An Efficient Map Reduce for Large Scale Deep Belief Nets

Mahankalaiah Bijjili
M.Tech Student, Department of CSE
CMR College of Engineering and Technology, Hyderabad

Mr. Venkateswara Rao
Assistant Professor, Department of CSE
CMR College of Engineering and Technology, Hyderabad

Abstract – Recent years, various applications acquire a great performance due to wide attention of Deep belief nets (DBNs) with restricted Boltzmann machines (RBMs). The learning of a DBN starts with pretraining a series of the RBMs followed by fine-tuning the whole net using back propagation. Generally, the sequential implementation of both RBMs and back propagation 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 back propagation algorithm using Map Reduce, a popular parallel programming model. Thus, the DBNs can be trained in a distributed way by stacking a series of distributed RBMs for pertaining and a distributed back propagation for ne-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 productivity.

Keywords – Big Data, Deep Learning, Map Reduce, Hadoop, Deep Belief Net (DBN), Restricted Boltzmann Machine (RBM).

I. INTRODUCTION

In today’s deep learning has been receiving great popularity from both academics and organizational level due to its excellent performance in many practical problems. Deep belief nets (DBNs) with stacked restricted Boltzmann machines (RBMs) 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 rest by using a greedy layer-wise unsupervised training and all the weights and biases in the global net are then ne-tuned by using a (supervised) back-propagation algorithm. Although DBNs have achieved great potential in various applications, such as image and object recognition, speech and phone recognition, information retrieval and human motion modeling 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.
framework. To the best of our knowledge, this is the rest work with implementation details of parallelizing RBMs and DBNs with the MapReduce framework. To leverage the data parallelism, we also propose a modified mini-batch approach for updating parameters.

II. RELATED WORK

In this module, we will give a brief introduction of Map Reduce and DBNs.

A. Review Of Map Reduce

Map Reduce provides a programming paradigm for performing distributed computation on computer clusters. Figure 1 gives an overview of the Map Reduce framework. In a Map Reduce 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).

The computation in one Map Reduce 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 inter-mediate <key, value> pairs which are written to local disks of the map computers. The underlying systems 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 literally 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 all results which are then written to an output le.

The Map Reduce 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 Map Reduce model. In recent years, there have been some parallel learning algorithms using the Map Reduce framework for efficient implementation.

Review Of RBMS and DBNS

1. Restricted Boltzmann Machines

Restricted Boltzmann Machines essentially perform a binary version of factor analysis. (This is one way of thinking about RBMs; there are, of course, others, and lots of different ways to use RBMs, but I’ll adopt this approach for this post.) Instead of users rating a set of movies on a continuous scale, they simply tell you whether they like a movie or not, and the RBM will try to discover latent factors that can explain the activation of these movie choices.

More technically, a Restricted Boltzmann Machine is a stochastic neural network (neural network meaning we have neuron-like units whose binary activations depend on the neighbors they’re connected to; stochastic meaning these activations have a probabilistic element) consisting of:

- One layer of visible units (users’ movie preferences whose states we know and set);
- One layer of hidden units (the latent factors we try to learn); and
- A bias unit (whose state is always on, and is a way of adjusting for the different inherent popularities of each movie).

Furthermore, each visible unit is connected to all the hidden units (this connection is undirected, so each hidden unit is also connected to all the visible units), and the bias unit is connected to all the visible units and all the hidden units. To make learning easier, we restrict the network so that no visible unit is connected to any other visible unit and no hidden unit is connected to any other hidden unit.

For example, suppose we have a set of six movies (Harry Potter, Avatar, LOTR 3, Gladiator, Titanic, and Glitter) and we ask users to tell us which ones they want to watch. If we want to learn two latent units underlying movie preferences – for example, two natural groups in our set of six movies appear to be SF/fantasy (containing Harry Potter, Avatar, and LOTR 3) and Oscar winners (containing LOTR 3, Gladiator, and Titanic), so we might hope that our latent units will correspond to these categories.

![Over view of Map Reduce framework](image1)

![Over view of RBMS network](image2)
I. State Activation

Restricted Boltzmann Machines and neural networks in general, work by updating the states of some neurons given the states of others, so let’s talk about how the states of individual units change. Assuming we know the connection weights in our RBM (we’ll explain how to learn these below), to update the state of unit i:

Compute the activation energy \( a_i = \sum_j w_{ij} x_j \) of unit i, where the sum runs over all units j that unit i is connected to, \( w_{ij} \) is the weight of the connection between i and j, and \( x_j \) is the 0 or 1 state of unit j. In other words, all of unit i’s neighbors send it a message, and we compute the sum of all these messages.

Let \( p_i = \sigma(a_i) \), here \( \sigma(x) = 1/(1+\exp(-x)) \) is the logistic function. Note that \( p_i \) is close to 1 for large positive activation energies, and \( p_i \) is close to 0 for negative activation energies.

We then turn unit i on with probability \( p_i \), and turn it off with probability \( 1 - p_i \).

(In layman’s terms, units that are positively connected to each other try to get each other to share the same state (i.e., be both on or off), while units that are negatively connected to each other are enemies that prefer to be in different states.)

For example, let’s suppose our two hidden units really do correspond to SF/fantasy and Oscar winners.

If Alice has told us her six binary preferences on our set of movies, we could then ask our RBM which of the hidden units her preferences activate (i.e., ask the RBM to explain her preferences in terms of latent factors). So the six movies send messages to the hidden units, telling them to update themselves. (Note that even if Alice has declared she wants to watch Harry Potter, Avatar, and LOTR 3, this doesn’t guarantee that the SF/fantasy hidden unit will turn on, but only that it will turn on with high probability. This makes a bit of sense: in the real world, Alice wanting to watch all three of those movies makes us highly suspect she likes SF/fantasy in general, but there’s a small chance she wants to watch them for other reasons. Thus, the RBM allows us to generate models of people in the messy, real world.)

Conversely, if we know that one person likes SF/fantasy (so that the SF/fantasy unit is on), we can then ask the RBM which of the movie units that hidden unit turns on (i.e., ask the RBM to generate a set of movie recommendations). So the hidden units send messages to the movie units, telling them to update their states. (Again, note that the SF/fantasy unit being on doesn’t guarantee that we’ll always recommend all three of Harry Potter, Avatar, and LOTR 3 because, hey, not everyone who likes science fiction liked Avatar.)

II. Learning Weights

So how do we learn the connection weights in our network? Suppose we have a bunch of training examples, where each training example is a binary vector with six elements corresponding to a user’s movie preferences. Then for each epoch, do the following:

Take a training example (a set of six movie preferences). Set the states of the visible units to these preferences.

Next, update the states of the hidden units using the logistic activation rule described above: for the jth hidden unit, compute its activation energy \( a_j = \sum_i w_{ij} x_i \), and set \( x_j \) to 1 with probability \( \sigma(a_j) \) and to 0 with probability \( 1 - \sigma(a_j) \). Then for each edge \( e_{ij} \), compute \( \text{Positive}(e_{ij}) = x_i \Lambda x_j \) (i.e., for each pair of units, measure whether they’re both on).

Now reconstruct the visible units in a similar manner: for each visible unit, compute its activation energy \( a_i \), and update its state. (Note that this reconstruction may not match the original preferences.) Then update the hidden units again, and compute \( \text{Negative}(e_{ij}) = x_i \Lambda x_j \) for each edge.

Update the weight of each edge \( e_{ij} \) by setting \( w_{ij} = w_{ij} + L \times (\text{Positive}(e_{ij}) - \text{Negative}(e_{ij})) \), where \( L \) is a learning rate.

Repeat over all training examples.

Continue until the network converges (i.e., the error between the training examples and their reconstructions falls below some threshold) or we reach some maximum number of epochs.

Why does this update rule make sense? Note that in the first phase, \( \text{Positive}(e_{ij}) \) measures the association between the ith and jth unit that we want the network to learn from our training examples;

In the “reconstruction” phase, where the RBM generates the states of visible units based on its hypotheses about the hidden units alone, \( \text{Negative}(e_{ij}) \) measures the association that the network itself generates (or “daydreams” about) when no units are fixed to training data.

So by adding \( \text{Positive}(e_{ij}) - \text{Negative}(e_{ij}) \) to each edge weight, we’re helping the network’s daydreams better match the reality of our training examples.

(You may hear this update rule called contrastive divergence, which is basically a funky term for “approximate gradient descent”.)

2. DEEP BELIEF NETS

DBNs are building block of 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. 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. proposed to integrate the label information into the input of top two layers and use 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 ne-tune the parameters in all layers as in conventional multilayer perceptron (MLP). Therefore, the
distributed implementation of stacked RBMs in this paper is conducted on the basis of MLP structure.

**III. EXISTING SYSTEM**

The performance benefits of distributing a deep belief network across multiple machines depend on the connectivity structure and computational needs of the data model. Models with a large number of parameters or high computational demands typically benefit from access to more CPUs and memory, and also cluster of large data sets, up to the point where communication costs dominate. Obviously the distributed deep belief nets without map reduce to the large scale deep networks pretraining with thousands of parameters and it can takes computation time very high and pretraining of RBms fail and also data transfer between clusters of networks neurons communications difficult and will increase the communication cost automatically.

The parallel iterative model on large volume of data sets it can fail to communicate all thousands of parameters and also take much more time to retrieve the results with effective manner.

**IV. PROPOSED WORK**

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

**A. Distributed RBM With Map Reduce**

Given an input dataset D D 1; 2; : : : ; N g, 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 Map Reduce, one Map Reduce job is required in every epoch. In this paper, we automate the execution of multiple Map Reduce jobs with the help of the mrjob [13] framework which enables the design of multi-step Map Reduce jobs.

**Procedure 1 mrjob_RBM**

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

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.

**I. 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 rest 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 N matrix if the rest element is ‘W’). In every epoch (except the rest 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 D is divided into a number of disjoint subsets which are stored as a sequence of les (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 of intermediate keys: delta_W, delta_b and delta_c which represent the increments of W, b and c, respectively, and the intermediate values 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 2 [1; T] is the epoch index. Steps 2 through 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.

**II. 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


**Procedure 2** mrjob_RBM::Map

**Input:** <mapID, value list> pairs

1: Parse value list 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_xjx:
   
   \[ \text{pos}_\text{prob}1 \text{D sigmoid}((W C \text{delta}_W)x)C \]
   
   c C delta_c
   
   and sample the states based on P_xjx:
   
   \[ \text{pos}_\text{h}\text{state} \text{D (pos}_\text{prob}1 > \text{randn)} \]

   where `randn' is a random number generator to generate a random number in [0; 1]

4: In negative phase, reconstruct the data:
   
   \[ xQ D \text{sigmoid}((W C \text{delta}_W)^T) \]
   
   \[ \text{pos}_\text{h}\text{state} C b C \text{delta}_b \]

   and compute P_xjQx:
   
   \[ \text{Pos}_\text{prob}2 D \text{sigmoid}((W C \text{delta}_W) \]
   
   \[ xQ C C \text{delta}_c \]

5: Compute the approximate gradients of the weight and biases:
   
   \[ Gw \text{ D momentum } Gw c x: \text{pos}_\text{prob}1 xQ: \text{pos}_\text{prob}2 \]

6: Update delta_W, delta_b and delta_c:
   
   \[ \text{delta}_W D \text{delta}_W C \]
   
   \[ Gw \text{delta}_b D \text{delta}_b C \]
   
   \[ Gb \text{delta}_c D \text{delta}_c C \]

7: end for

**Output:** Emit intermediate key-value pairs \(<0Gw, [W, Gw, t]>\), \(<0Gb, [b, Gb, t]>, \text{and}<0Gc, [c, Gc, t]>

key-value pairs, and applies the reduce function to rest 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 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 null 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.

**Procedure 3** mrjob_RBM::Reduce

**Input:** intermediate <key, value list> pairs

1: if key DD Gw then

2: Parse value list into W, t and a list of Gw (denoted as

   \[ Gw\text{_list} \]

3: Compute the increment of the weight:
   
   \[ \text{delta}_W \text{sum}(Gw\text{_list}) \]

4: Update W: W W C delta_W

5: if t DD T then

6: Save the learned W

7: else

8: Increase the epoch index: t t C 1

9: Emit key-value pairs to the mappers: <mapID, \([0W, \text{delta}_W]>, \text{and}<\text{mapID,}[0\delta\text{delta}_W, \text{delta}_W]>\)

10: end if

11: else if key DD Gb then

12: Parse value list into b, t and a list of Gb (denoted as Gb_list)

13: Compute the increment of the input layer bias:
   
   \[ \text{delta}_b \text{sum}(Gb\text{_list}) \]

14: Update b: b b C delta_b

15: if t DD T then

16: Save the learned b

17: else

18: Emit key-value pairs to the mappers: <mapID, \([0b, b]>, \text{and}<\text{mapID,}[0\delta\text{delta}_b, \text{delta}_b]>\)

19: end if

20: else if key DD Gc then

21: Parse value list into c, t and a list of Gc (denoted as Gc_list)

22: Compute the increment of the hidden layer bias:
   
   \[ \text{delta}_c \text{sum}(Gc\text{_list}) \]

23: Update c: c C delta_c

24: if t DD T then

25: Save the learned c

26: else

27: Emit key-value pairs to the mappers: <mapID, \([0c, c]>, \text{and}<\text{mapID,}[0\delta\text{delta}_c, \text{delta}_c]>\)

28: end if

29: end if

1. Distributed DBN with Map Reduce

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 re-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 l > 1) level RBM will be the conditional probability of hidden nodes computed in the (l 1)th level RBM, the pre-training and one distributed back-propagation algorithm for re-tuning the global network. In addition, a main controller is required to manage the entire learning process.
II. 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 $l$th ($H > l > 1$) level RBM will be the conditional probability of hidden nodes computed in the $(l - 1)$th level RBM.

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., $W_1; \ldots; W_H$). Actually, in the rest few epochs (e.g., 5), we rest ne-tune the weight $W_H$ connecting the $H$ hidden layer and the output layer, so that it has a reasonable initialization. Note that during the ne-tuning the 'weight' of each layer means the concatenation of the original weight and the bias.

For the distributed back-propagation based ne-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 parallel 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 ne-tuning. Step 1 loads the pre-trained weights $W_1; \ldots; W_H$ and

Procedure 4 mrjob_Fine-Tune

1: Load the learned weights $W_1; \ldots; W_H$ during the pretraining and initialize the variables
2: for each epoch do
   2.1: Map phase
      2.1.1: Input: <mapID, valuelist> pairs
      2.1.2: Parse valuelist into $W_1; \ldots; W_H$; $\delta_W_1; \ldots; \delta_W_H$ and $t$
      2.1.3: for each data batch do
         2.1.3.1: Feedforward and back-propagation to get the gradients of the weights of all layers $GW_1; \ldots; GW_H$ using gradient descent
         2.1.3.2: Update $\delta_W_1; \ldots; \delta_W_H$:
         2.1.3.3: $\delta_W_H = \delta_W_H + GW_H$
         2.1.3.4: $\delta_W_C1 = \delta_W_H + \sum GW_C1$
      2.1.4: end for
   2.2: Reduce phase
      2.2.1: Input: intermediate <key, value list> pairs if key $DD$ $GW_i$ where $i \in [1; H C]$ then
      2.2.2: Parse valuelist into $W_i$ and a list of $GW_i$ (denoted as $GW_list$)
      2.2.3: Compute the increment of the weight:
      2.2.4: $\delta_W_i = \sum GW_i$ (list)
      2.2.5: Update $W_i$: $W_i, \delta_W_i$ if $t$
      2.2.6: $DDT$, Save the learned $W_i$
      2.2.7: increase the epoch index: $t + 1$ and emit key-value pairs to the mappers: <mapID, $\delta_W_H; W_0, \delta_W_H$> and <mapID, $\delta_W_i; t$>
   2.3: end for
3: Output: the fine-tuned $W_1; \ldots; W_H$

initialize the variables such as the weight $W_H$ and some hyper-parameters. Steps 2 5 are for the map function and reduce function. In the map phase (Step 3), each mapper takes 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 last epoch, the reducers save the ne-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 Map Reduce jobs for each level RBM and the ne-tuning.

Procedure 5 Main Controller Thread

Input: training dataset $D$, number of RBM levels $H$
1: for each $l \in [1; H]$ do
2: if $l = 1$ then
3: Setup for the rest level RBM:
   3.1: Set the training dataset $D$ as the input data
   3.2: 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 $P_{h1}\ h_j$
6: else
7: Setup for other level RBM:
   7.1: Set $P_{h_l}\ h_j$, where $h \in D, x$, 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 $W_1$, bias $c_1$, and $P_{h_l}\ h_j$
10: end if
11: end for
12: Setup for ne-tuning:
   12.1: Set the training dataset $D$ and the corresponding labels as the input data
   12.2: Load the pretrained weights and biases of all RBM levels Set the number of epoch to train, and hyper-parameters

All copyrights reserved by NCRTCST-2015, Departments of CSE
CMR College of Engineering and Technology, Kandlakoya (V), Medchal Road, Hyderabad, India.
Published by IJECC (www.ijecc.org)
13: Invoke mrjob_Fine-tune (Procedure 4)
14: Save the final weights and bias of all layers
15: Exit

Procedure 5 outlines the pseudo code for the main controller of a DBN. Steps 1-11 are to run Map Reduce jobs for all \( H \) levels of distributed RBMs. For the rst level RBM, the input data will be the training dataset \( D \), and the pretrained weight \( W_1 \) and bias \( c_1 \) are saved for loading in the ne-tuning stage. For the other RBM levels, the input data will be \( P \cdot h_{j-1} \cdot j \cdot h_j \). Steps 12-14 are to run Map Reduce jobs for the distributed back-propagation based ne-tuning. The pertained weights and biases of all levels of RBM are loaded. The resulting weights and biases of all layers are saved as the nal output.

Thus, a distributed DBN is trained with Map Reduce 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.

V. EXPERIMENTS AND RESULTS

This section will demonstrate the performance of the distributed RBMs and DBNs on several benchmark datasets for various learning tasks. In particular, we investigate their accuracy, and the scalability under conditions of varying Hadoop cluster sizes and data samples.

I. 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 were size-normalized and centered in a fixed size of 28x28 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.

All the experiments were performed on a cluster of 8 computers (nodes) where each is equipped with a 64-bit AMD octacore dual-processor with the speed of 2.4 GHz, 96 GB RAM, and Linux RHEL. The computers are connected through 10Gb 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.)

II. EXPERIMENTS FOR ACCURACY AND TIME COMPARISON

Table 1: Summery of training data sets

![](image)

(a) Filters obtained by (a) and (b) RBM and Distributed RBM epoch 50.
The goal in this section is to compare the distributed RBMs and DBNs with their sequential versions of 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 letters (i.e., the weight) obtained by sequential RBM and the distributed RBM after epoch 50. Both of them learned visually excellent weights. Table 1 provides 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 corresponding sequential versions but with much less training time.

III.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 pro-grams 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 rises 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 sub linear with respect to the number of cpus.

We also performed the scalability experiment on the 20 Newsgroups dataset. Intuitively settled 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 expect that distributed DBNs, can be applied to other large-scale applications.

VI. CONCLUSION

In this paper we present a distributed DBNs using Map Reduce which can be accomplished by stacking several levels of distributed RBMs and then using a distributed back-propagation algorithm for the fine-tuning. Reducing the communication cost, only data-level parallelism is performed in the developed distributed algorithm since a fully connected multi-layer network is considered.

Our experiments show that our new large-scale training methods can use a cluster of machines to train even modestly sized deep networks significantly faster than a GPU. And also shows 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


