Newton-ADMM: A Distributed GPU-Accelerated Optimizer for Multiclass Classification Problems

Chih-Hao Fang
Computer Science
Purdue University
West Lafayette, US
fang150@purdue.edu

Sudhir B. Kylasa
Electrical and Computer Engineering
Purdue University
West Lafayette, US
skylasa@purdue.edu

Fred Roosta
School of Mathematics and Physics
University of Queensland
Brisbane, Australia
fred.roosta@uq.edu.au

Michael W. Mahoney
ICSI and Department of Statistics
UC Berkeley
Berkeley, US
mmahoney@stat.berkeley.edu

Ananth Grama
Computer Science
Purdue University
West Lafayette, US
ayg@cs.purdue.edu

***Abstract*—First-order optimization techniques, such as stochastic gradient descent (SGD) and its variants, are widely used in machine learning applications due to their simplicity and low per-iteration costs. However, they often require larger numbers of iterations, with associated communication costs in distributed environments. In contrast, Newton-type methods, while having higher per-iteration computation costs, typically require a significantly smaller number of iterations, which directly translates to reduced communication costs.**

We present a novel distributed optimizer for classification problems, which integrates a GPU-accelerated Newton-type solver with the global consensus formulation of Alternating Direction of Method Multipliers (ADMM). By leveraging the communication efficiency of ADMM, a highly efficient GPU-accelerated inexact-Newton solver, and an effective spectral penalty parameter selection strategy, we show that our proposed method (i) yields better generalization performance on several classification problems; (ii) significantly outperforms state-of-the-art methods in distributed time to solution; and (iii) offers better scaling on large distributed platforms.

***Index Terms*—Second-Order Method, Newton, ADMM, Convex Optimization, Machine Learning, Classification**

I. Introduction

Estimating parameters of a model from a given dataset is a critical component of a wide variety of machine learning (ML) applications. The parameter estimation problem often translates to one of finding a minima of a suitably formulated objective function. The key challenges in modern "big-data" problems relate to very large numbers of model parameters (which translates to high dimensional optimization problems), large training sets, and learning models with low generalization errors. Recognizing the importance of the problem, a significant amount of research effort has been invested into addressing these challenges.

The most commonly used optimization technique in machine learning is gradient descent and its stochastic variant, stochastic gradient descent (SGD). Gradient descent algorithms, which solely rely on gradient information, are often referred to as first-order methods. Recent results [3, 18, 24] have shown that the use of curvature information in the form of Hessian, or approximations thereof, can lead to significant improvements over SGD in terms of performance as manifest in their convergence rate, time, and quality of solutions.

A key challenge in optimization for machine learning problems is the large, often, distributed nature of the training dataset. It may be infeasible to collect the entire training set at a single node and process it serially because of resource constraints (the training set may be too large for a single node, or that the associated data transfer overhead may be large), privacy (data may be constrained to specific locations), or the need for reducing optimization time. In each of these cases, there is a need for optimization methods that are suitably adapted to parallel and distributed computing environments.

Distributed optimization solvers adopt one of two strategies – (i) executing each operation in conventional solvers (e.g., SGD or (quasi) Newton) in a distributed environment [8, 9, 11, 12, 15, 16, 22, 27, 30]; or (ii) executing an ensemble of local optimization procedures that operate on their own data, with a coordinating procedure that harmonizes the models over iterations [29]. The trade-offs between these two methods are relatively well understood in the context of existing solvers – namely that the communication overhead of methods in the first class is higher, whereas, the convergence rate of the second class of methods is compromised. For this reason, methods in the first class are generally preferred in tightly coupled data-center type environments, whereas methods in the latter class are preferred for wide area deployments. In this

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00 ©2020 IEEE

paper, we present a novel optimization technique that adapts the second strategy, integrating a novel node optimizer with a global consensus strategy.

Alternating Direction Method of Multipliers (ADMM), is a well known method in distributed optimization for solving consensus problems [5]. To achieve superior convergence and efficient solution of the corresponding sub-problems, the choices of penalty parameters associated with global consensus and inner (node-local) sub-problem solver are critical. In particular, the quality of inner sub-problem solutions dictates the accuracy of the descent direction computed by ADMM. In this paper, we present a novel solver that uses the Spectral Penalty Selection (SPS) method in ADMM [29] for setting the penalty parameters and a variant of Newton's method as sub-problem solver. Our choices are motivated by the observation that first-order solvers are known to suffer from slow convergence rates, and are notoriously sensitive to problem ill-conditioning and the choice of hyper-parameters. In contrast, Newton-type methods are less sensitive to such adversarial effects. However, this feature comes with increased per-iteration computation cost. In our solution, we leverage lower iteration counts to minimize communication cost and efficient GPU implementations to address increased computational cost, to engineer our solver that is over an order of magnitude faster on many benchmarks.

*a) **Contributions**:* Our contributions in this paper can be summarized as follows:

- *We propose a novel distributed, GPU-accelerated Newton-type method based on an ADMM framework that has low communication overhead, good per-iteration compute characteristics through effective use of GPU resources, superior convergence properties, and minimal resource overhead.*
- *Using a range of real-world datasets (both sparse and dense), we demonstrate that our proposed method yields significantly better results compared to a variety of state-of-the-art distributed optimization methods.*
- *Our pyTorch implementation is publicly available*[1]*. Our solver can be readily used for practical applications by data scientists and can be easily integrated with other well-known tools like Tensoflow.*

II. Problem Formulation and Algorithm Details

In this section, we describe the optimization problem formulation, and present our proposed Newton-ADMM optimizer.

*a) **Notation**:* We use bold lowercase letters to denote vectors, e.g., $\mathbf{v}$, and bold upper case letters to denote matrices, e.g., $\mathbf{V}$. $\nabla f(\mathbf{x})$ and $\nabla^2 f(\mathbf{x})$ represent the gradient and the Hessian of function f at $\mathbf{x}$, respectively. The superscript, e.g., $\mathbf{x}^{(k)}$, denotes iteration count, and the subscript, e.g., $\mathbf{x}_i$, denotes the *local*-value of the vector $\mathbf{x}$ at the i^{th} compute node in a distributed setting. $\mathcal{D}$ denotes the input dataset, and its cardinality is denoted by $|\mathcal{D}|$. Function $F_i(\mathbf{x})$ represents the objective function, $F(\mathbf{x})$, evaluated at point $\mathbf{x}$ using $i^{th}-$observation. Function $F_{\mathcal{D}}(\mathbf{x})$ represents the objective function evaluated on the entire dataset $\mathcal{D}$.

[1]Newton ADMM solver.

A. Problem Formulation

Consider a finite sum optimization problem of the form:

$$\min_{\mathbf{x}\in\mathbb{R}^d} F(\mathbf{x}) \triangleq \sum_{i=1}^{n} f_i(\mathbf{x}) + g(\mathbf{x}), \tag{1}$$

where each $f_i(\mathbf{x})$ is a smooth **convex** function and $g(\mathbf{x})$ is a (strongly) **convex** and smooth regularizer. In ML applications, $f_i(\mathbf{x})$ can be viewed as loss (or misfit) corresponding to the i^{th} observation (or measurement)[4, 14, 25, 26]. In our study, we choose multi-class classification using soft-max and cross-entropy loss function, as an important instance of finite sum minimization problem. Consider a p dimensional feature vector $\mathbf{a}$, with corresponding labels b, drawn from C classes. In such a classifier, the probability that $\mathbf{a}$ belongs to a class $c \in \{1, 2, \ldots, C\}$ is given by:

$$\mathbf{Pr}\left(b = c \mid \mathbf{a}, \mathbf{x}_1, \ldots, \mathbf{x}_C\right) = \frac{e^{\langle \mathbf{a}, \mathbf{x}_c\rangle}}{\sum_{c'=1}^{C} e^{\langle \mathbf{a}, \mathbf{x}_{c'}\rangle}},$$

where $\mathbf{x}_c \in \mathbb{R}^p$ is the weight vector corresponding to class c. Recall that there are only $C-1$ degrees of freedom, since probabilities must sum to one. Consequently, for training data $\{\mathbf{a}_i, b_i\}_{i=1}^n \subset \mathbb{R}^p \times \{1, 2, \ldots, C\}$, the cross-entropy loss function for $\mathbf{x} = [\mathbf{x}_1; \mathbf{x}_2; \ldots; \mathbf{x}_{C-1}] \in \mathbb{R}^{(C-1)p}$ can be written as:

$$\begin{aligned} &F(\mathbf{x}) \triangleq F(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{C-1}) \\ &= \sum_{i=1}^{n} \left(\log\left(1 + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}_i, \mathbf{x}_{c'}\rangle}\right) - \sum_{c=1}^{C-1} \mathbf{1}(b_i = c)\langle \mathbf{a}_i, \mathbf{x}_c\rangle \right) \qquad (2) \\ &= \sum_{i=1}^{n} \left(M(\mathbf{a}_i) + \log\left(\alpha(\mathbf{a}_i)\right) - \sum_{c=1}^{C-1} \mathbf{1}(b_i = c)\langle \mathbf{a}_i, \mathbf{x}_c\rangle \right), \qquad (3) \end{aligned}$$

where

$$M(\mathbf{a}) = \max\left\{0, \langle \mathbf{a}, \mathbf{x}_1\rangle, \langle \mathbf{a}, \mathbf{x}_2\rangle, \ldots, \langle \mathbf{a}, \mathbf{x}_{C-1}\rangle\right\}, \tag{4}$$

and

$$\alpha(\mathbf{a}) := e^{-M(\mathbf{a})} + \sum_{c'=1}^{C-1} e^{\langle \mathbf{a}, \mathbf{x}_{c'}\rangle - M(\mathbf{a})}. \tag{5}$$

This uses the "Log-Sum-Exp" trick used to avoid over-flow in the evaluation of exponential functions in (2) [20]. After the training phase, a new data instance $\mathbf{a}$ is classified as:

$$b = \arg\max \left\{ \left\{ \frac{e^{\langle \mathbf{a}, \mathbf{x}_c\rangle}}{\sum_{c'=1}^{C-1} e^{\langle \mathbf{a}, \mathbf{x}_{c'}\rangle}} \right\}_{c=1}^{C-1}, 1 - \frac{e^{\langle \mathbf{a}, \mathbf{x}_1\rangle}}{\sum_{c'=1}^{C} e^{\langle \mathbf{a}, \mathbf{x}_{c'}\rangle}} \right\}.$$

B. ADMM Framework

Let $\mathcal{N}$ denote the number of nodes (compute elements) in the distributed environment. Assume that the input dataset $\mathcal{D}$

is split among the $\mathcal{N}$ nodes as $\mathcal{D} = \mathcal{D}_1 \cup \mathcal{D}_2 \ldots \cup \mathcal{D}_{\mathcal{N}}$. Using this notation, (1) can be written as:

$$\min \sum_{i=1}^{\mathcal{N}} \sum_{j \in \mathcal{D}_i} f_j(\mathbf{x}_i) + g(\mathbf{z}) \quad \text{s.t.} \quad \mathbf{x}_i - \mathbf{z} = 0, \quad i = 1, \ldots, \mathcal{N}, \tag{6}$$

where $\mathbf{z}$ represents a global variable enforcing consensus among $\mathbf{x}_i$'s at all the nodes. In other words, the constraint enforces a consensus among the nodes so that all the local variables, $\mathbf{x}_i$, agree with global variable $\mathbf{z}$. The formulation (6) is often referred to as a *global consensus* problem. ADMM is based on an augmented Lagrangian framework; it solves the global consensus problem by alternating iterations on primal/ dual variables. In doing so, it inherits the benefits of the decomposability of dual ascent and the superior convergence properties of the method of multipliers.

ADMM methods introduce a penalty parameter ρ, which is the weight on the measure of *disagreement* between $\mathbf{x}_i$'s and global consensus variable, $\mathbf{z}$. The most common adaptive penalty parameter selection is Residual Balancing [5], which tries to balance the dual norm and residual norm of ADMM. Recent empirical results using SPS [29], which is based on the estimation of the local curvature of subproblem on each node, demonstrate significant improvement in the efficiency of ADMM. Using the SPS strategy for penalty parameter selection, ADMM iterates can be written as follows:

$$\mathbf{x}_i^{k+1} = \arg\min_{\mathbf{x}_i} f_i(\mathbf{x}_i) + \frac{\rho_i^k}{2} ||\mathbf{z}^k - \mathbf{x}_i + \frac{\mathbf{y}_i^k}{\rho_i^k}||_2^2, \tag{7a}$$

$$\mathbf{z}^{k+1} = \arg\min_{\mathbf{z}} g(\mathbf{z}) + \sum_{i=1}^{\mathcal{N}} \frac{\rho_i^k}{2} ||\mathbf{z} - \mathbf{x}_i^{k+1} + \frac{\mathbf{y}_i^k}{\rho_i^k}||_2^2, \tag{7b}$$

$$\mathbf{y}_i^{k+1} = \mathbf{y}_i^k + \rho_i^k (\mathbf{z}^{k+1} - \mathbf{x}_i^{k+1}). \tag{7c}$$

With ℓ_2-regularization, i.e., $g(\mathbf{x}) = \lambda \|\mathbf{x}\|^2/2$, (7b) has a closed-form solution given by

$$\mathbf{z}^{k+1}(\lambda + \sum_{i=1}^{\mathcal{N}} \rho_i^k) = \sum_{i=1}^{\mathcal{N}} [\rho_i^k \mathbf{x}_i^{k+1} - \mathbf{y}_i^k], \tag{8}$$

where λ is the regularization parameter.

Algorithm 1 presents our proposed method incorporating the above formulation of ADMM.

Steps 1-2 initialize the multipliers, $\mathbf{y}$, and consensus vectors, $\mathbf{z}$, to zeros. In each iteration, Single Node Newton method, Algorithm 2, is run with local $\mathbf{x}_i$, $\mathbf{y}_i$, and global $\mathbf{z}$ vectors. Upon termination of Algorithm 2 at all nodes, resulting local Newton directions, $\mathbf{x}_i^k$, are gathered at the master node, which generates the next iterates for vectors $\mathbf{y}$ and $\mathbf{z}$ using spectral step sizes described in [29]. These steps are repeated until convergence. To monitor the convergence of ADMM, we check the norm of primal and dual residuals, $\mathbf{r}_i^k$ and d_i^k, which are defined as follows:

$$\mathbf{r}_i^k = \mathbf{z}^k - \mathbf{x}_i^k, \mathrm{d}_i^k = -\rho_i^k (\mathbf{z}^k - \mathbf{z}^{k-1}) \tag{9}$$

As $k \to \infty$, $\mathbf{z}^k \to \mathbf{z}^*$ and $\forall i, \mathbf{x}_i^k \to \mathbf{z}^*$. Therefore, the norm of primal and dual residuals converge to zero. For a detailed discussion, please see [5].

Algorithm 1: ADMM method (outer solver)

Input : $\mathbf{x}^{(0)}$ (initial iterate), $\mathcal{N}$ (no. of nodes)
Parameters: β, λ and $\theta < 1$
1 Initialize $\mathbf{z}^0$ to 0
2 Initialize $\mathbf{y}_i^0$ to 0 on all nodes.
3 **foreach** $k = 0, 1, 2, \ldots$ **do**
4 Perform Algorithm 2 with, $\mathbf{x}_i^k$, $\mathbf{y}_i^k$, and $\mathbf{z}^k$ on all nodes
5 Collect all local $\mathbf{x}_i^{k+1}$
6 Evaluate $\mathbf{z}^{k+1}$ and $\mathbf{y}_i^{k+1}$ using (7b) and (7c).
8 Distribute $\mathbf{z}^{k+1}$ and $\mathbf{y}_i^{k+1}$ to all nodes.
9 Locally, on each node, compute spectral step sizes and penalty parameters as in [29]
10 **end**

C. Inexact Newton-CG Solver

The optimization problem (1) is decomposed by ADMM framework into sub-problems 7a, 7c, and 7b. Among these sub-problems, only 7a does not have closed form solution. Thus, it is critical to find an iterative method that can produce high quality solutions with low computation cost. To this end, we develop an *inexact Newton-CG solver* for solving sub-problem 7a. Let the objective in Equation 7a be $\hat{f}(\mathbf{x})$, in each iteration; the gradient and Hessian are given by:

$$\mathbf{g}(\mathbf{x}) \triangleq \sum_{j \in \mathcal{D}} \nabla \hat{f}_j(\mathbf{x}), \tag{10a}$$

$$\mathbf{H}(\mathbf{x}) \triangleq \sum_{j \in \mathcal{D}} \nabla^2 \hat{f}_j(\mathbf{x}). \tag{10b}$$

At each iterate $\mathbf{x}^{(k)}$, using the corresponding Hessian, $\mathbf{H}(\mathbf{x}^{(k)})$, and the gradient, $\mathbf{g}(\mathbf{x}^{(k)})$, we consider *inexact* Newton-type iterations of the form:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}_k, \tag{11a}$$

where $\mathbf{p}_k$ is a search direction satisfying:

$$\|\mathbf{H}(\mathbf{x}^{(k)})\mathbf{p}_k + \mathbf{g}(\mathbf{x}^{(k)})\| \leq \theta \|\mathbf{g}(\mathbf{x}^{(k)})\|, \tag{11b}$$

for some inexactness tolerance $0 < \theta < 1$ and α_k being the largest $\alpha \leq 1$ such that the Armijo–Goldstein condition [21] is satisfied.

$$F(\mathbf{x}^{(k)} + \alpha \mathbf{p}_k) \leq F(\mathbf{x}^{(k)}) + \alpha \beta \mathbf{p}_k^T \mathbf{g}(\mathbf{x}^{(k)}), \tag{11c}$$

for some $\beta \in (0, 1)$. This examines whether a step-wise movement from current position $\mathbf{x}^{(k)}$ along direction $\mathbf{p}_k$ to a new position $\mathbf{x}^{(k)} + \alpha \mathbf{p}_k$ achieves a sufficient decrease in the objective function $F(\cdot)$. To estimate the step-size, α_k, in eq. (11a), we use a *backtracking* line search that starts with a relatively large step size and iteratively decreases it by a factor $\gamma < 1$ until the Armijo–Goldstein condition is satisfied.

Condition (11b) is the θ-relative error approximation of the exact solution to the linear system:

$$\mathbf{H}(\mathbf{x}^{(k)})\mathbf{p}_k = -\mathbf{g}(\mathbf{x}^{(k)}), \tag{12}$$

Note that in (strictly) convex settings, where the Hessian matrix is symmetric positive definite (SPD), conjugate gradient (CG) with early stopping can be used to obtain an approximate solution to (12) satisfying (11b). In [24], it has been shown that a mild value for θ, in the order of inverse of *square-root of the condition number*, is sufficient to ensure that the convergence properties of the exact Newton's method are preserved. As a result, for ill-conditioned problems, an approximate solution to (12) using CG yields good performance, comparable to an exact update (see examples in Section III). Putting all of these together, we obtain Algorithm 2, which is known to be globally linearly convergent, with problem-independent local convergence rate [24].

Algorithm 2: Inexact Newton-type Method

Input : $\mathbf{x}^{(0)}$
Parameters: $0 < \beta, \theta < 1$
foreach $k = 0, 1, 2, \ldots$ **do**
  Form $\mathbf{g}(\mathbf{x}^{(k)})$ and $\mathbf{H}(\mathbf{x}^{(k)})$ as in (10)
  if $\|\mathbf{g}(\mathbf{x}^{(k)})\| < \epsilon$ **then**
    STOP
  end
  Update $\mathbf{x}^{(k+1)}$ as in (11)
end

D. GPU-accelerated Newton-type Method

Newton-type methods enjoy superior convergence rates and are less sensitive to ill-conditioned problems. This is largely attributed to their use of curvature information (*e.g.*, in the form of the Hessian matrix). However, the computational cost and memory footprint of each iteration of these methods can be high, if the Hessian matrix is explicitly formulated and solved for ((11b)). To this end, we develop a Hessian-free Newton-type method to solve the ADMM subproblem 7a. Specifically, given a vector $\mathbf{v} \in \mathbb{R}^d$, our goal is to compute the Hessian-vector product $\mathbf{Hv}$ without explicitly forming the Hessian $\mathbf{H}$. Define

$$h(\mathbf{a}, \mathbf{x}) := \frac{e^{\langle \mathbf{a}, \mathbf{x} \rangle - M(\mathbf{x})}}{\alpha(\mathbf{a})},$$

$$\mathbf{V} = \begin{bmatrix} \langle \mathbf{a}_1, \mathbf{v}_1 \rangle & \langle \mathbf{a}_1, \mathbf{v}_2 \rangle & \ldots & \langle \mathbf{a}_1, \mathbf{v}_{C-1} \rangle \\ \langle \mathbf{a}_2, \mathbf{v}_1 \rangle & \langle \mathbf{a}_2, \mathbf{v}_2 \rangle & \ldots & \langle \mathbf{a}_2, \mathbf{v}_{C-1} \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle \mathbf{a}_n, \mathbf{v}_1 \rangle & \langle \mathbf{a}_n, \mathbf{v}_2 \rangle & \ldots & \langle \mathbf{a}_n, \mathbf{v}_{(C-1)} \rangle \end{bmatrix}_{n \times (C-1)}, \tag{13}$$

and

$$\mathbf{W} = \begin{bmatrix} h(\mathbf{a}_1, \mathbf{x}_1) & h(\mathbf{a}_1, \mathbf{x}_2) & \ldots & h(\mathbf{a}_1, \mathbf{x}_{C-1}) \\ h(\mathbf{a}_2, \mathbf{x}_1) & h(\mathbf{a}_2, \mathbf{x}_2) & \ldots & h(\mathbf{a}_2, \mathbf{x}_{C-1}) \\ \vdots & \vdots & \ddots & \vdots \\ h(\mathbf{a}_n, \mathbf{x}_1) & h(\mathbf{a}_n, \mathbf{x}_2) & \ldots & h(\mathbf{a}_n, \mathbf{x}_{C-1}) \end{bmatrix}_{n \times (C-1)}, \tag{14}$$

we compute

$$\mathbf{U} = \mathbf{V} \odot \mathbf{W} - \mathbf{W} \odot \Big(\big((\mathbf{V} \odot \mathbf{W})\, \mathbf{e} \big) \mathbf{e}^T \Big), \tag{15}$$

to get

$$\mathbf{Hv} = \mathrm{vec}\left(\mathbf{A}^T \mathbf{U} \right), \tag{16}$$

where $\mathbf{v} = [\mathbf{v}_1; \mathbf{v}_2; \ldots; \mathbf{v}_{C-1}] \in \mathbb{R}^d$, $\mathbf{v}_i \in \mathbb{R}^p, i = 1, 2, \ldots, C-1$, $\mathbf{e} \in \mathbb{R}^{C-1}$ is a vector of all 1's, and each row of the matrix $\mathbf{A} \in \mathbb{R}^{n \times p}$ is a row vector corresponding to the i^{th} data point, i.e, $\mathbf{A}^T = \big[\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n\big]$. The Hessian-vector product, $\mathbf{Hv}$, can be efficiently computed using GEneral Matrix to Matrix Multiplication (GEMM) operations, for which we rely on Pytorch's Basic Linear Algebra Subprograms (BLAS) interface to the GPUs.

Note that the memory overhead of our GPU-accelerated Newton-type method is determined by the dimensions of the matrices $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$, which are determined by the local dataset size and number of classes in multi-class classification problem at hand. With reasonably sized GPU clusters this memory footprint can be easily managed for large datasets. This enables Newton-type method to scale to large problems inaccessible to traditional second-order methods.

E. Computational Cost and Overheads of Numerical Schemes

The communication cost, computation cost, and number of CPU-GPU transfers in each epoch of our proposed Newton-ADMM scheme and SGD are summarized in Table I.

TABLE I: Complexity measures for the Newton-ADMM and SGD methods. N_s and N_l denote, respectively, the total number of iterations for CG and line search; and C_f, C_g, and C_{Hv} denote, respectively, the computation cost of function evaluation, gradient computation, and Hessian-vector product.

	Newton-ADMM	**SGD**
Number of Communications	1	$\frac{n}{m}$
Computation Cost	$(1 + N_l) \times C_f + C_g + N_s \times C_{Hv}$	$\frac{n}{m} \times (C_f + C_g)$
Number of CPU-GPU Memory Transfers	1	$\frac{n}{m}$

Our Newton-ADMM method has practical advantages over first order methods. In practice, mini-batch SGD is generally preferred over full-batch gradient descent methods. However, SGD often requires a large number of epochs to achieve good generalization errors. Let n and m represent the number of samples and batch size, respectively, with $n >> 1$. SGD requires $\frac{n}{m}$ rounds of communication, with the overhead of $\frac{nd \log(\mathcal{N})}{m\mathcal{N}}$ in each epoch (recall that $\mathcal{N}$ is the number of

workers). On the other hand, in each ADMM iteration only *one* round of communication is required (a "gather" and a "scatter" operation), which can be executed in $\mathcal{O}(\log(\mathcal{N}))$ time.

The mini-batch update scheme results in significantly lower GPU occupancy (idle GPU cores because of smaller batch sizes). The number of CPU-GPU memory transfers per epoch for mini-batch SGD is $\frac{n}{m}$. In contrast, Newton's method utilizes the complete dataset for computing the Newton direction. Therefore, there is only one CPU-GPU memory transfer for computing the Newton direction, which greatly increases utilization of the GPU for reasonably sized datasets.

With respect to the computation cost, the main bottleneck for the Newton-ADMM method is the cost of computing the Hessian-vector product C_{Hv}. However, using carefully formulated Hessian-vector operations, we are able to transform this computation-heavy kernel into an efficient GPU-accelerated operation, as discussed in Section 7. The overall computation overhead of Newton-ADMM is, therefore, lower than SGD. Specifically, given that C_g is approximately $2 \times C_f$ [7], the overall computation costs for Newton-ADMM and SGD, are $(3 + N_l) \times C_f + N_s \times C_{Hv}$ and $\frac{3n}{m} \times C_f$, respectively. Since $n >> 1$, the number of CG iterations, N_s, and the number of line search iterations, N_l, are far less than n, and the computation time of Hessian-vector product C_{HV} is lower when using GPUs. The gains from using Newton's method with ADMM, along with low communication and high GPU utilization, as compared to first order methods, are demonstrated in our experimental results.

III. Experimental Evaluation

In this section, we evaluate the performance of Newton-ADMM as compared with several state-of-the-art alternatives.

TABLE II: Description of the datasets.

Classes	Dataset	Train Size	Test Size	Dims
2	HIGGS	10,000,000	1,000,000	28
10	MNIST	60,000	10,000	784
10	CIFAR-10	50,000	10,000	3,072
20	E18	1,300,128	6,000	279,998

*a) **Experimental Setup and Data:**.* All algorithms are implemented in PyTorch/0.3.0.post4 with Message Passing Interface (MPI) backend. We test performance of the methods on two hardware platforms. The first platform is a server with Intel Xeon Platinum 8168 processors and 8 Tesla P100 GPU cards. The second platform is a CentOS 7 cluster with 15 nodes with 100 Gbps Infiniband interconnect. Each node has 96GB RAM, two 12-Core Intel Xeon Gold processors, and 3 Tesla P100 GPU cards. We validate our proposed method using real-world datasets, described in Table II, and compare with state-of-the-art first-order and second-order optimizers. These datasets are chosen to cover a wide range of problem characteristics (problem-conditioning, features, problem-size). MNIST is a widely used dataset for validation – it is relatively well-conditioned. CIFAR-10 is 3.9x larger than MNIST and is relatively ill-conditioned. HIGGS[1] is a low-dimensional dataset, however it is the largest (in terms of problem size). This dataset is easy to solve for our method, but is harder for first-order variants because of high communication overhead. The largest data set, E18 [2] , in terms of dimension and number of samples, is used to highlight the scalability of our proposed method.

*b) **Algorithms Parameter Settings:**.* We generated all the experiment results using the following settings:

- SGD : we tune the step size from 10^{-4} to 10^4 and select the best result to report.
- Newton-ADMM : We used 10 CG iterations along with 10^{-4} CG tolerance to compute Newton direction at each compute node. The step size was chosen by line search with 10 iterations.
- GIANT : The configurations for CG and linesearch are the same as the configurations use in Newton-ADMM.
- Inexact DANE : we use learning rate $\eta = 1.0$ and regularization term $\mu = 0.0$ for solving subproblems as prescribed in [11]. We set SVRG iterations to 100 and update frequency as $2n$, where n is the number of local sample points. We run SVRG step size from the set 10^{-4} to 10^4 in increments of 10 and select the best value to report.
- AIDE : The configurations for SVRG is the same as the configurations used in Inexact DANE. As to the additional hyper-parameter introduced in AIDE, τ, we also run τ from the set 10^{-4} to 10^4 and select the best to report.
- Regularization parameter λ : we used $\lambda = 10^{-5}$ for all the experiments.

*c) **Newton-type method as a highly efficient subproblem solver for ADMM.**:* We establish our (GPU-accelerated) Newton-type optimizer as a highly efficient inner solver for ADMM by comparing its performance against an ADMM-L-BFGS solver. The per-iteration computation cost and memory footprint of L-BFGS is lower than our Newton-type method because of the rank-2 approximation of the Hessian matrix. This, however, comes at the cost of a worse convergence rate for L-BFGS. While Newton-type methods compute matrix vector products with the full Hessian, we use a Conjugate Gradient method with early stopping to solve the linear system, $\mathbf{Hx} = -\mathbf{g}$. In our experiments we use no more than 10 CG iterations and a tolerance level of 10^{-3}. The resulting *Inexact* Newton-type method is GPU-accelerated, with an efficient implementation of Hessian-vector product. We show that, in practice, ADMM method suitably aided by efficient implementation of Newton-type subproblem solvers, yields significantly better results compared to the state-of-the-art. Furthermore, the use of true Hessian in our inexact solver, a second-order method, makes it resilient to problem ill-conditioning and immune to hyper-parameter tuning. These results are shown in Figure 1. We clearly notice that the performance gap between L-BFGS and Inexact Newton-type method becomes larger when number of compute nodes is increased. The only exception is on HIGGS dataset. This is because the dimension of the HIGGS datasets is only 28 and it is a binary classification problem. Consequently, the dimension

[2] E18 data set source.

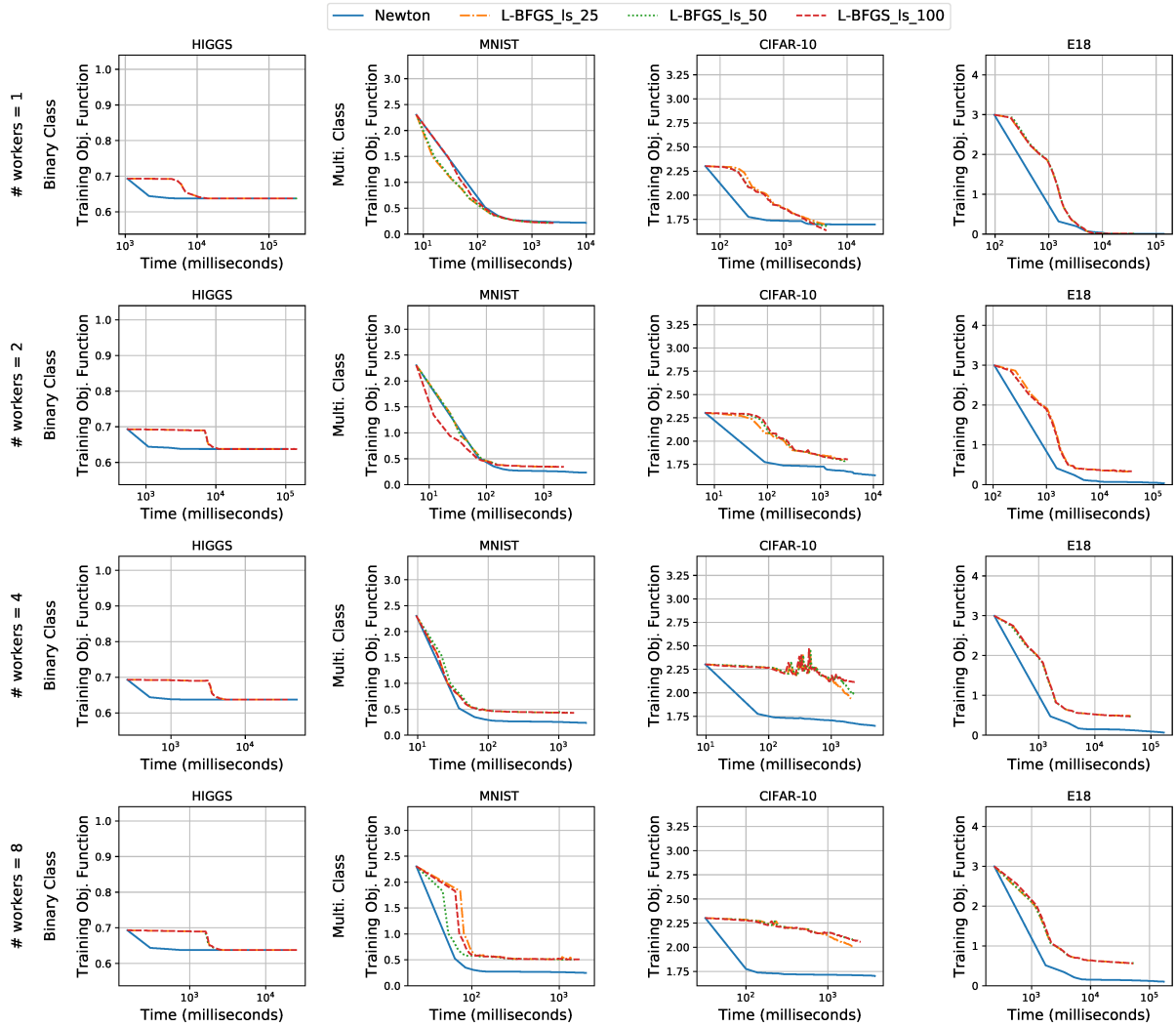

Fig. 1: Training Objective function comparison over time for different choice of inner-solve for ADMM. For the inner solver, we compare the performance of Inexact Newton solver with L-BFGS (with history size 25, 50, 100). The step size of Inexact Newton method is chosen by linesearch following Armijo rule, whereas the step size of L-BFGS is chosen by linesearch satisfying Strong Wolfe condition. We can see that the per-iteration computation cost of L-BFGS is lower than Inexact Newton with the exception on HIGGS dataset. This is because L-BFGS is sensitive to the scale of step size so that more iterations of Strong Wolfe linesearch procedure are required to satisfy the curvature condition. In general, we observe that L-BFGS performs well on binary class problems, while the performance degrades on multiclass problems, when the number of compute nodes increases.

of Hessian is significantly lower than among all the datasets. In this case, L-BFGS solver yields similar results compared to our Inexact Newton method. Most importantly, we note the following key results: (i) Inexact Newton yields performance improvements from 0 (MNIST) to 550% (HIGGS and CIFAR-10) over L-BFGS on a single node; (ii) when using 8 compute nodes, the performance of L-BFGS-ADMM *never* catches up with that of Newton-ADMM (in terms of training objective function) in three of four benchmarks (MNIST, CIFAR-10, and E18), conclusively establishing the superiority of our proposed method over L-BFGS-ADMM.

*d) **Comparison with Distributed First-order Methods.**:* While the per-iteration cost of first-order methods is relatively low, they require larger numbers of iterations, increasing associated communication overhead, and CPU-GPU transactions, if GPUs are used (Please see detailed discussion in section 7). In this experiment, we demonstrate that these drawbacks of first order methods are significant. We compare the performance of Newton-ADMM and SGD in terms of CPU time, GPU time, and communication time in both strong scaling and weak scaling cases, in Tables IIIa and IIIb, respectively. In strong scaling experiments, we keep the number of training samples constant, while increasing the number of workers, and

TABLE III: Table IIIa and Table IIIb compare the CPU time (ms), GPU time (ms), and communication time (ms) per epoch on strong scaling and weak scaling cases for Newton-ADMM and SGD algorithms.

(a) Table IIIa presents the CPU time, GPU time, and communication time per epoch for Newton-ADMM and SGD algorithms on MNIST, CIFAR-10, and HIGGS datasets for strong-scaling cases.

	S1					**S2**			
MNIST	**CPU Time**	**GPU Time**	**Comm. Time**	**Train. Obj.**	**MNIST**	**CPU Time**	**GPU Time**	**Comm. Time**	**Train. Obj.**
Newton-ADMM	3676.00	66.90	0.081	0.22	Newton-ADMM	1462.36	36.54	0.28	0.23
SGD	461.62	1034.63	18.97	0.24	SGD	229.16	530.92	23.84	0.23
CIFAR-10					**CIFAR-10**				
Newton-ADMM	11419.11	176.64	0.59	1.63	Newton-ADMM	5321.67	99.28	0.85	1.63
SGD	883.16	1280.91	43.76	1.67	SGD	443.71	663.99	54.28	1.65
HIGGS					**HIGGS**				
Newton-ADMM	23098.02	2159.53	0.08	0.64	Newton-ADMM	12356.39	1107.97	0.24	0.64
SGD	36791.37	125935.64	1739.15	0.65	SGD	18305.36	61597.04	1237.57	0.65
	S4					**S8**			
MNIST	**CPU Time**	**GPU Time**	**Comm. Time**	**Train. Obj.**	**MNIST**	**CPU Time**	**GPU Time**	**Comm. Time**	**Train. Obj.**
Newton-ADMM	787.38	26.29	0.39	0.24	Newton-ADMM	260.71	18.19	0.72	0.24
SGD	115.88	26751	39.31	0.26	SGD	61.63	138.30	28.82	0.28
CIFAR-10					**CIFAR-10**				
Newton-ADMM	2482.32	48.48	1.480	1.66	Newton-ADMM	1227.98	33.20	2.67	1.68
SGD	217.714	331.17	88.62	1.65	SGD	109.68	168.61	60.80	1.67
HIGGS					**HIGGS**				
Newton-ADMM	3784.05	443.10	0.22	0.64	Newton-ADMM	1581.05	215.86	0.12	0.64
SGD	9018.30	29904.36	1523.00	0.65	SGD	4629.18	16157.91	666.69	0.65

(b) Table IIIb presents the CPU time, GPU time, and communication time per epoch for Newton-ADMM and SGD algorithms on MNIST, CIFAR-10, HIGGS, and E18 datasets for weak-scaling cases.

	W1					**W2**			
MNIST	**CPU Time**	**GPU Time**	**Comm. Time**	**Train. Obj.**	**MNIST**	**CPU Time**	**GPU Time**	**Comm. Time**	**Train. Obj.**
Newton-ADMM	322.15	20.20	0.25	0.06	Newton-ADMM	310.03	17.85	0.31	0.178
SGD	60.58	133.94	8.67	0.09	SGD	61.03	152.75	10.75	0.16
CIFAR-10					**CIFAR-10**				
Newton-ADMM	1268.43	33.80	0.63	1.29	Newton-ADMM	1223.65	33.12	0.941	1.59
SGD	106.06	173.09	15.87	1.40	SGD	110.52	169.17	22.99	1.55
HIGGS					**HIGGS**				
Newton-ADMM	1612.71	226.69	0.07	0.64	Newton-ADMM	1740.02	202.35	0.07	0.64
SGD	4484.31	16169.06	220.46	0.65	SGD	4539.78	16437.12	297.99	0.65
E18					**E18**				
Newton-ADMM	60644.96	907.57	15.917	0.006	**Newton-ADMM**	84793.51	1003.51	19.66	0.007
SGD	10534.14	8084.51	1723.16	0.03	**SGD**	11433.75	8101.65	2366.56	0.058
	W4					**W8**			
MNIST	**CPU Time**	**GPU Time**	**Comm. Time**	**Train. Obj.**	**MNIST**	**CPU Time**	**GPU Time**	**Comm. Time**	**Train. Obj.**
Newton-ADMM	326.67	18.64	0.44	0.19	Newton-ADMM	260.71	18.19	0.72	0.24
SGD	57.63	142.86	15.39	0.20	SGD	61.63	138.30	28.82	0.28
CIFAR-10					**CIFAR-10**				
Newton-ADMM	1251.88	33.17	1.86	1.65	Newton-ADMM	1227.98	33.20	2.67	1.68
SGD	110.59	171.48	34.53	1.63	SGD	109.68	168.61	60.80	1.67
HIGGS					**HIGGS**				
Newton-ADMM	1444.01	212.38	0.08	0.64	Newton-ADMM	1581.05	215.86	0.12	0.64
SGD	4574.42	16272.24	445.67	0.65	SGD	4629.18	16157.91	666.69	0.65
E18					**E18**				
Newton-ADMM	74356.35	1015.58	55.03	0.05	Newton-ADMM	79368.43	1003.86	94.43	0.09
SGD	9442.47	6195.97	6317.41	0.08	SGD	9558.39	5882.13	6611.72	0.10

for weak scaling, the number of the training samples per node is kept constant. We observe that both average CPU time and GPU time for each epoch reduce by roughly half when the number of nodes double in strong scaling cases. The average CPU time and GPU time remain roughly the same when the number of nodes double in weak scaling cases. In both strong scaling and weak scaling cases, we observe that SGD has significantly higher communication overhead compared to Newton-ADMM for all datasets. In all cases, Newton-ADMM outperforms SGD, and is able to use GPU resources efficiently. In contrast, SGD achieves meaningful speedup from GPUs only for the largest dataset, E18.

*e) **Comparison with Distributed Second-order Methods.**:* We compare Newton-ADMM against DANE [11], AIDE [22], and GIANT [27], which have been shown in recent results to perform well. In each iteration, DANE [11] requires an exact solution of its corresponding subproblem at each node. This constraint is relaxed in an inexact version of DANE, called InexactDANE [22], which uses SVRG [17] to approximately solve the subproblems. Another version of DANE, called Accelerated Inexact DanE (AIDE), proposes techniques for accelerating convergence, while still using InexactDANE to solve individual subproblems [22]. However, using SVRG to solve subproblems is computationally inefficient due to its double loop formulation, with the outer loop requiring full gra-

dient recalculation and several stochastic gradient calculations in inner loop.

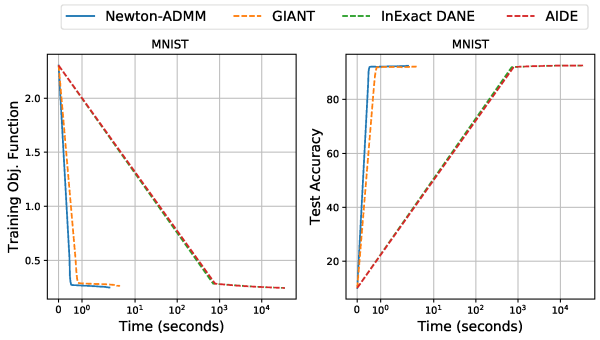

Fig. 2: Training objective function and test accuracy comparison over time for Newton-ADMM, GIANT, InexactDANE, and AIDE on MNIST dataset with $\lambda = 10^{-5}$. We run both Newton-ADMM and GIANT for 100 epochs. Since the computation times per epoch for InexactDANE and AIDE are high, we only run 10 epochs for these methods. We present details of hyperparameter settings in III-0b.

Figure 2 shows the comparison between these methods on the MNIST dataset. Although InexactDANE and AIDE start at lower objective function values, the average epoch time compared to Newton-ADMM and GIANT is orders of magnitude higher (*order of 1000x*). For instance, to reach an objective function value less than 0.25 on the MNIST dataset, Newton-ADMM takes only 2.4 seconds, whereas InexactDANE consumes *an hour and a half*. Since InexactDANE and AIDE are significantly slower than Newton-ADMM and GIANT (on other datasets as well – for which we do not show results here), we restrict our discussion of results on performance and scalability to Newton-ADMM and GIANT in the rest of this section.

f) ***Scalability of Newton-ADMM.:*** Figure 3 presents strong- and weak-scaling results for Newton-ADMM and GIANT. For strong-scaling, as number of workers increases, average epoch time for both Newton-ADMM and GIANT decreases. For all the datasets, as the number of workers is doubled, the average epoch time halved for both methods. For weak scaling, as the number of workers doubles, the average epoch time nearly remains constant for both methods. Both Newton-ADMM and GIANT use CG to compute Newton directions. However, compared to GIANT, Newton-ADMM has lower epoch times for the following reasons: first, to guarantee global convergence on non-quadratic problems, GIANT uses a globalization strategy based on line search. For this, the i-th worker computes the local objective function values $f_{\mathcal{D}_i}(\mathbf{x}_i + \alpha \mathbf{p})$ for all α's in a pre-defined set of step-sizes $S = \{2^0, 2^{-1}, ..., 2^{-k}\}$, where k is the maximum number of line search iterations. Thus, for each epoch, all workers need to compute a fixed number of objective function values. In contrast, Newton-ADMM performs line search only locally, allowing each worker to terminate line search before reaching the maximum number of line search iterations, and hence reducing the overhead of redundant computations. Second, Newton-ADMM only requires one round of messages per iteration, whereas GIANT needs three. Our experiments are performed on a Gigabit-interconnet cluster, where communication fabric is highly optimized. However, in environments with lower bandwidth and higher latency, we expect Newton-ADMM to perform significantly better compared to GIANT.

TABLE IV: Performance comparison of Newton-ADMM and GIANT – we present the number of epochs for a solver to reach $\theta < 0.05$. The speedup ratio is defined as the fraction of time taken by GIANT to achieve a specified value of θ to the corresponding time taken by Newton-ADMM on the same hardware platform.

	NT-ADMM Epochs	**GIANT Epochs**	**Speedup**
MNIST	252	1086	5.15
CIFAR-10	1204	3215	11.14
HIGGS	1	1	1.35

We now compare the convergence of Newton-ADMM with GIANT in a distributed setting. Instead of comparing the test accuracy or objective value over time, we compare how close the objective value obtained from the solver is to the optimal objective value. Specifically, define $\theta = (F(\mathbf{x}^k) - F(\mathbf{x}^*))/F(\mathbf{x}^*)$, we measure θ as a function of time. (Here, $F(.)$ denotes the objective function, $\mathbf{x}^k$ is the approximate solution obtained by the solver at the k-th iterate, and the "optimal" solution vector $\mathbf{x}^*$ is obtained by running Newton's method on a single node to high precision). Figure 4 shows θ, in log scale, as a function of time for MNIST, CIFAR-10, and HIGGS using 8 compute nodes. From Figure 4, we observe that, given the same amount of time, Newton-ADMM can reach lower θ in each case. We also measure the number of epochs taken by the solver to reach $\theta < 0.05$. Table IV shows the number of epochs for Newton-ADMM and GIANT to reach $\theta < 0.05$ on 8 nodes.

From Table IV, we can see that Newton-ADMM converges to optimal solution significantly faster than GIANT. Specifically, to reach $\theta \leq 0.05$, for the MNIST dataset, Newton-ADMM takes 252 epochs while GIANT takes 1086 epochs. For the CIFAR-10 dataset, Newton-ADMM takes 1204 epochs while GIANT takes 3215 epochs. The speed up ratio on MNIST and CIFAR-10 is 5.15 and 11.14, respectively. Both Newton-ADMM and GIANT behave well on HIGGS. It only takes 1 epoch for both solvers to reach $\theta \leq 0.05$. We note that the superior performance of these methods on HIGGS does not carry over to first-order methods.

Finally, we stress that Newton-ADMM scales well on large data sets and produces more stable solutions, in terms of test accuracy, in large-scale distributed environments. We measure the objective function and test accuracy as functions of time for both Newton-ADMM and GIANT in weak scaling scenarios with 16, 24, and 32 nodes (Figure 5). We note that Newton-ADMM takes significantly less time to achieve lower objective function values for all cases. In addition, we observe that

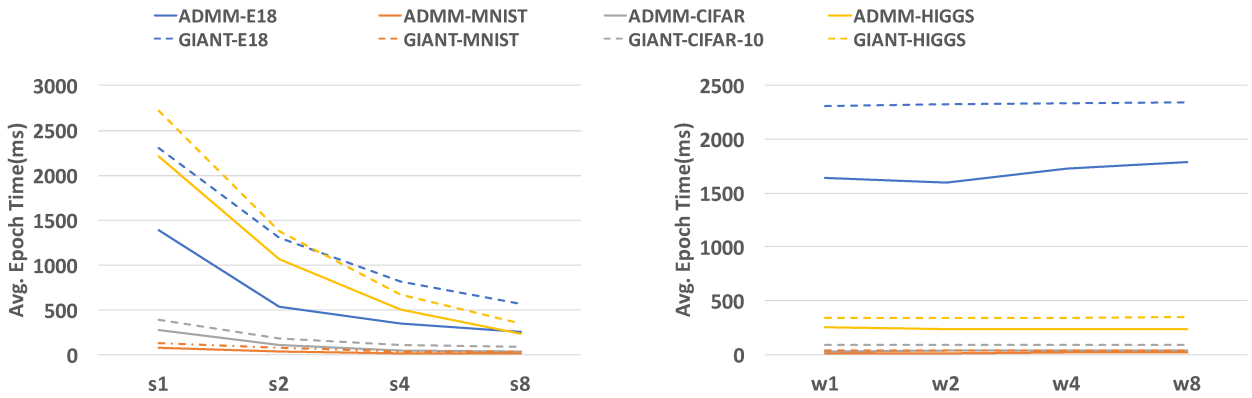

Fig. 3: Avg. Epoch Time for Strong and Weak Scaling for Newton-ADMM and GIANT.

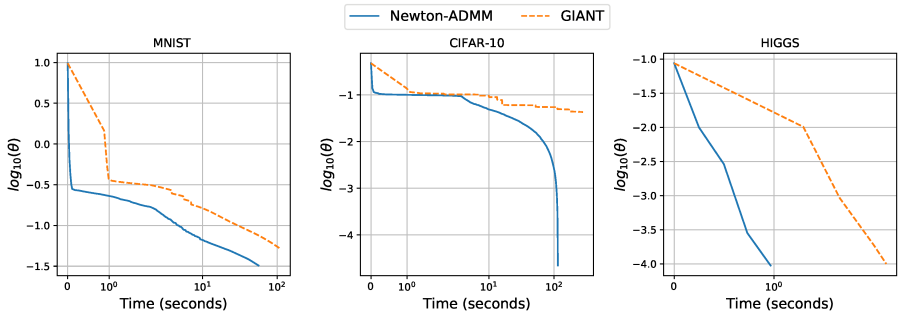

Fig. 4: Convergence performance of Newton-ADMM. Figure 4 shows $\log_{10}(\theta)$ as a function of time for Newton-ADMM and GIANT on MNIST, CIFAR-10, and HIGGS datasets. Newton-ADMM can reach lower θ, given the same amount of time, compared to GIANT. Note that for the HIGGS dataset, both methods can reach low θ soon.

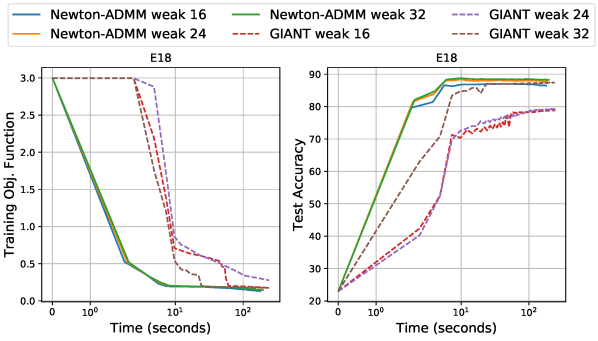

Fig. 5: Scalability of Newton-ADMM on largest data set (E18). Figure 5 shows training objective function and test accuracy as function of time for Newton-ADMM and GIANT on E18 dataset in weak scaling with 16, 24, and 32 compute nodes. We observe that Newton-ADMM achieves higher test accuracies faster than GIANT in all of the cases.

Newton-ADMM achieves higher test accuracy even in weak scaling scenario with 16 nodes, while GIANT requires twice the size of training set (32 nodes) to achieve comparable test accuracy as Newton-ADMM. Finally, the high dimensionality of E18 (280K) highlights the memory- and compute-efficient formulation of our Hessian-vector products and subproblem solves on GPUs – the average epoch time for the E18 dataset on 32 nodes is only 1.98 seconds!

IV. Related Research

First-order methods [2, 6] – gradient descent and its variants are commonly used in ML applications. This is mainly because these methods are simple to implement and have low per-iteration costs. However, it is known that these methods often take a large number of iterations to achieve reasonable generalization, primarily due to their sensitivity to problem ill-conditioning. Second-order methods make use of curvature information, in the form the Hessian matrix, and as a result are more robust to problem ill-conditioning [24], and hyper-parameter tuning [3, 28]. However, they can have higher memory and computation requirements due to the application of the Hessian matrix. In this context, quasi-Newton methods [21] can be used to approximate the Hessian by using the history of gradients. However, a history of gradients must be stored in order to approximate the Hessian matrix, and extra computation cost is incurred to satisfy the strong Wolfe condition. In addition, these methods are observed to be unstable when used on mini-batches [18].

Several distributed solvers have been developed recently [8, 9, 11, 12, 15, 16, 22, 27, 30]. Among these, [8, 12, 15,

16] are classified as first-order methods. Although they incur low computational overhead, they have higher communication costs due to a large number of messages exchanged per mini-batch and high total iteration counts. Second-order variants [9, 11, 22, 27, 30] are designed to improve convergence rate, as well as to reduce communication costs. DANE [11], and the accelerated scheme AIDE [22] use SVRG [17] as the subproblem solver to approximate the Newton direction. These methods are often sensitive to the fine-tuning of SVRG. DiSCO [30] uses distributed preconditioned conjugate gradient (PCG) to approximate the Newton direction. The number of communications across nodes per PCG call is proportional to the number of PCG iterations. In contrast to DiSCO, GIANT [27] executes CG at each node and approximates the Newton direction by averaging the solution from each CG call. Empirical results have shown that GIANT outperforms DANE, AIDE, and DiSCO. The solver of Dunner et al. [13] is shown to outperform GIANT, however, it is constrained to sparse datasets. A recently developed variant, DINGO [9], can be applied to a class of non-convex functions, namely invex [10], which includes convexity as a special sub-class. However, in the absence of invexity, the method can converge to undesirable stationary points.

A popular choice in distributed settings is ADMM [5], which combines dual ascent method and the method of multipliers. ADMM only requires one round of communication per iteration. However, ADMM's performance is greatly affected by the selection of the penalty parameter [29] as well as the choice of local subproblem solvers.

V. Conclusions

We have developed a novel distributed Inexact Newton method based on a global consensus ADMM formulation. We compare our method with state-of-the-art optimization methods and show that our method has much lower distributed computing overhead, achieves superior generalization errors, and has significantly lower epoch-times on standard benchmarks. We have also shown that our method can handle large datasets, while delivering sub-second epoch times – establishing desirable scalability characteristics of our method. Our work can be effectively extended to non-convex problems arising from deep neural networks by incorporating serial non-convex solvers into our distributed framework[19, 23].

VI. ACKNOWLEDGMENTS

The authors thank Dr. Amiya K Maji and Mr. Charles Donald Fultz for their help in configuring the computing environments. This work is supported by NSF Grants OAC 1908691 and CCF 0939370 for establishing the Center for Science of Information, and by the DARPA Assured Autonomy program.

References

[1] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. "Searching for exotic particles in high-energy physics with deep learning". In: *Nature communications* 5 (2014), p. 4308.

[2] Amir Beck. *First-Order Methods in Optimization*. Vol. 25. SIAM, 2017.

[3] Albert S Berahas, Raghu Bollapragada, and Jorge Nocedal. "An Investigation of Newton-Sketch and Subsampled Newton Methods". In: *arXiv preprint arXiv:1705.06211* (2017).

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

[5] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, Jonathan Eckstein, et al. "Distributed optimization and statistical learning via the alternating direction method of multipliers". In: *Foundations and Trends® in Machine learning* 3.1 (2011), pp. 1–122.

[6] Sébastien Bubeck et al. "Convex optimization: Algorithms and complexity". In: *Foundations and Trends® in Machine Learning* 8.3-4 (2015), pp. 231–357.

[7] Yves Chauvin and David E Rumelhart. *Backpropagation: theory, architectures, and applications*. Psychology press, 2013.

[8] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. "Revisiting distributed synchronous SGD". In: *arXiv preprint arXiv:1604.00981* (2016).

[9] Rixon Crane and Fred Roosta. "DINGO: Distributed Newton-Type Method for Gradient-Norm Optimization". In: *Proceedings of the Advances in Neural Information Processing Systems*. Accepted. 2019.

[10] BD Craven. "Invex functions and constrained local minima". In: *Bulletin of the Australian Mathematical society* 24.03 (1981), pp. 357–366.

[11] Hadi Daneshmand, Aurelien Lucchi, and Thomas Hofmann. "DynaNewton-Accelerating Newton's Method for Machine Learning". In: *arXiv preprint arXiv:1605.06561* (2016).

[12] Jeffrey Dean et al. "Large scale distributed deep networks". In: *Advances in neural information processing systems*. 2012, pp. 1223–1231.

[13] Celestine Dünner, Aurelien Lucchi, Matilde Gargiani, An Bian, Thomas Hofmann, and Martin Jaggi. "A Distributed Second-Order Algorithm You Can Trust". In: *arXiv preprint arXiv:1806.07569* (2018).

[14] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*. Vol. 1. Springer series in statistics Springer, Berlin, 2001.

[15] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. "Accurate, large minibatch SGD: training imagenet in 1 hour". In: *arXiv preprint arXiv:1706.02677* (2017).

[16] Peter H Jin, Qiaochu Yuan, Forrest Iandola, and Kurt Keutzer. "How to scale distributed deep learning?" In: *arXiv preprint arXiv:1611.04581* (2016).

[17] Rie Johnson and Tong Zhang. "Accelerating stochastic gradient descent using predictive variance reduction".

In: *Advances in neural information processing systems*. 2013, pp. 315–323.

[18] Sudhir Kylasa, Farbod Roosta-Khorasani, Michael W Mahoney, and Ananth Grama. "GPU Accelerated Sub-Sampled Newton's Method for Convex Classification Problems". In: *Proceedings of the 2019 SIAM International Conference on Data Mining*. SIAM. 2019, pp. 702–710.

[19] Sudhir B Kylasa. "HIGHER ORDER OPTIMIZATION TECHNIQUES FOR MACHINE LEARNING". PhD thesis. Purdue University Graduate School, 2019.

[20] Kevin P Murphy. *Machine learning: a probabilistic perspective*. The MIT Press, 2012.

[21] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[22] Sashank J Reddi, Jakub Konečnỳ, Peter Richtárik, Barnabás Póczós, and Alex Smola. "AIDE: Fast and communication efficient distributed optimization". In: *arXiv preprint arXiv:1608.06879* (2016).

[23] Fred Roosta, Yang Liu, Peng Xu, and Michael W Mahoney. "Newton-MR: Newton's Method Without Smoothness or Convexity". In: *arXiv preprint arXiv:1810.00303* (2018).

[24] Farbod Roosta-Khorasani and Michael W Mahoney. "Sub-sampled Newton methods". In: *Mathematical Programming* 174.1-2 (2019), pp. 293–326.

[25] Suvrit Sra, Sebastian Nowozin, and Stephen J Wright. *Optimization for machine learning*. Mit Press, 2012.

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

[27] Shusen Wang, Farbod Roosta-Khorasani, Peng Xu, and Michael W Mahoney. "GIANT: Globally Improved Approximate Newton Method for Distributed Optimization". In: *Advances in Neural Information Processing Systems (NIPS)*. 2018, pp. 2338–2348.

[28] Peng Xu, Farbod Roosta-Khorasani, and Michael W. Mahoney. "Second-Order Optimization for Non-Convex Machine Learning: An Empirical Study". In: *arXiv preprint arXiv:1708.07827* (2017).

[29] Zheng Xu, Gavin Taylor, Hao Li, Mario Figueiredo, Xiaoming Yuan, and Tom Goldstein. "Adaptive consensus ADMM for distributed optimization". In: *arXiv preprint arXiv:1706.02869* (2017).

[30] Yuchen Zhang and Xiao Lin. "DiSCO: Distributed optimization for self-concordant empirical loss". In: *International conference on machine learning*. 2015, pp. 362–370.