The Journal of Supercomputing (2021) 77:6197–6213
https://doi.org/10.1007/s11227-020-03509-2

Parallelization of the self-organized maps algorithm for federated learning on distributed sources

Ivan Kholod[1] · **Andrey Rukavitsyn**[1] · **Alexey Paznikov**[1] · **Sergei Gorlatch**[2]

Accepted: 2 November 2020 / Published online: 25 November 2020

Abstract
This paper describes a formally based approach for parallelizing the Kohonen algorithm used for the federated learning process in a special kind of neural networks—Self-Organizing Maps. Our approach enables executing the parallel algorithm version on the distributed data sources, taking into account the kind of data distribution on the nodes. Compared to the traditional approaches, we distinguish two kinds of data distributions—horizontal and vertical: for both, our suggested approach avoids gathering data in a single storage, but rather moves computations nearer to the data source nodes. This reduces the execution time of the algorithm, the network traffic, and the risk of an unauthorized access to the data during their transmission. Our experimental evaluation demonstrates the advantages of the approach.

Keywords Self-Organizing Maps (SOM) · Neural networks · Distributed data · Federated learning · Kohonen algorithm

1 Introduction

Many companies currently organize their work in a data-driven manner, i.e., they employ data from various sources to optimize their business. This brings the necessity to build platforms for data processing, which include machine learning,

✉ Ivan Kholod
iiholod@mail.ru

Andrey Rukavitsyn
rkvtsn@gmail.com

Alexey Paznikov
ashxz@mail.ru

Sergei Gorlatch
gorlatch@uni-muenster.de

[1] Saint Petersburg Electrotechnical University "LETI", Saint Petersburg, Russia

[2] University of Muenster, Muenster, Germany

enterprise data warehouses, data clouds, etc. A typical architecture of data processing platforms includes, as in [1]:

- data sources, which contain domain-oriented data;
- platform, which gathers and processes all data;
- consumers, which solve different business data-driven tasks.

Figure 1 shows an example platform that processes data from distributed sources. There are two possible kinds of data distributions used in the business domains:

- horizontal distribution shown in Fig. 1a: data sources are related to the same business domain and contain the data about different facts about this domain;
- vertical distribution as shown in Fig. 1b: data sources are related to different business domains and contain data about the same facts about those domains.

A data platform in Fig. 1 receives data from data sources. Its goals are:

- receiving data from the data sources from same or different domains;
- enriching and transforming the source data into trustworthy data that allow for addressing the needs of diverse consumers;
- providing services (including data analysis based on the data sets) to the broad community of consumers.

This current organization of data processing platforms has some weaknesses; in particular, it leads to an increase in total processing time, intensive network traffic, and a risk of unauthorized access to the data. At the same time, the problem of data security and users' data privacy becomes increasingly critical. Therefore, governmental structures establish regulations to protect users' data, for example GDPR in European Union [2] and CCPA [3] in the USA.

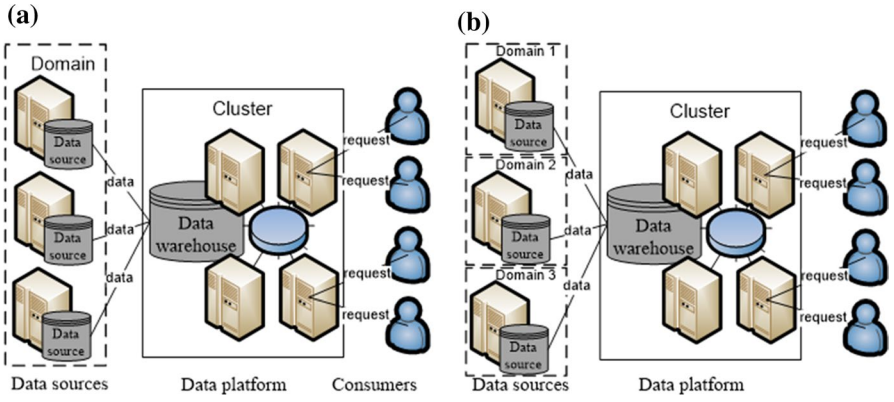

Fig. 1 Data processing platforms with **a** horizontally and **b** vertically distributed data

As a solution to these problems, Google proposed the concept of federated learning [4]: the main idea is to build machine learning models based on data sets distributed across multiple data sources without exchanging data among them. Federated learning systems are typically categorized in horizontal and vertical, depending on how data are distributed over sources [5], as explained in Fig. 1.

One of the important business tasks in the area of data-driven management is market segmentation, which is usually performed using clustering methods. In the area of data mining, these methods refer to the so-called unsupervised learning; they are often used for preliminary data analysis. Clustering allocates groups (clusters) among the objects of the analyzed data.

Recent years have witnessed a significant growth in popularity of neural networks which are also applied to solve the task of clustering. A useful and popular class of neural networks are Self-Organized Maps (SOM) proposed by Kohonen [6]. The advantages of the SOM are as follows: (1) detection of the clusters of arbitrary shapes with different sizes; (2) clustering without initial knowledge about the data; (3) iterative processing of large amounts of streaming data; (4) data dimension reduction; and (5) visualization of the output.

The goal of this paper is to overcome the current drawbacks of the data platforms. Our approach is clustering the data at data sources: we achieve this by moving computations to the data sources. For this, we decompose the SOM algorithm to perform its major parts locally on the data sources without transferring data in the network. This helps to reduce the execution time of applications and the network traffic; we achieve this by enhancing our earlier general approach for parallelizing data mining algorithms on distributed data sources [7].

2 The SOM algorithm

The SOM algorithm for building neural networks was proposed by Kohonen [6]. The network is trained at a set of input vectors $x(t)$ that contain attributes (such as place, count, and price) of an event at time t, for example purchases, provided services, and others.

A set of input vectors is usually represented in the form of a 2D array (data matrix), e.g., for discrete time indices $t = 1, \ldots, z$ and p attributes [8]:

$$d = \begin{pmatrix} x_1(t_1) & \ldots & x_k(t_1) & \ldots & x_p(t_1) \\ \ldots & & \ldots & & \ldots \\ x_1(t_j) & \ldots & x_k(t_k) & \ldots & x_p(t_j) \\ \ldots & & \ldots & & \ldots \\ x_1(t_z) & \ldots & x_k(t_z) & \ldots & x_p(t_z) \end{pmatrix} \quad (1)$$

where $x_k(t_j)$ is the value of the k^{th} attribute for the event at time t_j, represented by input vector $x(t_j)$. The input vectors d available at time t_z for training are called *epoch*.

The SOM compares a set of neurons:

$$m(t) = [n_1(t), \ldots., n_u(t)],$$

where u is a number of the neurons, which must be a priori determined by an analyst. Each neuron $n_i(t)$ is determined at time t by a weight vector having the same dimensionality p as input vectors:

$$n_i(t) = [\omega 1, \ldots, \omega p].$$

The SOM algorithm for each input vector $x(t)$ finds the neuron $n_w(t)$—so-called neuron winner—with the weight most similar to it:

$$n_w(t) = argmin_{i=1}^{u} \rho(x(t), n_i(t)).$$

We use the Euclidean distance to find the degree of similarity between two vectors (input vector and weight vector):

$$\rho(x(t), n_i(t)) = \sqrt{\sum_{k=1}^{p} (x_k(t) - n_i(t).\omega_k)^2} \tag{2}$$

Next, the algorithm corrects the weights of the winners and the winners' neighbors:

$$n_i(t+1) = n_i(t) + \alpha \cdot \eta(i, w) \cdot \rho(x(t), n_w(t)), \ \textit{where} \tag{3}$$

- α—the bargaining ratio of the weight update, which is mainly influenced by the learn rate (commonly around 0.1);
- enriching and transforming the source data into trustworthy data that allow for addressing the needs of diverse consumers;
- $\eta(i, w)$—the neighborhood function which is an exponentially decreasing function that reduces the influence of input vector x on neurons (for example, Gaussian neighborhood with a monotonously decreasing function):

$$\eta(i, w) = \exp(\rho(n_w(t), n_i(t))^2 / \delta(t)^2), \text{where}$$

- $\delta(t)$ represents the function which reduces the value between two values $\delta_{\max}$ and $\delta_{\min}$, to control the size of the neighborhood, thus influencing a given cell on the SOM:

$$\delta(t) = \delta_{\max} \cdot (\delta_{\min} / \delta_{\max})^{t/z}$$

Figure 2 shows the sequential pseudocode of the SOM algorithm.

```
1.  for i=1 to n    // loop for each neuron
2.     for k=1 to p // loop for each weight
3.        n_i.ω_k = random() // initialization of neuron's weights
4.     endfor
5.  endfor
6.  for j=1 to z // loop for each vector of data set
7.     for i=1 to u // loop for each neuron
          ρ_i = 0;
8.        for k=1 to p // loop for each attribute
9.           ρ_i = ρ_i +(x_k(t_j) - n_i.ω_k)^2 //calculate sum of Euclidean distance (2)
10.       endfor
11.       ρ_i = sqrt(ρ_i) // calculate Euclidean distance (2)
12.    endfor
13.    w = 1
14.    for i=2 to u // find neuron winner (1)
15.       if ρ_i > ρ_w then w = i
16.    endfor
17.    for i=1 to u // loop for each neuron
18.       for k=1 to p // loop for each attribute
19.          n_i.ω_k = n_i.ω_k +α·η(i,w)· ρ_w; // correction of weights (3)
20.       endfor
21.    endfor
```

Fig. 2 The SOM algorithm: sequential pseudocode

3 Related work

In traditional data processing platforms, data from distributed sources are gathered at the central point (e.g., in a single data warehouse, or computing node) for analysis, which is usually implemented based on the MapReduce programming model [9].

The MapReduce model employs the abstraction inspired by the *map* and *reduce* primitives, originated from functional programming and actively exploited in the skeleton-based approach for parallel computing [10]. Parts of MapReduce can run in parallel on different nodes of the network, thereby ensuring a high performance of data mining.

The adaptation of the SOM algorithm to the MapReduce programming model uses its batch version [11], because the original version of SOM (also called online SOM algorithm) is time dependent: each time step directly corresponds to the presentation of an input vector $x(t)$. The batch SOM algorithm corrects the neuron weights for whole epoch, thus removing the time dependency from the input:

$$n_i = \frac{\sum_{j=1}^{z}(\eta(i,w,j)\cdot x_j)}{\sum_{j=1}^{z}\eta(i,w,j)} \tag{4}$$

The analysis of the batch SOM algorithm on various data sets demonstrated a good quality of clusterization [11] when applied in practice.

Paper [12] shows that the batch SOM algorithm has the following advantages but also drawbacks:

- advantages: simplicity of the computations, high performance, no adaptation parameters that have to be tuned, deterministic and reproducible results;
- too unbalanced clusters, strong dependence of the initialization.

There are several variants of adapting the batch SOM algorithm to the MapReduce programming model [13–15]. They differ in distributing computations of the winning neurons, $\eta(i, w, j) \cdot x_j$ and $\eta(i, w, j)$ from (4) by the *map* and *reduce* functions.

Paper [13] proposes the following mapping of the SOM algorithm's parts to the MapReduce model:

- The *map* function computes the winning neuron, $\eta(i, w, j) \cdot x_j$ and $\eta(i, w, j)$;
- first *reduce* function accumulates the denominator from (4);
- second *reduce* function accumulates the numerator from (4) and corrects the weights of neurons.

Paper [14] suggests the mapping of the SOM algorithm's parts to two *map* functions and a single *reduce* function:

- first *map* function computes $\eta(i, w, j) \cdot x_j$ from (4);
- second map computes $\eta(i, w, j)$ from (4);
- *reduce* function corrects the weights of neurons.

In the approach of [15], the *map* function computes the winning neuron for each input vector, while the *reduce* function calculates the neighborhoods for each winning neuron and the weight updates needed for them.

In all these existing approaches, data are communicated from distributed sources to the single compute node of the network where data mining takes place. In the existing map–reduce implementations, deploying the functions of the map–reduce schema on data sources is often impossible, because the sources are not part of the whole data processing platform and have independent management.

The traditional implementation scheme in Fig. 3a implies the following problems:

- data communication may take quite a long time which may be disadvantageous for the target performance;
- the network traffic may be very intensive, which limits the use of modern limited-capacity communication channels, such as satellites and wireless;
- confidential data from the data sources are communicated via public channels, which increases the risk of unauthorized access to the data;
- since the volumes of data are large, their collection at a single location requires special protection of data security and reliability.

Paper [16] describes an approach for data clustering using SOM. This approach uses the same SOM algorithm at data sources and compute node, but for different

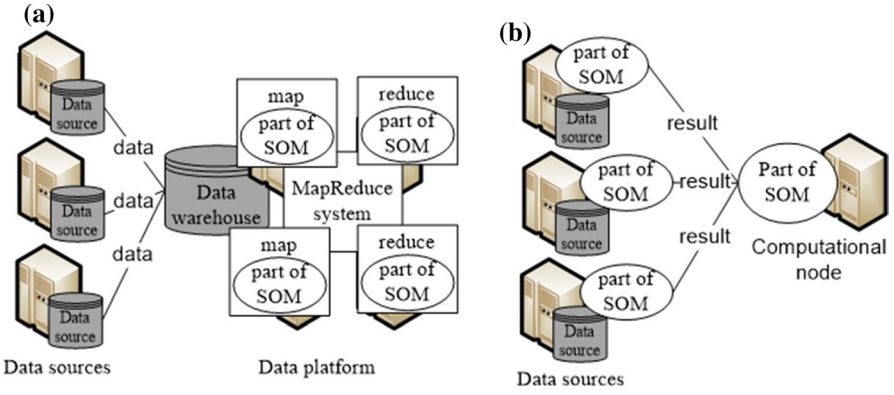

Fig. 3 Variants of SOM algorithm applied to distributed data: **a** traditional MapReduce; **b** our suggested approach

purposes. At a data source, the algorithm is applied locally to obtain a vector of data indexes; these index vectors are then transferred to a compute node. At the compute node, the algorithm is applied again to obtain a SOM base on gathering the index vectors. This method is applied to vertically distributed data only.

The previous work does not address the problem of implementing the SOM algorithm, taking into account the different types of data distribution without transferring all data to the single data warehouse where the compute note is located.

Currently, several open-source federated learning systems are under development [17]: TensorFlow Federated by Google [18], PySyft by open community OpenMined [19], Federated AI Technology Enabler by Webank's AI Department [20], and PaddleFL by Baidu [21]. They use different neural networks for federated learning, but our work is the first to consider parallelizing SOM for federated learning.

In our envisaged approach, as shown in Fig. 3b, we aim at improving the distributed data clustering and avoiding the disadvantages mentioned above. Our idea is to perform parts of a SOM algorithm at data sources, while intermediate results are sent to the central compute node. Our approach for distributed clustering also optimizes the structure of the SOM algorithm according to the type of data distribution: horizontal or vertical.

The trade-off is that our approach requires additional computations on the data sources. However, the amount of these computations is relatively low, so they can be performed using even low-power devices, such as mobile phones and IoT devices.

A major challenge for parallelizing federal learning is the complexity of partitioning the whole amount of work between the data sources on the one hand and the computing node on the other hand. Our general approach suggested in [7] addresses the issues of distributing data and correctly combining the results obtained on different data sources. In this paper, we show how our general approach from [7] can be applied to the SOM algorithm. We develop and estimate two versions of the SOM algorithm—batch and online—for horizontal and vertical data distributions.

4 Decomposition of SOM Algorithm

To deal with the SOM algorithm, we apply our general approach proposed in [7], which is formally based, and covers a broad class of data mining algorithms. We use capital letters for types and lowercase letters for variables and functions.

In our approach, a data mining algorithm is represented as a function taking a data set $d \in D$ as input and creating a mining model $m \in M$ as output:

$$dma : D \rightarrow M \tag{5}$$

We represent a mining model $m \in M$ as an array of elements $e_i, i = 0, \ldots, v$:

$$m = [e_0, e_1, \ldots, e_v]$$

In the general approach, mining model's elements describe knowledge (classification rules, clusters and other) extracted by a data mining algorithm from a data set. In case of the SOM algorithm, a model m is a SOM, represented as an array of elements:

- e_0 is index of the winner neuron (variable w in Fig. 2);
- $e_1, \ldots, e_u$ are Euclidean distances between current input vectors $x(t)$ and neurons $n_1, \ldots, n_u$ (variables $\rho_1 \ldots \rho_u$ in Fig. 2);
- $e_{u+1}, \ldots e_{2u}$ are neurons $n_1, \ldots, n_u$ (variables $n_1, \ldots, n_u$ in Fig. 2).

In the general case, we represent a data mining algorithm formally as the following sequential composition of functions:

$$dma = f_n \circ f_{n-1} \circ \ldots \circ f_1 \circ f_0 \tag{6}$$

where $\circ$ is the associative composition operator applied from right to left.

In representation (6), function $f_0 : D \rightarrow M$ takes a data set $d \in D$ as an argument and returns a mining model $m_0 \in M$. For the SOM algorithm, function f_0 initializes neuron's weights (lines 1–5 in Fig. 2).

The next functions in the composition, $f_t, t = 1, \ldots, n$ take the mining model $m_{t-1} \in M$ which is created by function f_{t-1} and return the updated mining model $m_t \in M$:

$$f_t : M \rightarrow M \tag{7}$$

Functions $f_t, t = 1, \ldots, n$ of type (5) are called Functional Mining Blocks (FMB). For the SOM algorithm, the FMBs are as follows:

- f_5 calculates Euclidean distance (line 11 in Fig. 2);
- f_6 initializes the index of the winner (line 13 in Fig. 2);
- f_8 selects the winner (lines 14–15 in Fig. 2);
- f_{11} corrects the weights of neurons (line 19 in Fig. 2).

Some steps of a data mining algorithm use data set d to change the mining model, i.e., they take the data set d as additional argument:

$$fd_t : D \to M \to M \tag{8}$$

To use these functions in the composition (4), we exploit partial function application with the fixed first argument: $f_t = fd_t\ d$. For SOM algorithm, FMB fd_4 uses data set d to calculate the sum of Euclidean distance (line 9 in Fig. 2);

In the general case, we invoke function fd_t in a loop, in order to apply it to all input vectors. We use loops over attributes and input vectors of the data set to process the data iteratively. We use an asterisk to denote the input vector (e.g., $d[j, *]$ refers to the j^{th} input vector) or the attribute (e.g., $d[*, k]$ refers to the k^{th} attribute) in a data set:

– *loopc* applies fd_t to the attributes of $d \in D$, from index i_s till index i_e:

$$\begin{aligned} &loopc : I \to I \to (M \to M) \to D \to M \to M \\ &loopc\ i_s\ i_e\ fd_t\ d\ m = (fd_t\ d[*, i_e]) \circ \ldots \circ (fd_t\ d[*, i_s])\ \mu \end{aligned} \tag{9}$$

– *loopr* applies fd_t to the input vectors of $d \in D$, from index i_s till index i_e:

$$\begin{aligned} &loopr : I \to I \to (M \to M) \to D \to M \to M \\ &loopr\ i_s\ i_e\ fd_t\ d\ m = (fd_t\ d[i_e, *]) \circ \ldots \circ (fd_t\ d[i_s, *])\ \mu \end{aligned} \tag{10}$$

The first four arguments are fixed for using loops in the composition (6).

To apply function f_t to every neuron of the SOM m from index i_s till index i_e, we invoke it in the following loop: $loopn\ i_s\ i_e\ f_t\ m = (f_t\ m[u + i_e]) \circ \ldots \circ (f_t\ m[u + i_s])$, where

$$loopn : I \to I \to (M \to M) \to M \to M \tag{11}$$

The first four arguments are fixed to use *loopn* in the composition (6). Therefore, all functions of the SOM algorithm are for each line as follows:

– f_1 is the loop for the input vectors (lines 6–22 in Fig. 2):

$$f1 = loopr\ 1\ z\ (fd_9 \circ f_7 \circ f_6 \circ fd_2);$$

– fd_2 is the loop for neurons (lines 7–12 in Fig. 2):

$$fd_2 = loopn\ 1\ n\ (f5 \circ fd_3);$$

– fd_3 is the loop for the attributes (lines 8–10 in Fig. 2):

$$fd_3 = loopc\ 1\ p\ fd_4;$$

– f_7 is the loop for neurons (lines 14–16 in Fig. 2):

$$f_7 = loopn\ 2\ n\ f_8;$$

– fd_9 is the loop for neurons (lines 17–21 in Fig. 2):

$$f_9 = loopn\ 1\ n\ f_{10};$$

– fd_{10} is the loop for the weights of neurons (lines 18–20 in Fig. 2):

$$f_{10} = loopc\ 1\ p\ f_{11};$$

The following composition of the functions represents the SOM algorithm:

$$\begin{aligned} som &= fd_1 \circ f_0 = (loopr\ 1\ z(f_9 \circ f_7 \circ f_6 \circ fd_2)\ d) \circ f_0 \\ &= (loopr\ 1\ z\ (loopn\ 1\ nf_{10}) \circ (loopn\ 2\ n\ f_8) \circ f_6 \circ (loopn\ 1\ n\ (f_5 \circ fd_3))\ d) \circ f_0 \\ &= (loopr\ 1\ z\ (loopn\ 1\ n\ (loopn\ 1\ p\ f_{11})) \circ (loopn\ 2\ n\ f_8) \circ f_6 \\ &\quad \circ (loopn\ 1\ n\ (f_5 \circ (loopc\ 1\ p\ fd_4)))\ d) \circ f_0 \end{aligned} \tag{12}$$

5 SOM algorithm for distributed data

5.1 Parallelization for distributed data

Figure 4 represents a distributed storage that splits data set d among s sources [22]:

$$d = d_1 \cup \cdots \cup d_s,$$

where data subset d_h is located at the data source number h.

If data are located on different sources, then FMBs of type (8) can be executed on them in parallel, which corresponds to parallel execution on a distributed memory.

In our general approach proposed in paper [7], we introduce function *paralleled* that specifies parallel execution of FMBs on a system with distributed memory:

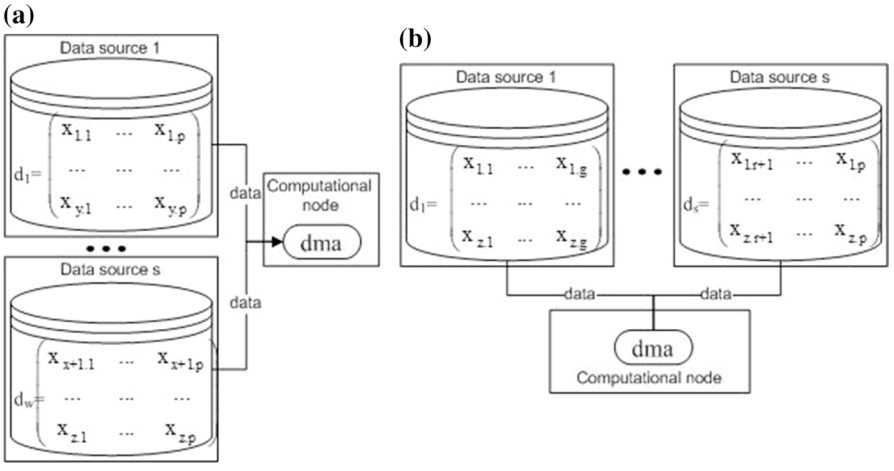

Fig. 4 Traditional distributions of data matrix: **a** horizontal; **b** vertical

$$\begin{aligned} &paralleled : [(M \to M)] \to M \to M \\ &paralleled\ [f_r, \ldots, f_s]\ m = join\ m\ (forkd\ [f_r, \ldots, f_s]\ m), \end{aligned} \tag{13}$$

where function *forkd* can invoke FMBs in parallel on distributed memory:

$$\begin{aligned} &forkd : [(M \to M)] \to M \to [M] \\ &forkd\ [f_r, \ldots, f_s]\ m = [f_r\ copy\ m, \ldots, f_s\ copy\ m], \end{aligned} \tag{14}$$

and function *copy* yields copies of the initial mining model in disjunctive areas of distributed memory for processing by FMBs in parallel:

$$\begin{aligned} &copy : M \to M \\ &copy\ m = [m[0], m[1], \ldots, m[v]]. \end{aligned} \tag{15}$$

Function *join* combines the mining models built by FMBs for disjunctive areas of distributed memory:

$$\begin{aligned} &join : M \to [M] \to M \\ &copym[m_r, \ldots, m_s] = [m'[0], \ldots, m'[g], \ldots, m'[v]], \\ &where\ m'[g] = \begin{cases} m[g]\ if\ m_i[g] = m[g]\ for\ all\ i = r..s \\ union\ m[g]\ [m_r[g], \ldots, m_s[g]]\ otherwise \end{cases} \end{aligned} \tag{16}$$

where the *union* function takes elements of different mining models with the same index and merges them to a single mining model's element:

$$union : E \to [E] \to E. \tag{17}$$

Function *union* is implemented depending on the structure of the elements.

5.2 Parallelizing SOM algorithm for different data distributions

Assuming that data sources are distributed, function fd_2 from (12) can be computed on the source nodes by transforming the sequential form of the SOM algorithm into a parallel form for a particular distribution of data.

There are two variants [23] of transforming (13):

1. using synchronization for every input vector by inserting function *paralleled* into outer loop *loopr* for function fd_2 (Fig. 5a):

$$sompar1 = (loopr\ 1\ z(f_9 \circ f_7 \circ f_6 \circ \mathbf{paralleled}[fd_2])\ d) \circ f_0 \tag{18}$$

2. using synchronization for the whole epoch when the *paralleled* function is applied to the outer loop *loopr* (Fig. 5b):

$$sompar2 = (\mathbf{paralleled}[loopr\ 1\ z(f_9 \circ f_7 \circ f_6 \circ fd_2)]\ d) \circ f_0 \tag{19}$$

Figure 5a shows the variant using synchronization for every input vector. Since there are interactions between the data sources and the compute node for each input vector

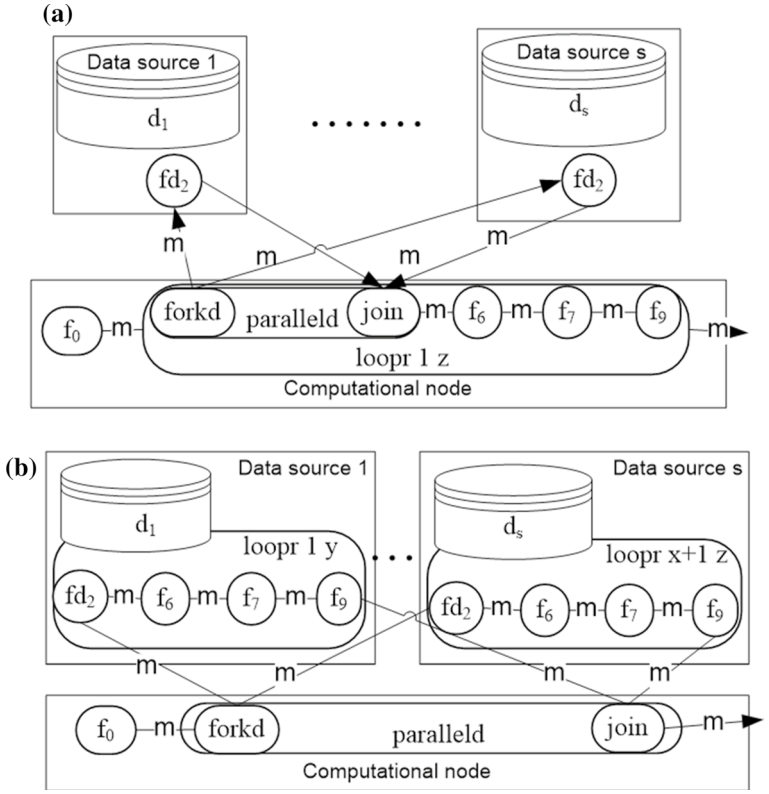

Fig. 5 Distributed execution of the SOM algorithm for distributed data: **a** with synchronization for each input vector; **b** with synchronization for whole epoch

(sending intermediate results to the computing node and sending back the generalized model), this approach is slow and generates high network traffic. This variant corresponds to the online version of the SOM algorithm.

Figure 5b depicts the variant using synchronization for each epoch. In this case, the network traffic is minimal (because interaction happens only once for an epoch after finishing the epoch analysis on the data source). Combining weights is performed after the analysis of the epochs on all data sources is finished. This variant corresponds to the batch version of the SOM algorithm: it updates the neuron weights of SOM for the whole epoch, rather than for each input vector.

For the horizontal distribution of data and synchronization for every input vector, function *join* cannot combine models created by FMB fd_2 at different data sources. The FMB fd_2 calculates the Euclidean distances (elements $e_1, \ldots, e_u$ in model m) between the current input vector at data source and each neuron. Function *join* receives these distances from different data sources and has no criteria to select correct distances for finding the winner neuron by FMB f_6. Hence, synchronization for every input vector cannot be implemented with horizontally distributed data.

If synchronization is performed for the whole epoch, we invoke function *union* (17) after processing all input vectors at data sources. It calculates the weights of the neurons with the same indices from different data sources:

$$unionm[u+1] = \alpha(m^1[u+1].\omega, \ldots, m^s[u+1].\omega_1),$$

where m_r, $r = 1..s$ is a SOM built at r^{th} data source; function α calculates the total weight, which, for example, is computed by averaging:

$$\alpha(m^1[i].\omega, \ldots, m^s[i].\omega_1) = (m^1[i].\omega + \ldots + m^s[i].\omega_1)/s$$

The variant with synchronization for each epoch corresponds to the batch version of the SOM algorithm, as function union updates the neuron weights after processing all input vectors on the data sources. Therefore, the result of the batch version of SOM may differ from the online SOM version in which the update of weights is performed after processing each vector. However, this difference does not mean a decrease in accuracy, because even in the online SOM version different results may be obtained for different arrival orders of the input vectors. It is rather important that the obtained clusters are similar, which is demonstrated in our previous work [23], as well as by other authors [11–16].

If data are distributed vertically and synchronization is done for each input vector, then function fd_4 returns the Euclidean distance between input vector $x(t_j)$ and neurons $n_i, i = 1..u$ in parts of the subset $d_h, p = 1..s$:

$$m^1[i] = \sqrt{\sum_{k=1}^{g}(m[u+i].\omega_k - x_k(t_j))^2}; \ldots;$$

$$m^s[i] = \sqrt{\sum_{k=r+1}^{p}(m[u+i].\omega_k - x_k(t_j))^2}$$

The total distance can be calculated by function union by summarizing:

$$union\ m[i] = m^1[i] + \ldots + m^s[i].$$

Though for Euclidean distance, the sum of the parts is not equal to the total distance:

$$m^1[i] + \ldots + m^s[i] \neq \sqrt{\sum_{k=1}^{p}(m[i].\omega_k - x_k(t_j))^2}$$

however, the inequality relation is still preserved for this function *union*:

$$if\ m[i] < m[i+1]\ then\ (m^1[i] + \ldots + m^s[i]) < (m^1[i+1] + \ldots + m^s[i+1]).$$

Therefore, we can select the winner neuron correctly. This results in the parallel implementation of the SOM algorithm for vertically distributed data with synchronization for each input vector. Another possibility is to perform synchronization for a whole epoch. SOM contains neurons which have weights that are partially calculated

at each data source for the corresponding attributes. For example, for neuron n_i the following weights are calculated at different data sources:

$$m^1[u+i].\omega_1 \neq 0, \ldots, m^1[u+i].\omega_g \neq 0, m^1[u+i].\omega_{g+1} = 0, \ldots, m^1[u+i], \omega_p = 0,$$

$$\ldots$$

$$m^s[u+i].\omega_1 = 0, \ldots, m^s[u+i].\omega_r \neq 0, m^s[u+i].\omega_{r+1} \neq 0, \ldots, m^s[u+i], \omega_p \neq 0.$$

Therefore, function *union* for each neuron must combine weights as follows:

$$union\ m[u+i] = [m^1[u+i].\omega_1, \ldots, m^1[u+i].\omega_g, \ldots, m^s[u+i].\omega_{r+1}, \ldots, m^s[u+i].\omega_p].$$

Thus, we obtain a parallel implementation of the SOM algorithm with synchronization for whole epoch on vertically distributed data.

The results of comparing different variants of synchronization for SOM are summarized in Table 1.

In summary, by applying our general approach to the SOM algorithm, we develop two versions of federated learning on data sources—the online and batch version. Each version has their advantages and disadvantages [12], so we analyzed them both for the two practical data distributions—horizontal and vertical.

Our analysis shows that, for the horizontal data distribution, only the batch SOM algorithm can be used, because for the online version we cannot choose a common winner neuron for input vectors on different data sources. For the vertical data distribution, both methods are applicable. However, formal analysis shows and our experiments in the next section confirm that the online version implies a higher network traffic and, therefore, longer execution time.

6 Experimental evaluation

In this section, we describe the distributed implementation of the SOM algorithm [24] using the Java-based library XelopesFL [25]. We use this implementation for the experimental evaluation described in the following. All experiments described in this section can be reproduced using our library version in [24, 25].

In our experiments, the data source nodes are as follows: CPU Intel Xeon (4 physical cores), 2.90 GHz, 4 Gb. The compute node contains: CPU Intel Xeon (12 physical cores), 2.90 GHz, 4 Gb. The data sources are connected to the computational node by a local network with bandwidth 1 Gbps.

Table 1 Using different ways of synchronization for different types of distribution

Data distribution	Version of SOM algorithm	Applicable	Run time	Traffic
Horizontal	Online (sompar1$_H$)	No	–	–
Vertical	Online (sompar1$_V$)	Yes	Slow	High
Horizontal	Batch (sompar2$_H$)	Yes	Fast	Low
Vertical	Batch (sompar2$_V$)	Yes	Fast	Low

Table 2 Distributed data sets

Type of distribution	Number of distributed data sources	Number of input vectors in each data source	Number of attributes in each data source	Size of each data set (Gb)
Horizontal	4	$4 * 10^6$	100	1
Vertical	4	$16 * 10^6$	25	1
Horizontal	2	$8 * 10^6$	100	2
Vertical	2	$16 * 10^6$	50	2
Single data warehouse	1	$16 * 10^6$	100	4

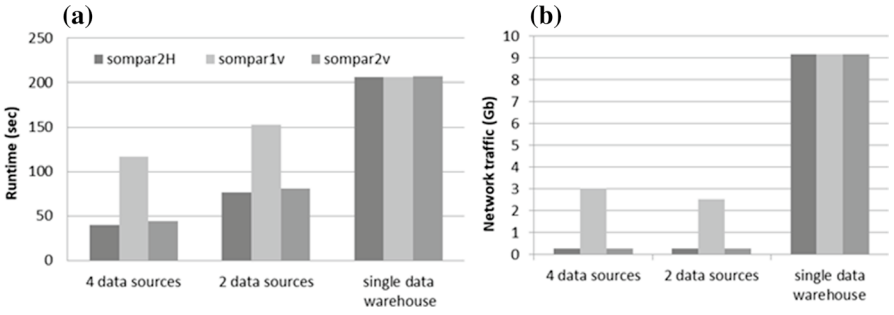

Fig. 6 Experiments with parallel SOM algorithm: **a** execution time; **b** network traffic

We use the data set generated on the basis of a Gaussian distribution with predetermined centers of the clusters. Each value in our data set can be viewed as independent according to one-dimensional distribution, so the input vectors are independent.

In our experiments, we evaluate SOM with 15 neurons (width = 3, height = 5). While this allows for a relatively low amount of computations, this amount still suffices to identify three clusters on data. For parameter "neighborhood of the learning" we assign the constant value $\eta = 1$, which corresponds to the "Winner Takes All" (WTA) configuration of SOM, and it also accelerates computations.

As shown in Table 2, data set is divided into two and four parts by input vectors (to simulate horizontal distribution) and by attributes (to simulate vertical distribution).

In Fig. 6a, we report the execution time for horizontal and vertical distributions of data. The comparison is shown for the parallel SOM algorithm adapted to horizontally ($sompar2_H$) and vertically ($sompar1_V$ and $sompar2_V$) distributed data. We observe that the execution time of $sompar2_H$, $sompar1_V$ and $sompar2_V$ for distributed clustering at data sources is lower than for clustering at a single data warehouse. The measurements also show that the execution time of $sompar1_V$ (with synchronization for every input vector) is longer than the run time of other variants. The reason is the large number of interactions between data sources and

computing node (many invocations of *paralleled* function) that increase the overhead of the distributed execution.

The differences among the variants are increasing with larger source nodes. We explain this by the higher number of invocations of *copy* and *join* functions for the distributed algorithm on a higher number of data sources.

Figure 6b shows a comparison of the network traffic for both types of data distribution and variants of parallel SOM algorithm. The traffic is higher for variant $sompar1_V$ with synchronization for every input vector. As with the execution time, network traffic is also higher when gathering data in a single data warehouse. This is because the volume of SOM that is transferred in our parallel implementation is much smaller than the volume of all data that are transferred in the traditional implementation.

In Fig. 6, we observe that, with an increasing amount of data, there will be a larger difference in the execution time (Fig. 6a) and network traffic (Fig. 6b) between parallelized federated learning and the version with a single data warehouse. The difference grows because of the increasing volumes of data transferred from data sources toward a single data warehouse, while the size of SOM communicated in federated learning does not depend on the data size. Differences in execution time (Fig. 6a) and network traffic (Fig. 6b) between the versions with synchronization for every input vector (variant sompar1v) and with synchronization for every epoch (variants sompar2h and sompar2v) also grow with the increasing amount of data, because of the growing number of interactions between the data sources and the computing node in the version with synchronization for every input vector (due to the higher number of input vectors).

7 Conclusion

This paper proposes a novel approach for optimizing the parallel implementation of the SOM clustering algorithm. Our approach formally transforms a high-level representation of a SOM algorithm into a parallel implementation that performs major calculations at the data source nodes, rather than transferring data for processing to a central computing node. We show that our approach is well suited for the technology of federated learning that is currently widely used for multilayer artificial neural networks.

We analyze the two possibilities of distributed federated learning—online and batch SOM algorithms. Each of the versions has advantages and weaknesses; therefore, we have considered their implementation for different kinds of data distributions between the data sources: horizontal and vertical. Our analysis confirmed by experiments shows that, for the horizontal data distribution, we can use only the batch SOM algorithm. For the vertical data distribution, our analysis shows and experiments confirm that the online version has a higher network traffic and longer execution time.

Acknowledgements We are grateful to the anonymous reviewers whose very helpful comments allowed us to significantly improve. This work was supported by the German Ministry of Education and Research (BMBF) in the framework of project HPC2SE at the University of Muenster.

References

1. Dehghani Z (2019) How to move beyond a monolithic data lake to a distributed data mesh. https://martinfowler.com/articles/data-monolith-to-mesh.html
2. Voigt P, Von dem Bussche A (2017) The EU general data protection regulation (GDPR). In: A practical guide, 1st ed. Springer International Publishing, Cham
3. California Consumer Privacy Act Home Page. https://www.caprivacy.org/
4. Konecný J, Brendan McMahan H, Ramage D, Richtárik P (2016) Federated optimization: distributed machine learning for on-device intelligence. arXiv:CoRRabs/1610.02527(2016)
5. Yang Q, Liu Y, Chen T, Tong Y (2019) Federated machine learning: concept and applications. ACM Trans Intell Syst Technol 10(2):12
6. Kohonen T (2001) Self-organizing maps (Third Extended Edition), New York
7. Kholod I, Shorov A, Efimova M, Gorlatch S (2019) Parallelization of algorithms for mining data from distributed sources. PaCT-2019. Springer. LNCS, pp 289–303 https://doi.org/10.1007/978-3-030-25636-4_23
8. Hastie T, Tibshirani R, Friedman J (2001) The elements of statistical learning: data mining, inference, and prediction. Springer
9. Dean J, Ghemawat S (2004) MapReduce: simplified data processing on large clusters. In Proceedings of Operating Systems Design and Implementation. San Francisco, CA
10. Gorlatch S, Cole M (2011) Parallel Skeletons. In: Padua D (ed.) Encyclopedia of parallel computing. Springer
11. Lawrence RD, Almasi GS, Rushmeier HE (1999) A scalable parallel algorithm for selfor-ganizing maps with applications to sparse data mining problems. Data Min Knowl Disc 3(2):171–195
12. Fort J, Letrémy P, Cottrell M (2002) Advantages and drawbacks of the Batch Kohonen algo-rithm. ESANN
13. Weichel Ch (2010) Adapting self-organizing maps to the mapreduce programming paradigm. STeP, pp 119–131. https://doi.org/10.1524/9783486853162.119
14. Sarazin T, Azzag H, Lebbah M (2014) SOM Clustering using spark-mapreduce. In: 2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops, pp 1727–1734 https://doi.org/10.1109/IPDPSW.2014.192
15. Dafonte C, Garabato D, Álvarez MA, Manteiga M (2018) Distributed fast self-organized maps for massive spectrophotometric data analysis. Sensors (Basel) 18(5):1419. Published 2018 May 3. https://doi.org/10.3390/s18051419
16. Flavius LG, Jose Alfredo FC (2008) Parallel self-organizing maps with application in clustering distributed data. Neural Networks. IJCNN 2008. IEEE International Joint Conference on IEEE World Congress on Computational Intelligence
17. Li Q, et al (2020) Federated learning systems: vision, hype and reality for data privacy and protection. arXiv:abs/1907.09693
18. Ingerman A, Ostrowski K (2019) Introducing TensorFlow Federated https://blog.tensorflow.org/2019/03/introducing-tensorflow-federated.html
19. Ryffel Th, Trask A, Dahl M, Wagner B, Mancuso J, Rueckert D, Passerat-Palmbach J (2018) A generic framework for privacy preserving deep learning. preprint arXiv:1811.04017
20. An Industrial Grade Federated Learning Framework https://fate.fedai.org/
21. Paddle Federated Learning https://github.com/PaddlePaddle/PaddleFL
22. Kholod I, Kuprianov M, Titkov E, Shorov A, Postnikova E, Mironenko I, Sokolov S (2019) Training normal Bayes classifier on distributed data. Proc Comput Sci 150:389–396. https://doi.org/10.1016/j.procs.2019.02.068
23. Kholod I, Rukavitsyn A, Reva N, Shorov A (2019) Distributed data clustering by neural network algorithms. In: Proceedings of the 2019 IEEE Russia Section Young Researchers in Electrical and Electronic Engineering Conference—IEEE. pp 249–253. https://doi.org/10.1109/EICONRus.2019.8657175
24. https://github.com/Awethon/SOM-FuncBlock
25. https://github.com/iiholod/XelopesFL