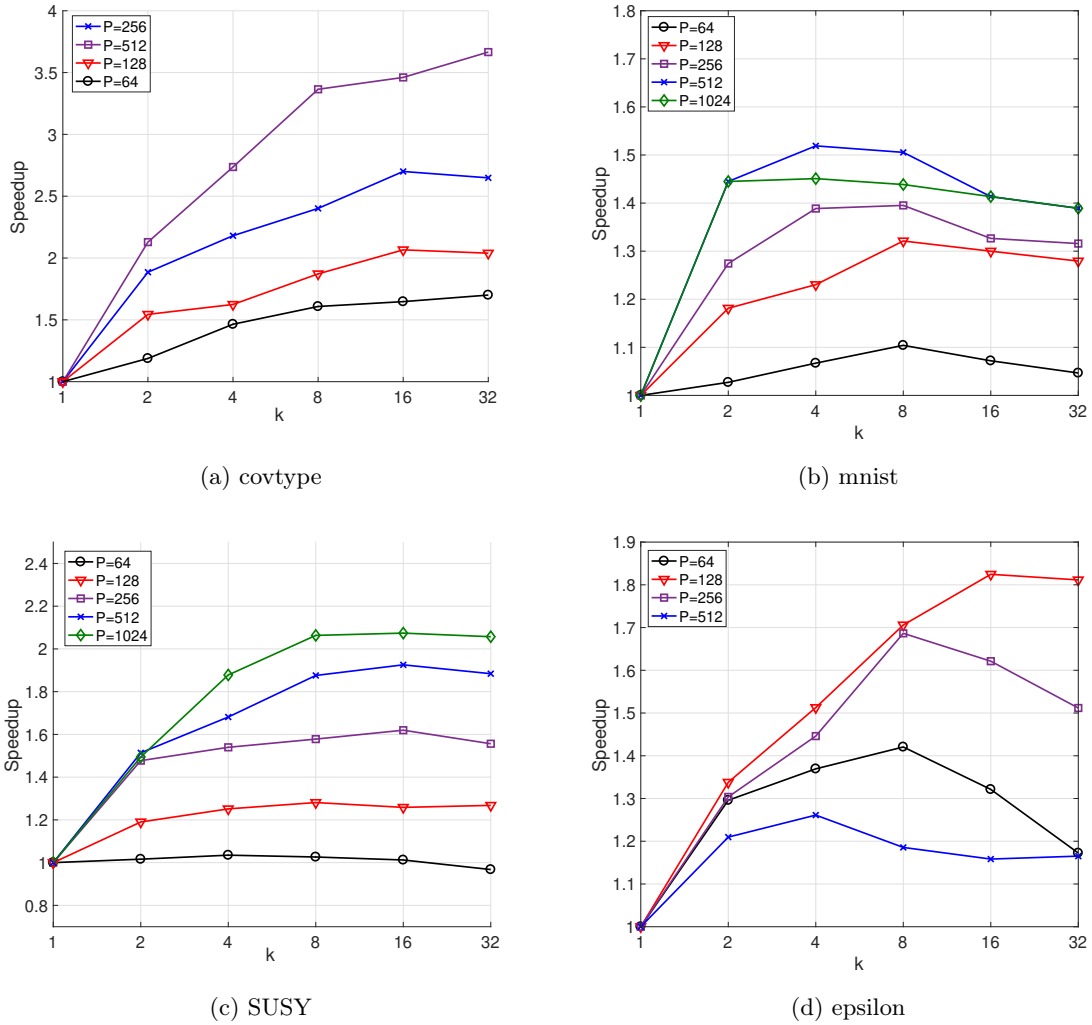
3.5.2 Convergence Results

This section shows the effect of the sampling rate b on the convergence of SFISTA where the algorithm's computation cost is significantly reduced with smaller values of b. We also demonstrate

the effect of parameters k and S on convergence rate and show that RC-SFISTA is numerically stable.

The effect of b on convergence. The relative objective error for RC-SFISTA for different values of sampling rate b is shown in Figure 3.2 (a) while setting k and S to 1. The convergence rates are almost identical compared to FISTA. Smaller values for b result in a smaller mini-batch size $\bar{m}$ and, therefore, a lower computation cost.

The effect of k on convergence. The iteration-overlapping parameter k does not change the convergence of RC-SFISTA since it is the same as SFISTA in exact arithmetic. For a fair comparison, random sampling is fixed by using the same random generator seed, thus, in both scenarios, the same data points are used in the algorithm. The convergence properties of RC-SFISTA for different values of k are shown in Figure 3.2 (b). The experiments demonstrate that changing k does not affect the stability and relative objective error. We tested the convergence rate and stability behavior of the algorithm for up to $k = 128$ and a similar trend was observed.



(a) covtype

(b) mnist

(c) SUSY

(d) epsilon

Figure 3.4: Speedup results for RC-SFISTA compared to SFISTA for different values of the iteration-overlapping parameter k.

The effect of S on convergence. The convergence behavior of RC-SFISTA for different values of S is shown in Figure 3.3. Increasing S reduces the total number of iterations in RC-SFISTA to reach the optimal solution. As seen in the figure, even for small values of S, the improvement in convergence is noticeable. However, according to (3.12), larger values of S degrade the convergence behavior of RC-SFISTA which can also be seen in the figure for S equal to 10 for all benchmarks.

3.5.3 Speedup Comparison

This section shows the speedup for RC-SFISTA compared to SFISTA for different values of k and S. The tolerance parameter *tol* is set to 0.01 in all the experiments. We show that increasing k reduces latency costs by a factor of k and improves the performance of RC-SFISTA on a distributed architecture. We also provide speedup results for the inner loop parameter S and show the trade-off between the computation cost of RC-SFISTA and data communication.

The effect of parameter k on speedup. The speedup of RC-SFISTA compared to SFISTA for a different number of processors (P) and different values of k are shown in Figure 3.4. Parameter S is set to one to only analyze the effect of k on the total runtime of the algorithm. As shown in the figure, increasing k results in up to 4× speedup for all datasets by reducing latency costs by a factor of k. However, for larger values of k, the performance of RC-SFISTA degrades for the dataset *epsilon* since the computation cost dominates the overall running time of the algorithm.

The value of k depends on machine specifications and the dataset size. For example, the machine parameters α and β for the XSEDE Comet nodes used in the experiments are 10^{-6} and 1.42×10^{-10}, thus, the theoretical upper bound (3.25) for the *covtype* dataset is 2. However, according to (3.24) all values of k reduce the total runtime of the algorithm, thus, RC-SFISTA continues to scale even for larger values of k.

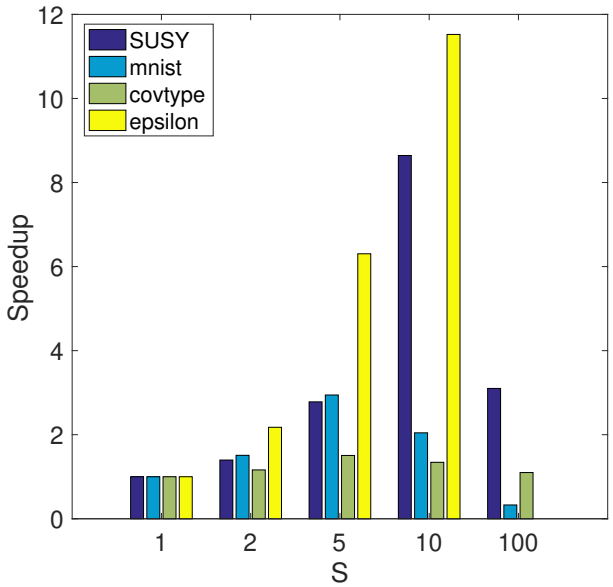
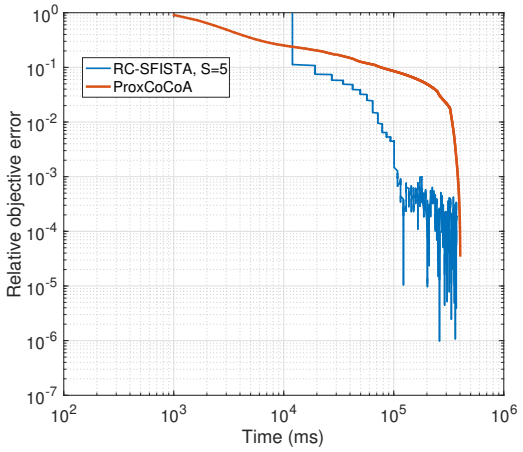


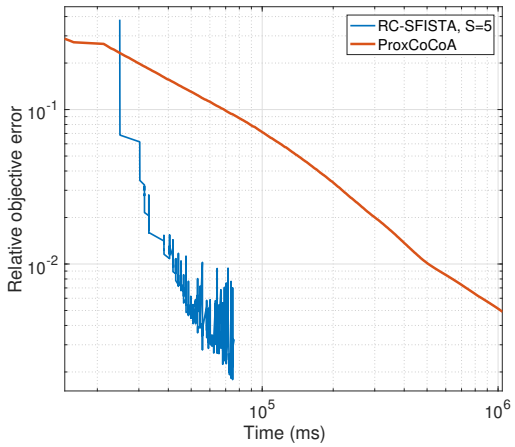
Figure 3.5: Speedup of RC-SFISTA vs. SFISTA for different S.

The effect of S on speedup. The speedup results of RC-SFISTA compared to SFISTA on 256 processors for different values of S are shown in Figure 3.5. The value of parameter k is tuned

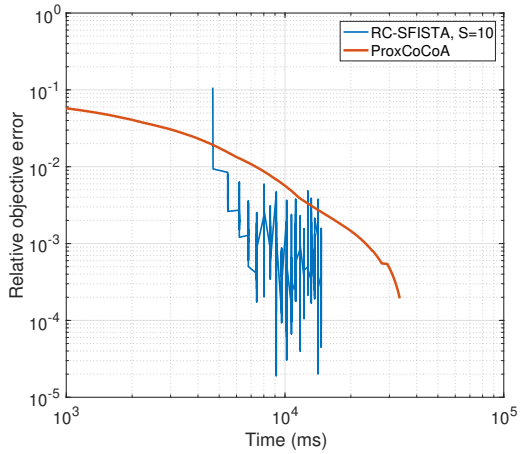
for all benchmarks. Increasing S will result in better convergence properties by reducing the total number of iterations. S is increased until an efficient trade-off between computation cost and data communication is reached. As shown in Figure 3.5, for larger values of S, the cost of redundant computation overwhelms the total cost of the algorithm, and the speedup decreases. For example, RC-SFISTA shows a speedup of 3× compared to SFISTA for the *mnist* dataset when $S = 5$, while it achieves a speedup of 2× when the inner loop parameter increases to 10. The upper bound for parameter S depends on both the architecture and the algorithm iterations N. Since on XSEDE Comet nodes the value of γ is 4×10^{-10}, based on the upper bound in (3.27) with values $k = 1$, $P = 256$, and $N = 200$ for the *mnist* dataset we have $S < 7$. As shown in Figure 3.5, $S = 5$ gives the best speedup for *mnist* on 256 processors.



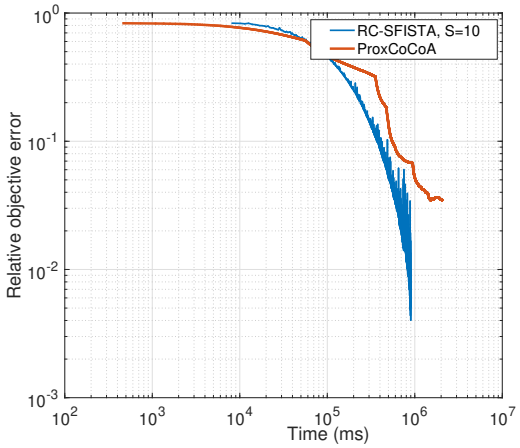
(a) covtype



(b) mnist



(c) SUSY



(d) epsilon

Figure 3.6: Relative objective error of RC-SFISTA compared to ProxCoCoA on 256 processors.

3.5.4 Comparison to ProxCoCoA

In this section, we compare RC-SFISTA with ProxCoCoA, a state-of-the-art framework for solving large-scale l1-regularized least-squares problems. Since ProxCoCoA is optimized and implemented in Apache Spark's MLlib we also implemented RC-SFISTA using Apache Spark MLlib [135]. When analyzing the performance of the algorithms, we measure the relative objective error in terms of wall-clock time. For all the experiments, the value of S is tuned for the best performance. The results with 256 workers on 256 processors are shown in Figure 3.6. ProxCoCoA has a slow convergence for all datasets, however, RC-SFISTA converges faster and reaches a lower relative objective error compared to ProxCoCoA. Table 3.3 summarizes the speedup of RC-SFISTA compared to ProxCoCoA on 256 workers. The parameter *tol* is set to 0.01 for all benchmarks and the sampling rate b is set to 1% for RC-SFISTA.

Dataset	SUSY	covtype	mnist	epsilon
Speedup	1.57×	4.74×	12.15×	3.53×

Table 3.3: Speedup of RC-SFISTA compared to ProxCoCoA.

3.5.5 Speedup Results for PN Methods

This section shows the results for RC-SFISTA used as an inner solver in PN methods (Algorithm 3.1). The speedups are normalized over the PN method with FISTA as an inner solver. The Hessian approximation for both algorithms is obtained using uniform sampling by initializing all processors with the same seed for the random number generator. The parameter S for RC-SFISTA and the number of inner solver iterations for PN methods are tuned for best performance. The speedup results on 512 processors are shown in Figure 3.7. As demonstrated, as long as the latency cost dominates the communication cost, increasing k results in a better speedup.

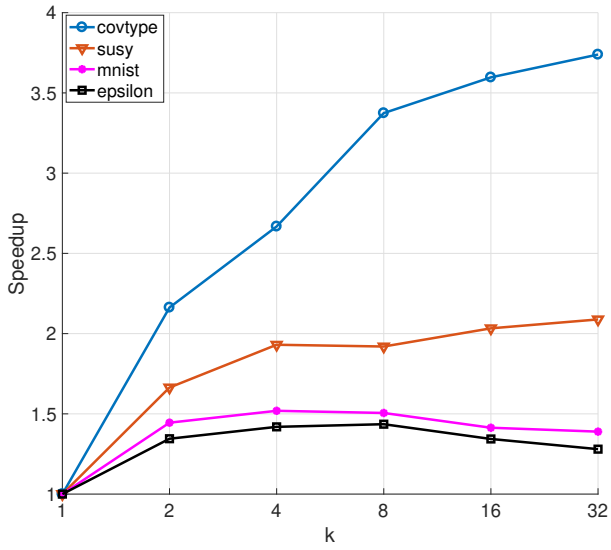


Figure 3.7: Speedup results for PN method with RC-SFISTA as inner solver compared to FISTA used as inner solver.

3.6 Conclusion

The performance of proximal Newton methods used for solving l1-regularized least-squares problems is limited by the performance of the inner solver used in these algorithms. PN methods do not scale well on distributed platforms when operating on large datasets. We propose a novel inner solver, RC-SFISTA, that leverages randomized sampling to overlap iterations and reduce latency costs by a factor of k. The RC-SFSITA algorithm keeps the convergence behavior and preserves the overall bandwidth cost. The performance of the inner solver is further improved by solving local subproblems on each processor at cost of more floating-point operations. Our experiments show that RC-SFISTA provides up to 12× speedup compared to SFISTA and the state-of-the-art method ProxCoCoA for the tested datasets on distributed platforms.

Limitations. In RC-SFISTA, we assume that all processors start and finish their tasks concurrently, which does not hold for systems with processors that have uncertain response times. Hence, the provided upper bounds can be improved by considering this uncertainty. Moreover, for datasets with many features, the upper bound for overlapping parameter k becomes smaller as the dimension increases. However, the loop parameter S is still tunable and can result in an improved performance.

Chapter 4

Asynchronous Quasi-Newton Method on Distributed Memory Systems

The distributed asynchronous quasi-Newton method presented in this chapter was developed by co-authors Konstantin Mischenko, Aryan Mokhtari, Maryam Mehri Dehnavi, and Mert Gurbuzbalaban. This work was published under the title "DAve-QN: A Distributed Averaged Quasi-Newton Method with Local Superlinear Convergence Rate" at the AISTATS 2020 conference [136].

In this chapter, we develop a quasi-Newton algorithm for solving the empirical risk minimization problem, which reduces the communication cost on distributed memory systems by using asynchronous communications. Unlike Chapter 3, in which we reduced the communication cost of quasi-Newton methods using infrequent synchronization, the proposed method relaxes the communication constraints and allows the processors for delayed updates. While the communication delay can potentially degrade the convergence rate, we show that as long as the delay is bounded, our method enjoys the local superlinear convergence rate. The proposed method is based on an asynchronous averaging scheme of decision vectors and gradients in a way to effectively capture the local Hessian information of the objective function. Our proposed algorithm, Distributed Averaged Quasi-Newton (DAve-QN), can be implemented in asynchronous master/worker distributed settings; allowing better scalability properties with parallelization while being robust to delays of the workers. The contributions of this chapter are:

- The first communication-efficient asynchronous optimization algorithm that can achieve superlinear convergence for solving the empirical risk minimization problem under the master/worker communication model which only requires sharing vectors of size $O(p)$ where p is the size of the optimization variable.

- Our theory supports asynchronous computations subject to both bounded delays and unbounded delays with a bounded time-average. We provide numerical experiments that show significant improvement compared to state-of-the-art distributed algorithms.

4.1 Introduction

In this work, we focus on distributed algorithms for empirical risk minimization problems. The setting is as follows: Given n machines, each machine has access to m_i samples $\{\xi_{i,j}\}_{j=1}^{m_i}$ for $i = 1, 2, \ldots, n$. The samples $\xi_{i,j}$ are random variables supported on a set $\mathcal{P} \subset \mathbb{R}^d$. Each machine has a loss function that is averaged over the local dataset:

$$f_i(\mathbf{x}) = \frac{1}{m_i} \sum_{j=1}^{m_i} \phi(\mathbf{x}, \xi_{i,j}) + \frac{\lambda}{2}\|\mathbf{x}\|_2^2$$

where the function $\phi : \mathbb{R}^p \times \mathbb{R}^d \to \mathbb{R}$ is convex in $\mathbf{x}$ for each $\xi \in \mathbb{R}^d$ fixed and $\lambda \geq 0$ is a regularization parameter. The goal is to develop communication-efficient distributed algorithms to minimize the overall empirical loss defined by

$$\mathbf{x}^* \coloneqq \operatorname*{argmin}_{\mathbf{x}\in\mathbb{R}^p} f(\mathbf{x}) \coloneqq \operatorname*{argmin}_{\mathbf{x}\in\mathbb{R}^p} \frac{1}{n} \sum_{i=1}^{n} f_i(\mathbf{x}). \tag{4.1}$$

The communication model we consider is the *centralized communication* model, also known as the *master/worker* model ([137]). In this model, the master machine possesses a copy of the global decision variable $\mathbf{x}$ which is shared with the worker machines. Each worker performs local computations on its local data which is then communicated to the master node to update the decision variable. Communications can be synchronous or asynchronous, resulting in different types of optimization algorithms and convergence guarantees. The merit of synchronization is that it prevents workers from using obsolete information and, thereby, from submitting a low-quality update of parameters to the master. However, all the nodes have to wait for the slowest worker, which leads to unnecessary overheads. Asynchronous algorithms do not suffer from this issue, maximizing the efficiency of the workers while minimizing the system overheads. Asynchronous algorithms are particularly preferable over networks with heterogeneous machines with different memory capacities, work overloads, and processing capabilities.

There has been a number of distributed algorithms suggested in the literature to solve the empirical risk minimization problem (4.1) based on primal first-order methods ([138–140]), their accelerated or variance-reduced versions ([141–145]), lock-free parallel methods ([62,146]), coordinate descent-based approaches ([137,147–149]), dual methods ([148,150]), primal-dual methods ([137,149,151–153]), distributed ADMM-like methods ([154]) as well as quasi-Newton approaches ([155,156]), inexact second-order methods ([9,157–161]) and general-purpose frameworks for distributed computing environments ([151,152]) both in the asynchronous and synchronous setting. The efficiency of these algorithms is typically measured by the *communication complexity* which is defined as the equivalent number of vectors in $\mathbb{R}^p$ sent or received across all the machines until the optimization algorithm converges to an ε-neighborhood of the optimum value. Lower bounds on the communication complexity have been derived by [162] as well as some linearly convergent algorithms achieving these lower bounds ([141,159]). However, in an analogy to the lower bounds obtained by [163] for first-order centralized algorithms, the lower bounds for the communication complexity are only effective if the dimension p of the problem is allowed to be larger than the number of iterations. This assumption is perhaps reasonable for very large-scale problems where p can be billions, however, it is conservative for moderate to large-scale problems where p is not as large.

Most existing state-of-the-art communication-efficient algorithms for strongly convex problems share vectors of size $O(p)$ at every iteration while having linear convergence guarantees. In this work, we propose the first communication-efficient asynchronous optimization algorithm that can achieve local superlinear convergence for solving the empirical risk minimization problem under the master/worker communication model. Our algorithm is communication-efficient in the sense that it also shares vectors of size $O(p)$. Our theory supports asynchronous computations subject to both bounded delays and unbounded delays with a bounded time-average. We provide numerical experiments that illustrate our theory and practical performance. Our proposed algorithm, Distributed Averaged Quasi-Newton (DAve-QN) is inspired by the Incremental Quasi-Newton (IQN) method proposed by [164] which is a deterministic incremental algorithm based on the BFGS method. In contrast to the IQN method, which is designed for centralized computation, our proposed scheme can be implemented in asynchronous master/worker distributed settings; allowing better scalability properties with parallelization, while being robust to delays of the workers.

Related work. Although the setup that we consider here is an asynchronous master/worker distributed setting, it also relates to incremental aggregated algorithms ([139, 165–170]), as at each iteration the information corresponding to one of the machines, i.e., functions, is evaluated while the variable is updated by aggregating the most recent information of all the machines. In fact, our method is inspired by an incremental quasi-Newton method proposed by [164] and a delay-tolerant method by [58]. However, in the IQN method, the update at iteration t is a function of the last n iterates $\{x^{t-1}, \ldots, x^{t-n}\}$, while in our asynchronous distributed scheme the updates are performed on delayed iterates $\{x^{t-d_1^t-1}, \ldots, x^{t-d_n^t-n}\}$. This major difference between the updates of these two algorithms requires a challenging different analysis. Further, our algorithm can be considered as an asynchronous distributed variant of traditional quasi-Newton methods that have been heavily studied in the numerical optimization community ([171–174]). Also, there have been some works on decentralized variants of quasi-Newton methods for consensus optimization where communications are performed over a fixed arbitrary graph where a master node is impractical or does not exist, this setup is also known as the *multi-agent setting* ([56, 67]). The work proposed by [155] introduces a linearly convergent decentralized quasi-Newton method for decentralized settings. [56] propose an asynchronous Newton-based approach that solves a penalized version of the problem whose solution lies in a $O(\alpha)$ neighborhood of the optimal solution where α is a penalty parameter. Their algorithm enjoys local superlinear convergence guarantees in this neighborhood. We emphasize that our setup is different in the sense that we have a star network topology obeying the master/slave hierarchy. Furthermore, our asymptotic convergence results are stronger than those available in the multi-agent setting as we establish a superlinear convergence rate for the proposed method to the global minimum x^* of the problem (4.1). There has also been recent progress in solving distributed non-convex problems with second-order methods. Among these, the most relevant to our work are [61] which proposes an asynchronous-parallel stochastic L-BFGS method, and the DINGO method ([175]). DINGO is a Newton-type method that optimizes the gradient's norm as a surrogate function with linear convergence guarantees to a local minimum for non-convex objectives that satisfy an invexity property.

Outline. In Section 4.2.1, we review the update of the BFGS algorithm that we build on our distributed quasi-Newton algorithm. We formally present our proposed DAve-QN algorithm in Section 4.2.2. We then provide our theoretical convergence results for the proposed DAve-QN method

in Section 4.3. Numerical results are presented in Section 4.4. Finally, we give a summary of our results and discuss future work in Section 5.

4.2 Algorithm

4.2.1 Preliminaries: The BFGS Algorithm

The update of the BFGS algorithm for minimizing a convex smooth function $f : \mathbb{R}^p \to \mathbb{R}$ is given by

$$\mathbf{x}^{t+1} = \mathbf{x}^t - \eta^t (\mathbf{B}^{t+1})^{-1} \nabla f(\mathbf{x}^t), \tag{4.2}$$

where $\mathbf{B}^{t+1}$ is an estimate of the Hessian $\nabla^2 f(\mathbf{x}^t)$ at time t and η^t is the step size (see e.g., [176]). The idea behind the BFGS (and, more generally, behind quasi-Newton) methods is to compute the Hessian approximation $\mathbf{B}^{t+1}$ using only first-order information. Like Newton's methods, BFGS methods work with step size $\eta_t = 1$ when the iterates are close to the optimum. However, at the initial stages of the algorithm, the step size is typically determined by a line search for avoiding the method to diverge.

A common rule for the Hessian approximation is to choose it to satisfy the secant condition $\mathbf{B}^{t+1}\mathbf{s}^{t+1} = \mathbf{y}^{t+1}$,where $\mathbf{s}^{t+1} = \mathbf{x}^t - \mathbf{x}^{t-1}$, and $\mathbf{y}^{t+1} = \nabla f(\mathbf{x}^t) - \nabla f(\mathbf{x}^{t-1})$ are called the *variable variation* and *gradient variation* vectors, respectively. The Hessian approximation update of BFGS which satisfies the secant condition can be written as a rank-two update

$$\mathbf{B}^{t+1} = \mathbf{B}^t + \mathbf{U}^{t+1} + \mathbf{V}^{t+1}, \quad \mathbf{U}^{t+1} = \frac{\mathbf{y}^{t+1}(\mathbf{y}^{t+1})^T}{(\mathbf{y}^{t+1})^T \mathbf{s}^{t+1}}, \quad \mathbf{V}^{t+1} = -\frac{\mathbf{B}^t \mathbf{s}^{t+1} (\mathbf{s}^{t+1})^T \mathbf{B}^t}{(\mathbf{s}^{t+1})^T \mathbf{B}^t \mathbf{s}^{t+1}}. \tag{4.3}$$

Note that both matrices $\mathbf{U}^t$ and $\mathbf{V}^t$ are rank-one. Therefore, the update (4.3) is rank two. Owing to this property, the inverse of the Hessian approximation $\mathbf{B}^{t+1}$ can be computed at a low cost of $\mathcal{O}(p^2)$ arithmetic iterations based on the Sherman-Morrison-Woodbury formula, instead of computing the inverse matrix directly with a complexity of $\mathcal{O}(p^3)$. For a strongly convex function f with the global minimum $\mathbf{x}^*$, a classical convergence result for the BFGS method shows that the iterates generated by BFGS are *superlinearly* convergent [177], i.e., $\lim_{t\to\infty} \frac{\|\mathbf{x}^{t+1} - \mathbf{x}^*\|}{\|\mathbf{x}^t - \mathbf{x}^*\|} = 0$. There are also limited-memory BFGS (L-BFGS) methods that require less memory ($\mathcal{O}(p)$) at the expense of having a linear (but not superlinear) convergence [176]. Our main goal here is to design a BFGS-type method that can solve problem (4.1) efficiently with superlinear convergence in an asynchronous setting under the master/slave communication model. We introduce our proposed algorithm in the following section.

4.2.2 A Distributed Averaged Quasi-Newton Method (DAve-QN)

In this section, we introduce a BFGS-type method that can be implemented in a distributed setting (master/slave) without any central coordination between the nodes, i.e., asynchronously. To do so, we consider a setting where n worker nodes (machines) are connected to a master node. Each worker node i has access to a component of the global objective function, i.e., node i has access only to the function f_i. The decision variable stored at the master node is denoted by x^t at time t. At each moment t, d_i^t denotes the delay in communication with the i-th worker, i.e., the last exchange with this worker was at time $t - d_i^t$. For convenience, if the last communication was performed exactly at

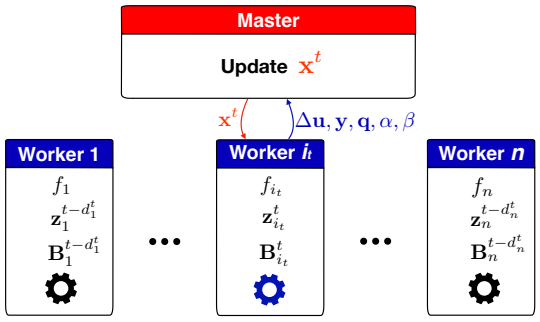


Figure 4.1: Asynchronous communication scheme used by the proposed algorithm.

moment t, then we set $d_i^t = 0$. In addition, D_i^t denotes the double delay in communication, which relates to the penultimate communication and can be expressed as follows: $D_i^t = d_i^t + d_i^{t-d_i^t-1} + 1$. Note that the time index t increases if one of the workers performs an update.

Every worker node i has two copies of the decision variable corresponding to the last two communications with the master, i.e., node i possesses $\mathbf{x}^{t-d_i^t}$ and $\mathbf{z}_i^t := \mathbf{x}^{t-D_i^t}$. Since there has been no communication after $t - d_i^t$, we will clearly have

$$\mathbf{z}_i^{t-d_i^t} = \mathbf{z}_i^t = \mathbf{x}^{t-D_i^t}. \tag{4.4}$$

We are interested in designing a distributed version of the BFGS method described in Section 4.2.1, where each node at time t has an approximation $\mathbf{B}_i^t$ to the local Hessian (Hessian matrix of f_i) where $\mathbf{B}_i^t$ is constructed based on the local delayed decision variables $\mathbf{x}^{t-d_i^t}$ and $\mathbf{z}_i^t$, and therefore the local Hessian approximation will also be outdated satisfying

$$\mathbf{B}_i^t = \mathbf{B}_i^{t-d_i^t}. \tag{4.5}$$

An instance of the setting that we consider in this work is illustrated in Figure 4.1. At time t, one of the workers, say i_t, finishes its task and sends a group of vectors and scalars (that we will precise later) to the master node, avoiding communication of any $p \times p$ matrices as it is assumed that this would be prohibitively expensive communication-wise. Then, the master node uses this information to update the decision variable $\mathbf{x}^t$ using the new information of node i_t and the old information of the remaining workers. After this process, the master sends the updated information to node i_t.

We define the *aggregate Hessian approximation* as

$$\mathbf{B}^t := \sum_{i=1}^n \mathbf{B}_i^t = \sum_{i=1}^n \mathbf{B}_i^{t-d_i^t} \tag{4.6}$$

where we used (4.5). In addition, we introduce

$$\mathbf{u}^t := \sum_{i=1}^n \mathbf{B}_i^t \mathbf{z}_i^t = \sum_{i=1}^n \mathbf{B}_i^{t-d_i^t} \mathbf{z}_i^{t-d_i^t}, \quad \mathbf{g}^t := \sum_{i=1}^n \nabla f_i(\mathbf{z}_i^t) = \sum_{i=1}^n \nabla f_i(\mathbf{z}_i^{t-d_i^t}) \tag{4.7}$$

as the *aggregate Hessian-variable product* and *aggregate gradient* respectively where we made use of

the identities (4.4)–(4.5). All these vectors and matrices are only available at the master node since it requires access to the information of all the workers.

Given that at step $t+1$ only a single index i_t is updated, using the identities (4.4)–(4.7), it follows that the master has the update rules

$$\mathbf{B}^{t+1} = \mathbf{B}^t + \left(\mathbf{B}_{i_t}^{t+1} - \mathbf{B}_{i_t}^t\right) = \mathbf{B}^t + \left(\mathbf{B}_{i_t}^t - \mathbf{B}_{i_t}^{t-d_i^t}\right), \tag{4.8}$$

$$\mathbf{u}^{t+1} = \mathbf{u}^t + \left(\mathbf{B}_{i_t}^{t+1}\mathbf{z}_{i_t}^{t+1} - \mathbf{B}_{i_t}^t\mathbf{z}_{i_t}^t\right) = \mathbf{u}^t + \left(\mathbf{B}_{i_t}^{t+1}\mathbf{x}^{t-d_{i_t}^t} - \mathbf{B}_{i_t}^{t-d_{i_t}^t}\mathbf{x}^{t-D_{i_t}^t}\right), \tag{4.9}$$

$$\mathbf{g}^{t+1} = \mathbf{g}^t + \left(\nabla f_{i_t}(\mathbf{z}_{i_t}^{t+1}) - \nabla f_{i_t}(\mathbf{z}_{i_t}^t)\right) = \mathbf{g}^t + \left(\nabla f_{i_t}(\mathbf{x}^{t-d_{i_t}^t}) - \nabla f_{i_t}(\mathbf{x}^{t-D_{i_t}^t})\right). \tag{4.10}$$

We observe that, only $\mathbf{B}_{i_t}^{t+1}$ and $\nabla f_{i_t}(\mathbf{z}_{i_t}^{t+1}) = \nabla f_{i_t}(\mathbf{x}^{t-d_{i_t}^t})$ are required to be computed at step $t+1$. The former is obtained by the standard BFGS rule applied to f_i carried out by the worker i_t:

$$\mathbf{B}_{i_t}^{t+1} = \mathbf{B}_{i_t}^t + \frac{\mathbf{y}_{i_t}^{t+1}(\mathbf{y}_{i_t}^{t+1})^\top}{\alpha^{t+1}} - \frac{\mathbf{q}_{i_t}^{t+1}(\mathbf{q}_{i_t}^{t+1})^\top}{\beta^{t+1}} \tag{4.11}$$

with

$$\alpha^{t+1} \coloneqq (\mathbf{y}_{i_t}^{t+1})^\top \mathbf{s}_{i_t}^{t+1} \quad \mathbf{y}_{i_t}^{t+1} \coloneqq \mathbf{z}_{i_t}^{t+1} - \mathbf{z}_{i_t}^t = \mathbf{x}^{t-d_{i_t}^t} - \mathbf{x}^{t-D_{i_t}^t}, \tag{4.12}$$

$$\mathbf{q}_{i_t}^{t+1} \coloneqq \mathbf{B}_{i_t}^t \mathbf{s}_{i_t}^{t+1}, \quad \beta^{t+1} \coloneqq (\mathbf{s}_{i_t}^{t+1})^\top \mathbf{B}_{i_t}^t \mathbf{s}_{i_t}^{t+1} = (\mathbf{s}_{i_t}^{t+1})^\top \mathbf{q}_{i_t}^{t+1}. \tag{4.13}$$

Then, the master computes the new iterate as $\mathbf{x}^{t+1} = (\mathbf{B}^{t+1})^{-1}\left(\mathbf{u}^{t+1} - \mathbf{g}^{t+1}\right)$ and sends it to the worker i_t. For the rest of the workers, we update the time counter without changing the variables, so $\mathbf{z}_i^{t+1} = \mathbf{z}_i^t$ and $\mathbf{B}_i^{t+1} = \mathbf{B}_i^t$ for $i \neq i_t$. Although, updating the inverse $(\mathbf{B}^{t+1})^{-1}$ may seem costly at first glance, in fact it can be computed efficiently in $\mathcal{O}(p^2)$ iterations, similar to standard implementations of the BFGS methods. More specifically, if we introduce a new matrix

$$\mathbf{U}^{t+1} \coloneqq (\mathbf{B}^t)^{-1} - \frac{(\mathbf{B}^t)^{-1}\mathbf{y}_{i_t}^{t+1}(\mathbf{y}_{i_t}^{t+1})^\top(\mathbf{B}^t)^{-1}}{(\mathbf{y}_{i_t}^{t+1})^\top \mathbf{s}_{i_t}^{t+1} + (\mathbf{y}_{i_t}^t)^\top(\mathbf{B}^t)^{-1}\mathbf{y}_{i_t}^{t+1}}, \tag{4.14}$$

then, by the Sherman-Morrison-Woodbury formula, we have the identity

$$(\mathbf{B}^{t+1})^{-1} = \mathbf{U}^{t+1} + \frac{\mathbf{U}^{t+1}(\mathbf{B}_{i_t}^{t-d_{i_t}^t}\mathbf{s}_{i_t}^{t+1})(\mathbf{B}_{i_t}^{t-d_{i_t}^t}\mathbf{s}_{i_t}^{t+1})^T\mathbf{U}^{t+1}}{(\mathbf{s}_{i_t}^{t+1})^T\mathbf{B}_{i_t}^{t-d_{i_t}^t}\mathbf{s}_{i_t}^{t+1} - (\mathbf{B}_{i_t}^{t-d_i^t}\mathbf{s}_{i_t}^{t+1})^T\mathbf{U}^{t+1}(\mathbf{B}_{i_t}^{t-d_{i_t}^t}\mathbf{s}_{i_t}^{t+1})}, \tag{4.15}$$

Therefore, if we already have $(\mathbf{B}^t)^{-1}$, it suffices to have only matrix vector products. If we denote $\mathbf{v}^{t+1} = (\mathbf{B}^t)^{-1}\mathbf{y}_{i_t}^{t+1}$ and $\mathbf{w}^{t+1} \coloneqq \mathbf{U}^{t+1}\mathbf{q}_{i_t}^{t+1}$, then these equations can be simplified as

$$\mathbf{U}^{t+1} = (\mathbf{B}^t)^{-1} - \frac{\mathbf{v}^{t+1}(\mathbf{v}^{t+1})^\top}{\alpha^{t+1} + (\mathbf{v}^{t+1})^\top \mathbf{y}_{i_t}^{t+1}}, \quad \mathbf{v}^{t+1} = (\mathbf{B}^t)^{-1}\mathbf{y}_{i_t}^{t+1}, \tag{4.16}$$

$$(\mathbf{B}^{t+1})^{-1} = \mathbf{U}^{t+1} + \frac{\mathbf{w}^{t+1}(\mathbf{w}^{t+1})^\top}{\beta^{t+1} - (\mathbf{q}^{t+1})^\top \mathbf{w}^{t+1}}, \quad \mathbf{w}^{t+1} \coloneqq \mathbf{U}^{t+1}\mathbf{q}_{i_t}^{t+1}, \tag{4.17}$$

where $\alpha^{t+1}, \beta^{t+1}, \mathbf{q}_{i_t}^{t+1}$ and $\mathbf{y}_{i_t}^{t+1}$ are defined by (4.12)–(4.13).

The steps of the DAve-QN at the master node and the workers are summarized in Algorithm 4.1.

Algorithm 4.1 DAve-QN (implementation)

Master:

Initialize $\mathbf{x}$, $\mathbf{B}_i$, $\mathbf{g} = \sum_{i=1}^n \nabla f_i(\mathbf{x})$, $\mathbf{B}^{-1} = (\sum_{i=1}^n \mathbf{B}_i)^{-1}$, $\mathbf{u} = \sum_{i=1}^n \mathbf{B}_i\mathbf{x}$,
for $t = 1$ **to** $T-1$ **do**
  If a worker sends an update:
  Receive $\Delta\mathbf{u}$, $\mathbf{y}$, $\mathbf{q}$, α, β from it
  $\mathbf{u} = \mathbf{u} + \Delta\mathbf{u}$, $\mathbf{g} = \mathbf{g} + \mathbf{y}$, $\mathbf{v} = (\mathbf{B})^{-1}\mathbf{y}$
  $\mathbf{U} = (\mathbf{B})^{-1} - \frac{\mathbf{v}\mathbf{v}^\top}{\alpha + \mathbf{v}^\top\mathbf{y}}$
  $\mathbf{w} = \mathbf{U}\mathbf{q}$, $(\mathbf{B})^{-1} = \mathbf{U} + \frac{\mathbf{w}\mathbf{w}^\top}{\beta - \mathbf{q}^\top\mathbf{w}}$
  $\mathbf{x} = (\mathbf{B})^{-1}(\mathbf{u} - \mathbf{g})$
  Send $\mathbf{x}$ to the worker in return
Interrupt all workers
Output $\mathbf{x}^T$

Worker i**:**

Initialize $\mathbf{x}_i = \mathbf{x}$, $\mathbf{B}_i$
while *not interrupted by master* **do**
  Receive $\mathbf{x}$
  $\mathbf{s}_i = \mathbf{x} - \mathbf{z}_i$
  $\mathbf{y}_i = \nabla f_i(\mathbf{x}) - \nabla f_i(\mathbf{z}_i)$
  $\mathbf{q}_i = \mathbf{B}_i\mathbf{s}_i$
  $\alpha = \mathbf{y}_i^\top\mathbf{s}_i$
  $\beta = \mathbf{s}_i^\top\mathbf{B}_i^t\mathbf{s}_i$
  $\mathbf{u} = \mathbf{B}_i\mathbf{z}_i$
  $\mathbf{B}_i = \mathbf{B}_i + \frac{\mathbf{y}_i\mathbf{y}_i^\top}{\alpha} - \frac{\mathbf{q}_i\mathbf{q}_i^\top}{\beta}$
  $\Delta\mathbf{u} = \mathbf{B}_i\mathbf{x} - \mathbf{u}$
  $\mathbf{z}_i = \mathbf{x}$
  Send $\Delta\mathbf{u}, \mathbf{y}_i, \mathbf{q}_i, \alpha, \beta$ to the master

Note that steps at worker i is devoted to performing the update in (4.11). Using the computed matrix $\mathbf{B}_i$, node i evaluates the vector $\Delta\mathbf{u}$. Then, it sends the vectors $\Delta\mathbf{u}$, $\mathbf{y}_i$, and $\mathbf{q}_i$ as well as the scalars α and β to the master node. The master node uses the variation vectors $\Delta\mathbf{u}$ and $\mathbf{y}$ to update $\mathbf{u}$ and $\mathbf{g}$. Then, it performs the update $\mathbf{x}^{t+1} = (\mathbf{B}^{t+1})^{-1}\left(\mathbf{u}^{t+1} - \mathbf{g}^{t+1}\right)$ by following the efficient procedure presented in (4.16)–(4.17). A more detailed version of Algorithm 1 with exact indices is presented in Appendix B.1.

We define *epochs* $\{T_m\}_m$ by setting $T_1 = 0$ and the following recursion:

$$\begin{aligned} T_{m+1} &= \min\{t : \text{ each machine made at least 2 updates on the interval } [T_m, t]\} \\ &= \min\{t : t - D_i^t \geq T_m \text{ for all } i = 1, .., M\}. \end{aligned}$$

The proof of the following simple lemma is provided in the Appendix.

Lemma 1. *Algorithm 4.1 iterates satisfy* $\mathbf{x}^t = \left(\frac{1}{n}\sum_{i=1}^n \mathbf{B}_i^t\right)^{-1}\left(\frac{1}{n}\sum_{i=1}^n \mathbf{B}_i^t\mathbf{z}_i^t - \frac{1}{n}\sum_{i=1}^n \nabla f_i(\mathbf{z}_i^t)\right)$.

The result in Lemma 1 shows that explicit relationship between the updated variable $\mathbf{x}^t$ based on the proposed DAve-QN and the local information at the workers. We will use this update to analyze DAve-QN.

Proposition 1 (Epochs' properties)**.** *The following relations between epochs and delays hold:*

- *For any* $t \in [T_{m+1}, T_{m+2})$ *and any* $i = 1, 2, \ldots, n$ *one has* $t - D_i^t \in [T_m, t)$.
- *If delays are uniformly bounded, i.e., there exists a constant* d *such that* $d_i^t \leq d$ *for all* i *and* t, *then for all* m *we have* $T_{m+1} - T_m \leq D := 2d + 1$ *and* $T_m \leq Dm$.
- *If we define average delays as* $\overline{d^t} := \frac{1}{n}\sum_{i=1}^n d_i^t$, *then* $\overline{d^t} \geq (n-1)/2$. *Moreover, assuming that* $\overline{d^t} \leq (n-1)/2 + \overline{d}$ *for all* t, *we get* $T_m \leq 4n(\overline{d} + 1)m$.

Clearly, without visiting every function we can not converge to $\mathbf{x}^*$. Therefore, it is more convenient to measure performance in terms of the number of passed epochs, which can be considered as our

alternative counter for time. Proposition 1 explains how one can get back to the iterations time counter assuming that delays are bounded uniformly or on average. However, uniform upper bounds are rather pessimistic which motivates the convergence in epochs that we consider.

4.3 Convergence Analysis

In this section, we study the convergence properties of the proposed distributed asynchronous quasi-Newton method. To do so, we first assume that the following conditions are satisfied.

Assumption 1. *The component functions f_i are L-smooth and μ-strongly convex, i.e., there exist positive constants $0 < \mu \leq L$ such that, for all i and $\mathbf{x}, \hat{\mathbf{x}} \in \mathbb{R}^p$*

$$\mu\|\mathbf{x} - \hat{\mathbf{x}}\|^2 \leq (\nabla f_i(\mathbf{x}) - \nabla f_i(\hat{\mathbf{x}}))^T(\mathbf{x} - \hat{\mathbf{x}}) \leq L\|\mathbf{x} - \hat{\mathbf{x}}\|^2. \tag{4.18}$$

Assumption 2. *The Hessians $\nabla^2 f_i$ are Lipschitz contunuous, i.e., there exists a positive constant $\tilde{L}$ such that, for all i and $\mathbf{x}, \hat{\mathbf{x}} \in \mathbb{R}^p$, we can write $\|\nabla^2 f_i(\mathbf{x}) - \nabla^2 f_i(\hat{\mathbf{x}})\| \leq \tilde{L}\|\mathbf{x} - \hat{\mathbf{x}}\|$.*

It is well-known and widely used in the literature on Newton's and quasi-Newton methods [177–180] that if the function f_i has Lipschitz continuous Hessian $x \mapsto \nabla^2 f_i(x)$ with parameter $\tilde{L}$ then

$$\left\|\nabla^2 f_i(\tilde{\mathbf{x}})(\mathbf{x} - \hat{\mathbf{x}}) - (\nabla f_i(\mathbf{x}) - \nabla f_i(\hat{\mathbf{x}}))\right\| \leq \tilde{L}\|\mathbf{x} - \hat{\mathbf{x}}\| \max\left\{\|\mathbf{x} - \tilde{\mathbf{x}}\|, \|\hat{\mathbf{x}} - \tilde{\mathbf{x}}\|\right\}, \tag{4.19}$$

for any arbitrary $\mathbf{x}, \tilde{\mathbf{x}}, \hat{\mathbf{x}} \in \mathbb{R}^p$. See, for instance, Lemma 3.1 in [177].

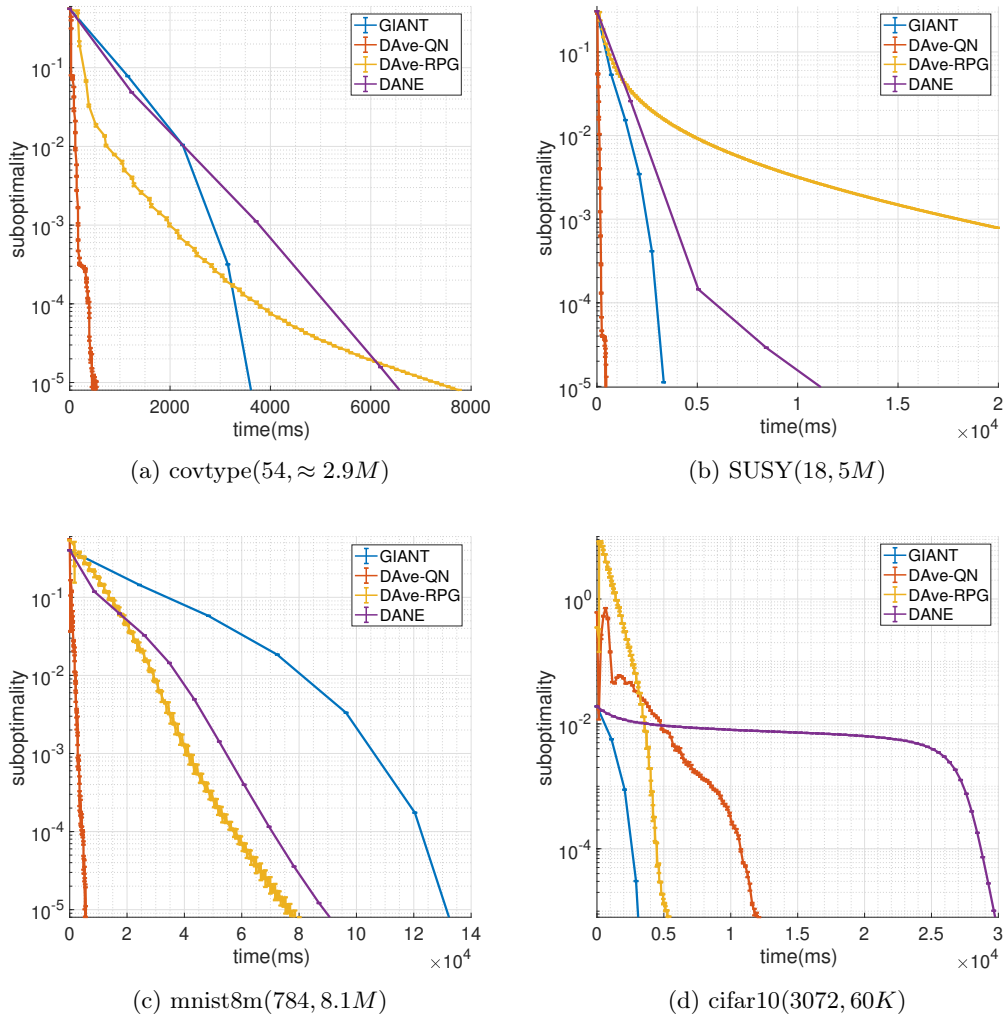
Lemma 2. *Consider the DAve-QN algorithm summarized in Algorithm 4.1. For any i, define the residual sequence for function f_i as $\sigma_i^t := \max\{\|\mathbf{z}_i^t - \mathbf{x}^*\|, \|\mathbf{z}_i^{t-D_i^t} - \mathbf{x}^*\|\}$ and set $\mathbf{M}_i = \nabla^2 f_i(\mathbf{x}^*)^{-1/2}$. If Assumptions 1 and 2 hold and the condition $\sigma_i^t < \mu/(3\tilde{L})$ is satisfied then a Hessian approximation matrix $\mathbf{B}_i^t$ and its last updated version $\mathbf{B}_i^{t-D_i^t}$ satisfy*

$$\left\|\mathbf{B}_i^t - \nabla^2 f_i(\mathbf{x}^*)\right\|_{\mathbf{M}_i} \leq \left[\left[1 - \alpha{\theta_i^{t-D_i^t}}^2\right]^{\frac{1}{2}} + \alpha_3 \sigma_i^{t-D_i^t}\right]\left\|\mathbf{B}_i^{t-D_i^t} - \nabla^2 f_i(\mathbf{x}^*)\right\|_{\mathbf{M}_i} + \alpha_4 \sigma_i^{t-D_i^t}, \tag{4.20}$$

where α, α_3, and α_4 are some positive constants and $\theta_i^{t-D_i^t} = \frac{\|\mathbf{M}_i(\mathbf{B}_i^{t-D_i^t} - \nabla^2 f_i(\mathbf{x}^*))\mathbf{s}_i^{t-D_i^t}\|}{\|\mathbf{B}_i^{t-D_i^t} - \nabla^2 f_i(\mathbf{x}^*)\|_{\mathbf{M}_i}\|\mathbf{M}_i^{-1}\mathbf{s}_i^{t-D_i^t}\|}$ with the convention that $\theta_i^{t-D_i^t} = 0$ in the special case $\mathbf{B}_i^{t-D_i^t} = \nabla^2 f_i(\mathbf{x}^*)$.

Lemma 2 shows that, if we neglect the additive term $\alpha_4 \sigma_i^{t-D_i^t}$ in (4.20), the difference between the Hessian approximation matrix $\mathbf{B}_i^t$ for the function f_i and its corresponding Hessian at the optimal point $\nabla^2 f_i(\mathbf{x}^*)$ decreases by following the update of Algorithm 4.1. To formalize this claim and show that the additive term is negligible, we prove in the following lemma that the sequence of errors $\|\mathbf{x}^t - \mathbf{x}^*\|$ converges to zero R-linearly which also implies linear convergence of the sequence σ_i^t.

Lemma 3. *Consider the DAve-QN method outlined in Algorithm 4.1. Further assume that the conditions in Assumptions 1 and 2 are satisfied. Then, for any $r \in (0, 1)$ there exist positive constants $\epsilon(r)$ and $\delta(r)$ such that if $\|\mathbf{x}^0 - \mathbf{x}^*\| < \epsilon(r)$ and $\|\mathbf{B}_i^0 - \nabla^2 f_i(\mathbf{x}^*)\|_{\mathbf{M}} < \delta(r)$ for $\mathbf{M} = \nabla^2 f_i(\mathbf{x}^*)^{-1/2}$ and $i = 1, 2, \ldots, n$, the sequence of iterates generated by DAve-QN satisfy $\|\mathbf{x}^t - \mathbf{x}^*\| \leq r^m \|\mathbf{x}^0 - \mathbf{x}^*\|$ for all $t \in [T_m, T_{m+1})$.*



(a) covtype$(54, \approx 2.9M)$ (b) SUSY$(18, 5M)$

(c) mnist8m$(784, 8.1M)$ (d) cifar10$(3072, 60K)$

Figure 4.2: Expected suboptimality versus time. The first and second numbers indicated next to the name of the datasets are the variables p and n respectively.

The result in Lemma 3 shows that the error for the sequence of iterates generated by the DAve-QN method converges to zero at least linearly in a neighborhood of the optimal solution. Using this result, in the following theorem we prove our main result, which shows a specific form of superlinear convergence.

Theorem 1. *Consider the proposed method outlined in Algorithm 4.1. Suppose that Assumptions 1 and 2 hold. Further, assume that the required conditions for the results in Lemma 2 and Lemma 3 are satisfied. Then, the sequence of residuals* $\|\mathbf{x}^t - \mathbf{x}^*\|$ *satisfies* $\lim_{t\to\infty} \frac{\max_{t\in[T_{m+1},T_{m+2})} \|\mathbf{x}^t - \mathbf{x}^*\|}{\max_{t\in[T_m,T_{m+1})} \|\mathbf{x}^t - \mathbf{x}^*\|} = 0.$

Proof. See Appendix B.

The result in Theorem 1 shows that the maximum residual in an epoch divided by the maximum residual for the previous epoch converges to zero. This observation shows that there exists a subsequence of residuals $\|\mathbf{x}^t - \mathbf{x}^*\|$ that converges to zero superlinearly.

Extension to non-strongly convex objectives: For convex objectives f that are not strongly convex, the Hessian matrix $\nabla^2 f(x)$ can be singular which causes stability issues with the Hessian approximation of BFGS-type methods. A popular approach to solve this issue is to add a regularizing $\varepsilon\|x\|^2$ term to the objective for an appropriately chosen $\varepsilon > 0$ that is small enough. This leads to a strongly convex approximation of the objective and regularizes the Hessian. With this approach, our algorithm and analysis can be applied to convex functions that are not strongly convex if the domain is compact, following similar ideas to [181, Section 5.4].

4.4 Experiments

We conduct our experiments on four datasets (`covtype`, `SUSY`, `mnist8m`, `cifar10`) from the LIBSVM library [132].[1] For the first two datasets, the objective considered is a binary logistic regression problem $f(x) = \frac{1}{n}\sum_{i=1}^{n} \log(1 + \exp(-b_i a_i^T x) + \frac{\lambda}{2}\|x\|^2$ where $a_i \in \mathbb{R}^p$ are the feature vectors and $b_i \in \{-1, +1\}$ are the labels. The other two datasets are about multi-class classification instead of binary classification. For comparison, we used two other algorithms designed for distributed optimization:

- Distributed Average Repeated Proximal Gradient (**DAve-RPG**, [58]). It is a recently proposed competitive state-of-the-art asynchronous method for first-order distributed optimization, numerically demonstrated to outperform incremental aggregated gradient methods ([139, 170]) and synchronous proximal gradient methods [58].

- Distributed Approximate Newton (**DANE**, [157]). This is a well-known Newton-like method that does not require a parameter server node but performs reduce operations at every step.

- Globally Improved Approximate Newton (**GIANT**, [182]). It is a distributed synchronous communication-efficient Newton-type method that reduces to the DANE method for quadratic objectives. For more general smooth objectives with a Lipschitz Hessian, it enjoys a local linear-quadratic convergence rate. It has also been shown in [182] to numerically outperform the DANE [157], L-BFGS [183], and Accelerated Gradient Descent [178] methods on a number of problems.

In our experiments, we did not implement algorithms that require shared memory (such as ASAGA [144] or Hogwild! [62]) because, in our setting of the master/worker communication model, the memory is not shared. Since the focus of this work is mainly on asynchronous algorithms where the communication delays are the main bottleneck, for fairness reasons, we are also not comparing our method with some other synchronous algorithms such as DISCO ([159]) that would not support asynchronous computations. Our code is publicly available at `https://github.com/DAve-QN/source`.

The experiments are conducted on XSEDE Comet CPUs (Intel Xeon E5-2680v3 2.5 GHz) [133]. For DAve-QN and DAve-RPG we build a cluster of 17 processes in which 16 of the processes are workers and one is the master. The DANE method does not require a master so we use 16 workers for its experiments. We split the data randomly among the processes so that each has the same

[1] We use all the datasets without any pre-processing except for the smaller-scale covtype dataset, which we enlarged 5 times for bigger scale experiments using the approach in [160].

amount of samples. In our experiments, Intel MKL 11.1.2 and MVAPICH2/2.1 are used for the BLAS (sparse/dense) operations and we use MPI programming compiled with mpicc 14.0.2.

For the methods' parameters, the best options provided by the method authors are used. For DAve-QN, we use a unit step size that is asymptotically optimal. We observed that DAve-QN converged to the optimal point for step size=1 (although sometimes progress was a bit slower in the beginning) and therefore we used it in all experiments. For DAve-RPG the step size $\frac{1}{L}$ is used where L is found by a standard backtracking line search similar to [184]. DANE has two parameters, η and μ. As recommended by the authors, we use $\eta = 1$ and $\mu = 3\lambda$. We tuned λ to the dataset, choosing $\lambda = 1$ for the `mnist8m` and `cifar10` datasets, $\lambda = 0.001$ for `SUSY` and $\lambda = 0.1$ for the `covtype`. The results we obtained were also qualitatively similar when we used the common choice $\lambda = 1/n$ for all the methods. Since DANE requires a local optimization problem to be solved, we use SVRG ([185]) as its local solver where its parameters are selected based on the experiments in [157].

We note that DAve-QN has a computation complexity of $O(p^2)$ per iteration which is due to the matrix-vector multiplications, a memory complexity of $O(p^2)$ because of storing a $p \times p$ matrix on the workers and master, and communication complexity of $O(p)$ as it only requires to send some scalars and vectors in $\mathbb{R}^p$. More specifically, DAve-QN exchanges $3p + 2$ numbers (slave to master) and p numbers (master to slave), whereas DAve-RPG exchanges $2p$ numbers and DANE exchanges $4p$ numbers (counting both slave and master side) which are all $O(p)$.

Our results are summarized in Figure 4.2 where we report the average suboptimality versus time on a logarithmic y-axis. For linearly convergent algorithms, the slope of the performance curves determines the convergence rate. The plots contain the error bars based on 5 runs. Since we are using silent machines from XSEDE/Comet that run an experiment in isolation, we do not see much variability in the runs. DANE method is the slowest on these datasets, but it does not need a master, therefore it can apply to multi-agent applications ([67]) where master nodes are often not available. We observe that DAve-QN performs significantly better on all the datasets except `cifar10`, illustrating the superlinear convergence behavior provided by our theory compared to other methods. For the `cifar10` dataset, p is the largest and GIANT is the fastest. Although DAve-QN starts faster than DAve-RPG, DAve-RPG has a cheaper iteration complexity ($O(p)$ compared to $O(p^2)$ of DAve-QN) and becomes eventually faster.

4.5 Conclusion

In this work, we focused on the problem of minimizing a large-scale empirical risk minimization in a distributed manner. We used an asynchronous architecture that requires no global coordination between the master node and the workers. Unlike distributed first-order methods that follow the gradient direction to update the iterates, we proposed a distributed averaged quasi-Newton (DAve-QN) algorithm that uses a quasi-Newton approximate Hessian of the workers' local objective function to update the decision variable. In contrast to second-order methods that require computation of the local functions Hessians, the proposed DAve-QN only uses gradient information to improve the convergence of first-order methods in ill-conditioned settings. It is worth mentioning that the computational cost of each iteration of DAve-QN is $\mathcal{O}(p^2)$, while the size of the vectors that are communicated between the master and workers is $\mathcal{O}(p)$. Our theoretical results show that the sequence of iterates generated at the master node by following the update of DAve-QN converges

superlinearly to the optimal solution when the objective functions at the workers are smooth and strongly convex. Our results hold for both bounded delays and unbounded delays with a bounded time-average. Numerical experiments illustrate the performance of our method.

The choice of the step size in the initial stages of the algorithm is the key to getting good overall iteration complexity for second-order methods. Investigating several line search techniques developed for BFGS and adapting them to the distributed asynchronous setting is a future research direction of interest. Another promising direction would be developing Newton-like methods that can go beyond superlinear convergence while preserving communication complexity. Finally, investigating the dependence of the convergence properties on the sample size m_i of each machine i would be interesting, in particular, one would expect the performance in terms of communication complexity to improve if the sample size of each machine is increased.

Limitations. In this chapter, we assume that each worker can store the approximation of the Hessian matrix locally which is infeasible for machines that have storage limitations. Therefore, limited-memory BFGS (LBFGS) methods can be used to tackle this challenge. However, this requires further analysis of the convergence. Moreover, in order to ensure global convergence, one can use an adaptive step size based on line search methods.

Chapter 5

Asynchronous Communications for Optimization Methods on Cloud

In this chapter, we develop a novel framework for machine learning practitioners to implement and dispatch asynchronous optimization methods with custom asynchronous execution models on cloud and distributed memory platforms. While we implemented an asynchronous quasi-Newton method using the MPI framework in Chapter 4, we observed a lack of necessary tools and frameworks for implementing it on cloud platforms. Hence, we propose ASYNC which is a bulk processing computing framework that is built on top of Spark [72] and provides necessary tools and API so that a wide range of asynchronous optimization algorithms can easily be implemented on distributed cloud platforms. ASYNC supports both first-order and second-order optimization methods with communication models ranging from synchronous to fully asynchronous. Moreover, ASYNC allows the efficient implementation of optimization methods that require *history* of gradients. Operations on a history of gradients augment the noise (stochasticity) to improve convergence [186]. The detailed contributions of ASYNC are:

- A novel framework for machine learning practitioners to implement and dispatch asynchronous machine learning applications with custom asynchronous communication models on cloud and distributed platforms. ASYNC introduces three modules to cloud engines, *ASYNCcoordinator*, *ASYNCbroadcaster*, and *ASYNCscheduler* to enable the asynchronous gather, broadcast, and schedule of tasks and results.

- An efficient history recovery strategy implemented with the *ASYNCbroadcaster* and bookkeeping *attributes*, to facilitate the implementation of variance-reduced optimization methods that operate on historical gradients.

- A robust programming model with extensions to the Spark API that enables the implementation of asynchrony and history while preserving the in-memory and fault-tolerant features of Spark.

- A demonstration of ease-of-implementation in ASYNC with the implementation and performance analysis of the stochastic gradient descent (SGD) [187] algorithm and its asynchronous variant. Our results demonstrate that asynchronous SAGA (ASAGA) [63] and asynchronous SGD

(ASGD) outperform their synchronous variants up to 4 times on a distributed system with stragglers.

5.1 Introduction

The challenges of dealing with huge datasets have lead to the development of optimizations methods with *asynchrony* and *history*. Asynchronous optimization methods reduce worker idle times and mitigate communication costs. Operations on a history of gradients augment the noise (stochasticity) to improve convergence [186]. Distributed optimization methods operate on batches of data and thus have to be implemented in cluster-computing engines with a bulk (coarse-grained) computation model.

Several general coarse-grained distributed data processing systems exist. Hadoop [188] and Spark [72] are based on the iterative map-reduce model but use a synchronous iterative communication pattern. Thus, because of not supporting asynchrony, their execution is vulnerable to the diverse performance profile caused by slow workers, i.e., stragglers, and network latency. Also, history cannot be efficiently maintained in these engines as it requires storing *bulky worker-results*, and introduces overheads to their lineage-based [72] or checkpointing fault-tolerant implementations.

Recently, several coarse-grained machine learning engines such as Petuum [75] and Litz [76], have adopted the parameter server [74] architecture to implement asynchronous communication between nodes with push-pull operations. Asynchrony in distributed optimization methods is implemented with consistency models, i.e., barriers, expressed via a dependency graph that maintains a trade-off between system efficiency and algorithm convergence. Parameter server paradigms implement a specific class of consistency models, i.e., stale synchronous parallel (SSP) paradigms using a fixed dependency graph, which uses a static staleness threshold to control worker wait times. However, recent advancements in distributed optimization [68, 69] demand a wider range of customized consistency models (CCMs), often defined by the user such as throttled-release [69], that control worker wait times using parameters such as worker-task-completion time [68] and require to adaptively adjust the parameters at runtime. CCMs can not be implemented in available parameter server frameworks as they need the underlying dependency graph to adaptively be reconfigured at runtime. Also, Petuum does not support history and Litz preserves the history by periodic checkpointing with significant overheads. Among parameter servers, Glint [74] which is built on top of Spark is the closest to our framework. Glint supports history but does not the implementation CCMs. Also, the consistency model in Glint is optimized for computing topic models and is not optimized for distributed optimization methods. Other distributed parameter server frameworks such as DistBelief [189] and TensorFlow [78] are specialized for deep learning applications and thus do not naturally support consistency models and history.

Amongst the fine-grained distributed data processing systems, primarily used for streaming applications, RAY [190] and Flink [191] support asynchronous function invocations with dynamic data flow graphs [192]. However, these frameworks do not support CCMs and are primarily designed for fine-grained tasks, and thus cannot naturally extend to a bulk-processing engine. Also, while streaming engines (because of processing fine tasks) can store local results and intermediate data on workers to support history with low overhead, bulk processing engines cannot efficiently store the worker-results because of processing coarse tasks.

In principle, with massive system engineering efforts, machine learning practitioners can implement one-off asynchronous optimization methods with re-engineering systems and interfaces. However, this comes at the cost of pushing system challenges such as scheduling, bookkeeping, and fault tolerance to the application developer. For example, Spark can support history if previous worker-results are stored on disk and checkpointed; this will lead to storage overheads. To implement CCMs the entire Spark engine has to change to support asynchronous execution. An expert MPI programmer can use asynchronous primitives to implement SSP [193]. However, this leads to increased program complexity and the complexity will increase if customized consistency models were to be implemented. Noteworthy, MPI does not have robust support for fault tolerance and thus is typically not used for cloud computing.

This work presents ASYNC, a bulk processing cloud-computing framework, built on top of Spark, that supports the implementation of distributed optimization methods with asynchrony and history. ASYNC implements an asynchronous execution to Spark's engine and enables the workers and/or the master to *bookkeep* (log) system-specific and application parameters. The asynchronous execution paradigm and the bookkeeping structures work together to construct a dynamic dependence graph for the implementation of custom consistency models and to recover history with a partial broadcast of model parameters.

Related work. To mitigate the negative effects of stale gradients on convergence, numerous optimization methods support asynchrony. The most widely used optimization algorithms with asynchrony are stochastic gradient methods [62, 189] and coordinate descent algorithms [194]. Other works implement asynchrony by altering the execution bound staleness [17,195], theoretically adapting the method to the stale gradients [196], and by using barrier control strategies [68, 197]. Variance reduction approaches use the history of gradients to reduce the variance incurred by stochastic gradients and to improve convergence [186, 198, 199]. Numerous algorithms implement variance reduction techniques in asynchronous methods, some of which include ASAGA and DisSVRG [198] which support asynchrony.

The demand for large-scale machine learning has led to the development of numerous cloud and distributed computing frameworks. Commodity distributed dataflow systems such as Hadoop [188] and Spark [72], as well as libraries implemented on top of them such as Mllib [135], are optimized for coarse-grained, often bulk synchronous, parallel data transformations and thus do not provide asynchrony in their execution models [72,188,200,201]. Recent work has modified frameworks such as Spark to support asynchronous optimization methods. ASIP [73] introduces a communication layer to Spark to support asynchrony, however, it only implements the asynchronous parallel consistency model [75] and does not support history. Glint [74] integrates the parameter server model on Spark. However, it is designed for topic models with a specialized consistency model.

Parameter server architectures such as [75, 76] are widely used in distributed machine learning since they support asynchrony in their execution models using a static dependency graph. Petuum [75] implements the SSP execution model. Other parameter server frameworks include MLNET [202] and Litz [76]. MLNET deploys a communication layer that uses tree-based overlays to implement distributed aggregation to only communicate the aggregated updates without the support for individual communication of worker-results. These implementations do not support custom consistency models required by asynchronous optimization methods nor the history of gradients. Finally, numerous distributed computing frameworks have been developed to support specific applications. For example,

DistBelief [189] and TensorFlow [78] support deep learning applications while fine-grained data processing systems such as RAY [190] and Flink [191] are designed for streaming problems. The frameworks can not be naturally extended to support mini-batch optimization methods that require coarse-grained computation models.

5.2 Preliminaries

Distributed machine learning often results in solving an optimization problem in which an objective function is optimized by iteratively updating the *model parameters* until convergence. Distributed implementation of optimization methods includes workers that are assigned tasks to process parts of the training data, and one or more servers, i.e., masters, that store and update the model parameters. Distributed machine learning models often result in the following structure:

$$\min_{w\in\mathbb{R}^d} F(w) = \frac{1}{m}\sum_{i=1}^{m} f^{(i)}(w) \tag{5.1}$$

where w is the model parameter to be learned, m is the number of workers, and $f^{(i)}(w)$ is the local loss function computed by worker i based on its assigned training data. Each worker has access to n_i data points, where the local cost has the form

$$f^{(i)}(w) := \sum_{j=1}^{n_i} \bar{f}_j^{(i)}(w) \tag{5.2}$$

for some loss functions $\bar{f}_j^{(i)} : \mathbb{R}^d \to \mathbb{R}$. For example, in supervised learning, given an input-output pair $\big(x_{ij}, y_{ij}\big)$, the loss function can be $\bar{f}_j^{i}(w) = \ell(\langle w, \phi(x_{ij})\rangle, y_{ij})$ where ϕ is a fixed function of choice and $\ell(\cdot,\cdot)$ is a convex loss function that measures the loss if y_{ij} is predicted from x_{ij} based on the model parameter w. This setting covers empirical risk minimization problems in machine learning that include linear and non-linear regression, and other classification problems such as logistic regression [186]. In particular, if $\phi(x) = x$ and the $\ell(\cdot,\cdot)$ function is the square of the Euclidean distance function, we obtain the familiar least-squares problem

$$\bar{f}_j^{i}(w) = \|x_{ij}^T w - y_{ij}\|^2 \tag{5.3}$$

where

$$f^{(i)}(w) := \sum_{j=1}^{n_i} \bar{f}_j^{(i)}(w) = \|A_i w - b_i\|^2 \tag{5.4}$$

with $b_i = \{y_{ij}\}_{j=1}^{n_i}$ is a column vector of length n_i and $A_i \in \mathbb{R}^{n_i\times d}$ is called the *data matrix* as its j-th row is given by the input x_{ij}^T.

In the following, we use the gradient descent (GD) algorithm as an example to introduce stochastic optimization and other terminology used throughout this work such as *mini-batch* size. The introduced terms are used in all optimization problems and are widely used in the machine learning literature. GD iteratively computes the gradient of the loss function $\nabla F(w_k) = \frac{1}{m}\sum_{i=1}^{m} \nabla f^{(i)}(w_k)$ to update the model parameters at iteration k. To implement gradient descent on a distributed system, each worker i computes its *local gradient* $\nabla f^{(i)}(w^k)$; the local gradients are aggregated by the master when ready. The full pass over the data at every iteration of the algorithm with synchronous updates leads

to large overheads. Distributed stochastic gradient descent (SGD) methods and their variants [203] are, on the other hand, scalable and popular methods for solving (5.1). Distributed SGD replaces the local gradient $\nabla f^{(i)}(w_k)$ with an unbiased stochastic estimate $\tilde{\nabla} f^{(i)}(w_k)$ of it, computed from a subset of local data points:

$$\nabla \tilde{f}^{(i)}(w_k) := \frac{1}{b_i} \sum_{s \in S_{i,k}} \nabla \bar{f}_s^{(i)}(w_k), \tag{5.5}$$

where $S_{i,k} \subset \{1, \ldots, n_i\}$ is a random subset that is sampled with or without replacement at iteration k, and $b_i := |S_{i,k}|$ is the number of elements in $S_{i,k}$ [187], also called the mini-batch size. To obtain desirable accuracy and performance, implementations of stochastic optimization methods require tuning algorithm parameters such as the step size and the mini-batch size in SGD [187].

5.3 Motivation for Asynchrony and History

Asynchrony is implemented to improve the convergence rate and time-to-solution of optimization methods on cluster-computing platforms with slow machines (stragglers). In distributed optimization, workers compute the local gradients of the objective function and then communicate the computed gradients to the server. To proceed to the next iteration of the algorithm, the server updates the shared model parameters with the received gradients, broadcasts the most recent model parameter, and schedules new tasks. In asynchronous optimization, the server can proceed with the update and broadcast of the model parameters without having to wait for all worker tasks to be complete. This asynchrony allows the algorithm to make progress in the presence of stragglers which is known as an increase in hardware efficiency [195]. However, this progress in computation comes at a cost, the asynchrony inevitably adds *staleness* to the system wherein some of the workers compute gradients using model parameters that may be several gradient steps behind the most updated set of model parameters which can lead to poor convergence. This is also referred to as a worsening in statistical efficiency [203].

Asynchronous optimization methods are formulated and implemented with properties that balance statistical efficiency and hardware efficiency to maximize the performance of the optimization methods on distributed systems. Consistency models, i.e., barrier control strategies, are used to design asynchronous optimization methods that enable this balance. Barriers in asynchronous algorithms determine if a worker should proceed to the next iteration or if it should wait until a specific number of workers have communicated their results to the server. The most well-known barrier control strategy is the Stale Synchronous Parallel (SSP), in which workers synchronize when staleness (determined by the number of stragglers) exceeds a threshold. ASYNC supports SSP and also facilitates the implementation of custom consistency models that apply barriers based on parameters such as worker-task-completion time and scheduling delays.

History augments the noise from stochastic gradients to improve the convergence rate of the optimization method. Distributed optimization methods, used in machine learning applications, are typically stochastic [187]. Stochastic optimization methods use a noisy gradient computed from random data samples instead of the true gradient which can lead to poor convergence. Variance reduction techniques, used in both synchronous and asynchronous optimization, augment the noisy gradient to reduce this variance. A class of variance-reduced asynchronous algorithms, that have

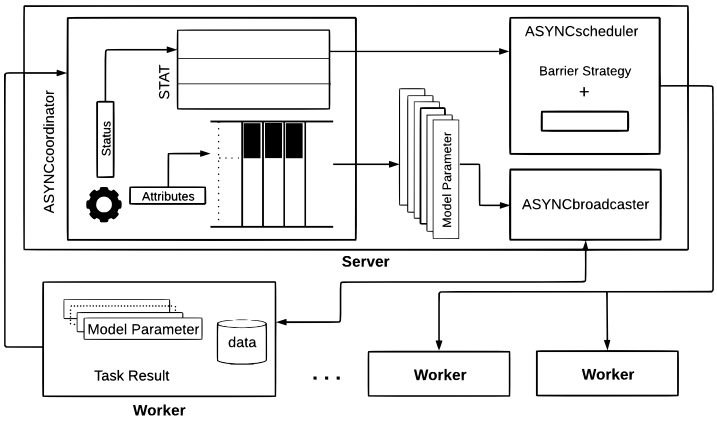


Figure 5.1: An overview of the ASYNC framework.

led to significant improvements over traditional methods, memorize the gradients computed in previous iterations, i.e., historical gradients [166]. Historical gradients cannot be implemented in cluster-computing engines such as Spark primarily because Spark can only broadcast the entire history of the model parameters which can be very large and lead to significant overheads.

5.4 ASYNC: A Cloud Computing Framework with Asynchrony and History

ASYNC is a framework, built on top of Spark [72], for the implementation and execution of asynchrony and history in optimization algorithms while retaining the map-reduce model, scalability, and fault tolerance of state-of-the-art cluster-computing engines. Figure 5.1 demonstrates an overview of the ASYNC engine. The three main modules in ASYNC are the *ASYNCcoordinator*, *ASYNCbroadcaster*, and *ASYNCscheduler*. ASYNC also collects and stores *bookkeeping structures*. These structures are communicated between the workers and the master and are either system-specific, i.e., *status*, or are related to the application, i.e., *attributes*. This section elaborates on how the internal elements of ASYNC work together to facilitate the implementation of asynchrony and history.

Bookkeeping structures in ASYNC. Bookkeeping structures are used by the main modules of ASYNC to enable the implementation of asynchrony and history. These structures are collected by ASYNC at runtime and stored on the master. With the help of the ASYNCcoordinator, each worker communicates to the master, application-specific attributes such as task results and the mini-batch size. Workers' recent status, such as worker staleness, average-task-completion time, and availability[1] are also logged and stored in a table called STAT with the help of the ASYNCcoordinator.

Implementing asynchrony with the ASYNCcoordinator, ASYNCscheduler, and the status structures. To implement asynchrony, ASYNC implements a dynamic task graph computation model which uses the consistency model to dynamically determine executing tasks and their assigned workers. The execution of tasks on workers is automatically triggered by the system using a computation graph. Task and data objects are the nodes in this graph and the edges are the dependency amongst nodes/tasks. The computation graph in classic consistency models such as SSP does not

[1] A worker is available if it is not executing a task and unavailable otherwise.

change at runtime because the models do not rely on runtime information such as the system state. However, many CCMs take information from the current state of the system as input and couple this information with the barrier control strategy to dynamically build the computation graph. To implement CCMs, the ASYNCcoordinator periodically communicates with the workers to update system-specific parameters in the STAT table. The ASYNCscheduler uses the parameters in STAT and a user-defined barrier control strategy to update the computation graph. Then the computation graph is executed to apply the desired consistency model.

Implementing history with the ASYNCbroadcaster and the attributes. In each iteration of an optimization method with history, the computed gradients from previous iterations are used together with the current model parameters to update the model parameters. Implementing history in a coarse-grained computation engine via explicitly storing bulky worker-results, i.e., previous gradients, leads to significant storage overheads. A fault-tolerant execution will also have overheads as large gradients have to be periodically check-pointed or recomputed explicitly using a lineage.

ASYNC does not explicitly store, communicate, or compute past gradients. Instead, we use the approach from [186] in which the history of past gradients is recovered, when needed, using previous model parameters. Recovering history has low storage and computation overheads in coarse-grained computation models. By recovering history, workers in ASYNC do not need to store any previously computed gradients and only the previous model parameters are stored on the master. The cost of storing the model parameters has an inverse relation to the batch size [187] and thus reduces as the granularity of tasks increases, e.g., larger batch sizes. Also, to recover a past gradient, a worker only needs to subtract its recent model parameter from the previous model parameter that is broadcasted to it from the master; the approach in [186] is then used to update the master-side model parameters based on the history. The ASYNCbroadcaster in ASYNC is responsible for the asynchronous broadcast of model parameters between the master and individual workers. Attributes such as the mini-batch size, required by the master to apply history to its model parameters, are also broadcast using the ASYNCbroadcaster.

5.5 Programming with ASYNC

To use ASYNC, developers are provided an additional set of ASYNC-specific functions, on top of what Spark provides, to access the bookkeeping structures and to implement asynchrony and history. The programming model in ASYNC is close to that of Spark [72]. It operates on resilient distributed datasets (RDD) to preserve the fault-tolerant and in-memory execution of Spark. The ASYNC-specific functions also either transform the RDDs, known as *transformations* in Spark, or conduct lazy actions. In this section, ASYNC's programming model and API are first discussed. We then show the implementation of SGD and its asynchronous variant which uses a CCM. A well-known history-based optimization method called SAGA [166] and its asynchronous variant with a CCM is also implemented.

5.5.1 The ASYNC Programming Model

Asynchronous Context (AC) is the entry to ASYNC and should be created only once in the beginning. The ASYNCscheduler, the ASYNCbroadcaster, and the ASYNCcoordinator communicate via the AC and with this communication create barrier controls, broadcast variables, and store workers'

Table 5.1: Transformations, actions, and methods in ASYNC. AC is ASYNCcontext and Seq[T] is a sequence of elements.

Actions	ASYNCreduce(f:(T,T) ⇒ T, AC) ASYNCaggregate(zeroValue: U) (seqOp: (U, T) ⇒ U, combOp: (U, U) ⇒ U), AC)	Reduces the elements of the RDD using the specified associative binary operator. Aggregates the elements of the partition using the combine functions and a neutral "zero" value.
Transformations	ASYNCbarrier(f:T ⇒ Bool, Seq[T])	Returns a RDD containing elements that satisfy a predicate *f*.
Methods	ASYNCcollect() ASYNCcollectAll() ASYNCbroadcast(T) AC.STAT AC.hasNext()	Returns a task result. Returns a task result and its attributes. Creates a dynamic broadcast variable. Returns the current status of all workers. Returns true if a task result exists.

task results and status. AC maintains the bookkeeping structures and ASYNC-specific functions, including actions and transformations that operate on RDDs. Workers use ASYNC functions to interact with AC and to store their results and attributes in the bookkeeping structures. The server queries AC to update the model parameters or to access workers' status. Table 5.1 lists the main functions available in ASYNC. We show the signature of each operation by demonstrating the type parameters in square brackets.

Collective operations in ASYNC. ASYNCreduce is an action that aggregates the elements of the RDD on the worker and returns the result to the server. ASYNCreduce differs from Spark's *reduce* in two ways. First, Spark aggregates data across each partition and then combines the partial results together to produce a final value. However, ASYNCreduce executes only on the worker and for each partition. Secondly, *reduce* returns only when all partial results are combined on the server, but ASYNCreduce returns immediately. Task results on the server are accessed using the *ASYNCcollect* and *ASYNCcollectAll* methods. ASYNCcollect returns task results in FIFO (first-in-first-out) order and also returns the worker attributes. The workers' status can also be accessed with *ASYNC.STAT*.

Barrier and broadcast in ASYNC. ASYNCbarrier is a *transformation*, i.e., a deterministic operation which creates a new RDD based on the workers' status. ASYNCbarrier takes the recent status of workers, *STAT*, and decides which workers to assign new tasks to, based on a user-defined function. For example, for a fully asynchronous barrier model, the following function is declared: $f : STAT.foreach(true)$. In Spark, broadcast parameters are "broadcast variable" objects that wrap around the to-be-broadcast value. *ASYNCBroadcast* also uses broadcast variables and similar to Spark, the method *value* can be used to access the broadcast value. However, ASYNCbroadcast differs from the broadcast implementation in Spark since it has access to an *index*. The index is used internally by ASYNCbroadcast to get the ID of the previously broadcast variables for the specified index. ASYNCbroadcast eliminates the need to broadcast values when accessing the history of broadcast values.

5.5.2 Case Studies

The robust programming model in ASYNC provides control of low-level features in both the algorithm and the execution platform to facilitate the implementation of asynchrony and history in optimization

methods. The following demonstrates the implementation of well-known asynchronous optimization methods ASGD and ASAGA in ASYNC as examples.

Algorithm 5.1 The SGD Algorithm

Input : points, numIterations, learning rate α_i, sampling rate b
Output : model parameter w

```
1 for i = 1 to numIterations do
2     w_br = sc.broadcast(w)
3     gradient = points.sample(b).map(p ⇒ ∇f_p(w_br.value)). reduce(_+_)
4     w -= α_i * gradient
5 return w
```

Algorithm 5.2 The ASGD Algorithm

Input : points, numIterations, learning rate α_i, sampling rate b
Output : model parameter w

```
1 AC = new ASYNCcontext
2 for i = 1 to numIterations do
3     w_br = sc.broadcast(w)
4     points.ASYNCbarrier(f, AC.STAT).sample(b).map(p ⇒ ∇f_p(w_br.value)) .ASYNCreduce(_+_, AC)
5     while AC.hasNext() do
6         gradient= AC.ASYNCcollect()
7         w -= α_i * gradient
8 return w
```

ASGD with ASYNC. An implementation of mini-batch stochastic gradient descent (SGD) using the map-reduce model in Spark is shown in Algorithm 5.1. The map phase applies the gradient function on the input data independently on workers. The reduce phase has to wait for all the map tasks to be complete. Afterward, the server aggregates the task results and updates the model parameter w. The asynchronous implementation of SGD in ASYNC is shown in Algorithm 5.2. With only a few extra lines from the ASYNC API, colored in blue, the synchronous implementation of SGD in Spark is transformed to ASGD. An ASYNCcontext is created in line 1 and is used in line 4 to create a barrier using the user-defined CCM indicated by f and based on the current workers' status, AC.STAT. The partial results from each partition are then obtained and stored in AC in line 4. Finally, these partial results are accessed in line 6 and are used to update the model parameter in line 7.

ASAGA with ASYNC. The SAGA implementation in Spark is shown in Algorithm 5.3. This implementation is inefficient and not practical for large datasets as it needs to synchronously broadcast a table of all stored model parameters to each worker, colored in red in Algorithm 5.3 line 5. The size of this increases after each iteration and thus broadcasting it leads to large communication overheads. As a result of the overhead, machine learning libraries that are built on top of Spark, such as MLlib [135], do not provide implementations of optimization methods such as SAGA that require the history of gradients. ASYNC resolves the overhead with ASYNCbroadcast. The implementation of ASAGA is shown in Algorithm 5.4. ASYNCbroadcast is used to define a dynamic broadcast in line 4. Then, the broadcast variable is used to compute the historical gradients in line 5. In order to access the last model parameters for sample *index*, the method *value* is called in line 5. As shown in Algorithm 5.4, there is no need to broadcast a table of parameters which allows for efficient implementation of both SAGA and ASAGA in ASYNC.

Algorithm 5.3 The SAGA Algorithm

Input : points, numIterations, learning rate α, sampling rate b, number of points n
Output : model parameter w

averageHistory = 0
store w in table
for *i = 1 to numIterations* **do**
    w_br =sc.broadcast(w)
    (gradient,history) = points.sample(b).map((index,p) $\Rightarrow \nabla f_p(w\_br.value), \nabla f_p(table[index])$).reduce(_+_)
    averageHistory += (gradient - history)∗ b∗n
    w -= α ∗ (gradient - history + averageHistory)
    update table
return w

Algorithm 5.4 The ASAGA Algorithm

Input : points, numIterations, learning rate α, sampling rate b, #points n, #partitions P
Output : model parameter w

AC = new ASYNCcontext
averageHistory = 0
for *i = 1 to numIterations* **do**
    w_br = AC.ASYNCbroadcast(w)
    points.ASYNCbarrier(f,AC.STAT).sample(b).map((index,p)$\Rightarrow \nabla f_p(w\_br.value)$, $\nabla f_p(w\_br.value(index))$).ASYNCreduce(_+_, AC)
    while *AC.hasNext()* **do**
        (gradient,history)= AC.ASYNCcollect()
        averageHistory += (gradient - history)∗ b∗n/P
        w -= α ∗ (gradient - history + averageHistory)
return w

CCMs in ASYNC. To enable the implementation of custom consistency models, ASYNC provides the interface to implement user-defined functions that selectively choose from available workers based on their status. Listing 5.1 demonstrates the implementation of two CCMs in ASYNC, CCM1 and CCM2, as well as the SSP model. CCM1 is the throttled-release [69] barrier strategy that submits tasks to available workers only when the number of available workers is at least k. CCM2 implements a fully asynchronous barrier that allows workers to progress as soon as their current task finishes.

```
f: STAT.foreach(Avaialble_Workers >= k) % CCM1
f: STAT.foreach(true) % CCM2
f: STAT.foreach(MAX_Staleness < s) % The SSP barrier control with a staleness threshold 's'
points.ASYNCbarrier(f, AC.STAT) % Apply the barrier
```

Listing 5.1: Pseudo-code for implementing CCMs in ASYNC.

5.6 Results

We evaluate the performance of ASYNC by implementing two asynchronous optimization methods, namely ASGD and ASAGA, and their synchronous variants to solve least-squares problems. We implement the throttled-release CCM for both asynchronous methods and use history in ASAGA and SAGA. The performance of ASGD and ASAGA are compared to their synchronous implementations in Spark. To the best of our knowledge, no library or implementation of asynchronous optimization

Table 5.2: Datasets for the experimental study.

Dataset	Row numbers	Column numbers	Size
rcv1_full.binary	697,641	47,236	851.2MB
mnist8m	8,100,000	784	19GB
epsilon	400,000	2000	12.16GB

methods exists on Spark. However, Glint which is built on top of Spark supports asynchrony for computing topic models. Therefore we implement the distributed optimization method ASGD using Glint and compared it with the implementation of ASGD in ASYNC. Glint requires more communication among its parameter servers and workers compared to ASYNC since it requires multiple push-pull operations for batches of data. This extra communication overhead results in poor performance for ASGD in Glint. Our experiments show that Glint converges approximately 10× slower compared to ASYNC. Furthermore, to demonstrate that the synchronous implementations of the algorithms using ASYNC are well-optimized, we first compare the performance of the synchronous variants of the tested optimization methods in ASYNC with the state-of-the-art machine learning library, MLlib [135]. MLlib is a library that provides implementations of a number of synchronous optimization methods. Afterward, we evaluate the performance of ASGD and ASAGA in ASYNC in the presence of stragglers.

5.6.1 Experimental Setup

We consider the distributed least-squares problem defined in (5.4). Our experiments use the datasets listed in Table 6.2 from the LIBSVM library [132], all of which vary in size and sparsity. For the experiments, we use ASYNC, Scala 2.11, MLlib [135], and Spark 2.3.2. Breeze 0.13.2 and netlib 1.1.2 are used for the (sparse/dense) BLAS operations in ASYNC. XSEDE Comet CPUs [133] are used to assemble the cluster. ASYNC is available at https://github.com/ASYNCframework/ASYNCframework.

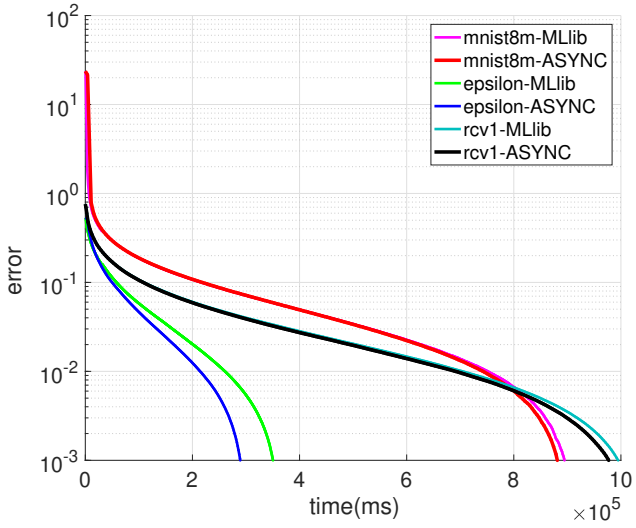


Figure 5.2: SGD implemented in ASYNC versus MLlib.

To demonstrate the performance of asynchronous algorithms and their robustness to the heterogeneity in cloud environments, we evaluate the implemented methods in the presence of stragglers. Two different straggler behaviors are used: *(i) Controlled Delay Straggler (CDS)* experiments in which a single worker is delayed with different intensities; *(ii)* the *Production Cluster Stragglers (PCS)* experiments in which straggler patterns from real production clusters are used. The CDS experiments are run with all three datasets on a cluster composed of a server and 8 workers. The PCS experiments require a larger cluster and thus are conducted on a cluster of 32 workers with one server using the two larger datasets (*mnist8m* and *epsilon*). In all configurations, a worker runs an executor with 2 cores. The number of data partitions is 32 for all datasets and in the implemented algorithms. The experiments are repeated three times; the average is reported.

Parameter tuning: A sampling rate of $b = 10\%$ is selected for the mini-batching SGD for *mnist8m* and *epsilon* and $b = 5\%$ is used for *rcv1_full.binary*. SAGA and ASAGA use $b = 10\%$ for *epsilon*, $b = 2\%$ for *rcv1_full.binary*, and use $b = 1\%$ for *mnist8m*. For the PCS experiment, we use $b = 1\%$ for *mnist8m* and *epsilon*. We use the same step size as MLlib and tune it for SGD to converge faster. A fixed step size is used in SAGA which is also tuned for faster convergence. The step size is not tuned for the asynchronous algorithms. Instead, we use the following heuristic, the step size of ASGD and ASAGA is computed by dividing the initial step size of their synchronous variants by the number of workers [62]. We run the SGD algorithm in MLlib for 15000 iterations with a sampling rate of 10% and use its final objective value as the *baseline* for the least-squares problem.

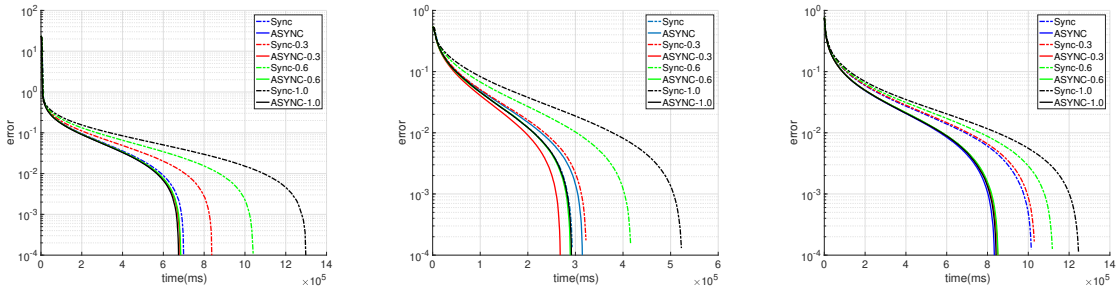
5.6.2 Comparison with MLlib

We use ASYNC for implementations of both synchronous and asynchronous variants of the algorithms because *(i)* ASYNC's performance for synchronous methods is similar to that of MLlib's; *(ii)* asynchronous methods are not supported in MLlib; *(iii)* synchronous methods that require the history of gradients can not be implemented in MLlib because of discussed overheads. To demonstrate that our implementations in ASYNC are optimized, we compare the performance of SGD in ASYNC and MLlib for solving the least-squares problem [199]. Both implementations use the same initial step size. The *error* is defined as the *objective function value* minus the *baseline*. Figure 5.2 shows the error for three different datasets. The figure demonstrates that SGD in ASYNC has a similar performance to that of Mllib's on 8 workers, the same pattern is observed on 32 workers. Therefore, for the rest of the experiments, we compare the asynchronous and synchronous implementations in ASYNC.

5.6.3 Robustness to Stragglers

Controlled Delay Straggler: We demonstrate the effect of different delay intensities in a single worker on SGD, ASGD, SAGA, and ASAGA by simulating a straggler with controlled delay [195, 204]. From the 8 workers in the cluster, a delay between 0% to 100% of the time of an iteration is added to one of the workers. The delay intensity, which we show with *delay-value* %, is the percentage by which a worker is delayed, e.g., a 100% delay means the worker is executing jobs at half speed. The controlled delay is implemented with the sleep command. The first 100 iterations of both the synchronous and asynchronous algorithms are used to measure the average iteration execution time.

The performance of SGD and ASGD for different delay intensities are shown in Figure 5.3 where



(a) mnist8m (b) epsilon (c) rcv1_full.binary

Figure 5.3: The performance of ASGD and SGD in ASYNC with 8 workers for delay intensities of 0%, 30%, 60% and 100% which are shown with ASYNC/SYNC, ASYNC/SYNC-0.3, ASYNC/SYNC-0.6 and ASYNC/SYNC-1.0 respectively.

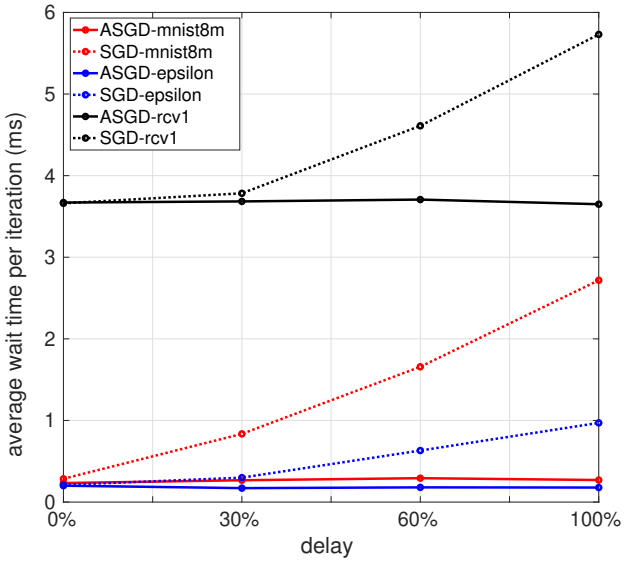


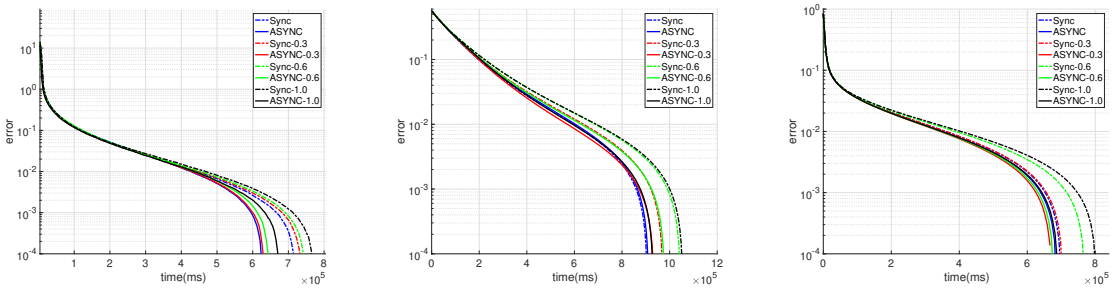
Figure 5.4: Average wait time per iteration with 8 workers for ASGD and SGD in ASYNC for different delay intensities.

for the same delay intensity the asynchronous implementation always converges faster to the optimal solution compared to the synchronous variant of the algorithm. As the delay intensity increases, the straggler has a more negative effect on the runtime of SGD. However, ASGD converges to the optimal point with almost the same rate for different delay intensities. This is because the ASYNCscheduler continues to assign tasks to workers without having to wait for the straggler. When the task result from the straggling worker is ready, it independently updates the model parameter. Thus, while ASGD in ASYNC requires more iterations to converge, its overall runtime is considerably faster than the synchronous method. With a delay intensity of 100%, a speedup of up to 2× is achieved with ASGD vs. SGD.

Figure 5.4 shows the average wait time for each worker for all iterations for SGD and ASGD. The wait time is defined as the time from when a worker submits its task result to the server until it receives a new task. In the asynchronous algorithm, workers proceed without waiting for

stragglers. Thus the average wait time does not change with changes in delay intensity. However, in the synchronous implementation worker wait times increase with a slower straggler. For example, for the *mnist8m* dataset in Figure 5.4, the average wait time for SGD increases significantly when the straggler is two times slower (delay = 100%). Comparing Figure 5.3 with Figure 5.4 shows that the overall runtime of ASGD and SGD is directly related to their average wait time where an increase in the wait time negatively affects the algorithm's convergence rate.

The slow worker pattern used for the ASGD experiments is also used for ASAGA. Figure 5.5 shows experiment results for SAGA and ASAGA. The communication pattern in ASAGA is different from ASGD because of the broadcast required to compute historical gradients. In ASAGA, the straggler and its delay intensity only affect the computation time of a worker and do not change the communication cost. Therefore, the delay intensity does not have a linear effect on the overall runtime. However, Figure 5.5 shows that increasing the delay intensity negatively affects the convergence rate of SAGA while ASAGA maintains the same convergence rate for different delay intensities.



(a) mnist8m (b) epsilon (c) rcv1_full.binary

Figure 5.5: The performance of ASAGA and SAGA in ASYNC for delay intensities of 0%, 30%, 60% and 100% which are shown with ASYNC/SYNC, ASYNC/SYNC-0.3, ASYNC/SYNC-0.6 and ASYNC/SYNC-1.0 respectively.

The workers' average wait time for ASAGA is shown in Figure 5.6. With an increase in delay intensity, workers in SAGA wait for more for new tasks. The difference between the average wait time of SAGA and ASAGA is more noticeable when the delay increases to 100%. In this case, the computation time is significant enough to affect the performance of the synchronous algorithm, however, ASAGA has the same wait time for all delay intensities.

Production Cluster Stragglers: Our PCS experiments are conducted on 32 workers with straggler patterns in real production clusters [205, 206]; these clusters are used frequently by machine learning practitioners. We use the straggler behaviors reported in previous research [207, 208] all of which are based on empirical analysis of production clusters from Microsoft Bing [206] and Google [205]. Empirical analysis from production clusters concluded that approximately 25% of machines in cloud clusters are stragglers. Of those, 80% have a uniform probability of being delayed between 150% to 250% of the average-task-completion time. The remaining 20% of the stragglers have abnormal delays and are known as long-tail workers. Long-tail workers have a random delay between 250% to 10×. Of the 32 workers in our experiment, 6 are assigned a random delay between 150%-250% and two are long-tail workers with a random delay over 250% up to 10×. The randomized delay seed is fixed across three executions of the same experiment.

The performance of SGD and ASGD on 32 workers with PCS is shown in Figure 5.7a. As shown,

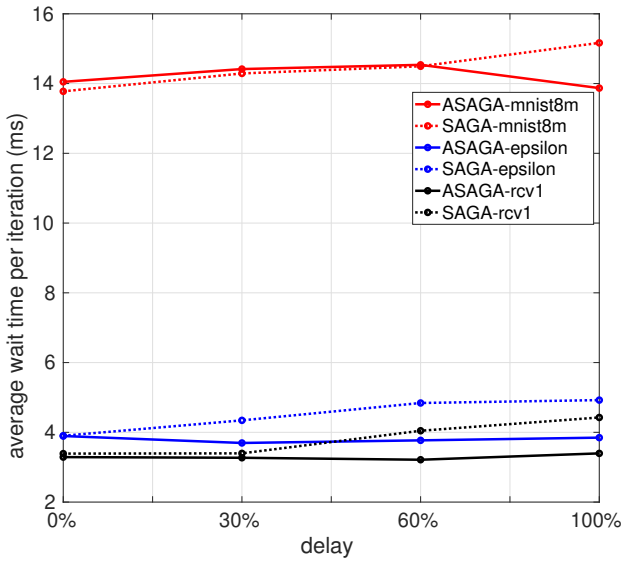


Figure 5.6: Average wait time per iteration with 8 workers for ASAGA and SAGA for different delay intensities.

ASGD converges to the solution considerably faster than SGD and leads to a speedup of 3× for *mnist8m* and 4× for *epsilon*. From Figure 5.7b, ASAGA compared to SAGA obtains a speedup of 3.5× and 4× for *mnist8m* and *epsilon* respectively. The average wait time for both algorithms on 32 workers is shown in Table 5.3. The wait time increases considerably for all synchronous implementations which results in slower convergence of the synchronous methods.

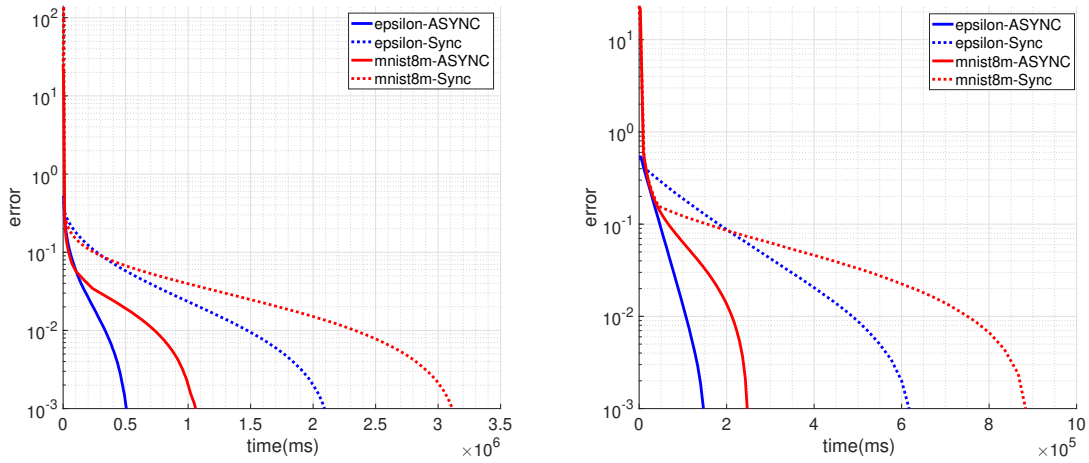
Table 5.3: Average wait time per iteration on 32 workers.

	SAGA	ASAGA	SGD	ASGD
mnist8m	42.8367 ms	9.8125 ms	6.4433 ms	3.5745 ms
epsilon	6.9926 ms	1.1721 ms	5.3112 ms	1.4165 ms

5.7 Conclusion

This work introduces the ASYNC framework that facilitates the implementation of asynchrony and history in machine learning methods on cloud and distributed platforms. Along with bookkeeping structures, the modules in ASYNC facilitate the implementation of numerous consistency models and history. ASYNC is built on top of Spark to benefit from Spark's in-memory computation model and fault-tolerant execution. We present the programming model and interface that comes with ASYNC and implement the synchronous and asynchronous variants of two well-known optimization methods as examples. These examples only scratch the surface of the types of algorithms that can be implemented in ASYNC. We hope that ASYNC helps machine learning practitioners with the implementation and investigation of the promise of asynchronous optimization methods.

Limitations. Storing the history in ASYNC can become a bottleneck for optimization methods with many iterations. Even though we did not observe this case in our experiments, this can slow



(a) The performance of ASGD and SGD in ASYNC shown with ASYNC and SYNC respectively.

(b) ASAGA and SAGA in ASYNC shown with ASYNC and SYNC respectively.

Figure 5.7: Comparing the performance of asynchronous methods vs their synchronous variants.

down the optimization process. Pruning strategies can reduce the size of history and hence improve the overall performance. Moreover, the communication cost of broadcasting the model can become comparable to the cost of communication of synchronous methods based on the choice of the barrier. An extreme example is the broadcasting of the model after every single update.

Chapter 6

Structured Sparse Approximation for Training Neural Networks

In this chapter, we develop a communication-efficient second-order optimization method for training deep neural networks. Our proposed method belongs to the class of Natural Gradient Descent (NGD) methods, which require computing the inverse of the Fisher information matrix (FIM). Computing the inverse of the neural network's Fisher information matrix is expensive because the Fisher matrix is large, i.e., has a size of $n \times n$, where n is the number of parameters. We propose HyLo, a *hybrid low-rank NGD* method, that applies a structured sparsification to the Fisher information matrix which can reduce the cost of communications and computations. The proposed approach uses a gradient-based switching heuristic to decide when to switch between two low-rank approximations; Khatri-Rao-based Interpolative Decomposition (KID) and Khatri-Rao-based Importance Sampling (KIS). The contributions of this chapter are:

- A novel Khatri-Rao-based interpolative decomposition, as well as an importance sampling approach that leverages the low-rank structure of FIM for fast factorization and inversion.
- A gradient-based switching strategy that determines when to switch between Khatri-Rao-based interpolative decomposition and importance sampling to maintain a good balance between accuracy and running time in NGD methods.
- An efficient distributed implementation of HyLo which is 1.7× and 1.4× faster than SGD and KAISA [209], the state-of-the-art distributed implementation of the second-order method for deep neural networks, on ResNet-50 model and 64 GPUs. We reduce the computation and communication cost of NGD by 350× and 10.7× compared to KAISA for the ImageNet-1k dataset on 64 GPUs.

6.1 Introduction

Natural gradient descent (NGD) [95], belongs to the class of second-order methods and accelerates the training of deep neural networks (DNN) by capturing the geometry of the optimization landscape with the Fisher Information Matrix (FIM) [210]. These methods demonstrate improved convergence

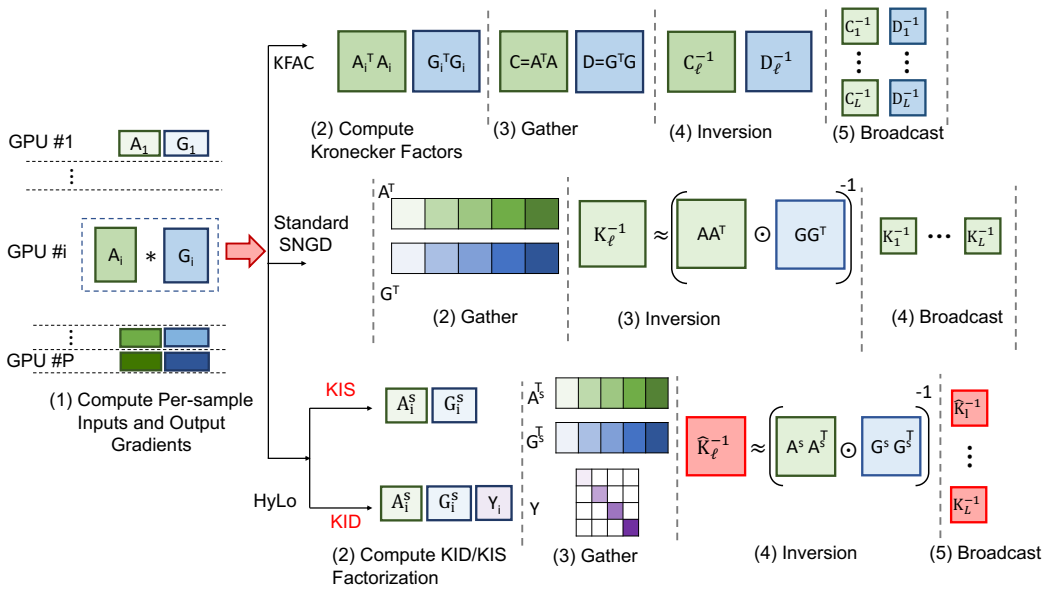


Figure 6.1: The HyLo method vs KFAC and standard SNGD approaches. HyLo reduces the computation and communication time of SNGD methods. It first factorizes the per-sample inputs and output gradients using Khatri-Rao-based interpolative decomposition and importance sampling. Then the reduced-size factors are gathered on workers to efficiently approximate the kernel matrix.

compared to first-order techniques such as stochastic gradient descent (SGD) as they compute the inverse of FIM and use it to precondition the gradients before parameter updates [95]. However, because the Fisher matrix is large and scales with the model size, finding its inverse is a major bottleneck in NGD methods.

Recent work [33] shows that Kronecker-Factored Approximate Curvature (KFAC) can reduce the computation cost of FIM by approximating the FIM for a batch of samples with a block-diagonal matrix, where blocks correspond to layers. Other variations of KFAC have been proposed, e,g, EKFAC [211], which improves the accuracy of approximation further by rescaling the Kronecker factors, or KBFGS [212], a Kronecker-based Quasi-Newton method that uses Broyden–Fletcher–Goldfarb–Shanno (BFGS) [183] updates to accelerate KFAC. Amongst these works, only the original KFAC method [33] has been implemented on large-scale distributed platforms [89, 209, 213–215] to our knowledge. Osawa *et al.* [7] proposed a memory-optimized implementation of KFAC by distributing the layers' factor computations across workers. Ueno *et al.* [213] improve [7] by using a custom 21-bit floating-point format for communication amongst workers and overlapping the communication with the computations involved in the backward pass. [214] proposes a communication-optimized implementation of KFAC by reducing the frequency of communications among workers and increasing the granularity of KFAC's computations. KAISA [209] improves on [7] and [214] by proposing a hybrid approach to choose between a communication-optimized and a memory-optimized implementation of KFAC. However, because their approach is based on KFAC, their scalability is limited by the computation and communication costs associated with computing the inverse and eigendecompositions of the Kronecker factors.

Sherman-Morrison-Woodbury-based NGD (SNGD) methods, recently introduced in [53, 216], leverage the structure of FIM for overparametrized models to use a matrix inversion technique called

Sherman-Morrison-Woodbury (SMW) identity. This reduces the computation cost of inversion in NGD methods for small batch sizes. As a result, SNGD methods perform better than KFAC approaches when the batch size is relatively smaller than the layer dimension. However, SNGD methods are not suitable for distributed settings because the inversion becomes a performance bottleneck as the overall batch size grows linearly with the number of workers. Also, SNGD approaches only support fully-connected layers and have not been extended to Convolutional Neural Networks (CNNs).

6.2 Motivation

In this section, we motivate our approach by analyzing the scalability of second-order optimization methods. We first provide background on deep neural networks (DNNs) as well as first- and second-order optimization methods and then compare the computation and communication complexity of KFAC and SNGD methods at scale. Finally, we show that HyLo outperforms NGD methods in distributed settings for models with large layers and large global batch sizes. Here, global batch size refers to the total number of data points (cumulative over workers) used per iteration and will be discussed further in Section 6.2.2.

6.2.1 Background

A deep neural network consists of L layers; each layer has parameters, i.e., weights, that are learned during the training. For simplicity, we consider a fully-connected layer with input and output dimensions d. The optimal parameters of a layer are obtained by minimizing the average loss $\mathcal{L}$ over the training dataset:

$$\mathcal{L}(w) = \sum_{i=1}^{N} \ell(w, x_i) \tag{6.1}$$

where $\{x_i\}_{i=1}^{N}$ is a dataset with N data points and w is the parameter to be learned. Typically, a batch of m samples is used to compute the loss.

Natural gradient descent (NGD). Natural gradient descent belongs to the class of second-order methods. It uses the Fisher Information Matrix (FIM) as a preconditioner for the gradient

$$w_{t+1} = w_t - \lambda(F_t + \alpha I)^{-1} g_t \tag{6.2}$$

where F_t is the *Fisher Information Matrix* (FIM) at the iterate w_t and α is the damping factor which is used to stabilize the training [95]. The FIM is an approximation of the Hessian and contains information about the curvature. Here, FIM is defined as [1][2]:

$$F(\cdot) = \frac{1}{m} \sum_{i}^{m} \nabla\mathcal{L}(\cdot, x_i) \nabla\mathcal{L}(\cdot, x_i)^\top = \frac{1}{m} U^\top U \tag{6.3}$$

where $U = [\nabla\mathcal{L}(w, x_1), \ldots, \nabla\mathcal{L}(w, x_m)]^\top$ and $F_t = F(w_t)$. We refer to the matrix $U \in \mathbb{R}^{m \times d^2}$ as the *Jacobian matrix*. It contains the gradients for each sample in the batch, i.e., $U = [U_1, U_2, ..., U_m]^\top$ where U_i is the gradient of the sample i. The Jacobian matrix has a row-wise Khatri-Rao structure,

[1] Note that FIM is sometimes defined differently as well based on the assumptions made on the model distribution [217]. Such assumptions do not affect our findings.

[2] For simplifying the notations, we drop the index t.

i.e.

$$U = A * G \tag{6.4}$$

where $A \in \mathbb{R}^{m \times d}$ and $G \in \mathbb{R}^{m \times d}$ are the per-sample layer's inputs and output gradients and $*$ is the row-wise Khatri-Rao product.

Kronecker Factorization Methods (KFAC). KFAC approximates the FIM with a block-diagonal matrix, each block corresponds to a layer. Then the inverse of a block is approximated using the Kronecker product of two matrices $C_1, C_2 \in \mathbb{R}^{d \times d}$ called the Kronecker factors, γ is the factor damping parameter

$$(F + \alpha I)^{-1} \approx \underbrace{(A^\top A + \gamma I)^{-1}}_{C_1} \otimes \underbrace{(G^\top G + \gamma I)^{-1}}_{C_2} \tag{6.5}$$

SMW-based Natural Gradient Descent Methods (SNGD). SMW-based NGD methods, which we call *SNGD*, leverage the structure of FIM in Equation 6.3 for overparametrized models as well as the Khatri-Rao structure of Jacobian in Equation 6.4 while using the SMW identity to obtain the inverse update:

$$(F + \alpha I)^{-1} = \frac{1}{\alpha}\big(I - U^\top \underbrace{\big(AA^\top \odot GG^\top + \alpha I\big)}_{K}^{-1} U\big) \tag{6.6}$$

where $\odot$ is the element-wise matrix product. We refer to $K = AA^\top \odot GG^\top + \alpha I$ as the *kernel matrix* throughout this chapter. The kernel matrix is symmetric positive semi-definite and has a dimension of the global batch size.

6.2.2 Complexity Analysis of Distributed NGD Methods

In this section, we discuss the efficient distributed implementations of KFAC (adopted from KAISA [209]) and standard SNGD methods and provide an analysis of the communication and computation costs when executed on a multi-GPU cluster; we refer to each GPU as a worker from here on. Figure 6.1 illustrates the distributed implementation of KFAC [209] and standard SNGD methods.

Distributed KFAC. Figure 6.1 shows the distributed implementation of KFAC on P workers, which involves five stages. On each worker *(1)* per-sample inputs A_i and output gradients G_i are computed using forward and backward passes on the network; *(2)* the Kronecker factors are computed according to Equation 6.5; *(3)* the Kronecker factors are communicated to other workers and their average is computed; *(4)* the Kronecker factors are inverted for worker's assigned layers; *(5)* the inverted factors are broadcast to other workers.

Distributed SNGD. To our knowledge SNGD methods have not been implemented on distributed platforms, so we show a communication-optimized implementation for SNGD based on the strategies proposed by [214] for second-order methods in Figure 6.1, and list the steps involved in the following. On each worker i, *(1)* the per-sample inputs A_i and output gradients G_i are computed during forward/backward passes on the network for a batch size of m; we refer to m as the *local batch size*, these matrices have a size of $m \times d$; *(2)* the matrices A_i and G_i are then gathered on the worker to create matrices A and G which have size $Pm \times d$. We refer to Pm as the *global batch size* since it shows the total number of samples used in each iteration of the training; *(3)* the kernel matrix of layer ℓ, K_ℓ is computed and inverted using SNGD inversion in Equation 6.6. *(4)* The inverted

Kernel matrices K_ℓ^{-1} are broadcast to other workers.

The computation cost in distributed KFAC includes computing and inverting the Kronecker factors in steps 2 and 4 with a total cost of $\mathcal{O}(d^3 + md^2)$. KFAC communicates the Kronecker factors and their inverses in steps 3 and 5 leading to an overall communication cost of $\mathcal{O}(d^2)$. SNGD computes the kernel matrix inversion in step 3 which has a computation cost of $\mathcal{O}(P^3m^3 + P^2m^2d)$ and communicates the per-sample inputs/output gradients and inverses in steps 2 and 4 with a total cost of $\mathcal{O}(P^2m^2)$. KFAC's computation and communication cost is proportional to the layer dimension and has a cubic time complexity while SNGD's cost grows with a cubic rate with respect to the global batch size.

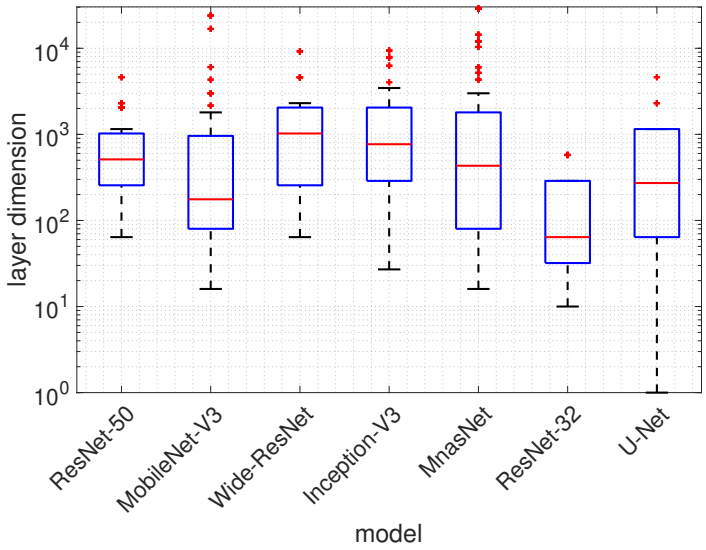


Figure 6.2: Distribution of layer dimensions for different DNN models. The layer dimension is large across all models.

6.2.3 Distributed HyLo vs. KFAC and SNGD

From the complexity analysis in Section 6.2.2, we observe that KFAC becomes inefficient for large layers and SNGD fails to scale for large global batch sizes. In many deep learning models, the layer dimension is large and hence KFAC becomes inefficient [218]. Figure 6.2 shows the distribution of layer dimension across the most popular deep learning models which shows the layer dimension is large for many layers in a model.

Figure 6.3 compares HyLo to standard SNGD and KFAC on 8 to 64 GPUs. While SNGD's running time is significantly less than KFAC on 8 and 16 GPUs, it fails to scale on 64 GPUs since the global batch size increases and hence its overall cost. However, HyLo outperforms both KFAC and SNGD on all settings because its overall cost is not adversely affected by the global batch size or the layer dimension.

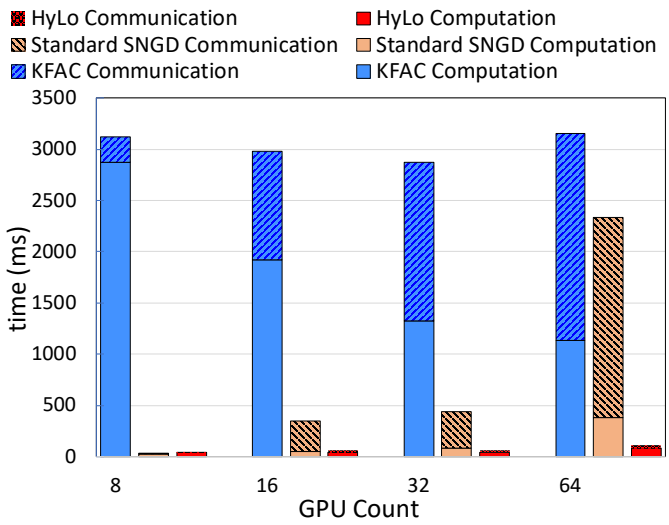


Figure 6.3: Computation and communication time of KFAC, HyLo and Standard SNGD on ResNet-50. Running time of KFAC and SNGD grows at scale. HyLo reduces the overall time by 28× and 20× compared to KFAC and SNGD.

6.3 HyLo: An Efficient Implementation of Hybrid Low-Rank Second-Order Method

We propose a hybrid low-rank second-order Method, called HyLo, that uses a Khatri-Rao-based interpolative decomposition and an importance sampling method to improve the performance of NGD methods on distributed platforms. HyLo belongs to the class of SNGD methods, however, it scales efficiently for large global batch sizes. The algorithm 6.1 shows the steps in HyLo. It first computes per worker, the per-sample inputs and output gradients (line 2). HyLo differs from prior approaches as it does not gather the per-sample inputs A_i and output gradients G_i. It instead uses a *Gradient-based Switching Heuristic* (lines 3-4) that decides when to switch between Khatri-Rao-based Interpolative Decomposition (KID) (lines 5 to 10) and Importance Sampling (lines 13 to 18) to reduce the size of A_i and G_i and hence the computation and communication overheads associated with them. In the following, we will first explain KID and analyze its computation/communication complexity and then discuss the importance sampling step. The gradient-based switching heuristic is explained at the end.

6.3.1 Khatri-Rao-based Interpolative Decomposition

HyLo per worker uses a Khatri-Rao-based interpolative decomposition to reduce the size of per-sample inputs/output gradients and hence the computation cost of the kernel matrix inversion and the communication cost of the gather and broadcast steps (steps 3 and 5 in Figure 6.1). In line 5, the per-sample inputs and output gradients matrices are approximated with smaller matrices, which we call *KID-factors*, using interpolative decomposition. Each worker then communicates the KID-factors to other workers. We propose a Khatri-Rao-based ID, shown in algorithm 6.2, that leverages the structure of Jacobian in Equation 6.4 to create the KID-factors. KID applies the factorization to the Gram matrix $A_i A_i^\top \odot G_i G_i^\top$, shown in line 1, which has a smaller size compared to individual

Algorithm 6.1 HyLo: A Hybrid Low-rank Natural Gradient Method

while *not converge* **do**
  Compute per-sample inputs A_i and output gradients G_i
  $R = |\|\Delta_{t+1}\| - \|\Delta_t\|| / \|\Delta_t\|$ /* Gradient-based heuristic */
  if $R \geq \eta$ *or learning rate decays* **then**
    /* Compute KID-factors with algorithm 6.2 */
    $A_i^s, G_i^s, Y_i = \text{KID}(A_i, G_i, r)$
      /* Gather KID-factors */
    $A^s = [A_1^s, ..., A_P^s], G^s = [G_1^s, ...G_P^s], Y = \text{diag}(Y_1, ..., Y_P)$
    for $\ell : 1, ..., L$ **do**
      if *layer is assigned to worker* **then**
        /* Inversion */
        $\hat{K}_\ell^{-1} = (A^s A^{s\top} \odot G^s G^{s\top})^{-1}$
          /* Broadcast */
        Broadcast$(\hat{K}_\ell^{-1})$
    update w using Equation 6.2 and Equation 6.7
  else
    /* Compute KIS-factors with algorithm 6.3 */
    $[A_i^s, G_i^s] = \text{KIS}(A_i, G_i, r)$
      /* Gather KIS-factors */
    $A^s = [A_1^s, ..., A_P^s], G^s = [G_1^s, ...G_P^s]$
    for $\ell : 1, ..., L$ **do**
      if *layer is assigned to worker* **then**
        /* Inversion */
        $\hat{K}_\ell^{-1} = (A^s A^{s\top} \odot G^s G^{s\top} + \alpha I)^{-1}$
          /* Broadcast */
        Broadcast$(\hat{K}_\ell^{-1})$
    update w using Equation 6.2 and Equation 6.8

Table 6.1: Comparison between the computation and communication complexities of distributed KFAC, standard SNGD, and HyLo on a system with P workers for a fully-connected layer with input/output dimension $= d$. m is the batch size per worker for KAISA and SNGD. r is the rank of the kernel matrix and $\rho = \frac{r}{P}$ is the number of samples per worker for HyLo.

	Computation		Communication		Storage
	Factorization	Inversion	Gather	Broadcast	
HyLo	$\mathcal{O}(m^2d + m^3)$	$\mathcal{O}(r^3 + r^2d)$	$\mathcal{O}(\rho d)$	$\mathcal{O}(r^2)$	$\mathcal{O}(rd + r^2)$
KFAC [209]	$\mathcal{O}(md^2)$	$\mathcal{O}(d^3)$	$\mathcal{O}(d^2)$	$\mathcal{O}(d^2)$	$\mathcal{O}(d^2)$
SNGD	-	$\mathcal{O}(P^3m^3 + P^2m^2d)$	$\mathcal{O}(md)$	$\mathcal{O}(P^2m^2)$	$\mathcal{O}(Pmd + P^2m^2)$

matrices A_i and G_i, and uses its decomposition to create the KID-factors A_i^s, G_i^s, and Y_i shown in line 4. In the second step, shown in line 6 of algorithm 6.1, A_i^s, G_i^s, and Y_i are gathered on the worker to create matrices $A^s = [A_1^s, ..., A_P^s]$, $G^s = [G_1^s, ..., G_P^s]$ and $Y = \text{diag}[Y_1, ..., Y_P]$, where diag(.) creates a block-diagonal matrix. Each worker computes and inverts the approximated kernel matrix for its assigned layers, shown in lines 8-9. We use A, G and Y to efficiently compute the kernel matrix inversion. To reduce the inversion cost, we apply the SMW formula [219] to the approximated kernel and obtain:

$$(F + \alpha I)^{-1} \approx \frac{1}{\alpha}\big(I - U^{s\top}\big(Y - Y(\hat{K}^{-1} + Y)^{-1}Y\big)U^s\big) \tag{6.7}$$

where $\hat{K} = A^s A^{s\top} \odot G^s G^{s\top}$ and $U^s = A^s * G^s$ are the reduced size kernel and Jacobian. Once $\hat{K}$ is inverted, it is broadcast to all workers, as shown in line 10.

Algorithm 6.2 Khatri-Rao-based Interpolative Decomposition

Input : Per-sample inputs and output gradients matrices $A_i \in \mathbb{R}^{m\times d}$, $G_i \in \mathbb{R}^{m\times d}$, rank r.
Output : $A_i^s \in \mathbb{R}^{r\times d}$, $G_i^s \in \mathbb{R}^{r\times d}$ and the projected approximation error matrix $Y_i \in \mathbb{R}^{r\times r}$.

/* Form Gram matrix */
1 $Q_i = A_i A_i^\top \odot G_i G_i^\top$
/* Compute row indices S and projection P: $Q_i \approx PQ_{i(S:)}$ */
2 $[P, S] = \text{ID}(Q_i, r)$
/* Compute the residue */
3 $R = Q_i - PQ_{i(S:)}$
/* Compute the KID-factors */
4 $A_i^s = A_{i(S:)}, G_i^s = G_{i(S:)}, Y_i = P^\top(R + \alpha I)^{-1}P$
return A_i^s, G_i^s, Y_i

6.3.2 Khatri-Rao-based Importance Sampling

An alternative to reduce the size of per-sample inputs A_i and output gradients G_i is to use randomized sampling. HyLo uses Khatri-Rao-based importance sampling [220], shown in line 13 in algorithm 6.1, to create the reduced-sized per-sample inputs and gradients, called *KIS-factors*. We apply norm-based sampling to reduce the size of A_i and G_i by choosing the samples that contribute the most to approximating the kernel matrix. The steps for importance sampling are shown in algorithm 6.3. It assigns scores using Euclidean norm to each sample of a batch and selects from those accordingly. The Khatri-Rao structure of Jacobian, (Equation 6.4), allows for an efficient evaluation of scores.

In particular, the score for sample j is obtained as the product of its input and gradient norm, i.e., $\|A_{i(j:)}\|\|G_{i(j:)}\|$ where $A_{i(j:)}$ is the j-th row of A_i and similarly for G_i. Once the KIS-factors A_i^s and G_i^s are created, they are gathered on the workers to form $A^s = [A_1^s, ...A_P^s]$ and $G^s = [G_1^s, ..., G_P^s]$.

Each worker inverts the kernel matrix K for its assigned layers, shown in lines 16-17. K is approximated with a smaller matrix using A^s and G^s. Hence the FIM is inverted by:

$$(F + \alpha I)^{-1} \approx \frac{1}{\alpha}\big(I - U^{s\top}\hat{K}^{-1}U^s\big) \tag{6.8}$$

where $\hat{K} = A^s A^{s\top} \odot G^s G^{s\top} + \alpha I$ and $U^s = A^s * G^s$ are the reduced size kernel and Jacobian. Once the kernel matrix for a layer is inverted, it is broadcast to all workers, as shown in line 18.

Algorithm 6.3 Khatri-Rao-based Importance Sampling

Input : Per-sample inputs and output gradients matrices $A_i \in \mathbb{R}^{m\times d}$, $G_i \in \mathbb{R}^{m\times d}$, number of samples r.

Output : $A_i^s \in \mathbb{R}^{r\times d}, G_i^s \in \mathbb{R}^{r\times d}$

/* Compute norm of inputs and gradients */

1 **for** $j : 1, ..., m$ **do**

2 $\quad P_j = \|A_{i(j:)}\|$, $Q_j = \|G_{i(j:)}\|$

/* Compute sample scores */

3 $\Omega = P \odot Q$

/* Choose r samples at random based on scores Ω */

4 $A_i^s, G_i^s = \text{sample}(A_i, G_i, \Omega, r)$

Table 6.1 summarizes the complexity analysis of HyLo and KFAC [209] implemented on a distributed platform with P workers for each step of the methods[3]. The computation cost of HyLo involves the cost of factorization and inversion step and is $\mathcal{O}\big(r^3 + m^3 + (m^2 + r^2)d\big)$. HyLo reduced the computation cost compared to KFAC and SNGD from $\mathcal{O}(d^3)$ and $\mathcal{O}(P^3m^3)$ to $\mathcal{O}(r^3 + m^3)$. HyLo also reduced the communication cost compared to KFAC and SNGD from $\mathcal{O}(d^2)$ and $\mathcal{O}(P^2m^2)$ to $\mathcal{O}(r^2)$.

6.3.3 Gradient-based Switching

HyLo chooses between Khatri-Rao-based interpolative decomposition and importance sampling at the beginning of an epoch. The accumulated gradient of the objective function w.r.t the parameters is used as a metric to determine which epochs benefit the most from KID (the method that has a lower approximation error) or KIS (the method with the lower computation complexity). From [221], epochs with a larger change in their accumulated gradient, i.e., *critical epochs*, contribute more to the training. Small gradient errors in critical epochs can impair the training progress. Hence, our switching method uses KID for critical epochs and importance sampling for others. As shown in algorithm 6.1 lines 3-4, an epoch is critical when the learning rate decays or when the accumulated gradient changes pass a threshold η:

$$\left|\frac{\|\Delta_{t+1}\| - \|\Delta_t\|}{\|\Delta_t\|}\right| \geq \eta \tag{6.9}$$

where $\|\Delta_t\|$ is the l2-norm of accumulated gradients.

[3]KIS has a computation complexity of $\mathcal{O}(m^2 + md)$ which is asymptotically smaller than KID and hence omitted.

6.4 HyLo For Convolutional Neural Networks

HyLo is an SNGD-based approach and since these approaches are not formulated to support convolutional neural networks (CNNs), in this section, we present the first extension of SNGD methods to CNNs to our knowledge. First, we briefly review convolutional layers and then propose an efficient formulation of SNGD for convolutional layers.

A convolutional layer operates on large dimensional tensors. In particular, the output tensor $\mathcal{Y}$ of a convolutional layer is obtained by applying convolution to its input $\mathcal{X}$ and weights. To simplify the notations, we assume $\mathcal{X}, \mathcal{Y} \in \mathbb{R}^{d \times s \times s}$, i.e., they contain d images of size $s \times s$. Similarly, we denote the corresponding output gradient with $\mathcal{G}$.

The SNGD method in Equation 6.6 does not apply to CNNs because convolutional layers typically operate on 4D tensors while the SMW identity is only for matrices (2D tensors). We extend the SNGD formulation to convolutional layers by first expressing the convolution operation as a matrix multiplication between reshaped input and output tensors. Formally, the gradient for one sample is computed by $X^\top \cdot G$ where $X = \texttt{im2col}(\mathcal{X})$ and $G = vec(\mathcal{G})$ in which `im2col` unfolds a three dimensional tensor to a matrix by reshaping its blocks into 1-D vectors and vec reshapes the tensor $\mathcal{G}$ by flattening its spatial dimension. Then the reshaped tensors are approximated over their spatial dimensions. In particular, $\hat{x} = \sum_i^S X_{(i:)}$ and $\hat{g} = \sum_i^S G_{(i:)}$ where $X_{(i:)}$ and $G_{(i:)}$ are the i-th row of X and G respectively. By concatenating the vectors $\hat{x}$ and $\hat{g}$ for a batch of samples, the approximated per-sample inputs and gradients matrices are obtained, i.e., $\hat{X} = [\hat{x}_1^\top, ..., \hat{x}_m^\top]^\top$ and $\hat{G} = [\hat{g}_1^\top, ..., \hat{g}_m^\top]^\top$. The approximated Jacobian $\hat{U}$ is computed with their row-wise Khatri-Rao product, similar to Equation 6.4. Finally, we leverage this structure and reformulate SNGD:

$$(F + \alpha I)^{-1} \approx \frac{1}{\alpha}(I - \hat{U}^\top \big(C_1 \odot C_2 + \alpha I\big)^{-1} \hat{U}) \tag{6.10}$$

where $C_1 = \hat{X}\hat{X}^\top$ and $C_2 = \hat{G}\hat{G}^\top$.

6.5 Results

We compare the performance of HyLo on a cluster of 64 GPUs to KAISA [209], the state-of-the-art distributed implementation of KFAC, and the efficient distributed implementation of stochastic gradient descent, used as a first-order optimization method.[4] To show that HyLo also improves the performance for small batch sizes, we compare it on a single-GPU to KFAC and to other two more-recent methods EKFAC [211] and KBFGS-L [222]. The methods EKFAC and KBFGS-L do not have distributed implementations.

6.5.1 Methodology

The datasets and deep learning models listed in Table 6.2 are used for the experimental results.

Datasets. We use datasets from image classification and segmentation applications. ImageNet-1k [223] has 1000 categories with approximately 1.3M training images and 50K validation images. CIFAR-10 and CIFAR-100 [224] consist of 50K training images and 10K validation images in 10

[4]The only work that attempts to implement SNGD is SENG [216], however, their approach at large-scale does not communicate second-order information and hence is not a (standard) NGD method.

Table 6.2: Model and dataset used for experimental results.

Model	Dataset	Target	GPU (#, type)	
ResNet-50	ImageNet-1k	75.9%	64	V100
U-Net	LGG Segmentation	91%	4	V100
ResNet-32	CIFAR-10	92.5%	32	K80
DenseNet	CIFAR-100	75%	1	V100
3C1F	Fashion-MNIST	93%	1	V100

classes. The LGG Segmentation dataset [225] contains Magnetic Resonance (MR) images of the brain. We use 3336 images for training and 332 images for validation. Fashion-MNIST [226] consists of a training set of 60K images and a test set of 10K examples that belong to 10 classes.

Models. We use state-of-the-art deep learning models listed in Table 6.2 for the experimental results. For Fashion-MNIST a network with three convolutional layers and one fully-connected layer is used, hence called 3C1F. For other benchmarks, the following DNN models are used: ResNet-50, U-Net, ResNet-32 and DenseNet.

Target accuracy. Following lists our baselines for target accuracy for different experiments: *(1)* For ResNet-50, we use the MLPerf benchmark target results [227]; *(2)* For U-Net, we use the target validation Dice similarity coefficient (DSC) [225]. *(3)* For ResNet-32, we use the target test accuracy reported in [228]. *(4)* For DenseNet we use the test accuracy in [229]. *(5)* For 3C1F we use the test accuracy of tuned SGD.

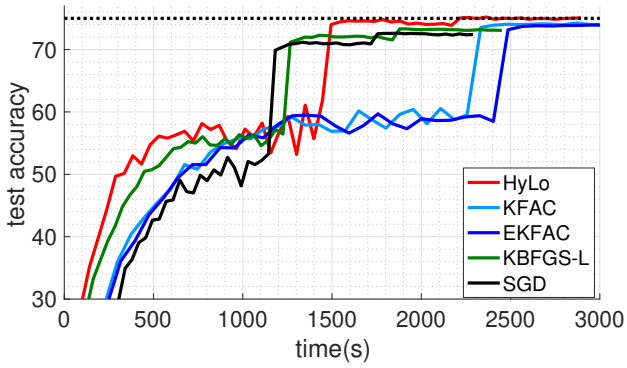
GPU Clusters. Results are obtained on two clusters: *(1)* The GPU system Mist [230] has 54 nodes, each with 32 IBM Power9 cores with 256GB RAM and 4 NVIDIA V100 GPUs with 32GB memory and NVLink in between. Nodes are connected via InfiniBand EDR. *(2)* The GPU cluster of Amazon Web Services (AWS) P3 and P2 instances, in which each node of P3.16xlarge is equipped with 8 NVIDIA V100 with 32 GB memory connected via NVLink and P2.8xlarge has 8 K80 GPUs with 12 GB memory.

Software. We use PyTorch 1.7.1, CUDA 10.2, CUDNN 7.6.5, and NCCL 2.7.8.

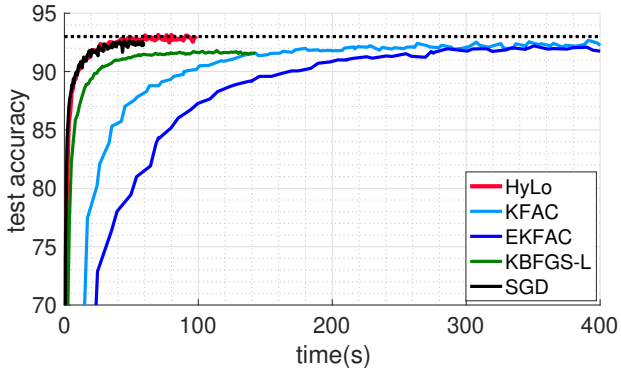
Training parameters. We train ResNet-50 for 50 epochs for HyLo and KAISA, and 90 epochs for SGD with a batch size of 80 per GPU, similar to [209]. We use a similar approach to [209] and train U-Net for 30 epochs for HyLo and KAISA and 50 epochs for ADAM with a batch size of 16 per GPU. ResNet-32 is trained for 100 epochs for HyLo and KAISA, and 200 for SGD with a batch size of 128 per GPU, similar to [209]. DenseNet and 3C1F models are trained for 60 epochs with a batch size of 128. We use momentum, and tune the learning rate and weight decay for all methods. For HyLo and KAISA damping is tuned. All the methods except SGD update the second-order information every few iterations. The frequency of the update for HyLo is similar to that of KAISA and is chosen according to the authors' default parameters for KFAC, EKFAC, and KBFGS-L. We choose the parameter r in KID and KIS as 10% of the global batch size in all experiments unless otherwise stated.

6.5.2 Single-GPU Setting

This section shows the performance of HyLo on a single-GPU for smaller models DenseNet and 3C1F. We compare HyLo to KFAC, EKFAC, KBFGS-L, and SGD. In order to demonstrate the performance of HyLo, we compare the single-GPU implementation of these methods for small batch sizes.



(a) DenseNet



(b) 3C1F

Figure 6.4: Test accuracy comparison between HyLo and KFAC, EKFAC, KBFGS-L and SGD for a) DenseNet b) 3C1F models on single-GPU. The dotted line shows the target metric.

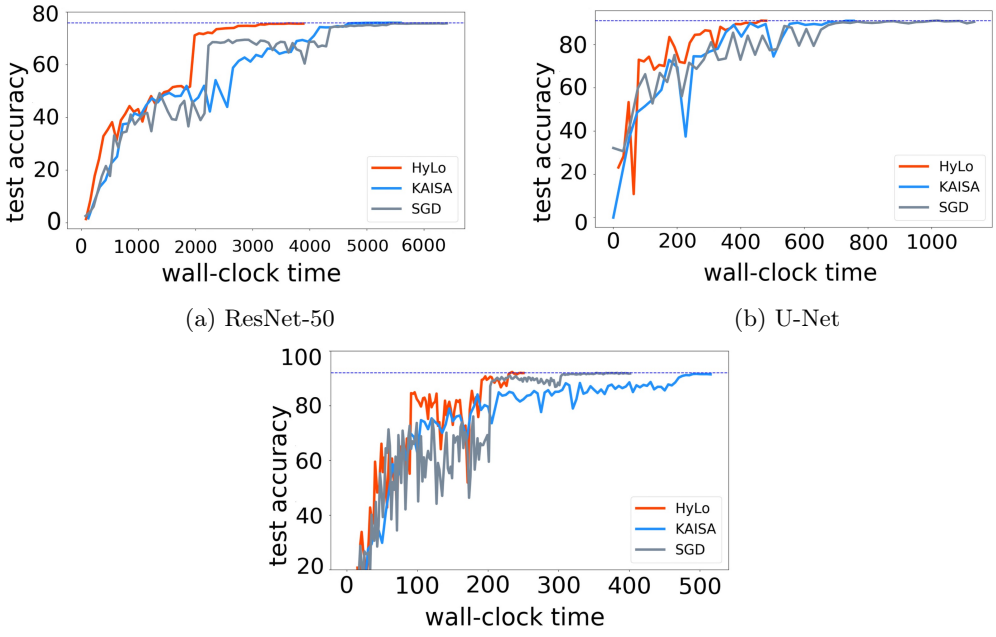
Figure 6.4a shows the test accuracy vs. time on DenseNet model. HyLo converges to the target test accuracy of 75% in 36.9 minutes while KFAC and EKFAC achieve an accuracy of 74.2% in 50 minutes. Both KBFGS-L and SGD, respectively, achieve a lower accuracy of 73.2% and 73% with a time-to-convergence of 40.8 and 38 minutes. HyLo outperforms all methods in accuracy and is 1.4× faster than KFAC.

To further demonstrate the generalization performance of HyLo, we compare its test accuracy to KFAC, EKFAC, KBFGS-L, and SGD for the 3C1F model tested on Fashion-MNIST in Figure 6.4b. HyLo achieves the target accuracy and outperforms KBFGS-L with a margin of 1.5% and KFAC and EKFAC with 0.95%. HyLo also accelerates the end-to-end training time up to 3× compared to KFAC and EKFAC.

6.5.3 Multi-GPU Setting

To demonstrate the performance of HyLo at scale, we conduct experiments on ResNet-50, U-Net, and ResNet-32 on up to 64 GPUs and compare HyLo with KAISA and SGD. First, we show the end-to-end accuracy and time-to-convergence and then analyze HyLo's performance with communication and computation cost breakdown.

Accuracy and time-to-convergence. Figure 6.5a compares the test accuracy of HyLo to



(a) ResNet-50

(b) U-Net

(c) ResNet-32

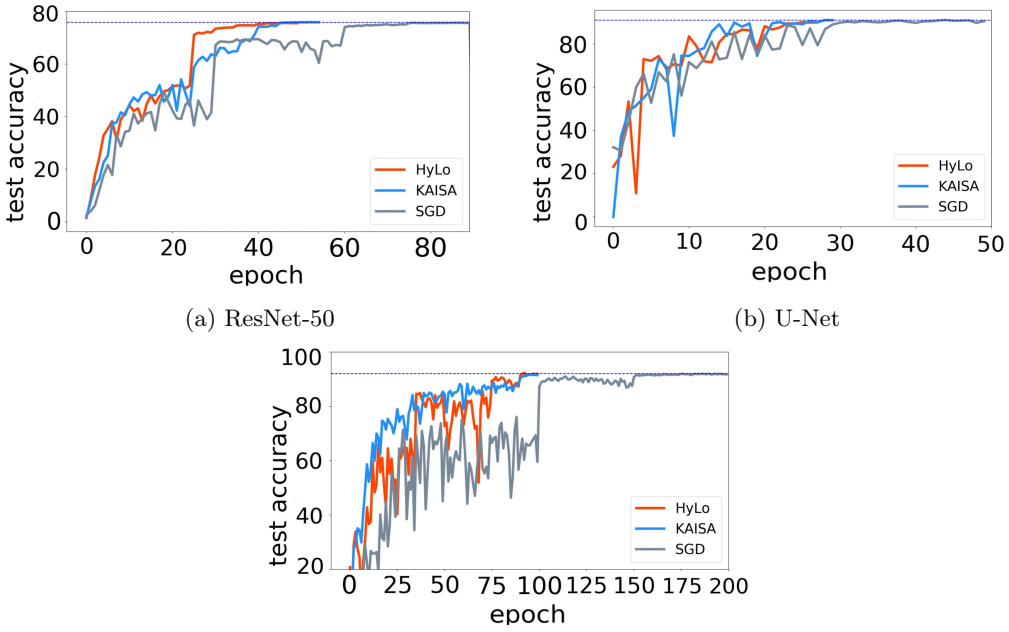
Figure 6.5: Comparison of test accuracy vs time between HyLo, KAISA and SGD on ResNet-50, U-Net and ResNet-32. The dotted line is the target accuracy.

KAISA and SGD for ResNet-50 model on 64 GPUs. HyLo converges to the target accuracy in 64.8 minutes which is 1.4× faster than KAISA with 93.2 minutes and 1.7× faster than SGD with 106.7 minutes. We also show the test accuracy curve w.r.t epochs in Figure 6.6a. HyLo improved the convergence and outperforms KAISA and SGD in per-epoch accuracy.

HyLo outperforms KAISA and ADAM with a large factor for U-Net and ResNet-32 models as shown in Figure 6.5b and Figure 6.5c. For U-Net, the time-to-convergence for HyLo is 480s which is 2.4× and 1.6× faster than ADAM and KAISA respectively. For ResNet-32, HyLo reduces the training time by factors 2.1× and 1.6× compared to KAISA and SGD. Figure 6.6c and Figure 6.6b compare the per-epoch accuracy of HyLo with KAISA and SGD on ResNet-32 and U-Net. For ResNet-32, HyLo shows a better convergence compared to SGD and outperforms KAISA in per-epoch accuracy.

Performance analysis. We compare the cost of different steps involved in HyLo and KAISA to show the advantage of HyLo when distributed. HyLo and KAISA on ResNet-50 are trained on ImageNet-1k on 64 GPUs and ResNet-32 for CIFAR-10 is on 32 GPUS. Even though HyLo is a hybrid approach, we intentionally report separate times for its KID and KIS methods to further examine their computation and communication costs. The computation time consists of the time for factorization and inversion steps and the communication time includes the gather and broadcast steps.

Computation cost. HyLo reduces the factorization time compared to KAISA by 27× for KID and 350× for KIS on ResNet-50, as shown in Figure 6.7a. The large speedup of KIS compared to KAISA is due to its low-cost norm-based sampling which is efficiently computed with algorithm 6.3. The KID iterations involve interpolative decomposition and hence have a higher execution time. HyLo also reduces the inversion time to 3ms which is ∼135× less than KAISA with 410ms. As



(a) ResNet-50

(b) U-Net

(c) ResNet-32

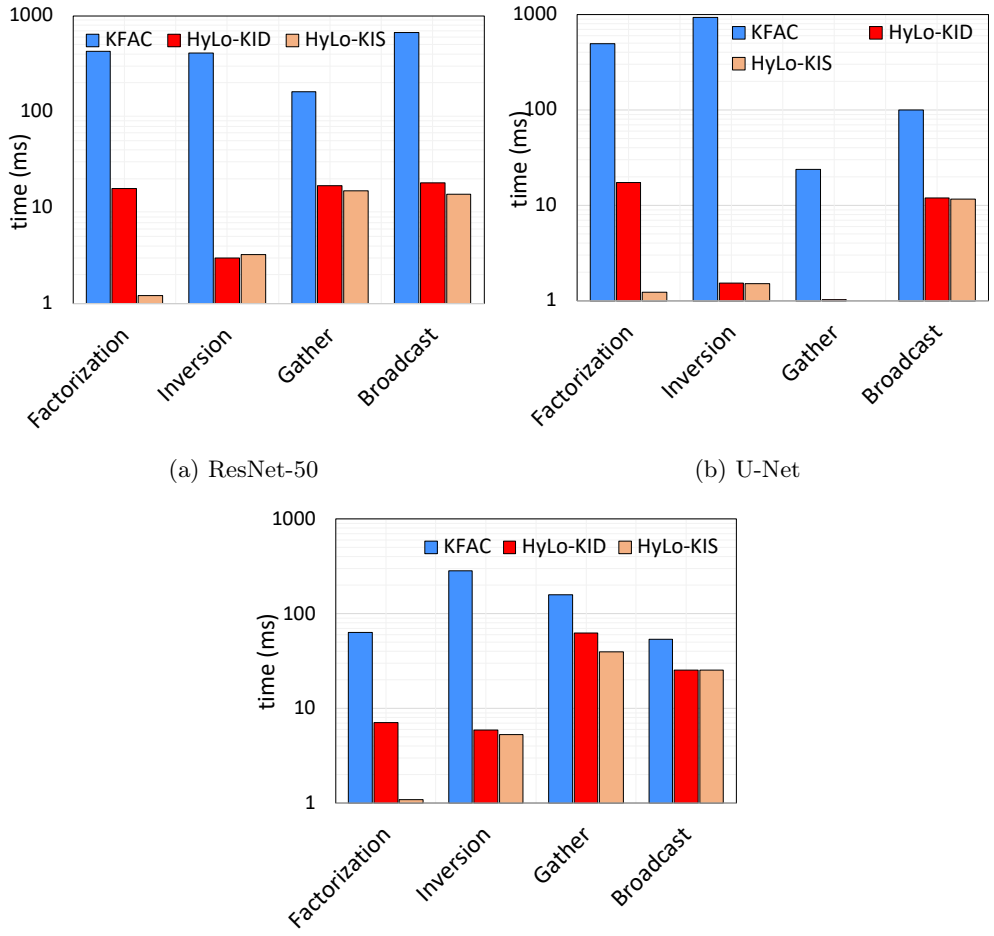
Figure 6.6: Comparison of test accuracy vs epoch between HyLo, KAISA, and SGD on ResNet-50, U-Net, and ResNet-32. The dotted line is the target accuracy.

shown in Figure 6.2, ResNet-50 has large layers and hence KAISA's inversion becomes inefficient. Figure 6.7b shows that the speedups gained from HyLo are more significant for U-Net model in which the inversion cost is reduced to 1.5ms which is 600× less than KAISA.

HyLo also reduces the computation time when the model has layers with smaller dimensions. As shown in Figure 6.7c, the time of the factorization and inversion steps for KID are 6ms and 7ms which are 9× and 47× faster than that of KAISA. The speedups gained on ResNet-32 are relatively smaller compared to ResNet-50. This correlates with the observation in Figure 6.2 (which shows the layer dimension distribution) that ResNet-50 has larger layers.

Communication cost. The time for communicating the factors in the gather step is reduced by a factor of 9.4× for KID and 10.7× for KIS compared to KAISA on ResNet-50, as shown in Figure 6.7a. The KID method transfers three matrices, i.e., KID-factors, per layer and hence has a slightly higher communication time in the gather step while KIS only communicates two matrices per layer. Finally, we observe that the dominant communication time is in the broadcast step, with 18ms for KID and 13ms for KIS which are 36× and 48× less than that of KAISA with 664ms. Figure 6.7b shows the communication time breakdown of HyLo and KAISA on U-Net. The time for the gather step in HyLo is ∼1ms which is ∼20× less than KAISA. HyLo reduces the communication cost of broadcast step 8× compared to KAISA. Figure 6.7c shows the communication time breakdown of HyLo and KAISA for ResNet-32. The gather time is reduced compared to KAISA by factors 2.5× and 4× for KID and KIS methods. Both KID and KIS have similar times in the broadcast step and improve over KAISA by a factor of 2.1×.

Scaling. To demonstrate the scalability of HyLo, Similar to [209], we report the projected end-to-end training time speedup for HyLo over SGD in Figure 6.8a. We measure the average time-

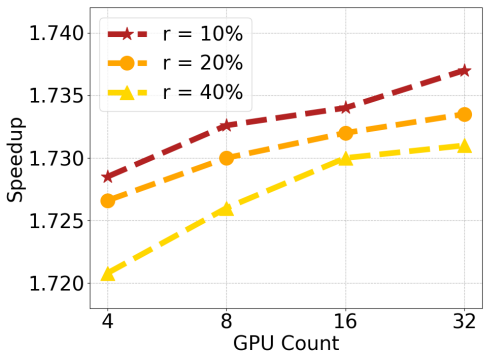


(a) ResNet-50

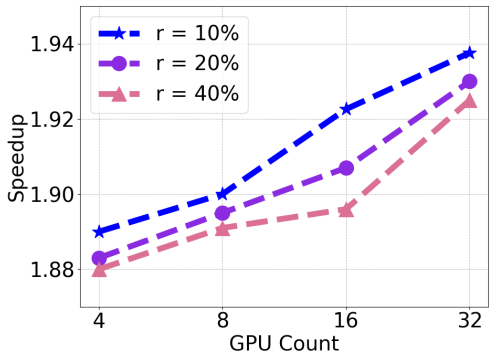
(b) U-Net

(c) ResNet-32

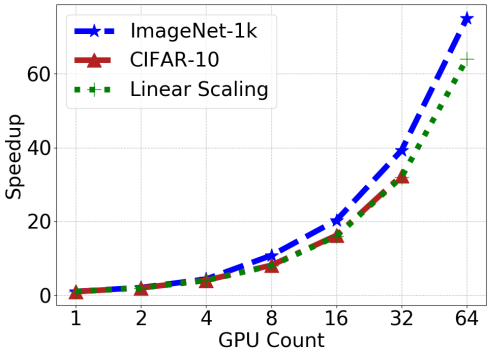
Figure 6.7: Computation and communication time breakdown for HyLo and KAISA on ResNet-50, U-Net, and ResNet-32 models. The computation time is measured for the factorization and inversion steps and communication time includes the gather and broadcast times.



(a) Speedup on ResNet-50.



(b) Speedup on ResNet-32.



(c) Scalability on ResNet-50 and ResNet-32.

Figure 6.8: The speedup of HyLo over SGD for a) ResNet-50 b) ResNet-32 model on different number of GPUs, r is the rank parameter, c) shows the scalability of HyLo on ResNet-50 and ResNet-32 models up to 64 GPUs.

per-epoch for HyLo and SGD on models that execute on a large number of GPUs, i.e., ResNet-50 and ResNet-32; the number of GPUs is varied from 8 to 64 for ResNet-50 and 4 to 32 for ResNet-32. Also, to show the effect of the Kernel matrix rank on the scaling, r is set (See algorithm 6.1) to 10%, 20%, and 40% of the global batch size. For ResNet-50, we project the training time to 90 epochs in SGD and 50 epochs in HyLo, and for ResNet-32 the training time is projected to 200 and 100 epochs for SGD and HyLo respectively. The frequency of HyLo updates is scaled inversely with the number of GPUs to keep the number of updates per training sample constant. The speedup of HyLo improves over SGD with an increasing number of GPUs. HyLo achieves $\sim$1.9$\times$ speedup For ResNet-32 and $\sim$1.7 $\times$ speedup for ResNet-50 model on 32 and 64 GPUs. For a more accurate scalability analysis, in Figure 6.8c we show the running time of HyLo vs its single-GPU time for varying numbers of GPUs on ResNet-50 and ResNet-32 models. Figure 6.8c shows that HyLo scales superlinearly for ResNet-50 and linearly for ResNet-32.

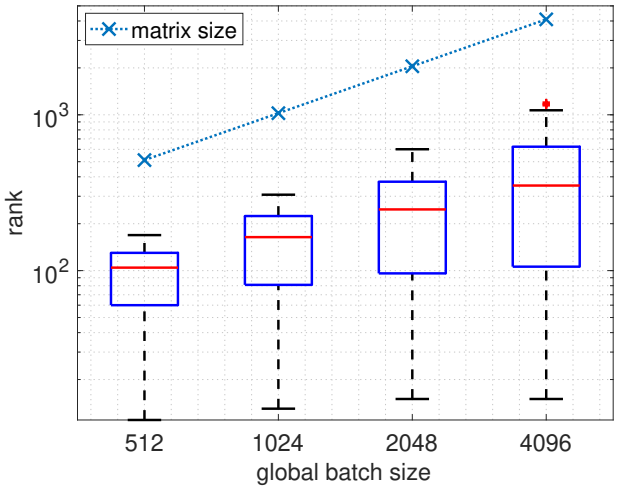
Analysis of rank and gradient norm. HyLo is built on the assumption that the Kernel matrix has a low-rank structure, this property allows us to replace the per-sample inputs and gradients matrices with KID/KIS-factors. The following shows that the kernel matrix has a low-rank structure when global batch sizes are large. We also provide an analysis of our switching-based method which decides how often to choose KID versus KIS.

Analysis of kernel matrix rank. For different batch sizes, we analyze the rank distribution of the kernel matrix on ResNet-50 and ResNet-32. We compute the eigenvalue decomposition of the kernel matrix for each layer and report its numerical rank, i.e., the number of eigenvalues that contribute to 90% of their sum. The global batch size ranges from 512 to 4096 for ResNet-50 and ResNet-32. The kernel matrix maintains a low-rank structure for all global batch sizes. Figure 6.9a shows the rank distribution on ResNet-50, the median rank is 104, 164, 247, and 351 which respectively, show a ratio of 20%, 16%, 12%, and 8.5 % of the global batch size. Figure 6.9b for ResNet-32 shows that on a large global batch size of 4096 we only require $\sim$2% of the samples to approximate the matrix as the median rank is small, 89, compared to the batch size.

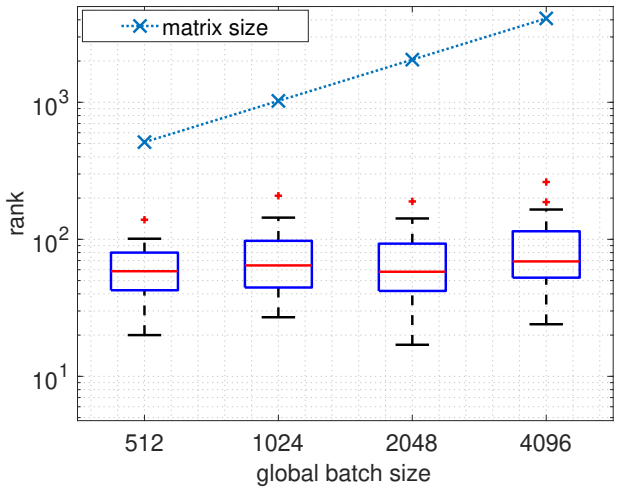
Analysis of switching method. We provide an analysis of how often HyLo chooses KID versus KIS. We compute the norm of the gradient at each epoch on the ResNet-32 model and report in Figure 6.10 for various layers. The gradient norm changes rapidly in the initial epochs and also after the learning rate decays at epochs 35 and 75. Hence, HyLo chooses KID over KIS in 20% of epochs, i.e., 1-10, 35-39, and 75-79. A similar analysis leads to choosing KID in 30% of the epochs on ResNet-50, in particular epochs 1-10 and 25-29.

6.6 Related Work

Second-order methods specifically Natural Gradient Descent (NGD) [95, 231–233] can accelerate training by improving the convergence rate of DNNs, specifically, exact NGD [101] methods have demonstrated improved convergence compared to first-order techniques such as SGD [85]. Zhang *et al.* [101] extend NGD to deep nonlinear networks with non-smooth activations and show that NGD converges to the global optimum with a linear rate. However, their method fails to scale to large or even moderate size models primarily because it relies heavily on backpropagating Jacobian matrices, which scale with the network's output dimension [97]. In [53] the authors use Woodbury identity for the inversion of the Fisher matrix and propose a unified framework for subsampled Gauss-Newton



(a) ResNet-50



(b) ResNet-32

Figure 6.9: Distribution of the rank of kernel matrix on a) ResNet-50 and b) ResNet-32. The kernel matrix has a low-rank structure for all global batch sizes.

and NGD methods. Their framework is targeted at fully-connected networks and relies on empirical Fisher. This requires extra forward-backward passes to perform parameter updates [97, 234].

Approximate NGD approaches such as [32,54,98,235–237] attempt to improve the overall execution time of NGD with FIM inverse approximation. For example, KFAC [54] approximates each block inverse using the Kronecker product of two smaller matrices, i.e., Kronecker factors. However, these factors have large sizes for wide layers and hence their inversion is expensive. EKFAC [98] improves the approximation used in KFAC by rescaling the Kronecker factors with a diagonal matrix obtained via costly singular value decompositions. Other work such as KBFGS [235] further estimates the inverse of Kronecker factors using low-rank BFGS type updates. WoodFisher [238] estimates the empirical FIM block inverses using rank-one updates, however, this estimation will not contain enough useful curvature information to produce a good search direction [33]. Goldfarb *et al.* [235]

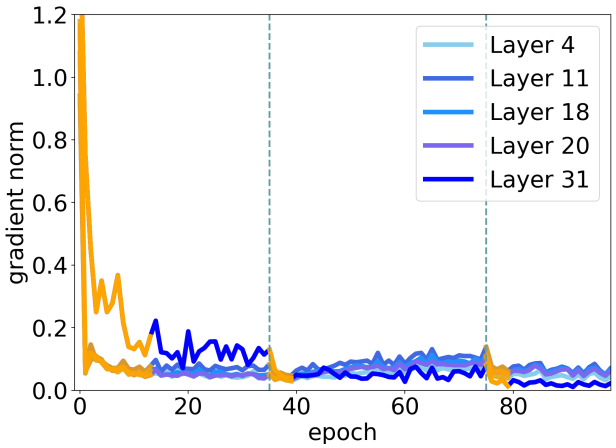


Figure 6.10: Gradient norms throughout ResNet-32 training.

follow the framework for stochastic quasi-Newton methods and prove that KBFGS converges with a sublinear rate for a network with bounded activation functions.

Distributed implementations of KFAC methods have been recently explored in [7, 89, 209, 213, 214, 239]. Ba *et al.* [89] implement an asynchronous distributed KFAC on a parameter server. Their work assigns the factorization step in KFAC to some of the workers while others compute inversions. Their approach leads to a 5.9% accuracy loss on ImageNet-1k compared to state-of-the-art target accuracies. Osawa *et al.* [7] perform the factorization step on all workers and compute the factor inverses using a model-parallel approach. They apply mixed-precision computations (FP32 and FP16) and use symmetry-aware communication to reduce the volume of transferred data by only sending the upper triangular part of Kronecker factors. Pauloski *et al.* [214] use a similar approach to [7] and implement KFAC layer-wise, their method achieves the MLPerf target accuracy and improves training time over SGD. This work was further improved in [213], by applying a custom 21-bit floating-point format for collective communications amongst GPUs and overlapping the gather/broadcast steps with the backward pass computations. KAISA [209] combines previous distributed KFAC strategies into one hybrid approach using a tunable memory footprint approach to balance the memory and communication costs. Ma *et al.* [239] provides an extensive analysis of distributed KFAC and shows that KFAC does not exhibit good scalability behavior and loses accuracy on large batch sizes.

6.7 Conclusion

In this work, we propose a distributed algorithm and implementation for Natural Gradient Decent methods which we call HyLo. HyLo is based on a gradient-based switching approach that decides when to use a Khatri-Rao-based interpolative decomposition method and when to use importance sampling. As a result, the computation and communication costs associated with computing the Fisher matrix inverse in NGD methods are significantly reduced on distributed platforms. HyLo also supports convolutional neural networks as we extend the SNGD formulation to support CNNs. Our results show that HyLo outperforms state-of-the-art implementations of NGD, such as KFAC, KAISA, EKFAC, KBFGS-L, on both single-GPU and multi-GPU platforms.

Limitations. In this work, we approximate the Fisher matrix for a batch of data. This stochastic approximation can slow down the convergence when the dataset is very large. Moving average methods can accumulate the second-order information and hence improve the convergence. Nonetheless, it is still unclear how approximate NGD methods affect the convergence in theory. Moreover, in our implementations, we perform many small matrix-matrix multiplications during the preconditions step, which can slow down the training on GPUs. Therefore, batching the small matrix-matrix multiplications can ensure better utilization of GPU and hence improved overall performance.

Chapter 7

Conclusion and Future Works

This chapter summarizes the contributions of this thesis and possibilities for future work in this research area.

7.1 Summary

In this thesis, we study the second-order optimization methods in machine learning. Specifically, this thesis introduces communication-efficient second-order optimization algorithms, in particular, the class of quasi-Newton and natural gradient descent optimization methods, deployed on parallel systems. The summary of each chapter is as follows:

- In chapter 3, we reduce the communication cost of a quasi-Newton method by reformulating the optimization algorithm using iteration overlapping techniques. The proposed algorithm, called RC-SFISTA, is implemented on both MPI and Spark and outperforms state-of-the-art framework ProxCoCoA up to 12 times on 256 processors.

- In chapter 4, we develop a quasi-Newton optimization method that reduces the communication cost on distributed memory systems by using asynchronous communications. The proposed method, called DAve-QN, achieves a superlinear local convergence rate.

- In chapter 5, We develop a novel framework to implement and dispatch asynchronous optimization methods with custom asynchronous execution models on cloud and distributed memory platforms. Our framework, called ASYNC, provides the necessary tools and API to support both first-order and second-order optimization methods with communication models ranging from synchronous to fully asynchronous. ASYNC outperforms the implementations in Spark up to 4 times on a distributed system with 32 workers.

- In chapter 6, we develop a scalable natural gradient descent method, called HyLo, for training deep neural networks on distributed graphical processing units using low-rank approximations and random sampling techniques. We provide an efficient distributed implementation of HyLo which is up to 2.1 times faster than the state-of-the-art distributed implementation of the second-order methods for deep neural networks.

7.2 Future Directions

- In chapter 6, we develop second-order methods using sparsification techniques such as low-rank approximations. In a different direction, quantization methods can be used to reduce the amount of data transferred by quantizing it with a fixed number of bits per dimension. Currently, quantization techniques have not been investigated for the second-order methods. Especially, stochastic quantization techniques are gaining attention in training deep neural networks and can be further extended to second-order optimization methods [41, 240, 241].

- Future works in this area could involve further analysis of second-order methods on hybrid distributed-shared memory architectures. The communication cost in these systems involves both the cost of accessing shared memory and transferring data via the network. Hybrid architectures are becoming more prevalent in recent years because of the increasing popularity of multi-GPU architectures. Designing novel optimization algorithms on these systems require a thorough understanding of the memory hierarchy and the cost of data transfers between these levels. Moreover, recent multi-GPU systems support mixed-precision operations. Therefore, utilizing the new features of these systems can accelerate the performance of optimization methods for many machine learning applications, especially since it has been shown that ML algorithms are robust to round-off errors [242].

- Convergence properties of many approximate second-order methods are still unknown. For example, the only convergence analysis of approximate NGD methods was recently proposed for two layers neural networks [34] and is limited to fully-connected layers. Moreover, the curvature information is obtained partially on a subset of data which introduces randomness in the Hessian. It is still unclear how this randomness plays role in training deep neural networks. Analyzing the convergence properties of approximate second-order methods can lead to a better understanding of how these methods work and open new ways to design novel methods.

Appendix A

Appendix for Reducing Communication in Proximal Newton Methods

A.1 Convergence Proofs for SFISTA

In this section, we show the convergence proof of SFISTA which is stated in Theorem 1.1. First, we consider the full gradient update as:

$$\bar{w}_n = \text{Prox}_\gamma(v_n - \gamma \nabla f(v_n)) \tag{A.1}$$

Let's define $\epsilon_n = \nabla \hat{f}(v_n) - \nabla f(v_n)$, therefore, $\mathbb{E}[\epsilon_n] = 0$. Let $\delta_n = \frac{v_n - w_n}{\gamma}$ be the proximal mapping. Let's define z_n as follows:

$$z_n = t_n w_n + (1 - t_n) w_{n-1} \tag{A.2}$$

Lemma 1. Let $\phi(w)$ be a μ-strongly convex function and $w^* = \underset{w \in \mathbb{R}^d}{\text{argmin}}\, \phi(w)$, then:

$$\phi(w) \geq \phi(w^*) + \frac{\mu}{2}\|w - w^*\|^2 \tag{A.3}$$

Lemma 2. For $n \geq 0$, F is quadratically bounded from below:

$$\begin{aligned} F(w) \geq & F(w_n) + \langle \delta_n, w - v_n \rangle + \langle \epsilon_n, w_n - w \rangle \\ & + \gamma(1 - \frac{\gamma L}{2})\|\delta_n\|^2 \end{aligned} \tag{A.4}$$

Lemma 3. Let $g : \mathbb{R}^d \to \mathbb{R}$ be a convex function. Then for every $x, y \in dom(g)$:

$$\|\text{Prox}_\gamma(x) - \text{Prox}_\gamma(y)\| \leq \|x - y\| \tag{A.5}$$

Lemma 4. Assume we estimate the gradient using (3.9), then based on [185]:

$$\begin{aligned}\mathbb{E}[\|\epsilon_n\|^2] \leq & \frac{1}{\bar{m}} \frac{m-\bar{m}}{m-1}(2L^2\|v_n - w_n\|^2 \\ & +8L(F(w_n) - F(w^*) + F(\hat{w}_s) - F(w^*)))\end{aligned} \tag{A.6}$$

where expectation is taken with respect to $\mathbb{I}_n$.

Lemma 5. Assume that for $n \geq 1$ and $\gamma < \frac{1}{L}$, then

$$\begin{aligned}F(w_n) - F(w) \leq & (1 - t_n^{-1})(F(w_{n-1}) - F(w)) \\ & - \frac{\gamma}{2}(1 - \gamma L)\|\delta_n\|^2 + \gamma\|\epsilon_n\|^2 \\ & + \frac{1}{2\gamma t_n^2}(\|w - z_{n-1}\|^2 - \|w - z_n\|^2) \\ & + \langle \epsilon_n, t_n^{-1}(w - z_{n-1}) + v_n - \bar{w}_n \rangle\end{aligned} \tag{A.7}$$

Proof. Define $\phi_n(w) = \langle \delta_n, w - v_n \rangle + \frac{1}{2\gamma t_n}\|w - z_{n-1}\|^2$. It's easy to see that $z_n = \underset{w \in \mathbb{R}^d}{\operatorname{argmin}} \phi_n(w)$. Also, $\phi_n(w)$ is $\frac{1}{\gamma t_n}$-strongly convex. Therefore by applying Lemma 1 and Lemma 2:

$$\begin{aligned}\phi_n(z_n) &\leq \phi_n(w) - \frac{1}{2\gamma t_n}\|w - z_n\|^2 \\ &= \langle \delta_n, w - v_n \rangle + \frac{1}{2\gamma t_n}\|w - z_{n-1}\|^2 - \frac{1}{2\gamma t_n}\|w - z_n\|^2 \\ &\leq F(w) - F(w_n) + \langle \epsilon_n, w - w_n \rangle \\ &- \gamma(1 - \frac{\gamma L}{2})\|\delta_n\|^2 + \frac{1}{2\gamma t_n}\|w - z_{n-1}\|^2 \\ &- \frac{1}{2\gamma t_n}\|w - z_n\|^2\end{aligned} \tag{A.8}$$

Therefore by plugging $\phi_n(z_n)$ in (A.8),

$$\begin{aligned}F(w_n) \leq F(w) + & < \delta_n, v_n - z_n > - \frac{1}{2\gamma t_n}\|z_n - z_{n-1}\|^2 \\ & + < \epsilon_n, w - w_n > - \gamma(1 - \frac{\gamma L}{2})\|\delta_n\|^2 \\ & + \frac{1}{2\gamma t_n}\|w - z_{n-1}\|^2 - \frac{1}{2\gamma t_n}\|w - z_n\|^2\end{aligned} \tag{A.9}$$

and now by using Lemma 3 and setting $w = w_{n-1}$,

$$\begin{aligned}F(w_n) - F(w_{n-1}) \leq & \langle \delta_n, v_n - w_{n-1} \rangle + \langle \epsilon_n, w_{n-1} - w_n \rangle \\ & - \gamma(1 - \frac{\gamma L}{2})\|\delta_n\|^2\end{aligned} \tag{A.10}$$

by multiplying (A.9) by t_n^{-1} and (A.10) by $(1-t_n^{-1})$ and adding them together:

$$\begin{aligned} F(w_n) - F(w) \leq & (1-t_n^{-1})(F(w_{n-1}) - F(w)) \\ & - \gamma(1-\frac{\gamma L}{2})\|\delta_n\|^2 - \frac{1}{2\gamma t_n^2}\|z_n - z_{n-1}\|^2 \\ & + \frac{1}{2\gamma t_n^2}(\|w - z_{n-1}\|^2 - \|w - z_n\|^2) + \mathcal{A} + \mathcal{B} \end{aligned} \tag{A.11}$$

where $\mathcal{A} = \langle \delta_n, t_n^{-1}(v_n - z_n) + (1-t_n^{-1})(v_n - w_{n-1}))\rangle$ and $\mathcal{B} = t_n^{-1}\langle \epsilon_n, w - w_n\rangle + (1-t_n^{-1})\langle \epsilon_n, w_{n-1} - w_n\rangle$.

Because of the update rule $v_n = (1-t_n^{-1})w_{n-1} + t_n^{-1}z_{n-1}$ and considering that $z_n - z_{n-1} = -t_n\gamma\delta_n$:

$$\begin{aligned} \mathcal{A} &= \langle \delta_n, t_n^{-1}(v_n - z_{n-1}) + (1-t_n^{-1})(v_n - w_{n-1}))\rangle \\ &\quad + \langle \delta_n, t_n^{-1}(z_{n-1} - z_n)\rangle \\ &= \langle \delta_n, t_n^{-1}(z_{n-1} - z_n)\rangle \\ &= \gamma\|\delta_n\|^2 \end{aligned} \tag{A.12}$$

and using v_n update rule, Lemma 3 and Cauchy-Schwartz inequality ,

$$\begin{aligned} \mathcal{B} &= t_n^{-1}\langle \epsilon_n, w - w_n\rangle + (1-t_n^{-1})\langle \epsilon_n, w_{n-1} - w_n\rangle \\ &= \langle \epsilon_n, t_n^{-1}w + (1-t_n^{-1})w_{n-1} - v_n + v_n - w_n\rangle \\ &= \langle \epsilon_n, t_n^{-1}(w - z_{n-1}) + v_n - \bar{w}_n\rangle + \langle \epsilon_n, \bar{w}_n - w_n\rangle \\ &\leq \langle \epsilon_n, t_n^{-1}(w - z_{n-1}) + v_n - \bar{w}_n\rangle + \gamma\|\epsilon_n\|^2 \end{aligned} \tag{A.13}$$

by plugging (A.12) and (A.13) in (A.11) we get the theorem.

A.1.1 Proof of Theorem 1

Since the gradient estimation is unbiased, i.e. $\mathbb{E}[\epsilon_n] = 0$, clearly $\mathbb{E}[\langle \epsilon_n, t_n^{-1}(w^* - z_{n-1}) + v_n - \bar{w}_n\rangle] = 0$, therefore by setting $w = w^*$ and taking expectation of both sides of (A.7) and using (A.6):

$$\begin{aligned} \mathbb{E}[F(w_n)] - F(w^*) \leq & (1-t_n^{-1})(\mathbb{E}[F(w_{n-1})] - F(w^*)) \\ & + \frac{1}{2\gamma t_n^2}(\mathbb{E}[\|w^* - z_{n-1}\|^2] - \mathbb{E}[\|w^* - z_n\|^2]) \\ & + \mathbb{E}\left[(-\frac{\gamma}{2}(1-\gamma L) + \frac{2L^2\gamma^3}{\bar{m}}\frac{m-\bar{m}}{m-1})\|\delta_n\|^2\right] \\ & + \mathbb{E}\Bigg[\frac{8\gamma L(m-\bar{m})}{\bar{m}(m-1)}(F(w_n) - F(w^*) \\ & + F(\hat{w}_s) - F(w^*))\Bigg] \end{aligned} \tag{A.14}$$

now by setting coefficients of $\|\delta_n\|^2$ to be non-positive we have:

$$\gamma^{-1} \geq \max\left(\frac{L}{2} + \sqrt{\Delta}, L\right) \tag{A.15}$$

where $\Delta = \frac{1}{4} + \frac{4L^2(m-\bar{m})}{\bar{m}(m-1)}$. If $\gamma^{-1} > \frac{8L(m-\bar{m})}{\bar{m}(m-1)}$ then,

$$\beta = \frac{8\gamma L(m-\bar{m})}{\bar{m}(m-1)} < 1 \tag{A.16}$$

now by noticing that $t_n^2(1-t_n^{-1}) \leq t_{n-1}^2$

$$\begin{aligned}\mathbb{E}[F(w_n)] - F(w^*) \leq & \frac{t_{n-1}^2}{t_n^2(1-\beta)}(\mathbb{E}[F(w_{n-1})] - F(w^*)) \\ & + \frac{1}{2\gamma t_n^2(1-\beta)}(\mathbb{E}[\|w^* - z_{n-1}\|^2] - \mathbb{E}[\|w^* - z_n\|^2]) \\ & + \frac{\beta}{1-\beta}\mathbb{E}\left[F(\hat{w}_s) - F(w^*)\right]\end{aligned} \tag{A.17}$$

therefore in order to make sure that $\frac{t_{n-1}^2}{t_n^2(1-\beta)} < 1$, we should set $\beta < 1 - \frac{t_{N-1}^2}{t_N^2}$ which using (A.16) :

$$\gamma < \left(1 - \frac{t_{N-1}^2}{t_N^2}\right)\frac{\bar{m}(m-1)}{8L(m-\bar{m})} \tag{A.18}$$

The equations (A.15) and (A.18) together are the conditions on step size. Finally :

$$\begin{aligned}\mathbb{E}[F(w_N)] - F(w^*) \leq & (\frac{1}{t_N^2(1-\beta)^N})(\mathbb{E}[F(\hat{w}_s)] - F(w^*)) \\ & + \frac{D}{2\gamma t_N^2(1-\beta)^N}\end{aligned} \tag{A.19}$$

Appendix B

Appendix for Asynchronous Quasi-Newton Method on Distributed Memory Systems

B.1 DAve-QN Method with Exact Time Indices

We show a detailed version of the DAve-QN algorithm in algorithm B.1.

Algorithm B.1 DAve-QN

Master:

Initialize $\mathbf{x}^0$, $\mathbf{B}_i^0$, $(\mathbf{B}^0)^{-1} = (\sum_{i=1}^n \mathbf{B}_i^0)^{-1}$, $\mathbf{u}^0 = \sum_{i=1}^n \mathbf{B}_i^0 \mathbf{x}^0$, $\mathbf{g}^0 = \sum_{i=1}^n \nabla f_i(\mathbf{x}^0)$
for $t = 1$ **to** *T-1* **do**
  If a worker sends an update:
  Receive $\Delta\mathbf{u}^t$, $\mathbf{y}$, $\mathbf{q}$, α^t, β^t from it
  $\mathbf{u}^t = \mathbf{u}^{t-1} + \Delta\mathbf{u}^t$
  $\mathbf{g}^t = \mathbf{g}^{t-1} + \mathbf{y}$
  $\mathbf{v}^t = (\mathbf{B}^{t-1})^{-1}\mathbf{y}$
  $\mathbf{U}^t = (\mathbf{B}^{t-1})^{-1} - \frac{\mathbf{v}^t \mathbf{v}^{t\top}}{\alpha^t + \mathbf{v}t^\top \mathbf{y}}$
  $\mathbf{w}^t = \mathbf{U}^t \mathbf{q}$
  $(\mathbf{B}^t)^{-1} = \mathbf{U}^t + \frac{\mathbf{w}^t \mathbf{w}^{t\top}}{\beta^t - \mathbf{q}^T \mathbf{w}^t}$
  $\mathbf{x}^t = (\mathbf{B}^t)^{-1}(\mathbf{u}^t - \mathbf{g}^t)$
  Send $\mathbf{x}^t$ to the slave in return
Interrupt all slaves
Output $\mathbf{x}^T$

Slave i**:**

Initialize $\mathbf{x}_i^0 = \mathbf{x}^0$, $\mathbf{B}_i^0$
while *not interrupted by master* **do**
  Receive $\mathbf{x}^{t-D_i^t}$ at moment $t - D_i^t$
  Perform below steps by moment $t - d_i^t$
  $\mathbf{z}_i^t = \mathbf{z}_i^{t-d_i^t} = \mathbf{x}^{t-D_i^t}$
  $\mathbf{s}_i^t = \mathbf{s}_i^{t-d_i^t} = \mathbf{z}^{t-d_i^t} - \mathbf{z}_i^{t-D_i^t}$
  $\mathbf{y}_i^t = \mathbf{y}_i^{t-d_i^t} = \nabla f_i(\mathbf{x}^{t-d_i^t}) - \nabla f_i(\mathbf{z}_i^t)$
  $\mathbf{q}_i^t = \mathbf{q}_i^{t-d_i^t} = \mathbf{B}_i^{t-D_i^t}\mathbf{s}_i^{t-d_i^t}$
  $\alpha^{t-d_i^t} = \mathbf{y}_i^{tT}\mathbf{s}_i^t$
  $\beta^{t-d_i^t} = (\mathbf{s}_i^{t-d_i^t})^T \mathbf{B}_i^{t-D_i^t}\mathbf{s}_i^t$
  $\mathbf{B}_i^{t-d_i^t} = \mathbf{B}_i^{t-D_i^t} + \frac{\mathbf{y}_i^t \mathbf{y}_i^{tT}}{\alpha^{t-d_i^t}} - \frac{\mathbf{q}_i^t \mathbf{q}_i^{tT}}{\beta^{t-d_i^t}}$
  $\Delta\mathbf{u}^{t-d_i^t} = \mathbf{B}_i^{t-d_i^t}\mathbf{z}_i^{t-d_i^t} - \mathbf{B}_i^{t-D_i^t}\mathbf{z}_i^{t-D_i^t}$
  Send $\Delta\mathbf{u}^{t-d_i^t}, \mathbf{y}_i^{t-d_i^t}, \mathbf{q}_i^{t-d_i^t}, \alpha^{t-d_i^t}, \beta^{t-d_i^t}$ to the master at moment $t - d_i^t$

B.2 Proofs

In this section, we show the proofs for Lemmas and Theorems of DAve-QN.

B.2.1 Proof of Lemma 1

Proof. To verify the claim, we need to show that $\mathbf{u}^t = \sum_{i=1}^n \mathbf{B}_i^t \mathbf{z}_i^t$ and $\mathbf{g}^t = \sum_{i=1}^n \nabla f_i(\mathbf{z}_i^t)$. They follow from our delayed vectors notation $\mathbf{z}_i^t = \mathbf{z}_i^{t-d_i^t}$ and how $\Delta\mathbf{u}^{t-d_i^t}$ and $\mathbf{y}_i^{t-d_i^t}$ are computed by the corresponding worker. □

B.2.2 Proof of Lemma 2

To prove the claim in Lemma 2 we first prove the following intermediate lemma using the result of Lemma 5.2 in [177].

Lemma 4. *Consider the proposed method outlined in Algorithm B.1. Let* $\mathbf{M}$ *be a nonsingular symmetric matrix such that*

$$\|\mathbf{M}\mathbf{y}_i^t - \mathbf{M}^{-1}\mathbf{s}_i^t\| \leq \beta\|\mathbf{M}^{-1}\mathbf{s}_i^t\|, \tag{B.1}$$

for some $\beta \in [0, 1/3]$ *and vectors* $\mathbf{s}_i^t$ *and* $\mathbf{y}_i^t$ *in* $\mathbb{R}^p$ *with* $\mathbf{s}_i^t \neq \mathbf{0}$*. Let's denote* i *as the index that has been updated at time* t*. Then, there exist positive constants* α*,* α_1*, and* α_2 *such that, for any symmetric* $\mathbf{A} \in \mathbb{R}^{p\times p}$ *we have,*

$$\begin{aligned}\|\mathbf{B}_i^t - \mathbf{A}\|_{\mathbf{M}} \leq &\left[(1-\alpha\theta^2)^{1/2} + \alpha_1 \frac{\|\mathbf{M}\mathbf{y}_i^{t-D_i^t} - \mathbf{M}^{-1}\mathbf{s}_i^{t-D_i^t}\|}{\|\mathbf{M}^{-1}\mathbf{s}_i^{t-D_i^t}\|}\right]\|\mathbf{B}_i^{t-D_i^t} - \mathbf{A}\|_{\mathbf{M}} \\ &+ \alpha_2 \frac{\|\mathbf{y}_i^{t-D_i^t} - \mathbf{A}\mathbf{s}_i^{t-D_i^t}\|}{\|\mathbf{M}^{-1}\mathbf{s}_i^{t-D_i^t}\|},\end{aligned} \tag{B.2}$$

where $\alpha = (1-2\beta)/(1-\beta^2) \in [3/8, 1]$*,* $\alpha_1 = 2.5(1-\beta)^{-1}$*,* $\alpha_2 = 2(1+2\sqrt{p})\|\mathbf{M}\|_{\mathbf{F}}$*, and*

$$\theta = \frac{\|\mathbf{M}(\mathbf{B}_i^{t-D_i^t} - \mathbf{A})\mathbf{s}_i^{t-D_i^t}\|}{\|\mathbf{B}_i^{t-D_i^t} - \mathbf{A}\|_{\mathbf{M}}\|\mathbf{M}^{-1}\mathbf{s}_i^{t-D_i^t}\|} \quad \text{for } \mathbf{B}_i^{t-D_i^t} \neq \mathbf{A}, \quad \theta = 0 \quad \text{for } \mathbf{B}_i^{t-D_i^t} = \mathbf{A}. \tag{B.3}$$

Proof. By definition of delays d_i^t, the function f_i was updated at step $t - d_i^t$ and $\mathbf{B}_i^{t-1}$ is equal to $\mathbf{B}_i^{t-D_i^t}$. Considering this observation and the result of Lemma 5.2 in [177], the claim follows. □

Note that the result in Lemma 4 characterizes an upper bound on the difference between the Hessian approximation matrices $\mathbf{B}_i^t$ and $\mathbf{B}_i^{t-D_i^t}$ and any positive definite matrix $\mathbf{A}$. Let us show that matrices $\mathbf{M} = \nabla^2 f_i(\mathbf{x}^*)^{-1/2}$ and $\mathbf{A} = \nabla^2 f_i(\mathbf{x}^*)$ satisfy the conditions of Lemma 4. By strong convexity of f_i we have $\|\nabla^2 f_i(\mathbf{x}^*)^{1/2}\mathbf{s}_i^{t-D_i^t}\| \geq \sqrt{\mu}\|\mathbf{s}_i^{t-D_i^t}\|$. Combined with Assumption 2, it gives that

$$\frac{\|\mathbf{y}_i^{t-D_i^t} - \nabla^2 f_i(\mathbf{x}^*)\mathbf{s}_i^{t-D_i^t}\|}{\|\nabla^2 f_i(\mathbf{x}^*)^{1/2}\mathbf{s}_i^{t-D_i^t}\|} \leq \frac{\tilde{L}\|\mathbf{s}_i^{t-D_i^t}\|\max\{\|\mathbf{z}_i^{t-D_i^t} - \mathbf{x}^*\|, \|\mathbf{z}_i^t - \mathbf{x}^*\|\}}{\sqrt{\mu}\|\mathbf{s}_i^{t-D_i^t}\|} = \frac{\tilde{L}}{\sqrt{\mu}}\sigma_i^t \tag{B.4}$$

This observation implies that the left hand side of the condition in (B.1) for $\mathbf{M} = \nabla^2 f_i(\mathbf{x}^*)^{-1/2}$ is

bounded above by

$$\frac{\|\mathbf{M}\mathbf{y}_i^{t-D_i^t} - \mathbf{M}^{-1}\mathbf{s}_i^{t-D_i^t}\|}{\|\mathbf{M}^{-1}\mathbf{s}_i^{t-D_i^t}\|} \leq \frac{\|\nabla^2 f_i(\mathbf{x}^*)^{-1/2}\|\|\mathbf{y}_i^{t-D_i^t} - \nabla^2 f_i(\mathbf{x}^*)\mathbf{s}_i^{t-D_i^t}\|}{\|\nabla^2 f_i(\mathbf{x}^*)^{1/2}\mathbf{s}_i^{t-D_i^t}\|} \leq \frac{\tilde{L}}{\mu}\sigma_i^t \tag{B.5}$$

Thus, the condition in (B.1) is satisfied since $\tilde{L}\sigma_i^t/\mu < 1/3$. Replacing the upper bounds in (B.4) and (B.5) into the expression in (B.2) implies the claim in (4.20) with

$$\beta = \frac{\tilde{L}}{\mu}\sigma_i^t,\ \alpha = \frac{1-2\beta}{1-\beta^2},\ \alpha_3 = \frac{5\tilde{L}}{2\mu(1-\beta)},\ \alpha_4 = \frac{2(1+2\sqrt{p})\tilde{L}}{\sqrt{\mu}}\|\nabla^2 f_i(\mathbf{x}^*)^{-\frac{1}{2}}\|_{\mathbf{F}}, \tag{B.6}$$

and the proof is complete.

B.2.3 Proof of Lemma 3

We first state the following result from Lemma 6 in [164], which shows an upper bound for the error $\|\mathbf{x}^t - \mathbf{x}^*\|$ in terms of the gap between the delayed variables $\mathbf{z}_i^t$ and the optimal solution $\mathbf{x}^*$ and the difference between the Newton direction $\nabla^2 f_i(\mathbf{x}^*)\,(\mathbf{z}_i^t - \mathbf{x}^*)$ and the proposed quasi-Newton direction $\mathbf{B}_i^t\,(\mathbf{z}_i^t - \mathbf{x}^*)$.

Lemma 5. *If Assumptions 1 and 2 hold, then the sequence of iterates generated by Algorithm B.1 satisfies*

$$\|\mathbf{x}^t - \mathbf{x}^*\| \leq \frac{\tilde{L}\Gamma^t}{n}\sum_{i=1}^n \left\|\mathbf{z}_i^t - \mathbf{x}^*\right\|^2 + \frac{\Gamma^t}{n}\sum_{i=1}^n \left\|\left(\mathbf{B}_i^t - \nabla^2 f_i(\mathbf{x}^*)\right)\left(\mathbf{z}_i^t - \mathbf{x}^*\right)\right\|, \tag{B.7}$$

where $\Gamma^t := \|((1/n)\sum_{i=1}^n \mathbf{B}_i^t)^{-1}\|$.

We use the result in Lemma 5 to prove the claim of Lemma 3. We will prove the claimed convergence rate in Lemma 3together with an additional claim

$$\left\|\mathbf{B}_i^t - \nabla^2 f_i(\mathbf{x}^*)\right\|_{\mathbf{M}} \leq 2\delta$$

by inductions on m and on $t \in [T_m, T_{m+1})$. The base case of our induction is $m = 0$ and $t = 0$, which is the initialization step, so let us start with it.

Since all norms in finite dimensional spaces are equivalent, there exists a constant $\eta > 0$ such that $\|\mathbf{A}\| \leq \eta\|\mathbf{A}\|_{\mathbf{M}}$ for all $\mathbf{A}$. Define $\gamma := 1/\mu$ and $d := \max_m(T_{m+1} - T_m)$, and assume that $\epsilon(r) = \epsilon$ and $\delta(r) = \delta$ are chosen such that

$$(2\alpha_3\delta + \alpha_4)\frac{d\epsilon}{1-r} \leq \delta \quad \text{and} \quad \gamma(1+r)[\tilde{L}\epsilon + 2\eta\delta] \leq r, \tag{B.8}$$

where α_3 and α_4 are the constants from Lemma 2. As $\|\mathbf{B}_i^0 - \nabla^2 f_i(\mathbf{x}^*)\|_{\mathbf{M}} \leq \delta$, we also have

$$\|\mathbf{B}_i^0 - \nabla^2 f_i(\mathbf{x}^*)\| \leq \eta\delta.$$

Therefore, by triangle inequality from $\|\nabla^2 f_i(\mathbf{x}^*)\| \leq L$ we obtain $\|\mathbf{B}_i^0\| \leq \eta\delta + L$, so $\|(1/n)\sum_{i=1}^n \mathbf{B}_i^0\| \leq \eta\delta + L$. The second part of inequality (B.8) also implies $2\gamma(1+r)\eta\delta \leq r$. Moreover, it holds that

$\|\mathbf{B}_i^0 - \nabla^2 f_i(\mathbf{x}^*)\| \leq \eta\delta < 2\eta\delta$ and by Assumption 1 $\gamma \geq \|\nabla^2 f_i(\mathbf{x}^*)^{-1}\|$, so we obtain by Banach Lemma that

$$\|(\mathbf{B}_i^0)^{-1}\| \leq (1+r)\gamma.$$

We formally prove this result in the following lemma.

Lemma 6. *If the Hessian approximation $\mathbf{B}_i$ satisfies the inequality $\|\mathbf{B}_i - \nabla^2 f_i(\mathbf{x}^*)\| \leq 2\eta\delta$ and $\|\nabla^2 f_i(\mathbf{x}^*)^{-1}\| \leq \gamma$, then we have $\|\mathbf{B}_i^{-1}\| \leq (1+r)\gamma$.*

Proof. Note that according to Banach Lemma, if a matrix $\mathbf{A}$ satisfies the inequality $\|\mathbf{A} - \mathbf{I}\| \leq 1$, then it holds $\|\mathbf{A}^{-1}\| \leq \frac{1}{1-\|\mathbf{A}-\mathbf{I}\|}$.

We first show that $\|\nabla^2 f_i(\mathbf{x}^*)^{-1/2}\mathbf{B}_i\nabla^2 f_i(\mathbf{x}^*)^{-1/2} - \mathbf{I}\| \leq 1$. To do so, note that

$$\begin{aligned}\|\nabla^2 f_i(\mathbf{x}^*)^{-1/2}\mathbf{B}_i\nabla^2 f_i(\mathbf{x}^*)^{-1/2} - \mathbf{I}\| &\leq \|\nabla^2 f_i(\mathbf{x}^*)^{-1/2}\|\|\mathbf{B}_i - \nabla^2 f_i(\mathbf{x}^*)\|\|\nabla^2 f_i(\mathbf{x}^*)^{-1/2}\| \\ &\leq 2\eta\delta\gamma \\ &\leq \frac{r}{r+1} \\ &< 1. \end{aligned} \tag{B.9}$$

Now using this result and Banach Lemma we can show that

$$\begin{aligned}\|\nabla^2 f_i(\mathbf{x}^*)^{1/2}\mathbf{B}_i^{-1}\nabla^2 f_i(\mathbf{x}^*)^{1/2}\| &\leq \frac{1}{1 - \|\nabla^2 f_i(\mathbf{x}^*)^{-1/2}\mathbf{B}_i\nabla^2 f_i(\mathbf{x}^*)^{-1/2} - \mathbf{I}\|} \\ &\leq \frac{1}{1 - \frac{r}{r+1}} \\ &= 1 + r \end{aligned} \tag{B.10}$$

Further, we know that

$$\|\nabla^2 f_i(\mathbf{x}^*)^{1/2}\mathbf{B}_i^{-1}\nabla^2 f_i(\mathbf{x}^*)^{1/2}\| \geq \frac{\|\mathbf{B}_i^{-1}\|}{\gamma} \tag{B.11}$$

By combining these results we obtain that

$$\|\mathbf{B}_i^{-1}\| \leq (1+r)\gamma. \tag{B.12}$$

□

Similarly, for matrix $((1/n)\sum_{i=1}^n \mathbf{B}_i^0)^{-1}$ we get from $\|(1/n)\sum_{i=1}^n \mathbf{B}_i^0 - (1/n)\sum_{i=1}^n \nabla^2 f_i(\mathbf{x}^*)\| \leq (1/n)\sum_{i=1}^n \|\mathbf{B}_i^0 - \nabla^2 f_i(\mathbf{x}^*)\| \leq \eta\delta$ and $\|\nabla^2 f(\mathbf{x}^*)^{-1}\| \leq \gamma$ that

$$\left\| \left(\frac{1}{n}\sum_{i=1}^n \mathbf{B}_i^0 \right)^{-1} \right\| \leq (1+r)\gamma.$$

We have by Lemma 2 and induction hypothesis

$$\begin{aligned}\left\|\mathbf{B}_i^t - \nabla^2 f_i(\mathbf{x}^*)\right\|_{\mathbf{M}} - \left\|\mathbf{B}_i^{t-D_i^t} - \nabla^2 f_i(\mathbf{x}^*)\right\|_{\mathbf{M}} &\leq \alpha_3 \sigma_i^{t-D_i^t} \left\|\mathbf{B}_i^{t-D_i^t} - \nabla^2 f_i(\mathbf{x}^*)\right\|_{\mathbf{M}} + \alpha_4 \sigma_i^{t-D_i^t} \\ &\leq \left(\alpha_3 \left\|\mathbf{B}_i^{t-D_i^t} - \nabla^2 f_i(\mathbf{x}^*)\right\|_{\mathbf{M}} + \alpha_4\right) r^{m-1}\epsilon \\ &\leq \left(2\alpha_3\delta + \alpha_4\right) r^{m-1}\epsilon,\end{aligned}$$

By summing this inequality over all moments in the current epoch when worker i performed its update, we obtain that

$$\left\|\mathbf{B}_i^t - \nabla^2 f_i(\mathbf{x}^*)\right\|_{\mathbf{M}} - \left\|\mathbf{B}_i^{T_m - d_i^{T_m}} - \nabla^2 f_i(\mathbf{x}^*)\right\|_{\mathbf{M}} \leq \left(2\alpha_3\delta + \alpha_4\right) d r^{m-1}\epsilon,$$

Summing the new bound again, but this time over all passed epoch, we obtain

$$\left\|\mathbf{B}_i^t - \nabla^2 f_i(\mathbf{x}^*)\right\|_{\mathbf{M}} - \left\|\mathbf{B}_i^0 - \nabla^2 f_i(\mathbf{x}^*)\right\|_{\mathbf{M}} \leq \left(2\alpha_3\delta + \alpha_4\right) d\epsilon \sum_{k=0}^{m-1} r^k \leq \frac{\left(2\alpha_3\delta + \alpha_4\right) d\epsilon}{1-r} \leq \delta.$$

Therefore, $\left\|\mathbf{B}_i^t - \nabla^2 f_i(\mathbf{x}^*)\right\|_{\mathbf{M}} \leq 2\delta$. By using the Banach argument again, we can show that $\|(\frac{1}{n}\sum_{i=1}^n \mathbf{B}_i^t)^{-1}\| \leq (1+r)\gamma$. Using this result, for any $t \in [T_m, T_{m+1})$ we have $z_i^t = x^{t-D_i^t} \in [T_{m-1}, t)$ and we can write

$$\begin{aligned}\|\mathbf{x}^t - \mathbf{x}^*\| &\leq (1+r)\gamma \left[\frac{\tilde{L}}{n}\sum_{i=1}^n \left\|\mathbf{z}_i^t - \mathbf{x}^*\right\|^2 + \frac{1}{n}\sum_{i=1}^n \left\|\left[\mathbf{B}_i^t - \nabla^2 f_i(\mathbf{x}^*)\right]\left(\mathbf{z}_i^t - \mathbf{x}^*\right)\right\|\right] \\ &\leq (1+r)\gamma \left[\tilde{L}\epsilon + 2\eta\delta\right] \max_i \|\mathbf{z}_i^t - \mathbf{x}^*\| \\ &\leq r \max_i \|\mathbf{z}_i^t - \mathbf{x}^*\| \\ &\leq r^m \|\mathbf{x}^0 - \mathbf{x}^*\|. \end{aligned} \tag{B.13}$$

B.2.4 Proof of Theorem 1

Dividing both sides of (B.7) by $(1/n)\sum_{i=1}^n \|\mathbf{z}_i^t - \mathbf{x}^*\|$, we get

$$\frac{\|\mathbf{x}^t - \mathbf{x}^*\|}{\frac{1}{n}\sum_{i=1}^n \|\mathbf{z}_i^t - \mathbf{x}^*\|} \leq \tilde{L}\Gamma^t \sum_{i=1}^n \frac{\|\mathbf{z}_i^t - \mathbf{x}^*\|^2}{\sum_{i=1}^n \|\mathbf{z}_i^t - \mathbf{x}^*\|} + \Gamma^t \sum_{i=1}^n \frac{\left\|\left(\mathbf{B}_i^t - \nabla^2 f_i(\mathbf{x}^*)\right)\left(\mathbf{z}_i^t - \mathbf{x}^*\right)\right\|}{\sum_{i=1}^n \|\mathbf{z}_i^t - \mathbf{x}^*\|} \tag{B.14}$$

As every term in $\sum_{i=1}^n \|\mathbf{z}_i^t - \mathbf{x}^*\|$ is non-negative, the upper bound in (B.14) will remain valid if we keep only one summand out of the whole sum in the denominators of the right-hand side, so

$$\begin{aligned}\frac{\|\mathbf{x}^t - \mathbf{x}^*\|}{\frac{1}{n}\sum_{i=1}^n \|\mathbf{z}_i^t - \mathbf{x}^*\|} &\leq \tilde{L}\Gamma^t \sum_{i=1}^n \frac{\|\mathbf{z}_i^t - \mathbf{x}^*\|^2}{\|\mathbf{z}_i^t - \mathbf{x}^*\|} + \Gamma^t \sum_{i=1}^n \frac{\left\|\left(\mathbf{B}_i^t - \nabla^2 f_i(\mathbf{x}^*)\right)\left(\mathbf{z}_i^t - \mathbf{x}^*\right)\right\|}{\|\mathbf{z}_i^t - \mathbf{x}^*\|} \\ &= \tilde{L}\Gamma^t \sum_{i=1}^n \left\|\mathbf{z}_i^t - \mathbf{x}^*\right\| + \Gamma^t \sum_{i=1}^n \frac{\left\|\left(\mathbf{B}_i^t - \nabla^2 f_i(\mathbf{x}^*)\right)\left(\mathbf{z}_i^t - \mathbf{x}^*\right)\right\|}{\|\mathbf{z}_i^t - \mathbf{x}^*\|}. \end{aligned} \tag{B.15}$$

Now using the result in Lemma 5 of [164], the second sum in (B.15) converges to zero. Further, Γ^t is bounded above by a positive constant. Hence, by computing the limit of both sides in (B.15)

we obtain

$$\lim_{t\to\infty} \frac{\|\mathbf{x}^t - \mathbf{x}^*\|}{\frac{1}{n}\sum_{i=1}^n \|\mathbf{z}_i^t - \mathbf{x}^*\|} = 0.$$

Therefore, if T is big enough, for $t > T$ we have

$$\|\mathbf{x}^t - \mathbf{x}^*\| \leq \frac{1}{n}\sum_{i=1}^n \left\|\mathbf{z}_i^t - \mathbf{x}^*\right\| = \frac{1}{n}\sum_{i=1}^n \left\|\mathbf{x}_i^{t-D_i^t} - \mathbf{x}^*\right\| \leq \max_i \left\|\mathbf{x}_i^{t-D_i^t} - \mathbf{x}^*\right\|. \tag{B.16}$$

Now, let $t_0 = t_0(m) \coloneqq \min\{\tilde{t} \in [T_{m+1}, T_{m+2}) : \|\mathbf{x}^{\tilde{t}} - \mathbf{x}^*\| = \max_{t\in[T_{m+1},T_{m+2})} \|\mathbf{x}^t - \mathbf{x}^*\|\}$. In other words, t_0 is the first moment in epoch $m+1$ attaining the maximal distance from x^*. Then, for all $t \in [T_{m+1}, t_0)$ we have $\|\mathbf{x}^t - \mathbf{x}^*\| < \|\mathbf{x}^{t_0} - \mathbf{x}^*\|$. Furthermore, from equation (B.16) and the fact that, according to Proposition 1, $t_0 - D_i^{t_0} \in [T_m, t_0)$ we get

$$\max_{t\in[T_{m+1},T_{m+2})} \|\mathbf{x}^t - \mathbf{x}^*\| = \|\mathbf{x}^{t_0} - \mathbf{x}^*\| \leq \max_i \left\|\mathbf{x}_i^{t_0-D_i^{t_0}} - \mathbf{x}^*\right\| \leq \max_{t\in[T_m,t_0)} \|\mathbf{x}^t - \mathbf{x}^*\|.$$

Note that it can not happen that $\max_{t\in[T_m,t_0)} \|\mathbf{x}^t - \mathbf{x}^*\| = \max_{t\in[T_{m+1},t_0)} \|\mathbf{x}^t - \mathbf{x}^*\|$ as that would mean that there exists a $\hat{t} \in [T_{m+1}, t_0)$ such that $\|\mathbf{x}^{\hat{t}} - \mathbf{x}^*\| \geq \|\mathbf{x}^{t_0} - \mathbf{x}^*\|$, which we made impossible when defining t_0. Then, the only option is that in fact

$$\max_{t\in[T_m,t_0)} \|\mathbf{x}^t - \mathbf{x}^*\| = \max_{t\in[T_m,T_{m+1})} \|\mathbf{x}^t - \mathbf{x}^*\|.$$

Finally,

$$\begin{aligned}
\lim_{t\to\infty} \frac{\max_{t\in[T_{m+1},T_{m+2})} \|\mathbf{x}^t - \mathbf{x}^*\|}{\max_{t\in[T_m,T_{m+1})} \|\mathbf{x}^t - \mathbf{x}^*\|} &= \lim_{t\to\infty} \frac{\|\mathbf{x}^{t_0(m)} - \mathbf{x}^*\|}{\max_{t\in[T_m,T_{m+1})} \|\mathbf{x}^t - \mathbf{x}^*\|} \\
&= \lim_{t\to\infty} \frac{\|\mathbf{x}^{t_0(m)} - \mathbf{x}^*\|}{\max_{t\in[T_m,t_0(m))} \|\mathbf{x}^t - \mathbf{x}^*\|} \\
&\leq \lim_{t\to\infty} \frac{\|\mathbf{x}^{t_0(m)} - \mathbf{x}^*\|}{\max_i \|\mathbf{x}^{t_0(m)-D_i^{t_0(m)}} - \mathbf{x}^*\|} \\
&\leq \lim_{t\to\infty} \frac{\|\mathbf{x}^{t_0(m)} - \mathbf{x}^*\|}{\frac{1}{n}\sum_{i=1}^n \|\mathbf{z}_i^{t_0(m)} - \mathbf{x}^*\|} = 0,
\end{aligned}$$

where at the last step we used again the fact that $\mathbf{z}_i^t = \mathbf{x}^{t-D_i^t}$.

B.3 Implementation of DAve-QN

In Algorithm 4.1, we provide a simplified version of the DAve-QN in terms of notation and indices of the variables, which illustrates how DAve-QN can be implemented from master's and worker nodes' side further. We observe the steps at which worker i is devoted to performing the update in (4.11). Using the computed matrix $\mathbf{B}_i$, node i evaluates the vector $\Delta\mathbf{u}$. Then, it sends the vectors $\Delta\mathbf{u}$, $\mathbf{y}_i$, and $\mathbf{q}_i$ as well as the scalars α and β to the master node. The master node uses the variation vectors $\Delta\mathbf{u}$ and $\mathbf{y}$ to update $\mathbf{u}$ and $\mathbf{g}$. Then, it performs the update $\mathbf{x}^{t+1} = (\mathbf{B}^{t+1})^{-1}\left(\mathbf{u}^{t+1} - \mathbf{g}^{t+1}\right)$ by following the efficient procedure presented in (4.16)–(4.17).

Bibliography

[1] Maryellen L Giger. Machine learning in medical imaging. *Journal of the American College of Radiology*, 15(3):512–520, 2018.

[2] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[4] Dmitrii Kochkov, Jamie A Smith, Ayya Alieva, Qing Wang, Michael P Brenner, and Stephan Hoyer. Machine learning–accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences*, 118(21), 2021.

[5] Konstantinos Vassakis, Emmanuel Petrakis, and Ioannis Kopanakis. Big data analytics: applications, prospects and challenges. In *Mobile big data*, pages 3–20. Springer, 2018.

[6] Ron Bekkerman, Mikhail Bilenko, and John Langford. *Scaling up machine learning: Parallel and distributed approaches*. Cambridge University Press, 2011.

[7] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. Large-scale distributed second-order optimization using Kronecker-factored approximate curvature for deep convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12359–12367, 2019.

[8] Rustem Islamov, Xun Qian, and Peter Richtárik. Distributed second order methods with fast rates and compressed communication. In *International Conference on Machine Learning*, pages 4617–4628. PMLR, 2021.

[9] Celestine Dünner, Aurelien Lucchi, Matilde Gargiani, An Bian, Thomas Hofmann, and Martin Jaggi. A Distributed Second-Order Algorithm You Can Trust. *arXiv e-prints*, page arXiv:1806.07569, Jun 2018.

[10] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901, 2011.

[11] James Demmel. Communication-avoiding algorithms for linear algebra and beyond. In *IPDPS*, page 585, 2013.

[12] Chaoyue Liu and Mikhail Belkin. Accelerating sgd with momentum for over-parameterized learning. In *International Conference on Learning Representations*, 2020.

[13] Atsushi Nitanda. Accelerated stochastic gradient descent for minimizing finite sums. In *Artificial Intelligence and Statistics*, pages 195–203. PMLR, 2016.

[14] X. Lian, M. Wang, and J. Liu. Finite-sum composition optimization via variance reduced gradient descent. In *Artificial Intelligence and Statistics*, pages 1159–1167, 2017.

[15] Dimitri Bertsekas and John Tsitsiklis. *Parallel and distributed computation: numerical methods*. Athena Scientific, 2015.

[16] Zhi-Quan Luo and Paul Tseng. On the convergence of the coordinate descent method for convex differentiable minimization. *Journal of Optimization Theory and Applications*, 72(1):7–35, 1992.

[17] Alekh Agarwal and John C Duchi. Distributed delayed stochastic optimization. In *Advances in Neural Information Processing Systems*, pages 873–881, 2011.

[18] Ashok Cutkosky and Róbert Busa-Fekete. Distributed stochastic optimization via adaptive sgd. *Advances in Neural Information Processing Systems*, 31, 2018.

[19] Shai Shalev-Shwartz and Tong Zhang. Stochastic dual coordinate ascent methods for regularized loss. *The Journal of Machine Learning Research*, 14:567–599, 2013.

[20] Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. Scalable second order optimization for deep learning. *arXiv preprint arXiv:2002.09018*, 7:15, 2021.

[21] John L Nazareth. Conjugate gradient method. *Wiley Interdisciplinary Reviews: Computational Statistics*, 1(3):348–353, 2009.

[22] James Martens et al. Deep learning via hessian-free optimization. In *ICML*, volume 27, pages 735–742, 2010.

[23] Majid Jahani, Xi He, Chenxin Ma, Aryan Mokhtari, Dheevatsa Mudigere, Alejandro Ribeiro, and Martin Takác. Efficient distributed hessian free algorithm for large-scale empirical risk minimization via accumulating sample strategy. In *International Conference on Artificial Intelligence and Statistics*, pages 2634–2644. PMLR, 2020.

[24] Huishuai Zhang, Caiming Xiong, James Bradbury, and Richard Socher. Block-diagonal hessian-free optimization for training neural networks. *arXiv preprint arXiv:1712.07296*, 2017.

[25] Yu-Hong Dai. Convergence properties of the bfgs algoritm. *SIAM Journal on Optimization*, 13(3):693–701, 2002.

[26] Farbod Roosta-Khorasani and Michael W Mahoney. Sub-sampled Newton methods. *Mathematical Programming*, 174(1):293–326, 2019.

[27] Joanna Maria Papakonstantinou. *Historical development of the BFGS secant method and its characterization properties*. Rice University, 2009.

[28] Nicol N Schraudolph, Jin Yu, and Simon Günter. A stochastic quasi-Newton method for online convex optimization. In *International Conference on Artificial Intelligence and Statistics*, pages 436–443, 2007.

[29] Aryan Mokhtari and Alejandro Ribeiro. Stochastic quasi-Newton methods. *Proceedings of the IEEE*, 108(11):1906–1922, 2020.

[30] Richard H Byrd, SL Hansen, Jorge Nocedal, and Yoram Singer. A stochastic quasi-Newton method for large-scale optimization. *SIAM Journal on Optimization*, 26(2):1008–1031, 2016.

[31] Stephen Wright, Jorge Nocedal, et al. Numerical optimization. *Springer Science*, 35(67-68):7, 1999.

[32] Aleksandar Botev, Hippolyt Ritter, and David Barber. Practical Gauss-Newton optimisation for deep learning. In *International Conference on Machine Learning*, pages 557–565. PMLR, 2017.

[33] James Martens and Roger Grosse. Optimizing neural networks with Kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015.

[34] Guodong Zhang, James Martens, and Roger Grosse. Fast convergence of natural gradient descent for overparameterized neural networks. *arXiv preprint arXiv:1905.10961*, 2019.

[35] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and TrendsÂ® in Machine Learning*, 3(1):1–122, 2011.

[36] Jack Dongarra, Jeffrey Hittinger, John Bell, Luis Chacon, Robert Falgout, Michael Heroux, Paul Hovland, Esmond Ng, Clayton Webster, and Stefan Wild. Applied mathematics research for exascale computing. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2014.

[37] Mark Hoemmen. *Communication-avoiding Krylov subspace methods*. University of California, Berkeley, 2010.

[38] Aditya Devarakonda, Kimon Fountoulakis, James Demmel, and Michael W Mahoney. Avoiding communication in primal and dual block coordinate descent methods. *arXiv preprint arXiv:1612.04003*, 2016.

[39] Yang You, James Demmel, Kenneth Czechowski, Le Song, and Richard Vuduc. Ca-svm: Communication-avoiding support vector machines on distributed systems. In *IEEE International Parallel and Distributed Processing Symposium*, pages 847–859, 2015.

[40] Penporn Koanantakool, Alnur Ali, Ariful Azad, Aydin Buluc, Dmitriy Morozov, Leonid Oliker, Katherine Yelick, and Sang-Yun Oh. Communication-avoiding optimization methods for distributed massive-scale sparse inverse covariance estimation. In *International Conference on Artificial Intelligence and Statistics*, pages 1376–1386. PMLR, 2018.

[41] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. *Advances in Neural Information Processing Systems*, 30, 2017.

[42] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. *Advances in neural information processing systems*, 30, 2017.

[43] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. Gradient sparsification for communication-efficient distributed optimization. *Advances in Neural Information Processing Systems*, 31, 2018.

[44] Ali Ramezani-Kebrya, Fartash Faghri, Ilya Markov, Vitalii Aksenov, Dan Alistarh, and Daniel M Roy. Nuqsgd: Provably communication-efficient data-parallel sgd via nonuniform quantization. *Journal of Machine Learning Research*, 22(114):1–43, 2021.

[45] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*. Citeseer, 2014.

[46] Foivos Alimisis, Peter Davies, and Dan Alistarh. Communication-efficient distributed optimization with quantized preconditioners. *CoRR*, abs/2102.07214, 2021.

[47] Dan Alistarh, Torsten Hoefler, Mikael Johansson, Nikola Konstantinov, Sarit Khirirat, and Cedric Renggli. The convergence of sparsified gradient methods. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[48] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[49] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. Gradient sparsification for communication-efficient distributed optimization. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[50] Yi Cai, Yujun Lin, Lixue Xia, Xiaoming Chen, Song Han, Yu Wang, and Huazhong Yang. Long live time: improving lifetime for training-in-memory engines by structured gradient sparsification. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

[51] Ekaterina Lobacheva, Nadezhda Chirkova, Alexander Markovich, and Dmitry Vetrov. Structured sparsification of gated recurrent neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 4989–4996, 2020.

[52] James Martens. New insights and perspectives on the natural gradient method. *Journal of Machine Learning Research*, 21(146):1–76, 2020.

[53] Yi Ren and Donald Goldfarb. Efficient subsampled Gauss-Newton and natural gradient methods for training neural networks. *arXiv preprint arXiv:1906.02353*, 2019.

[54] Roger Grosse and James Martens. A Kronecker-factored approximate Fisher matrix for convolution layers. In *International Conference on Machine Learning*, pages 573–582. PMLR, 2016.

[55] Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhi-Ming Ma, and Tie-Yan Liu. Asynchronous stochastic gradient descent with delay compensation. In *International Conference on Machine Learning*, pages 4120–4129. PMLR, 2017.

[56] Fatemeh Mansoori and Ermin Wei. Superlinearly convergent asynchronous distributed network Newton method. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 2874–2879. IEEE, 2017.

[57] Mahmoud Assran, Arda Aytekin, Hamid Reza Feyzmahdavian, Mikael Johansson, and Michael G Rabbat. Advances in asynchronous parallel and distributed optimization. *Proceedings of the IEEE*, 108(11):2013–2031, 2020.

[58] Konstantin Mishchenko, Franck Iutzeler, Jérôme Malick, and Massih-Reza Amini. A delay-tolerant proximal-gradient algorithm for distributed learning. In *International Conference on Machine Learning*, pages 3587–3595. PMLR, 2018.

[59] Tianyi Chen, Georgios Giannakis, Tao Sun, and Wotao Yin. Lag: Lazily aggregated gradient for communication-efficient distributed learning. *Advances in Neural Information Processing Systems*, 31, 2018.

[60] Huan Zhang, Cho-Jui Hsieh, and Venkatesh Akella. Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 629–638. IEEE, 2016.

[61] Umut Simsekli, Cagatay Yildiz, Than Huy Nguyen, Taylan Cemgil, and Gael Richard. Asynchronous stochastic quasi-Newton mcmc for non-convex optimization. In *International Conference on Machine Learning*, pages 4674–4683. PMLR, 2018.

[62] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[63] Rémi Leblond, Fabian Pedregosa, and Simon Lacoste-Julien. Asaga: asynchronous parallel saga. *arXiv preprint arXiv:1606.04809*, 2016.

[64] Sashank J Reddi, Ahmed Hefny, Suvrit Sra, Barnabas Poczos, and Alexander J Smola. On variance reduction in stochastic gradient descent and its asynchronous variants. *Advances in neural information processing systems*, 28, 2015.

[65] Rémi Leblond, Fabian Pedregosa, and Simon Lacoste-Julien. Asaga: Asynchronous parallel saga. In *Artificial Intelligence and Statistics*, pages 46–54. PMLR, 2017.

[66] Umut Şimşekli, Mert Gürbüzbalaban, Thanh Huy Nguyen, Gaël Richard, and Levent Sagun. On the heavy-tailed theory of stochastic gradient descent for deep neural networks. *arXiv preprint arXiv:1912.00018*, 2019.

[67] Angelia Nedic and Asuman Ozdaglar. Distributed subgradient methods for multi-agent optimization. *IEEE Transactions on Automatic Control*, 54(1):48, 2009.

[68] Jilin Zhang, Hangdi Tu, Yongjian Ren, Jian Wan, Li Zhou, Mingwei Li, and Jue Wang. An adaptive synchronous parallel strategy for distributed machine learning. *IEEE Access*, 6:19222–19230, 2018.

[69] Jeannie R Albrecht, Christopher Tuttle, Alex C Snoeren, and Amin Vahdat. Loose synchronization for large-scale networked systems. In *USENIX Annual Technical Conference, General Track*, pages 301–314, 2006.

[70] Mingwei Li, Jilin Zhang, Jian Wan, Yongjian Ren, Li Zhou, Baofu Wu, Rui Yang, and Jue Wang. Distributed machine learning load balancing strategy in cloud computing services. *Wireless Networks*, 26(8):5517–5533, 2020.

[71] Hang Shi, Yue Zhao, Bofeng Zhang, Kenji Yoshigoe, and Athanasios V Vasilakos. A free stale synchronous parallel strategy for distributed machine learning. In *Proceedings of the 2019 International Conference on Big Data Engineering*, pages 23–29, 2019.

[72] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[73] Joseph E Gonzalez, Peter Bailis, Michael I Jordan, Michael J Franklin, Joseph M Hellerstein, Ali Ghodsi, and Ion Stoica. Asynchronous complex analytics in a distributed dataflow architecture. *arXiv preprint arXiv:1510.07092*, 2015.

[74] Rolf Jagerman and Carsten Eickhoff. Web-scale topic models in spark: An asynchronous parameter server. *arXiv preprint arXiv:1605.07422*, 2016.

[75] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data*, 1(2):49–67, 2015.

[76] Aurick Qiao, Abutalib Aghayev, Weiren Yu, Haoyang Chen, Qirong Ho, Garth A Gibson, and Eric P Xing. Litz: Elastic framework for high-performance distributed machine learning. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 631–644, 2018.

[77] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[78] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[79] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[80] Aitor Lewkowycz and Guy Gur-Ari. On the training dynamics of deep networks with l_2 regularization. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 4790–4799. Curran Associates, Inc., 2020.

[81] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[82] Jerome Friedman, Trevor Hastie, Robert Tibshirani, et al. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[83] David G Kleinbaum, K Dietz, M Gail, Mitchel Klein, and Mitchell Klein. *Logistic regression*. Springer, 2002.

[84] Jiashi Feng, Huan Xu, Shie Mannor, and Shuicheng Yan. Robust logistic regression and classification. *Advances in neural information processing systems*, 27:253–261, 2014.

[85] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[86] L. Bottou. Large-scale machine learning with stochastic gradient descent. *In Proceedings of COMPSTAT'2010*, pages 177–186, Physica-Verlag HD, 2010.

[87] Shiliang Sun, Zehui Cao, Han Zhu, and Jing Zhao. A survey of optimization methods from a machine learning perspective. *IEEE transactions on cybernetics*, 50(8):3668–3681, 2019.

[88] Huan Li, Cong Fang, and Zhouchen Lin. Accelerated first-order optimization algorithms for machine learning. *Proceedings of the IEEE*, 108(11):2067–2082, 2020.

[89] Jimmy Ba, Roger Grosse, and James Martens. Distributed second-order optimization using Kronecker-factored approximations. 2016.

[90] Richard H Byrd, Gillian M Chin, Will Neveitt, and Jorge Nocedal. On the use of stochastic hessian information in optimization methods for machine learning. *SIAM Journal on Optimization*, 21(3):977–995, 2011.

[91] Si Yi Meng, Sharan Vaswani, Issam Hadj Laradji, Mark Schmidt, and Simon Lacoste-Julien. Fast and furious convergence: Stochastic second order methods under interpolation. In *International Conference on Artificial Intelligence and Statistics*, pages 1375–1386. PMLR, 2020.

[92] Christopher M Bishop et al. *Neural networks for pattern recognition*. Oxford university press, 1995.

[93] Yurii Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2003.

[94] Antonio Coppola and Brandon M Stewart. lbfgs: Efficient l-bfgs and owl-qn optimization in r, 2020.

[95] Shun-Ichi Amari. Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276, 1998.

[96] Shun-ichi Amari, Ryo Karakida, and Masafumi Oizumi. Fisher information and natural gradient learning in random deep networks. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 694–702. PMLR, 2019.

[97] Felix Dangel, Frederik Kunstner, and Philipp Hennig. Backpack: Packing more into backprop. In *International Conference on Learning Representations*, 2019.

[98] Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. Fast approximate natural gradient descent in a Kronecker factored eigenbasis. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[99] Roger Grosse and James Martens. A Kronecker-factored approximate fisher matrix for convolution layers. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 573–582, New York, New York, USA, 20–22 Jun 2016. PMLR.

[100] Nicol N Schraudolph. Fast curvature matrix-vector products for second-order gradient descent. *Neural computation*, 14(7):1723–1738, 2002.

[101] Guodong Zhang, James Martens, and Roger B Grosse. Fast convergence of natural gradient descent for over-parameterized neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[102] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel & Distributed Technology: Systems & Applications*, 4(2):63–71, 1996.

[103] Dan C Marinescu. *Cloud computing: theory and practice*. Morgan Kaufmann, 2017.

[104] Mark Schmidt, Alexandru Niculescu-Mizil, Kevin Murphy, et al. Learning graphical model structure using l1-regularization paths. In *AAAI*, volume 7, pages 1278–1283, 2007.

[105] Julien Mairal, Francis Bach, Jean Ponce, and Guillermo Sapiro. Online dictionary learning for sparse coding. In *Proceedings of the 26th annual international conference on machine learning*, pages 689–696. ACM, 2009.

[106] Virginia Smith, Simone Forte, Michael I Jordan, and Martin Jaggi. L1-regularized distributed optimization: A communication-efficient primal-dual framework. *arXiv preprint arXiv:1512.04011*, 2015.

[107] Neal Parikh, Stephen Boyd, et al. Proximal algorithms. *Foundations and Trends® in Optimization*, 1(3):127–239, 2014.

[108] Ingrid Daubechies, Michel Defrise, and Christine De Mol. An iterative thresholding algorithm for linear inverse problems with a sparsity constraint. *Communications on pure and applied mathematics*, 57(11):1413–1457, 2004.

[109] Virginia Smith, Simone Forte, Chenxin Ma, Martin Takac, Michael I Jordan, and Martin Jaggi. Cocoa: A general framework for communication-efficient distributed optimization. *arXiv preprint arXiv:1611.02189*, 2016.

[110] Dhruv Mahajan, S Sathiya Keerthi, and S Sundararajan. A distributed block coordinate descent method for training l 1 regularized linear classifiers. *Journal of Machine Learning Research*, 18(91):1–35, 2017.

[111] Jerome Friedman, Trevor Hastie, and Rob Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of statistical software*, 33(1):1, 2010.

[112] Tyler Johnson and Carlos Guestrin. Blitz: A principled meta-algorithm for scaling sparse optimization. In *International Conference on Machine Learning*, pages 1171–1179, 2015.

[113] Erin Carson, Nicholas Knight, and James Demmel. Avoiding communication in nonsymmetric lanczos-based krylov subspace methods. *SIAM Journal on Scientific Computing*, 35(5):S42–S61, 2013.

[114] Grey Ballard, E Carson, J Demmel, M Hoemmen, Nicholas Knight, and Oded Schwartz. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica*, 23:1–155, 2014.

[115] A.T. Chronopoulos and C.D. Swanson. Parallel iterative s-step methods for unsymmetric linear systems. *Parallel Computing*, 22(5):623 – 641, 1996.

[116] Erin Claire Carson. *Communication-avoiding Krylov subspace methods in theory and practice*. University of California, Berkeley, 2015.

[117] Zeyuan Allen Zhu, Weizhu Chen, Gang Wang, Chenguang Zhu, and Zheng Chen. P-packSVM: Parallel primal gradient descent kernel svm. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 677–686, Washington, DC, USA, 2009. IEEE Computer Society.

[118] Aditya Devarakonda, Kimon Fountoulakis, James Demmel, and Michael W Mahoney. Avoiding synchronization in first-order methods for sparse convex optimization. *arXiv preprint arXiv:1712.06047*, 2017.

[119] Jakub Konečnỳ, Jie Liu, Peter Richtárik, and Martin Takáč. Mini-batch semi-stochastic gradient descent in the proximal setting. *IEEE Journal of Selected Topics in Signal Processing*, 10(2):242–255, 2016.

[120] R. Ge, F. Huang, C. Jin, and Y. Yuan. Escaping from saddle points—online stochastic gradient for tensor decomposition. In *Conference on Learning Theory*, pages 797–842, 2015.

[121] Mingyi Hong and Tsung-Hui Chang. Stochastic proximal gradient consensus over random networks. *IEEE Transactions on Signal Processing*, 65(11):2933–2948, 2017.

[122] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *arXiv preprint arXiv:1606.04838*, 2016.

[123] Atsushi Nitanda. Stochastic proximal gradient descent with acceleration techniques. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 1574–1582. Curran Associates, Inc., 2014.

[124] Lin Xiao and Tong Zhang. A proximal stochastic gradient method with progressive variance reduction. *SIAM Journal on Optimization*, 24(4):2057–2075, 2014.

[125] Jason D Lee, Yuekai Sun, and Michael A Saunders. Proximal newton-type methods for minimizing composite functions. *SIAM Journal on Optimization*, 24(3):1420–1443, 2014.

[126] Antonin Chambolle and Thomas Pock. A first-order primal-dual algorithm for convex problems with applications to imaging. *Journal of mathematical imaging and vision*, 40(1):120–145, 2011.

[127] Dimitri P Bertsekas. Incremental proximal methods for large scale convex optimization. *Mathematical Programming*, 129(2):163–195, 2011.

[128] Tong Tong Wu, Kenneth Lange, et al. Coordinate descent algorithms for lasso penalized regression. *The Annals of Applied Statistics*, 2(1):224–244, 2008.

[129] Samuel H Fuller and Lynette I Millett. Computing performance: Game over or next level? *Computer*, 44(1):31–38, 2011.

[130] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-optimal parallel and sequential qr and lu factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239, 2012.

[131] Atsushi Nitanda. Stochastic proximal gradient descent with acceleration techniques. In *Advances in Neural Information Processing Systems*, pages 1574–1582, 2014.

[132] Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):27, 2011.

[133] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D Peterson, et al. Xsede: accelerating scientific discovery. *Computing in Science & Engineering*, 16(5):62–74, 2014.

[134] Stephen R Becker, Emmanuel J Candès, and Michael C Grant. Templates for convex cone problems with applications to sparse signal recovery. *Mathematical programming computation*, 3(3):165–218, 2011.

[135] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[136] Saeed Soori, Konstantin Mishchenko, Aryan Mokhtari, Maryam Mehri Dehnavi, and Mert Gurbuzbalaban. DAve-QN: A distributed averaged quasi-Newton method with local superlinear convergence rate. In *International Conference on Artificial Intelligence and Statistics*, pages 1965–1976. PMLR, 2020.

[137] Lin Xiao, Adams Wei Yu, Qihang Lin, and Weizhu Chen. Dscovr: Randomized primal-dual block coordinate algorithms for asynchronous distributed optimization. *Journal of Machine Learning Research*, 20(43):1–58, 2019.

[138] Nuri Denizcan Vanli, Mert Gürbüzbalaban, and Asu Ozdaglar. Global convergence rate of proximal incremental aggregated gradient methods. *arXiv preprint arXiv:1608.01713*, 2016.

[139] Mert Gürbüzbalaban, Asuman Ozdaglar, and Pablo A Parrilo. On the convergence rate of incremental aggregated gradient algorithms. *SIAM Journal on Optimization*, 27(2):1035–1048, 2017.

[140] Ashok Cutkosky and Róbert Busa-Fekete. Distributed stochastic optimization via adaptive sgd. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 1910–1919. Curran Associates, Inc., 2018.

[141] Jason D. Lee, Qihang Lin, Tengyu Ma, and Tianbao Yang. Distributed stochastic variance reduced gradient methods by sampling extra data with replacement. *Journal of Machine Learning Research*, 18(122):1–43, 2017.

[142] Hoi-To Wai, Nikolaos M Freris, Angelia Nedic, and Anna Scaglione. Sucag: Stochastic unbiased curvature-aided gradient method for distributed optimization. In *2018 IEEE Conference on Decision and Control (CDC)*, pages 1751–1756. IEEE, 2018.

[143] Hoi-To Wai, Nikolaos M Freris, Angelia Nedic, and Anna Scaglione. Sucag: Stochastic unbiased curvature-aided gradient method for distributed optimization. *arXiv preprint arXiv:1803.08198*, 2018.

[144] Rémi Leblond, Fabian Pedregosa, and Simon Lacoste-Julien. ASAGA: Asynchronous Parallel SAGA. In Aarti Singh and Jerry Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 46–54, Fort Lauderdale, FL, USA, 20–22 Apr 2017. PMLR.

[145] Fabian Pedregosa, Rémi Leblond, and Simon Lacoste-Julien. Breaking the nonsmooth barrier: A scalable parallel method for composite optimization. In *Advances in Neural Information Processing Systems*, pages 56–65, 2017.

[146] Z. Peng, Y. Xu, M. Yan, and W. Yin. Arock: An algorithmic framework for asynchronous parallel coordinate updates. *SIAM Journal on Scientific Computing*, 38(5):A2851–A2879, 2016.

[147] Martin Takáč, Peter Richtárik, and Nathan Srebro. Distributed Mini-Batch SDCA. *arXiv e-prints*, page arXiv:1507.08322, Jul 2015.

[148] Tianbao Yang. Trading computation for communication: Distributed stochastic dual coordinate ascent. In *Advances in Neural Information Processing Systems*, pages 629–637, 2013.

[149] Pascal Bianchi, Walid Hachem, and Franck Iutzeler. A coordinate descent primal-dual algorithm and application to distributed asynchronous optimization. *IEEE Transactions on Automatic Control*, 61(10):2947–2957, 2015.

[150] Alekh Agarwal and John C Duchi. Distributed delayed stochastic optimization. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 873–881. Curran Associates, Inc., 2011.

[151] V. Smith, S. Forte, C. Ma, M. Takac, M. I. Jordan, and M. Jaggi. CoCoA: A General Framework for Communication-Efficient Distributed Optimization. *ArXiv e-prints*, November 2016.

[152] Chenxin Ma, Virginia Smith, Martin Jaggi, Michael Jordan, Peter Richtárik, and Martin Takác. Adding vs. averaging in distributed primal-dual optimization. In *International Conference on Machine Learning*, pages 1973–1982, 2015.

[153] Tianyi Chen, Georgios Giannakis, Tao Sun, and Wotao Yin. Lag: Lazily aggregated gradient for communication-efficient distributed learning. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 5050–5060. Curran Associates, Inc., 2018.

[154] Ruiliang Zhang and James Kwok. Asynchronous distributed ADMM for consensus optimization. In *International Conference on Machine Learning*, pages 1701–1709, 2014.

[155] Mark Eisen, Aryan Mokhtari, and Alejandro Ribeiro. Decentralized quasi-Newton methods. *IEEE Transactions on Signal Processing*, 65(10):2613–2628, 2017.

[156] Ching-pei Lee, Cong Han Lim, and Stephen J Wright. A distributed quasi-Newton algorithm for empirical risk minimization with nonsmooth regularization. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1646–1655. ACM, 2018.

[157] Ohad Shamir, Nati Srebro, and Tong Zhang. Communication-efficient distributed optimization using an approximate Newton-type method. In *International conference on machine learning*, pages 1000–1008, 2014.

[158] Sashank J Reddi, Jakub Konečnỳ, Peter Richtárik, Barnabás Póczós, and Alex Smola. Aide: Fast and communication efficient distributed optimization. *arXiv preprint arXiv:1608.06879*, 2016.

[159] Yuchen Zhang and Xiao Lin. Disco: Distributed optimization for self-concordant empirical loss. In *International Conference on Machine Learning*, pages 362–370, 2015.

[160] S. Wang, F. Roosta-Khorasani, P. Xu, and M. W. Mahoney. GIANT: Globally Improved Approximate Newton Method for Distributed Optimization. *ArXiv e-prints*, September 2017.

[161] Mert Gürbüzbalaban, Asuman Ozdaglar, and Pablo Parrilo. A globally convergent incremental Newton method. *Mathematical Programming*, 151(1):283–313, 2015.

[162] Yossi Arjevani and Ohad Shamir. Communication complexity of distributed convex learning and optimization. In *Advances in Neural Information Processing Systems*, pages 1756–1764, 2015.

[163] Arkadii Nemirovskii, David Borisovich Yudin, and Edgar Ronald Dawson. *Problem complexity and method efficiency in optimization*. Wiley, 1983.

[164] Aryan Mokhtari, Mark Eisen, and Alejandro Ribeiro. IQN: An incremental quasi-Newton method with local superlinear convergence rate. *SIAM Journal on Optimization*, 28(2):1670–1698, 2018.

[165] Nicolas L. Roux, Mark Schmidt, and Francis R. Bach. A stochastic gradient method with an exponential convergence _rate for finite training sets. In *Advances in Neural Information Processing Systems*, pages 2663–2671, 2012.

[166] Aaron Defazio, Francis Bach, and Simon Lacoste-Julien. Saga: A fast incremental gradient method with support for non-strongly convex composite objectives. In *Advances in Neural Information Processing systems*, pages 1646–1654, 2014.

[167] Aaron Defazio, Justin Domke, and Tiberio Caetano. Finito: A faster, permutable incremental gradient method for big data problems. In *Proceedings of the 31st international conference on machine learning (ICML-14)*, pages 1125–1133, 2014.

[168] Julien Mairal. Incremental majorization-minimization optimization with application to large-scale machine learning. *SIAM Journal on Optimization*, 25(2):829–855, 2015.

[169] Aryan Mokhtari, Mert Gürbüzbalaban, and Alejandro Ribeiro. Surpassing gradient descent provably: A cyclic incremental method with linear convergence rate. *SIAM Journal on Optimization*, 28(2):1420–1447, 2018.

[170] N. Denizcan Vanli, Mert Gurbuzbalaban, and Asu Ozdaglar. Global convergence rate of proximal incremental aggregated gradient methods. *SIAM Journal on Optimization*, 28(2):1282–1300, 2018.

[171] Donald Goldfarb. A family of variable-metric methods derived by variational means. *Mathematics of computation*, 24(109):23–26, 1970.

[172] Charles George Broyden, JE Dennis Jr, and Jorge J Moré. On the local and superlinear convergence of quasi-Newton methods. *IMA Journal of Applied Mathematics*, 12(3):223–245, 1973.

[173] John E Dennis and Jorge J Moré. A characterization of superlinear convergence and its application to quasi-Newton methods. *Mathematics of computation*, 28(126):549–560, 1974.

[174] Michael JD Powell. Some global convergence properties of a variable metric algorithm for minimization without exact line searches. *Nonlinear programming*, 9(1):53–72, 1976.

[175] Rixon Crane and Fred Roosta. Dingo: Distributed Newton-type method for gradient-norm optimization. *Advances in Neural Information Processing Systems*, 32:9498–9508, 2019.

[176] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[177] C. G. Broyden, J. E. Dennis Jr., Wang, and J. J. More. On the local and superlinear convergence of quasi-Newton methods. *IMA J. Appl. Math*, 12(3):223–245, June 1973.

[178] Yurii Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2013.

[179] M. J. D. Powell. *Some global convergence properties of a variable metric algorithm for minimization without exact line search.* Academic Press, London, UK, 2 edition, 1971.

[180] Jr. J. E. Dennis and J. J. More. A characterization of super linear convergence and its application to quasi-Newton methods. *Mathematics of computation*, 28(126):549–560, 1974.

[181] Laurent Lessard, Benjamin Recht, and Andrew Packard. Analysis and design of optimization algorithms via integral quadratic constraints. *SIAM Journal on Optimization*, 26(1):57–95, 2016.

[182] Shusen Wang, Fred Roosta, Peng Xu, and Michael W Mahoney. Giant: Globally improved approximate Newton method for distributed optimization. *Advances in Neural Information Processing Systems*, 31:2332–2342, 2018.

[183] Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1):503–528, 1989.

[184] Mark Schmidt, Reza Babanezhad, Mohamed Ahmed, Aaron Defazio, Ann Clifton, and Anoop Sarkar. Non-uniform stochastic average gradient method for training conditional random fields. In *Artificial Intelligence and Statistics*, pages 819–828, 2015.

[185] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in Neural Information Processing Systems 26, Lake Tahoe, Nevada, United States*, pages 315–323, 2013.

[186] Sashank J Reddi, Ahmed Hefny, Suvrit Sra, Barnabas Poczos, and Alexander J Smola. On variance reduction in stochastic gradient descent and its asynchronous variants. In *Advances in Neural Information Processing Systems*, pages 2647–2655, 2015.

[187] Léon Bottou. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer, 2012.

[188] Apache Hadoop. Apache hadoop. *URL http://hadoop. apache. org*, 2011.

[189] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[190] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 561–577, 2018.

[191] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[192] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. Lineage stash: Fault tolerance off the critical path. In *Proceedings of Symposium on Operating Systems Principles, SOSP*, volume 19, 2019.

[193] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. Open mpi: A flexible high performance mpi. In *International Conference on Parallel Processing and Applied Mathematics*, pages 228–239. Springer, 2005.

[194] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Advances in Neural Information Processing Systems*, pages 2737–2745, 2015.

[195] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. Solving the straggler problem with bounded staleness. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, 2013.

[196] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. Staleness-aware async-sgd for distributed deep learning. *arXiv preprint arXiv:1511.05950*, 2015.

[197] Liang Wang, Ben Catterall, and Richard Mortier. Probabilistic synchronous parallel. *arXiv preprint arXiv:1709.07772*, 2017.

[198] Yuewei Ming, Yawei Zhao, Chengkun Wu, Kuan Li, and Jianping Yin. Distributed and asynchronous stochastic gradient descent with variance reduction. *Neurocomputing*, 281:27–36, 2018.

[199] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in Neural Information Processing Systems*, pages 315–323, 2013.

[200] Apache Mahout. Scalable machine-learning and data-mining library. *available at mahout. apache. org*, 2008.

[201] Afaf G Bin Saadon and Hoda MO Mokhtar. iihadoop: an asynchronous distributed framework for incremental iterative computations. *Journal of Big Data*, 4(1):24, 2017.

[202] Luo Mai, Chuntao Hong, and Paolo Costa. Optimizing network performance in distributed machine learning. In *7th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

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

[204] Can Karakus, Yifan Sun, Suhas Diggavi, and Wotao Yin. Straggler mitigation in distributed optimization through data encoding. In *Advances in Neural Information Processing Systems*, pages 5434–5442, 2017.

[205] Ismael Solis Moreno, Peter Garraghan, Paul Townend, and Jie Xu. Analysis, modeling and simulation of workload patterns in a large-scale utility cloud. *IEEE Transactions on Cloud Computing*, 2(2):208–221, 2014.

[206] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *Osdi*, volume 10, page 24, 2010.

[207] Peter Garraghan, Xue Ouyang, Renyu Yang, David McKee, and Jie Xu. Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters. *IEEE Transactions on Services Computing*, 2016.

[208] Xue Ouyang, Peter Garraghan, David McKee, Paul Townend, and Jie Xu. Straggler detection in parallel computing systems through dynamic threshold calculation. In *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, pages 414–421. IEEE, 2016.

[209] J Gregory Pauloski, Qi Huang, Lei Huang, Shivaram Venkataraman, Kyle Chard, Ian Foster, and Zhao Zhang. Kaisa: an adaptive second-order optimizer framework for deep neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[210] Nihat Ay. On the locality of the natural gradient for learning in deep bayesian networks. *Information Geometry*, pages 1–49, 2020.

[211] Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. Fast approximate natural gradient descent in a Kronecker-factored eigenbasis. *arXiv preprint arXiv:1806.03884*, 2018.

[212] Donald Goldfarb, Yi Ren, and Achraf Bahamou. Practical quasi-Newton methods for training deep neural networks. *arXiv preprint arXiv:2006.08877*, 2020.

[213] Yuichiro Ueno, Kazuki Osawa, Yohei Tsuji, Akira Naruse, and Rio Yokota. Rich information is affordable: A systematic performance analysis of second-order optimization using K-FAC. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2145–2153, 2020.

[214] J Gregory Pauloski, Zhao Zhang, Lei Huang, Weijia Xu, and Ian T Foster. Convolutional neural network training with distributed K-FAC. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2020.

[215] Shaohuai Shi, Lin Zhang, and Bo Li. Accelerating distributed K-FAC with smart parallelism of computing and communication tasks. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 550–560. IEEE, 2021.

[216] Minghan Yang, Dong Xu, Zaiwen Wen, Mengyun Chen, and Pengxiang Xu. Sketchy empirical natural gradient methods for deep learning, 2020.

[217] Frederik Kunstner, Lukas Balles, and Philipp Hennig. Limitations of the empirical fisher approximation for natural gradient descent. *arXiv preprint arXiv:1905.12558*, 2019.

[218] Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. Fast approximate natural gradient descent in a Kronecker factored eigenbasis. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[219] Gene H Golub and Charles F Van Loan. *Matrix computations*. JHU press, 2013.

[220] Petros Drineas and Michael W Mahoney. Randnla: randomized numerical linear algebra. *Communications of the ACM*, 59(6):80–90, 2016.

[221] Saurabh Agarwal, Hongyi Wang, Kangwook Lee, Shivaram Venkataraman, and Dimitris Papailiopoulos. Accordion: Adaptive gradient communication via critical learning regime identification. *arXiv preprint arXiv:2010.16248*, 2020.

[222] Donald Goldfarb, Yi Ren, and Achraf Bahamou. Practical quasi-Newton methods for training deep neural networks, 2020.

[223] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[224] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[225] Mateusz Buda. Brain mri segmentation, May 2019.

[226] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[227] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. Mlperf inference benchmark, 2019.

[228] 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.

[229] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks, 2016.

[230] Compute Canada. `www.computecanada.ca`.

[231] Shun-ichi Amari. Neural learning in structured parameter spaces - natural riemannian gradient. In M. C. Mozer, M. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems*, volume 9. MIT Press, 1997.

[232] Tianle Cai. A Gram-Gauss-Newton method learning overparameterized deep neural networks for regression problems. *Machine learning*, (1/28), 2019.

[233] Ryo Karakida and Kazuki Osawa. Understanding approximate Fisher information for fast convergence of natural gradient descent in wide neural networks. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 10891–10901. Curran Associates, Inc., 2020.

[234] Frederik Kunstner, Philipp Hennig, and Lukas Balles. Limitations of the empirical Fisher approximation for natural gradient descent. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[235] Donald Goldfarb, Yi Ren, and Achraf Bahamou. Practical Quasi-Newton methods for training deep neural networks. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 2386–2396. Curran Associates, Inc., 2020.

[236] Tom Heskes. On "natural" learning and pruning in multilayered perceptrons. *Neural Computation*, 12(4):881–901, 2000.

[237] Guillaume Desjardins, Karen Simonyan, Razvan Pascanu, and koray kavukcuoglu. Natural neural networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[238] Sidak Pal Singh and Dan Alistarh. Woodfisher: Efficient second-order approximation for neural network compression. *Advances in Neural Information Processing Systems*, 33, 2020.

[239] Linjian Ma, Gabe Montague, Jiayu Ye, Zhewei Yao, Amir Gholami, Kurt Keutzer, and Michael Mahoney. Inefficiency of K-FAC for large batch size training. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 5053–5060, 2020.

[240] Yinpeng Dong, Renkun Ni, Jianguo Li, Yurong Chen, Jun Zhu, and Hang Su. Learning accurate low-bit deep neural networks with stochastic quantization. *arXiv preprint arXiv:1708.01001*, 2017.

[241] Sungho Shin, Yoonho Boo, and Wonyong Sung. Sqwa: Stochastic quantized weight averaging for improving the generalization capability of low-precision deep neural networks. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 8052–8056. IEEE, 2021.

[242] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. In *International Conference on Learning Representations*, 2018.