Hindawi Publishing Corporation
Computational Intelligence and Neuroscience
Volume 2015, Article ID 297672, 13 pages
http://dx.doi.org/10.1155/2015/297672

Research Article

MapReduce Based Parallel Neural Networks in Enabling Large Scale Machine Learning

Yang Liu,[1] Jie Yang,[1] Yuan Huang,[1] Lixiong Xu,[1] Siguang Li,[2] and Man Qi[3]

[1]*School of Electrical Engineering and Information, Sichuan University, Chengdu 610065, China*
[2]*The Key Laboratory of Embedded Systems and Service Computing, Tongji University, Shanghai 200092, China*
[3]*Department of Computing, Canterbury Christ Church University, Canterbury, Kent CT1 1QU, UK*

Correspondence should be addressed to Lixiong Xu; xulixiong@scu.edu.cn

Received 28 May 2015; Accepted 11 August 2015

Academic Editor: Ladislav Hluchy

Artificial neural networks (ANNs) have been widely used in pattern recognition and classification applications. However, ANNs are notably slow in computation especially when the size of data is large. Nowadays, big data has received a momentum from both industry and academia. To fulfill the potentials of ANNs for big data applications, the computation process must be speeded up. For this purpose, this paper parallelizes neural networks based on MapReduce, which has become a major computing model to facilitate data intensive applications. Three data intensive scenarios are considered in the parallelization process in terms of the volume of classification data, the size of the training data, and the number of neurons in the neural network. The performance of the parallelized neural networks is evaluated in an experimental MapReduce computer cluster from the aspects of accuracy in classification and efficiency in computation.

1. Introduction

Recently, big data has received a momentum from both industry and academia. Many organizations are continuously collecting massive amounts of datasets from various sources such as the World Wide Web, sensor networks, and social networks. In [1], big data is defined as a term that encompasses the use of techniques to capture, process, analyze, and visualize potentially large datasets in a reasonable time frame not accessible to standard IT technologies. Basically, big data is characterized with three Vs [2]:

(i) Volume: the sheer amount of data generated.

(ii) Velocity: the rate at which the data is being generated.

(iii) Variety: the heterogeneity of data sources.

Artificial neural networks (ANNs) have been widely used in pattern recognition and classification applications. Back-propagation neural network (BPNN), the most popular one of ANNs, could approximate any continuous nonlinear functions by arbitrary precision with an enough number of neurons [3]. Normally, BPNN employs the back-propagation algorithm for training which requires a significant amount of time when the size of the training data is large [4]. To fulfill the potentials of neural networks in big data applications, the computation process must be speeded up with parallel computing techniques such as the Message Passing Interface (MPI) [5, 6]. In [7], Long and Gupta presented a scalable parallel artificial neural network using MPI for parallelization. It is worth noting that MPI was designed for data intensive applications with high performance requirements. MPI provides little support in fault tolerance. If any fault happens, an MPI computation has to be started from the beginning. As a result, MPI is not suitable for big data applications, which would normally run for many hours during which some faults might happen.

This paper presents a MapReduce based parallel back-propagation neural network (MRBPNN). MapReduce has become a de facto standard computing model in support of big data applications [8, 9]. MapReduce provides a reliable, fault-tolerant, scalable, and resilient computing framework for storing and processing massive datasets. MapReduce

scales well with ever increasing sizes of datasets due to its use of hash keys for data processing and the strategy of moving computation to the closest data nodes. In MapReduce, there are mainly two functions which are the Map function (mapper) and the Reduce function (reducer). Basically, a mapper is responsible for actual data processing and generates intermediate results in the form of ⟨key, value⟩ pairs. A reducer collects the output results from multiple mappers with secondary processing including sorting and merging the intermediate results based on the key values. Finally the Reduce function generates the computation results.

We present three MRBPNNs (i.e., MRBPNN_1, MRBPNN_2, and MRBPNN_3) to deal with different data intensive scenarios. MRBPNN_1 deals with a scenario in which the dataset to be classified is large. The input dataset is segmented into a number of data chunks which are processed by mappers in parallel. In this scenario, each mapper builds the same BPNN classifier using the same set of training data. MRBPNN_2 focuses on a scenario in which the volume of the training data is large. In this case, the training data is segmented into data chunks which are processed by mappers in parallel. Each mapper still builds the same BPNN but uses only a portion of the training dataset to train the BPNN. To maintain a high accuracy in classification, we employ a bagging based ensemble technique [10] in MRBPNN_2. MRBPNN_3 targets a scenario in which the number of neurons in a BPNN is large. In this case, MRBPNN_3 fully parallelizes and distributes the BPNN among the mappers in such a way that each mapper employs a portion of the neurons for training.

The rest of the paper is organized as follows. Section 2 gives a review on the related work. Section 3 presents the designs and implementations of the three parallel BPNNs using the MapReduce model. Section 4 evaluates the performance of the parallel BPNNs and analyzes the experimental results. Section 5 concludes the paper.

2. Related Work

ANNs have been widely applied in various pattern recognition and classification applications. For example, Jiang et al. [11] employed a back-propagation neural network to classify high resolution remote sensing images to recognize roads and roofs in the images. Khoa et al. [12] proposed a method to forecast the stock price using BPNN.

Traditionally, ANNs are employed to deal with a small volume of data. With the emergence of big data, ANNs have become computationally intensive for data intensive applications which limits their wide applications. Rizwan et al. [13] employed a neural network on global solar energy estimation. They considered the research as a big task, as traditional approaches are based on extreme simplicity of the parameterizations. A neural network was designed which contains a large number of neurons and layers for complex function approximation and data processing. The authors reported that in this case the training time will be severely affected. Wang et al. [14] pointed out that currently large scale neural networks are one of the mainstream tools for big data analytics. The challenge in processing big data with large scale neural networks includes two phases which are the training phase and the operation phase. To speed up the computations of neural networks, there are some efforts that try to improve the selection of initial weights [15] or control the learning parameters [16] of neural networks. Recently, researchers have started utilizing parallel and distributed computing technologies such as cloud computing to solve the computation bottleneck of a large neural network [17–19]. Yuan and Yu [20] employed cloud computing mainly for exchange of privacy data in a BPNN implementation in processing ciphered text classification tasks. However, cloud computing as a computing paradigm simply offers infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS). It is worth noting that cloud computing still needs big data processing models such as the MapReduce model to deal with data intensive applications. Gu et al. [4] presented a parallel neural network using in-memory data processing techniques to speed up the computation of the neural network but without considering the accuracy aspect of the implemented parallel neural network. In this work, the training data is simply segmented into data chunks which are processed in parallel. Liu et al. [21] presented a MapReduce based parallel BPNN in processing a large set of mobile data. This work further employs AdaBoosting to accommodate the loss of accuracy of the parallelized neural network. However, the computationally intensive issue may exist not only at the training phase but also at the classification phase. In addition, AdaBoosting is a popular sampling technique; it may enlarge the weights of wrongly classified instances which would deteriorate the algorithm accuracy.

3. Parallelizing Neural Networks

This section presents the design details of the parallelized MRBPNN_1, MRBPNN_2, and MRBPNN_3. First, a brief review of BPNN is introduced.

3.1. Back-Propagation Neural Network. Back-propagation neural network is a multilayer feed forward network which trains the training data using an error back-propagation mechanism. It has become one of the most widely used neural networks. BPNN can perform a large volume of input-output mappings without knowing their exact mathematical equations. This benefits from the gradient-descent feature of its back-propagation mechanism. During the error propagation, BPNN keeps tuning the parameters of the network until it adapts to all input instances. A typical BPNN is shown in Figure 1, which consists of an arbitrary number of inputs and outputs.

Generally speaking, a BPNN can have multiple network layers. However, it has been widely accepted that a three-layer BPNN would be enough to fit the mathematical equations which approximate the mapping relationships between inputs and outputs [3]. Therefore, the topology of a BPNN usually contains three layers: input layer, one hidden layer, and output layer. The number of inputs in the input layer is mainly determined by the number of elements in an input eigenvector; for instance, let $\mathbf{s}$ denote an input instance:

$$\mathbf{s} = \{a_1, a_2, a_3, \ldots, a_n\}. \quad (1)$$

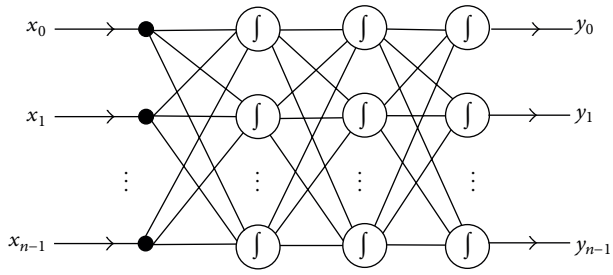


Figure 1: The structure of a typical BPNN.

Then, the number of inputs is n. Similarly, the number of neurons in the output layer is determined by the number of classifications. And the number of neurons in the hidden layer is determined by users. Every input of a neuron has a weight w_{ij}, where i and j represent the source and destination of the input. Each neuron also maintains an optional parameter θ_j which is actually a bias for varying the activity of the jth neuron in a layer. Therefore, let $o_{j'}$ denote the output from a previous neuron and let o_j denote the output of this layer; the input I_j of neurons located in both the hidden and output layer can be represented by

$$I_j = \sum_i w_{ij} o_{j'} + \theta_j. \tag{2}$$

The output of a neuron is usually computed by the sigmoid function, so the output o_j can be computed by

$$o_j = \frac{1}{1 + e^{-I_j}}. \tag{3}$$

After the feed forward process is completed, the back-propagation process starts. Let Err_j represent the error-sensitivity and let t_j represent the desirable output of neuron j in the output layer; thus,

$$\mathrm{Err}_j = o_j \left(1 - o_j\right) \left(t_j - o_j\right). \tag{4}$$

Let Err_k represent the error-sensitivity of one neuron in the last layer and let w_{kj} represent its weight; thus, Err_j of a neuron in the other layers can be computed using

$$\mathrm{Err}_j = o_j \left(1 - o_j\right) \sum_k \mathrm{Err}_k w_{kj}. \tag{5}$$

After Err_j is computed, the weights and biases of each neuron are tuned in back-propagation process using

$$\begin{aligned} \Delta w_{ij} &= \mathrm{Err}_j o_j, \\ w_{ij} &= w_{ij} + \Delta w_{ij}, \\ \Delta \theta_j &= \mathrm{Err}_j, \\ \theta_j &= \theta_j + \Delta \theta_j. \end{aligned} \tag{6}$$

After the first input vector finishes tuning the network, the next round starts for the following input vectors. The input keeps training the network until (7) is satisfied for a single output or (8) is satisfied for multiple outputs:

$$\min\left(E\left[e^2\right]\right) = \min\left(E\left[(t-o)^2\right]\right), \tag{7}$$

$$\min\left(E\left[e^T e\right]\right) = \min\left(E\left[(t-o)^T (t-o)\right]\right). \tag{8}$$

3.2. MapReduce Computing Model. MapReduce has become the de facto standard computing model in dealing with data intensive applications using a cluster of commodity computers. Popular implementations of the MapReduce computing model include Mars [22], Phoenix [23], and Hadoop framework [24, 25]. The Hadoop framework has been widely taken up by the community due to its open source feature. Hadoop has its Hadoop Distributed File System (HDFS) for data management. A Hadoop cluster has one name node (Namenode) and a number of data nodes (Datanodes) for running jobs. The name node manages the metadata of the cluster whilst a data node is the actual processing node. The Map functions (mappers) and Reduce functions (reducers) run on the data nodes. When a job is submitted to a Hadoop cluster, the input data is divided into small chunks of an equal size and saved in the HDFS. In terms of data integrity, each data chunk can have one or more replicas according to the cluster configuration. In a Hadoop cluster, mappers copy and read data from either remote or local nodes based on data locality. The final output results will be sorted, merged, and generated by reducers in HDFS.

3.3. The Design of MRBPNN_1. MRBPNN_1 targets the scenario in which BPNN has a large volume of testing data to be classified. Consider a testing instance $s_i = \{a_1, a_2, a_3, \ldots, a_{in}\}$, $s_i \in S$, where

(i) s_i denotes an instance;

(ii) S denotes a dataset;

(iii) in denotes the length of s_i; it also determines the number of inputs of a neural network;

(iv) the inputs are capsulated by a format of $\langle \mathrm{instance}_k, \mathrm{target}_k, \mathrm{type} \rangle$;

(v) $\mathrm{instance}_k$ represents s_i, which is the input of a neural network;

(vi) target_k represents the desirable output if $\mathrm{instance}_k$ is a training instance;

(vii) type field has two values, "train" and "test," which marks the type of $\mathrm{instance}_k$; if "test" value is set, target_k field should be left empty.

Files which contain instances are saved into HDFS initially. Each file contains all the training instances and a portion of the testing instances. Therefore, the file number n determines the number of mappers to be used. The file content is the input of MRBPNN_1.

When the algorithm starts, each mapper initializes a neural network. As a result, there will be n neural networks in the cluster. Moreover, all the neural networks have exactly the same structure and parameters. Each mapper reads data in

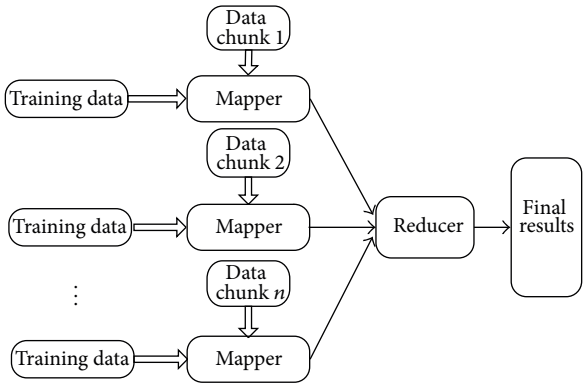


Figure 2: MRBPNN_1 architecture.

the form of $\langle \text{instance}_k, \text{target}_k, \text{type} \rangle$ from a file and parses the data records. If the value of type field is "train," instance_k is input into the input layer of the neural network. The network computes the output of each layer using (2) and (3), until the output layer generates an output which indicates the completion of the feed forward process. And then the neural network in each mapper starts the back-propagation process. It computes and updates new weights and biases for its neurons using (4) to (6). The neural network inputs instance_{k+1}. Repeat the feed forward and back-propagation process until all the instances which are labeled as "train" are processed and the error is satisfied.

Each mapper starts classifying instances labeled as "test" by running the feed forward process. As each mapper only classifies a portion of the entire testing dataset, the efficiency is improved. At last, each mapper outputs intermediate output in the form of $\langle \text{instance}_k, o_{jm} \rangle$, where instance_k is the key and o_{jm} represents the output of the mth mapper.

One reducer starts collecting and merging all the outputs of the mappers. Finally, the reducer outputs $\langle \text{instance}_k, o_{jm} \rangle$ into HDFS. In this case, o_{jm} represents the final classification result of instance_k. Figure 2 shows the architecture of MRBPNN_1 and Algorithm 1 shows the pseudocode.

3.4. The Design of MRBPNN_2. MRBPNN_2 focuses on the scenario in which a BPNN has a large volume of training data. Consider a training dataset S with a number of instances. As shown in Figure 3, MRBPNN_2 divides S into n data chunks of which each data chunk s_i is processed by a mapper for training, respectively:

$$S = \bigcup_{1}^{n} s_i, \quad \{\forall s \in s_i \mid s \notin s_n,\ i \neq n\}. \tag{9}$$

Each mapper in the Hadoop cluster maintains a BPNN, and each s_i is considered as the input training data for the neural network maintained in mapper_i. As a result, each BPNN in a mapper produces a classifier based on the trained parameters:

$$(\text{mapper}_i, \text{BPNN}_i, s_i) \longrightarrow \text{classifier}_i. \tag{10}$$

To reduce the computation overhead, each classifier_i is trained with a part of the original training dataset. However, a critical issue is that the classification accuracy of a mapper will be significantly degraded using only a portion of the training data. To solve this issue, MRBPNN_2 employs ensemble technique to maintain the classification accuracy by combining a number of weak learners to create a strong learner.

3.4.1. Bootstrapping. Training diverse classifiers from a single training dataset has been proven to be simple compared with the case of finding a strong learner [26]. A number of techniques exist for this purpose. A widely used technique is to resample the training dataset based on bootstrap aggregating such as bootstrapping and majority voting. This can reduce the variance of misclassification errors and hence increases the accuracy of the classifications.

As mentioned in [26], balanced bootstrapping can reduce the variance when combining classifiers. Balanced bootstrapping ensures that each training instance equally appears in the bootstrap samples. It might not be always the case that each bootstrapping sample contains all the training instances. The most efficient way of creating balanced bootstrap samples is to construct a string of instances $X_1, X_2, X_3, \ldots, X_n$ repeating B times so that a sequence of $Y_1, Y_2, Y_3, \ldots, Y_{Bn}$ can be achieved. A random permutation p of the integers from 1 to B_n is taken. Therefore, the first bootstrapping sample can be created from $Y_p(1), Y_p(2), Y_p(3), \ldots, Y_p(n)$. In addition, the second bootstrapping sample is created from $Y_p(n+1), Y_p(n+2), Y_p(n+3), \ldots, Y_p(2n)$ and the process continues until $Y_p((B-1)n+1), Y_p((B-1)n+2), Y_p((B-1)n+3), \ldots, Y_p(Bn)$ is the Bth bootstrapping sample. The bootstrapping samples can be used in bagging to increase the accuracy of classification.

3.4.2. Majority Voting. This type of ensemble classifiers performs classifications based on the majority votes of the base classifiers [26]. Let us define the prediction of the ith classifier P_i as $P_{i,j} \in \{1, 0\}$, $i = 1, \ldots, I$ and $j = 1, \ldots, c$, where I is the number of classifiers and c is the number of classes. If the ith classifier chooses class j, then $P_{i,j} = 1$; otherwise, $P_{i,j} = 0$. Then, the ensemble prediction for class k is computed using

$$P_{i,k} = \max_{j=1}^{c} \sum_{i=1}^{I} P_{i,j}. \tag{11}$$

3.4.3. Algorithm Design. At the beginning, MRBPNN_2 employs balanced bootstrapping to generate a number of subsets of the entire training dataset:

$$\text{balanced bootstrapping} \longrightarrow \{S_1, S_2, S_3, \ldots, S_n\}, \quad \bigcup_{i=1}^{n} S_i = S, \tag{12}$$

where S_i represents the ith subset, which belongs to entire dataset S. n represents the total number of subsets.

Each S_i is saved in one file in HDFS. Each instance $s_k = \{a_1, a_2, a_3, \ldots, a_{in}\}$, $s_k \in S_i$, is defined in the format of $\langle \text{instance}_k, \text{target}_k, \text{type} \rangle$, where

(i) instance_k represents one bootstrapped instance s_k, which is the input of neural network;

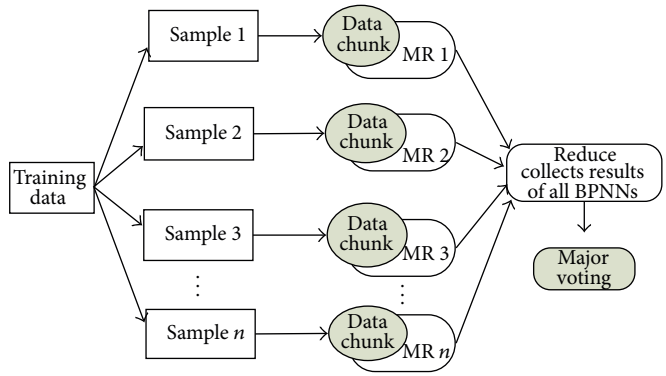


Figure 3: MRBPNN_2 architecture.

Input: S, T **Output**: C
m mappers one reducer
(1) Each mapper constructs one BPNN with in inputs, o outputs, h neurons in hidden layer
(2) Initialize $w_{ij} = \text{random}_{1ij} \in (-1, 1)$, $\theta_j = \text{random}_{2j} \in (-1, 1)$
(3) $\forall s \in S, s_i = \{a_1, a_2, a_3, \ldots, a_{in}\}$
Input $a_i \rightarrow in_i$, neuron_j in hidden layer computes

$$I_{jh} = \sum_{i=1}^{in} a_i \cdot w_{ij} + \theta_j$$

$$o_{jh} = \frac{1}{1 + e^{I_{jh}}}$$

(4) Input $o_j \rightarrow \text{out}_i$, neuron_j in output layer computes

$$I_{jo} = \sum_{i=1}^{h} o_{jh} \cdot w_{ij} + \theta_j$$

$$o_{jo} = \frac{1}{1 + e^{I_{jo}}}$$

(5) In each output, compute

$$\text{Err}_{jo} = o_{jo}\left(1 - o_{jo}\right)\left(\text{target}_j - o_{jo}\right)$$

(6) In hidden layer, compute

$$\text{Err}_{jh} = o_{jh}\left(1 - o_{jh}\right)\sum_{i=1}^{o} \text{Err}_i w_{io}$$

(7) Update

$$w_{ij} = w_{ij} + \eta \cdot \text{Err}_j \cdot o_j$$
$$\theta_j = \theta_j + \eta \cdot \text{Err}_j$$

Repeat (3), (4), (5), (6), (7)
Until

$$\min\left(E\left[e^2\right]\right) = \min\left(E\left[\left(\text{target}_j - o_{jo}\right)^2\right]\right)$$

Training terminates
(8) Divide T into $\{T_1, T_2, T_3, \ldots, T_m\}$, $\bigcup_{i=0}^{m} T_i = T$
(9) Each mapper inputs $t_j = \{a_1, a_2, a_3, \ldots, a_{in}\}$, $t_j \in T_i$
(10) Execute (3), (4)
(11) Mapper outputs $\langle t_j, o_j \rangle$
(12) Reducer collects and merges all $\langle t_j, o_j \rangle$
Repeat (9), (10), (11), (12)
Until T is traversed.
(13) Reducer outputs C
End

Algorithm 1: MRBPNN_1.

(ii) in represents the number of inputs of the neural network;

(iii) target_k represents the desirable output if instance_k is a training instance;

(iv) type field has two values, "train" and "test," which marks the type of instance_k; if "test" value is set, target_k field should be left empty.

When MRBPNN_2 starts, each mapper constructs one BPNN and initializes weights and biases with random values between −1 and 1 for its neurons. And then a mapper inputs one record in the form of $\langle \text{instance}_k, \text{target}_k, \text{type} \rangle$ from the input file.

The mapper firstly parses the data and retrieves the type of the instance. If the type value is "train," the instance is fed into the input layer. Secondly, each neuron in different layers computes its output using (2) and (3) until the output layer generates an output which indicates the completion of the feed forward process. Each mapper starts a back-propagation process and computes and updates weights and biases for neurons using (4) to (6). The training process finishes until all the instances marked as "train" are processed and error is satisfied. All the mappers start feed forwarding to classify the testing dataset. In this case, each neural network in a mapper generates the classification result of an instance at the output layer. Each mapper generates an intermediate output in the form of $\langle \text{instance}_k, o_{jm} \rangle$, where instance_k is the key and o_{jm} represents the outputs of the mth mapper.

Finally, a reducer collects the outputs of all the mappers. The outputs with the same key are merged together. The reducer runs majority voting using (11) and outputs the result of instance_k into HDFS in the form of $\langle \text{instance}_k, r_k \rangle$, where r_k represents the voted classification result of instance_k. Figure 3 shows the algorithm architecture and Algorithm 2 presents the pseudocode of MRBPNN_2.

3.5. The Design of MRBPNN_3. MRBPNN_3 aims at the scenario in which a BPNN has a large number of neurons. The algorithm enables an entire MapReduce cluster to maintain one neural network across it. Therefore, each mapper holds one or several neurons.

There are a number of iterations that exist in the algorithm with l layers. MRBPNN_3 employs a number of $l - 1$ MapReduce jobs to implement the iterations. The feed forward process runs in $l - 1$ rounds whilst the back-propagation process occurs only in the last round. A data format in the form of $\langle \text{index}_k, \text{instance}_n, w_{ij}, \theta_j, \text{target}_n, \{w_{ij}^2, \theta_j^2, \ldots, w_{ij}^{l-1}, \theta_j^{l-1}\} \rangle$ has been designed to guarantee the data passing between Map and Reduce operations, where

(i) index_k represents the kth reducer;

(ii) instance_n represents the nth training or testing instance of the dataset; one instance is in the form of $\text{instance}_n = \{a_1, a_2, \ldots, a_{in}\}$, where in is length of the instance;

(iii) w_{ij} represents a set of weights of an input layer, whilst θ_j represents the biases of the neurons in the first hidden layer;

(iv) target_n represents the encoded desirable output of a training instance instance_n;

(v) the list of $\{w_{ij}^2, \theta_j^2, \ldots, w_{ij}^{l-1}, \theta_j^{l-1}\}$ represents the weights and biases for next layers; it can be extended based on the layers of the network; for a standard three-layer neural network, this option becomes $\{w_{ijo}, \theta_{jo}\}$.

Before MRBPNN_3 starts, each instance and the information defined by data format are saved in one file in HDFS. The number of the layers is determined by the length of $\{w_{ij}^2, \theta_j^2 \ldots, w_{ij}^{l-1}, \theta_j^{l-1}\}$ field. The number of neurons in the next layer is determined by the number of files in the input folder. Generally, different from MRBPNN_1 and MRBPNN_2, MRBPNN_3 does not initialize an explicit neural network; instead, it maintains the network parameters based on the data defined in the data format.

When MRBPNN_3 starts, each mapper initially inputs one record from HDFS. And then it computes the output of a neuron using (2) and (3). The output is generated by a mapper, which labels index_k as a key and the neuron's output as a value in the form of

$$\langle \text{index}_k, o_j, \{w_{ij}^2, \theta_j^2 \ldots, w_{ij}^{l-1}, \theta_j^{l-1}\}, \text{target}_n \rangle, \text{ where } o_j \text{ represents the neuron's output.}$$

Parameter index_k can guarantee that the kth reducer collects the output, which maintains the neural network structure. It should be mentioned that if the record is the first one processed by MRBPNN_3, $\{w_{ij}^2, \theta_j^2 \ldots, w_{ij}^{l-1}, \theta_j^{l-1}\}$ will be also initialized with random values between −1 and 1 by the mappers. The kth reducer collects the results from the mappers in the form of $\langle \text{index}_{k'}, o_j, \{w_{ij}^2, \theta_j^2 \ldots, w_{ij}^{l-1}, \theta_j^{l-1}\}, \text{target}_n \rangle$. These k reducers generate k outputs. The $\text{index}_{k'}$ of the reducer output explicitly tells the k'th mapper to start processing this output file. Therefore, the number of neurons in the next layer can be determined by the number of reducer output files, which are the input data for the next layer neurons. Subsequently, mappers start processing their corresponding inputs by computing (2) and (3) using w_{ij}^2 and θ_j^2.

The above steps keep looping until reaching the last round. The processing of this last round consists of two steps. The first step is that mappers also process $\langle \text{index}_{k'}, o_j, \{w_{ij}^{l-1}, \theta_j^{l-1}\}, \text{target}_n \rangle$, compute neurons' outputs, and generate results in the forms of $\langle o_j, \text{target}_n \rangle$. One reducer collects the output results of all the mappers in the form of $\langle o_{j1}, o_{j2}, o_{j3}, \ldots, o_{jk}, \text{target}_n \rangle$. In the second step, the reducer executes the back-propagation process. The reducer computes new weights and biases for each layer using (4) to (6). MRBPNN_3 retrieves the previous outputs, weights, and biases from the input files of mappers, and then it writes the updated weights and biases $w_{ij}, \theta_j, \{w_{ij}^2, \theta_j^2, \ldots, w_{ij}^{l-1}, \theta_j^{l-1}\}$ into the initial input file in the form of $\langle \text{index}_k, \text{instance}_n, w_{ij}, \theta_j, \text{target}_n, \{w_{ij}^2, \theta_j^2, \ldots, w_{ij}^{l-1}, \theta_j^{l-1}\} \rangle$. The reducer reads the second instance in the form of $\langle \text{instance}_{n+1}, \text{target}_{n+1} \rangle$ for which the fields instance_n and target_n in the input file are replaced by instance_{n+1} and target_{n+1}. The training process continues until all the instances are processed and error is satisfied.

Input: S, T **Output**: C
n mappers one reducer
(1) Each mapper constructs one BPNN with in inputs, o outputs, h neurons in hidden layer
(2) Initialize $w_{ij} = \text{random}_{1ij} \in (-1, 1), \theta_j = \text{random}_{2j} \in (-1, 1)$
(3) Bootstrap $\{S_1, S_2, S_3, \ldots, S_n\}, \bigcup_{i=0}^{n} S_i = S$
(4) Each mapper inputs $s_k = \{a_1, a_2, a_3, \ldots, a_{in}\}, s_k \in S_i$
Input $a_i \rightarrow in_i$, neuron_j in hidden layer computes

$$I_{jh} = \sum_{i=1}^{in} a_i \cdot w_{ij} + \theta_j$$

$$o_{jh} = \frac{1}{1 + e^{I_{jh}}}$$

(5) Input $o_j \rightarrow \text{out}_i$, neuron_j in output layer computes

$$I_{jo} = \sum_{i=1}^{h} o_{jh} \cdot w_{ij} + \theta_j$$

$$o_{jo} = \frac{1}{1 + e^{I_{jo}}}$$

(6) In each output, compute

$$\text{Err}_{jo} = o_{jo}\left(1 - o_{jo}\right)\left(\text{target}_j - o_{jo}\right)$$

(7) In hidden layer, compute

$$\text{Err}_{jh} = o_{jh}\left(1 - o_{jh}\right)\sum_{i=1}^{o} \text{Err}_i w_{io}$$

(8) Update

$$w_{ij} = w_{ij} + \eta \cdot \text{Err}_j \cdot o_j$$
$$\theta_j = \theta_j + \eta \cdot \text{Err}_j$$

Repeat (3), (4), (5), (6), (7)
Until

$$\min\left(E\left[e^2\right]\right) = \min\left(E\left[\left(\text{target}_j - o_{jo}\right)^2\right]\right)$$

Training terminates
(9) Each mapper inputs $t_i = \{a_1, a_2, a_3, \ldots, a_{in}\}, t_i \in T$
(10)Execute (4), (5)
(11) Mapper outputs $\langle t_j, o_j \rangle$
(12) Reducer collects $\langle t_j, o_{jm} \rangle, m = (1, 2, \ldots, n)$
(13) For each t_j

$$\text{Compute } C = \max_{j=1}^{c} \sum_{i=1}^{I} o_{jm}$$

Repeat (9), (10), (11), (12), (13)
Until T is traversed.
(14) Reducer outputs C
End

Algorithm 2: MRBPNN_2.

For classification, MRBPNN_3 only needs to run the feed forwarding process and collects the reducer output in the form of $\langle o_j, \text{target}_n \rangle$. Figure 4 shows three-layer architecture of MRBPNN_3 and Algorithm 3 presents the pseudocode.

4. Performance Evaluation

We have implemented the three parallel BPNNs using Hadoop, an open source implementation framework of the MapReduce computing model. An experimental Hadoop cluster was built to evaluate the performance of the algorithms. The cluster consisted of 5 computers in which 4 nodes are Datanodes and the remaining one is Namenode. The cluster details are listed in Table 1.

Two testing datasets were prepared for evaluations. The first dataset is a synthetic dataset. The second is the Iris dataset which is a published machine learning benchmark dataset [27]. Table 2 shows the details of the two datasets.

Table 1: Cluster details.

Namenode	CPU: Core i7@3 GHz Memory: 8 GB SSD: 750 GB OS: Fedora
Datanodes	CPU: Core i7@3.8 GHz Memory: 32 GB SSD: 250 GB OS: Fedora
Network bandwidth	1 Gbps
Hadoop version	2.3.0, 32 bits

Input: S, T **Output**: C
m mappers m reducers
Initially each mapper inputs
$\langle \text{index}_k, \text{instance}_n, w_{ij}, \theta_j, \text{target}_n, w_{ijo}, \theta_{jo} \rangle$, $k = (1, 2, \ldots, m)$
(1) For $\text{instance}_n = \{a_1, a_2, a_3, \ldots, a_{in}\}$
Input $a_i \rightarrow in_i$, *neuron* in *mapper* computes

$$I_{jh} = \sum_{i=1}^{in} a_i \cdot w_{ij} + \theta_j$$

$$o_{jh} = \frac{1}{1 + e^{I_{jh}}}$$

(2) *Mapper* outputs *output*
$\langle \text{index}_k, o_{jh}, w_{ijo}, \theta_{jo}, \text{target}_n \rangle$, $r = (1, 2, \ldots, m)$
(3) kth reducer collects *output*
Output $\langle \text{index}_{k'}, o_{jh}, w_{ijo}, \theta_{jo}, \text{target}_n \rangle$, $k' = (1, 2, \ldots, m)$
(4) $\text{Mapper}_{k'}$ inputs
$\langle \text{index}_{k'}, o_{jh}, w_{ijo}, \theta_{jo}, \text{target}_n \rangle$
Execute (2), outputs $\langle o_{jo}, \text{target}_n \rangle$
(5) mth reducer collects and merges all $\langle o_{jo}, \text{target}_n \rangle$ from m mappers
Feed forward terminates
(6) In mth reducer
Compute
$\text{Err}_{jo} = o_{jo}\left(1 - o_{jo}\right)\left(\text{target}_n - o_{jo}\right)$
Using HDFS API, retrieve $w_{ij}, \theta_j, o_{jh}, \text{instance}_n$
Compute

$$\text{Err}_{jh} = o_{jh}\left(1 - o_{jh}\right)\sum_{i=1}^{o} \text{Err}_i w_{io}$$

(7) Update $w_{ij}, \theta_j, w_{ijo}, \theta_{jo}$

$$w_{ij} = w_{ij} + \eta \cdot \text{Err}_j \cdot o_j$$

$$\theta_j = \theta_j + \eta \cdot \text{Err}_j$$

into
$\langle \text{index}_k, \text{instance}_n, w_{ij}, \theta_j, \text{target}_n, w_{ijo}, \theta_{jo} \rangle$
Back propagation terminates
(8) Retrieve instance_{n+1} and target_{n+1}
Update into $\langle \text{index}_k, \text{instance}_{n+1}, w_{ij}, \theta_j, \text{target}_{n+1}, w_{ijo}, \theta_{jo} \rangle$
Repeat (1), (2), (3), (4), (5), (6), (7), (8), (9)
Until

$$\min\left(E\left[e^2\right]\right) = \min\left(E\left[\left(\text{target}_j - o_{jo}\right)^2\right]\right)$$

Training terminates
(9) Each mapper inputs $\langle \text{index}_k, \text{instance}_t, w_{ij}, \theta_j, \text{target}_n, w_{ijo}, \theta_{jo} \rangle$, $\text{instance}_t \in T$
(10) Execute (1), (2), (3), (4), (5), (6), outputs $\langle o_{jo}, \text{target}_n \rangle$
(11) mth reducer outputs C
End

Algorithm 3: MRBPNN_3.

Table 2: Dataset details.

Data type	Instance number	Instance length	Element range	Class number
Synthetic data	200	32	0 and 1	4
Iris data	150	4	(0, 8)	3

We implemented a three-layer neural network with 16 neurons in the hidden layer. The Hadoop cluster was configured with 16 mappers and 16 reducers. The number of instances was varied from 10 to 1000 for evaluating the precision of the algorithms. The size of the datasets was varied from 1 MB to 1 GB for evaluating the computation efficiency of the algorithms. Each experiment was executed five times and the final result was an average. The precision p is computed using

$$p = \frac{r}{r + w} \times 100\%, \tag{13}$$

where r represents the number of correctly recognized instances. w represents the number of wrongly recognized instances. p represents the precision.

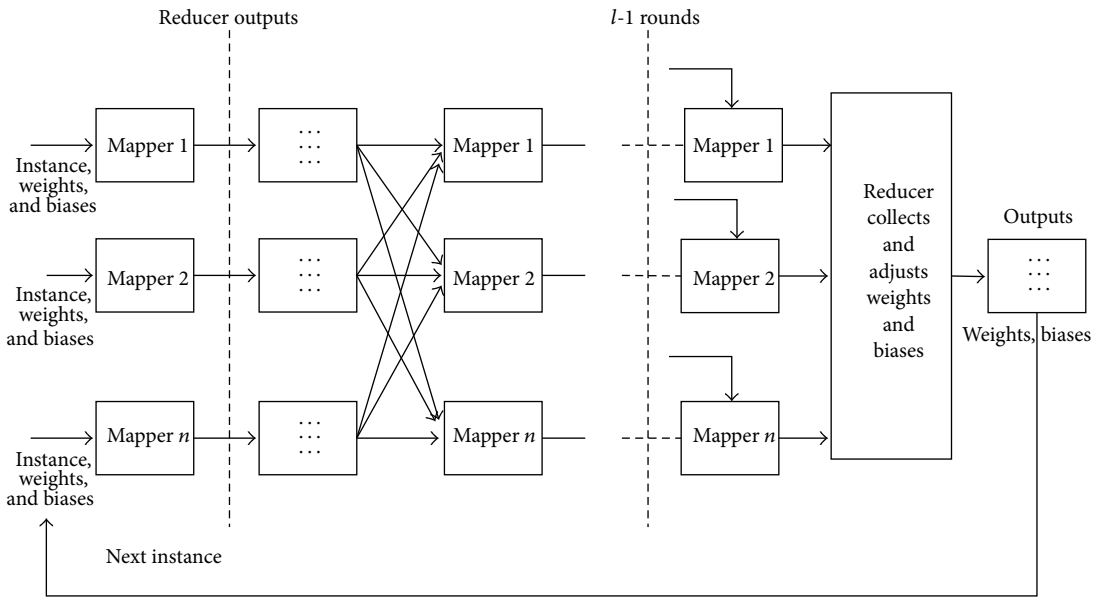
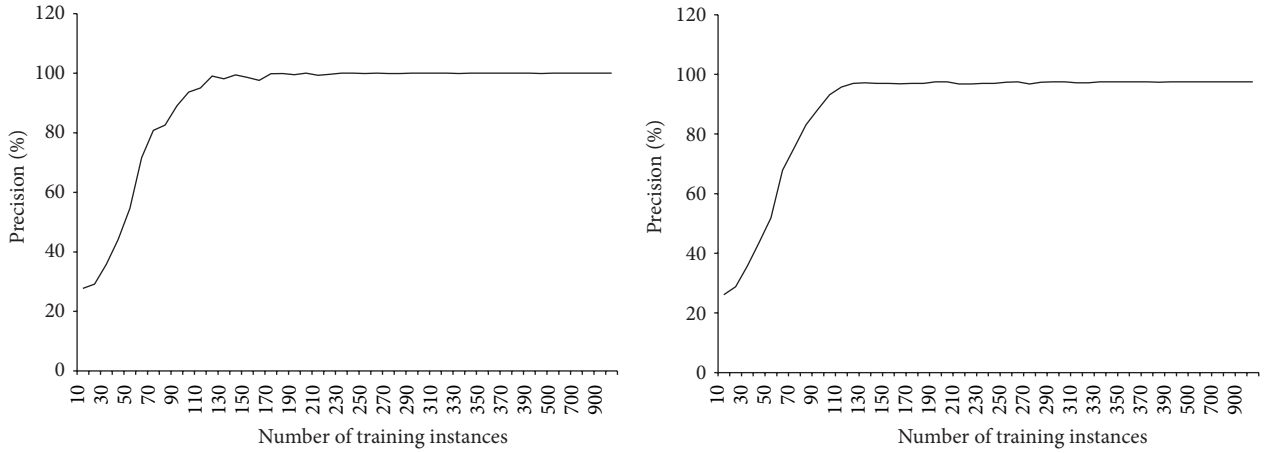


Figure 4: MRBPNN_3 structure.



(a) The synthetic dataset

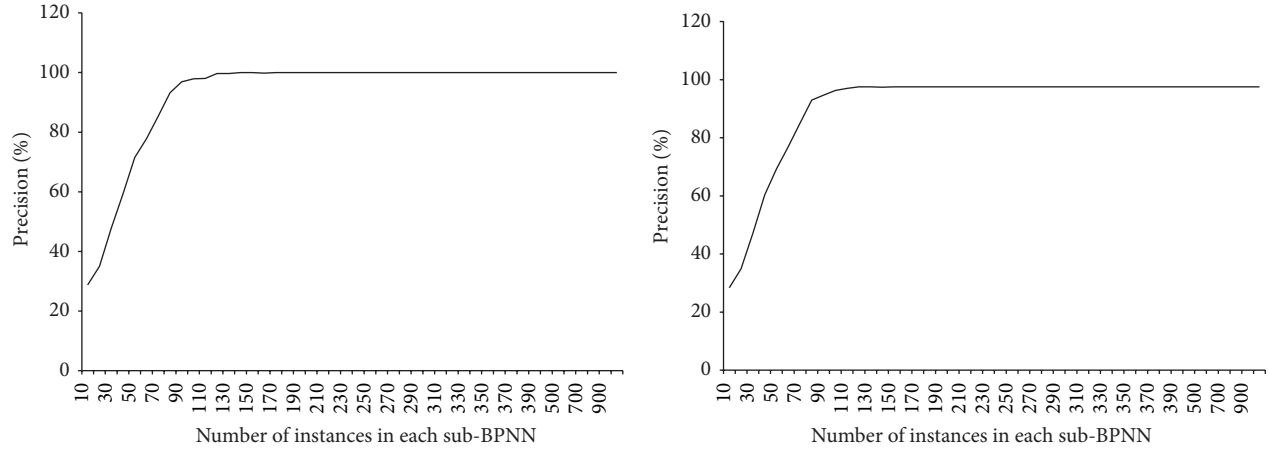
(b) The Iris dataset

Figure 5: The precision of MRBPNN_1 on the two datasets.

4.1. Classification Precision. The classification precision of MRBPNN_1 was evaluated using a varied number of training instances. The maximum number of the training instances was 1000 whilst the maximum number of the testing instances was also 1000. The large number of instances is based on the data duplication. Figure 5 shows the precision results of MRBPNN_1 in classification using 10 mappers. It can be observed that the precision keeps increasing with an increase in the number of training instances. Finally, the precision of MRBPNN_1 on the synthetic dataset reaches 100% while the precision on the Iris dataset reaches 97.5%. In this test, the behavior of the parallel MRBPNN_1 is quite similar to that of the standalone BPNN. The reason is that MRBPNN_1 does not distribute the BPNN among the Hadoop nodes; instead, it runs on Hadoop to distribute the data.

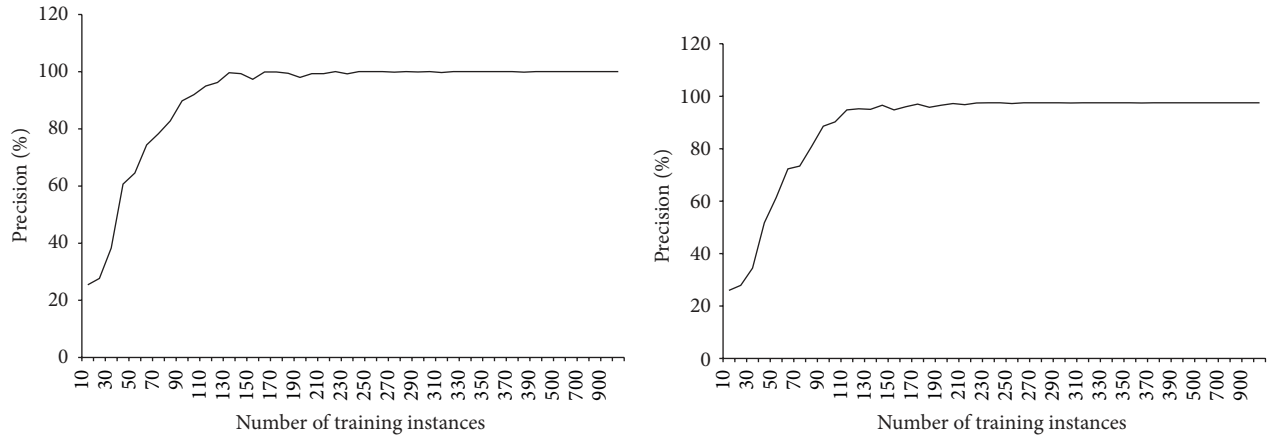
To evaluate MRBPNN_2, we designed 1000 training instances and 1000 testing instances using data duplication. The mappers were trained by subsets of the training instances and produced the classification results of 1000 testing instances based on bootstrapping and majority voting. MRBPNN_2 employed 10 mappers each of which inputs a number of training instances varying from 10 to 1000. Figure 6 presents the precision results of MRBPNN_2 on the two testing datasets. It also shows that, along with the increasing number of training instances in each subneural network, the achieved precision based on majority voting keeps increasing. The precision of MRBPNN_2 on the synthetic dataset reaches 100% whilst the precision on the Iris dataset reaches 97.5%, which is higher than that of MRBPNN_1.

MRBPNN_3 implements a fully parallel and distributed neural network using Hadoop to deal with a complex neural network with a large number of neurons. Figure 7 shows the performance of MRBPNN_3 using 16 mappers. The precision also increases along with the increasing number of training



(a) The synthetic dataset (b) The Iris dataset

Figure 6: The precision of MRBPNN_2 on the two datasets.



(a) The synthetic dataset (b) The Iris dataset

Figure 7: The precision of MRBPNN_3 on the two datasets.

instances for both datasets. It also can be observed that the stability of the curve is quite similar to that of MRBPNN_1. Both curves have more fluctuations than that of MRBPNN_2.

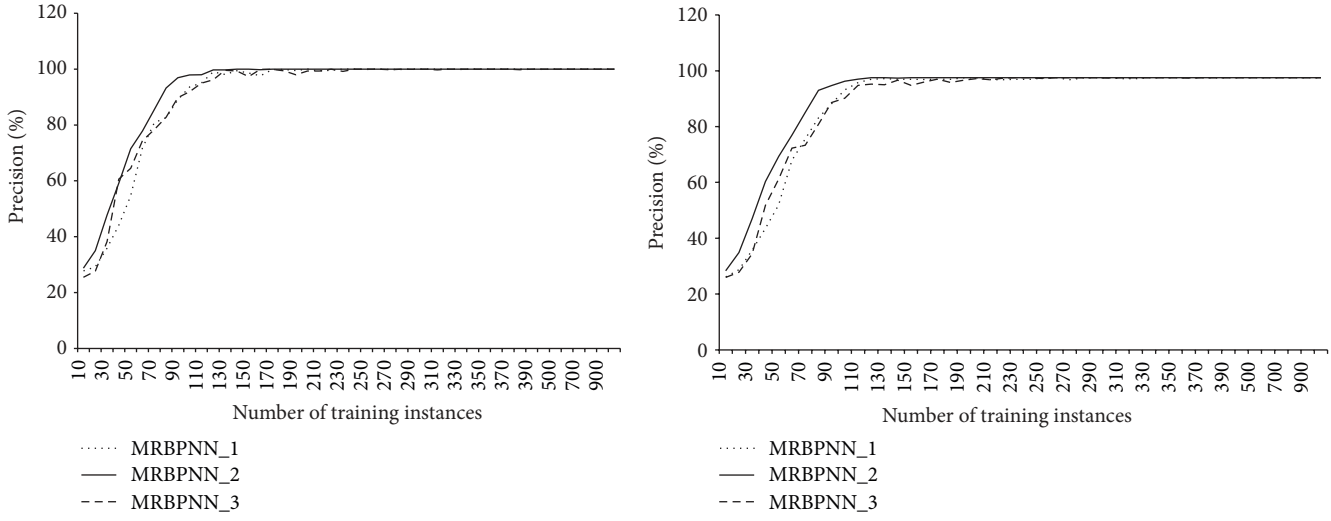
Figure 8 compares the overall precision of the three parallel BPNNs. MRBPNN_1 and MRBPNN_3 perform similarly, whereas MRBPNN_2 performs the best using bootstrapping and majority voting. In addition, the precision of MRBPNN_2 in classification is more stable than that of both MRBPNN_1 and MRBPNN_3.

Figure 9 presents the stability of the three algorithms on the synthetic dataset showing the precision of MRBPNN_2 in classification is highly stable compared with that of both MRBPNN_1 and MRBPNN_3.

4.2. Computation Efficiency. A number of experiments were carried out in terms of computation efficiency using the synthetic dataset. The first experiment was to evaluate the efficiency of MRBPNN_1 using 16 mappers. The volume of data instances was varied from 1 MB to 1 GB. Figure 10 clearly shows that the parallel MRBPNN_1 significantly outperforms the standalone BPNN. The computation overhead of the standalone BPNN is low when the data size is less than 16 MB. However, the overhead of the standalone BPNN increases sharply with increasing data sizes. This is mainly because MRBPNN_1 distributes the testing data into 4 data nodes in the Hadoop cluster, which runs in parallel in classification.

Figure 11 shows the computation efficiency of MRBPNN_2 using 16 mappers. It can be observed that when the data size is small, the standalone BPNN performs better. However, the computation overhead of the standalone BPNN increases rapidly when the data size is larger than 64 MB. Similar to MRBPNN_1, the parallel MRBPNN_2 scales with increasing data sizes using the Hadoop framework.

Figure 12 shows the computation overhead of MRBPNN_3 using 16 mappers. MRBPNN_3 incurs a higher overhead than both MRBPNN_1 and MRBPNN_2. The reason is that both MRBPNN_1 and MRBPNN_2 run training and classification within one MapReduce job, which means mappers and reducers only need to start once. However, MRBPNN_3 contains a number of jobs. The algorithm has to



(a) The synthetic dataset

(b) The Iris dataset

Figure 8: Precision comparison of the three parallel BPNNs.

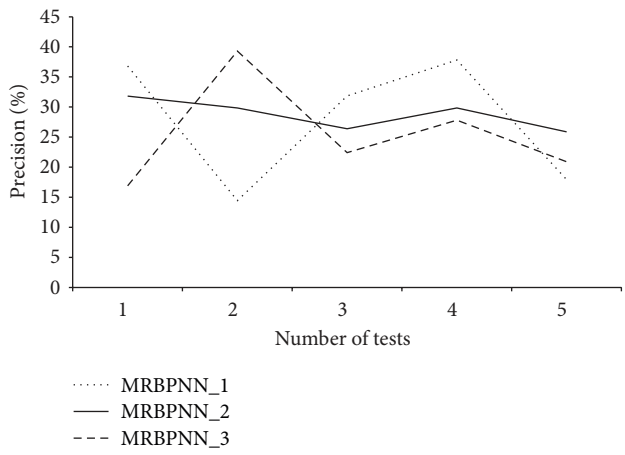


Figure 9: The stability of the three parallel BPNNs.

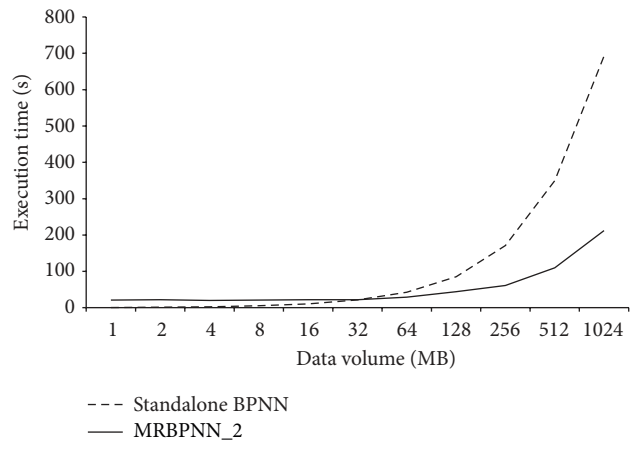


Figure 11: Computation efficiency of MRBPNN_2.

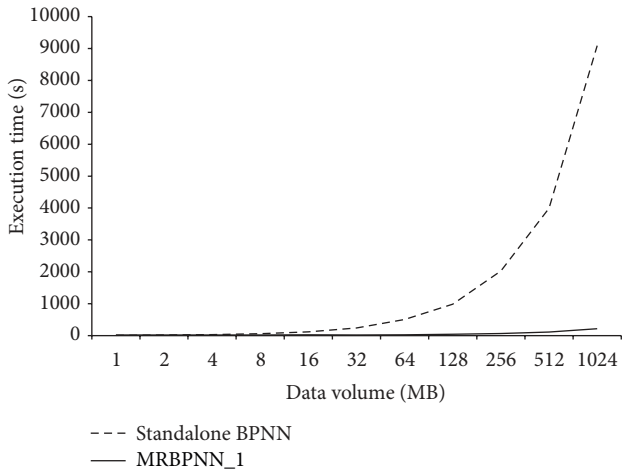


Figure 10: Computation efficiency of MRBPNN_1.

start mappers and reducers a number of times. This process incurs a large system overhead which affects its computation efficiency. Nevertheless, Figure 12 shows the feasibility of fully distributing a BPNN in dealing with a complex neural network with a large number of neurons.

5. Conclusion

In this paper, we have presented three parallel neural networks (MRBPNN_1, MRBPNN_2, and MRBPNN_3) based on the MapReduce computing model in dealing with data intensive scenarios in terms of the size of classification dataset, the size of the training dataset, and the number of neurons, respectively. Overall, experimental results have shown the computation overhead can be significantly reduced using a number of computers in parallel. MRBPNN_3 shows the feasibility of fully distributing a BPNN in a computer cluster but incurs a high overhead of computation due to continuous

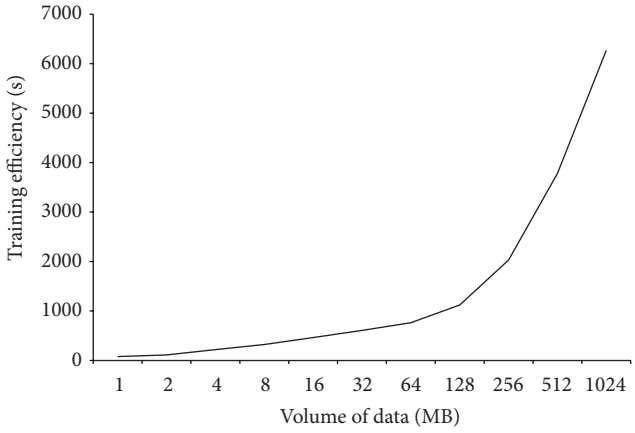


Figure 12: Computation efficiency of MRBPNN_3.

starting and stopping of mappers and reducers in Hadoop environment. One of the future works is to research in-memory processing to further enhance the computation efficiency of MapReduce in dealing with data intensive tasks with many iterations.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

The authors would like to appreciate the support from the National Science Foundation of China (no. 51437003).

References

[1] Networked European Software and Services Initiative (NESSI), "Big data, a new world of opportunities," Networked European Software and Services Initiative (NESSI) White Paper, 2012, http://www.nessi-europe.com/Files/Private/NESSI_WhitePaper_BigData.pdf.

[2] P. C. Zikopoulos, C. Eaton, D. deRoos, T. Deutsch, and G. Lapis, *Understanding Big Data, Analytics for Enterprise Class Hadoop and Streaming Data*, McGraw-Hill, 2012.

[3] M. H. Hagan, H. B. Demuth, and M. H. Beale, *Neural Network Design*, PWS Publishing, 1996.

[4] R. Gu, F. Shen, and Y. Huang, "A parallel computing platform for training large scale neural networks," in *Proceedings of the IEEE International Conference on Big Data*, pp. 376–384, October 2013.

[5] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing*, Benjamin Cummings/Addison Wesley, San Francisco, Calif, USA, 2002.

[6] Message Passing Interface, 2015, http://www.mcs.anl.gov/research/projects/mpi/.

[7] L. N. Long and A. Gupta, "Scalable massively parallel artificial neural networks," *Journal of Aerospace Computing, Information and Communication*, vol. 5, no. 1, pp. 3–15, 2008.

[8] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[9] Y. Liu, M. Li, M. Khan, and M. Qi, "A mapreduce based distributed LSI for scalable information retrieval," *Computing and Informatics*, vol. 33, no. 2, pp. 259–280, 2014.

[10] N. K. Alham, M. Li, Y. Liu, and M. Qi, "A MapReduce-based distributed SVM ensemble for scalable image classification and annotation," *Computers and Mathematics with Applications*, vol. 66, no. 10, pp. 1920–1934, 2013.

[11] J. Jiang, J. Zhang, G. Yang, D. Zhang, and L. Zhang, "Application of back propagation neural network in the classification of high resolution remote sensing image: take remote sensing image of Beijing for instance," in *Proceedings of the 18th International Conference on Geoinformatics*, pp. 1–6, June 2010.

[12] N. L. D. Khoa, K. Sakakibara, and I. Nishikawa, "Stock price forecasting using back propagation neural networks with time and profit based adjusted weight factors," in *Proceedings of the SICE-ICASE International Joint Conference*, pp. 5484–5488, IEEE, Busan, Republic of korea, October 2006.

[13] M. Rizwan, M. Jamil, and D. P. Kothari, "Generalized neural network approach for global solar energy estimation in India," *IEEE Transactions on Sustainable Energy*, vol. 3, no. 3, pp. 576–584, 2012.

[14] Y. Wang, B. Li, R. Luo, Y. Chen, N. Xu, and H. Yang, "Energy efficient neural networks for big data analytics," in *Proceedings of the 17th International Conference on Design, Automation and Test in Europe (DATE '14)*, March 2014.

[15] D. Nguyen and B. Widrow, "Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights," in *Proceedings of the International Joint Conference on Neural Networks*, vol. 3, pp. 21–26, June 1990.

[16] H. Kanan and M. Khanian, "Reduction of neural network training time using an adaptive fuzzy approach in real time applications," *International Journal of Information and Electronics Engineering*, vol. 2, no. 3, pp. 470–474, 2012.

[17] A. A. Ikram, S. Ibrahim, M. Sardaraz, M. Tahir, H. Bajwa, and C. Bach, "Neural network based cloud computing platform for bioinformatics," in *Proceedings of the 9th Annual Conference on Long Island Systems, Applications and Technology (LISAT '13)*, pp. 1–6, IEEE, Farmingdale, NY, USA, May 2013.

[18] V. Rao and S. Rao, "Application of artificial neural networks in capacity planning of cloud based IT infrastructure," in *Proceedings of the 1st IEEE International Conference on Cloud Computing for Emerging Markets (CCEM '12)*, pp. 38–41, October 2012.

[19] A. A. Huqqani, E. Schikuta, and E. Mann, "Parallelized neural networks as a service," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN '14)*, pp. 2282–2289, July 2014.

[20] J. Yuan and S. Yu, "Privacy preserving back-propagation neural network learning made practical with cloud computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 212–221, 2014.

[21] Z. Liu, H. Li, and G. Miao, "MapReduce-based back propagation neural network over large scale mobile data," in *Proceedings of the 6th International Conference on Natural Computation (ICNC '10)*, pp. 1726–1730, August 2010.

[22] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors," in *Proceedings of the 17th International Conference on Parallel Architectures*

and Compilation Techniques (PACT '08), pp. 260–269, October 2008.

[23] K. Taura, K. Kaneda, T. Endo, and A. Yonezawa, "Phoenix: a parallel programming model for accommodating dynamically joining/leaving resources," *ACM SIGPLAN Notices*, vol. 38, no. 10, pp. 216–229, 2003.

[24] Apache Hadoop, 2015, http://hadoop.apache.org.

[25] J. Venner, *Pro Hadoop*, Springer, New York, NY, USA, 2009.

[26] N. K. Alham, *Parallelizing support vector machines for scalable image annotation [Ph.D. thesis]*, Brunel University, Uxbridge, UK, 2011.

[27] The Iris Dataset, 2015, https://archive.ics.uci.edu/ml/datasets/Iris.