Received April 7, 2014, accepted April 20, 2014, date of publication April 23, 2014, date of current version May 12, 2014.

Digital Object Identifier 10.1109/ACCESS.2014.2319813

Large-Scale Deep Belief Nets With MapReduce

KUNLEI ZHANG AND XUE-WEN CHEN, (Senior Member, IEEE)
Department of Computer Science, Wayne State University, Detroit, MI 48202, USA
Corresponding author: X.-W. Chen (xwen.chen@gmail.com)

ABSTRACT Deep belief nets (DBNs) with restricted Boltzmann machines (RBMs) as the building block have recently attracted wide attention due to their great performance in various applications. The learning of a DBN starts with pretraining a series of the RBMs followed by fine-tuning the whole net using backpropagation. Generally, the sequential implementation of both RBMs and backpropagation algorithm takes significant amount of computational time to process massive data sets. The emerging big data learning requires distributed computing for the DBNs. In this paper, we present a distributed learning paradigm for the RBMs and the backpropagation algorithm using MapReduce, a popular parallel programming model. Thus, the DBNs can be trained in a distributed way by stacking a series of distributed RBMs for pretraining and a distributed backpropagation for fine-tuning. Through validation on the benchmark data sets of various practical problems, the experimental results demonstrate that the distributed RBMs and DBNs are amenable to large-scale data with a good performance in terms of accuracy and efficiency.

INDEX TERMS Big data, deep learning, MapReduce, Hadoop, deep belief net (DBN), restricted Boltzmann machine (RBM).

I. INTRODUCTION

In recent years, deep learning has been receiving great popularity from both academia and industry due to its excellent performance in many practical problems. Deep belief nets (DBNs) with stacked restricted Boltzmann machines (RBMs) [1], [2] are one of the most important multiple-layer network architectures in deep learning. DBNs are generative models that are trained to extract a deep hierarchical representation of the input data by maximizing the likelihood of the training data. For the learning of a DBN, the weights and biases in each level RBM are initialized at first by using a greedy layer-wise unsupervised training [3], and all the weights and biases in the global net are then fine-tuned by using a (supervised) back-propagation algorithm [4].

Although DBNs have achieved great potential in various applications, such as image and object recognition [1], [2], [5], speech and phone recognition [6]–[8], information retrieval [9] and human motion modeling [10], the current sequential implementation of both RBM and the back-propagation based fine-tuning limits their application to large scale datasets due to the memory demanding and time-consuming computation. Scalable and efficient learning on emerging big data requires distributed computing for RBMs and DBNs.

MapReduce is a programming model introduced by Google [11] for processing massive datasets. It is typically used for parallel computing in a distributed environment on a large number of computation nodes. MapReduce has been implemented in several systems. One of the most powerful implementations is Apache Hadoop [12], a popular free open-source software framework. In addition to high data throughput, the Hadoop system can automatically not only manage the data partition, inter-computer communication and MapReduce task schedule across clusters of computers, but also handle computer failure with a high degree. With a suitable configuration of the Hadoop ecosystem to the problem at hand, all the users need to do is to design a *master* controller and provide a *Map* function and a *Reduce* function. Nevertheless, Hadoop does not easily allow iterative processing which is common in machine learning algorithms.

To make DBNs amenable to large-scale datasets stored on computer clusters, this paper develops a distributed learning paradigm for DBNs with MapReduce. We design proper *key-value* pairs for each level RBM, and the pre-training is achieved via layer-wise distributed learning of RBMs in the MapReduce framework. Subsequently, the fine-tuning is done via the use of a distributed back-propagation algorithm based on MapReduce. In particular, mrjob [13] is used in the implementation to automatically run multi-step MapReduce jobs, which provides a way for Hadoop to perform iterative computing required during both the training of RBMs and the back-propagation. Thus, the distributed learning of DBNs

is accomplished by stacking a series of distributed RBMs for pretraining and a distributed back-propagation for fine-tuning.

Recently, increasing attention on massive data and large scale network architectures has driven parallel implementation of deep learning techniques. Locally connected neural networks [14] and convolutional-alike neural networks [15] were successfully paralleled on computer clusters. Different from these works, this paper explores the performance of deep neural networks with unsupervised pretraining under distributed settings. In addition, parallel processing of deep unsupervised learning models, such as stacked RBMs and sparse coding, using graphical processors (GPUs) has been discussed in [16], but the use of GPUs may reduce model performance and is hardly scalable to big data due to limited memory (typically less than 6 gigabytes). Conversely, our work enjoys the benefit of high data throughput inherent in the MapReuce framework. To the best of our knowledge, this is the first work with implementation details of parallelizing RBMs and DBNs with the MapReuce framework. To leverage the data parallelism, we also propose a modified mini-batch approach for updating parameters.

The remaining of the paper is organized as follows. Section II provides some basic background on MapReduce, RBMs and DBNs. Section III elaborates the developed scheme for distributed RBMs and DBNs based on MapReduce. Experiments and evaluation results on the benchmark datasets are given in Section IV with respect to accuracy and scalability. Finally, this paper is concluded in Section V.

II. BACKGROUND

In this section, we will give a brief introduction of MapReduce and DBNs.

A. REVIEW OF MAPREDUCE

MapReduce provides a programming paradigm for performing distributed computation on computer clusters. Figure 1 gives an overview of the MapReduce framework. In a MapReduce system such as hadoop, the user program forks a *Master controller* process and a series of Map tasks (*Mappers*) and Reduce tasks (*Reducers*) at different computers (nodes of a cluster). The responsibilities of the Master involve creating some number of Mappers and Reducers and keeping track of the status of each Mapper and Reducer (executing, complete or idle).

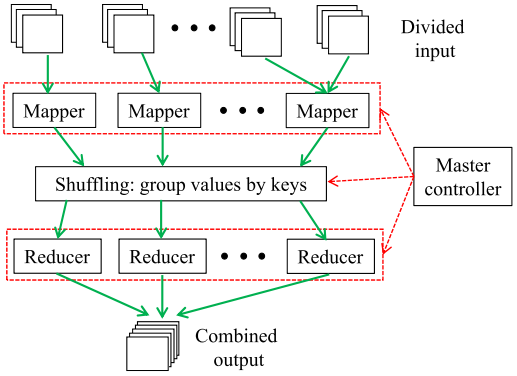


FIGURE 1. An overview of the MapReduce framework.

The computation in one MapReduce job consists of two phases, i.e., a map phase and a reduce phase. In the Map phase, the input dataset (stored in a distributed file system, e.g., HDFS) is divided into a number of disjoint subsets which are assigned to mappers in terms of <key, value> pairs. In parallel, each Mapper applies the user-specified map function to each input <key, value> pair and outputs a set of intermediate <key, value> pairs which are written to local disks of the map computers. The underlying system pass the locations of these intermediate pairs to the master who is responsible to notify the reducers about these locations. In the Reduce phase, when the reducers have remotely read all intermediate pairs, they sort and group them by the intermediate keys. Each Reducer literately invokes a user-specified reduce function to process all the values for each unique key and generate a new value for each key. The resulting <key, value> pairs from all of the Reducers are collected as final results which are then written to an output file.

In the MapReduce system, all the map tasks (and reduce tasks) are executed in a fully parallel way. Therefore, high-level parallelism can be achieved for data processing through the use of the MapReduce model. In recent years, there have been some parallel learning algorithms [17]–[23] using the MapReduce framework for efficient implementation.

B. REVIEW OF RBMS AND DBNS

1) RESTRICTED BOLTZMANN MACHINES

An RBM is composed of an input (visible) layer and a hidden layer with an array of connection weights between the input and hidden neurons but no connections between neurons of the same layer. Figure 2 illustrates the undirected graphical network of an RBM.

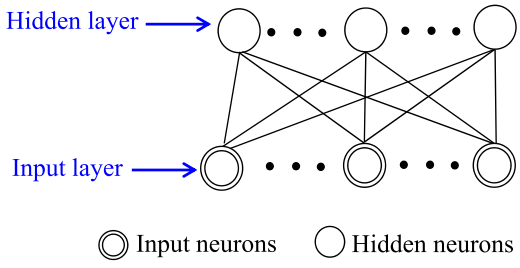


FIGURE 2. An illustration of an RBM network.

Rooted in the probabilistic model, RBM is also one particular type of energy model. Consider an RBM with the input layer x, hidden layer h, the energy function of the pair of observation and hidden variables is bilinear (assume the vectors in this paper are column vectors):

$$\text{Energy}\,(\mathrm{x}, \mathrm{h}) = -\mathrm{b}^T\mathrm{x} - \mathrm{c}^T\mathrm{h} - \mathrm{h}^T\mathrm{W}\mathrm{x}, \qquad (1)$$

where vectors b and c are the biases of the input layer and the hidden layer, respectively, and matrix W is the fully connected weight between the two layers. Thus, the distribution of input is tractable as follows:

$$P(\mathrm{x}) = \sum_{\mathrm{h}} P(\mathrm{x}, \mathrm{h}) = \sum_{\mathrm{h}} \frac{e^{-\mathrm{Energy}(\mathrm{x},\mathrm{h})}}{Z}.$$

where Z is the partition function. By introducing a new definition

$$\begin{aligned}\mathrm{freeE}(\mathrm{x}) &= -\log \sum_{\mathrm{h}} e^{-\mathrm{Energy}(\mathrm{x},\mathrm{h})} \\ &= -\mathrm{b}^T\mathrm{x} - \sum_i \log \sum_{\mathrm{h}_i} e^{\mathrm{h}_i^T(\mathrm{c}_i + \mathrm{W}_i\mathrm{x})}, \end{aligned} \tag{2}$$

the input likelihood can be expressed in an easier way as

$$P(\mathrm{x}) = \frac{e^{-\mathrm{freeE}(\mathrm{x})}}{\tilde{Z}}, \tag{3}$$

where $\tilde{Z} = \sum_{\mathrm{x}} e^{-\mathrm{freeE}(\mathrm{x})}$.

According to [1] and [24], the conditional distribution in RBM can be factorized due to the lack of input-input and hidden-hidden connections. That is to say, calculation of the conditional distribution can be decomposed into that on the single node $P(\mathrm{h}|\mathrm{x}) = \prod_i P(\mathrm{h}_i|\mathrm{x})$. In the binary case where hidden node takes either zero or one, the probability of hidden node taking value one happens to be a sigmoid function of the input as follows:

$$\begin{aligned} P(\mathrm{h}_i = 1|\mathrm{x}) &= \mathrm{sigmod}(\mathrm{c}_i + \mathrm{W}_i\mathrm{x}) \\ &= \frac{1}{1 + \mathrm{e}^{-(\mathrm{c}_i + \mathrm{W}_i\mathrm{x})}}. \end{aligned} \tag{4}$$

The essential goal of the training is actually for the hidden random variables to maintain the distribution of the input data as much as possible, say, find the optimal parameters $\boldsymbol{\Theta} = \{\mathrm{W}, \mathrm{b}, \mathrm{c}\}$ to maximize the input likelihood. According to the gradient descent method, the parameters can be iteratively updated proportional to the gradient of log-likelihood

$$\frac{\partial \log P(\mathrm{x})}{\partial \boldsymbol{\Theta}} = -\frac{\partial \mathrm{freeE}(\mathrm{x})}{\partial \boldsymbol{\Theta}} + \sum_{\tilde{x}} P(\tilde{x}) \frac{\partial \mathrm{freeE}(\tilde{x})}{\partial \boldsymbol{\Theta}}, \tag{5}$$

where $\tilde{x}$ denotes the reconstructed x. One commonly used updating rule to train RBM with approximate data log-likelihood gradient is contrastive divergence (CD) [25]. In CD, the second term in Eq. 5, statistically representing an average over all the possible inputs, is replaced with a single term since the iteration itself has done the average job. Thus the gradient can be written as

$$\Delta\boldsymbol{\Theta} \approx -\frac{\partial \mathrm{freeE}(\mathrm{x})}{\partial \boldsymbol{\Theta}} + \frac{\partial \mathrm{freeE}(\tilde{x})}{\partial \boldsymbol{\Theta}}. \tag{6}$$

One can run a Markov chain Monte Carlo (MCMC) to obtain the input reconstructed by the model. K-step CD takes the input x as the initial state x_1, and runs the chain for k times $\mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_{k+1}$ by reconstructing input using the learned model. Although longer MCMC chain promises better performance, the pain is there regarding the computational cost. Note that small values of k normally suffices for a good result, even in the case when $k = 1$.

2) DEEP BELIEF NETS

As a building block for deeper architecture, single RBM is stacked on top of each other taking the output of previous RBM as the input after parameters of each RBM are learned properly. Figure 3 gives an illustration of a DBN with stacked RBMs.

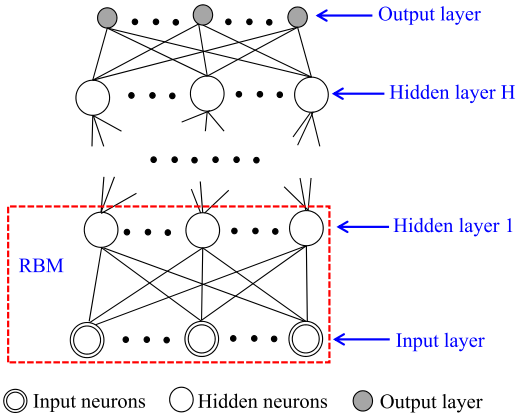


FIGURE 3. An illustration of a DBN with stacked RBMs.

In DBNs, concerning that parameters learned in previous RBMs might not be optimal for parameters learned afterwards, label information is involved for improvement on the discriminative power. Hinton et al. [1] proposed to integrate the label information into the input of top two layers and fine tune the stacked RBMs with a contrastive version of the "wake-sleep" algorithm, which performed a bottom-up pass followed by a top-down pass. As far as we are concerned, the process is tedious and the efficiency is not guaranteed. It is more straightforward to put the label layer on top as the output layer and fine-tune the parameters in all layers as in conventional multilayer perceptron (MLP) [2]. Therefore, the distributed implementation of stacked RBMs in this paper is conducted on the basis of MLP structure.

III. METHODOLOGY

This section will describe the main design of distributed RBMs and DBNs using MapReduce. The key is to design both a Map function and a Reduce function with proper input/output key-values pairs for the MapReduce jobs.

A. DISTRIBUTED RBM WITH MAPREDUCE

Given an input dataset $\mathcal{D} = \{\mathrm{x}_i | i = 1, 2, \ldots, N\}$, the goal of training an RBM is to learn the weights W, the biases b and c. In general, an iterative procedure with a number of epochs to reach convergence is necessary. In the case of distributed RBM with MapReduce, one MapReduce job is required in every epoch. In this paper, we automate the execution flow of multiple MapReduce jobs with the help of the mrjob [13] framework which enables the design of multi-step MapReduce jobs.

Procedure 1 mrjob_RBM

1: Initialize the variables
2: **for** each epoch **do**
3: *Map phase*
Input: <mapID, valuelist>
Take the values and perform Gibbs sampling to compute the approximate gradients of W, b and c
Output: <key, valuelist>
4: *Reduce phase*
Input: <key, valuelist>
Sum up the approximate gradients to get the increments of W, b and c, and then update them
Output: <mapID, valuelist>
5: **end for**

Output: the learned W, b and c

Since Gibbs sampling needs to do substantial matrix-matrix multiplications, it dominates the computation time during the training of RBM. Hence parallelizing Gibbs sampling on different data subsets in the Map phase will improve the efficiency. Procedure 1 outlines the pseudo code for distributed RBM. First, some variables are initialized such as the numbers of neurons for both visible and hidden layers, the weight W, the input layer bias b, the hidden layer bias c, the number of epochs (e.g., T) to run, and the hyper-parameters (e.g., learning rate, momentum factor). Then both the map phase and the reduce phase are repeated for T times. In each epoch, each mapper performs Gibbs sampling to compute the approximate gradients of W, b and c, and the reducer updates them with the calculated increments. (The details for the map phase and the reduce phase are provided in the following sections.) It is noteworthy that the format of key-value pairs emitted by the reducer should be the same as that of the input for the mapper so that the output of the reducer can be as the input of the mapper in the next epoch.

1) MAP PHASE

For each mapper, the corresponding mapper ID (a number) is as the input *key* and the input *value* is a list of values. Each of the values has two elements: the first is a string (e.g., $'W'$) identifying the type of this value, the second is the corresponding data (e.g., it can be an $M \times N$ matrix if the first element is $'W'$). In every epoch (except the first one), the *value* is the output of the reducer in the previous epoch, which is the updated W, b and c and their accumulated approximate gradients.

The input dataset $\mathcal{D}$ is divided into a number of disjoint subsets which are stored as a sequence of files (blocks) on Hadoop Distributed File System (HDFS). After reading all of the key-value pairs, each mapper loads one subset from the HDFS into memory. Given the information, each mapper can compute the approximate gradients of the weight and biases by going through all the mini-batches of the subset of the training dataset. Each mapper will emit three types

Procedure 2 mrjob_RBM::Map

Input: <mapID, valuelist> pairs

1: Parse *valuelist* into W, b, c, *delta*_W, *delta*_b, *delta*_c and t
2: **for** each data batch x **do**
3: In positive phase, compute P(h|x):
$$pos\_prob1 = \text{sigmoid}((\text{W} + delta\_\text{W})\text{x} + \text{c} + delta\_\text{c})$$
and sample the states based on P(h|x):
$$pos\_h\_state = (pos\_prob1 > \text{randn})$$
where 'randn' is a random number generator to generate a random number in [0, 1]
4: In negative phase, reconstruct the data:
$$\tilde{x} = \text{sigmoid}((\text{W} + delta\_\text{W})^T pos\_h\_state + \text{b} + delta\_\text{b})$$
and compute P(h|$\tilde{x}$):
$$pos\_prob2 = \text{sigmoid}((\text{W} + delta\_\text{W})\tilde{x} + \text{c} + delta\_\text{c})$$
5: Compute the approximate gradients of the weight and biases:
$$Gw = momentum * Gw + \frac{\text{x}.*pos\_prob1 - \tilde{x}.*pos\_prob2}{\#samplings\_batch}$$
$$Gb = momentum * Gb + \frac{\text{sum(x)} - \text{sum}(\tilde{x})}{\#samplings\_batch}$$
$$Gc = momentum * Gc + \frac{\text{sum}(pos\_prob1) - \text{sum}(pos\_prob2)}{\#samplings\_batch}$$
6: Update *delta*_W, *delta*_b and *delta*_c:
$$delta\_\text{W} = delta\_\text{W} + Gw$$
$$delta\_\text{b} = delta\_\text{b} + Gb$$
$$delta\_\text{c} = delta\_\text{c} + Gc$$
7: **end for**

Output: Emit intermediate key-value pairs $<'Gw'$, [W, Gw, t]>, $<'Gb'$, [b, Gb, t]>, and $<'Gc'$, [c, Gc, t]>

of intermediate *key*s: *delta*_W, *delta*_b and *delta*_c which represent the increments of W, b and c, respectively, and the intermediate *value*s have three elements: the value of *delta*_W, *delta*_b or *delta*_c, the corresponding increment and the current epoch index.

Procedure 2 provides the pseudo code for the map function executed by each mapper. Step 1 gets the parameters' values, where $t \in [1, T]$ is the epoch index. Steps 2–7 go through each data batch to compute the approximate gradients of both the weight and the biases, and update their increments. Finally, the intermediate key-value pairs are emitted as the output.

2) REDUCE PHASE

For the training of RBM, there are three reducers in ideal case. Each reducer reads as input one type (i.e., *delta*_W, *delta*_b or *delta*_c) of the intermediate key-value pairs, and applies the reduce function to first calculate the increments and then update parameter. The reducer takes the mapper ID as the output *key*, and the resulting increment and the updated parameter as the output *value*.

Procedure 3 gives the pseudo code for the reduce function executed by each reducer. Steps 1–10 are to process the

Procedure 3 mrjob_RBM::Reduce

Input: intermediate <key, valuelist> pairs

1: **if** *key* == *Gw* **then**
2: Parse *valuelist* into W, t and a list of *Gw* (denoted as *Gw_list*)
3: Compute the increment of the weight: *delta*_W ← sum(*Gw_list*)
4: Update W: W ← W + *delta*_W
5: **if** $t == T$ **then**
6: Save the learned W
7: **else**
8: Increase the epoch index: $t \leftarrow t + 1$
9: Emit key-value pairs to the mappers: <mapID, ['W', W]>, <mapID, ['*delta_W*', *delta*_W]> and <mapID, ['t', t] >
10: **end if**
11: **else if** *key* == *Gb* **then**
12: Parse *valuelist* into b, t and a list of *Gb* (denoted as *Gb_list*)
13: Compute the increment of the input layer bias: *delta*_b ← sum(*Gb_list*)
14: Update b: b ← b + *delta*_b
15: **if** $t == T$ **then**
16: Save the learned b
17: **else**
18: Emit key-value pairs to the mappers: <mapID, ['b', b]>, and <mapID, ['*delta_b*', *delta*_b]>
19: **end if**
20: **else if** *key* == *Gc* **then**
21: Parse *valuelist* into c, t and a list of *Gc* (denoted as *Gc_list*)
22: Compute the increment of the hidden layer bias: *delta*_c ← sum(*Gc_list*)
23: Update c:c ← c + *delta*_c
24: **if** $t == T$ **then**
25: Save the learned c
26: **else**
27: Emit key-value pairs to the mappers: <mapID, ['c', c]>, and <mapID, ['*delta_c*', *delta*_c]>
28: **end if**
29: **end if**

weight where Step 2 gets the current weight, epoch index, and a list of approximate gradients for weight, Steps 3–4 compute the weight increment and update the weight, Steps 5–10 decide to whether save the learned weight when it is the final epoch or increase the epoch index and emit the key-value pairs to the mappers. In a similar way, Steps 11–19 and Steps 20–28 are to process the input layer bias and hidden layer bias, respectively.

B. DISTRIBUTED DBN WITH MAPREDUCE

Considering a DBN with H hidden layers, the training of this distributed DBN consists of learning H distributed RBMs for the pre-training and one distributed back-propagation algorithm for fine-tuning the global network. In addition, a main controller is required to manage the entire learning process.

1) DISTRIBUTED RBMS FOR PRE-TRAINING

The bottom-level RBM is trained in the same way as that described in Section III-A. The training of the rest level RBMs is also similar to the bottom-level RBM except that the input dataset is changed accordingly. The input data for the lth ($H \geq l > 1$) level RBM will be the conditional probability of hidden nodes computed in the $(l-1)$th level RBM, that is

$$\begin{cases} P(\mathrm{h}_1|\mathrm{x}), & \text{when } l = 2; \\ P(\mathrm{h}_l|\mathrm{h}_{l-1}), & \text{when } H \geq l > 2. \end{cases} \tag{7}$$

Thus, the details of both the map function and the reduce function are omitted here.

2) DISTRIBUTED BACK-PROPAGATION ALGORITHM FOR FINE-TUNING

In the completion of pre-training of all the hidden layers, it is time to gain discriminative power by simply putting the label layer on top of the network and iteratively tuning the weights of all the layers (i.e., $\mathrm{W}_1, \ldots, \mathrm{W}_{H+1}$). Actually, in the first few epochs (e.g., 5), we first fine-tune the weight W_{H+1} connecting the H hidden layer and the output layer, so that it has a reasonable initialization. Note that during the fine-tuning the 'weight' of each layer means the concatenation of the original weight and the bias.

For the distributed back-propagation based fine-tuning, the feed-forward and back-propagation procedure [4] to compute the gradient of weights using gradient descent is dominated the computation time. Thus, in each epoch, this procedure is executed parallely on each subset of the data in the map phase, and then the reducers compute the weight increments and update the weights.

Procedure 4 outlines the pseudo code for the distributed back-propagation based fine-tuning. Step 1 loads the pre-trained weights $\mathrm{W}_1, \ldots, \mathrm{W}_H$ and initialize the variables such as the weight W_{H+1} and some hyper-parameters. Steps 2–5 are for the map function and reduce function. In the map phase (Step 3), each mapper take the mapper ID as the input key, and the weight and its increment as the input value. For each data batch, the mappers calculate the gradient of weights and update the weight increments. Finally, each mapper emits the intermediate key-value pairs. In the reduce phase (Step 4), each reducer takes one or more type of weights, computes the weight increments, updates the weight, and then passes back to the mappers. In the final epoch, the reducers save the fine-tuned weights, which are the final output.

3) MAIN CONTROLLER DESIGN

In this section, we further design a main controller to manage the entire learning process of a DBN. The main controller schedules the running of MapReduce jobs for each level RBM and the fine-tuning.

Procedure 4 mrjob_Fine-Tune

1: Load the learned weights $W_1, \ldots, W_H$ during the pre-training and initialize the variables
2: **for** each epoch **do**
3: *Map phase*
– Input: <mapID, valuelist> pairs
– Parse *valuelist* into $W_1, \ldots, W_{H+1}$, *delta*_W_1, ..., *delta*_W_{H+1} and t
– for each data batch do
∘ Feedforward and back-propagation to get the gradients of the weights of all layers $GW_1, \ldots, GW_{H+1}$ using gradient descent
∘ Update *delta*_$W_1, \ldots,$ *delta*_W_{H+1}:
$delta\_W_1 \leftarrow delta\_W_1 - GW_1$
...
$delta\_W_{H+1} \leftarrow delta\_W_{H+1} - GW_{H+1}$
– end for
– Output: Emit intermediate key-value pairs $<'GW_1', [W_1, GW_1, t]>, \ldots,$ and $<'GW_{H+1}', [W_{H+1}, GW_{H+1}, t]>$
4: *Reduce phase*
– Input: intermediate <key, valuelist> pairs
– if $key == GW_i$ where $i \in [1, H+1]$ then
∘ Parse *valuelist* into W_i, t and a list of GW_i (denoted as GW_i_list)
∘ Compute the increment of the weight:
$delta\_W_i \leftarrow \text{sum}(GW_i\_list)$
∘ Update W_i: $W_i \leftarrow W_i - delta\_W_i$
∘ if $t == T$, Save the learned W_i
∘ else, increase the epoch index: $t \leftarrow t+1$ and emit key-value pairs to the mappers: <mapID, $['delta\_W_i', W_i, delta\_W_i]$> and <mapID, $['t', t]$ >
– end if
5: **end for**

Output: the fine-tuned $W_1, \ldots, W_{H+1}$

Procedure 5 outlines the pseudo code for the main controller of a DBN. Steps 1-11 are to run MapReduce jobs for all H levels of distributed RBMs. For the first level RBM, the input data will be the training dataset $\mathcal{D}$, and the pretrained weight W_1 and bias c_1 are saved for loading in the fine-tuning stage. For the other RBM levels, the input data will be $P(h_{l-1}|h_{l-2})$. Steps 12-14 are to run MapReduce jobs for the distributed back-propagation based fine-tuning. The pretrained weights and biases of all levels of RBM are loaded. The resulting weights and biases of all layers are saved as the final output.

Thus, a distributed DBN is trained with MapReduce programming model via the help of the *mrjob* framework. The training can be done off-line. Given a learned DBN, testing on new data samples can be directly performed.

IV. EXPERIMENTS AND RESULTS

This section will demonstrate the performance of the distributed RBMs and DBNs on several benchmark datasets

Procedure 5 MainControllerThread

Input: training dataset $\mathcal{D}$, number of RBM levels H

1: **for** each $l \in [1, H]$ **do**
2: **if** $l == 1$ **then**
3: Setup for the first level RBM:
Set the training dataset $\mathcal{D}$ as the input data
Set the number of input neurons, the number of hidden neurons, number of epoch to train, and hyper-parameters
4: Invoke mrjob_RBM (**Procedure 1**)
5: Save the learned weight W_1, bias c_1, and $P(h_1|x)$
6: **else**
7: Setup for other level RBM:
Set $P(h_{l-1}|h_{l-2})$ where $h_0 = x$, as the input data
Set the number of input neurons, the number of hidden neurons, number of epoch to train, and hyper-parameters
8: Invoke mrjob_RBM (**Procedure 1**)
9: Save the learned weight W_l, bias c_l, and $P(h_l|h_{l-1})$
10: **end if**
11: **end for**
12: Setup for fine-tuning:
Set the training dataset $\mathcal{D}$ and the corresponding labels as the input data
Load the pretrained weights and biases of all RBM levels
Set the number of epoch to train, and hyper-parameters
13: Invoke mrjob_Fine-tune (**Procedure 4**)
14: Save the final weights and bias of all layers
15: Exit

for various learning tasks. In particular, we investigate their accuracy, and the scalability under conditions of varying Hadoop cluster sizes and data samples.

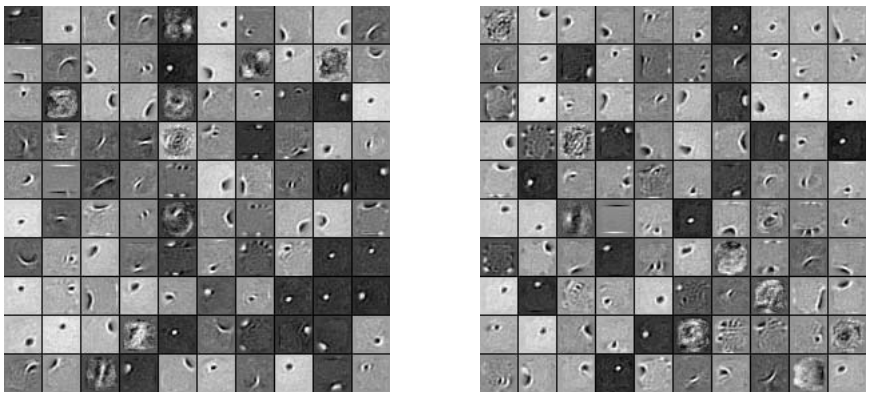
A. EXPERIMENTAL SETUP

The datasets we tested are the MNIST[1] for hand-written digits recognition, and the 20 Newsgroups[2] document set. For the MNIST dataset, there are 60,000 images as the training set and 10,000 images as the testing set. All the images was size-normalized and centered in a fixed size of 28 × 28 pixels. The intensity was normalized to have a value in [0, 1]. The labels are integers in [0, 9] indicating which digit the image presents. For the 20 Newsgroups dataset, there are 18,774 postings taken from the Usenet newsgroup collection with 11,269 training documents and 7,505 test documents. Each document is represented as a 2000-dimensional vector whose elements are the probabilities of the 2000 most frequently used words. The labels for each document are integers in [0, 19] indicating which topic the document belongs to. In this paper, the training set of original MNIST and original 20 Newsgroups is copied with 10-times, 20-times, 30-times, 40-times and 50-times,

[1] http://yann.lecun.com/exdb/mnist/
[2] http://qwone.com/ jason/20Newsgroups/

TABLE 1. Summary of training datasets.

MNIST Datasets	10-times	20-times	30-times	40-times	50-times
Dimension	600,000×784	1,200,000×784	1,800,000×784	2,400,000×784	3,000,000×784
Size ($\times 10^9$B)	1.75	3.5	5.25	7.0	8.75
20 Newsgroups Datasets	10-times	20-times	30-times	40-times	
Dimension	112,690×2,000	225,380×2,000	338,070×2,000	450,760×2,000	
Size ($\times 10^9$B)	1.8	3.6	5.4	7.2	



(a) (b)

FIGURE 4. Filters obtained by (a) and (b) RBM and the distributed RBM at epcoch 50.

which are summarized in Table 1, to evaluate the scalability performance.

All the experiments were performed on a cluster of 8 computers (nodes) where each is equipped with a 64-bit AMD octo-core dual-processor with the speed of 2.4 GHz, 96 GB RAM, and Linux RHEL. The computers are connected through 10Gbit Ethernet. The cluster is configured with Hadoop 1.0.4, Java 1.7.0, and Python 2.7.5 with mrjob 0.4.1.

We set the HDFS block size to be 64 MB and the replication factor to be 4. Each node is set to simultaneously run 26 mappers and 4 reducers in maximum. It should be noted that the cluster is generally shared with other users (except when we occupy all the cores.)

B. EXPERIMENTS FOR ACCURACY AND TRAINING TIME COMPARISON

The goal in this section is to compare the distributed RBMs and DBNs with their sequential versions (i.e., the original RBMs and DBNs) in terms of both testing accuracy and training time. To provide fair comparisons, we run both the sequential version and the distributed version in the same conditions. That is, in both cases, we utilize the same parameter setting including the training set (10-times of original MNIST dataset), the testing set (10,000 images), the network architecture (784-500 for RBMs, 784-500-500-2000-10 for DBNs), the initialization of the weight and the bias, the learning rate, the momentum factor, and the number of epoch to train. And both of them were programmed completely in the python codes. The sequential programs were run on one cpu while the distributed programs were run on 16 cpus of a node.

Figure 4 shows the filters (i.e., the weight) obtained by sequential RBM and the distributed RBM after epcoch 50. Both of them learned visually excellent weights. Table 2 and Table 3 provide the result comparison for RBM and DBN, respectively. One can see that both the distributed RBM and the distributed DBN obtained similar accuracy to the

TABLE 2. Reconstruction error and training time by RBM and the distributed RBM (with 16 cores) on 10-times of original MNIST dataset.

Algorithms	Reconstruction error	Training time
RBM	3.78×10^6	6.44 hrs
distributed RBM	3.81×10^6	1.31 hrs

TABLE 3. Testing error rate and training time by DBN and the distributed DBN (with 16 cores) on 10-times of original MNIST dataset.

Algorithms	Testing error	Training time
DBN	1.17%	31.04 hrs
distributed DBN	1.19%	10.35 hrs

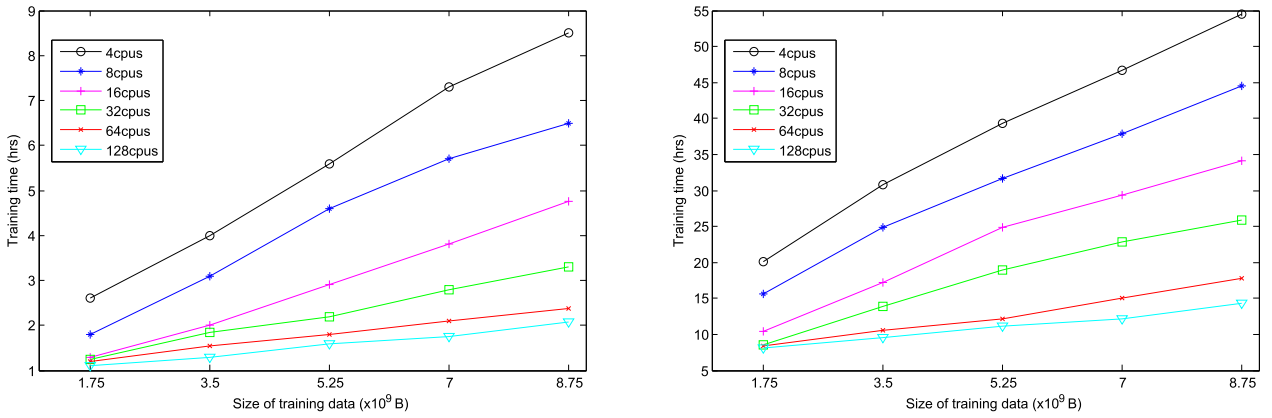


FIGURE 5. Training time versus data size for various numbers of cpus by (a) and (b) distributed RBM and DBN.

corresponding sequential versions but with much less training time.

C. EXPERIMENTS FOR THE SCALABILITY PERFORMANCE

We evaluate the parallel performance of distributed RBMs and DBNs with respect to the *scalability*. In particular, we study running time (for training) versus data size for various numbers of cpus. In the implementation, the distributed programs were run on the datasets summarized in Table 1 using the number of cpus varying from 4 to 128. Figure 5 shows the results on various times of original MNIST dataset obtained by the distributed RBMs and DBNs. First, one can observe that the running time raises as the increased size of training data, and significantly decreases as using more cpus. Next, it is also observed that the benefit of using more cpus decreases when the size of data becomes small. The reason behind this is system overhead (e.g., communication costs, job setup for per iteration) dominates the processing time when the size of data is small. Actually, it is the overhead in the Hadoop system that makes the speedup when adding more cpus is sublinear with respect to the number of cpus.

We also performed the scalability experiment on the 20 Newsgroups dataset. An intuitively setted network architecture, i.e., 2000-500-1000-20, is trained. It should be noted that our purpose of testing on 20 Newsgroups is for measuring the scalability performance of the developed distributed DBNs but not for achieving the accurate document classification or retrieval. The specific running time of distributed DBNs on various times of 20 Newsgroups dataset using different number of cpus is listed in Table 4. Note the similar observations as before, which expects that distributed DBNs can be applied to other large-scale applications.

TABLE 4. Running time (hrs) of distributed DBN.

	Size of training data ($\times 10^9$B)			
No. of cpus	1.8	3.6	5.4	7.2
4	19.2	23.4	28.9	33.4
8	16.5	19.8	23.2	28.0
16	12.9	16.7	20.1	22.4
32	8.8	11.9	15.1	17.9
64	8.1	9.7	11.4	13.3
128	7.7	8.6	10.1	10.9

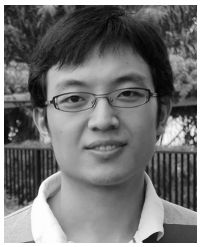
V. CONCLUSION

In this paper, we have presented a type of distributed DBNs using MapReduce which can be accomplished by stacking several levels of distributed RBMs and then using a distributed back-propagation algorithm for the fine-tuning. Concerning the communication cost, only data-level parallelism is performed in the developed distributed algorithm since a fully connected multi-layer network is considered. Experiments demonstrate that the distributed DBNs not only have achieved similar testing accuracy on the large version of MNIST dataset to the sequential version, but also scale well even there is big amount overhead for Hadoop system to do iterative computing. We expect the developed distributed DBNs would be able to process other massive datasets with good performances. In the future work, we will conduct the experiments on more large-scale learning problems.

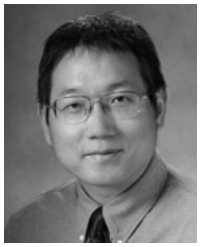
REFERENCES

[1] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, 2006.
[2] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.
[3] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in *Advances in Neural Information Processing Systems*, vol. 19. Cambridge, MA, USA: MIT Press, 2007, p. 153.
[4] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, Oct. 1986.
[5] V. Nair and G. E. Hinton, "3D object recognition with deep belief nets," in *Advances in Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2009, pp. 1339–1347.

[6] G. Hinton *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, Nov. 2012.
[7] G. E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition," *IEEE Trans. Audio, Speech, Lang. Process.*, vol. 20, no. 1, pp. 30–42, Jan. 2012.
[8] A.-R. Mohamed, T. N. Sainath, G. Dahl, B. Ramabhadran, G. E. Hinton, and M. A. Picheny, "Deep belief networks using discriminative features for phone recognition," in *Proc. IEEE ICASSP*, May 2011, pp. 5060–5063.
[9] R. Salakhutdinov and G. Hinton, "Semantic hashing," *Int. J. Approx. Reasoning*, vol. 50, no. 7, pp. 969–978, 2009.
[10] G. W. Taylor, G. E. Hinton, and S. T. Roweis, "Modeling human motion using binary latent variables," in *Advances in Neural Information Processing Systems*. Cambridge, MA, USA: MIT Press, 2006, pp. 1345–1352.
[11] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[12] (2014, Jan. 20). *Apach Hadoop* [Online]. Available: http://hadoop.apache.org/
[13] (2014, Jan. 20). *mrjob* [Online]. Available: http://pythonhosted.org/mrjob/
[14] J. Dean *et al.*, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems*. Cambridge, MA, USA: MIT Press, 2012, pp. 1232–1240.
[15] Q. Le *et al.*, "Building high-level features using large scale unsupervised learning," in *Proc. Int. Conf. Mach. Learning*, 2012.
[16] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proc. IEEE 26th Annu. Int. Conf. Mach. Learning*, vol. 9. Jun. 2009, pp. 873–880.
[17] C. Chu *et al.*, "Map-reduce for machine learning on multicore," in *Neural Information Processing Systems*, vol. 19. Cambridge, MA, USA: MIT Press, 2007, p. 281.
[18] S. Papadimitriou and J. Sun, "DisCo: Distributed co-clustering with map-reduce: A case study towards petabyte-scale end-to-end mining," in *Proc. 8th IEEE ICDM*, Dec. 2008, pp. 512–521.
[19] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo, "PLANET: Massively parallel learning of tree ensembles with MapReduce," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1426–1437, 2009.
[20] T. Sun, C. Shu, F. Li, H. Yu, L. Ma, and Y. Fang, "An efficient hierarchical clustering method for large datasets with map-reduce," in *Proc. Int. Conf. Parallel and Distributed Comput., Appl. Technol.*, 2009, pp. 494–499.
[21] W. Zhao, H. Ma, and Q. He, "Parallel K-means clustering based on MapReduce," in *Proc. Int. Conf. Cloud Comput.*, 2009, pp. 674–679.
[22] Y. He *et al.*, "MR-DBSCAN: An efficient parallel density-based clustering algorithm using MapReduce," in *Proc. IEEE 17th ICPADS*, Dec. 2011, pp. 473–480.
[23] K. Zhai, J. Boyd-Graber, N. Asadi, and M. L. Alkhouja, "Mr. LDA: A flexible large scale topic modeling package using variational inference in MapReduce," in *Proc. 21st Int. Conf. World Wide Web*, 2012, pp. 879–888.
[24] Y. Bengio, *Learning Deep Architectures for AI*, vol. 2. Norwell, MA, USA: Now Publishers, 2009.
[25] M. A. Carreira-Perpinan and G. E. Hinton, "On contrastive divergence learning," in *Proc. Int. Workshop Artificial Intell. and Statist.*, 2005, pp. 33–40.



KUNLEI ZHANG received the bachelor's and master's degrees from the School of Communication Engineering, Jilin University, Changchun, China, in 2005 and 2007, respectively, and the Ph.D. degree from the School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore, in 2013. He is currently a (Post-Doctoral) Research Associate with the Department of Computer Science, Wayne State University, Detroit, MI, USA. His current research interests are in machine learning, and biomedical imaging analysis and processing.



XUE-WEN CHEN (M'00–SM'03) is currently a Professor and the Chair with the Department of Computer Science, Wayne State University, Detroit, MI, USA. He received the Ph.D. degree from Carnegie Mellon University, Pittsburgh, PA, USA, in 2001. He is currently serving as an Associate Editor or an Editorial Board Member for several international journals, including the IEEE Access, *BMC Systems Biology*, and the IEEE Transactions on Information Technology in Biomedicine. He served as a Conference Chair or Program Chair for a number of conferences, such as the 21st ACM Conference on Information and Knowledge Management in 2012 and the 10th IEEE International Conference on Machine Learning and Applications in 2011. He is a Senior Member of the IEEE Computer Society. His current research interests include big data machine learning, data mining, and bioinformatics.