Distributed Newton Methods for Deep Neural Networks

Chien-Chih Wang,[1] **Kent Loong Tan,**[1] **Chun-Ting Chen,**[1]

Yu-Hsiang Lin,[2] **S. Sathiya Keerthi,**[3] **Dhruv Mahajan,**[4]

S. Sundararajan,[3] **Chih-Jen Lin**[1]

[1]Department of Computer Science, National Taiwan University, Taipei 10617, Taiwan

[2]Department of Physics, National Taiwan University, Taipei 10617, Taiwan

[3]Microsoft

[4]Facebook Research

Abstract

Deep learning involves a difficult non-convex optimization problem with a large number of weights between any two adjacent layers of a deep structure. To handle large data sets or complicated networks, distributed training is needed, but the calculation of function, gradient, and Hessian is expensive. In particular, the communication and the synchronization cost may become a bottleneck. In this paper, we focus on situations where the model is distributedly stored, and propose a novel distributed Newton method for training deep neural networks. By variable and feature-wise data partitions, and some careful designs, we are able to explicitly use the Jacobian matrix for matrix-vector products in the Newton method. Some techniques are incorporated to reduce the running time as well as the memory consumption. First, to reduce the communication cost, we propose a diagonalization method such that an approximate Newton direction can be obtained without communication between machines. Second, we consider subsampled Gauss-Newton matrices for reducing the running time as well as the communication cost. Third, to reduce the synchronization cost, we terminate the process of finding an approximate Newton direction even though some nodes have not finished their tasks. Details of some implementation issues in distributed environments are thoroughly investigated. Experiments demonstrate that the proposed method is effective for the distributed training of deep neural networks. In compared with stochastic gradient methods, it is more robust and may give better test accuracy.

Keywords: Deep Neural Networks, Distributed Newton methods, Large-scale classification, Subsampled Hessian.

1 Introduction

Recently deep learning has emerged as a useful technique for data classification as well as finding feature representations. We consider the scenario of multi-class classification. A deep neural network maps each feature vector to one of the class labels by the connection of nodes in a multi-layer structure. Between two adjacent layers a weight matrix maps the inputs (values in the previous layer) to the outputs (values in the current layer). Assume the training set includes $(\boldsymbol{y}^i, \boldsymbol{x}^i)$, $i = 1, \ldots, l$, where $\boldsymbol{x}^i \in \Re^{n_0}$ is the feature vector and $\boldsymbol{y}^i \in \Re^K$ is the label vector. If $\boldsymbol{x}^i$ is associated with label k, then

$$\boldsymbol{y}^i = [\underbrace{0, \ldots, 0}_{k-1}, 1, 0, \ldots, 0]^T \in \Re^K,$$

where K is the number of classes and $\{1, \ldots, K\}$ are possible labels. After collecting all weights and biases as the model vector $\boldsymbol{\theta}$ and having a loss function $\xi(\boldsymbol{\theta}; \boldsymbol{x}, \boldsymbol{y})$, a neural-network problem can be written as the following optimization problem.

$$\min_{\boldsymbol{\theta}} \quad f(\boldsymbol{\theta}), \tag{1}$$

where

$$f(\boldsymbol{\theta}) = \frac{1}{2C}\boldsymbol{\theta}^T\boldsymbol{\theta} + \frac{1}{l}\sum_{i=1}^{l} \xi(\boldsymbol{\theta}; \boldsymbol{x}^i, \boldsymbol{y}^i). \tag{2}$$

The regularization term $\boldsymbol{\theta}^T\boldsymbol{\theta}/2$ avoids overfitting the training data, while the parameter C balances the regularization term and the loss term. The function $f(\boldsymbol{\theta})$ is non-convex because of the connection between weights in different layers. This non-convexity and the large number of weights have caused tremendous difficulties in training large-scale deep neural networks. To apply an optimization algorithm for solving (2), the calculation of function, gradient, and Hessian can be expensive. Currently, stochastic gradient (SG) methods are the most commonly used way to train deep neural networks (e.g., Bottou, 1991; LeCun et al., 1998b; Bottou, 2010; Zinkevich et al., 2010; Dean et al., 2012; Moritz et al., 2015). In particular, some expensive operations can be efficiently conducted in GPU environments (e.g., Ciresan et al., 2010; Krizhevsky et al., 2012; Hinton et al., 2012). Besides stochastic gradient methods, some works such as Martens (2010); Kiros (2013); He et al. (2016) have considered a Newton method of using Hessian information. Other optimization methods such as ADMM have also been considered (Taylor et al., 2016).

When the model or the data set is large, distributed training is needed. Following the design of the objective function in (2), we note it is easy to achieve data parallelism: if data instances are stored in different computing nodes, then each machine can calculate the local sum of training losses independently.[1] However, achieving model parallelism is more difficult because of the complicated structure of deep neural networks. In this work, by considering that the model is distributedly stored we propose a novel distributed Newton method for deep learning. By variable and feature-wise data partitions, and some careful designs, we are able to explicitly use the Jacobian matrix for matrix-vector products in the Newton method. Some techniques are incorporated to reduce the running time as well as the memory consumption. First, to reduce the communication cost, we propose a diagonalization method such that an approximate Newton direction can be obtained without communication between machines. Second, we consider subsampled Gauss-Newton matrices for reducing the running time as well as the communication cost. Third, to reduce the synchronization cost, we terminate the process of finding an approximate Newton direction even though some nodes have not finished their tasks.

To be focused, among the various types of neural networks, we consider the standard feedforward networks in this work. We do not consider other types such as the convolution networks that are popular in computer vision.

This work is organized as follows. Section 2 introduces existing Hessian-free Newton methods for deep learning. In Section 3, we propose a distributed Newton method for training neural networks. We then develop novel techniques in Section 4 to reduce running time and memory consumption. In Section 5 we analyze the cost of the pro-

[1] Training deep neural networks with data parallelism has been considered in SG, Newton and other optimization methods. For example, He et al. (2015) implement a parallel Newton method by letting each node store a subset of instances.

posed algorithm. Additional implementation techniques are given in Section 6. Then Section 7 reviews some existing optimization methods, while experiments in Section 8 demonstrate the effectiveness of the proposed method. Programs used for experiments in this paper are available at

`http://csie.ntu.edu.tw/~cjlin/papers/dnn`.

Supplementary materials including a list of symbols and additional experiments can be found at the same web address.

2 Hessian-free Newton Method for Deep Learning

In this section, we begin with introducing feedforward neural networks and then review existing Hessian-free Newton methods to solve the optimization problem.

2.1 Feedforward Networks

A multi-layer neural network maps each feature vector to a class vector via the connection of nodes. There is a weight vector between two adjacent layers to map the input vector (the previous layer) to the output vector (the current layer). The network in Figure 1 is an example. Let n_m denote the number of nodes at the mth layer. We use n_0(input)-n_1-$\ldots$-n_L(output) to represent the structure of the network.[3] The weight

[2] This figure is modified from the example at `http://www.texample.net/tikz/examples/neural-network`.

[3] Note that n_0 is the number of features and $n_L = K$ is the number of classes.

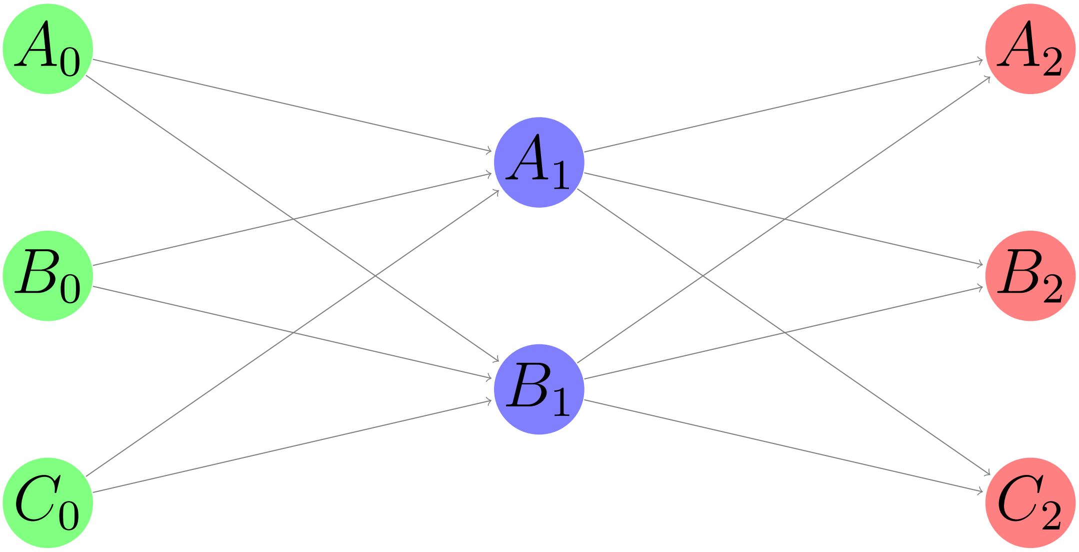


Figure 1: An example of feedforward neural networks.[2]

matrix W^m and the bias vector $\boldsymbol{b}^m$ at the mth layer are

$$W^m = \begin{bmatrix} w^m_{11} & w^m_{12} & \cdots & w^m_{1n_m} \\ w^m_{21} & w^m_{22} & \cdots & w^m_{2n_m} \\ \vdots & \vdots & \vdots & \vdots \\ w^m_{n_{m-1}1} & w^m_{n_{m-1}2} & \cdots & w^m_{n_{m-1}n_m} \end{bmatrix}_{n_{m-1}\times n_m} \quad \text{and} \quad \boldsymbol{b}^m = \begin{bmatrix} b^m_1 \\ b^m_2 \\ \vdots \\ b^m_{n_m} \end{bmatrix}_{n_m\times 1}.$$

Let

$$\boldsymbol{s}^{0,i} = \boldsymbol{z}^{0,i} = \boldsymbol{x}^i$$

be the feature vector for the ith instance, and $\boldsymbol{s}^{m,i}$ and $\boldsymbol{z}^{m,i}$ denote vectors of the ith instance at the mth layer, respectively. We can use

$$\begin{aligned} \boldsymbol{s}^{m,i} &= (W^m)^T \boldsymbol{z}^{m-1,i} + \boldsymbol{b}^m,\ m = 1,\ldots,L,\ i = 1,\ldots,l \\ z^{m,i}_j &= \sigma(s^{m,i}_j),\ j = 1,\ldots,n_m,\ m = 1,\ldots,L,\ i = 1,\ldots,l \end{aligned} \tag{3}$$

to derive the value of the next layer, where $\sigma(\cdot)$ is the activation function.

If W^m's columns are concatenated to the following vector

$$\boldsymbol{w}^m = \begin{bmatrix} w_{11}^m & \dots & w_{n_{m-1}1}^m & w_{12}^m & \dots & w_{n_{m-1}2}^m & \dots & w_{1n_m}^m & \dots & w_{n_{m-1}n_m}^m \end{bmatrix}^T,$$

then we can define

$$\boldsymbol{\theta} = \begin{bmatrix} \boldsymbol{w}^1 \\ \boldsymbol{b}^1 \\ \vdots \\ \boldsymbol{w}^L \\ \boldsymbol{b}^L \end{bmatrix}$$

as the weight vector of a whole deep neural network. The total number of parameters is

$$n = \sum_{m=1}^{L} \left(n_{m-1} \times n_m + n_m \right).$$

Because $\boldsymbol{z}^{L,i}$ is the output vector of the ith data, by a loss function to compare it with the label vector $\boldsymbol{y}^i$, a neural network solves the following regularized optimization problem

$$\min_{\boldsymbol{\theta}} \; f(\boldsymbol{\theta}),$$

where

$$f(\boldsymbol{\theta}) = \frac{1}{2C}\boldsymbol{\theta}^T\boldsymbol{\theta} + \frac{1}{l}\sum_{i=1}^{l} \xi(\boldsymbol{z}^{L,i}; \boldsymbol{y}^i), \tag{4}$$

$C > 0$ is a regularization parameter, and $\xi(\boldsymbol{z}^{L,i}; \boldsymbol{y}^i)$ is a convex function of $\boldsymbol{z}^{L,i}$. Note that we rewrite the loss function $\xi(\boldsymbol{\theta}; \boldsymbol{x}^i, \boldsymbol{y}^i)$ in (2) as $\xi(\boldsymbol{z}^{L,i}; \boldsymbol{y}^i)$ because $\boldsymbol{z}^{L,i}$ is decided by $\boldsymbol{\theta}$ and $\boldsymbol{x}^i$. In this work, we consider the following loss function

$$\xi(\boldsymbol{z}^{L,i}; \boldsymbol{y}^i) = ||\boldsymbol{z}^{L,i} - \boldsymbol{y}^i||^2. \tag{5}$$

The gradient of $f(\boldsymbol{\theta})$ is

$$\nabla f(\boldsymbol{\theta}) = \frac{1}{C}\boldsymbol{\theta} + \frac{1}{l}\sum_{i=1}^{l} (J^i)^T \nabla_{\boldsymbol{z}^{L,i}} \xi(\boldsymbol{z}^{L,i}; \boldsymbol{y}^i), \tag{6}$$

where

$$J^i = \begin{bmatrix} \frac{\partial z_1^{L,i}}{\partial \theta_1} & \cdots & \frac{\partial z_1^{L,i}}{\partial \theta_n} \\ \vdots & \vdots & \vdots \\ \frac{\partial z_{n_L}^{L,i}}{\partial \theta_1} & \cdots & \frac{\partial z_{n_L}^{L,i}}{\partial \theta_n} \end{bmatrix}_{n_L \times n}, \ i = 1, \ldots, l, \tag{7}$$

is the Jacobian of $\boldsymbol{z}^{L,i}$, which is a function of θ. The Hessian matrix of $f(\boldsymbol{\theta})$ is

$$\begin{aligned} \nabla^2 f(\boldsymbol{\theta}) =& \frac{1}{C}\mathcal{I} + \frac{1}{l}\sum_{i=1}^{l}(J^i)^T B^i J^i \\ &+ \frac{1}{l}\sum_{i=1}^{l}\sum_{j=1}^{n_L} \frac{\partial \xi(\boldsymbol{z}^{L,i}; \boldsymbol{y}^i)}{\partial z_j^{L,i}} \begin{bmatrix} \frac{\partial^2 z_j^{L,i}}{\partial \theta_1 \partial \theta_1} & \cdots & \frac{\partial^2 z_j^{L,i}}{\partial \theta_1 \partial \theta_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 z_j^{L,i}}{\partial \theta_n \partial \theta_1} & \cdots & \frac{\partial^2 z_j^{L,i}}{\partial \theta_n \partial \theta_n} \end{bmatrix}, \end{aligned} \tag{8}$$

where $\mathcal{I}$ is the identity matrix and

$$B_{ts}^i = \frac{\partial^2 \xi(\boldsymbol{z}^{L,i}; \boldsymbol{y}^i)}{\partial z_t^{L,i} \partial z_s^{L,i}}, \ t = 1, \ldots, n_L, \ s = 1, \ldots, n_L. \tag{9}$$

From now on for simplicity we let

$$\xi_i \equiv \xi_i(\boldsymbol{z}^{L,i}; \boldsymbol{y}^i).$$

2.2 Hessian-free Newton Method

For the standard Newton methods, at the kth iteration, we find a direction $\boldsymbol{d}^k$ minimizing the following second-order approximation of the function value:

$$\min_{\boldsymbol{d}} \quad \frac{1}{2}\boldsymbol{d}^T H^k \boldsymbol{d} + \nabla f(\boldsymbol{\theta}^k)^T \boldsymbol{d}, \tag{10}$$

where $H^k = \nabla^2 f(\boldsymbol{\theta}^k)$ is the Hessian matrix of $f(\boldsymbol{\theta}^k)$. To solve (10), first we calculate the gradient vector by a backward process based on (3) through the following equations:

$$\frac{\partial \xi_i}{\partial s_j^{m,i}} = \frac{\partial \xi_i}{\partial z_j^{m,i}} \sigma'(s_j^{m,i}),\ i = 1, \dots, l,\ m = 1, \dots, L,\ j = 1, \dots, n_m \tag{11}$$

$$\frac{\partial \xi_i}{\partial z_t^{m-1,i}} = \sum_{j=1}^{n_m} \frac{\partial \xi_i}{\partial s_j^{m,i}} w_{tj}^m,\ i = 1, \dots, l,\ m = 1, \dots, L,\ t = 1, \dots, n_{m-1} \tag{12}$$

$$\frac{\partial f}{\partial w_{tj}^m} = \frac{1}{C} w_{tj}^m + \frac{1}{l} \sum_{i=1}^{l} \frac{\partial \xi_i}{\partial s_j^{m,i}} z_t^{m-1,i},\ m = 1, \dots, L,\ j = 1, \dots, n_m,\ t = 1, \dots, n_{m-1} \tag{13}$$

$$\frac{\partial f}{\partial b_j^m} = \frac{1}{C} b_j^m + \frac{1}{l} \sum_{i=1}^{l} \frac{\partial \xi_i}{\partial s_j^{m,i}},\ m = 1, \dots, L,\ j = 1, \dots, n_m. \tag{14}$$

Note that formally the summation in (13) should be

$$\sum_{i=1}^{l} \sum_{i'=1}^{l} \frac{\partial \xi_i}{\partial s_j^{m,i'}} z_t^{m-1,i'},$$

but it is simplified because ξ_i is associated with only $s_j^{m,i}$.

If H^k is positive definite, then (10) is equivalent to solving the following linear system:

$$H^k \boldsymbol{d} = -\nabla f(\boldsymbol{\theta}^k). \tag{15}$$

Unfortunately, for the optimization problem (10), it is well known that the objective function may be non-convex and therefore H^k is not guaranteed to be positive definite. Following Schraudolph (2002), we can use the Gauss-Newton matrix as an approximation of the Hessian. That is, we remove the last term in (8) and obtain the following positive-definite matrix.

$$G = \frac{1}{C} \mathcal{I} + \frac{1}{l} \sum_{i=1}^{l} (J^i)^T B^i J^i. \tag{16}$$

Note that from (9), each B^i, $i = 1, \ldots, l$ is positive semi-definite if we require that $\xi(\boldsymbol{z}^{L,i}; \boldsymbol{y}^i)$ is a convex function of $\boldsymbol{z}^{L,i}$. Therefore, instead of using (15), we solve the following linear system to find a $\boldsymbol{d}^k$ for deep neural networks.

$$(G^k + \lambda_k \mathcal{I})\boldsymbol{d} = -\nabla f(\boldsymbol{\theta}^k), \tag{17}$$

where G^k is the Gauss-Newton matrix at the kth iteration and we add a term $\lambda_k \mathcal{I}$ because of considering the Levenberg-Marquardt method (see details in Section 4.5).

For deep neural networks, because the total number of weights may be very large, it is hard to store the Gauss-Newton matrix. Therefore, Hessian-free algorithms have been applied to solve (17). Examples include Martens (2010); Ngiam et al. (2011). Specifically, conjugate gradient (CG) methods are often used so that a sequence of Gauss-Newton matrix vector products are conducted. Martens (2010); Wang et al. (2015) use $\mathcal{R}$-operator (Pearlmutter, 1994) to implement the product without storing the Gauss-Newton matrix.

Because the use of $\mathcal{R}$ operators for the Newton method is not the focus of this work, we leave some detailed discussion in Sections II-III in supplementary materials.

3 Distributed Training by Variable Partition

The main computational bottleneck in a Hessian-free Newton method is the sequence of matrix-vector products in the CG procedure. To reduce the running time, parallel matrix-vector multiplications should be conducted. However, the $\mathcal{R}$ operator discussed in Section 2 and Section II in supplementary materials is inherently sequential. In a forward process results in the current layer must be finished before the next. In this section, we propose an effective distributed algorithm for training deep neural networks.

3.1 Variable Partition

Instead of using the $\mathcal{R}$ operator to calculate the matrix-vector product, we consider the whole Jacobian matrix and directly use the Gauss-Newton matrix in (16) for the matrix-vector products in the CG procedure. This setting is possible because of the following reasons.

1. A distributed environment is used.
2. With some techniques we do not need to explicitly store every element of the Jacobian matrix.

Details will be described in the rest of this paper. To begin we split each J^i to P partitions

$$J^i = \begin{bmatrix} J^i_1 & \cdots & J^i_P \end{bmatrix}.$$

Because the number of columns in J^i is the same as the number of variables in the optimization problem, essentially we partition the variables to P subsets. Specifically, we split neurons in each layer to several groups. Then weights connecting one group of the current layer to one group of the next layer form a subset of our variable partition. For example, assume we have a 150-200-30 neural network in Figure 2. By splitting the three layers to 3, 2, 3 groups, we have a total number of partitions $P = 12$. The partition (A_0, A_1) in Figure 2 is responsible for a 50×100 sub-matrix of W^1. In addition, we distribute the variable $\boldsymbol{b}^m$ to partitions corresponding to the first neuron sub-group of the

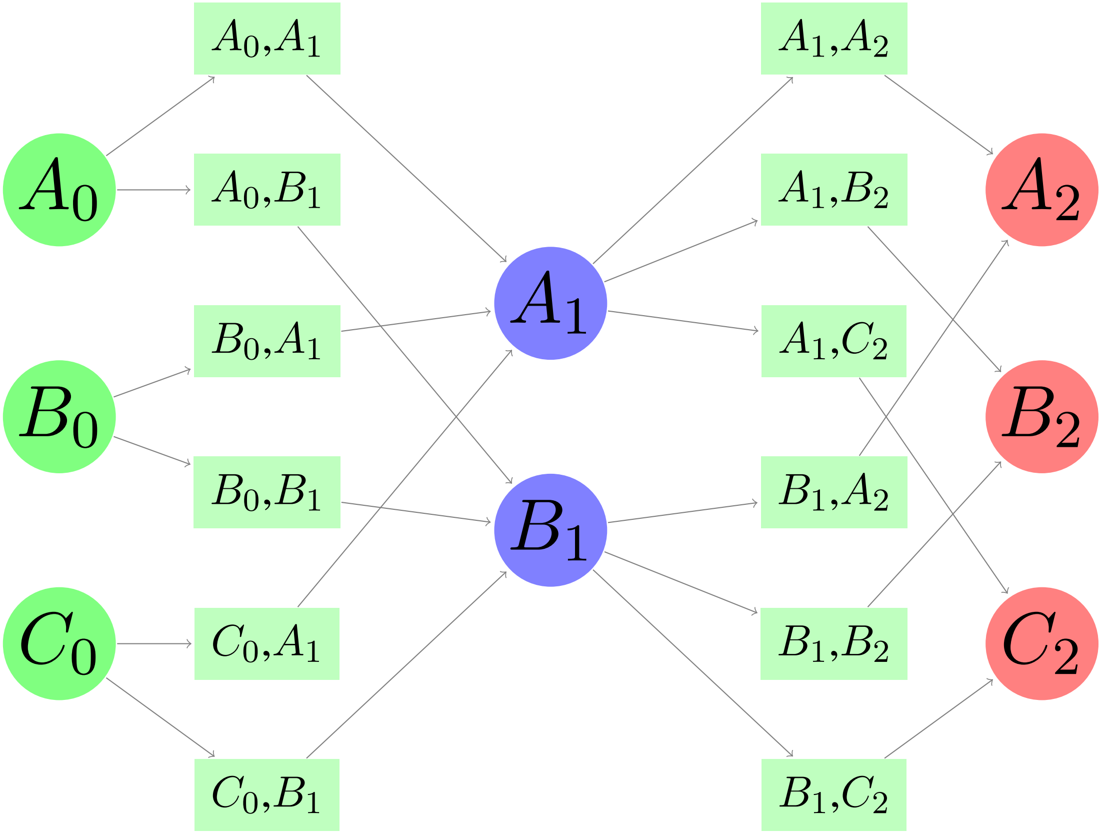


Figure 2: An example of splitting variables in Figure 1 to 12 partitions by a split structure of 3-2-3. Each circle corresponds to a neuron sub-group in a layer, while each square is a partition corresponding to weights connecting one neuron sub-group in a layer to one neuron sub-group in the next layer.

mth layer. For example, the 200 variables of $\boldsymbol{b}^1$ is split to 100 in the partition (A_0, A_1) and 100 in the partition (A_0, B_1).

By the variable partition, we achieve model parallelism. Further, because $\boldsymbol{z}^{0,i} = \boldsymbol{x}^i$ from (2.1), our data points are split in a feature-wise way to nodes corresponding to partitions between layers 0 and 1. Therefore, we have data parallelism.

With the variable partition, the second term in the Gauss-Newton matrix (16) for the

ith instance can be represented as

$$(J^i)^T B^i J^i = \begin{bmatrix} (J_1^i)^T B^i J_1^i & \cdots & (J_1^i)^T B^i J_P^i \\ & \ddots & \\ (J_P^i)^T B^i J_1^i & \cdots & (J_P^i)^T B^i J_P^i \end{bmatrix}.$$

In the CG procedure to solve (17), the product between the Gauss-Newton matrix and a vector $\boldsymbol{v}$ is

$$G\boldsymbol{v} = \begin{bmatrix} \frac{1}{l}\sum_{i=1}^{l}(J_1^i)^T B^i(\sum_{p=1}^{P} J_p^i \boldsymbol{v}_p) + \frac{1}{C}\boldsymbol{v}_1 \\ \vdots \\ \frac{1}{l}\sum_{i=1}^{l}(J_P^i)^T B^i(\sum_{p=1}^{P} J_p^i \boldsymbol{v}_p) + \frac{1}{C}\boldsymbol{v}_P \end{bmatrix}, \text{where } \boldsymbol{v} = \begin{bmatrix} \boldsymbol{v}_1 \\ \vdots \\ \boldsymbol{v}_P \end{bmatrix} \tag{18}$$

is partitioned according to our variable split. From (9) and the loss function defined in (5),

$$B_{ts}^i = \frac{\partial^2 \left(\sum_{j=1}^{n_L}(z_j^{L,i} - y_j^i)^2\right)}{\partial z_t^{L,i} \partial z_s^{L,i}} = \frac{\partial \left(2(z_t^{L,i} - y_t^i)\right)}{\partial z_s^{L,i}} = \begin{cases} 2 & \text{if } t = s, \\ 0 & \text{otherwise.} \end{cases}$$

However, after the variable partition, each J^i may still be a huge matrix. The total space for storing J_p^i, $\forall i$ is roughly

$$n_L \times \frac{n}{P} \times l.$$

If l, the number of data instances, is so large such that

$$l \times \frac{n_L}{P} > n,$$

than storing J_p^i, $\forall i$ requires more space than the $n \times n$ Gauss-Newton matrix. To reduce the memory consumption, we will propose effective techniques in Sections 3.3, 4.3, and 6.1.

With the variable partition, function, gradient, and Jacobian calculations become complicated. We discuss details in Sections 3.2 and 3.3.

3.2 Distributed Function Evaluation

From (3) we know how to evaluate the function value in a single machine, but the implementation in a distributed environment is not trivial. Here we check the details from the perspective of an individual partition. Consider a partition that involves neurons in sets T_{m-1} and T_m from layers $m-1$ and m, respectively. Thus

$$T_{m-1} \subset \{1, \ldots, n_{m-1}\} \text{ and } T_m \subset \{1, \ldots, n_m\}.$$

Because (3) is a forward process, we assume that

$$s_t^{m-1,i}, \; i = 1, \ldots, l, \; \forall t \in T_{m-1}$$

are available at the current partition. The goal is to generate

$$s_j^{m,i}, \; i = 1, \ldots, l, \; \forall j \in T_m$$

and pass them to partitions between layers m and $m+1$. To begin, we calculate

$$z_t^{m-1,i} = \sigma(s_t^{m-1,i}), \; i = 1, \ldots, l \text{ and } t \in T_{m-1}. \tag{19}$$

Then, from (3), the following local values can be calculated for $i = 1, \ldots, l, \; j \in T_m$

$$\begin{cases} \sum_{t \in T_{m-1}} w_{tj}^m z_t^{m-1,i} + b_j^m & \text{if } T_{m-1} \text{ is the first neuron sub-group of layer } m-1, \\ \sum_{t \in T_{m-1}} w_{tj}^m z_t^{m-1,i} & \text{otherwise.} \end{cases} \tag{20}$$

After the local sum in (20) is obtained, we must sum up values in partitions between layers $m-1$ and m.

$$s_j^{m,i} = \sum_{T_{m-1} \in P_{m-1}} \Big(\text{local sum in (20)} \Big), \tag{21}$$

where $i = 1, \ldots, l$, $j \in T_m$, and

$$P_{m-1} = \{T_{m-1} \mid T_{m-1} \text{ is any sub-group of neurons at layer } m-1\}.$$

The resulting $s_j^{m,i}$ values should be broadcasted to partitions between layers m and $m+1$ that correspond to the neuron subset T_m. We explain details of (21) and the *broadcast* operation in Section 3.2.1.

3.2.1 Allreduce and Broadcast Operations

The goal of (21) is to generate and broadcast $s_j^{m,i}$ values to some partitions between layers m and $m+1$, so a *reduce* operation seems to be sufficient. However, we will explain in Section 3.3 that for the Jacobian evaluation and then the product between Gauss-Newton matrix and a vector, the partitions between layers $m-1$ and m corresponding to T_m also need $s_j^{m,i}$ for calculating

$$z_j^{m,i} = \sigma(s_j^{m,i}), \ i = 1, \ldots, l, \ j \in T_m. \tag{22}$$

To this end, we consider an *allreduce* operation so that not only are values reduced from some partitions between layers $m-1$ and m, but also the result is broadcasted to them. After this is done, we make the same result $s_j^{m,i}$ available in partitions between layers m and $m+1$ by choosing the partition corresponding to the first neuron sub-group of layer $m-1$ to conduct a *broadcast* operation. Note that for partitions between layers $L-1$ and L (i.e., the last layer), a broadcast operation is not needed.

Consider the example in Figure 2. For partitions (A_1, A_2), (A_1, B_2), and (A_1, C_2), all of them must get $s_j^{1,i}, j \in A_1$ calculated via (21):

$$s_j^{1,i} = \underbrace{\sum_{t \in A_0} w_{tj}^1 z_t^{0,i} + b_j^1}_{(A_0, A_1)} + \underbrace{\sum_{t \in B_0} w_{tj}^1 z_t^{0,i}}_{(B_0, A_1)} + \underbrace{\sum_{t \in C_0} w_{tj}^1 z_t^{0,i}}_{(C_0, A_1)}. \tag{23}$$

The three local sums are available at partitions (A_0, A_1), (B_0, A_1) and (C_0, A_1) respectively. We first conduct an *allreduce* operation so that $s_j^{1,i},\ j \in A_1$ are available at partitions (A_0, A_1), (B_0, A_1), and (C_0, A_1). Then we choose (A_0, A_1) to broadcast values to (A_1, A_2), (A_1, B_2), and (A_1, C_2).

Depending on the system configurations, suitable ways can be considered for implementing the *allreduce* and the *broadcast* operations (Thakur et al., 2005). In Section IV of supplementary materials we give details of our implementation.

To derive the loss value, we need one final *reduce* operation. For the example in Figure 2, in the end we have $z_j^{2,i},\ j \in A_2,\ B_2,\ C_2$ respectively available in partitions

$$(A_1, A_2),\ (A_1, B_2),\ \text{and}\ (A_1, C_2).$$

We then need the following *reduce* operation

$$||\boldsymbol{z}^{2,i} - \boldsymbol{y}^i||^2 = \sum_{j \in A_2} (z_j^{2,i} - y_j^i)^2 + \sum_{j \in B_2} (z_j^{2,i} - y_j^i)^2 + \sum_{j \in C_2} (z_j^{2,i} - y_j^i)^2 \tag{24}$$

and let (A_1, A_2) have the loss term in the objective value.

We have discussed the calculation of the loss term in the objective value, but we also need to obtain the regularization term $\boldsymbol{\theta}^T\boldsymbol{\theta}/2$. One possible setting is that before the loss-term calculation we run a *reduce* operation to sum up all local regularization terms. For example, in one partition corresponding to neuron subgroups T_{m-1} at layer

$m-1$ and T_m at layer m, the local value is

$$\sum_{t \in T_{m-1}} \sum_{j \in T_m} (w_{tj}^m)^2. \tag{25}$$

On the other hand, we can embed the calculation into the forward process for obtaining the loss term. The idea is that we append the local regularization term in (25) to the vector in (20) for an *allreduce* operation in (21). The cost is negligible because we only increase the length of each vector by one. After the *allreduce* operation, we broadcast the resulting vector to partitions between layers m and $m+1$ that corresponding to the neuron subgroup T_m. We cannot let each partition collect the broadcasted value for subsequent *allreduce* operations because regularization terms in previous layers would be calculated several times. To this end, we allow only the partition corresponding to T_m in layer m and the first neuron subgroup in layer $m+1$ to collect the value and include it with the local regularization term for the subsequent *allreduce* operation. By continuing the forward process, in the end we get the whole regularization term.

We use Figure 2 to give an illustration. The *allreduce* operation in (23) now also calculates

$$\underbrace{\sum_{t \in A_0} \sum_{j \in A_1} (w_{tj}^1)^2 + \sum_{j \in A_1} (b_j^1)^2}_{(A_0,A_1)} + \underbrace{\sum_{t \in B_0} \sum_{j \in A_1} (w_{tj}^1)^2}_{(B_0,A_1)} + \underbrace{\sum_{t \in C_0} \sum_{j \in A_1} (w_{tj}^1)^2}_{(C_0,A_1)}. \tag{26}$$

The resulting value is broadcasted to

$$(A_1, A_2),\ (A_1, B_2),\ \text{and } (A_1, C_2).$$

Then only (A_1, A_2) collects the value and generate the following local sum:

$$(26) + \sum_{t \in A_1} \sum_{j \in A_2} (w_{tj}^2)^2 + \sum_{j \in A_2} (b_j^2)^2.$$

In the end we have

1. (A_1, A_2) contains regularization terms from

$$(A_0, A_1),\ (B_0, A_1),\ (C_0, A_1),\ (A_1, A_2),\ (A_0, B_1),\ (B_0, B_1),\ (C_0, B_1),\ (B_1, A_2).$$

2. (A_1, B_2) contains regularization terms from

$$(A_1, B_2),\ (B_1, B_2).$$

3. (A_1, C_2) contains regularization terms from

$$(A_1, C_2),\ (B_1, C_2).$$

We can then extend the reduce operation in (24) to generate the final value of the regularization term.

3.3 Distributed Jacobian Calculation

From (7) and similar to the way of calculating the gradient in (11)-(14), the Jacobian matrix satisfies the following properties.

$$\frac{\partial z_u^{L,i}}{\partial w_{tj}^m} = \frac{\partial z_u^{L,i}}{\partial s_j^{m,i}} \frac{\partial s_j^{m,i}}{\partial w_{tj}^m}, \tag{28}$$

$$\frac{\partial z_u^{L,i}}{\partial b_j^m} = \frac{\partial z_u^{L,i}}{\partial s_j^{m,i}} \frac{\partial s_j^{m,i}}{\partial b_j^m}, \tag{29}$$

where $i = 1, \dots, l,\ u = 1, \dots, n_L,\ m = 1, \dots, L,\ j = 1, \dots, n_m$, and $t = 1, \dots, n_{m-1}$. However, these formulations do not reveal how they are calculated in a distributed setting. Similar to Section 3.2, we check details from the perspective of any variable partition. Assume the current partition involves neurons in sets T_{m-1} and T_m from layers $m-1$ and m, respectively. Then we aim to obtain the following Jacobian components.

$$\frac{\partial z_u^{L,i}}{\partial w_{tj}^m} \text{ and } \frac{\partial z_u^{L,i}}{\partial b_j^m},\ \forall t \in T_{m-1},\ \forall j \in T_m,\ u = 1, \dots, n_L,\ i = 1, \dots, l.$$

Before showing how to calculate them, we first get from (3) that

$$\frac{\partial z_u^{L,i}}{\partial s_j^{m,i}} = \frac{\partial z_u^{L,i}}{\partial z_j^{m,i}} \frac{\partial z_j^{m,i}}{\partial s_j^{m,i}} = \frac{\partial z_u^{L,i}}{\partial z_j^{m,i}} \sigma'(s_j^{m,i}), \tag{30}$$

$$\frac{\partial s_j^{m,i}}{\partial w_{tj}^m} = z_t^{m-1,i} \text{ and } \frac{\partial s_j^{m,i}}{\partial b_j^m} = 1, \tag{31}$$

$$\frac{\partial z_u^{L,i}}{\partial z_j^{L,i}} = \begin{cases} 1 & \text{if } j = u, \\ 0 & \text{otherwise.} \end{cases} \tag{32}$$

From (28)-(32), the elements for the local Jacobian matrix can be derived by

$$\frac{\partial z_u^{L,i}}{\partial w_{tj}^m} = \frac{\partial z_u^{L,i}}{\partial z_j^{m,i}} \frac{\partial z_j^{m,i}}{\partial s_j^{m,i}} \frac{\partial s_j^{m,i}}{\partial w_{tj}^m} = \begin{cases} \frac{\partial z_u^{L,i}}{\partial z_j^{m,i}} \sigma'(s_j^{m,i}) z_t^{m-1,i} & \text{if } m < L, \\ \sigma'(s_u^{L,i}) z_t^{L-1,i} & \text{if } m = L,\ j = u, \\ 0 & \text{if } m = L,\ j \neq u, \end{cases} \tag{33}$$

and

$$\frac{\partial z_u^{L,i}}{\partial b_j^m} = \frac{\partial z_u^{L,i}}{\partial z_j^{m,i}} \frac{\partial z_j^{m,i}}{\partial s_j^{m,i}} \frac{\partial s_j^{m,i}}{\partial b_j^m} = \begin{cases} \frac{\partial z_u^{L,i}}{\partial z_j^{m,i}} \sigma'(s_j^{m,i}) & \text{if } m < L, \\ \sigma'(s_u^{L,i}) & \text{if } m = L,\ j = u, \\ 0 & \text{if } m = L,\ j \neq u, \end{cases} \tag{34}$$

where $u = 1, \ldots, n_L$, $i = 1, \ldots, l$, $t \in T_{m-1}$, and $j \in T_m$.

We discuss how to have values in the right-hand side of (33) and (34) available at the current computing node. From (19), we have

$$z_t^{m-1,i},\ \forall i = 1, \ldots, l,\ \forall t \in T_{m-1}$$

available in the forward process of calculating the function value. Further, in (21)-(22) to obtain $z_j^{m,i}$ for layers m and $m+1$, we use an *allreduce* operation rather than a *reduce* operation so that

$$s_j^{m,i},\ \forall i = 1, \ldots, l,\ \forall j \in T_m$$

are available at the current partition between layers $m-1$ and m. Therefore, $\sigma'(s_j^{m,i})$ in (33)-(34) can be obtained. The remaining issue is to generate $\partial z_u^{L,i}/\partial z_j^{m,i}$. We will show that they can be obtained by a backward process. Because the discussion assumes that currently we are at a partition between layers $m-1$ and m, we show details of generating $\partial z_u^{L,i}/\partial z_t^{m-1,i}$ and dispatching them to partitions between $m-2$ and $m-1$. From (3) and (30), $\partial z_u^{L,i}/z_t^{m-1,i}$ can be calculated by

$$\frac{\partial z_u^{L,i}}{\partial z_t^{m-1,i}} = \sum_{j=1}^{n_m} \frac{\partial z_u^{L,i}}{\partial s_j^{m,i}} \frac{\partial s_j^{m,i}}{\partial z_t^{m-1,i}} = \sum_{j=1}^{n_m} \frac{\partial z_u^{L,i}}{\partial z_j^{m,i}} \sigma'(s_j^{m,i}) w_{tj}^m. \tag{35}$$

Therefore, we consider a backward process of using $\partial z_u^{L,i}/\partial z_j^{m,i}$ to generate $\partial z_u^{L,i}/\partial z_t^{m-1,i}$. In a distributed system, from (32) and (35),

$$\frac{\partial z_u^{L,i}}{\partial z_t^{m-1,i}} = \begin{cases} \sum_{T_m \in P_m} \sum_{j \in T_m} \frac{\partial z_u^{L,i}}{\partial z_j^{m,i}} \sigma'(s_j^{m,i}) w_{tj}^m & \text{if } m < L, \\ \sum_{T_m \in P_m} \sigma'(s_u^{L,i}) w_{tu}^L & \text{if } m = L, \end{cases} \tag{36}$$

where $i = 1, \ldots, l,\ u = 1, \ldots, n_L,\ t \in T_{m-1}$, and

$$P_m = \{T_m \mid T_m \text{ is any sub-group of neurons at layer } m\}. \tag{37}$$

Clearly, each partition calculates the local sum over $j \in T_m$. Then a *reduce* operation is needed to sum up values in all corresponding partitions between layers $m-1$ and m. Subsequently, we discuss details of how to transfer data to partitions between layers $m-2$ and $m-1$.

Consider the example in Figure 2. The partition (A_0, A_1) must get

$$\frac{\partial z_u^{L,i}}{\partial z_t^{1,i}},\ t \in A_1,\ u = 1, \ldots, n_L,\ i = 1, \ldots, l.$$

From (36),

$$\frac{\partial z_u^{L,i}}{\partial z_t^{1,i}} = \underbrace{\sum_{j \in A_2} \frac{\partial z_u^{L,i}}{\partial z_j^{2,i}} \sigma'(s_j^{2,i}) w_{tj}^2}_{(A_1, A_2)} + \underbrace{\sum_{j \in B_2} \frac{\partial z_u^{L,i}}{\partial z_j^{2,i}} \sigma'(s_j^{2,i}) w_{tj}^2}_{(A_1, B_2)} + \underbrace{\sum_{j \in C_2} \frac{\partial z_u^{L,i}}{\partial z_j^{2,i}} \sigma'(s_j^{2,i}) w_{tj}^2}_{(A_1, C_2)}. \tag{38}$$

Note that these three sums are available at partitions (A_1, A_2), (A_1, B_2), and (A_1, C_2), respectively. Therefore, (38) is a *reduce* operation. Further, values obtained in (38) are needed in partitions not only (A_0, A_1) but also (B_0, A_1) and (C_0, A_1). Therefore, we need a *broadcast* operation so values can be available in the corresponding partitions.

For details of implementing *reduce* and *broadcast* operations, see Section IV of supplementary materials. Algorithm 2 summarizes the backward process to calculate $\partial z_u^{L,i}/\partial z_j^{m,i}$.

3.3.1 Memory Requirement

We have mentioned in Section 3.1 that storing all elements in the Jacobian matrix may not be viable. In the distributing setting, if we store all Jacobian elements corresponding to the current partition, then

$$|T_{m-1}| \times |T_m| \times n_L \times l \tag{39}$$

space is needed. We propose a technique to save space by noting that (28) can be written as the product of two terms. From (30)-(31), the first term is related to only T_m, while the second is related to only T_{m-1}:

$$\frac{\partial z_u^{L,i}}{\partial w_{tj}^m} = [\frac{\partial z_u^{L,i}}{\partial s_j^{m,i}}][\frac{\partial s_j^{m,i}}{\partial w_{tj}^m}] = [\frac{\partial z_u^{L,i}}{\partial z_j^{m,i}}\sigma'(s_j^{m,i})][z_t^{m-1,i}]. \tag{40}$$

They are available in our earlier calculation. Specifically, we allocate space to receive $\partial z_u^{L,i}/\partial z_j^{m,i}$ from previous layers. After obtaining the values, we replace them with

$$\frac{\partial z_u^{L,i}}{\partial z_j^{m,i}}\sigma'(s_j^{m,i}) \tag{41}$$

for the future use. Therefore, the Jacobian matrix is not explicitly stored. Instead, we use the two terms in (40) for the product between the Gauss-Newton matrix and a vector in the CG procedure. See details in Section 4.2. Note that we also need to calculate and store the local sum before the reduce operation in (36) for getting $\partial z_u^{L,i}/\partial z_t^{m-1,i}$, $\forall t \in T_{m-1}$, $\forall u$, $\forall i$. Therefore, the memory consumption is proportional to

$$l \times n_L \times (|T_{m-1}| + |T_m|).$$

This setting significantly reduces the memory consumption of directly storing the Jacobian matrix in (39).

3.3.2 Sigmoid Activation Function

In the discussion so far, we consider a general differentiable activation function $\sigma(s_j^{m,i})$. In the implementation in this paper, we consider the sigmoid function except the output layer:

$$z_j^{m,i} = \sigma(s_j^{m,i}) = \begin{cases} \frac{1}{1+e^{-s_j^{m,i}}} & \text{if } m < L, \\ s_j^{m,i} & \text{if } m = L. \end{cases} \tag{42}$$

Then,

$$\sigma'(s_j^{m,i}) = \begin{cases} \frac{e^{-s_j^{m,i}}}{\left(1+e^{-s_j^{m,i}}\right)^2} = z_j^{m,i}(1 - z_j^{m,i}) & \text{if } m < L, \\ 1 & \text{if } m = L. \end{cases}$$

and (33)-(34) become

$$\frac{\partial z_u^{L,i}}{\partial w_{tj}^m} = \begin{cases} \frac{\partial z_u^{L,i}}{\partial z_j^{m,i}} z_j^{m,i}(1 - z_j^{m,i}) z_t^{m-1,i}, \\ z_t^{L-1,i}, \\ 0, \end{cases} , \frac{\partial z_u^{L,i}}{\partial b_j^m} = \begin{cases} \frac{\partial z_u^{L,i}}{\partial z_j^{m,i}} z_j^{m,i}(1 - z_j^{m,i}) & \text{if } m < L, \\ 1 & \text{if } m = L,\ j = u, \\ 0 & \text{if } m = L,\ j \neq u, \end{cases}$$

where $u = 1, \ldots, n_L$, $i = 1, \ldots, l$, $t \in T_{m-1}$, and $j \in T_m$.

3.4 Distributed Gradient Calculation

For the gradient calculation, from (4),

$$\frac{\partial f}{\partial w_{tj}^m} = \frac{1}{C} w_{tj}^m + \frac{1}{l} \sum_{i=1}^{l} \frac{\partial \xi_i}{\partial w_{tj}^m} = \frac{1}{C} w_{tj}^m + \frac{1}{l} \sum_{i=1}^{l} \sum_{u=1}^{n_L} \frac{\partial \xi_i}{\partial z_u^{L,i}} \frac{\partial z_u^{L,i}}{\partial w_{tj}^m}, \tag{43}$$

where $\partial z_u^{L,i}/\partial w_{tj}^m,\ \forall t,\ \forall j$ are components of the Jacobian matrix; see also the matrix form in (6). From (33), we have known how to calculate $\partial z_u^{L,i}/\partial w_{tj}^m$. Therefore, if $\partial \xi_i/\partial z_u^{L,i}$ is passed to the current partition, we can easily obtain the gradient vector via (43). This can be finished in the same backward process of calculating the Jacobian matrix.

On the other hand, in the technique that will be introduced in Section 4.3, we only consider a subset of instances to construct the Jacobian matrix as well as the Gauss-Newton matrix. That is, by selecting a subset $S \subset \{1, \ldots, l\}$, then only $J^i, \forall i \in S$ are considered. Thus we do not have all the needed $\partial z_u^{L,i}/\partial w_{tj}^m$ for (43). In this situation, we can separately consider a backward process to calculate the gradient vector. From a derivation similar to (33),

$$\frac{\partial \xi_i}{\partial w_{tj}^m} = \frac{\partial \xi_i}{\partial z_j^{m,i}} \sigma'(s_j^{m,i}) z_t^{m-1,i},\ m = 1, \ldots, L. \tag{44}$$

By considering $\partial \xi_i/\partial z_j^{m,i}$ to be like $\partial z_u^{L,i}/\partial z_j^{m,i}$ in (36), we can apply the same backward process so that each partition between layers $m-2$ and $m-1$ must wait for $\partial \xi_i/\partial z_j^{m-1,i}$ from partitions between layers $m-1$ and m:

$$\frac{\partial \xi_i}{\partial z_t^{m-1,i}} = \sum_{T_m \in P_m} \sum_{j \in T_m} \frac{\partial \xi_i}{\partial z_j^{m,i}} \sigma'(s_j^{m,i}) w_{tj}^m, \tag{45}$$

where $i = 1, \ldots, l$, $t \in T_{m-1}$, and P_m is defined in (37). For the initial $\partial \xi_i/\partial z_j^{L,i}$ in the backward process, from the loss function defined in (5),

$$\frac{\partial \xi_i}{\partial z_j^{L,i}} = 2 \times \left(z_j^{L,i} - y_j^i \right).$$

From (43), a difference from the Jacobian calculation is that here we obtain a sum over all instances i. Earlier we separately maintain terms related to T_{m-1} and T_m to avoid storing all Jacobian elements. With the summation over i, we can afford to store $\partial f / \partial w_{tj}^m$ and $\partial f / \partial b_j^m$, $\forall t \in T_{m-1}$, $\forall j \in T_m$.

4 Techniques to Reduce Computational, Communication, and Synchronization Cost

In this section we propose some novel techniques to make the distributed Newton method a practical approach for deep neural networks.

4.1 Diagonal Gauss-Newton Matrix Approximation

In (18) for the Gauss-Newton matrix-vector products in the CG procedure, we notice that the communication occurs for reducing P vectors

$$J_1^i \boldsymbol{v}_1, \ldots, J_P^i \boldsymbol{v}_P,$$

each with size $\mathcal{O}(n_L)$, and then broadcasting the sum to all nodes. To avoid the high communication cost in some distributed systems, we may consider the diagonal blocks of the Gauss-Newton matrix as its approximation:

$$\hat{G} = \frac{1}{C}\mathcal{I} + \begin{bmatrix} \frac{1}{l}\sum_{i=1}^{l}(J_1^i)^T B^i J_1^i & & \\ & \ddots & \\ & & \frac{1}{l}\sum_{i=1}^{l}(J_P^i)^T B^i J_P^i \end{bmatrix}. \tag{49}$$

Then (17) becomes P independent linear systems

$$\begin{aligned} (\frac{1}{l}\sum_{i=1}^{l}(J_1^i)^T B^i J_1^i + \frac{1}{C}\mathcal{I} + \lambda_k \mathcal{I})\boldsymbol{d}_1^k &= -\boldsymbol{g}_1^k, \\ &\vdots \\ (\frac{1}{l}\sum_{i=1}^{l}(J_P^i)^T B^i J_P^i + \frac{1}{C}\mathcal{I} + \lambda_k \mathcal{I})\boldsymbol{d}_P^k &= -\boldsymbol{g}_P^k, \end{aligned} \tag{50}$$

where $\boldsymbol{g}_1^k, \ldots, \boldsymbol{g}_P^k$ are local components of the gradient:

$$\nabla f(\boldsymbol{\theta}^k) = \begin{bmatrix} \boldsymbol{g}_1^k \\ \vdots \\ \boldsymbol{g}_P^k \end{bmatrix}.$$

The matrix-vector product becomes

$$G\boldsymbol{v} \approx \hat{G}\boldsymbol{v} = \begin{bmatrix} \frac{1}{l}\sum_{i=1}^{l}(J_1^i)^T B^i J_1^i \boldsymbol{v}_1 + \frac{1}{C}\boldsymbol{v}_1 \\ \vdots \\ \frac{1}{l}\sum_{i=1}^{l}(J_P^i)^T B^i J_P^i \boldsymbol{v}_P + \frac{1}{C}\boldsymbol{v}_P \end{bmatrix}, \tag{51}$$

in which each $(G\boldsymbol{v})_p$ can be calculated using only local information because we have independent linear systems. For the CG procedure at any partition, it is terminated if the following relative stopping condition holds

$$||\frac{1}{l}\sum_{i=1}^{l}(J_p^i)^T B^i J_p^i \boldsymbol{v}_p + (\frac{1}{C} + \lambda_k)\boldsymbol{v}_p + \boldsymbol{g}_p^k|| \leq \sigma ||\boldsymbol{g}_p^k|| \tag{52}$$

or the number of CG iterations reaches a pre-specified limit. Here σ is a pre-specified tolerance. Unfortunately, partitions may finish their CG procedures at different time, a situation that results in significant waiting time. To address this synchronization cost, we propose some novel techniques in Section 4.4.

Some past works have considered using diagonal blocks as the approximation of the Hessian. For logistic regression, Bian et al. (2013) consider diagonal elements of the Hessian to solve several one-variable sub-problems in parallel. Mahajan et al. (2017) study a more general setting in which using diagonal blocks is a special case.

4.2 Product Between Gauss-Newton Matrix and a Vector

In the CG procedure the main computational task is the matrix-vector product. We present techniques for the efficient calculation. From (51), for the pth partition, the product between the local diagonal block of the Gauss-Newton matrix and a vector $\boldsymbol{v}_p$ takes the following form.

$$(J_p^i)^T B^i J_p^i \boldsymbol{v}_p.$$

Assume the pth partition involves neuron sub-groups T_{m-1} and T_m respectively in layers $m-1$ and m, and this partition is not responsible to handle the bias term $\boldsymbol{b}_j^m$, $\forall j \in T_m$. Then

$$J_p^i \in \mathcal{R}^{n_L \times (|T_{m-1}| \times |T_m|)} \text{ and } \boldsymbol{v}_p \in \mathcal{R}^{(|T_{m-1}| \times |T_m|) \times 1}.$$

Let $\text{mat}(\boldsymbol{v}_p) \in \mathcal{R}^{|T_{m-1}| \times |T_m|}$ be the matrix representation of $\boldsymbol{v}_p$. From (40), the uth component of $(J_p^i \boldsymbol{v}_p)_u$ is

$$\sum_{t \in T_{m-1}} \sum_{j \in T_m} \frac{\partial z_u^{L,i}}{\partial w_{tj}^m} (\text{mat}(\boldsymbol{v}_p))_{tj} = \sum_{t \in T_{m-1}} \sum_{j \in T_m} \frac{\partial z_u^{L,i}}{\partial s_j^{m,i}} z_t^{m-1,i} (\text{mat}(\boldsymbol{v}_p))_{tj}. \quad (53)$$

A direct calculation of the above value requires $\mathcal{O}(|T_{m-1}| \times |T_m|)$ operations. Thus to get all $u = 1, \ldots, n_L$ components, the total computational cost is proportional to

$$n_L \times |T_{m-1}| \times |T_m|.$$

We discuss a technique to reduce the cost by rewriting (53) as

$$\sum_{j \in T_m} \frac{\partial z_u^{L,i}}{\partial s_j^{m,i}} \left(\sum_{t \in T_{m-1}} z_t^{m-1,i} (\text{mat}(\boldsymbol{v}_p))_{tj} \right).$$

While calculating

$$\sum_{t \in T_{m-1}} z_t^{m-1,i} (v_p)_{tj}, \ \forall j \in T_m$$

still needs $\mathcal{O}(|T_{m-1}| \times |T_m|)$ cost, we notice that these values are independent of u. That is, they can be stored and reused in calculating $(J_p^i \boldsymbol{v}_p)_u, \ \forall u$. Therefore, the total computational cost is significantly reduced to

$$|T_{m-1}| \times |T_m| + n_L \times |T_m|. \tag{54}$$

The procedure of deriving $(J_p^i)^T (B^i J_p^i \boldsymbol{v}_p)$ is similar. Assume

$$\bar{\boldsymbol{v}} = B^i J_p^i \boldsymbol{v}_p \in \mathcal{R}^{n_L \times 1}.$$

From (40),

$$\begin{aligned}
\text{mat}\left((J_p^i)^T \bar{\boldsymbol{v}}\right)_{tj} &= \sum_{u=1}^{n_L} \frac{\partial z_u^{L,i}}{\partial w_{tj}^m} \bar{v}_u \\
&= \sum_{u=1}^{n_L} \frac{\partial z_u^{L,i}}{\partial s_j^{m,i}} z_t^{m-1,i} \bar{v}_u \\
&= z_t^{m-1,i} \left(\sum_{u=1}^{n_L} \frac{\partial z_u^{L,i}}{\partial s_j^{m,i}} \bar{v}_u \right).
\end{aligned} \tag{55}$$

Because

$$\sum_{u=1}^{n_L} \frac{\partial z_u^{L,i}}{\partial s_j^{m,i}} \bar{v}_u, \ \forall j \in T_m \tag{56}$$

are independent of t, we can calculate and store them for the computation in (55). Therefore, the total computational cost is proportional to

$$|T_{m-1}| \times |T_m| + n_L \times |T_m|, \tag{57}$$

which is the same as that for $(J_p^i \boldsymbol{v}_p)$.

In the above discussion, we assume that diagonal blocks of the Gauss-Newton matrix are used. If instead the whole Gauss-Newton matrix is considered, then we calculate

$$(J_{p_1}^i)^T (B^i (J_{p_2}^i \boldsymbol{v}_{p_2})),$$

for any two partitions p_1 and p_2. The same techniques introduced in this section can be applied because (53) and (55) are two independent operations.

4.3 Subsampled Hessian Newton Method

From (16) we see that the computational cost between the Gauss-Newton matrix and a vector is proportional to the number of data. To reduce the cost, subsampled Hessian Newton method (Byrd et al., 2011; Martens, 2010; Wang et al., 2015) have been proposed for selecting a subset of data at each iteration to form an approximate Hessian. Instead of $\nabla^2 f(\boldsymbol{\theta})$ in (15) we use a subset S to have

$$\frac{\mathcal{I}}{C} + \frac{1}{|S|} \sum_{i \in S} \nabla^2_{\boldsymbol{\theta}\boldsymbol{\theta}} \xi(\boldsymbol{z}^{L,i}; \boldsymbol{y}^i).$$

Note that $\boldsymbol{z}^{L,i}$ is a function of $\boldsymbol{\theta}$. The idea behind this subsampled Hessian is that when a large set of points are under the same distribution,

$$\frac{1}{|S|} \sum_{i \in S} \xi(\boldsymbol{z}^{L,i}; \boldsymbol{y}^i).$$

is a good approximation of the average training losses. For neural networks we consider the Gauss-Newton matrix, so (16) becomes the following subsampled Gauss-Newton matrix.

$$G^S = \frac{\mathcal{I}}{C} + \frac{1}{|S|} \sum_{i \in S} (J^i)^T B^i J^i. \tag{58}$$

Now denote the subset at the kth iteration as S_k. The linear system (17) is changed to

$$(G^{S_k} + \lambda_k \mathcal{I}) \boldsymbol{d}^k = -\nabla f(\boldsymbol{\theta}^k). \tag{59}$$

After variable partitions, the independent linear systems are

$$\begin{aligned} \left(\lambda_k \mathcal{I} + \frac{1}{C}\mathcal{I} + \frac{1}{|S_k|} \sum_{i \in S_k} (J_1^i)^T B^i J_1^i \right) \boldsymbol{d}_1^k &= -\boldsymbol{g}_1^k, \\ &\vdots \\ \left(\lambda_k \mathcal{I} + \frac{1}{C}\mathcal{I} + \frac{1}{|S_k|} \sum_{i \in S_k} (J_P^i)^T B^i J_P^i \right) \boldsymbol{d}_P^k &= -\boldsymbol{g}_P^k. \end{aligned} \tag{60}$$

While using diagonal blocks of the Gauss-Newton matrix avoids the communication between partitions, the resulting direction may not be as good as that of using the whole Gauss-Newton matrix. Here we extend an approach by Wang et al. (2015) to pay some extra cost for improving the direction. Their idea is that after the CG procedure of using a sub-sampled Hessian, they consider the full Hessian to adjust the direction. Now in the CG procedure we use a block diagonal approximation of the sub-sampled matrix G^{S_k}, so after that we consider the whole G^{S_k} for adjusting the direction. Specifically, if $\boldsymbol{d}^k$ is obtained from the CG procedure, we solve the following two-variable optimization problem that involves G^{S_k}.

$$\min_{\beta_1, \beta_2} \frac{1}{2} (\beta_1 \boldsymbol{d}^k + \beta_2 \bar{\boldsymbol{d}}^k)^T G^{S_k} (\beta_1 \boldsymbol{d}^k + \beta_2 \bar{\boldsymbol{d}}^k) + \nabla f(\boldsymbol{\theta}^k)^T (\beta_1 \boldsymbol{d}^k + \beta_2 \bar{\boldsymbol{d}}^k), \tag{61}$$

where $\bar{\boldsymbol{d}}^k$ is a chosen vector. Then the new direction is

$$\boldsymbol{d}^k \leftarrow \beta_1 \boldsymbol{d}^k + \beta_2 \bar{\boldsymbol{d}}^k.$$

Here we follow Wang et al. (2015) to choose

$$\bar{\boldsymbol{d}}^k = \boldsymbol{d}^{k-1}.$$

Notice that we choose $\bar{\boldsymbol{d}}^0$ to be the zero vector. A possible advantage of considering $\boldsymbol{d}^{k-1}$ is that it is from the previous iteration of using a different data subset S_{k-1} for the subsampled Gauss-Newton matrix. Thus it provides information from instances not in the current S_k.

To solve (61), because G^{S_k} is positive definite, it is equivalent to solving the following two-variable linear system.

$$\begin{pmatrix} (\boldsymbol{d}^k)^T G^{S_k} \boldsymbol{d}^k & (\bar{\boldsymbol{d}}^k)^T G^{S_k} \boldsymbol{d}^k \\ (\bar{\boldsymbol{d}}^k)^T G^{S_k} \boldsymbol{d}^k & (\bar{\boldsymbol{d}}^k)^T G^{S_k} \bar{\boldsymbol{d}}^k \end{pmatrix} \begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix} = \begin{pmatrix} -\nabla f(\boldsymbol{\theta}^k)^T \boldsymbol{d}^k \\ -\nabla f(\boldsymbol{\theta}^k)^T \bar{\boldsymbol{d}}^k \end{pmatrix}. \tag{62}$$

Note that the construction of (62) involves the communication between partitions; see detailed discussion in Section V of supplementary materials. The effectiveness of using (61) is investigated in Section VII.

In some situations, the linear system (62) may be ill-conditioned. We set $\beta_1 = 1$ and $\beta_2 = 0$ if

$$\begin{vmatrix} (\boldsymbol{d}^k)^T G^{S_k} \boldsymbol{d}^k & (\bar{\boldsymbol{d}}^k)^T G^{S_k} \boldsymbol{d}^k \\ (\bar{\boldsymbol{d}}^k)^T G^{S_k} \boldsymbol{d}^k & (\bar{\boldsymbol{d}}^k)^T G^{S_k} \bar{\boldsymbol{d}}^k \end{vmatrix} \leq \varepsilon, \tag{63}$$

where ε is a small number.

4.4 Synchronization Between Partitions

While the setting in (51) has made each node conduct its own CG procedure without communication, we must wait until all nodes complete their tasks before getting into the next Newton iteration. This synchronization cost can be significant. We note that the running time at each partition may vary because of the following reasons.

1. Because we select a subset of weights between two layers as a partition, the number of variables in each partition may be different. For example, assume the network structure is

   $$50\text{-}100\text{-}2.$$

   The last layer has only two neurons because of the small number of classes. For the weight matrix W^m, a partition between the last two layers can have at most 200 variables. In contrast, a partition between the first two layers may have more variables. Therefore, in the split of variables we should make partitions as balanced as possible. A example will be given later when we introduce the experiment settings in Section 8.1.

2. Each node can start its first CG iteration after the needed information is available. From (30)-(34), the calculation of the information needed for matrix-vector products involves a backward process, so partitions corresponding to neurons in the last layers start the CG procedure earlier than those of the first layers.

To reduce the synchronization cost, a possible solution is to terminate the CG procedure for all partitions if one of them reaches its CG stopping condition:

$$||(\lambda_k + \frac{1}{C})\boldsymbol{v}_p + \frac{1}{|S_k|}\sum_{i\in S_k}(J_p^i)^T B^i J_p^i \boldsymbol{v}_p + \boldsymbol{g}_p|| \leq \sigma ||\boldsymbol{g}_p||. \tag{64}$$

However, under this setting the CG procedure may terminate too early because some partitions have not conducted enough CG steps yet. To strike for a balance, in our implementation we terminate the CG procedure for all partitions when the following conditions are satisfied:

1. Every partition has reached a pre-specified minimum number of CG iterations, CG_{min}.

2. A certain percentage of partitions have reached their stopping conditions, (64).

In Section 8.1, we conduct experiments with different percentage values to check the effectiveness of this setting.

4.5 Summary of the Procedure

We summarize in Algorithm 3 the proposed distributed subsampled Hessian Newton algorithm. Besides materials described earlier in this section, here we explain other steps in the algorithm.

First, in most optimization algorithms, after a direction $\boldsymbol{d}^k$ is obtained, a suitable step size α_k must be decided to ensure the sufficient decrease of $f(\boldsymbol{\theta}^k + \alpha_k \boldsymbol{d}^k)$. Here we consider a backtracking line search by selecting the largest $\alpha_k \in \{1, \frac{1}{2}, \frac{1}{4}, \ldots\}$ such that the following sufficient decrease condition on the function value holds.

$$f(\boldsymbol{\theta}^k + \alpha_k \boldsymbol{d}^k) \leq f(\boldsymbol{\theta}^k) + \eta \alpha_k \nabla f(\boldsymbol{\theta}^k)^T \boldsymbol{d}^k, \tag{65}$$

where $\eta \in (0, 1)$ is a pre-defined constant.

Secondly, we follow Martens (2010); Martens and Sutskever (2012); Wang et al. (2015) to apply the Levenberg-Marquardt method by introducing a term $\lambda_k \mathcal{I}$ in the linear system (17). Define

$$\rho_k = \frac{f(\boldsymbol{\theta}^k + \alpha_k \boldsymbol{d}^k) - f(\boldsymbol{\theta}^k)}{\alpha_k \nabla f(\boldsymbol{\theta}^k)^T \boldsymbol{d}^k + \frac{1}{2}(\alpha_k)^2 (\boldsymbol{d}^k)^T G^{S_k} \boldsymbol{d}_k}$$

as the ratio between the actual function reduction and the predicted reduction. Based on ρ_k, the following rule derives the next λ_{k+1}.

$$\lambda_{k+1} = \begin{cases} \lambda_k \times \text{drop} & \rho_k > 0.75, \\ \lambda_k & 0.25 \leq \rho_k \leq 0.75, \\ \lambda_k \times \text{boost} & \text{otherwise}, \end{cases} \tag{66}$$

where (drop,boost) are given constants. Therefore, if the predicted reduction is close to the true function reduction, we reduce λ_k such that a direction closer to the Newton direction is considered. In contrast, if ρ_k is small, we enlarge λ_k so that a conservative direction close to the negative gradient is considered.

Note that line search already serves as a way to adjust the direction according to the function-value reduction, so in optimization literature line search and Levenberg-Marquardt method are seldom applied concurrently. Interestingly, in recent studies of Newton methods for neural networks, both techniques are considered. Our preliminary investigation in Section VI of supplementary materials shows that using Levenberg-Marquardt method together with line search is very helpful, but more detailed studies can be a future research issue.

In Algorithm 3 we show a master-master implementation, so the same program is used at each partition. Some careful designs are needed to ensure that all partitions get consistent information. For example, we can use the same random seed to ensure that at each iteration all partitions select the same set S_k in constructing the subsampled

Gauss-Newton matrix.

5 Analysis of the Proposed Algorithm

In this section, we analyze Algorithm 3 on the memory requirement, the computational cost, and the communication cost. We assume that the full training set is used. If the subsampled Hessian method in Section 4.3 is applied, then in the Jacobian calculation and the Gauss-Newton matrix vector product the "l" term in our analysis should be replaced by the subset size $|S|$.

5.1 Memory Requirement at Each Partition

Assume the partition corresponds to the neuron sub-groups T_{m-1} at layer $m-1$ and T_m at layer m. We then separately consider the following situations.

1. Local weight matrix: Each partition must store the local weight matrix.

   $$w_{tj}^m,\ \forall t \in T_{m-1}, \text{and } \forall j \in T_m.$$

   If T_{m-1} is the first neuron sub-group of layer $m-1$, it also needs to store

   $$b_j^m,\ \forall j \in T_m.$$

   Therefore, the memory usage at each partition for the local weight matrix is proportional to

   $$|T_{m-1}| \times |T_m| + |T_m|.$$

2. Function evaluation: From Section 3.2, we must store part of $\boldsymbol{z}^{m-1,i}$ and $\boldsymbol{z}^{m,i}$ vectors.[4] The memory usage at each partition is

$$l \times (|T_{m-1}| + |T_m|). \tag{67}$$

3. Gradient evaluation: First, we must store

$$\frac{\partial f}{\partial w^m_{tj}} \text{ and } \frac{\partial f}{\partial b^m_j}, t \in T_{m-1}, j \in T_m$$

after the gradient evaluation. Second, for the backward process, from (45), we must store

$$\frac{\partial \xi_i}{\partial z^{m-1,i}_t},\ \forall t \in T_{m-1},\ \forall i \ \text{ and } \ \frac{\partial \xi_i}{\partial z^{m,i}_j},\ \forall j \in T_m,\ \forall i.$$

Therefore, the memory usage in each partition is proportional to

$$(|T_{m-1}| \times |T_m| + |T_m|) + l \times (|T_{m-1}| + |T_m|). \tag{68}$$

4. Jacobian evaluation: From the discussion in Section 3.3.1, the memory consumption is proportional to

$$l \times n_L \times (|T_{m-1}| + |T_m|). \tag{69}$$

In summary, the memory bottleneck is on terms that are related to the number of instances. To reduce the memory use, we have considered a technique in Section 4.3 to replace the term l in (69) with a smaller subset size $|S^k|$. We will further discuss a technique to reduce the memory consumption in Section 6.1.

[4] Note that the same vector is used to store the $\boldsymbol{s}$ vector before it is transformed to $\boldsymbol{z}$ by the activation function.

5.2 Computational Cost

We analyze the computational cost at each partition. For the sake of simplicity, we make the following assumptions.

- At the mth layer neurons are evenly split to several sub-groups, each of which has $|T_m|$ elements.
- Calculating the activation function $\sigma(s)$ needs 1 operation.

The following analysis is for a partition between layers $m-1$ and m.

1. Function evaluation: From Algorithm 1, after $s_t^{m-1,i},\ i=1,\ldots,l,\ t\in T_{m-1}$ are available, we must calculate (19) and (20). The dominant one is (20), so the computational cost of function evaluation is

$$\mathcal{O}(l\times|T_m|\times|T_{m-1}|). \tag{70}$$

2. Gradient evaluation: Assume that the current partition has received $\partial\xi_i/\partial z_j^{m,i},\ i=1,\ldots,l,\ j\in T_m$. From (44), we calculate

$$\begin{aligned}\frac{\partial f}{\partial w_{tj}^m} &= \frac{1}{C}w_{tj}^m+\frac{1}{l}\sum_{i=1}^{l}\frac{\partial\xi_i}{\partial w_{tj}^m}\\ &= \frac{1}{C}w_{tj}^m+\frac{1}{l}\sum_{i=1}^{l}\frac{\partial\xi_i}{\partial z_j^{m,i}}\sigma'(s_j^{m,i})z_t^{m-1,i},\ \forall t\in T_{m-1},\ \forall j\in T_m,\end{aligned}$$

which costs

$$\mathcal{O}(l\times|T_m|\times|T_{m-1}|).$$

Then for the *reduce* operation in (45), calculating the local sum

$$\sum_{j\in T_m}\frac{\partial\xi_i}{\partial z_j^{m,i}}\sigma'(s_j^{m,i})w_{tj}^m,\ i=1,\ldots,l,\ t\in T_{m-1}.$$

has a similar cost. Thus the computational cost of gradient evaluation is

$$\mathcal{O}(l \times |T_m| \times |T_{m-1}|). \tag{71}$$

3. Jacobian evaluation: From (46) and (47) in Algorithm 2, the computational cost is

$$\mathcal{O}(n_L \times l \times |T_m| \times |T_{m-1}|). \tag{72}$$

4. Gauss-Newton matrix-vector products: Following (57) in Section 4.2, , the computational cost for Gauss-Newton matrix vector products is

$$\#\,\text{CG iterations} \times (l \times (|T_{m-1}| \times |T_m| + n_L \times |T_m|))\,. \tag{73}$$

From (70)-(73), we can derive the following conclusions.

1. The computational cost is proportional to the number of training data, the number of classes, and the number of variables in a partition.

2. In general, (72) and (73) dominate the computational cost. Especially, when the number of CG iterations is large, (73) becomes the bottleneck.

3. If the subsampling techniques in Section 4.3 is used, then l in (72)-(73) is replaced with the size of the subset. Therefore, the computational cost at each partition in a Newton iteration can be effectively reduced. However, the number of iterations may be increased.

4. The computational cost can be reduced by splitting neurons at each layer to as many sub-groups as possible. However, because each partition corresponds to a computing node, more partitions imply a higher synchronization cost. Further, the total number of neurons at each layer is different, so the size of each partition may significant vary, a situation that further worsens the synchronization issue.

5.3 Communication Cost

We have shown in Section 3.1 that by using diagonal blocks of the Gauss-Newton matrix, each partition conducts a CG procedure without communicating with others. However, communication cost still occurs for function, gradient, and Jacobian evaluation. We discuss details for the Jacobian evaluation because the situation for others is similar.

To simplify the discussion we make the following assumptions.

1. At the mth layer neurons are evenly split to several sub-groups, each of which has $|T_m|$ elements. Thus the number of neuron sub-groups at layer m is $n_m/|T_m|$.

2. Each partition sends or receives one message at a time.

3. Following Barnett et al. (1994), the time to send or receive a vector $\boldsymbol{v}$ is

$$\alpha + \beta \times |\boldsymbol{v}|,$$

where $|\boldsymbol{v}|$ is the length of $\boldsymbol{v}$, α is the start-up cost of a transfer and β is the transfer rate of the network.

4. The time to add a vector $\boldsymbol{v}$ and another vector of the same size is

$$\gamma \times |\boldsymbol{v}|.$$

5. Operations (including communications) of independent groups of nodes can be conducted in parallel. For example, the two trees in Figure IV.3 of supplementary materials involve two independent sets of partitions. We assume that the two *reduce* operations can be conducted in parallel.

From (36), for partitions between layers $m-1$ and m that correspond to the same neuron sub-group T_{m-1} at layer $m-1$, the *reduce* operation on $\partial z_u^{L,i}/\partial z_t^{m-1,i},\ u = 1,\ldots,n_L,\ t \in T_{m-1},\ i = 1,\ldots,l$ sums up

$$\frac{n_m}{|T_m|} \text{ vectors of } l \times n_L \times |T_{m-1}| \text{ size.}$$

For example, the layer 2 in Figure 2 is split to three groups A_2, B_2 and C_2, so for the sub-group A_1 in layer 1, three vectors from (A_1, A_2), (A_1, B_2) and (A_1, C_2) are reduced. Following the analysis in Pješivac-Grbović et al. (2007), the communication cost for the *reduce* operation is

$$\mathcal{O}(\lceil(\log_2(\frac{n_m}{|T_m|})\rceil \times (\alpha + (\beta+\gamma) \times (l \times n_L \times |T_{m-1}|)) \,. \tag{74}$$

Note that between layers $m-1$ and m

$$\frac{n_{m-1}}{|T_{m-1}|} \textit{ reduce} \text{ operations}$$

are conducted and each takes the communication cost shown in (74). However, by our assumption they can be fully parallelized.

The reduced vector of size $l \times n_L \times |T_{m-1}|$ is then broadcasted to $n_{m-2}/|T_{m-2}|$ partitions. Similar to (74), the communication cost is

$$\mathcal{O}(\lceil(\log_2(\frac{n_{m-2}}{|T_{m-2}|})\rceil \times (\alpha + \beta \times (l \times n_L \times |T_{m-1}|))). \tag{75}$$

The γ factor in (74) does not appear here because we do not need to sum up vectors.

Therefore, the total communication cost of the Jacobian evaluation is the sum of (74) and (75). We can make the following conclusions.

1. The communication cost is proportional to the number of training instances as well as the number of classes.

2. From (74) and (75), a smaller $|T_{m-1}|$ reduces the communication cost. However, we can not split neurons at each layer to too many groups because of the following reasons. First, we assumed earlier that for independent sets of partitions, their operations including communication within each set can be fully parallelized. In practice, the more independent sets the higher synchronization cost. Second, when there are too many partitions the block diagonal matrix in (49) may not be a good approximation of the Gauss-Newton matrix.

6 Other Implementation Techniques

In this section, we discuss additional techniques implemented in the proposed algorithm.

6.1 Pipeline Techniques for Function and Gradient Evaluation

The discussion in Section 5 indicates that in our proposed method the memory requirement, the computational cost and the communication cost all linearly increase with the number of data. For the product between the Gauss-Newton matrix and a vector, we have considered using subsampled Gauss-Newton matrices in Section 4.3 to effectively reduce the cost. To avoid that function and gradient evaluations become the bottleneck, here we discuss a pipeline technique.

The idea follows from the fact that in (4)

$$\xi_i, \forall i$$

are independent from each other. The situation is the same for

$$(J^i)^T \nabla_{\boldsymbol{z}^{L,i}} \xi(\boldsymbol{z}^{L,i}; \boldsymbol{y}^i), \forall i$$

in (6). Therefore, in the forward (or the backward) process, once results related to an instance $\boldsymbol{x}^i$ are ready, they can be passed immediately to partitions in the next (or previous) layers. Here we consider a mini-batch implementation. Take the function evaluation as an example. Assume $\{1, \ldots, l\}$ is split to R equal-sized subsets $S_1, \ldots, S_R$. At a variable partition between layers $m-1$ and m, we showed earlier that local values in (20) are obtained for all instances $i = 1, \ldots, l$. Now instead we calculate

$$\sum_{t \in T_{m-1}} w_{tj}^m z_t^{m-1,i} + b_j^m, \ j \in T_m, \ i \in S_r.$$

The values are used to calculate

$$s_j^{m,i}, \ \forall i \in S_r.$$

By this setting we achieve better parallelism. Further, because we split $\{1, \ldots, l\}$ to subsets with the same size, the memory space allocated for a subset can be reused by another. Therefore, the memory usage is reduced by R folds.

6.2 Sparse Initialization

A well-known problem in training neural networks is the easy overfitting because of an enormous number of weights. Following the approach in Section 5 of Martens (2010), we implement the sparse initialization for the weights to train deep neural networks. For each neuron in the mth layer, among the n_{m-1} weights connected to it, we randomly assign several weights to have values from the $\mathcal{N}(0, 1)$ distribution. Other weights are kept zero.

We will examine the effectiveness of this initialization in Sections 8.2 and 8.3.

7 Existing Optimization Methods for Training Neural Networks

Besides Newton methods considered in this work, many other optimization methods have been applied to train neural networks. We briefly discuss the most commonly used one in this section.

7.1 Stochastic Gradient Methods

For deep neural networks, it is time-consuming to calculate the gradient vector because from (6), we must go through the whole training data set. Instead of using all data instances, stochastic gradient (SG) methods randomly choose an example $(\boldsymbol{y}^{i_k}, \boldsymbol{x}^{i_k})$ to derive the following sub-gradient vector to update the weight matrix.

$$\nabla f^{i_k}(\boldsymbol{\theta}^k) = \frac{\boldsymbol{\theta}^k}{C} + (J^{i_k})^T \nabla_{\boldsymbol{z}^{L,i_k}} \xi(\boldsymbol{z}^{L,i_k}; \boldsymbol{y}^{i_k}).$$

Algorithm 4 gives the standard setting of SG methods.

Assume that one epoch means the SG procedure goes through the whole training data set once. Based on the frequent updates of the weight matrix, SG methods can get a reasonable solution in a few epochs. Another advantage of SG methods is that Algorithm 4 is easy to implement. However, if the variance of the gradient vector for each instance is large, SG methods may have slow convergence. To address this issue, mini-batch SG method have been proposed to accelerate the convergence speed (e.g., Bottou, 1991; Dean et al., 2012; Ngiam et al., 2011; Baldi et al., 2014). Assume $S_k \subset \{1, \ldots, l\}$ is a subset of the training data. The sub-gradient vector can be as follows:

$$\nabla f^{S_k}(\boldsymbol{\theta}^k) = \frac{\boldsymbol{\theta}^k}{C} + \frac{1}{|S_k|} \sum_{i \in S_k} (J^i)^T \nabla_{\boldsymbol{z}^{L,i}} \xi(\boldsymbol{z}^{L,i}; \boldsymbol{y}^i).$$

However, when SG methods meet ravines which cause the particular dimension apparent to other dimensions, they are easier to drop to local optima. Polyak (1964) proposes using the previous direction with momentum as part of the current direction. This setting may decrease the impact of a particular dimension. Algorithm 5 gives details of a mini-batch SG method with momentum implemented in *Theano/Pylearn2* (Goodfellow et al., 2013).

Many other variants of SG methods have been proposed, but it has been shown (e.g., Sutskever et al., 2013) that the mini-batch SG with momentum is a strong baseline. Thus in this work we do not include other types of SG algorithms for comparison.

Unfortunately, both SG and mini-batch SG methods have a well known issue in choosing a suitable learning rate and a momentum coefficient for different problems. We will conduct some experiments in Section 8.

8 Experiments

We consider the following data sets for experiments. All except Sensorless come with training and test sets. We split Sensorless as described below.

- HIGGS: This binary classification data set is from high energy physics applications. It is selected for our experiments because feedforward networks have been successfully applied (Baldi et al., 2014). Note that a scalar output y is enough to represent two classes in a binary classification problem. Based on this idea, we set $n_L = 1$, and have each $y^i \in \{-1, 1\}$. The predicted outcome is the first class if $y \geq 0$ and is the second class if $y < 0$. This data set is mainly used in Section 8.3 for a comparison with results in Baldi et al. (2014).

- Letter: This set is from the Statlog collection (Michie et al., 1994) and we scale values of each feature to be in $[-1, 1]$.

- MNIST: This data set for hand-written digit recognition (LeCun et al., 1998a) is widely used to benchmark classification algorithms. We consider a scaled version, where every feature value is divided by 255.

- Pendigits: This data set is originally from Alimoglu and Alpaydin (1996).

- Poker: This data set is from UCI machine learning repository (Lichman, 2013). It has been studied by, for example, Li (2010).

- Satimage: This set is from the Statlog collection (Michie et al., 1994) and we scale values of each feature to be in $[-1, 1]$.

- SensIT Vehicle: This data set, from Duarte and Hu (2004), includes signals from acoustic and seismic sensors in order to classify the different vehicles. We use the original version without scaling.

- Sensorless: This data set is from Paschke et al. (2013). We scale values of each feature to be in $[0, 1]$, and then conduct stratified random sampling to select $10,000$ instances to be the test set and the rest of the data to be the training set.

- SVHN: This data, originally from Google Street View images, consists of colored images of house numbers (Netzer et al., 2011). We scale the data set to $[0, 1]$ by considering the largest and the smallest feature values of the entire data set.

  $$M \equiv \max_i \max_p (\boldsymbol{x}_i)_p \text{ and } m \equiv \min_i \min_p (\boldsymbol{x}_i)_p.$$

  Then the pth element of $\boldsymbol{x}_i$ is changed to

  $$(\boldsymbol{x}_i)_p \leftarrow \frac{(\boldsymbol{x}_i)_p - m}{M - m}.$$

- USPS: This data set, from Hull (1994), is used on recognizing handwritten ZIP codes and we scale values of each feature to be in $[-1, 1]$.

All data sets, with statistics in Table 1, are publicly available.[5] Detailed settings for each data such as the network structure are given in Table 2. How to decide a suitable network structure is beyond the scope of this work, but if possible, we follow the setting in earlier works. For example, we consider the structure in Wan et al. (2013) for MNIST and Neyshabur et al. (2015) for SVHN. From Table 2, the model used for SVHN is the largest. If the number of neurons in each layer is further increased, then the model must be stored in different machines.

We give parameters used in our algorithm. For the sparse initialization discussed in Section 6.2, among n_{m-1} weights connected to a neuron in layer m, $\lceil \sqrt{n_{m-1}} \rceil$ are selected to have non-zero values. For the CG stopping condition (52), we set $\sigma = 0.001$ and $\text{CG}_{\text{max}} = 250$. Further, the minimal number of CG steps run at each partition, CG_{min}, is set to be 3. For the implementation of the Levenberg-Marquardt method, we

[5] All data sets used can be found at `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`.

Table 1: Summary of the data sets: n_0 is the number of features, l is the number of training instances, l_t is the number of testing instances, and K is the number of classes.

Data set	n_0	l	l_t	K
Letter	16	15,000	5,000	26
MNIST	784	60,000	10,000	10
Pendigits	16	7,494	3,498	10
Poker	10	25,010	1,000,000	10
Satimage	36	4,435	2,000	6
SensIT Vehicle	100	78,823	19,705	3
Sensorless	48	48,509	10,000	11
SVHN	3,072	73,257	26,032	10
USPS	256	7,291	2,007	10
HIGGS	28	10,500,000	500,000	2

set the initial $\lambda_1 = 1$. The (drop, boost) constants in (66) are $(2/3, 3/2)$. For solving (61) to get the update direction after the CG procedure, we set $\varepsilon = 10^{-5}$ in (63).

8.1 Analysis of Distributed Newton Methods

We have proposed several techniques to improve upon the basic implementation of the Newton method in a distributed environment. Here we investigate their effectiveness by considering the following methods. Note that because of the high memory consumption of some larger sets, we always implement the subsampled Hessian Newton method

Table 2: Details of the distributed network for each data. Sampling rate is the percentage of training data used to calculate the subsampled Gauss-Newton matrix.

Data set	Sampling rate	Network structure	Split structure	# partitions
Letter	20%	16-300-300-300-300-26	1-2-1-1-1-1	7
MNIST	20%	784-800-800-10	1-1-3-1	7
Pendigits	20%	16-300-300-10	1-2-2-1	8
Poker	20%	10-200-200-200-10	1-1-1-1-1	4
SensIT Vehicle	20%	100-300-300-3	1-2-2-1	8
Sensorless	20%	48-300-300-300-11	1-2-1-2-1	8
Satimage	20%	36-1000-500-6	1-2-2-1	8
SVHN	10%	3072-4000-4000-10	3-2-2-1	12
USPS	20%	256-300-300-10	1-2-2-1	8

discussed in Section 4.3.

1. *subsampled-GN*: we use the whole subsampled Gauss-Newton matrix defined in (58) to conduct the matrix-vector product in the CG procedure and then solve (61) to get the update direction after the CG procedure (Wang et al., 2015).

2. *diag*: it is the same as *subsampled-GN* except that only diagonal blocks of the subsampled Gauss-Newton matrix are used; see (60).

3. *diag* + *sync* 50%: it is the same as *diag* except that we consider the technique in Section 4.4 to reduce the synchronization time. We terminate the CG procedure when 50% of partitions have reached their local stopping conditions (64).

4. *diag* $+$ *sync* 25%: it is the same as *diag* $+$ *sync* 50% except that we terminate the CG procedure when 25% of partitions have reached their local stopping conditions (64).

For each of the above methods, we consider the following implementation details.

1. We set $C = l$ as the regularization parameter.

2. We run experiments on G1 type instances on Microsoft Azure and let each instance use only one core. If instances are not virtual machines on the same computer, our setting ensures that each variable partition corresponds to one machine.

3. To make the computational cost in each partition as balanced as possible, in our experiments we choose our partitions such that the maximum ratio between the numbers of variables ($|T_m| \times |T_{m-1}|$) among any two partitions is as low as possible. For example, in Pendigits, the largest partition has $150 \times 150 = 22,500$ weight variables, and the smallest partition has $150 \times 10 = 1,500$ weight variables, with their ratio being $22500/1500 = 15$. For most data sets, the ratio is between 10 and 100 but not lower because the numbers of classes is relatively small, making the number of variables in the partitions involving the output layer smaller than those in other partitions.

In Figure 4, we show the comparison results and have the following observations.

1. For test accuracy versus number of iterations, *subsampled-GN* in general has the fastest convergence rate. The reason should be that the direction in *subsampled-GN*

by solving the linear system (59) is closer to the full Newton direction than other methods, which consider further approximations of the Gauss-Newton matrix or the early termination of the CG procedure. However, the cost per iteration is high, so for training time we see that *subsampled-GN* may become worse than other approaches.

2. The early termination of the CG procedure can effectively reduce the cost per iteration. However, if we stop the CG procedure too early, the total training time may even increase. For example,

   $$\textit{diag} + \textit{sync}\ 25\%$$

   is generally the fastest in the beginning because of the least cost per iteration. It is still the fastest in the end for MNIST, Letter, USPS, Satimage, and Pendigits. However, it has the slowest final convergence for SensIT Vechicle, Poker, and Sensorless. Take the data set Poker as an example. As listed in Table 2, the variables are split into four partitions, and the CG procedure stops if one partition (i.e., 25% of the partitions) reaches its local stopping condition. This partition may have the lightest computational load or is the earliest one to start solving the local linear system.[6] Thus the other partitions may not have run enough CG iterations.

   The approach

   diag + sync 50%

   does not terminate the CG procedure that early. Overall we find that it is efficient

[6] Note that because of the backward process in Section 3.3, the partitions corresponding to the last two layers begin their CG procedures earlier than the others.

and stable. Therefore, in subsequent comparisons with stochastic gradient methods, we use it as the setting of our Newton method.

Because of the space consideration, we have evaluated only some techniques proposed in Section 4. For the following two techniques we leave details in Sections VI and VII of the supplementary materials.

1. In Section 4.3, we propose combining $\boldsymbol{d}^k$ and $\boldsymbol{d}^{k-1}$ as the update direction. We show that this technique is very effective.

2. We mentioned in Section 4.5 that line search and the Levenberg-Marquardt (LM) method may not be both needed. Our preliminary results show that the training speed is improved when both techniques are applied.

8.2 Comparison with Stochastic Gradient Methods and Support Vector Machines (SVM)

In this section, we compare our methods with SG methods and SVMs, which are popularly used for multi-class classification. Settings of these methods are described as follows.

1. *Newton*: for our method we use the setting *diag* + *sync* 50% considered in Section 8.1 and let $C = l$.

2. *SVM* (Boser et al., 1992): We consider the RBF kernel.

$$K(\boldsymbol{x}^i, \boldsymbol{x}^j) = e^{-\gamma||\boldsymbol{x}^i - \boldsymbol{x}^j||^2},$$

where $\boldsymbol{x}^i$ and $\boldsymbol{x}^j$ are two data instances, and γ is the kernel parameter chosen by users. Note that *SVM* solves an optimization problem similar to (4), so the regularization parameter, C, must be decided as well. We conduct five-fold cross validation on the training set to select the best $C \in \{2^{-5}l, 2^{-3}l, \ldots, 2^{15}l\}$ and the best $\gamma \in \{2^{-15}, 2^{-13}, \ldots, 2^{3}\}$.[7] We use the library *LIBSVM* (Chang and Lin, 2011) for training and prediction.

3. *SG*: We use the code from Baldi et al. (2014), which implements Algorithm 5. The objective function is the same as (4).[8] The network structure for each data set is identical to the corresponding one used in *Newton*, and we also set the regularization parameter $C = l$. The major modification we make is that we replace their activation functions with ours. In Baldi et al. (2014), the authors use $\tanh$ as their activation functions in layers $1, \ldots, L-1$ and the sigmoid function in layer L, while in our experiments of Newton methods in Section 8.1, we use the sigmoid function in layers $1, \ldots, L-1$ and the linear function in layer L. The initial learning rate is selected from $\{0.05, 0.025, 0.01, 0.005, 0.002, 0.001\}$ by the five-fold cross validation. After the initial learning rate has been selected, we conduct the training process to generate a model for the prediction on the test set.

As regards the stopping condition for the training process, we terminate the *Newton*

[7] Here we consider an SVM formulation represented as (2). In the form considered in *LIBSVM*, the two terms C and $1/l$ are combined together, so C/l is the actual parameter to be selected. For SVHN because of the lengthy time for parameter selection, we selected only $10,000$ instances by stratified sampling to conduct the five-fold cross validation.

[8] Following Baldi et al. (2014), we regularized only the weights but not the biases. Through several experiments, we found that the performance is similar with/without the regularization of the biases.

method at the 100th iteration. For *SG*, it terminates after a minimal number of epochs have been conducted and the objective function value on the validation set does not improve much within the last N epochs (see Algorithm 5). To implement the stopping condition, for *SG* we split the input training set into 90% for training and 10% for validation.[9] For *SVM*, we use the default stopping condition of *LIBSVM*.[10]

Here we also investigate the effect of the initialization by considering the following two settings.

1. The sparse initialization discussed in Section 6.2.

2. The dense initialization discussed in Baldi et al. (2014). The initial weights are drawn from the normal distribution $\mathcal{N}(0, 0.1^2)$ for the first layer, $\mathcal{N}(0, 0.001^2)$ for the output layer, and $\mathcal{N}(0, 0.05^2)$ for other hidden layers. The biases are initialized as zeros.

To make a fair comparison, for each setting, *Newton* and *SG* are trained with the same initial weights and biases.

We present a comparison on test accuracy in Table 3, and make the following observations.

1. For neural networks, the sparse initialization usually results in better accuracy than the dense initialization does. The difference can be huge in some cases, such as

[9] Note that in the CV procedure we also need a stopping condition in training each sub-problem. We do an 80-20 split of every four folds of data so that the 20% of data are used to implement the stopping condition.

[10] *LIBSVM* terminates when the violation of the optimality condition calculated based on the gradient is smaller than a tolerance.

training using *SG* on the data set Letter. The low accuracy of the densely initialized *SG* on Letter may be because of the poor differentiation between neurons in dense initialization (Martens, 2010). Other possible causes include the vanishing gradient problem (Bengio et al., 1994), or that the activations are trapped in the saturation regime of the sigmoid function (Glorot and Bengio, 2010). Note that the impact of the initialization scheme on the *Newton* method is much weaker.[11]

2. Between *SG* and *Newton*, if sparse initialization is used, we can see that *Newton* generally gives higher accuracy.

3. If sparse initialization is used, our *Newton* method for training neural networks gives similar or higher accuracy than *SVM*. In particular, the results are much better for Poker and SVHN.

We compare our results on MNIST with those reported in earlier works. Wan et al. (2013) use a fully connected neural network with two 800-neuron hidden layers to derive an error rate 1.36%, under the setting of dense initialization,[12] sigmoid activations, and the dropout technique. By the same network structure and the same activation function, our error rate is 1.34% at the 100th iteration.

For SVHN, we compare our results with Neyshabur et al. (2015), in which the same network structure as ours is adopted, except that they use ReLU activations in the hidden layers. They choose the cross-entropy as their objective function, and utilize the

[11] We observe similar phenomena in the experiments with HIGGS later in Section 8.3. See Table 5.

[12] In Wan et al. (2013), the initial weights are drawn from $\mathcal{N}(0, 0.01)$, slightly different from the dense initialization we use.

dropout regularization. Under dense initialization,[13] they train their network with the Path-SGD method, which uses a proximal gradient method to solve the optimization problem. They report an accuracy slightly below 87% (see their Figure 3), while the accuracy obtained by our *Newton* method with sparse initialization is 83.12%.

For Poker, we note that Li (2010) uses *abc-logitboost* to obtain a slightly higher accuracy, but his setting is different from ours. He expands the training set by including half of the test set, with the remaining half of the test set used for evaluation.

An issue found out in our experiments is that *SG* is sensitive to the initial learning rate. In Table 4, we present the test accuracy of *SG* under different initial rates for the Poker problem. Clearly an inappropriate initial learning rate can lead to much worse accuracy.

8.3 Detailed Investigation on the HIGGS Data

We compare AUC values obtained by our *Newton* and *SG* implementations with those reported in Baldi et al. (2014) on HIGGS. In our method, the sampling rate for calculating the subsampled Gauss-Newton matrix is set to be 1%. Following the setting in Section 8.2, we consider two initializations (dense and sparse). Then for each type of initialization, both *SG* and *Newton* start with the same initial weights and biases. Note that our SG results are different from those in Baldi et al. (2014) because we use different activation functions and initial values for weights and biases.[14] Because of

[13] In Neyshabur et al. (2015), the initial weights w_{tj}^m are drawn from $\mathcal{N}(0, 1/n_{m-1})$, slightly different from the dense initialization we use.

[14] Their initialization setting is the same as our dense initialization, but the values used by them are not available.

Table 3: Test accuracy of *SVM*, *Newton* and *SG*. For *SVM*, we also show parameters (C, γ) used. For *SG*, we show (the initial learning rate, number of epochs to reach the stopping criterion). The bold-faced entries indicate the best accuracy obtained using the neural networks.

	SVM	Neural Networks			
		Dense Initialization		Sparse Initialization	
		Newton	*SG*	*Newton*	*SG*
Letter	97.90% ($2^7l, 2$)	90.26%	8.02% ($0.025, 245$)	**96.68%**	96.28% ($0.002, 906$)
MNIST	98.57% ($2^3l, 2^{-5}$)	98.52%	98.26% ($0.002, 801$)	**98.66%**	98.33% ($0.002, 909$)
Pendigits	98.06% ($2^7l, 2^{-15}$)	97.51%	97.71% ($0.001, 513$)	**97.83%**	97.71% ($0.002, 1179$)
Poker	58.78% ($2^{-1}l, 2^{-3}$)	99.25%	99.24% ($0.005, 316$)	99.25%	**99.29%** ($0.002, 895$)
Satimage	91.85% ($2l, 2$)	89.35%	82.00% ($0.01, 246$)	**89.85%**	89.35% ($0.001, 1402$)
SensIT Vehicle	83.90% ($2l, 2^{-1}$)	**85.16%**	83.34% ($0.01, 311$)	84.60%	84.00% ($0.01, 296$)
Sensorless	99.83% ($2^5l, 2^3$)	97.19%	97.64% ($0.01, 412$)	**99.05%**	98.24% ($0.005, 382$)
SVHN	74.54% ($2^5l, 2^{-7}$)	80.96%	82.99% ($0.001, 986$)	**83.12%**	82.67% ($0.001, 720$)
USPS	95.32% ($2^5l, 2^{-5}$)	95.17%	94.97% ($0.025, 395$)	**95.27%**	95.07% ($0.001, 1617$)

resource constraints, we did not conduct a validation procedure to select SG's initial learning rate. Instead, we used the learning rate 0.05 by following Baldi et al. (2014). The results are shown in Table 5 and we can see that the Newton method often gives the best AUC values.

In Section 8.2 we have mentioned that *SG*'s performance may be sensitive to the initial learning rate. The poor results of *SG* in Table 5 might be because we did not conduct a selection procedure. Thus we decide to investigate the effect of the initial learning rate on the AUC value with the network structure 28-300-300-1 used in the

Table 4: Test accuracy on Poker using *SG* with different initial learning rates η. Dense initialization is used. Note that although $\eta = 0.005$ does not yield the highest test accuracy, it was selected for experiments in Table 3 because of giving the highest CV accuracy.

Initial learning rate η	0.05	0.025	0.01	0.005	0.002	0.001
Test accuracy	68.83%	98.81%	99.24%	99.24%	99.24%	99.25%

earlier experiment in Table 5. To compare the running time, both *SG* and *Newton* run on the same G3 type machine with 8 cores in Microsoft Azure. The results of the AUC values versus the number of iterations and the training time are shown in Figure 5. We clearly see again that the performance of *SG* depends significantly on the initial learning rate. Our experiments indicate that while *SG* can yield good performances under suitable parameters, the parameter selection procedure is essential. In contrast, Newton methods are more robust because we do not need to fine tune their parameters.

9 Discussion and Conclusions

For the future works, we list the following directions.

1. It is important to extend the proposed method for other types of neural networks. For example, convolutional neural networks (CNNs) are popular for computer vision applications (e.g., Krizhevsky et al., 2012; Simonyan and Zisserman, 2014). Because

Table 5: A comparison between the AUC obtained by *SG* and that by the distributed *Newton* on the HIGGS data set. We list the results in Baldi et al. (2014) as a reference, where “NA” means that the result is not reported. See explanation in Section 8.3 about the different results between our SG and Baldi et al.’s.

Network	Split	Dense Initialization		Sparse Initialization		Baldi et al. (2014)
		Newton	*SG*	*Newton*	*SG*	
28-300-1	2-2-1	0.843	0.469	0.843	0.684	0.816
28-600-1	2-3-1	0.849	0.501	0.849	0.759	NA
28-1000-1	2-4-1	0.851	0.500	0.853	0.734	0.841
28-2000-1	2-8-1	0.853	0.500	0.855	0.504	0.842
28-300-300-1	2-2-1-1	0.851	0.530	0.860	0.825	NA
28-300-300-300-1	2-2-2-1-1	0.867	0.482	0.879	0.849	0.850
28-300-300-300-300-1	2-2-2-2-1-1	0.867	0.504	0.875	0.848	0.872

CNNs generally have fewer weights per layer, our method has the potential to train deep networks for large-scale image classification.

2. Instead of the Gauss-Newton matrix, we may consider other ways to use or approximate the Hessian such as the recent works by He et al. (2016).

3. For results in Tables 3 and 5, we consider the model after running 100 Newton iterations. An advantage of Newton over stochastic gradient is that we can apply a gradient-based stopping condition. We plan to investigate its practical use.

4. It is known that using suitable preconditioners can effectively reduce the number of

CG steps in solving a linear system. Studies of applying preconditioned CG methods in training neural networks include, for example, Chapelle and Erhan (2011). We plan to investigate how to apply preconditioning in our distributed framework.

In summary, in this paper we proposed novel techniques to implement distributed Newton methods for training large-scale neural networks, and achieved both data and model parallelisms.

Acknowledgements

This work was supported in part by MOST of Taiwan via the grant 105-2218-E-002-033 and Microsoft via Azure for Research programs.

References

Alimoglu, F. and Alpaydin, E. (1996). Methods of combining multiple classifiers based on different representations for pen-based handwritten digit recognition. In *Proceedings of the Fifth Turkish Artificial Intelligence and Artificial Neural Networks Symposium*.

Baldi, P., Sadowski, P., and Whiteson, D. (2014). Searching for exotic particles in high-energy physics with deep learning. *Nature Communications*, 5.

Barnett, M., Gupta, S., Payne, D. G., Shuler, L., van De Geijn, R., and Watts, J. (1994). Interprocessor collective communication library (InterCom). In *Proceedings of the Scalable High-Performance Computing Conference*, pages 357–364.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166.

Bian, Y., Li, X., Cao, M., and Liu, Y. (2013). Bundle CDN: a highly parallelized approach for large-scale l1-regularized logistic regression. In *Proceedings of European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML/ PKDD)*.

Boser, B. E., Guyon, I., and Vapnik, V. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, pages 144–152. ACM Press.

Bottou, L. (1991). Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nımes*, 91(8).

Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT 2010*, pages 177–186.

Byrd, R. H., Chin, G. M., Neveitt, W., and Nocedal, J. (2011). On the use of stochastic Hessian information in optimization methods for machine learning. *SIAM Journal on Optimization*, 21(3):977–995.

Chang, C.-C. and Lin, C.-J. (2011). LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27:1–27:27. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

Chapelle, O. and Erhan, D. (2011). Improved preconditioner for Hessian free optimization. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*.

Ciresan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. (2010). Deep, big, simple neural nets for handwritten digit recognition. *Neural Computation*, 22:3207–3220.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M., Senior, A. W., Tucker, P. A., et al. (2012). Large scale distributed deep networks. In *Advances in Neural Information Processing Systems (NIPS) 25*.

Duarte, M. and Hu, Y. H. (2004). Vehicle classification in distributed sensor networks. *Journal of Parallel and Distributed Computing*, 64(7):826–838.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 249–256.

Goodfellow, I. J., Warde-Farley, D., Lamblin, P., Dumoulin, V., Mirza, M., Pascanu, R., Bergstra, J., Bastien, F., and Bengio, Y. (2013). Pylearn2: a machine learning research library.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)*.

He, X., Mudigere, D., Smelyanskiy, M., and Takáč, M. (2016). Large scale distributed Hessian-free optimization for deep neural network. arXiv preprint arXiv:1606.00511.

Hinton, G. E., Deng, L., Yu, D., Dahl, G., rahman Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., and Kingsbury, B. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97.

Hull, J. J. (1994). A database for handwritten text recognition research. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(5):550–554.

Kiros, R. (2013). Training neural networks with stochastic Hessian-free optimization. arXiv preprint arXiv:1301.3641.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998a). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324. MNIST database available at `http://yann.lecun.com/exdb/mnist/`.

LeCun, Y., Bottou, L., Orr, G. B., and Müller, K.-R. (1998b). Efficient backprop. In *Neural Networks, Tricks of the Trade*, Lecture Notes in Computer Science LNCS 1524. Springer Verlag.

Li, P. (2010). An empirical evaluation of four algorithms for multi-class classification: Mart, abc-mart, robust logitboost, and abc-logitboost. arXiv preprint arXiv:1001.1020.

Lichman, M. (2013). UCI machine learning repository.

Mahajan, D., Keerthi, S. S., and Sundararajan, S. (2017). A distributed block coordinate descent method for training l1 regularized linear classifiers. *Journal of Machine Learning Research*, 18(91):1–35.

Martens, J. (2010). Deep learning via Hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*.

Martens, J. and Sutskever, I. (2012). Training deep and recurrent networks with Hessian-free optimization. In *Neural Networks: Tricks of the Trade*, pages 479–535. Springer.

Michie, D., Spiegelhalter, D. J., Taylor, C. C., and Campbell, J., editors (1994). *Machine learning, neural and statistical classification*. Ellis Horwood, Upper Saddle River, NJ, USA. Data available at `http://archive.ics.uci.edu/ml/machine-learning-databases/statlog/`.

Moritz, P., Nishihara, R., Stoica, I., and Jordan, M. I. (2015). SparkNet: Training deep networks in Spark. *arXiv preprint arXiv:1511.06051*.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. (2011). Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*.

Neyshabur, B., Salakhutdinov, R. R., and Srebro, N. (2015). Path-SGD: Path-normalized optimization in deep neural networks. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28*, pages 2422–2430.

Ngiam, J., Coates, A., Lahiri, A., Prochnow, B., Le, Q. V., and Ng, A. Y. (2011). On optimization methods for deep learning. In *Proceedings of the 28th International Conference on Machine Learning*, pages 265–272.

Paschke, F., Bayer, C., Bator, M., Mönks, U., Dicks, A., Enge-Rosenblatt, O., and Lohweg, V. (2013). Sensorlose zustandsüberwachung an synchronmotoren. In *Proceedings of Computational Intelligence Workshop*.

Pearlmutter, B. A. (1994). Fast exact multiplication by the Hessian. *Neural Computation*, 6(1):147–160.

Pješivac-Grbović, J., Angskun, T., Bosilca, G., Fagg, G. E., Gabriel, E., and Dongarra, J. J. (2007). Performance analysis of MPI collective operations. *Cluster Computing*, 10:127–143.

Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17.

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

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML)*, pages 1139–1147.

Taylor, G., Burmeister, R., Xu, Z., Singh, B., Patel, A., and Goldstein, T. (2016). Training neural networks without gradients: A scalable ADMM approach. In *Proceedings of The Thirty Third International Conference on Machine Learning*, pages 2722–2731.

Thakur, R., Rabenseifner, R., and Gropp, W. (2005). Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66.

Wan, L., Zeiler, M., Zhang, S., LeCun, Y., and Fergus, R. (2013). Regularization of neural networks using DropConnect. In *Proceedings of the 30th International Conference on Machine Learning (ICML)*, pages 1058–1066.

Wang, C.-C., Huang, C.-H., and Lin, C.-J. (2015). Subsampled Hessian Newton methods for supervised learning. *Neural Computation*, 27:1766–1795.

Zinkevich, M., Weimer, M., Smola, A., and Li, L. (2010). Parallelized stochastic gradient descent. In Lafferty, J., Williams, C. K. I., Shawe-Taylor, J., Zemel, R., and Culotta, A., editors, *Advances in Neural Information Processing Systems 23*, pages 2595–2603.

Algorithm 1 Function evaluation in a distributed system

```
Let $T_{m-1}$ and $T_m$ be the subsets of neurons at the $(m-1)$th and $m$th layers corresponding to the current partition.
if $m = 1$ then
    Read $s_t^{m-1,i}$ from input, where $i = 1, \ldots, l$, and $t \in T_{m-1}$.
else
    Wait for $s_t^{m-1,i}$, $i = 1, \ldots, l,\ t \in T_{m-1}$.
    Calculate $z_t^{m-1,i}$ by (19).
end if
After calculating (20), run an allreduce operation to have
    $$s_j^{m,i},\ i = 1, \ldots, l \text{ and } j \in T_m, \qquad (27)$$
available in all partitions between layers $m-1$ and $m$ corresponding to $T_m$.
if $T_{m-1}$ is the first neuron sub-group of layer $m-1$ then
    if $m < L$ then
        We broadcast values in (27) to partitions between layers $m$ and $m+1$
        corresponding to the neuron subgroup $T_m$; see the description after (23)
    else
        Calculate
            $$\sum_{i=1}^{l} \sum_{j \in T_L} \xi(z_j^{L,i}; y_j^i) + \text{accumulated regularization terms}$$
        If $T_L$ is the first neuron sub-group of layer $L$, run a reduce operation
        to get the final $f$; see (24).
    end if
end if
```

Algorithm 2 Calculation of $\partial z_u^{L,i}/\partial s_j^{m,i},\ u = 1,\ldots,n_L,\ j = 1,\ldots,|T_m|$ in a distributed system.

```
Let $T_{m-1}$ and $T_m$ be the subsets of neurons at the $(m-1)$th and $m$th layers corresponding to the current partition.
if $m = L$ then
    Calculate
    $\frac{\partial z_u^{L,i}}{\partial z_j^{m,i}} = \begin{cases} 2(z_u^{L,i} - y_u^i) & \text{if } j = u, \\ 0 & \text{if } j \neq u, \end{cases},\ u = 1,\ldots,n_L,\ i = 1,\ldots,l,$ and $j \in T_m$.
else
    Wait for $\partial z_u^{L,i}/\partial z_j^{m,i}$, $u = 1,\ldots,n_L$, $i = 1,\ldots,l$, and $j \in T_m$.
end if
Calculate
    $\frac{\partial z_u^{L,i}}{\partial s_j^{m,i}} = \frac{\partial z_u^{L,i}}{\partial z_j^{m,i}} \sigma'(s_j^{m,i}),\ u = 1,\ldots,n_L,\ i = 1,\ldots,l,$ and $j \in T_m$.    (46)
if $m > 1$ then
    Calculate the local sum
        $\sum_{j \in T_m} \frac{\partial z_u^{L,i}}{\partial s_j^{m,i}} w_{tj}^m,\ t \in T_{m-1}$    (47)
    and do the *reduce* operation to obtain
        $\frac{\partial z_u^{L,i}}{\partial z_t^{m-1,i}},\ u = 1,\ldots,n_L,\ i = 1,\ldots,l,$ and $t \in T_{m-1}$.    (48)
    if $T_m$ is the first neuron sub-group of layer $m$ then
        Broadcast values in (48) to partitions between layers $m-2$ and $m-1$
        corresponding to the neuron sub-group $T_{m-1}$ at layer $m-1$;
        see the description after (38).
    end if
end if
```

Algorithm 3 A distributed subsampled Hessian Newton method with variable partition.

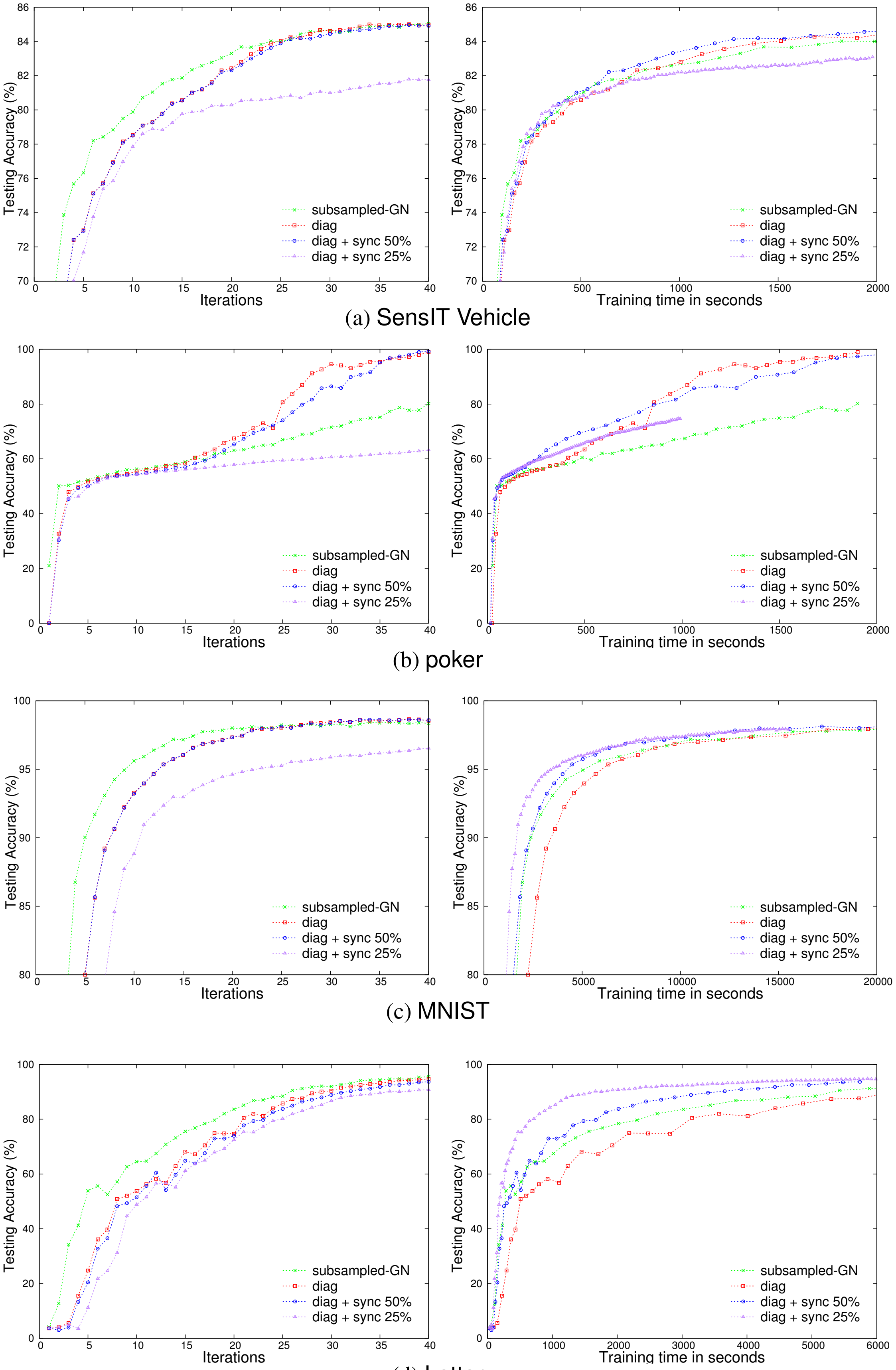
```
Given $\epsilon \in (0,1)$, $\lambda_1$, $\sigma \in (0,1)$, $\eta \in (0,1)$, CG$_{\max}$, CG$_{\min}$, and $r \in (0,100]$.
Let $p$ be the index of the current partition and generate the initial local model vector $\boldsymbol{\theta}_p^1$.
Compute $f(\boldsymbol{\theta}^1)$.
for $k = 1, \ldots,$ do
    Choose a set $S_k \subset \{1, \ldots, l\}$.
    Compute $\boldsymbol{g}_p^k$ and $J_p^i, \forall i \in S_k$.
    Approximately solve the linear system in (60) by CG to obtain a direction $\boldsymbol{d}_p^k$ after
        $$||(\lambda_k \mathcal{I} + \frac{1}{C}\mathcal{I} + \frac{1}{|S_k|}\sum_{i=1}^{|S_k|}(J_p^i)^T B^i J_p^i)\boldsymbol{d}_p^k + \boldsymbol{g}_p^k|| \leq \sigma ||\boldsymbol{g}_p^k||$$
    is satisfied or $\#\text{CG}_p^k \geq$ CG$_{\max}$ or
        $\{$# partitions finished $\geq r\% \times P$ and $\#\text{CG}_p^k \geq$ CG$_{\min}\}$,
    where $\#\text{CG}_p^k$ is the number of CG iterations that have been run so far.
    Derive $\boldsymbol{d}_p^k = \beta_1 \boldsymbol{d}_p^k + \beta_2 \boldsymbol{d}_p^{k-1}$ by solving (61).
    $\alpha^k = 1$.
    while true do
        Update $\boldsymbol{\theta}_p^{k+1} = \boldsymbol{\theta}_p^k + \alpha^k \boldsymbol{d}_p^k$ and then compute $f(\boldsymbol{\theta}^{k+1})$.
        if $T_m$ and $T_{m-1}$ are the first neuron subgroups at layers $L$ and $L-1$, respectively, then
            if (65) is satisfied then
                Notify all partitions to stop.
            end if
        else
            Wait for the notification to stop.
        end if
        if the stop notification has been received then
            break;
        end if
        $\alpha^k = \alpha^k/2$.
    end while
    Update $\lambda_{k+1}$ based on (66).
end for
```

Algorithm 4 Standard stochastic gradient methods

```
Given a learning rate $\eta$.
for $k = 0, \ldots$ do
    Choose $i_k \in \{1, \ldots, l\}$.
    $\boldsymbol{\theta}^{k+1} = \boldsymbol{\theta}^k - \eta \nabla f^{i_k}(\boldsymbol{\theta}^k)$.
end for
```

Algorithm 5 Mini-batch stochastic gradient methods in *Theano/Pylearn2* (Goodfellow et al., 2013).

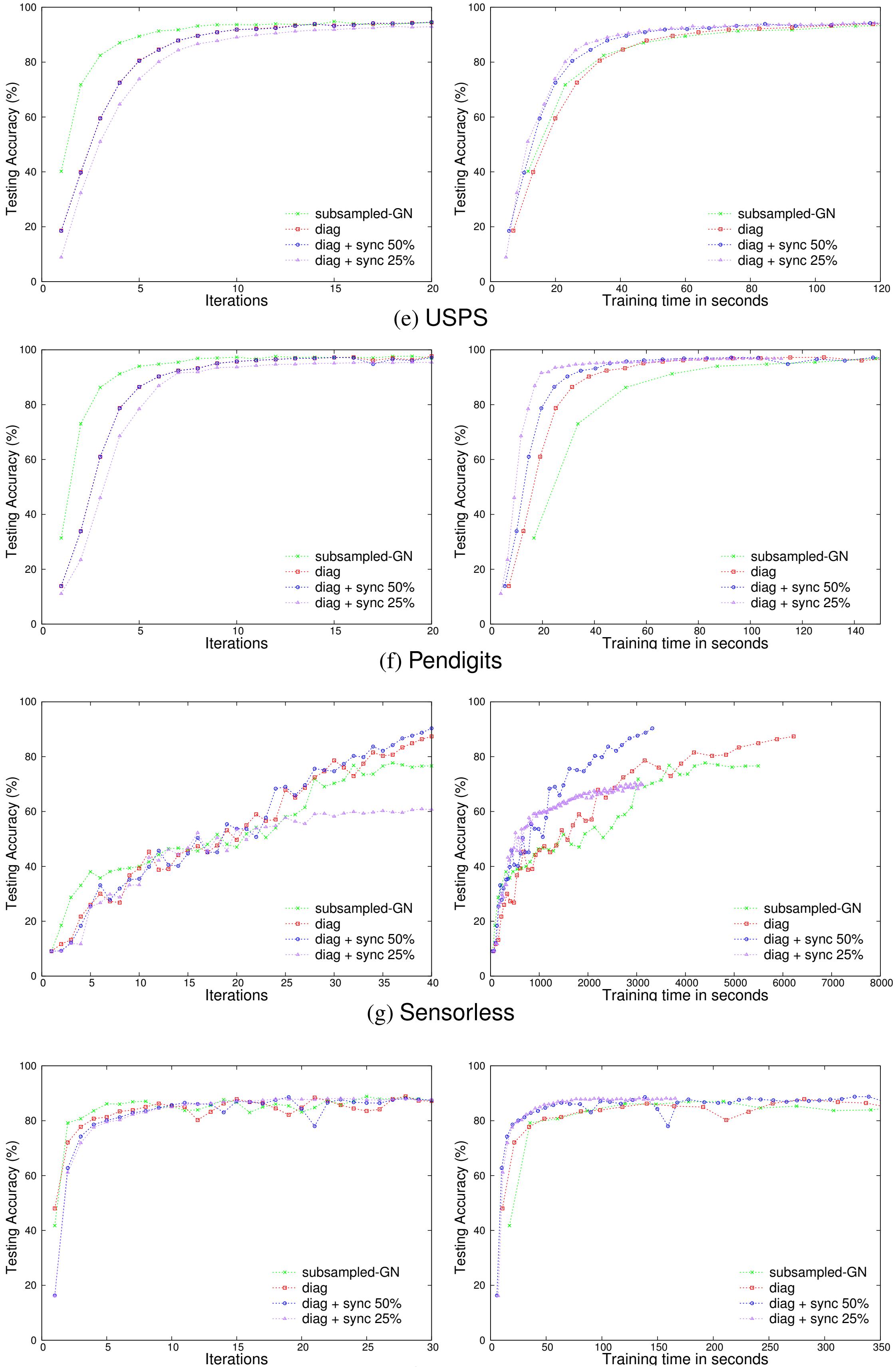
```
Given epoch = 0, min_epochs = 200, a learning rate $\eta$, a minimum learning rate
    $\eta_{\min} = 10^{-6}$, $\alpha = 0$, $r = 0$, $X = 10^{-5}$, $N = 10$, a batch size $b = |S_k| = 100$,
    an initial momentum $m_0 = 0.9$, a final momentum $m_f = 0.99$, an exponentially
    decay factor $\gamma = 1.0000002$, and an updating vector $\boldsymbol{v} \leftarrow \mathbf{0}$.
counter $\leftarrow N$.
lowest_value $\leftarrow \infty$.
while epoch < min_epochs or counter > 0 do
    Split the whole training data into $K$ disjoint subsets, $S_k,\ k = 1, \ldots, K$.
    $\alpha \leftarrow \min(\text{epoch}/\text{min\_epochs}, 1.0)$.
    $m \leftarrow (1 - \alpha) m_0 + \alpha m_f$.
    for $k = 1, \ldots, K$ do
        $\boldsymbol{v} \leftarrow m\boldsymbol{v} - \max(\eta/\gamma^r,\ \eta_{\min}) \nabla f^{S_k}(\boldsymbol{\theta})$.
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.
        $r \leftarrow r + 1$.
    end for
    epoch $\leftarrow$ epoch $+ 1$.
    Calculate the function value $h$ of the validation set.
    if $(h < (1 - X) \times$ lowest_value) then
        counter $\leftarrow N$.
    else
        counter $\leftarrow$ counter $- 1$.
    end if
    lowest_value $\leftarrow \min(\text{lowest\_value},\ h)$.
end while
```


(a) SensIT Vehicle

(b) poker

(c) MNIST

(d) Letter



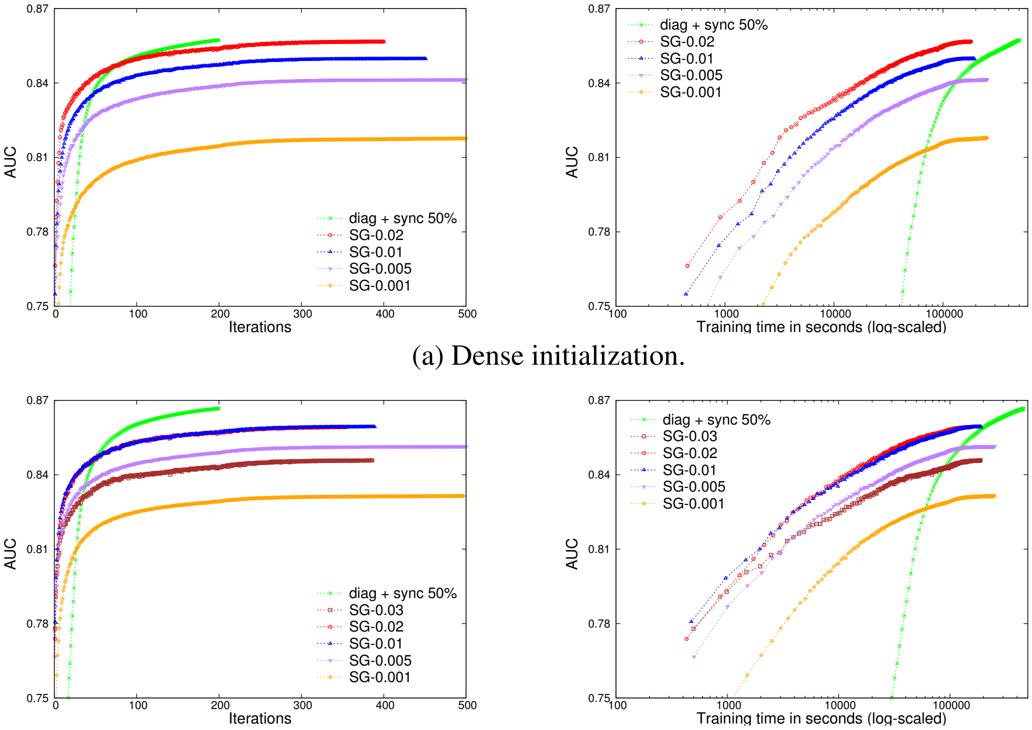
(e) USPS

(f) Pendigits

(g) Sensorless

(h) Satimage

Figure 4: A comparison of different techniques to implement distributed Newton methods. Left: testing accuracy versus number of iterations. Right: testing accuracy versus training time.



(a) Dense initialization.

(b) Sparse initialization.

Figure 5: A comparison between *SG* and *Newton*. A 28-300-300-1 network is applied to train HIGGS. *SG*-x means that the initial learning rate x is used. For *Newton*, each iteration means that we go through line 5 to line 24 in Algorithm 3, while for *SG*, each iteration means that we go through the whole training data once. The curve of *SG-0.03* in the dense initialization is not presented because the AUC value never exceeds 0.5. Left: AUC versus number of iterations. Right: AUC versus training time in seconds (log-scaled).