IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 11, 2018, accepted October 31, 2018, date of publication November 9, 2018,
date of current version December 7, 2018.

*Digital Object Identifier 10.1109/ACCESS.2018.2879877*

# SingleCaffe: An Efficient Framework for Deep Learning on a Single Node

**CHENXU WANG[1,2], YIXIAN SHEN[2,3], JIA JIA[4], YUTONG LU[2,3], ZHIGUANG CHEN[2,3], AND BO WANG[5]**

[1]School of Computer Science, National University of Defense Technology, Changsha 410073, China
[2]National Supercomputer Center in Guangzhou, Guangzhou 510006, China
[3]School of Data and Computer Science, Sun Yat-sen University, Guangzhou 510006, China
[4]Beijing Special Engineering and Design Institute, Beijing 100028, China
[5]State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450002, China

Corresponding author: Chenxu Wang (wangchenxu_nudt@163.com)

This work is supported by the National Key R&D Program of China (Grant 2017YFB0202201), in part by the National Natural Science Foundation of China under Grant U161126 and the National Science Foundation of China under Grant 61872392, and by the Program for Guangdong Introducing Innovative and Entrepreneurial Teams under Grant 2016ZT06D211.

**ABSTRACT** Deep learning (DL) is currently the most promising approach in complicated applications such as computer vision and natural language processing. It thrives with large neural networks and large datasets. However, larger models and larger datasets result in longer training times that impede research and development progress. The modern high-performance and data-parallel nature of hardware equipped with high computing power, such as GPUs, has triggered the widespread adoption of such hardware in DL frameworks, such as Caffe, Torch, and TensorFlow. However, most DL frameworks cannot make full use of this high-performance hardware, and computational efficiency is low. In this paper, we present SingleCaffe[1], a DL framework that can make full use of such hardware and improve the computational efficiency of the training process. SingleCaffe opens up multiple threads to speed up the training process within a single node and adopts data parallelism on multiple threads. During the training process, SingleCaffe selects a thread as a parameter server thread and the other threads as worker threads. Both data and workloads are distributed across worker threads, while the server thread maintains the globally shared parameters. The framework also manages memory allocation carefully to reduce the memory overhead. The experimental results show that SingleCaffe can improve training efficiency well, and the performance on a single node can even achieve the distributed training of a dozen nodes.

**INDEX TERMS** Deep learning, framework, single node, multiple threads, speed up, data parallelism, parameter server.

## I. INTRODUCTION

In recent years, DL has quickly grown in prominence as a key technology to analyze and characterize large volumes of data. State-of-the-art performance has been reported in many practical applications, including speech recognition [1], [2], visual object recognition [3], [4], text processing [5], [6], breast cancer detection [7], and self-driving cars [8]. DL has become one of the hottest topics of interest for both academic research groups as well as prominent companies like Google, Baidu, Facebook, and Microsoft.

Increasing the scale of DL with respect to the number of training examples, model parameters, or both can drastically improve classification accuracy [9]. This has encouraged DL and AI communities to design better, bigger, and deeper neural networks (DNNs) for improving the accuracy of trained models like AlexNet [10], CaffeNet [11], GoogLeNet [12], Network in Network [13], and VGG [14]. However, as model and data scales grow, the training time also lengthen. The time for training some models even reach several months, but no one wants to wait more than a few days or a week for a result. Thus, long training time is a key problem at the root of the development of DL.

Researchers have conducted extensive research to accelerate the training process of DL. These research have triggered the development and adoption of DL frameworks like Caffe [15], Mxnet [16], Theano [17], TensorFlow [18], and Microsoft CNTK [19]. Some studies have been devoted

[1]Our source code and some experimental data can be downloaded from https://github.com/skywang666/SingleCaffe.

to improving the efficiency of distributed training for DL, like S-Caffe [20], FireCaffe [21], and DistBelief [22]. Some researchers have attempted to accelerate the training process by extending existing DL frameworks in a distributed manner with Message Passing Interface (MPI) [40], [41] like TensorFlow with MPI [34], Theano-MPI [35], and MXNET-MPI [36]. However, they have not made detailed studies to improve the computational efficiency of DL on a single node. DeepBench [23] has shown that the efficiency of training DL is sometimes as low as 20%-30% sometimes. High-performance computing hardware, like multi-core CPUs or GPUs, has strong computing power. If we make full use of this hardware, the training speed will increase a great deal.

In order to improve the computational efficiency of the hardware, a proven method is increasing the batch size to make full use of the computational power of each core. However, increasing the batch size will face scale matters, in which the prediction accuracy of the model will decrease dramatically as the batch size increases [24]–[26]. As a result, we must be cautious in increasing the batch size, otherwise we have made a big change at the algorithm level. For the purpose of speeding up the training process, we can learn from the idea of distributed training [27] by which the number of concurrently running tasks are increased in a single node. This method also applies to the proposal of increasing the batch size.

In this paper, we propose a DL framework called SingleCaffe, which can concurrently open up multiple threads to train DL in a single node and improve the computational efficiency of DL in high-performance hardware. SingleCaffe adopts a data parallelism [28] strategy between the multiple threads, in which the training data is divided into different shards, and each thread is responsible for the training of a piece of shard training data. Threads are organized as parameter server architectures [29]–[31] in which a thread is chosen as a centralized sharded parameter server that collectively updates the shared model parameters. The other threads are chosen as worker threads, which are replicas of DL models. In each iteration, each worker thread independently uses its own training data to determine what changes should be made to the parameters saved in parameter server. The framework manages synchronous data communication between threads and supports the on-demand memory usage principle to reduce memory consumption.

Our experiments with several well-known datasets, such as Mnist and Cifar-10 reveal several surprising results about SingleCaffe. Firstly, SingleCaffe works very well in terms of accelerating the training process of DL in a single node. Secondly, SingleCaffe has very good scalability. Thirdly, SingleCaffe can get better computing speed with less computing resources, whose training speed can reach about up to the distributed training of a dozen nodes.

The remainder of this paper is organized as follows. Related works are discussed in Section II. Section III explains some preliminary knowledge about the training of DL. Section IV presents the detailed design of SingleCaffe and how it accelerates the training process. The experimental results evaluating the performance of SingleCaffe are presented in Section V, and our concluding remarks are given in Section VI.

## II. RELATED WORKS

CNTK [19] was developed by Microsoft. It is a unified DL toolkit that describes neural networks as a series of computational steps via a directed graph. It supports distributed training by implementing stochastic gradient descent (SGD) learning with automatic differentiation and parallelization across multiple GPUs and servers. It uses MPI for weight transfer between different nodes. MXNet [32] is a DL framework developed by Li Mu *et al*. It is designed to solve the general problem of executing the bunch of functions according to their dependencies. MXNet has optimized the memory consumption and data loading module for DL. It also supports distributed training, which adopts the parameter server. However, none of these methods have been optimized well in the case of single node training.

Dean *et al*. [22] developed a software framework called DistBelief, which can utilize computing clusters with thousands of machines to train large models. It is the first framework that proposed the introduction of a parameter server to improve distributed DL training. Xing *et al.* [33] was developed as a general-purpose framework that systematically addresses data-parallel and model-parallel challenges in large-scale Machine Learning (ML). Petuum implements the Eager Stale Synchronous Parallel (ESSP) consistency model that allows the workers to run at different speeds, but their speed gap cannot exceed a certain threshold.

FireCaffe [21] successfully scales Caffe across a cluster of GPUs. It adopts reduction tree communication to improve communication efficiency between different workers. S-Caffe [20] is another framework that scales Caffe across a cluster of nodes. It combines Caffe framework and the MVAPICH2-GDR MPI runtime. Using the co-design methodology, it modifies the workflow of Caffe to maximize the overlap between computation and communication with multi-stage data propagation and gradient aggregation schemes. It also brings DL awareness to the MPI runtime by proposing a hierarchical reduction design that benefits from CUDA-aware features.

Vishnu *et al*. [34] extended Google TensorFlow for execution on large-scale clusters using MPI. They apply the strategy of data parallelism and synchronous update. Meanwhile, they also split the samples across all model copies unequally to balance the loads of different computing devices. The evaluation of several well-known datasets indicates the efficiency of the proposed implementation. This work only changed the TensorFlow runtime a little, making the proposed

implementation generic and readily usable for increasingly large users of TensorFlow.

Ma *et al*. [35] developed a scalable and extendable DL training framework based on Theano called Theano-MPI. It can utilize GPUs across nodes to accelerate the training of DL models in a data parallelism manner. Both synchronous and asynchronous training are implemented in the framework. Parameter exchange among GPUs is based on a CUDA-aware MPI. They also explore novel ways to reduce the communication overhead caused by exchanging parameters. In the experiments, they analyze the convergence and capability of the framework to reduce the training time when scaling the synchronous training of AlexNet and GoogLeNet from 2 GPUs to 8 GPUs.

Mamidala *et al*. [36] discuss the drawbacks of the Parameter Server approach and MPI parallelism. They evaluate the drawbacks of such approaches and propose a generic framework named MXNET-MPI to support both PS and MPI programming paradigms running simultaneously. They embed the scaling benefits of MPI parallelism into the loosely coupled PS task model. As well as providing a practical usage model of MPI in the cloud, this framework allows for novel communication-avoiding algorithms that apply parameter averaging in SGD approaches. Furthermore, they also optimize the critical components of the framework, namely global aggregation and allreduce operations, using the novel concept of tensor collectives.

There are also some techniques that can be used in distributed DL frameworks to improve the training efficiency. Goyal *et al*. [25] increase the Bachsize of ResNet-50 to 8,192 without reducing accuracy by adopting a linear scaling rule for adjusting learning rates as a function of batch size and a new warmup scheme that overcomes optimization challenges early in training. You *et al*. [24], [26] propose a method called Layer-wise Adaptive Rate Scaling (LARS). Through this method and the technology mentioned above, they scale the SGD batch size to 32,000 for ImageNet training and finish 100-epoch ImageNet training with AlexNet in 24 minutes.

## III. BACKGROUND

Here, we present an overview of DL, Caffe, data parallelism, and the parameter server architecture.

### A. DEEP LEARNING

DL is a new field in ML research. Its motivation lies in building and simulating the neural network of the human brain to learn knowledge, and it imitates the mechanisms of the human brain to interpret data such as images, sounds, and texts. In the learning process, the programmers or users do not specify which specific features of the raw input data correlate with the outcomes being associated. Instead, the DL algorithm determines which features correlate in the strongest way by training a neural network with a large number of hidden layers [37] and which consists of a layered network of nodes and edges (connections).

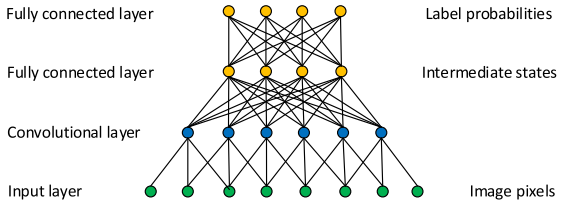



**FIGURE 1. A convolutional neural network with one convolutional layer and two fully connected layers.**

The DL system adopts a hierarchical structure comprising an input layer, hidden layers (multi-layers), and output layers. As depicted in Figure 1, this network consists of a convolutional layer and two fully connected layers. There are connections between the adjacent layer nodes and no connection between the same layer or cross-layer nodes. This layered structure is similar to human brain. Because DL is somewhat difficult to describe, the following section does so in the context of how it works in image classification.

The image classification task often uses the convolutional neural network (CNN) model. The first layer of the nodes (the input of the network) are the raw pixels of the input image, and the last layer of the nodes (the output of the network) are the probabilities that this image should be assigned to each label. The nodes in the middle are intermediate states. To classify an image using such a neural network, the image pixels will be assigned as the values for the first layer of nodes, and these nodes will activate their connected nodes of the next layer. There is a weight associated with each connection, and the value of each node at the next layer is a prespecified function of the weighted values of its connected nodes. Each layer of nodes is activated, one by one, by the setting of the node values for the layer below. This procedure is called a forward pass.

A common way of training a neural network is to use a SGD algorithm. For each training image, a forward pass is executed to activate all nodes using the current weights. The values computed for each node are retained as intermediate states. At the last layer, an error term is calculated by comparing the predicted label probabilities with the true label. Then, the error terms are propagated back through the network with a backward pass. During the backward pass, the gradient of each connection weight is calculated from the error terms and the retained node values, and the connection weights are updated using these gradients. This procedure is called a backward propagation. Through the continuous iteration of forward pass and backward propagation, a convolutional neural network that can predict information is trained.

### B. CAFFE

Caffe is a well-known DL framework developed by the Berkeley Vision and Learning Center. It is written in C++ with CUDA for highly optimized GPU computation, and it also include support for generic CPU architectures. The Caffe framework consists of four main components: the Net (or the

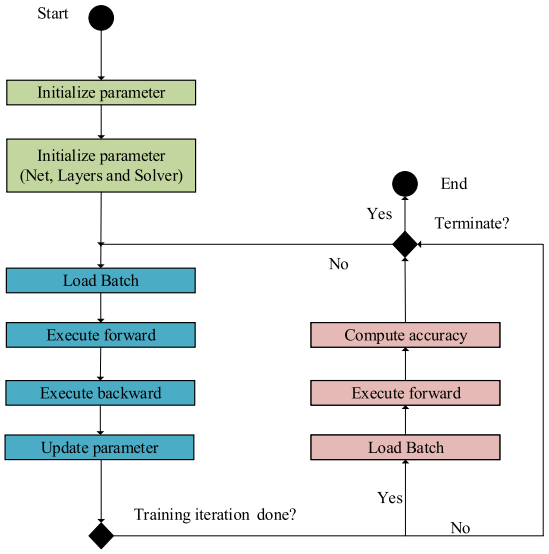



**FIGURE 2.** Caffe architecture.

Model), the Layers, the Solvers, and the Forward/Backward computational Passes. We discuss them in this subsection, but a detailed description can be obtained from [38]. Figure 2 highlights the architecture of Caffe.

Caffe's Net class is used to manage various important elements such as computing, communication, and I/O. Net is the abstraction of the DL network, such as AlexNet or GoogLeNet. The heart of Caffe, which orchestrates the execution of different computations using layers, is called a Solver. It is an abstraction of a class of possible solutions for DL. Caffe mainly provides six kinds of Solvers for solving DL algorithms, such as SGD, AdaDelta, Adaptive Gradient, Adam, Nesterov's Accelerated Gradient, and RMSprop. Each Solver has its own Net object that operates on the Layer data. Layers are abstractions for a defined set of computations for the data. It is an interface that can be designed by DL scientists for specific DL models. The Caffe layers contain two important data structures: the parameter data used in the Forward Pass and the parameter gradients calculated during the Backward Pass.

The first phase of Caffe, data propagation, provides the required parameters. The Forward Pass phase from the first to the last layer generates a loss value. Backward Pass propagates back the loss value to calculate gradients.The gradient updating phase is used to update the parameters. Caffe trains the Net in multiple iterations. Each iteration consists of four phases, as mentioned above. By iterating the above mentioned four processes, Caffe solves the optimal parameters of a DL algorithm.

### C. DATA PARALLELISM AND PARAMETER SERVER

#### 1) DATA PARALLELISM

The solution process of DL is essentially an optimization problem which iteratively solves the maximum or minimum value of the loss function. Use $w$ as the DL weights, $X$ as the training data, $n$ as the number of samples in $X$, and $Y$ as the labels of $X$. Denote $x_i$ as a sample of $X$ and $l(x_i, w)$ as the loss computed by $x_i$ and its label $y_i$ ($i \in 1, 2, \ldots, n$). Typically, people use the loss function as a cross-entropy loss. The goal of DL training is to minimize the loss function in Equation (1).

$$L(w) = \frac{1}{n}\sum_{i=1}^{n} l(x_i, y_i, w) \tag{1}$$

The most popular method to solve this loss function is SGD. It iteratively finds the optimal solution step by step. More precisely, at $t$-th, we use forward propagation to get the loss and calculate the gradients through this loss. Then we use the gradients to update the weights, as shown in Equation (2).

$$w_{t+1} = w_t - \eta \nabla l(x_i, y_i, w) \tag{2}$$

Usually, a single sample is not used to compute the loss and the gradients, as it can cause very poor convergence efficiency. It is chosen to use a batch of samples at each iteration. This method is called Mini-Batch SGD. Denote the batch of samples at the $t$-th iteration as $B_t$. The size of $B_t$ is $b$. We then update the weights based on Equation (3).

$$w_{t+1} = w_t - \frac{\eta}{b}\sum_{x \in B_t} \nabla l(x, y, w) \tag{3}$$

With the increasing sizes of datasets and training model parameters, training time lengthens a great deal. In order to accelerate the training process, data parallelism is the most appropriate parallel strategy. According to this method, the input data is partitioned into $P$ parts stored on each worker. Each worker has a local copy of the DL model and the weights ($w$). The $i$-th worker is responsible for computing $\nabla w_i$ according to the corresponding data $P_i$ and weights $w_i$. Accumulate all gradients ($\nabla l(x_1, y_1, w) \ldots \nabla l(x_P, y_P, w)$) calculated by all workers, and use these gradients to update the weights. This process is expressed in the form of Equation (4).

$$\nabla l(x, y, w) = \sum_{i=1}^{P} \nabla l(x_i, y_i, w) \tag{4}$$

#### 2) PARAMETER SERVER

The data parallelism strategy can be perfectly implemented with the parameter server, as shown in Figure 3. The parameter server has been proven as an essential component of efficient distributed DL, and it is used in numerous applications and frameworks. It consists of two types of nodes: 1) worker nodes (Model Replicas), which partition the input data and calculate partial updates, and 2) parameter server nodes, which partition the model parameters and aggregate/apply the partial updates sent by worker nodes. In each iteration, the worker nodes calculate the partial weights and send them to the parameter server nodes. The parameter server nodes then use these weights to update the global weights. The above process is repeated again and again until convergence is achieved.

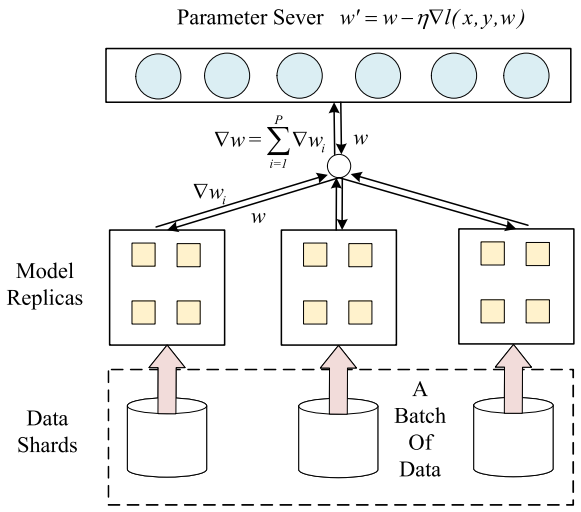



**FIGURE 3.** Parameter server architecture.

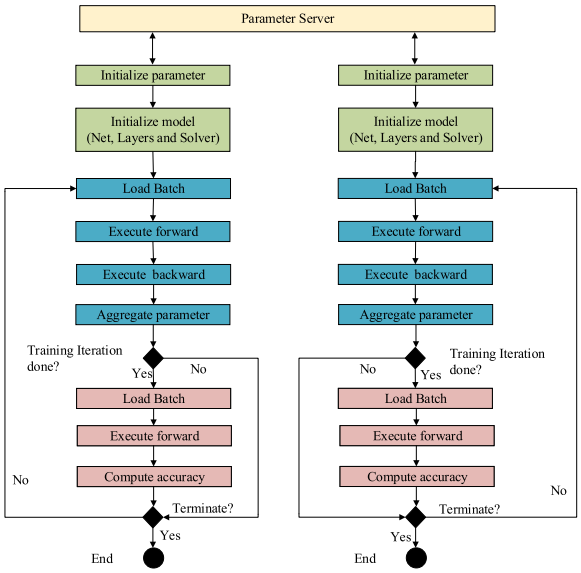



**FIGURE 4.** Design of singleCaffe.

DL applications share the distinct property that they are often robust against small errors in their execution. As a result, DL frameworks adopt a variety of consistent communication strategies, such as bulk synchronous parallel (BSP), asynchronous parallel (ASP), and staleness synchronous parallel (SSP). Using the BSP strategy, the parameter server nodes do not update the global weights until all of the worker nodes commit their local model parameters. Even in a load-balanced cluster, a portion of the computational nodes can be randomly and unpredictably slower than the others [39]. Therefore, this method performs poorly when there are dragged worker nodes. According to the ASP strategy, when a worker node commits its local model parameters, it sends them to the parameter server nodes immediately. The parameter server then updates the global weights without waiting for the other worker nodes' results and sends back updated parameters. Meanwhile, the worker nodes start the next iteration. It is important to note that this method has low convergence efficiency. Using the SSP strategy, the worker nodes can perform the next iteration directly instead of waiting for the other worker nodes to finish, unless the fastest computational node is ahead of the slowest computational node by some (a threshold) iterations.

## IV. DESIGN AND IMPLEMENTATION

### A. DESIGN

Modern high-performance hardware often consists of many computing cores with powerful computing ability. For example, each node of Tianhe-2 is made up of two Xeon E5 CPUs, and each Xeon E5 contains 12 calculation cores. When training a DL model, using only a single process will result in low device utilization and low computational efficiency. To increase equipment utilization and improve computational efficiency, SingleCaffe opens up multiple threads to train a single DL model, and each thread is a complete training process.

Inspired by the distributed DL frameworks, SingleCaffe organizes threads into the parameter server architecture. As shown in Figure 4, the Server thread maintains the globally shared parameters, and some Worker threads calculate weights and send them to the parameter thread. There is also a helper thread not shown in Figure 4 called Scheduler thread, which is responsible for task scheduling. The details of these threads are introduced in IV-B.

In each iteration, the Worker threads firstly fetch the parameters from the parameter server to initialize weights and models. The Worker threads then load data and undertake Forward/Backward propagation to calculate the sub-gradients and send them to the Server thread. The Server thread then accumulates all the sub-gradients and updates the parameters. After a certain number of iterations, SingleCaffe enters the test phase to decide whether to stop training. For the sake of accelerating this phase and load balancing, SingleCaffe divides the test task evenly between the Worker threads.

SingleCaffe adopts the data parallelism strategy. In each iteration, the worker threads divide the batch size of the data equally. Each Worker thread calculates the local gradient corresponding to the data that it fetches. After all the local gradients have been calculated, SingleCaffe uses the bulk synchronous parallel strategy to aggregate the local gradients. The Server thread then updates the weights using these aggregated gradients. As SingleCaffe does not need to send messages across nodes, it only needs to manipulate the gradients in the shared memory, saving time for message transference. Compared to the distributed framework, this is an important advantage concerning training DL model on a single node.

### B. IMPLEMENTATION

SingleCaffe organizes multiple threads into a parameter server architecture to train the DL model in high concurrency

**Algorithm 1** SingleCaffe Scheduling Algorithm

**Scheduler thread:**

**for** iteration $t = 0, \cdots, T$ **do**
    issue LoadBatch() to all workers
    issue WorkerIterate($t$) to all Worker threads
    wait for all finished signals from all Worker threads
    wait for finished update from Server thread
**end for**

**Worker thread** $r = 1, 2, \cdots, m$**:**

**function** LoadBatch()
    load a part of training data $(X_r, Y_r)$
    pull the working weights set $w_r$ from parameter server
**end function**
**function** WorkerIterate($t$)
    $W_r^t \leftarrow \sum_{k=1}^{n_r} \nabla(x_{i_k}, y_{i_k}, w_r^t)$
    push $W_r^t$ to parameter server
    pull $w_r^{t+1}$ from parameter server
    send finished signal to Scheduler thread
**end function**

**Solver thread:**

**function** ServerIterate($t$)
    aggregate $W^t \leftarrow \sum_{r=1}^{m} W_r^t$
    $W^{t+1} \leftarrow W^t - \eta W^t$
    send finished update to Scheduler thread
**end function**

manner. The framework contains three kinds of thread: the Scheduler, Worker, and Solver threads. They each have different functions and collaborate to accelerate the DL training process. Specific details are shown in Algorithm 1.

The Scheduler thread is used to control the jobs of the Worker and Solver threads. In each iteration, the Scheduler thread starts the Worker threads in order to load data and start training. Meanwhile, it monitors whether the Worker and Server threads have finished their jobs. Due to SingleCaffe's bulk synchronous data parallel strategy, the Scheduler thread does not start the next iteration until it receives all the finished signals from the Worker and Server threads.

Each Worker thread is a replica of the DL model. When it receives the signal to start training, it first loads the input data. In this step, in order to speed up the training of a batch of data, it divides the batch equally among the Worker threads instead of loading the entire batch. After reading the input data, the Worker threads compute the sub-gradient and send them to the Server thread. As memory is shared, the Worker threads do not need to actually send a message. They only need to tell the Server thread the memory address at which the sub-gradients are stored, which saves great deal of time in sending messages. Finally, the Worker thread notifies the Scheduler thread that it has finished this iteration.

The Server thread is used to update and maintain the parameters. After receiving the memory address, the sub-gradients are accumulated and averaged over the number of Worker threads. To save computing resources, we do not assign actual hardware resources to the Server thread. We integrate the function of the Server thread into the Worker threads. In addition, we also optimize the workload of the Worker threads.

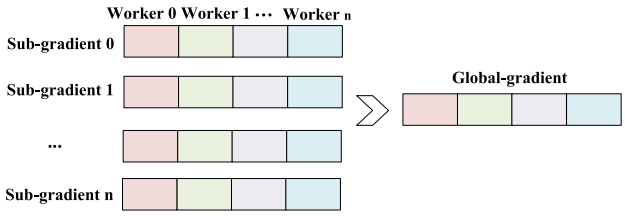



**FIGURE 5.** Workload division overview.

SingleCaffe is developed using C++ and it is based on Caffe. Each Worker thread is a complete Caffe training process, and SingleCaffe organizes these threads together using OpenMP for concurrent training. Although SingleCaffe is based on Caffe, the proposed method is flexible and general. It can also be adapted to different frameworks for the sake of improving the computing efficiency of hardware devices and speeding up the training process.

### C. IMPROVING EFFICIENCY

We also conduct some optimization measures to improve computational efficiency. First, balance the workload between the Worker threads to reduce workload inconsistency. Second, increase the number of pieces of data loaded each time appropriately to improve execution efficiency.

#### 1) WORKLOAD BALANCING

The Server thread is required to add up the calculated sub-gradients and calculate the mean of these sub-gradients. In order to save hardware resources, SingleCaffe assigns this workload to Worker threads instead of the Server thread. As the datasets and DL model are constantly increasing, number of parameters also increases, from MB to GB. Therefore, accumulating the sub-gradients is a considerable workload. If the workload is unevenly distributed, it will result in low performance. On the other contrary, DL training is a repeated process, and the imbalance increases with each iteration, which will lead to a great performance degradation.

An overview of the workload division is shown in Figure 5. SingleCaffe allocates the memory space for these sub-gradients, and Worker threads can directly operate because memory is shared. We slice every sub-gradient storage space according to the number of Worker threads. Then a Worker thread is responsible for a slice space. For example, Worker thread 0 accumulates and averages the slice 0 of all sub-gradients. Simultaneously, to save memory space, we assign the Server's task to a Worker thread and use this Worker thread's sub-gradient space to hold the updated parameters.

#### 2) INCREASING THE BATCH SIZE

SingleCaffe can train a batch of data concurrently. Increasing the batch size appropriately can improve computational efficiency without changing the accuracy of the training [24]–[26]. There are two main reasons for this phenomenon. a) DL essentially operates as a dense matrix multiplication. Increasing the batch size can increase the data that is taken by each thread. This process can expand the scale of the matrix multiplication and improve the efficiency of the underlying library functions. b) Increasing the amount of data taken at each iteration while the total training data is unchanged leads to a reduction in the number of iterations. Meanwhile, each iteration's computational efficiency is improved, and the total training time is reduced.

## V. EXPERIMENTAL EVALUATION

### A. EXPERIMENTAL ENVIRONMENT

In this section, we analyze the performance of SingleCaffe. The experiments in this paper are all performed on Tianhe-2 at the National Supercomputer Center in Guangzhou. There are about 16,000 computing nodes which are connected through Optoelectronics Hybrid Transport Technology. Each node has 64GB of memory, and two Ivy Bridge-EX 2.2 GHz Xeon E5-2692v2 processors, and three Xeon Phi coprocessors, each of which has 8GB of memory and 57 computing cores on board. In the next generation exascale computer, Tianhe-2 will adopt a CPU + Matrix 2000 heterogeneous architecture. Matrix 2000 is a general-purpose DSP coprocessor developed by the National University of Defense Technology (NUDT). The experiments in this paper mainly use the CPUs in Tianhe-2.

First, we compare the performance of SingleCaffe using different thread counts in order to evaluate the performance and scalability of SingleCaffe. Second, we test the performance of SingleCaffe against different batch size to evaluate the efficiency of the framework. Finally, we compare the performance of SingleCaffe with three DL frameworks that all support distributed training. The first one is called DistCaffe. We developed DistCaffe based on native Caffe. It supports distributed training and adopts a synchronous communication strategy. The others are Intel-Caffe and TensorFlow. All three frameworks use MPI for communication between nodes. Both Intel-Caffe and TensorFlow support multi-core parallelism inside a node. By default, the number of cores that a node use is set to the total number of CPU cores.

We train two models to evaluate the performance of the proposed method on AlexNet [42] and LeNet-5 [43]. AlexNet is developed by Krizhevsky *et al*., the first model to prove the efficiency of deep neural networks for image recognition. LeNet-5 was designed by LeCun *et al*. and contains one input layer, one output layer, three convolution layers, two pooled layers, and one fully connected layer. The batch size is 64 and the maximum number of iterations is 10,000. AlexNet is trained by the dataset Cifar-10 [44]. Cifar-10 consists of 60,000 images in 10 classes. Among the 60,000 images, we use 50,000 as training images and 10,000 as validation images. LeNet-5 is trained by the dataset Mnist [45], which has a training set of 60,000 examples and a test set of 10,000 examples.

In order to verify SingleCaffe's suitability for different operations, we trained three different variants of networks for AlexNet with batch normalization and different activation functions (ReLU, Sigmod). Furthermore, in order to demonstrate the effect of SingleCaffe on different solvers, we used three solvers (native SGD, Adam, RMSProp) to solve LeNet-5. The results for LeNet-5 demonstrate the time taken by the proposed method to undergo 10,000 iterations, and the results for AlexNet demonstrate the time taken by the proposed method to undergo 60,000 iterations. Each result is the average of multiple experiments.

### B. PERFORMANCE ANALYSIS OF SINGLECAFFE

In this section, we analyze the performance of SingleCaffe. To evaluate its performance, we conduct many experiments using the Cifar10 and Mnist datasets. In the experiments, the number of threads ranges from 1 to 24 because the computing device that is being used has 24 cores. If the number of threads equals 1, native Caffe is used. If the number of threads is greater than 1, SingleCaffe is used. With this setup, we can clearly analyze the acceleration and scalability of SingleCaffe. For each result, we undertake five experiments in order to allow for the average. The results for Cifar10 are shown in Figure 6, and the results for Mnist are shown in Figure 7.

From Figure 6, we find that SingleCaffe performs much better than native Caffe. SingleCaffe is even 16 times faster than native Caffe in the best case. Furthermore, SingleCaffe has good scalability. As the number of threads increases, the acceleration effect increases and essentially remains constant in the end. SingleCaffe's performance on the Mnist dataset is basically the same as that of Cifar10. From Figure 7, we can see that SingleCaffe performs much better than native Caffe too. However, SingleCaffe's acceleration effect on the Mnist dataset is not as effective as Cifar10. This is because Mnist is a much smaller dataset than Cifar10, and LeNet-5 is much simpler than AlexNet. So increasing the number of threads, SingleCaffe cannot use computing resources more efficiently for Mnist.

We also find that when the number of threads reaches a certain scale, SingleCaffe's acceleration efficiency no longer increases or decreases. This might be because the hardware resources are limited. The competition for computing resources like I/O and memory becomes increasingly profound when the number of threads increase, which leads to low computing efficiency. If competition for resources is very serious, it may even cause a decrease in computing performance. Another reason is the efficiency of the linear algebra library. DL operations are dense matrix multiplications. Small dense matrix multiplication is less efficient than big dense matrix multiplication for linear algebra library. The higher the number of threads, the smaller the data that is fed

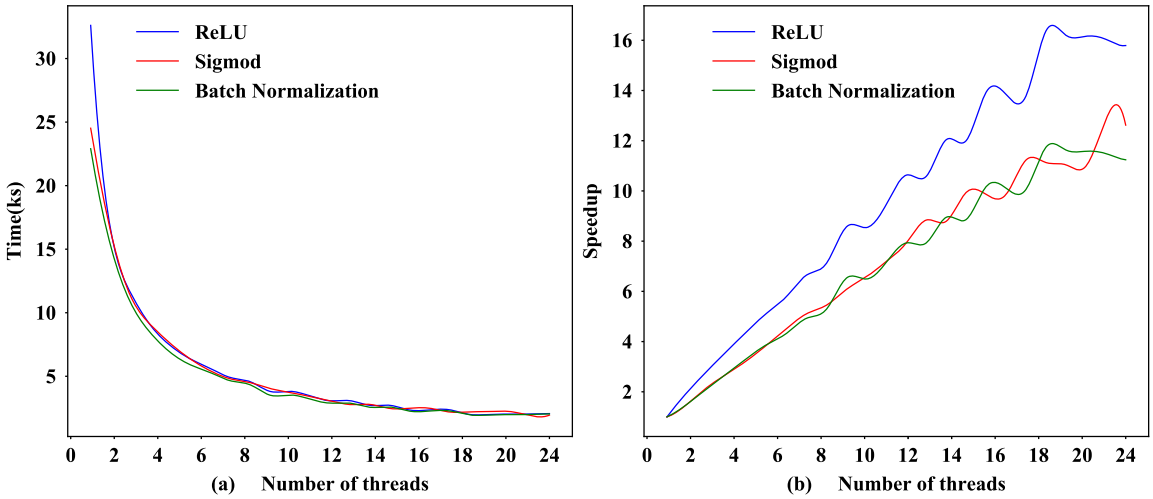



**FIGURE 6.** Performance of SingleCaffe on Cifar10.

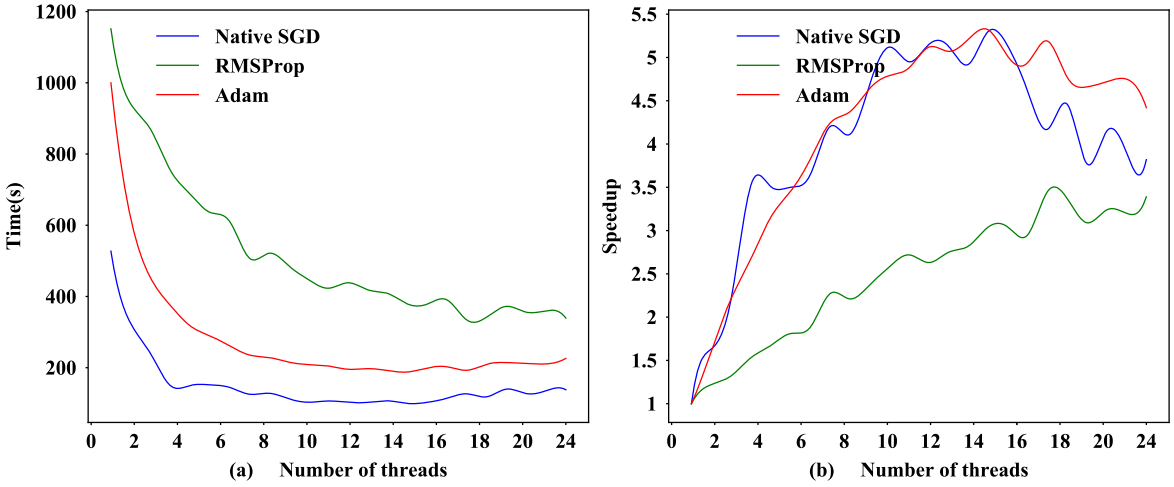



**FIGURE 7.** Performance of SingleCaffe on Mnist.

to each Worker thread. Too little data results in a smaller dense matrix, which can also lead to a decline in computing performance.

### C. INFLUENCE OF THE BATCH SIZE

The batch size is the amount of data that is fed to the training model at anyone time. In theory, the larger the batchsize, the more efficient the calculation. However, the relationship between the batch size and the performance of the framework is very complex and affected by many factors, including number of threads and the linear algebra library. Therefore, we will evaluate the effect of the batch size on SingleCaffe in this section. We also conduct many experiments using the Cifar10 and Mnist datasets and show the performance at intervals of four threads, in which the batch size is set to 100, 150, and 200 for Cifar10 and 64, 96, and 128 for Mnist. Since the results of different models and Solvers are similar, we only give the results of ReLu and native SGD. The results are shown in Figure 8.

From Figure 8, it is clear that when the number of threads are the same, the larger the batch size is, the faster the training is. For example, if the number of threads is 12 in Cifar10, the training speed is fastest when the batch size is equal to 200. This is because increasing the batch size increases the amount of data that is fed to each computing core at a time, which also increases the scale of the matrix operations and makes the linear algebra library more efficient.

We also find that when batch size is constant, when the number of threads increases, the training speed does not increase. For example, when the batch size is 64 in Mnist, the training speed rises first and then drops when the number of threads increasing. This is because SingleCaffe divides batch size data between the computing cores, resulting in less data per core processing and a decrease in computational efficiency. As computing efficiency decreases, the gains from

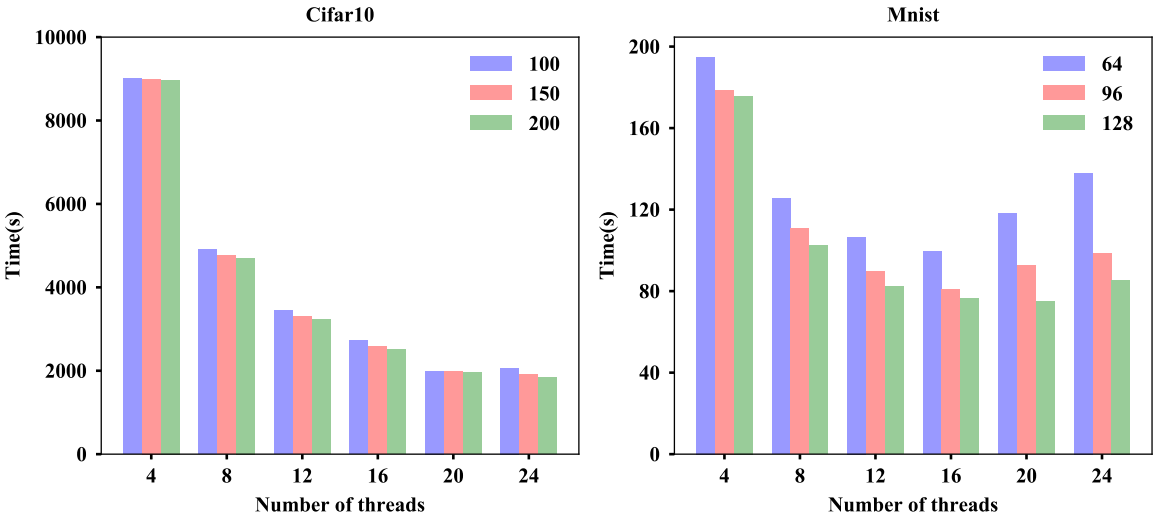



**FIGURE 8.** The effect of the batch size on the performance of SingleCaffe.

increasing the computing resources also decrease. This leads to the training speed first rising and then dropping along with the increasing number of threads.

### D. COMPARISON WITH OTHER DL FRAMEWORKS

In this section, we compare the performance of SingleCaffe with three other DL training frameworks that all support distributed training with MPI. The first one is called DistCaffe, based on Caffe and MPI. The second one is Intel-Caffe based on Caffe and optimized for Intel processors. Intel-Caffe is a good contrast with the design proposed in this paper. The last one is TensorFlow, developed by Google. It is currently one of the most widely used DL frameworks. These three DL frameworks have very good reference value relative to SingleCaffe.

#### 1) COMPARING SINGLECAFFE WITH DIFFERENT FRAMEWORKS

In order to evaluate the differences in performance of these frameworks, we compare both single-node and distributed performance. We set the number of computing nodes from 1 to 24 and use Speed Ratio to indicate the performance comparison between these frameworks. The value of the Speed Ratio is the best performance of SingleCaffe divided by the performance of other frameworks. If the value of Speed Ratio is greater than 1, the performance of SingleCaffe is better. If the value of SingleCaffe is less than 1, the performance of SingleCaffe is poor. The results are shown in Figure 9.

We compare the performance of SingleCaffe and DistCaffe. For Cifar10, SingleCaffe performs better than DistCaffe in most cases. We find that the Speed Ratio value is bigger than 1 if the number of nodes is smaller than 15. Furthermore, when the number of nodes is greater than 15, the Speed Ratio value is slightly smaller than 1. This means that using SingleCaffe for single-node training can achieve the performance of 15 nodes by DistCaffe. Even when the number of nodes is greater than 15, the performance gap between them is slight. For Mnist, the Speed Ratio value is greater than 1 in all cases. Therefore, SingleCaffe performs better than DistCaffe in all cases for Mnist.

Next, compare the performance of SingleCaffe and Intel-Caffe. We find that the Speed Ratio value is greater than 1 in all cases regardless of whether Cifar10 or Mnist is being used. This means that SingleCaffe outperforms Intel-Caffe in all cases. A special feature of Intel-Caffe is that its performance on a single node is very good. For example, the Speed Ratio value of a single node is smaller than some cases of distributed training for Intel-Caffe. This might be because the computational efficiency of Intel-Caffe on a single node is very high, which can lead to a significant impact of communication time on total performance. Since the performance variance of Intel-Caffe is not within the scope of this article, we do not analyze this phenomenon.

The situation of TensorFlow is similar to the case of Intel-Caffe. The Speed Ratio value is greater than 1 in all cases regardless of whether Cifar10 or Mnist is being used. So SingleCaffe outperforms TensorFlow in all cases. When the number of nodes is less than six, TensorFlow performs significantly better using Cifar10 than Mnist, which shows that TensorFlow performs poorly on small datasets.

Therefore, we can conclude that SingleCaffe performs excellently on a single node. Compared with the other three frameworks, it can even achieve a distributed training effect of at least 15 nodes. If we use SingleCaffe as a single-node training unit and extend it to distributed training, more benefits will result.

#### 2) COMPARISON USING DIFFERENT CONFIGURATIONS

In order to analyze the performance comparison between SingleCaffe and other deep learning frameworks with different

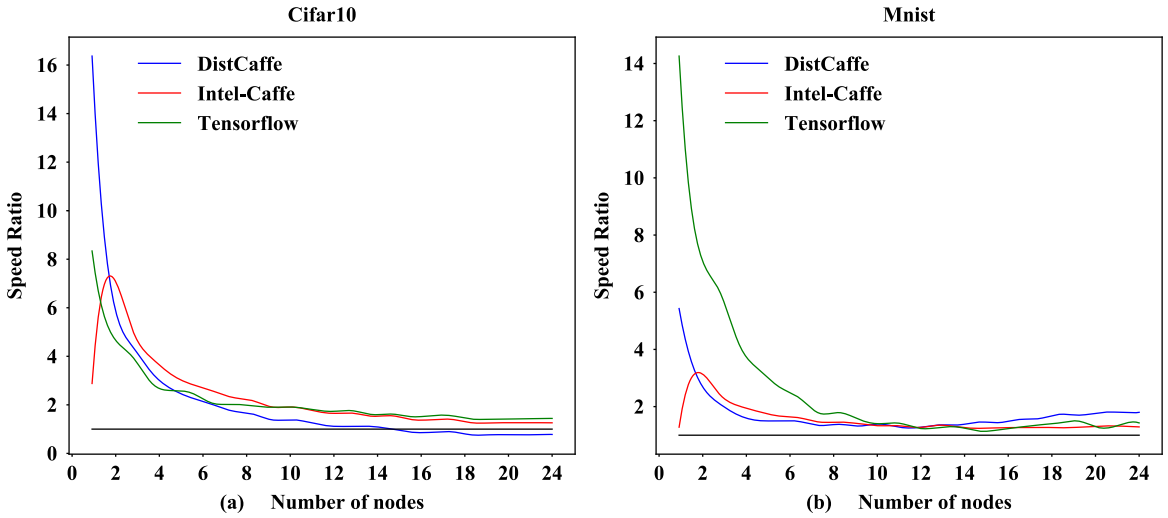



**FIGURE 9.** Performance comparison between SingleCaffe and other DL frameworks.

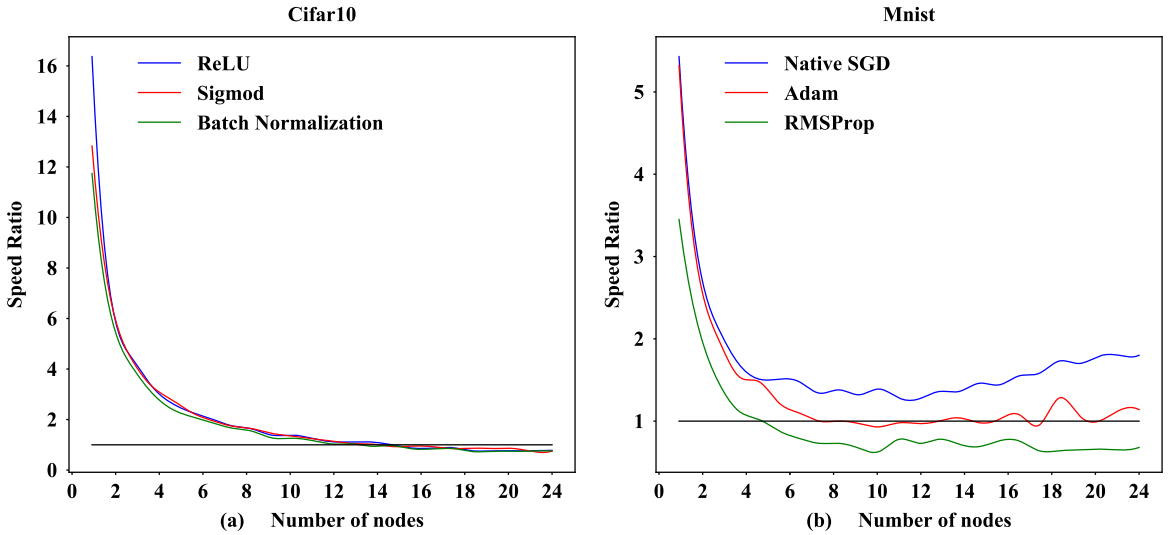



**FIGURE 10.** A comparison of the performance of SingleCaffe and DistCaffe with different configurations.

operations and different Solvers, we conduct three different variants (Batch Normalization, ReLU, Sigmod) of networks for Cifar10 and three different Solvers (native SGD, Adam, RMSProp) for Mnist. The experiment setup is the same as last subsection. In reference to the experimental results, we find that the results of the three frameworks are basically the same. Due to the length of this paper, we only show the performance comparison between SingleCaffe and DistCaffe with different configurations. The results are shown in Figure 10.

For Cifar10, the comparison of the performance of SingleCaffe concerning the three network variants is basically the same. If there are less than four computing nodes, the difference in the performance of the three variants is slightly large. If there are more than four nodes, their performance tends to be constant. However, SingleCaffe outperforms DistCaffe concerning the performance of the three network variants when the number of compute nodes is less than 15. When there are more than 14 nodes, the Speed Ratio values of these three variants are only slightly smaller than 1, which means that SingleCaffe's performance is only slightly poorer than that of DistCaffe.

SingleCaffe performs better than distributed training in all cases for native SGD and most cases for Adam on Mnist. Even with respect to RMSProp, when there are more than 5 nodes, the Speed Ratio value is only slightly less than 1. This means that SingleCaffe can perform well on native SGD and Adam. Even for RMSProp, when there are more than 5 nodes, the difference between their performance is minor.

Therefore, we can conclude that SingleCaffe can secure a faster training speed using less computing resources with different operations and Solvers.

## VI. CONCLUSION AND FUTURE WORK

We have proposed SingleCaffe for single-node DL training. It makes full use of hardware resources and improves computational efficiency. It opens up multiple threads and adopts data parallelism to distribute data and workloads between those threads. We also prove that SingleCaffe has high computational efficiency and scalability by means of many experiments. Furthermore, it can secure a higher training speed with less computing resources compared to other frameworks.

In the future, we plan to apply this single-node optimization to distributed DL. At the same time, we are also planning to optimize the communication and workflow of distributed DL.

## REFERENCES

[1] G. E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition," *IEEE Trans. Audio, Speech, Language Process.*, vol. 20, no. 1, pp. 30–42, Jan. 2012.

[2] G. Hinton *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, Nov. 2012.

[3] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep, big, simple neural nets for handwritten digit recognition," *Neural Comput.*, vol. 22, no. 12, pp. 3207–3220, 2010.

[4] A. Coates, A. Ng, and H. Lee, "An analysis of single-layer networks in unsupervised feature learning," in *Proc. 14th Int. Conf. Artif. Intell. Statist.*, 2011, pp. 215–223.

[5] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A neural probabilistic language model," *J. Mach. Learn. Res.*, vol. 3, pp. 1137–1155, Feb. 2003.

[6] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proc. 25th Int. Conf. Mach. Learn.*, 2008, pp. 160–167.

[7] D. Wang, A. Khosla, R. Gargeya, H. Irshad, and A. H. Beck. (2016). "Deep learning for identifying metastatic breast cancer." [Online]. Available: https://arxiv.org/abs/1606.05718

[8] (2018).*Nvidia Development Scalable AI Platform for Autonomous Driving*. Accessed: Apr. 11, 2018. [Online]. Available: https://www.nvidia.com/en-us/self-driving-cars/drive-platform/

[9] Q. V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Y. Ng, "On optimization methods for deep learning," in *Proc. 28th Int. Conf. Int. Conf. Mach. Learn.* Athens, Greece: Omni Press, 2011, pp. 265–272.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.

[11] (2018). *Caffenet*. Accessed: Apr. 11, 2018. [Online]. Available: http://papers.nips.cc/book/advances-in-neural-information-processing-systems-25-2012

[12] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. CVPR*, Jun. 2015, pp. 1–9.

[13] M. Lin, Q. Chen, and S. Yan. (2013). "Network in network." [Online]. Available: https://arxiv.org/abs/1312.4400

[14] K. Simonyan and A. Zisserman. (2014). "Very deep convolutional networks for large-scale image recognition." [Online]. Available: https://arxiv.org/abs/1409.1556

[15] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. 22nd ACM Int. Conf. Multimedia*, 2014, pp. 675–678.

[16] T. Chen *et al.* (2015). "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems." [Online]. Available: https://arxiv.org/abs/1512.01274

[17] F. Bastien *et al.* (2012). "Theano: New features and speed improvements." [Online]. Available: https://arxiv.org/abs/1211.5590

[18] M. Abadi *et al.* (2016). "TensorFlow: Large-scale machine learning on heterogeneous distributed systems." [Online]. Available: https://arxiv.org/abs/1603.04467

[19] (2018). *CNTK*. Accessed: Apr. 11, 2018. [Online]. Available: https://docs.microsoft.com/en-us/cognitive-toolkit/setup-cntk-on-your-machine

[20] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-Caffe: Co-designing MPI Runtimes and Caffe for scalable deep learning on modern GPU clusters," in *Proc. 22nd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2017, pp. 193–205.

[21] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer, "FireCaffe: Near-linear acceleration of deep neural network training on compute clusters," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 2592–2600.

[22] J. Dean *et al.*, "Large scale distributed deep networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1223–1231.

[23] (2018). *Deepbench*. Accessed: Apr. 11, 2018. [Online]. Available: https://github.com/baidu-research/DeepBench

[24] Y. You, I. Gitman, and B. Ginsburg. (2017). "Large batch training of convolutional networks." [Online]. Available: https://arxiv.org/abs/1708.03888

[25] P. Goyal *et al.* (2017). "Accurate, large minibatch SGD: Training ImageNet in 1 hour." [Online]. Available: https://arxiv.org/abs/1706.02677

[26] Y. You, Z. Zhang, C.-J. Hsieh, J. Demmel, and K. Keutzer, "Imagenet training in minutes," in *Proc. 47th Int. Conf. Parallel Process.*, 2018, p. 1.

[27] A. Qiao *et al.*, "Litz: Elastic framework for high-performance distributed machine learning," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Boston, MA, USA, 2018, pp. 631–644. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/qiao

[28] R. Wu, S. Yan, Y. Shan, Q. Dang, and G. Sun. (2015). "Deep Image: Scaling up image recognition." [Online]. Available: https://arxiv.org/abs/1501.02876

[29] M. Li *et al.*, "Scaling distributed machine learning with the parameter server," in *Proc. OSDI*, vol. 14, 2014, pp. 583–598.

[30] Q. Ho *et al.*, "More effective distributed ML via a stale synchronous parallel parameter server," in *Proc. Adv. Neural Inf. Process. Syst.*, 2013, pp. 1223–1231.

[31] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server," in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, Art. no. 4.

[32] T. Chen *et al.* (2015). "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems." [Online]. Available: https://arxiv.org/abs/1512.01274

[33] E. P. Xing *et al.*, "Petuum: A new platform for distributed machine learning on big data," *IEEE Trans. Big Data*, vol. 1, no. 2, pp. 49–67, Jun. 2015.

[34] A. Vishnu, C. Siegel, and J. Daily. (2016). "Distributed TensorFlow with MPI." [Online]. Available: https://arxiv.org/abs/1603.02339

[35] H. Ma, F. Mao, and G. W. Taylor, "Theano-MPI: A theano-based distributed training framework," in *Proc. Eur. Conf. Parallel Process.* Cham, Switzerland: Springer, 2016, pp. 800–813.

[36] A. R. Mamidala, G. Kollias, C. Ward, and F. Artico. (2018). "MXNET-MPI: Embedding MPI parallelism in parameter server task model for scaling deep learning." [Online]. Available: https://arxiv.org/abs/1801.03855

[37] Y. Bengio and Y. LeCun, "Scaling learning algorithms towards AI," *Large-Scale Kernel Mach.*, vol. 34, no. 5, pp. 1–41, 2007.

[38] (2018). *Caffe Website*. Accessed: Apr. 11, 2018. [Online]. Available: http://caffe.berkeleyvision.org/

[39] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *Proc. OSDI*, vol. 14, 2014, pp. 571–582.

[40] W. Gropp *et al.*, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Comput.*, vol. 22, no. 6, pp. 789–828, 1996.

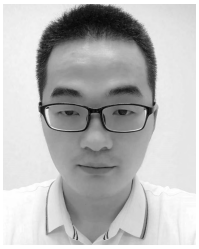
[41] A. Geist *et al.*, "Mpi-2: Extending the message-passing interface," in *Euro-Par'96 Parallel Processing*, L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, Eds. Berlin, Germany: Springer, 1996, pp. 128–135.

[42] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Red Hook, NY, USA: Curran Associates, 2012, pp. 1097–1105.

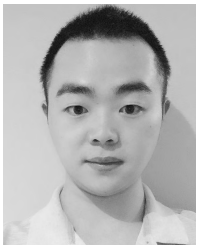
[43] Y. LeCun *et al.*, "Handwritten digit recognition with a back-propagation network," in *Advances in Neural Information Processing Systems 2*, D. S. Touretzky, Ed. San Mateo, CA, USA: Morgan Kaufmann, 1990, pp. 396–404.

[44] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Univ. Toronto, Toronto, ON, Canada, Tech. Rep., 2009, vol. 1, no. 4.

[45] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.



**CHENXU WANG** received the B.S. degree from the Information Institute, Northeastern University, Shenyang, China, in 2013, and the M.S. degree from the School of Computer Science, National University of Defense Technology, Changsha, China, in 2015, where he is currently pursuing the Ph.D. degree. His research interests include machine learning, DL, high-performance computing, and parallel algorithm.



**YIXIAN SHEN** received the B.S. degree in electronic and information engineering from Zhejiang Communication University, Hangzhou, China, in 2017. He is currently pursuing the M.S. degree with Sun Yat-sen University, Guangzhou, China. His research interests include data mining and parallel computing.

**JIA JIA** was born in 1981. He received the Ph.D. degree from the School of Computer Science, National University of Defense Technology, in 2011. His main research interests include computer architecture, parallel computation, and fault-tolerance technique.

**YUTONG LU** received the Ph.D. degree from the School of Computer Science, National University of Defense Technology, Changsha, China, in 2009. She is currently a Professor with the School of Data and Computer Science, Sun Yat-sen University. She is also the ISC Fellow and the Director of the National Supercomputer Center, Guangzhou. Her research interests include parallel system management, high-speed communication, distributed file systems, and advanced programming environments with MPI.

**ZHIGUANG CHEN** received the Ph.D. degree from the School of Computer Science, National University of Defense Technology, Changsha, China, in 2013. He is currently a Researcher with the School of Data and Computer Science, Sun Yat-sen University. His research interests include high-performance computing and large-scale storage systems.

**BO WANG** received the M.S. degree from the School of Computer Science, National University of Defense Technology, Changsha, China, in 2015. He is currently an Officer with the State Key Laboratory of Mathematical Engineering and Advanced Computing. His research interests include DL, parallel computing, and high-performance computing.