Int J Parallel Prog
DOI 10.1007/s10766-016-0401-1

The Parallelization of Back Propagation Neural Network in MapReduce and Spark

Yang Liu[1] · **Lixiong Xu**[1] · **Maozhen Li**[2,3]

Received: 22 December 2015 / Accepted: 11 February 2016

Abstract Artificial neural network is proved to be an effective algorithm for dealing with recognition, regression and classification tasks. At present a number of neural network implementations have been developed, for example Hamming network, Grossberg network, Hopfield network and so on. Among these implementations, back propagation neural network (BPNN) has become the most popular one due to its sensational function approximation and generalization abilities. However, in the current big data researches, BPNN, as a both data intensive and computational intensive algorithm, its efficiency has been significantly impacted. Therefore, this paper presents a parallel BPNN algorithm based on data separation in three distributed computing environments including Hadoop, HaLoop and Spark. Moreover to improve the algorithm performance in terms of accuracy, ensemble techniques have been employed. The algorithm is firstly evaluated in a small-scale cluster. And then it is further evaluated in a commercial cloud computing environment. The experimental results indicate that the proposed algorithm can improve the efficiency of BPNN with guaranteeing its accuracy.

Keywords Neural network · MapReduce · Hadoop · HaLoop · Spark · Ensemble technique

✉ Lixiong Xu
xulixiong@scu.edu.cn

1 School of Electrical Engineering and Information Systems, Sichuan University, Chengdu 610065, China

2 Department of Electronic and Computer Engineering, Brunel University London, Uxbridge UB8 3PH, UK

3 The Key Laboratory of Embedded Systems and Service Computing, Tongji University, Shanghai 200092, China

Published online: 24 February 2016

1 Introduction

At present, data explosion leads to significant developments of big data researches. Great numbers of industrial and academic organizations for example e-business, World Wide Web, sensor networks and social networks keep generating data, and moreover trying to understand and extract knowledge from large volumes of data. It motivates big data researches to capture, process, analyse and visualize potentially large datasets in reasonable time [1].

As one of the most popular data processing algorithm, artificial neural networks (ANNs) have been widely used in pattern recognition and classification applications. It has excellent potential to approximate non-linear functions by arbitrary precisions based on its neurons and function combination inside the neurons. Among numbers of implementations of ANNs, back propagation neural network (BPNN) is a widely used one because of its magnificent function approximation and generalization abilities. However, BPNN has to face a critical issue in dealing with big data: it performs in a low efficiency due to large number of iterations and parameters computations. To speed up the performance of BPNN, its computation could be distributed using parallel computing techniques such as Message Passing Interface (MPI) [2]. In research [3], Long et al. presented a scalable parallel ANN using MPI for parallelization. It is worth noting that MPI was designed for data intensive applications with high performance requirements. However, MPI provides little support in fault tolerance. If fault occurs, the computation in MPI has to be restarted. As a result, a large-scale task in MPI may cost extremely long time for processing [4].

Therefore, this paper presents a parallelized back propagation neural network algorithm (PBPNN) using MapReduce and Spark. At present, MapReduce has become a de facto standard computing model in support of big data applications [5,6]. It provides a reliable, fault-tolerant, scalable and resilient computing framework for storing and processing large 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 node. There are two main functions Map and Reduce in the MapReduce computing model. Basically Map is responsible for actual data processing and it outputs intermediate outputs in the form of $<$*key-value*$>$ pairs. Reduce collects the outputs of Maps with secondary processing including sorting and merging based on the values of keys. And then it outputs the ultimate results into distributed file system. Basically our algorithm creates one BPNN in each Map. And then the training data is segmented into data chunks processed by Maps in parallel. In this case, each BPNN in each Map is only trained by a portion of training data, which results in accuracy loss of training. Therefore to maintain a high accuracy during classification, ensemble techniques have been employed. And then the Reduce function executes the ensemble operations and finally outputs the classification results.

Although MapReduce is a well-designed distributed computing model, several researches [7–10] pointed out that its implementation for example Hadoop [11] is not quite suitable for iterative jobs [12] that BPNN is belonged to. To overcome the issue, some other distributed computing implementations such as HaLoop and Spark have been developed. HaLoop [7,13] is based on Hadoop however it has a number of improvements especially aiming at serving the iteration based jobs. Its Map and

Reduce functions work similar to those of Hadoop do, but the additional iteration-orientated interfaces help to deal with the iterative processes. Spark [14] inherits the advantages of MapReduce and implements a number of optimizations such as in memory computation, IO improvements, RDD data structure and so on. In our algorithm implementation, its Map and FlatMap functions help to execute data separation whilst its Join and Reduce functions help to execute ensemble operations. In order to observe the algorithm performances, this paper implements the presented algorithm in Hadoop, HaLoop and Spark respectively.

The rest of the paper is organized as: Sect. 2 reviews the related work; Sect. 3 presents the technical details of BPNN and three types of MapReduce implementations; Sect. 5 presents the design of parallel BPNN and its implementation in Hadoop, HaLoop and Spark; Sect. 6 shows the experimental results and Sect. 7 concludes the paper.

2 Related Work

Artificial neural network has been successfully used in pattern recognition, image recognition, regression, and classification tasks. Among types of ANN algorithms, BPNN has become the most popular one due to its remarkable function approximation. Jiang et al. [15] employed BPNN to process the high resolution remote sensing image classification task. They claimed that BPNN improves the classification precision in recognition of roads and roofs in the images. Khoa et al. [16] proposed a method to forecast the stock price using BPNN. Based on their experimental results the algorithm achieved accuracy improvement in forecasted results.

Initially, BPNN is designed to process small volume of data which only contains a less number of instances. However, at present the analysis for large volume of data becomes quite common. Rizwan et al. [17] carried out a research on global solar energy estimation. They innovatively modeled a high accurate model for avoiding extremely simple parameterizations in traditional ways. The authors admitted that their topic is a big task, which results in low efficiency in BPNN training phase. Wang et al. [18] also pointed out that large-scale neural networks have already become mainstream tools for big data analytics. However, they also figured out that the training phase in a large-scale neural network may generate extreme large overhead. A number of researchers [19] tried to improve the selection of the initial weights or controlling the learning parameters [20] in terms of speeding up BPNN in processing big data. However, more researchers focus on employing distributed computing and cloud computing technologies to improve BPNN's efficiency. Hasan and Taha [21] built up a distributed hardware environment with multi-core system to speed up the feed forward neural network for modeling routing bandwidth model. Huqqani et al. [22] established both CUDA and OpenMP environments in N2Sky which is a neural network simulation environment for the parallelization. Their parallelization is based on the separation of structural and topological data, but without considering accuracy loss due to data separation. Yuan and Yu [23] employed BPNN for processing ciphered text classification tasks in cloud computing environment. However, they only focused on how to facilitate data processing but without considering the algorithm efficiency.

The works [24,25] also implemented BPNN in cloud computing. However, one thing needs to be pointed out that the cloud computing is highly loosely coupled therefore the algorithm efficiency could be impacted. As a result, researchers started speeding up BPNN using distributed computing or parallel computing techniques. Gu et al. [26] have built up a parallel computing platform aiming at training large-scale neural networks. They used in-memory computations and event-driven messaging communications. However, they just simply distributed the training data without discussing the loss of accuracy issue. And also they haven't discussed the scalability and fault tolerance of their platform. Liu et al. [27] proposed a MapReduce based distributed BPNN for processing large-scale mobile data. They also divided the training dataset and employed adaboosting to improve accuracy. However, although adaboosting is a popular sampling technique, it may enlarge the weight of wrongly classified instances which would deteriorate the algorithm accuracy.

In order to distribute BPNN, this paper employs distributed computing and ensemble techniques to parallelize the algorithm with guaranteeing its accuracy. Three popular distributed computing frameworks, Hadoop, HaLoop and Spark are employed in terms of comparing the algorithm performances.

3 Technical Review

3.1 Back Propagation Neural Network

BPNN is a multi-layer feed forward network which is trained by the training data while it tunes the network parameters using an error back propagation mechanism. BPNN can perform a large volume of input-output mappings without knowing their exact mathematical equations. Benefiting from the gradient-descent feature, BPNN keeps tuning the parameters of the network during the error propagation until its parameters adapt to all input instances. The structure of a typical BPNN is shown in Fig. 1.

One BPNN is consisted by multiple network layers. However it's 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 [28]. Therefore, the topology of a BPNN usually contains three layers: the input layer, one hidden

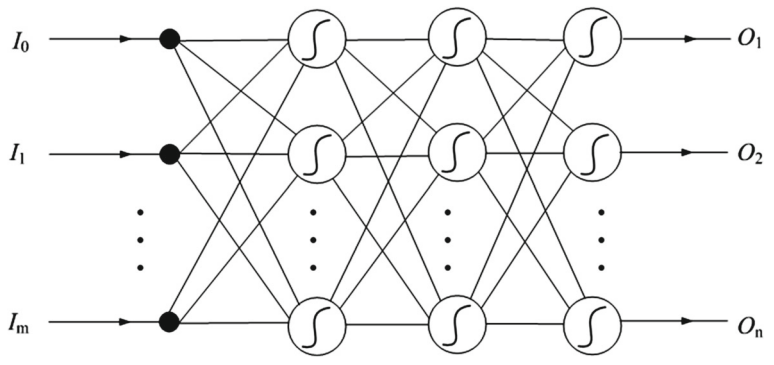

Fig. 1 The structure of a typical BPNN

layer and the 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 $\boldsymbol{s}$ denote one input instance, n denote the number of inputs in the input layer:

$$\boldsymbol{s} = \{a_1, a_2, a_3, \ldots, a_n\}$$

Each input of a neuron is companied by one parameter named as 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 the last neuron, the input I_j of the next neuron can be calculated using Eq. (1).

$$I_j = \sum\nolimits_i w_{ij} o_j + \theta_j \tag{1}$$

The output of a neuron is usually computed by the sigmoid function, so the output o_j of a neuron can be calculated using Eq. (2).

$$o_j = \frac{1}{1 + e^{-I_j}} \tag{2}$$

As long as the feed forward process completes, the back propagation process starts.

Let Err_j represent the error-sensitivity, and t_j represent the desirable output of a neuron j in the output layer, thus:

$$Err_j = o_j \left(1 - o_j\right) \left(t_j - o_j\right) \tag{3}$$

Let Err_k represent the error-sensitivity of a neuron in the last layer, and w_{kj} represent its weight, thus the Err_j of the neuron in the next layer can be calculated using Eq. (4).

$$Err_j = o_j \left(1 - o_j\right) \sum\nolimits_k Err_k w_{kj} \tag{4}$$

Therefore the weights and biases of each neuron can be tuned in the back propagation process using Eqs. (5–8), where η represents the learning rate.

$$\Delta w_{ij} = Err_j o_j \tag{5}$$
$$w_{ij} = w_{ij} + \eta \Delta w_{ij} \tag{6}$$
$$\Delta \theta_j = Err_j \tag{7}$$
$$\theta_j = \theta_j + \eta \Delta \theta_j \tag{8}$$

After one input instance finishes tuning the network, BPNN starts inputting the next instance to train the network parameters. Until Eq. (9) is satisfied for a single output or Eq. (10) is satisfied for multiple outputs, BPNN terminates it training phase.

$$min\left(E\left[e^2\right]\right) = min\left(E\left[(t-o)^2\right]\right) \tag{9}$$

$$min\left(E\left[e^T e\right]\right) = min\left(E\left[(t-o)^T (t-o)\right]\right) \tag{10}$$

For the classification, BPNN only needs to execute the feed forward. The outputs in the output layer indicate the classification result.

4 MapReduce and Spark

MapReduce targets on serving data intensive tasks in a distributed way. Programmatically inspired from functional programming, at its core there are two primary features, namely Map function and Reduce function. From a logical perspective, all data is treated as a Key (K)-Value (V) pair. At an atomic level however a Map takes a {K1, V1} pair and emits an intermediate list {K2, V2} pairs. Reduces take all values represented by the same key in the intermediate list generated by Maps and process them accordingly, emitting a final new list {V2}. All Maps and Reduces run independently in parallel.

4.1 Hadoop

Hadoop is a MapReduce based distributed computing framework. It supplies remarkable scalability, which can scale up the cluster from one machine to thousands of machines. Within one Hadoop cluster, individual computers contribute their resources including processors, hard disks, memory and network adapters to form the Hadoop Distributed File System (HDFS), in which the computers are logically categorized into one Namenode and a number of Datanodes. The Namenode manages the metadata of the cluster whilst the datanodes are the actual data processing nodes. The Map function (mapper) and Reduce function (reducer) run in the Datanodes. In order to manage the resources and scheduling tasks efficiently, Hadoop employs Yarn [11] with four main components including Container, ResourceManager, NodeManager and ApplicationMaster to manage the job executions.

Briefly, when a job is submitted to a Hadoop cluster, each mapper inputs one data chunk and starts processing. Each reducer collects the intermediate output of mappers based on the key distribution of the mapper outputs, after merging, shuffling and sorting it outputs the ultimate results into HDFS. However, Hadoop doesn't offer any optimization for iterations which BPNN algorithm uses a lot. It has to employ a number of MapReduce jobs to implement iterations. Figure 2 shows how Hadoop processes iterations.

4.2 HaLoop

HaLoop is also a MapReduce model based distributed computing framework proposed by Bu et al. [7]. It reuses most of the source code of Hadoop, but a number of improvements have been added which enables better iterative job adaption: 1. HaLoop supplies new programming interfaces and API implementations including iterative programming model, loop body, iteration termination and so on, which facilitate the expression

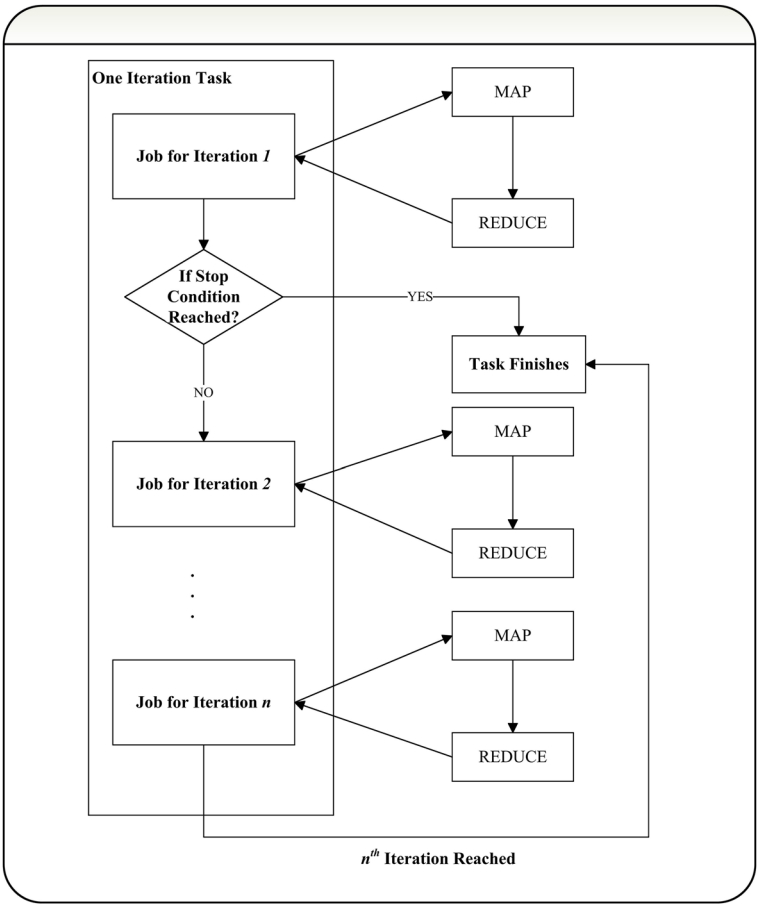

Fig. 2 Hadoop in dealing with iterative job

of iterations in MapReduce. 2. The master node in HaLoop contains a new loop control module that repeatedly starts new MapReduce steps that compose the loop body, until a user-specified stopping condition is met. 3. A specially designed task scheduler is embedded in HaLoop in terms of facilitating iterative job processing based on data locality. 4. HaLoop caches and indexes application data in slave nodes. Generally HaLoop inherits most of Hadoop characteristics but it significantly modifies and improves the functions of Hadoop. Therefore HaLoop is not only responsible for task control but also for loop control, caching, indexing and local file system access. Figure 3 shows how HaLoop processes iterations.

4.3 Spark

Spark is a newly developed parallel computing framework, which improves big data processing efficiency whilst maintains high fault tolerance and scalability. Spark

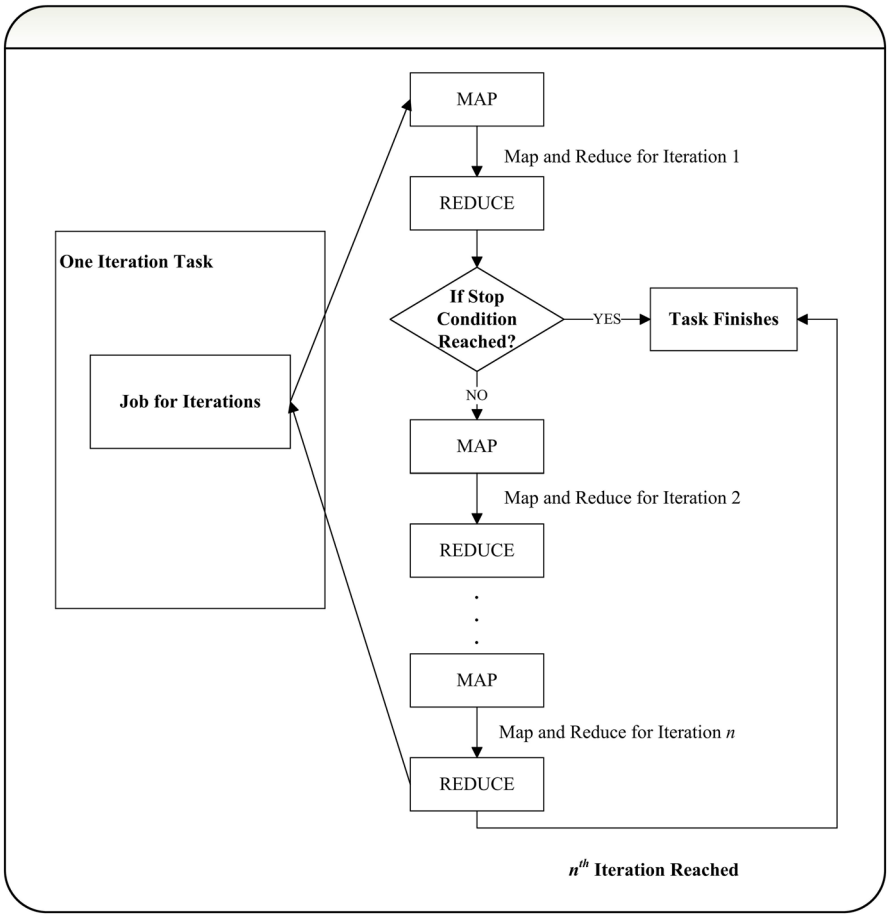

Fig. 3 HaLoop in dealing with iterative job

employs master-slave model to build its overall architecture. Master node is a cluster commander which responses for the cluster running. Slave is the actual computing node, which receives the commands from the master node and reports its own states back. One remarkable feature distinguishing Spark and other MapReduce implementations are the RDD (Resilient Distributed Dataset) data structure. It can create and control data partitions on a number of nodes in the cluster. Spark works as: client initially submits an application. Master finds an available worker and launches an application driver. The driver applies computing resources from the master, and then it converses the application to RDD graph that will be finally conversed to stage. TaskScheduler receives the stage, and ultimately delivers it to the executor. Figure 4 indicates the data processing of Spark.

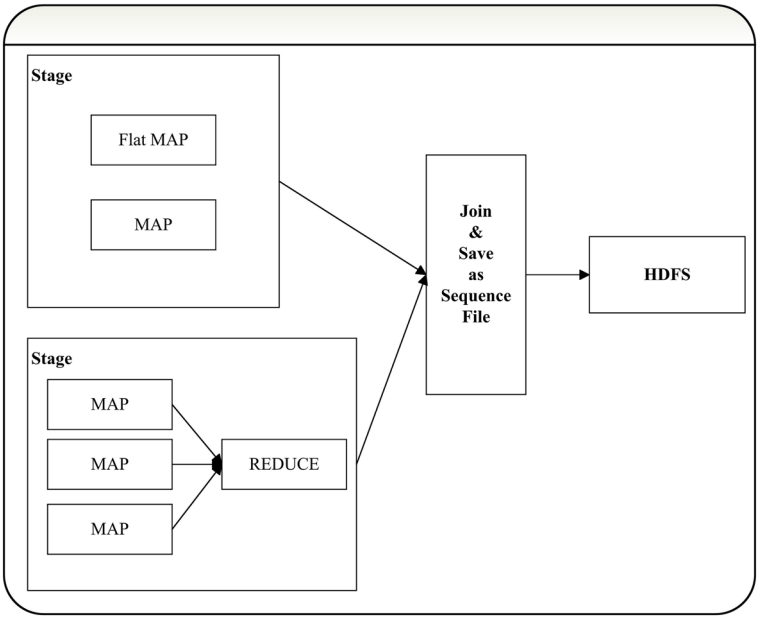

Fig. 4 Spark in processing

5 Parallelizing BPNN Using Hadoop, HaLoop and Spark

In Hadoop, HaLoop and Spark, they supply the similar processing functions. For example, Hadoop and HaLoop supply mappers, while Spark supplies FlatMap and Map. Hadoop and HaLoop supply reducer, while Spark supplies Join and reduceByKey. This point facilitates the design of the algorithm. In the following presentation, the operations in Map phase represent the implementations in mapper of Hadoop and HaLoop, and also the implementation in FlatMap and Map in Spark. Similarly, the operations in Reduce phase represent the implementations in reducer of Hadoop and HaLoop, and also the Join and reduceByKey in Spark.

Let S denote the entire training dataset, n denote the number of Maps in a distributed environment. Therefore, PBPNN divides S into n data chunks of which each data chunk s_i is input into one Map for training respectively.

$$S = \bigcup_1^n s_i, \{\forall s \in s_i | s \notin s_n, i \neq n\} \tag{11}$$

In each Map map_i, it individually initializes one neural network $BPNN_i$ with its own topology and initial values of parameters. Each data chunk s_i is used as the input training data for $BPNN_i$. The weights and biases of $BPNN_i$ are iteratively tuned by each instance in s. Finally each Map becomes one classifier:

$$(Map_i, BPNN_i, s_i) \rightarrow classifier_i$$

The testing dataset is also separated into n parts. Each classifier inputs one and generates the final classification result for each testing instance. Theoretically the neural network algorithm is speeded up.

The data separation effectively reduces the processing overhead of BPNN algorithm. However, each $classifier_i$ is only trained by a data portion of the original dataset S. In this case, the classification accuracy of one Map could be significantly degraded. To solve the issue, PBPNN employs ensemble techniques including bootstrapping and majority voting to maintain the classification accuracy by combining weak classifiers into one strong classifier.

5.1 Bootstrapping

Balanced bootstrapping is a variance reduction technique for efficient bootstrap simulation. The method is based on the idea of controlling the number of times that the training instances appear in the bootstrap samples, so that in the B bootstrap samples, each instance appears the same number of times. For the bootstrapping to work, some instances must be missing in certain bootstrap samples, while others may appear two or more times. The sampling dose not force each bootstrapping sample to contain all the training instances; the first instance may appear twice in the first bootstrapping sample and not appear at all in the second bootstrapping sample, while the second instance may appear once in each sample. The most efficient way of creating balanced bootstrap samples is to construct a string of the instances $X_1, X_2, X_3, \ldots, X_n$ repeating B times so that a sequence $Y_1, Y_2, Y_3, \ldots, Y_{Bn}$ is achieved. A random permutation p of the integers from 1 to B_n is taken. Therefore the first bootstrapping sample can be created from $Y_p(1), Y_p(2), Y_p(3), \ldots, Y_p(n)$, moreover the second bootstrapping sample from $Y_p(n+1), Y_p(n+2), Y_p(n+3), \ldots, Y_p(2n)$ and so on, 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 employed in bagging to increase the classification accuracy [29].

5.2 Majority Voting

The majority voting is a commonly used combination technique. The ensemble classifier predicts a class for a test instance which is predicted by the majority of the base classifiers [29]. 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 $P_{i,j} = 0$. The ensemble predict for class k if:

$$P_{i,k} = max_{j=1}^{c} \sum_{i=1}^{I} P_{i,j} \tag{12}$$

5.3 Algorithm Design

The algorithm firstly generates subsets from the entire training dataset using the balanced bootstrapping:

$$banlanced\ bootstrapping \rightarrow \{S_1, S_2, S_3, \ldots, S_n\}, \bigcup_{i=1}^{n} S_i = S$$

- S_i represents the ith subset of the entire dataset S.
- n represents the total number of subsets.

Every S_i is saved in one individual file which locates in the algorithm input path in HDFS. For facilitating the algorithm IO, each file does not only contain the training instances but also the testing instances, thus the trained $BPNN_i$ in Map_i can directly start classification without any extra operation. Every instance $s_k = \{a_1, a_2, a_3, \ldots, a_{in}\}, s_k \in S_i$ is defined by the data format:

$$< instance_k, target_k, type >$$

- $instance_k$ represents one bootstrapped instance s_k, which is the input of neural network.
- in represents the number of input in the input layer.
- $target_k$ represents the desirable output if $instance_k$ is a training instance.
- $type$ field has two values: *"train"* and *"test"*, which distinguish the type of $instance_k$.

When the algorithm starts working, a number of n Maps input the number of n files in parallel. Each Map constructs one BPNN and initializes weights and biases with random values between -1 and 1 for its neurons. And then a Map inputs one instance in the form of $<instance_k, target_k, type>$ from the input file.

The Map firstly parses the data and retrieves the type of the instance. If the type value is *"train"*, the instance is input into the input layer. Secondly, each neuron in different layers computes its output using Eqs. (1) and (2) until the output layer outputs. Feed forward process terminates. Thirdly the Map starts back propagation process. It computes and updates weights and biases for neurons using Eqs. (3) to (8). Back propagation process terminates. Until all instances marked as *"train"* are processed and error is satisfied, the training of the neural network finishes.

However, if the type of one instance is *"test"*, the Map inputs the instance and starts feed forward to classify it using the trained neural network. In this case, each neural network in one Map outputs the classification result of one instance at the output layer. The Map generates the intermediate output for each instance in the form of:

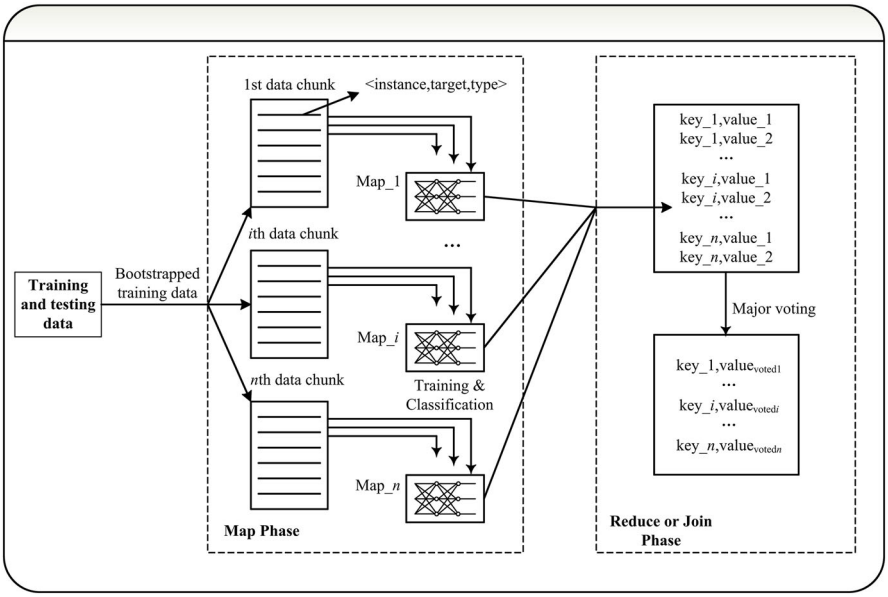

Fig. 5 PBPNN algorithm structure

$$< instance_k, o_{jm} >$$

- $instance_k$ represents the key. Therefore one reducer can collect all o_{jm} of the same $instance_k$ and further executes the majority voting.
- o_{jm} represents the outputs of the mth Map, $m \in n$.

As long as Maps finish their tasks, one Reduce (including reducer, join and reduceByKey) starts collecting the outputs of all Maps. The outputs with the same key are merged together. As Hadoop, HaLoop and Spark can create a number of collections in each of which contains values with the same key. Therefore it enables the majority voting for $instance_k$ using all o_{jm} in Reduce using Eq. (12). And then it outputs the result of $instance_k$into HDFS:

$$< instance_k, r_k >$$

- r_k represents the voted classification result of $instance_k$.

Figure 5 indicates the algorithm architecture. Algorithm 1 shows the pseudo code of PBPNN.

Algorithm1: PBPNN

Input: *S, T* (dataset to be classified) Output: *C*

n Maps one Reduce or Join

1. Each Map constructs one BPNN with *in* inputs, *o* outputs, *h* neurons in hidden layer
2. Initialize $w_{ij} = random_{1ij} \in (-1, 1), \theta_j = random_{2j} \in (-1, 1)$
3. Bootstrap $\{S_1, S_2, S_3, \dots, S_n\}, \bigcup_{i=1}^{n} S_i = S$
4. Each Map inputs $s_k = \{a_1, a_2, a_3, \dots, a_{in}\}, s_k \in S_i$

   Input $a_i \rightarrow in_i, neuron_j$ in hidden layer computes

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

   $o_{jh} = 1/(1 + e^{I_{jh}})$
5. Input $o_j \rightarrow out_i, neuron_j$ in output layer computes

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

   $o_{jo} = 1/(1 + e^{I_{jo}})$
6. In each output, compute

   $Err_{jo} = o_{jo}(1 - o_{jo})(target_j - o_{jo})$
7. In hidden layer, compute

   $Err_{jh} = o_{jh}(1 - o_{jh}) \sum_{i=1}^{o} Err_i w_{io}$
8. Update

   $w_{ij} = w_{ij} + \eta \cdot Err_j \cdot o_j$

   $\theta_j = \theta_j + \eta \cdot Err_j$

Repeat 3, 4, 5, 6, 7, 8

Until

$\min(\mathrm{E}[e^2]) = \min(E[(target_j - o_{jo})^2])$

Training terminates

9. Each Map inputs $t_i = \{a_1, a_2, a_3, \dots, a_{in}\}, t_i \in T$
10. Execute 4, 5
11. Map outputs $< t_j, o_j >$
12. Reduce or Join collects $< t_j, o_{jm} > m = (1, 2, \dots, n)$
13. For each t_j

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

Repeat 9, 10, 11, 12, 13

Until *T* is traversed.

14. Reduce or Join outputs *C*

Algorithm terminates.

6 Performance Evaluation

6.1 Evaluations in Small Scale Cluster

The PBPNN algorithm is firstly evaluated in a small-scale cluster. The cluster is consisted of 5 computers of which four nodes are Datanodes and the rest one is Namenode. The cluster details are listed in Table 1.

Table 1 Cluster details

Namenode	CPU: Core i7@3GHz
	Memory: 8GB
	SSD: 750GB
	OS: Ubuntu
Datanodes	CPU: Core i7@3.8GHz
	Memory: 32GB
	SSD: 250GB
	OS: Ubuntu
Network bandwidth	1Gbps
Deployed Software	Hadoop, HaLoop and Spark

Table 2 Dataset details

Type	Dataset characteristics	Instance number	Attribute number	Class number
Iris	Multivariate	150	4	3

The dataset employed in the evaluations is Iris dataset [30], which is a published machine learning benchmark dataset. Table 2 shows the details of the dataset.

The following tests contain two types of experiments. The first one is aiming at the algorithm precision and the second one is aiming at the efficiency. The cluster physically supplies 16 Maps whilst the value of bootstrapped times is 4. The neural network employed in each Map has three layers with 16 neurons in hidden layer. The three classes are encoded as {0,0}, {0,1}, {1,0}, therefore the output number in output layer is 2. Each instance is normalized using:

For one instance $s_i = \{a_1, a_2, a_3, \ldots, a_{in}\}$,

- a_{max} denotes the maximum element in s_i.
- a_{min} denotes the minimum element in s_i.
- na_i denotes the normalized a_i in s_i.

Then

$$na_i = \left(a_i - a_{min)}\right) \Big/ \left(a_{max} - a_{min)}\right) \tag{13}$$

The precision p can be calculated using:

$$p = \frac{r}{r + w} \times 100\,\% \tag{14}$$

- r represents the number of correctly classified instances.
- w represents the number of wrongly classified instances.

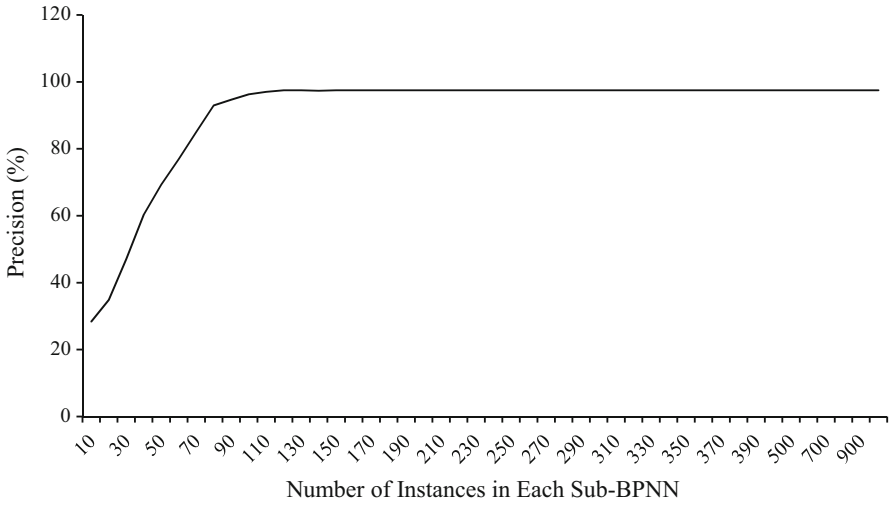

Fig. 6 The precision of iris dataset

6.1.1 Precision

This section focuses on the accuracy of the algorithm. The following tests were carried out using Hadoop. Due to the algorithm design, the platform doesn't affect algorithm precision.

In this test, 1000 training instances and 1000 testing instances based on data duplication are generated. There are ten mappers employed, each of which inputs the number of training instances from 10 to 1000. The figure shows that with the increasing number of training instances in each sub-BPNN, the precision based on bootstrapping and majority voting keeps increasing (Fig. 6).

For indicating the improvement in terms of algorithm precision, we also implement a standalone BPNN. The comparison is shown in Fig. 7. The figure indicates that when the number of training instances is small, PBPNN outperforms the Standalone BPNN, which proves the combination of the weak classifiers can contribute accuracy result.

Figure 8 evaluates the stability of the algorithm in dealing with a less number of training instances. The experiments of the standalone BPNN and the presented PBPNN with training 10 instances are carried out five times for each. The results show that PBPNN performs more stably than standalone BPNN does.

6.1.2 Efficiency

Figure 9 shows the efficiencies of PBPNN on Hadoop, HaLoop and Spark with processing an increasing volume of data. The data is consisted by 80 % training instances and 20 % testing instances. The efficiency of the standalone BPNN is also recorded in terms of comparison. It can be observed that Spark performs the most efficiently due to its in-memory computation. HaLoop slightly outperforms Hadoop. The reason is due to the algorithm design, the iterations is only processed within Maps but not by a

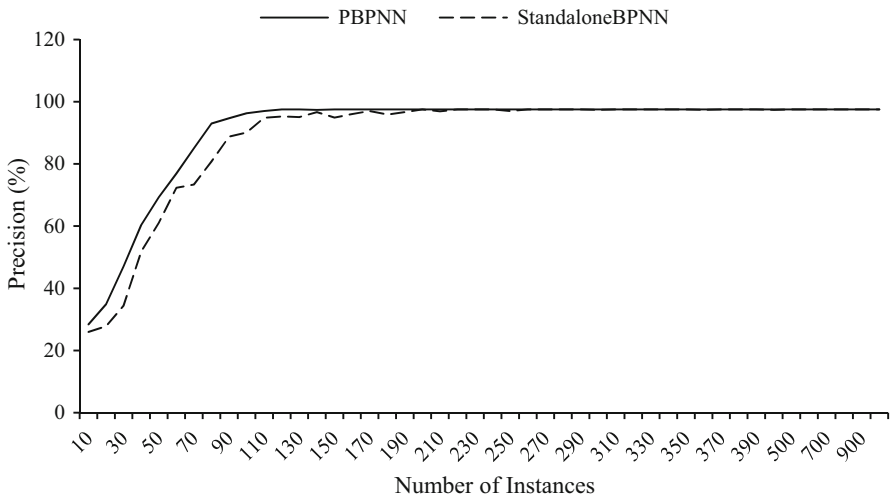

Fig. 7 Comparisons of PBPNN and Standalone NN

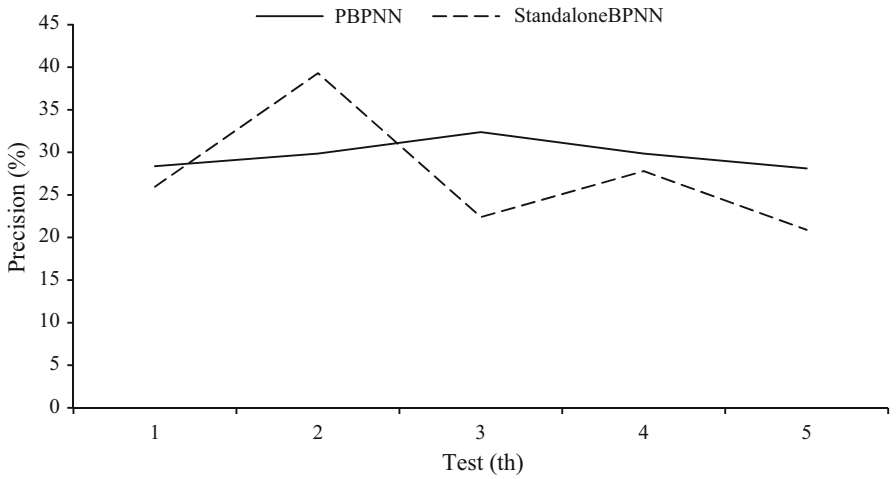

Fig. 8 Precision stability

series of MapReduce jobs. In this case HaLoop cannot fully contribute its potential. However, due to the cache and Java virtual machine reuse, HaLoop still works a little more efficiently than Hadoop does.

Figure 10 shows the algorithm efficiency with an increasing number of Maps in dealing with 1GB data. The results show that although the algorithm performance is improved with the increasing number of Maps, Spark still outperforms Hadoop and HaLoop.

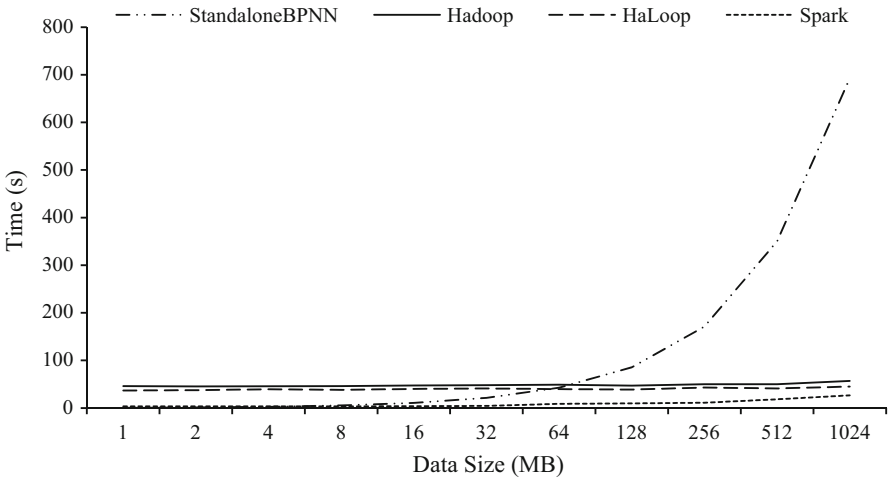

Fig. 9 Comparisons of efficiency with increasing data sizes

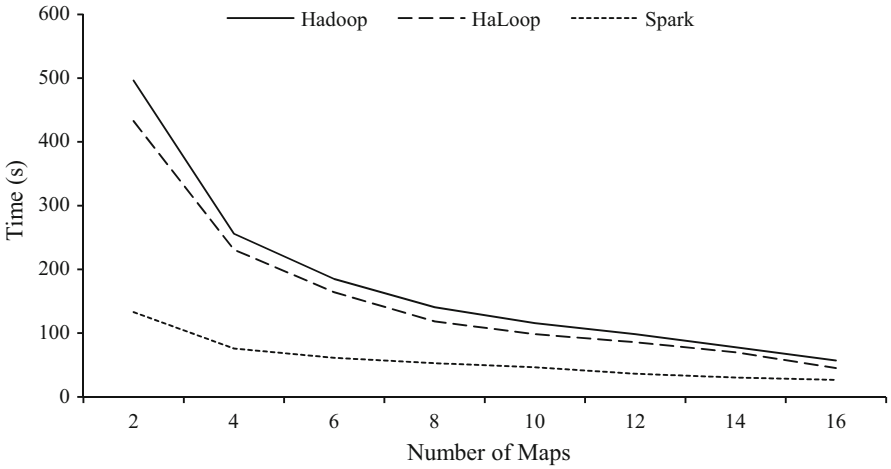

Fig. 10 Comparisons of efficiency with increasing Maps

6.2 Evaluations in Aliyun Cloud

This section focuses on evaluating algorithm efficiency in a commercial cloud computing environment Aliyun [31]. The parameters of the cloud are listed in the following table (Table 3).

In the first experiment, we fixed 16 Maps with increasing volumes of data. Figure 11 indicates that among three platforms, Spark still performs the most efficiently. When the data size is small, Hadoop and HaLoop give similar performances. However, when the data size keeps increasing, HaLoop outperforms Hadoop.

Figure 12 shows the algorithm efficiency with increasing number of Maps in dealing with 1GB data. The result shows that although all platforms give better performances

Table 3 Cloud details

Core number	Physical Core: 16 Threading: 32
CPU frequency	2.6GHz
Memory	16GB
Hard disk	20GB
OS	CentOS 6.3 64bit
Deployed software	Hadoop, HaLoop and Spark
Bandwidth	1Gbps
Location	Shenzhen, China

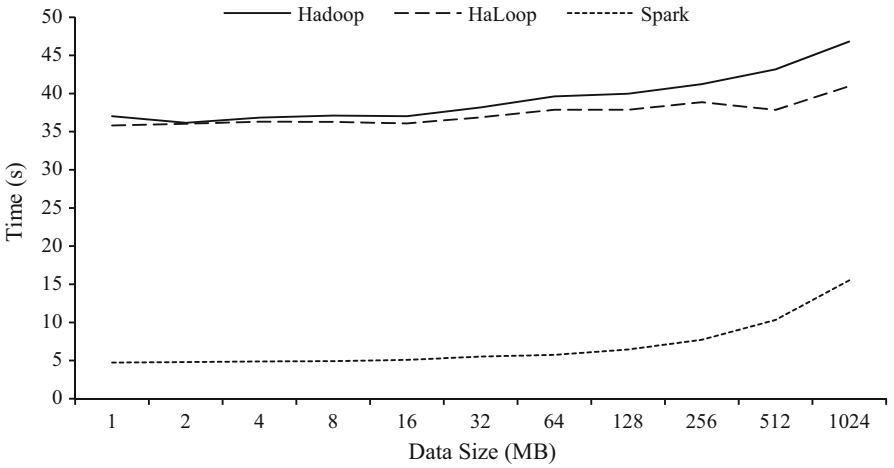

Fig. 11 Comparisons of efficiency with increasing data sizes

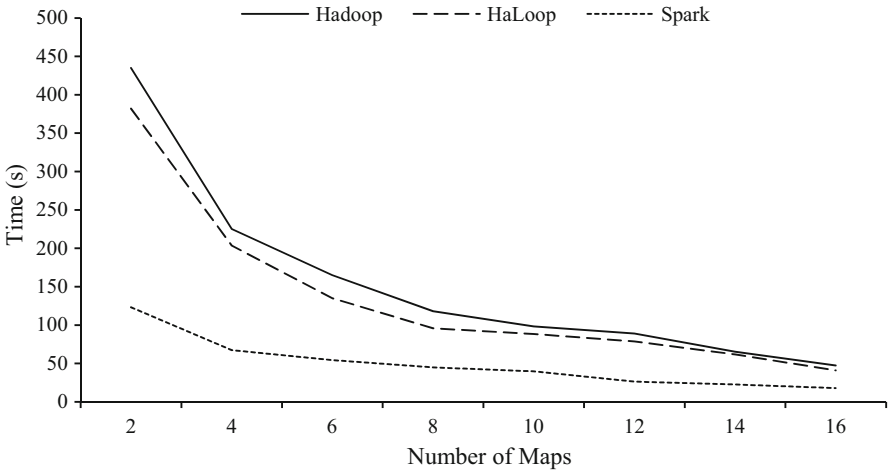

Fig. 12 Comparisons of efficiency with increasing Maps

with increasing number of Maps, Spark outperforms Hadoop and HaLoop. And also the algorithm performances in Aliyun are slightly better than those in our experimental small-scale cluster.

7 Conclusion

This paper presents a parallelized back propagation neural network algorithm PBPNN. To improve the algorithm accuracy, PBPNN employs ensemble techniques including bootstrapping and majority voting. Basically bootstrapping can help to maintain original data information in sub-dataset, whilst majority voting can generate strong classifier based on aggregating of weak classifiers. The experimental results show that PBPNN outperforms standalone BPNN in terms of accuracy and stability.

This paper also evaluates the algorithm efficiency on different distributed computing platforms. The experimental results also indicate that among the platforms, Spark has the best ability in dealing with iterative jobs. With extra improvements for iterative job, HaLoop can also improve the algorithm efficiency. Hadoop, as a standard MapReduce implementation, it doesn't supply enough support for iterative job, which results in the worst performance in terms of efficiency.

Acknowledgments The authors would like to appreciate the support from the National Natural Science Foundation of China (No. 51437003) and the National Basic Research Program (973) of China under Grant 2014CB340404.

Conflict of interest The authors declare that there is no conflict of interests regarding the publication of this article.

References

1. "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. Gu, R., Shen, F., Huang, Y.: A parallel computing platform for training large scale neural networks. In: IEEE International Conference on Big Data, pp. 376–384 (2013)
3. Long, L.N., Gupta, A.: Scalable massively parallel artificial neural networks. J. Aerosp. Comput. Inf. Commun. **5**(1), 3–15 (2008)
4. Liu, Y., Yang, J., Huang, Y., Xu, L., Li, S., Qi, M.: MapReduce based parallel neural networks in enabling large scale machine learning. Comput. Intell. Neurosci. (2015). doi:10.1155/2015/297672
5. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**, 107–113 (2008)
6. Liu, Y., Li, M., Khan, M., Qi, M.: A mapreduce based distributed LSI for scalable information retrieval. Comput. Inf. **33**(2), 259–280 (2014)
7. Bu, Y., Howe, B., Balazinska, M., Ernst, M.D.: HaLoop: efficient iterative data processing on large clusters. In: 36th International Conference on Very Large Data Bases, Singapore (2010)
8. Wang, C., Tai, T., Huang, K., Liu, T., Chang, J., Shieh, C.: FedLoop: looping on federated MapReduce. In: IEEE 13th Conference on Trust, Security and Privacy in Computing and Communications, pp. 755–762. Beijing (2014)
9. Zhang, Y., Gao, Q., Gao, L., Wang, C.: iMapReduce: a distributed computing framework for iterative computation. In: IEEE International Parallel & Distributed Processing Symposium, pp. 1112–1121. Shanghai (2011)
10. Bhuiyan, M.A., Hasan, M.A.: An iterative MapReduce based frequent subgraph mining algorithm. IEEE Trans. Knowl. Data Eng. **27**(3), 608–620 (2015)

11. URL: http://hadoop.apache.org. Last accessed 25 May 2015
12. URL: http://mahout.apache.org. Last accessed 25 May 2015
13. URL: https://code.google.com/p/haloop/. Last accessed 25 May 2015
14. URL: http://spark.apache.org. Last accessed 25 May 2015
15. Jiang, J., Zhang, J., Yang, G., Zhang, D., Zhang, L.: Application of back propagation neural network in the classification of high resolution remote sensing image: take remote sensing image of beijing for instance. In: 18th International Conference on Geoinformatics, pp. 1–6. Beijing (2010)
16. Khoa, N., Sakakibara, K., Nishikawa, I.: Stock price forecasting using back propagation neural networks with time and profit based adjusted weight factors. In: International Joint Conference SICE-ICASE, pp. 5484–5488. Busan (2006)
17. Rizwan, M., Jamil, M., Kothari, D.P.: Generalized neural network approach for global solar energy estimation in India. IEEE Trans. Sustain. Energy **3**, 576–584 (2012)
18. Wang, Y., Li, B., Luo, R., Chen, Y., Xu, N., Yang, H.: Energy efficient neural networks for big data analytics. In: Design, Automation and Test in Europe Conference and Exhibition, pp. 1–2. Dresden (2014)
19. Nguyen, D., Widrow, B.: Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. In: International Joint Conference on Neural Networks, vol. 3, pp. 21–26. Washington (1990)
20. Kanan, H., Khanian, M.: Reduction of neural network training time using an adaptive fuzzy approach in real time applications. Int. J. Inf. Electron. Eng. **2**(3), 470–474 (2012)
21. Hasan, R., Taha, T.M.: Routing bandwidth model for feed forward neural networks on multicore neuromorphic architectures. In: International Joint Conference on Neural Networks, pp. 1–8. Dallas (2013)
22. Huqqani, A.A., Schikuta, E., Mann, E.: Parallelized neural networks as a service. In: International Joint Conference on Neural Networks, pp. 2282–2289. Beijing (2014)
23. Yuan, J., Yu, S.: Privacy preserving back-propagation neural network learning made practical with cloud computing. IEEE Trans. Parallel Distrib. Syst. **25**, 212–221 (2014)
24. Ikram, A.A., Ibrahim, S., Sardaraz, M., Tahir, M., Bajwa, H., Bach, C.: Neural network based cloud computing platform for bioinformatics. In: Systems Applications and Technology Conference (LISAT), pp. 1–6. Long Island (2013)
25. Rao, V., Rao, S.: Application of artificial neural networks in capacity planning of cloud based IT infrastructure. In: IEEE International Conference on Cloud Computing in Emerging Markets (CCEM), pp. 1–4. Bangalore (2012)
26. Gu, R., Shen, F., Huang, Y.: A parallel computing platform for training large scale neural networks. In: IEEE International Conference on Big Data, pp. 376–384. Silicon Valley (2013)
27. Liu, Z., Li, H., Miao, G.: MapReduce-based backpropagation neural network over large scale mobile data. In: Sixth International Conference on Natural Computation (ICNC 2010), pp. 1726–1730. Yantai (2010)
28. Hagan, M.H., Demuth, H.B., Beale, M.H.: Neural Network Design. PWS Publishing Company, Boston (1996)
29. Nasullah, K.A.: Parallelizing support vector machines for scalable image annotation. Ph.D. Thesis, Brunel University, UK (2011)
30. Lichman, M.: UCI machine learning repository [http://archive.ics.uci.edu/ml]. Irvine, University of California, School of Information and Computer Science, CA (2013)
31. Aliyun. http://www.aliyun.com. Last accessed 25 May 2015