A Parallel Computing Platform for Training Large Scale Neural Networks

Rong Gu, Furao Shen, Yihua Huang
National Key Laboratory for Novel Software Technology
Nanjing University, Nanjing, China 210093
Email: gurong@smail.nju.edu.cn, {frshen, yhuang}@nju.edu.cn

***Abstract*—Artificial neural networks (ANNs) have been proved to be successfully used in a variety of pattern recognition and data mining applications. However, training ANNs on large scale datasets are both data-intensive and computation-intensive. Therefore, large scale ANNs are used with reservation for their time-consuming training to get high precision. In this paper, we present cNeural, a customized parallel computing platform to accelerate training large scale neural networks with the backpropagation algorithm. Unlike many existing parallel neural network training systems working on thousands of training samples, cNeural is designed for fast training large scale datasets with millions of training samples. To achieve this goal, firstly, cNeural adopts HBase for large scale training dataset storage and parallel loading. Secondly, it provides a parallel in-memory computing framework for fast iterative training. Third, we choose a compact, event-driven messaging communication model instead of the heartbeat polling model for instant messaging delivery. Experimental results show that the overhead time cost by data loading and messaging communication is very low in cNeural and cNeural is around 50 times faster than the solution based on Hadoop MapReduce. It also achieves nearly linear scalability and excellent load balancing.**

***Keywords*-parallel computing; neural network; big data; fast training; distributed storage**

I. Introduction

Artificial neural networks (ANNs) have been used in a variety of data mining and pattern recognition applications, such as protein structure analysis, speech recognition, image and signal processing, handwriting recognition [1]. However, the process of training large scale neural networks is both computation-intensive and data-intensive. On one hand, an entire training workflow usually needs to carry out thousands of epochs' iteration, which makes it computationally expensive. On the other hand, in order to generate solid results, large scale training datasets are usually used in applications. As a result, training large scale neural networks on a single PC is usually very time-consuming, which may take several days to weeks to finish and sometimes even can not be done. Thus, the slow training speed of large scale neural networks has limited their use for processing many complex and valuable problems in practice.

On the other side, the amount of data in real world has been exploding since last several years, and analyzing big data becomes quite popular and necessary in many knowledge discovery related areas [2]. The big data situation is either true for neural networks [3]. From the intuition, it is usually recognized that training over large scale samples leads better learning results than over small amount of samples. Thus, for those neural network-based applications, training large scale neural networks plays an important role in achieving optimal precisions and results.

In this paper, we design and implement cNeural, a customized parallel computing platform for training large scale neural networks. In cNeural, a training workflow can be divided into two phases: training data loading and training process executing. To reduce the time cost of data loading, we store the large scale training datasets in HBase, and concurrently load one of them into the memory of computing nodes across the cluster when needed. Also, a parallel in-memory computing framework is adopted for fast iterative training. During the entire training process, computing nodes need to communicate with each other for cooperation and further processing. We employ Apache Avro RPC to build an event-driven messaging communication framework in cNeural, for its high communication efficiency and rich data structures. Our platform can be deployed on commodity hardware, Amazon EC2, or even general PCs interconnected with network.

This paper is organized in eight sections. Section II describes the related work. In section III, we present the background of neural network along with the back propagation training algorithm. In section IV, we introduce the parallel training framework and algorithm used in cNeural. In section V, we describe data storage mechanism adopted to support fast training. Then, we illustrate the architecture overview and major components of cNeural in section VI. Section VII performs evaluations. Section VIII concludes this paper.

II. Related Work

Many researchers have been dedicating to implementing computationally expensive ANN algorithms on parallel or distributed computing systems. The related work can be traced back to the 70s of last century and the research in this area keep growing today.

In early time, researchers prefer to adopt special-purposed hardware to improve training speed, which is classified as neurohardware or neurocomputers in [6]. Glesner and Pochnuller [11] presented a general overview of special

978-1-4799-1293-3/13/$31.00 ©2013 IEEE

purpose hardware in their book. Special-purposed implementations can be fast and efficient. However, they offer very little flexibility and scalability. After the1990s, designing parallel neural networks over general-purposed architectures, such as parallel computing model or grid computing model, became the mainstream [12], [13]. These systems are mostly implemented on clusters and multi-processor computers. However, these previous work made few efforts on managing large scale training datasets. They usually focused on studying how to parallelize neural network training and only perform experiments with several thousands of training samples and Megabyte-class in data size [14], [15], [16], [17].

In recent years, some researchers study on training neural networks over big data. [10] stores large scale datasets on HDFS and trains them by MapReduce. However, Hadoop is designed for processing offline data-intensive problems but not for computation-intensive problems. Thus, the speed of training ANNs on Hadoop is slow. GPU has also been used for ANN training but the training dataset's size is limited to GPU's global memory [16]. Study in [18] performed the large scale unsupervised feature learning for building features from unlabeled data. They spent a lot of effort on training algorithms, such as model parallelism and asynchronous stochastic gradient descent. Unlike the studies above, cNeural not only considers the parallel algorithms for speedup neural network training, but also spent efforts on big data management to better support the fast execution of these parallel algorithms.

As Hadoop is not suitable for iterative processing, many studies proposed methods to improve it, such as Twister [19] and HaLoop [20]. They try to reduce the time cost on job initialization or support caching of data in-memory at nodes between iterations. [21] proposed Spark, a totally new distributed system for parallel in-memory computing. Compare with these processing engines, cNeural also implemented parallel neural network training algorithms in it. The underlying processing engine of cNeural holds the similar idea as in-memory computing. Moreover, we adopted a customized implementation for better support our on-top algorithms and applications.

III. Background

In this section we give a brief introduction to a widely used neural network training algorithm, the backpropagation algorithm. We take multilayer perceptron as a typical example to describe the training algorithm.

The *feed-forward, back propagation* neural network [4] is one of the most popular neural network architectures in use today [5]. [4] proved that three-layer feed-forward neural networks, such as multilayer perceptrons, trained by the backpropagation algorithm could approximate any continuous non-linear functions by arbitrary precision with enough hidden neurons. Thus, a three-layer feed-forward perceptron is introduced here for describing relevant algorithms. The structure of a three-layer perceptron is given in Fig. 1. It includes an input layer, a hidden layer and an output layer. The neurons in the same layer are not interlinked while the neurons in adjacent layers are fully connected with weights and biases.

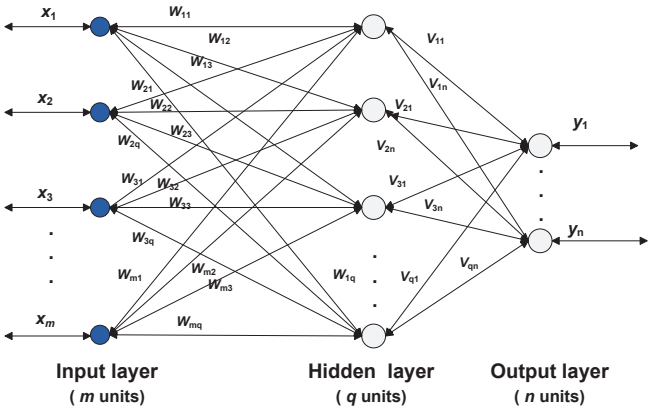

Figure 1. Structure of a three-layer multilayer perceptron with the backpropagation algorithm.

Backpropagation (BP) [31] with gradient-descent technique is one of the most widely used algorithms for supervised training multilayer feed-forward neural networks. The backpropagation algorithm has two phases, i.e. the forward phase and the backward phase.

In the forward phase, the input layer receives input signals and propagates the information to each of neurons in the hidden layer. Then, the hidden layer processes the information in local and finally propagates the processed information to the output layer. For an input vector $\boldsymbol{x} = (x_1, x_2, ..., x_m)$, the input and output information of each neuron in hidden layer, denoted as u_j and h_j , can be carried out by the (1) and (2) respectively.

$$u_j = \sum_{i=1}^{m} W_{ij} x_i + \theta_j \quad j = 1, 2, \cdots, q \tag{1}$$

$$h_j = f(u_j) = \frac{1}{1 + \exp(-u_j)} \quad j = 1, 2, \ldots, q \tag{2}$$

Where W_{ij} is the weights between input neuron i and hidden neuron j, and θ_j is the bias.

The output layer also needs to process the input information it gets from the hidden layer. The input l_k and output c_k of each neuron in the output layer is calculated using (3) and (4).

$$l_k = \sum_{j=1}^{q} V_{jk} h_j + \gamma_k \quad k = 1, 2, \cdots, n \tag{3}$$

$$c_k = f(l_k) = \frac{1}{1+\exp(-l_k)} \quad k = 1, 2, \ldots, n \tag{4}$$

Where V_{jk} is the weights between hidden neuron j and output neuron k, and γ_k is the bias.

That is the end of one-pass information forward process. The weights W, V and biases θ, γ does not change during forward stages. If the actual output of the neural network is equal to the expected output of the input vector, then a new input vector will be put into the neural network and the forward phase restarts, otherwise the algorithm enters the backward phase. The difference between the actual output and the expected output is called error.

During the backward phase, errors of neurons d_k in the output layer are computed using (5) at first. Then, errors of neurons e_j in the hidden layer are carried out using (6).

$$d_k = (y_k - c_k)c_k(1 - c_k) \quad k = 1, 2, \cdots, n \tag{5}$$

$$e_j = (\sum_{k=1}^{n} d_k V_{jk})h_j(1 - h_j) \quad j = 1, 2, \ldots, q \tag{6}$$

The errors are propagated backward from the output layer to the hidden layer and the connection weights between the layers are updated with the back propagated errors using (7), and the weights between the hidden layer and input layer are revised using (8).

$$\begin{aligned} V_{jk}(N+1) &= V_{jk}(N) + \alpha_1 d_k(N) h_j \\ \gamma_k(N+1) &= \gamma_k(N) + \alpha_1 d_k(N) \end{aligned} \tag{7}$$

$$\begin{aligned} W_{ij}(N+1) &= W_{ij}(N) + \alpha_2 e_j(N) x_i \\ \theta_j(N+1) &= \theta_j(N) + \alpha_2 e_j(N) \end{aligned} \tag{8}$$

In the above equations, where $i = 1, 2, \cdots, m$; $j = 1, 2, \cdots, q$ and $k = 1, 2, \cdots n$. α_1 and α_2 are the learning rates ranging from 0 to 1. N is the training epoch ID.

Generally, the BP algorithm has two modes for weights updating: the online mode and the batch mode. In the online mode, training samples are processed one by one. In the batch mode, the training process is carried out for all the training samples in batch. The generated ΔW (ΔW here represents the changed value of W, V, θ, γ between two epoches) of each sample is accumulated in one epoch. After that, the cumulative ΔW is used for revising the weights of connected layers together. The training work continues until the terminating condition is satisfied. The mostly adopted terminating condition is that the mean square error gets lower than the specified threshold or the training epoch reaches the limited round. To calculate the total error, the whole training dataset needs to be propagated through the neural network. This results in slow training speed for the backpropagation training algorithm when dealing with large training datasets.

IV. Parallel Neural Network Training Algorithm In cNeural

In this section, we firstly analyze the widely used parallel training strategies. Then, the parallel training algorithm in cNeural along with its parallel computing framework is introduced.

A. Analysis of Parallelization Strategies for Training Neural Network

There are many parallelization approaches to accelerate training neural networks [6]. Most approaches can be classified into two categories, namely the node parallelism and training dataset parallelism. The node parallelism is neural network structure oriented. These approaches achieve parallelism by mapping neurons into different computing nodes for pipeline processing. Each computing node only takes charge of a part of the computation of a neural network. The methods proposed by [7], [8], and [9] adopt this approach. Conversely, in training dataset parallelism, each computing node has the complete neural network in local and conducts computation for entire neural network. The training dataset is divided into several subsets and the subsets are assigned to different computing nodes for parallel processing.

Different parallelization approaches fit in different scenarios. For the node parallelism, every training sample needs to be handled down between computing nodes step by step for processing. It is usually used when training datasets are small and neural network structures are complicated. This kind of approaches is suitable to be implemented over the multi-core or many-core architectures which have low communication cost. When implemented in distributed systems with large amount of training samples, the system will be overwhelmed by the overhead of the I/O and cluster network communication costs. As the I/O and network communication are usually the major time costs in distributed environments, this kind of approaches is not efficient. Therefore, the node parallelism approach is not appropriate to be used in a distributed computing environment. Similar conclusions are also drawn in [10]. On the other side, for training data parallelism, each training data subset is processed on one computing node and does not need to be transmitted to the other computing nodes during the whole training process. Therefore, the training dataset parallelism approach is suitable for processing large scale training dataset in distributed systems, as it can significantly reduce data access and network communication costs.

B. Parallel BP Alogrithm and Computing Framework in cNeural

cNeural focuses on training large scale datasets. For reducing the time cost of accessing and transferring training data, we adopt the training dataset parallelism as our

basic parallel approach. We build a distributed and parallel computing framework that implements the batch mode backpropagation algorithm based on the training dataset parallelism approach. This parallel computing framework is in a typical master-worker parallel model. It includes one master node and n computing nodes as workers. The main duty of the master node is to coordinate the whole training process. The actual training process is conducted on computing nodes. Before training, the subsets of the training dataset are distributed to the computing nodes. Each computing node contains the entire neural network and takes charge of training its local training subset. The parallel training algorithm used in cNeural is described in Fig. 2.

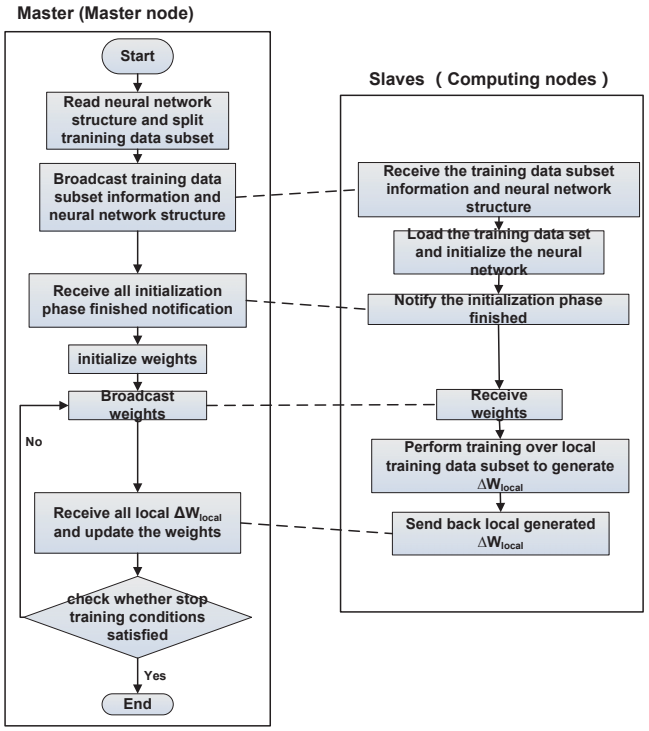

Figure 2. The parallel training algorithm used in cNeural. Dash line means synchronization point.

As shown in Fig. 2, the master node and computing nodes need to initialize themselves firstly. After the initialization phase, the master node broadcasts initial weights W to all computing nodes. When receiving the W, each computing node simultaneously performs the training task over its local training data subset. It conducts the processing of each sample in the subset for both the forward phase and backward phase. During local training, each computing node also needs to accumulate the ΔW_{local_i} of each local training sample. When finishing its training task, each computing node sends the ΔW_{local} of its local training data subset to the master node. On the master side, after receiving the ΔW_{local} from all the computing nodes, it applies all these ΔW_{local} to the W of the previous epoch to update the weights. The whole training process is iterative. Finally, it checks whether the training termination condition is satisfied. If satisfied, the whole training process stops, otherwise the next training epoch begins.

V. Training Data Storage and Loading

The time cost by loading and transferring large scale training datasets can be regarded as the major overhead of the entire training process [10]. In this section, we discuss the training data storage mechanism that adopted in cNeural to support fast large scale data loading and training.

The storage model of cNeural is shown in Fig. 3. We choose Apache HBase, a distributed data management system modeled after Google's BigTable, to store the training datasets. The training datasets are organized as tables in HBase. Each table contains a training data set with varying number of regions according to its size. For one training dataset, each sample is stored as a record in a table with the sample ID as the record rowkey. The content of the sample lies in the content field of the record. This way even large scale training datasets with billions of samples can be easily stored and managed. During initialization of a training process, computing nodes can access regions on different storage nodes simultaneously over network. Therefore, this underlying scalable and distributed storage can not only solve the problem for large scale dataset storage but also help to reduce the time cost of data loading by loading training datasets in parallel.

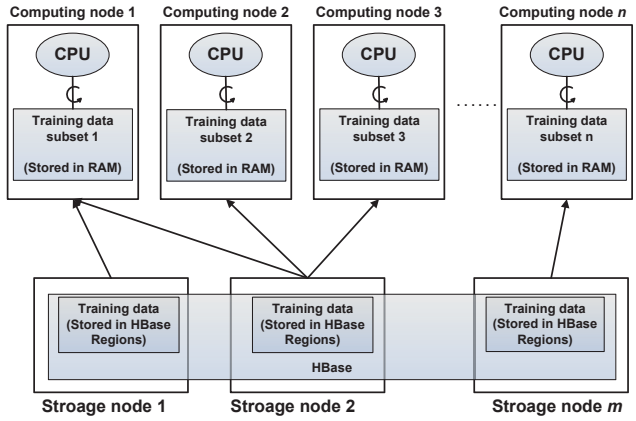

Figure 3. The data storage model of cNeural. Straight arrow lines indicate the transmission of training data. The loop arrows indicate that the data access repeats for many times during a training process.

Moreover, after a dataset is loaded for training, it needs to be frequently accessed during the thousands of training epochs. During the training process, each node resides its training data subset in memory. We refer this storage mechanism as *load once, read many*. In case the data is too large can be totally held in memory, cNeural will cache part

of them in the cluster's memory to avoid loading the whole data from HBase each time.In fact, cNeural can always offer enough memory storage capacity to store the training dataset in use. By avoiding repeated data loading and network transmission, this in-memory mechanism can significantly improve the training performance.

VI. System Architecture And Implementation

A. System Overview

Fig. 4 shows the system architecture of cNeural as well as the processing workflow when receiving a training request from the end user. **Logically**, cNeural consists of four types of modules, including a client node, a master node, n storage nodes, and m computing nodes. The latter two can also be referred as worker nodes. **In reality**, one machine can play multi nodes' role at the same time. For example, in our cluster, machines both act as storage nodes and computing nodes. Network communications among these nodes are implemented based on Avro RPC. The data storage management and transmission between computing nodes and storage nodes are supported by HBase. To better explain major components of cNeural along with their functions in more detail, we walk through a complete execution process of a training request from a user in next subsection.

B. Modules

1) Client Node: To perform a training job in cNeural, users need to set some important training configurations through a simple program. These configurations include the table name of training data stored in HBase, neural network settings such as the number of hidden layers, neuron numbers in each layer and training termination conditions. After finishing the configuration, the client node packs related information to generate a training job and submit it to the master node. If the job is submitted successfully, the client node waits for the training states and results generated by cNeural. Users can watch the online status of the training job on the client's console.

2) Master Node: The master node is the most complicated module in cNeural. It is responsible for managing and coordinating the whole training process. After receiving a training job, the master node splits it into several training tasks according to the size of the training dataset and the number of computing nodes. The training dataset is partitioned into n data subsets. Each computing node takes charge of one training task with its corresponding data subset. In cNeural, each machine can host k computing nodes. The number k is determined by the core number and memory capacity of the machine. In our implementation, $k = min\{core\_number, memory\_capacity/quota\}$, the $quota$ denotes the least memory capacity reserved for each computing node and this parameter is configurable.

After finished the job splitting and task assignment, the master node is responsible for coordinating the training tasks. Firstly, it notifies each computing node to load its corresponding training data subset from HBase. The training data loading happens among computing and storage nodes in parallel. After receiving the reply informing the completion of data loading from all computing nodes, then the master node starts to coordinate the parallel training process among computing nodes. It also takes charge of reporting running states and final results to the client node during the whole process.

3) Computing Node: The responsibility of the computing nodes is running training tasks. When receiving a training task request from the master node, the computing node sets up the neural network and loads its training data subset from the corresponding storage nodes through HBase interfaces. After that, it notifies the master node that it has done the initialize work. When all computing nodes finish the initialization work, the master node starts the training work by sending weights to each computing node. After that, at each epoch, the computing nodes execute the training tasks with the epoch's weights over its training data subset. During the entire training process, the training data subsets are resided in the memory of the computing nodes.

For each training sample in a training data subset, the computing node computes out ΔW_{local_i} and accumulates it. The average of all accumulated ΔW_{local_i} on each computing node will be calculated (denoted as ΔW_{local}) and sent to the master node. After receiving ΔW_{local} from all computing nodes, the master node updates the parameters by applying all ΔW_{local} to the W of last epoch. Then the master node starts a new training epoch by sending the updated parameters to each computing node. When the terminating conditions are satisfied, the whole iterative training process terminates and the all related resources on master node and computing nodes will be released.

4) Storage Node: In cNeural, we utilize HBase to store training datasets. The storage nodes of cNeural are region servers in HBase. Each region server can interact with the computing nodes independently. It can control the transmission of its local training data to the computing nodes without interacting to the master node in HBase. In cNeural, each training dataset is organized as a table that consists of many regions. These regions are stored across the region servers and can be moved to any region servers through programming APIs. We implement a utility to balance the regions of one table across the storage nodes for improving parallel data loading performance in cNeural.

C. Features

Here, we conclude some key features of cNeural as below:

- **Scalability.** When a new machine is added into cNeural, it takes a share of the training dataset from the existing machine. Adding more computing nodes leads less data for each computing node and faster processing speed .

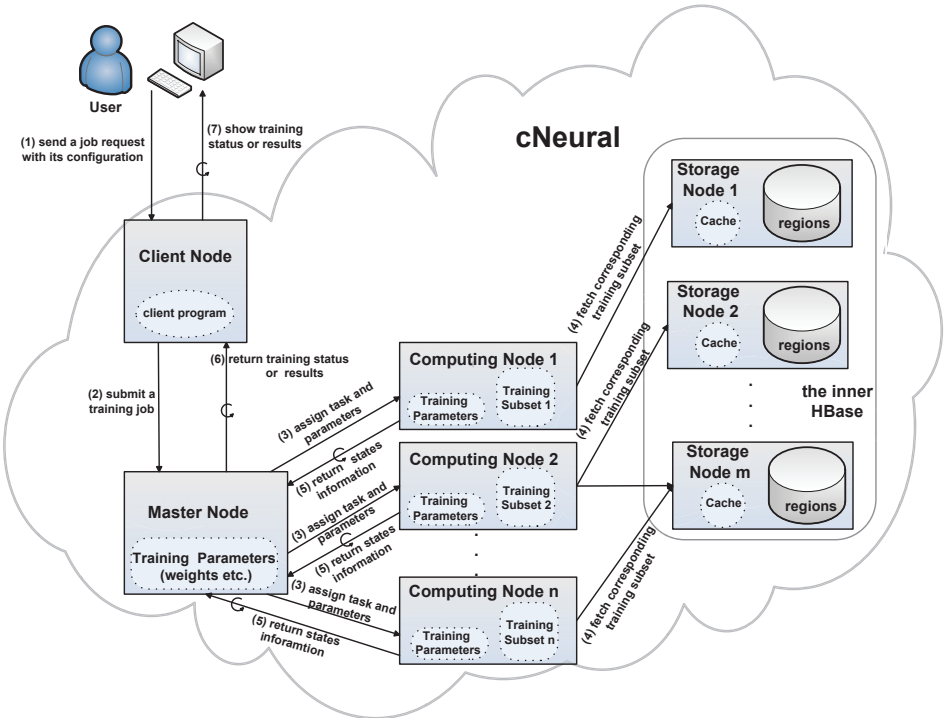

Figure 4. Architecture of cNeural and an overview of the training execution's workflow. The loop arrows indicate that the information are transferred continuously in a training job. Users can employ cNeural from simple interfaces without knowing detailed components and internal mechanisms.

- **Failure Recoverability.** At the end of each training epoch, cNeural saves the most recent parameters like weights into log files. A training job may fail for various faults like machine crash *etc*. During recovering process, cNenural only needs to restart the job with the most recent parameters recorded in log. By this method, the training work continues running at the most recent failure point rather than totally from the scratch.
- **Load Balance.** A computing node is a logical computing resource concept in cNeural. In reality, a machine can host several computing nodes according to its hardware capability. Also, in a heterogeneous cluster, machines can host different suitable number depends on their hardware resource capability. This way helps lead to load balance in cNeural.

VII. Evaluation

In section, we evaluate the performance of cNeural in detail. In cNeural, a training job has two phases: large scale training data loading and executing training process. Thus, we design two groups of experiments for evaluating the performance of them respectively. The first group of experiments are conducted to evaluate the performance of training data loading. The experiments in the second group are carried out to evaluate the training performance. In addition, we conduct several comparative experiments to compare the training performance of the MapReduce approach proposed by [10] with cNeural. All the execution time presented here is the average of 10 runs.

A. Experiment Setup

There are 37 nodes in our experimental cluster. Each node is equipped with 2 of the Intel(R) Quad Core E5620 Xeon(R) CPU at 2.4GHz, with 24GB memory and 4TB 7200r/s hard disks. All nodes are connected with each other by 1 Gigabit Ethernet. Among them, one is reserved as the master machine and the rest are slave machines. The operating system of the machines is Red Hat Linux 6.0. The version of the HBase adopted in cNeural is 0.92, and the version of employed Hadoop for comparative experiments is 1.0.3. Both cNeural and Hadoop run with OpenJDK 1.6 with the same JVM heap size 16 GB.

1) Datasets and Neural Network Configuration: For training dataset, we use the MNIST dataset with 2 million of samples for experiments (downloaded from http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html). MNIST dataset is a database of handwritten digits for training handwriting recognition systems. Each digit is a training pattern consisting of 784 size-normalized features and a classification label ranges from 0 to 9. In our test, the employed training datasets contain 0.5 million, 1 million and 2 million samples and their physical data sizes are 1.2 GB, 2.4 GB, and 4.8 GB stored in compressed coded style. During the training process they need to be decoded, and

the corresponding sizes are 4 to 5 times larger and reach around 20 GB. The training work is performed on a three-layer perceptron with the structure as 784-40-10, meaning 784 neurons in the input layer, 40 neurons in the hidden layer and 10 neurons in the output layer.

2) System Setup:

cNeural: The client node and master node of cNeural both run on the master machine. Each slave machine consists of 8 computing nodes and one storage node. Each storage node is maintained by an RegionServer daemon in HBase. Thus, there are 288 computing nodes in total. As each slave machine is configured 16 GB memory, each computing node has 2 GB memory, and it is enough for storing the training data subset in use.

MapReduce Approach: The master machine acts as the JobTracker and NameNode. And, each of the rest 36 slaves both act as TaskTracker and DataNode. Each machine is configured 8 slots. There are also 288 slots in total and each slot's Child JVM is also configured 2 GB as each computing node in cNeural.

B. Performance of Training Data Loading

In this experiment, cNeural is build up by 1, 2, 4, up to 36 machines respectively. Its inner HBase consists of 36 region servers. During data loading, the computing nodes access HBase concurrently. The experiment is performed on cNeural with different number of machines (each with 8 computing nodes) and different size of dataset.

The curves in Fig. 5 shows the load time for the different sizes of training samples respectively. The experiment with 2 million training samples on 1 machine failed to proceed due to insufficient memory to load all training data. Obviously, as the size of input samples increases, the time cost on training data loading increases too. When the input sample size is fixed, the loading speed increases almost linearly with machine number. This is because that the inner HBase provides concurrent data access, and when new machines added in the system, they takes a share of loading the training dataset. However, when the number of machines increases to some degree (etc. 20 in the case of test with 1 million training samples), the performance of data loading can hardly be further improved. This because the network transmission, I/O blocking on storage nodes in HBase become bottlenecks.

To get valid results, neural network training usually needs to run thousands of epochs. And, in cNeural, the computing nodes only need to load data from HBase once. Fig. 6 illustrates the percentage of time cost on training data loading with varying training epochs. The time cost on training data loading is fixed. Therefore, the proportion of training data loading time decreases as training epochs increases. With 36 machines, when training epochs reach 200, the time cost in training data loading accounts only 0.3% of the total execution time. The experimental results demonstrate that the time cost of data loading is very little in cNeural.

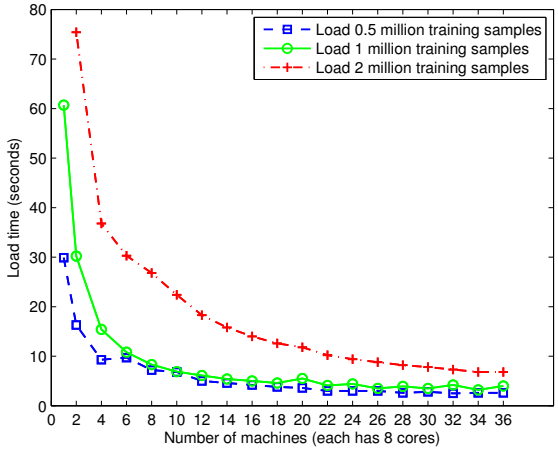

Figure 5. Performance of training data loading with different number of machines on different size of training samples. The experiment failed to load 2 million training samples with only one machine. This is the case that the training work can not be processed on a single machine.

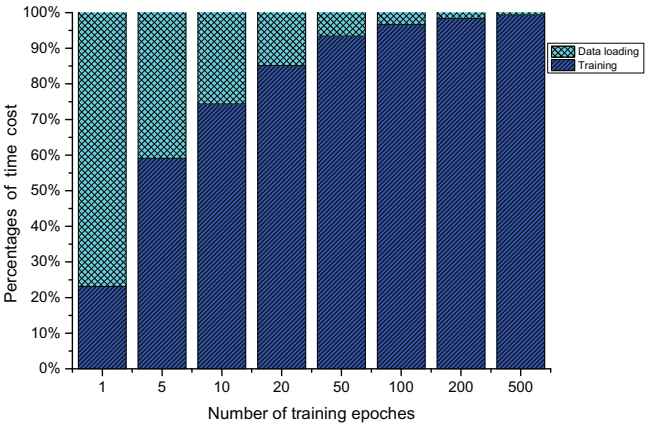

Figure 6. Percentages of time cost of data loading and training with different training epochs on 1 million training samples and 36 machines

C. Performance of Executing Training Process

The performance of executing training process is in shown in Fig. 7. We can see that, with the same number of machines, training 0.5 million samples is always about one time faster than training 1 million samples and three times faster than training 2 million samples. This indicates that time cost of our parallel algorithm increases near linear with the number of input training samples. On the other perspective, if the size of training data is fixed, the training speed of cNeural is much improved when more machines are utilized. For 1 million training samples, when the numbers of machines are 2, 4, 8, 16, 32, the respective time costs of one training epoch are 18.3 seconds, 9.2 seconds, 4.7 seconds, 2.4 seconds, 1.3 seconds, almost getting half of time cost decrease. This shows that the training process of

cNeural achieves good speedup performance.

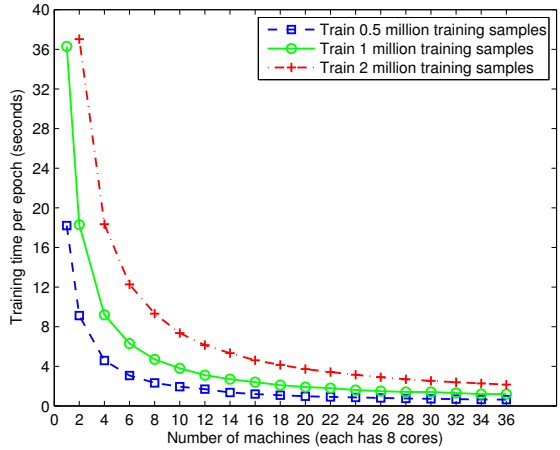

Figure 7. Performance of each epoch's training time cost in cNeural with various numbers of machines and different sizes of training samples.

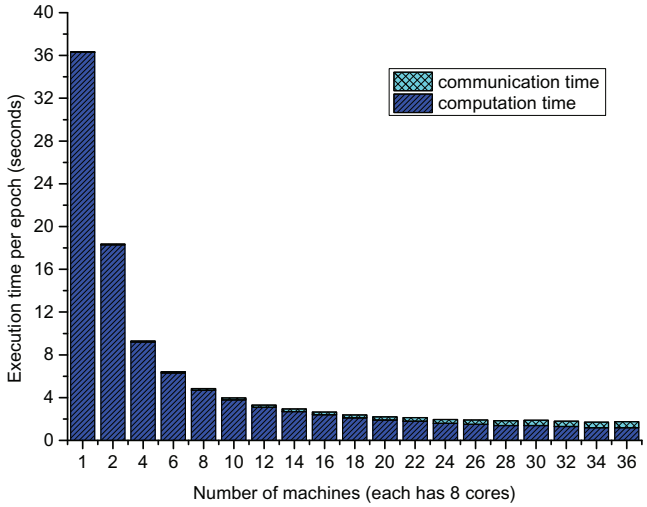

Figure 8. Time cost of communication and computation during a training epoch with different number of machines.

The histogram in Fig. 8 shows the time cost by communication and computation during each epoch with different machines used respectively. When more machines adopted, the proportion of communication time cost increases. However, it always only occupies a small proportion. Moreover, the whole execution time still decreases.

D. Comparative Experiments

We compared the performance of cNeural with the approach proposed by [10] as they also target towards training neural networks with large scale training datasets. This approach adopts HDFS to store large scale training data and uses MapReduce as the parallel processing engine. The comparative experiments are conducted with identical number of machines and the same size of training dataset. Both the execution performance and speed scalability are evaluated here.

Table I
EXECUTION TIME (IN SECOND.) ON MAPREDUCE APPROACH AND CNEURAL WITH 1-MILLION SAMPLES

# Machine	MapReduce appraoch	cNerual	Speedup ratio
1	784.0	36.3	21.6
6	177.0	6.3	28.2
12	107.0	3.1	34.1
18	95.0	2.4	40
24	82.0	1.6	50.5
30	79.0	1.4	57.9
36	60.0	1.2	52.2

1) Training Execution Performance: As shown in Table I, cNeural achieves around 50 times speedup over the MapReduce approach under the same conditions. This improvement can be attributed to two facts. First, cNeural resides the training data across computer nodes' memory when training, however, the MapReduce approach needs to reload the training data from hard disks in each training epoch. Second, in messaging communication, cNerual adopts a event-driven model, while the MapReduce approach uses the heartbeat polling model. The former makes the training process more compact.

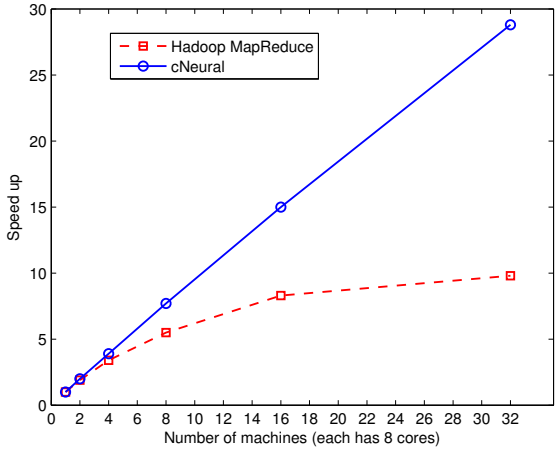

Figure 9. Speedup salability of cNeural and Hadoop MapReduce approach with different number of machines.

2) Speedup Scalability: We also tested the speedup scalability of both systems with varying machine numbers. The experimental results are shown in Fig. 9. On one side, they both achieve good speedup scalability within 10 nodes in this algorithm. On the other side, the speedup of the MapReduce approach failed to keep linear as cNeural because of the

overhead of job initialization in Hadoop is much larger than cNeural.

VIII. Conclusion and Future Work

The past several years have witnessed an ever-increasing growth speed of data. To address large scale neural network training problems, in this paper we proposed a customized parallel computing platform called cNeural. Different from many previous studies, cNeural is designed and built on perspective of the whole architecture, from the distributed storage system at the bottom level to the parallel computing framework and algorithm on the top level. Experimental results show that cNeural is able to train neural networks over millions of samples and around 50 times faster than Hadoop with dozens of machines.

In the future, we plan to develop and add more neural network algorithms such as deep belief networks into cNeural in order to make further support training large scale neural networks for various problems. Finally, with more technical work such as GUI done, we would like to make it as a toolbox and open source it.

Acknowledgment

This work is funded in part by China NSF Grants (No. 61223003), the National High Technology Research and Development Program of China (863 Program)(No. 2011AA01A202) and the USA Intel Labs University Research Program.

References

[1] C. Bishop, *Neural networks for pattern recognition*. Clarendon press Oxford, 1995.

[2] J. Collins, "Sailing on an ocean of 0s and 1s," *Science*, vol. 327, no. 5972, pp. 1455–1456, 2010.

[3] S. Haykin, *Neural networks and learning machines*. Englewood Cliffs, NJ: Prentice Hall, 2009.

[4] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Proc. Int. Joint Conf. on Neural Networks,IJCNN.* IEEE, 1989, pp. 593–605.

[5] Y. Loukas., "Artificial neural networks in liquid chromatography: Efficient and improved quantitative structure-retention relationship models," *Journal of Chromatography A*, vol. 904, pp. 119–129, 2000.

[6] N. Serbedzija, "Simulating artificial neural networks on parallel architectures," *Computer*, vol. 29, no. 3, pp. 56–63, 1996.

[7] M. Pethick, M. Liddle, P. Werstein, and Z. Huang, "Parallelization of a backpropagation neural network on a cluster computer," in *Proc. Int. Conf. on parallel and distributed computing and systems (PDCS)*, 2003.

[8] K. Ganeshamoorthy and D. Ranasinghe, "On the performance of parallel neural network implementations on distributed memory architectures," in *Proc. Int. Symp. on Cluster Computing and the Grid (CCGRID)*. IEEE, 2008, pp. 90–97.

[9] S. Suresh, S. Omkar, and V. Mani, "Parallel implementation of back-propagation algorithm in networks of workstations," *IEEE Trans. Parallel and Distributed Systems*, vol. 16, no. 1, pp. 24–34, 2005.

[10] Z. Liu, H. Li, and G. Miao, "Mapreduce-based backpropagation neural network over large scale mobile data," in *Proc. Int. Conf. on Natural Computation (ICNC)*, vol. 4. IEEE, 2010, pp. 1726–1730.

[11] M. Glesner and W. Pöchmüller, *Neurocomputers: an overview of neural networks in VLSI*. CRC Press, 1994.

[12] Y. Bo and W. Xun, "Research on the performance of grid computing for distributed neural networks," *International Journal of Computer Science and Netwrok Security*, vol. 6, no. 4, pp. 179–187, 2006.

[13] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," *Advances in neural information processing systems*, vol. 19, pp. 281–288, 2007.

[14] U. Seiffert, "Artificial neural networks on massively parallel computer hardware," *Neurocomputing*, vol. 57, pp. 135–150, 2004.

[15] D. Calvert and J. Guan, "Distributed artificial neural network architectures," in *Proc. Int. Symp. on High Performance Computing Systems and Applications*. IEEE, 2005, pp. 2–10.

[16] H. Kharbanda and R. Campbell, "Fast neural network training on general purpose computers," in *Proc. Int. Conf. on High Performance Computing (HiPC)*. IEEE, 2011.

[17] U. Lotrič and e. a. Dobnikar, A., "Parallel implementations of feed-forward neural network using mpi and c# on ṅet platform," in *Proc. Int. Conf. on Adaptive and Natural Computing Algorithms*. Coimbra, 2005, pp. 534–537.

[18] Q. V. Le, R. Monga, and M. e. a. Devin, "Building high-level features using large scale unsupervised learning," in *Proc. Int. Conf. on Machine Learning (ICML)*. ACM, 2012, pp. 2–16.

[19] J. Ekanayake and H. e. a. Li, "Twister: a runtime for iterative mapreduce," in *Proc. of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 2010, pp. 810–818.

[20] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proc. of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[21] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. USENIX Conf. on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–16.