**CS6003 - Big Data Analytics**

**Assignment 4(i) – Mini Project on MR**

**Anushri, 2018103007**

**Varshini E, 2018103076**

**Batch N**

**Project Title** - ***Handwritten Digits Recognition using MapReduce-based Neural Networks***

**Project Objective**

The main objectives of the project are:

1. MapReduce implementation of the Unsupervised Restricted Boltzmann Machine (RBM) for pre-training the initial weights to improve the efficiency of back propagation and reach the solution in a faster approach.
2. MapReduce implementation of supervised back-propagation fine-tuning in multiple layer neural networks.
3. To ensure effective and reliable approaches for recognition of handwritten digits

**Problem Statement**

Handwriting is unique to everyone thereby it can be difficult to decipher handwritten documents and texts. As computerization becomes more prevalent, handwritten character recognition is indispensable. A wide range of algorithms can be used for recognition of handwritten digits, one of the most common being Neural Networks.

The original back-propagation neural network-based machine learning technique has two significant difficulties:

* It's challenging to retain both efficiency and accuracy when there is a huge amount of data.
* The workload on the system is exacerbated by redundant data.

The project aims to handle the above drawbacks by combining deep learning algorithms with a big data approach to deal with large-scale data. A **MapReduce-based handwriting character recognizer** is designed which will be able to achieve an efficiency improvement on training and practical large-scale data. We present a distributed learning paradigm for the RBMs and the backpropagation algorithm using MapReduce.

**Project Approach**

***What is MapReduce?***

MapReduce is a programming model for processing big data sets with a parallel, distributed algorithm on a cluster. MapReduce job consists of the mapper and reducer functions, of which the mapper function will extract key/value pairs from input data and transfer them into a list of intermediate key/value pairs, and reducer function merges these intermediate values corresponding to the same key generated from mapper function to produce output values.

***How can MapReduce be applied to Machine Learning/Deep Learning?***

The input value in the instance of Machine/Deep Learning will be data from a specific item that the machine will learn. The weights will be the intermediary key/value pair that a machine will use to verify whether it has successfully recognised the item. These weights are used by the Reducer function to compute the machine's acknowledgment of an intended object. If the precision of the training set differs significantly from the required precision, the MapReduce process will loop until an acceptable result is found.

***Workflow for Artificial Neural Network using MapReduce:*** A many-layered neural network could be effectively pre-trained one layer at a time, treating each layer in turn as an unsupervised Restricted Boltzmann Machine, then followed by supervised back-propagation fine-tuning.

***Why do we use the Restricted Boltzmann Machine (RBM)?***

We use a Restricted Boltzmann machine (RBM) to simulate an ensemble of binary vectors using a two-layer network, in which stochastic, binary pixels are coupled to stochastic, binary feature detectors using symmetrically weighted connections. 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. Large initial weights always lead to poor local minima when there are several layers, whereas small initial weights lead to low gradients in early levels. We must make the initial weights as close to the solution as possible by using a RBM.

**Citations:**

1. K. Zhang and X. Chen, "Large-Scale Deep Belief Nets With MapReduce," in IEEE Access, vol. 2, pp. 395-403, 2014, doi: 10.1109/ACCESS.2014.2319813.
2. Kairan Sun, Xu Wei, Gengtao Jia, Risheng Wang and Ruizhi Li, “Large-scale Artificial Neural Network: MapReduce-based Deep Learning”, 2015, doi: <https://arxiv.org/abs/1510.02709>

**Detailed Design Diagram**

Diagram

Description automatically generated

**RBM Mapper**

**Class Description:**

 This is the mapper of restricted Boltzmann machine (RBM) training part in deep learning MapReduce program. The task of this MapReduce is to train the rbm between two layers in unsupervised way and output the updated weight. Each mapper only trains the weights for one iteration using one test case. So, to train the weights of the whole neural network, the MapReduce program needs to execute (maxEpoch \* layerNum) times.

**Class Structure:**

It contains six parts:

1. configure() reads all the configurations and distributed cache from outside.
2. initialize() parse the input strings into parameters, and initialize parameters for algorithm.
3. getposphase() does the positive phase of RBM training
4. getnegphase() does the negative phase of RBM training
5. update() computes the update of weights using previous results and parameters
6. map() implements the mapper. It outputs the original key and updated value pair.

**Algorithm Description:**

This program trains Restricted Boltzmann Machine in which visible, binary, stochastic pixels are connected to hidden, binary, stochastic feature detectors using symmetrically weighted connections. Learning is done with 1-step

Contrastive Divergence. The program assumes that the following variables are set externally:

* maxepoch  -- maximum number of epochs
* numhid    -- number of hidden units
* batchdata -- the data that is divided into batches (numcases numdims numbatches)
* restart   -- set to 1 if learning starts from beginning

**Code**

//Importing Java Packages

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.FileReader;

import java.io.IOException;

import java.util.HashSet;

import java.util.Random;

import java.util.Set;

import java.util.StringTokenizer;

import java.io.\*;

import java.util.\*;

//Importing Hadoop Packages

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.filecache.DistributedCache;

import org.apache.hadoop.conf.\*;

import org.apache.hadoop.io.\*;

import org.apache.hadoop.mapred.\*;

import org.apache.hadoop.util.\*;

public class RBMMapper extends MapReduceBase implements Mapper<LongWritable, Text, IntWritable, DoubleWritable>

{

    /\*

     \* This is the mapper of RBM training MapReduce program

     \*

     \* Note that the format of intermediate output is <IntWritable, DoubleWritable>,

     \* because the key is the number of weight (an integer), and the value is the weight's value (double)

     \*/

    // this is the intermediate output

    private DoubleWritable weightValue = new DoubleWritable();

    private IntWritable weightPos = new IntWritable();

    // These are the variables and parameters used in algorithm

    private double hidbiases,epsilonvb,espilonhb,weightcost,initialmomentum,finalmomentum;

    private int numhid,numdims,numbatches,maxepoch;

    private Matrix hidbiases,visbiases,poshidprobs,neghidprobs,posprods,negprods,vishidinc,hdibiasinc,visbiasinc,poshidstates,data,vishid;

    private String weightline,inputData,inputNumdims,inputNumhid;

    public void configure(JobConf conf) {

        /\*

         It reads all the configurations and distributed cache from outside.

         \*/

        // Read number of nodes in input layer and output layer from configuration

        inputNumdims = conf.get("numdims");

        inputNumhid  = conf.get("numhid");

        // Read the weights from distributed cache

        Path[] pathwaysFiles = new Path[0];

        try {

               pathwaysFiles = DistributedCache.getLocalCacheFiles(conf);

               for (Path path: pathwaysFiles) {

                   /\*

                    Reads all the distributed cache files

                    The driver program ensures that there is only one distributed cache file

                    \*/

                   BufferedReader fis = new BufferedReader(new FileReader(path.toString()));

                   weightline = fis.readLine();

              }

         } catch (Exception e) {

                 e.printStackTrace();

         }

    }

    private void  initialize(){

        /\*

         It parses the input strings into parameters, and initialize parameters for algorithm.

         \*/

        epsilonW = 0.1;

        epsilonvb = 0.1;

        espilonhb = 0.1;

        weightcost = 0.000;

        initialmomentum = 0.5;

        finalmomentum = 0.9;

        // Parse the number of nodes in input layer and output layer

        numhid = Integer.parseInt(inputNumhid);

        numdims = Integer.parseInt(inputNumdims);

        // Parse the weights

        String [] tokens = inputData.split("\t");

        String [] DataString;

        if (tokens.length == 1)

        // This case happens when first time read the data

        {

            DataString = tokens[0].trim().split("\\s+");

        }

        else

        // Else, the input line is output by previous layer

        {

            DataString = tokens[1].trim().split("\\s+");

        }

        double [] DataVector = new double[numdims];

        double [] VishidMatrix = new double[numdims \* numhid];

        int count = 0;

        String line;

        String [] tempst;

        line = weightline;

        tempst = line.trim().split(" ");

        count = tempst.length;

        if (numdims != DataString.length || numdims \* numhid != count)

        {

            /\*

             Check if the input data match the expectation

             \*/

            throw new IllegalArgumentException("Input data and value do not match!");

        }

        for(int i = 0; i < numdims; i++)

        {

            DataVector[i] = (double)(Integer.parseInt(DataString[i]))/255.0;

        }

        for(int i = 0; i < count; i++)

        {

            VishidMatrix[i] = Double.parseDouble(tempst[i]);

        }

        // initialize the variables

        // Most of them are matrix.

        this.data = new Matrix(DataVector,1);

        this.vishid = new Matrix(VishidMatrix, numdims);

        // We create matrices for Hidden Bias,Visible Bias, positive phase samples, negative phase samples

        hidbiases = new Matrix(1,numhid);

        visbiases = new Matrix(1,numdims);

        poshidprobs = new Matrix(1,numhid);

        neghidprobs = new Matrix(1,numhid);

        posprods = new Matrix(numdims,numhid);

        negprods = new Matrix(numdims,numhid);

        vishidinc = new Matrix(numdims,numhid);

        hdibiasinc = new Matrix(1,numhid);

        visbiasinc = new Matrix(1,numhid);

        poshidstates = new Matrix(1,numhid);

    }

    private void getposphase()

    {

        /\*

        It does the positive phase of unsupervised RBM training algorithm

         \*/

        //Start calculate the positive phase

        //Calculate the cured value of h0

        poshidprobs = data.times(vishid);

        //(1 \* numdims) \* (numdims \* numhid)

        poshidprobs.plusEquals(hidbiases);

        //data\*vishid + hidbiases

        double [] [] product\_tmp2 = poshidprobs.getArray();

        int i2 = 0;

        while( i2 < numhid)

        {

                product\_tmp2[0][i2] = 1/(1 + Math.exp(-product\_tmp2[0][i2]));

                i2++;

        }

        posprods = data.transpose().times(poshidprobs);

        //(numdims \* 1) \* (1 \* numhid)

        //End of the positive phase calculation, find the binary presentation of h0

        int i3 =0;

        double [] [] tmp1 = poshidprobs.getArray();

        double [] [] tmp2 = new double [1][numhid];

        Random randomgenerator = new Random();

        while (i3 < numhid)

        {

            /\*

             A sampling according to possiblity given by poshidprobs

             \*/

                if (tmp1[0][i3] > randomgenerator.nextDouble())

                        tmp2[0][i3] = 1;

                else tmp2[0][i3] = 0;

                i3++;

        }

        // poshidstates is a binary sampling according to possiblity given by poshidprobs

        poshidstates = new Matrix(tmp2);

    }

    private void getnegphase()

    {

        /\*

        It does the negative phase of unsupervised RBM training algorithm

         \*/

        /\*start calculate the negative phase

        1. Calculate the curved value of v1,h1

        2. Find the vector of v1

        \*/

        Matrix negdata = poshidstates.times(vishid.transpose());

        //(1 \* numhid) \* (numhid \* numdims) = (1 \* numdims)

        negdata.plusEquals(visbiases);

        //poshidstates\*vishid' + visbiases

        double [] [] tmp1 = negdata.getArray();

        int i1 = 0;

        while( i1 < numdims)

        {

                tmp1[0][i1] = 1/(1 + Math.exp(-tmp1[0][i1]));

                i1++;

        }

        //Find the vector of h1

        neghidprobs = negdata.times(vishid);

        //(1 \* numdims) \* (numdims \* numhid) = (1 \* numhid)

        neghidprobs.plusEquals(hidbiases);

        double [] [] tmp2 = neghidprobs.getArray();

        int i2 = 0;

        while( i2 < numhid)

        {

            tmp2[0][i2] = 1/(1 + Math.exp(-tmp2[0][i2]));

            i2++;

        }

        negprods = negdata.transpose().times(neghidprobs);

        //(numdims \* 1) \*(1 \* numhid) = (numdims \* numhid)

    }

    // Update the weights and biases

    // This serves as a reducer

    private void update()

    {

        /\*

         \* It computes the update of weights using previous results and parameters

         \*/

        double momentum;

        Matrix temp1 = posprods.minus(negprods);

        Matrix temp2 = vishid.times(weightcost);

        temp1.minusEquals(temp2);

        temp1.timesEquals(hidbiases);

        // the final updates of weights are written in vishidinc

        vishidinc.plusEquals(temp1);

    }

    public void map(LongWritable key, Text value, OutputCollector<IntWritable, DoubleWritable> output, Reporter reporter) throws IOException

    {

        /\*

         It implements the mapper. It outputs the numbers of weight and updated weights.

         Note that the format of intermediate output is <IntWritable, DoubleWritable>,

         because the key is the number of weight (an integer), and the value is the weight's value (double)

         \*/

        inputData = value.toString();

        // Go through the process

        initialize();

        getposphase();

        getnegphase();

        update();

        // Output the intermediate data

        // The <key, value> pairs are <weightID, weightUpdate>

        double [][] vishidinc\_array = vishidinc.getArray();

        for(int i = 0; i < numdims; i++ )

        {

            for(int j=0; j < numhid; j++ )

            {

                weightPos.set(i \* numhid + j);

                weightValue.set(vishidinc\_array[i][j]);

                output.collect(weightPos, weightValue);

            }

        }

    }

 }

**RBM Reducer**

**Class Description:**

This is the reducer of restricted Boltzmann machine (RBM) training part in deep learning MapReduce program. The task of this MapReduce is to train the rbm between two layers in unsupervised way and output the updated weight. Each reducer collects all the weight updates from one single weight ID. It adds the updates from the same weight up and write to the final output.

**Code**

//Importing Java Packages

import java.io.IOException;

import java.util.Iterator;

//Importing Hadoop Packages

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.filecache.DistributedCache;

import org.apache.hadoop.conf.\*;

import org.apache.hadoop.io.\*;

import org.apache.hadoop.mapred.\*;

import org.apache.hadoop.util.\*;

public class RBMReducer extends MapReduceBase implements Reducer<IntWritable, DoubleWritable, IntWritable, DoubleWritable>

{

    /\*

    Note that the format of intermediate data it taking is <IntWritable, DoubleWritable>, as mapper output.

    The format of final output is also <IntWritable, DoubleWritable>,because the key is the number of weight (an integer), and the value is the update of weight's value (double)

     \*/

    public void reduce(IntWritable key, Iterator<DoubleWritable> values, OutputCollector<IntWritable, DoubleWritable> output, Reporter reporter) throws IOException

    {

       double sum = 0;

       while (values.hasNext())

       {

           //Calculate the sum of all the updates

           sum += values.next().get();

       }

       // Output the sum

       output.collect(key, new DoubleWritable(sum));

    }

}