# Efficient Implementation of Self-Organizing Map for Sparse Input Data

Josué Melka and Jean-Jacques Mariage
*Laboratoire d'Informatique Avancée de Saint-Denis, Université Paris 8, 2 Rue de la Liberté, Saint-Denis, France*

Keywords: Neural-based Data-mining, Self-Organizing Map Learning Algorithm, Complex Information Processing, Parallel Implementation, Sparse Vectors.

Abstract: Neural-based learning algorithms, which in most cases implement a lengthy iterative convergence procedure, are often hardly adapted to very sparse input data, both due to practical issues concerning time and memory usage, and to the inherent difficulty of learning in high dimensional space. However, the description of many real-world data sets is sparse by nature, and learning algorithms must circumvent this barrier. This paper proposes adaptations of the standard and the batch versions of the Self-Organizing Map algorithm, specifically fine-tuned for high dimensional sparse data, with parallel implementation efficiency in mind. We extensively evaluate the performance of both adaptations on a set of experiments carried out on several real and artificial large benchmark datasets of sparse format from the LIBSVM Data: Classification. Results show that our approach brings a significant improvement in execution time.

## 1 INTRODUCTION

The *Self-Organizing Map* (SOM) algorithm (Kohonen, 1982) is an unsupervised neural network (NN) model that maps input data from a high-dimensional vector space onto an ordered two-dimensional surface. This mapping preserves topological relations in the data space.

SOM extracts the characteristic features of possibly complex non linear relations between categories implicit in the original data space. These relations would otherwise stay hidden from the researcher's eye due to the dimensionality and sparsity of noisy samples. Thanks to this property, and despite a highly time-consuming iterative convergence procedure, SOM is widely applied to visual inspection and clustering of huge data sets of many kinds.

**Applying NN to Large Datasets.** In data-mining applications[1], the amount of generated data quickly becomes prohibitive, if not intractable. Obviously, the hugeness of the data mass is unavoidable, because of the nature of the task itself. Using NNs for raw data inspection, one has no warranty that the original data space has been rigorously sampled. Apart from handmade carefully calibrated clean benchmark datasets, in real world applications samples remain noisy with spatial and temporal variations. We cannot know in advance to which extent the possible states of the characteristic features of the underlying processes are represented with sufficient resolution.

The only way to overcome this problem is to try and apprehend the inner variability of the data space through a reasonably wide amount of samples. The widest the amount of data, the best one may compensate for the inherent lack of precision of the sampling methods applied to data spaces. It becomes easier to do so, as data storage becomes more affordable, and data sets constantly increase in size and complexity and tend to reach considerable volumes. But, for such large-scale data processing, NNs training typically takes days to weeks (Wittek, 2013).

On the other hand, generalization of multi-core CPU make parallel computation resources more interesting. Open source specialized machine learning libraries on GPU now offer a wide access to a growing number of learning algorithms (Torch, Theano, GPUMLib...). Implementations are available for multi-core CPU and GPU, either locally on a single computer or such GPU as the Nvidia Tesla, Titan or GeForce GTX, or on cluster computing systems on GPU (Nvidia) or CPU grids (Spark MLlib).

**Reducing SOM Time Costs.** The computational cost of the SOM algorithm is highly dependent on the

[1]Following (Ultsch, 1999), "*We define Data Mining as the inspection of a large dataset with the aim of Knowledge Discovery*".

Melka J. and Mariage J.
Efficient Implementation of Self-Organizing Map for Sparse Input Data.
DOI: 10.5220/0006499500540063
In *Proceedings of the 9th International Joint Conference on Computational Intelligence (IJCCI 2017)*, pages 54-63
ISBN: 978-989-758-274-5

number of input vectors in the database, but the algorithmic complexity is mainly concerned by the dimensionality of the input vectors.

In many cases, data extraction methods (e.g. those commonly used in text-mining) produce sparse vectors with spatial correlation, albeit large in dimensionality. Depending on the preprocessing methods applied to the data, the number of non zero values in a vector can merely reach about a few percents of its dimension, and sometimes even far less than 1%. In their experiments, (Bernard et al., 2015) reported 1 non zero value in 5,000 for vectors up to half a million dimensions. In such cases, it becomes possible to drastically reduce the computing time and therefore greatly improve the efficiency of the algorithm.

Besides applying dimensionality reduction techniques such as feature compression (PCA, random-mapping, etc.) or feature selection, which will not be our purpose here, various modifications have been proposed in the literature to implement parallel versions of the SOM algorithm, that we will review in section 4.

Strikingly however, little attention seems to have been paid to possible rewritings of some crucial low level parts of the original algorithm, which can bring a substantial gain in execution time. The, rather scarce, related work concerning sparsity in the standard SOM algorithm is given at the beginning of section 3.

**Our Contribution.** We present a rewrite of the standard version of the algorithm, also refereed to as "on-line"[2], adapted to sparse input. We will also present a modified batch SOM version specifically tailored for both sparsity and parallelism efficiency. In the following, we will refer to our variants as **Sparse-Som** and **Sparse-BSom**.

In order to evaluate their respective performance, we compare them using **Somoclu** (Wittek et al., 2017) as a standard benchmark. The **Somoclu** library offers parallel computing facilities, relying on both OpenMP and MPI for multicore execution.

We thus measure the speed performance of three different SOM implementations. Regardless of result precision, maps are identically-configured to avoid unwanted parametric influence effects, while focusing on speed performance improvements brought by our modifications. We evaluate the execution time evolution of the batch versions, with respect to increasing parallelization levels, by varying the number of cores and threads devoted to the calculations.

We also report results from a series of extensive experiments over nine artificial and real datasets, with vectors varying in number, size and densities. Training result accuracy is investigated with the usual average quantization error, and by mean of recall and precision measure following majority-voting calibration of the maps.

The remainder of this paper is organized as follows. We first briefly recall the main characteristics of the standard SOM algorithm and its batch variant and proceed to their computational complexity analysis (section 2). We describe our modified versions of the standard SOM and of the batch SOM (section 3). We next consider the parallel implementation of these algorithms, and present our batch version with OpenMP acceleration (section 4). We then evaluate their performance on sparse artificial and real data sets and comment the obtained results (section 5). Finally, we draw conclusions from the experiments and suggest further developments for the proposed methods (section 6).

# 2 SOM ALGORITHM

In what follows, the standard SOM algorithm (Kohonen, 1982) is supposed to be known. The reader interested in a more detailed description must refer to the abundant literature about SOM and its thousands of applications in numerous domains. We here only briefly recall the main steps of the sequence of operations in the standard algorithm and the difference with its batch version, in order to state our adaptations to sparse data and parallelization.

## 2.1 Standard Algorithm

In the standard algorithm (Kohonen, 1997), weight vectors (the *codebook*) are updated at each time step $t$, immediately after the presentation of each data vector, in accordance with the following algorithm:

First, the squared euclidean distance[3] between an input vector $\mathbf{x}$ and weight vectors $\mathbf{w}$ is computed for every unit:

$$d_k(t) = \|\mathbf{x}(t) - \mathbf{w}_k(t)\|^2 \quad (1)$$

and the best-matching node $c$ is determined by

$$d_c(t) = \min_k d(t) \quad (2)$$

[2]We will hereafter systematically use the term *standard*, to avoid misleading interpretation of "on-line" with its general acceptance as real-time learning mode.

[3]Using the squared distance here is equivalent to using the euclidean distance, and avoids the square root computation.

Then the weight vectors are updated using

$$\mathbf{w}_k(t+1) = \mathbf{w}_k(t) + \alpha(t) h_{ck}(t) [\mathbf{x}(t) - \mathbf{w}_k(t)] \quad (3)$$

where $0 < \alpha(t) < 1$ is the learning-rate factor which decreases monotonically over time, and $h_{ck}(t)$ is the neighborhood function.

A commonly used neighborhood function is the Gaussian

$$h_{ck}(t) = \exp\left(-\frac{\|r_k - r_c\|^2}{2\sigma(t)^2}\right) \quad (4)$$

where $r_k$ and $r_c$ denote the coordinates of the nodes $k$ and $c$ respectively, and the width of the neighborhood $\sigma(t)$ decreases over time during $t_{\max}$ training iterations.

## 2.2 Batch Algorithm

The batch version of the SOM batches all the input samples together in each epoch (Kohonen, 1993; Mulier and Cherkassky, 1995). Equations (1) and (2) are computed once at the start of each epoch. As in the standard algorithm, weight vectors of the triggered nodes and their neighbors are updated, but only once at the end of each epoch, with the average of all the training samples that trigger them:

$$\mathbf{w}_k(t_f) = \frac{\sum_{t_0}^{t_f} h_{ck}(t')\mathbf{x}(t')}{\sum_{t_0}^{t_f} h_{ck}(t')} \quad (5)$$

where $t_0$, $t'$ and $t_f$ respectively refer to the first, current and last time indexes over the running epoch, and the neighborhood does not shrink during the epoch, thus $\sigma(t') = \sigma(t_0)$.

A proof of the convergence and ordering of the Batch Map is established in (Cheng, 1997). Another batch oriented version closer to the original algorithm has been proposed by (Ienne et al., 1997) but is much less used.

## 2.3 Complexity

Hereafter, we will use the following notations. $M$ is the number of units in the network grid, $D$ is the dimensions of the vectors, $N$ is the number of sample vectors, $T$ is the $t_{\max}$ of the standard version and $K$ is the number of epochs of the batch version.

**Time.** The computational complexity of the standard version is $O(TMD)$ for both equations (1) and (3) [4]. For the batch version, the complexity of equations (1) and (5) is $O(KNMD)$. Complexity of the two versions being similar if one chooses $T = N \times K$, we therefore only refer hereafter to the standard version for simplicity.

Since we use sparse vectors as inputs, let us define $d = D \times f$ where $f$ is the fraction of nonzero values in the inputs, the resulting complexity can be $O(TMd)$ if we express the equations appropriately, which may be very attractive in the case of $d \ll D$.

**Memory.** Memory requirements for the SOM algorithm depend on three factors, namely vectors size, units number and input data size.

With the sparse version, the size of the codebook remains unchanged and still requires $O(MD)$ space, but the size of the input data is reduced from $O(ND)$ to $O(Nd)$. This can considerably lower memory requirements for highly sparse large data sets, especially when $M \ll N$, which is usually the case for complex information processing in data mining applications.

# 3 EXPLOITING SPARSENESS

To turn the sparseness drawback to our advantage, we can appropriately rewrite the distance computation for the batch version, similarly to (Lawrence et al., 1999; Maiorana, 2008). We also exploit the key idea from the SD-SOM variant proposed by (Natarajan, 1997) to adapt the standard version to sparse input.

Another option for taking advantage of data sparseness, already proposed in (Kohonen, 1997; Kohonen, 2013), is to replace euclidean distance with dot-product. It is limited to cosine similarity metric, and requires units normalization after each update, which makes it less convenient for the standard algorithm.

## 3.1 Batch Version

The computation of eq. (5) depends only on the nonzeros values in the input. Rewriting eq. (1) accordingly, gives:

$$d_k(t) = \|\mathbf{w}_k(t)\|^2 + \|\mathbf{x}(t)\|^2 - 2(\mathbf{w}_k(t) \cdot \mathbf{x}(t)) \quad (6)$$

The values of the squared norms can be precomputed, once for $\mathbf{x}$ and before each epoch for $\mathbf{w}$, and their influence on the computation time is thus negligible.

## 3.2 Standard Version

To simplify the notation, $\beta(t)$ replaces $\alpha(t)h_{ck}(t)$ in the following.

[4] Complexity of eq. (2) does not depend on vector size and it is only $O(TM)$

#### 3.2.1 Codebook Update

We can express equation (3) as:

$$\begin{aligned}\mathbf{w}_k(t+1) &= \mathbf{w}_k(t) + \beta(t)[\mathbf{x}(t) - \mathbf{w}_k(t)] \\ &= \mathbf{w}_k(t) - \beta(t)\mathbf{w}_k(t) + \beta(t)\mathbf{x}(t) \\ &= (1-\beta(t))\mathbf{w}_k(t) + \beta(t)\mathbf{x}(t) \qquad (7a) \\ &= (1-\beta(t))\left[\mathbf{w}_k(t) + \frac{\beta(t)}{1-\beta(t)}\mathbf{x}(t)\right] \qquad (7)\end{aligned}$$

If we store the coefficient $(1-\beta(t))$ separately, we don't need to update all the values of $\mathbf{w}$ in the update phase, but only those affected by $\mathbf{x}(t)$.

#### 3.2.2 Distance Computations

We can rewrite eq. (1) as we did in section 3.1 for eq. (6), but the computation of $\mathbf{w}(t)$ at each step still remains problematic. However, if we keep the value of $\|\mathbf{w}(t)\|^2$ at each step, we can compute $\|\mathbf{w}(t+1)\|^2$ efficiently from eq. (7a).

$$\begin{aligned}\|\mathbf{w}_k(t+1)\|^2 &= \|(1-\beta(t))\mathbf{w}_k(t) + \beta(t)\mathbf{x}(t)\|^2 \\ &= \|(1-\beta(t))\mathbf{w}_k(t)\|^2 + \|\beta(t)\mathbf{x}(t)\|^2 \\ &\quad + 2((1-\beta(t))\mathbf{w}_k(t) \cdot \beta(t)\mathbf{x}(t)) \\ &= (1-\beta(t))^2\|\mathbf{w}_k(t)\|^2 + \beta(t)^2\|\mathbf{x}(t)\|^2 \\ &\quad + 2\beta(t)(1-\beta(t))(\mathbf{w}_k(t) \cdot \mathbf{x}(t)) \qquad (8)\end{aligned}$$

### 3.3 Modified Algorithm

Putting all of these changes together, we obtain the Algorithm 1 for the modified standard version.

**Numerical Stability.** To avoid division by very small values in line 24, we rescale $\mathbf{z}_k$ every time $\gamma_k$ becomes very small (below some given $\varepsilon$ value). Such cases remain rare enough to have no impact on the overall complexity.

**Algorithm 1:** Standard Sparse SOM.

```
Input: x a set of N sparse vectors of D components.
Data: z the codebook of M dense vectors.
Data: γ an array of reals, satisfying w_k = γ_k z_k
Data: ω an array of reals, satisfying ω_k = Σ_j w_kj^2
Data: χ an array of reals, satisfying χ_i = Σ_j x_ij^2
Data: ε to control the numerical stability, set it to
      very small value.
1  Procedure Init
2      for i ← 1 to N do χ_i ← Σ_j x_ij^2 ; init χ
3      for k ← 1 to M do init z, ω and γ for t = 0
4          initialize z_k ;
5          ω_k ← Σ_{j∈1,...,D} z_kj^2 ;
6          γ_k ← 1 ;
7  Procedure Rescale
       Input: k
8      for j ← 1 to D do
9          z_kj ← γ_k z_kj
10     γ_k ← 1
11 Procedure Main
12     Init () ;
13     for t ← 1 to t_max do
14         choose an input i ∈ 1...N ;
15         for k ← 1 to M do compute distance
               between x_i and w_k
16             d_k ← ω_k + χ_i − 2γ_k Σ_j z_kj x_ij
17         c ← arg min_k d ;
18         interpolate α ;
19         foreach k ∈ N_c do update z_k and ω_k
20             interpolate σ ;
21             β ← α exp(‖r_k − r_c‖^2 / 2σ^2) ;
22             ω_k ← (1−β)^2 ω_k + β^2 χ_i + 2β(1−
                   β)γ_k Σ_j z_kj x_ij ;
23             foreach j such as x_ij ≠ 0 do
24                 z_kj ← z_kj + β/((1−β)γ_k) x_ij
25             γ_k ← (1−β)γ_k ;
26             if γ_k < ε then rescale z_k
27                 Rescale (k)
28     for k ← 1 to M do get the actual codebook w
29         Rescale (k)
```

## 4 PARALLELISM

The SOM algorithm has experienced numerous parallel implementation attempts, both with dedicated hardware (neurocomputers) and massively parallel computers in the early years (Wu et al., 1991; Seiffert and Michaelis, 2001) and later by using different cluster architectures (Guan et al., 1997; Bandeira et al., 1998; Tomsich et al., 2000). A comprehensive, but somewhat outdated review of the different approaches can be found in (Hämäläinen, 2002).

It should be noted that the batch version is often preferred for computational performance reason, as it only needs a few iteration cycles and it can be parallelized efficiently, which greatly speeds up the learning process (Kohonen et al., 2000; Lagus et al., 2004; Lawrence et al., 1999; Maiorana, 2008; Wittek et al., 2017).

### 4.1 Workload Partitioning

Different levels of parallelism are suitable for neural network computations (Nordström, 1992), but the following ones are most widely applicable:

- *Network partitioning* splits the NN, dividing up the neuron units among different processors; that

is advantageous since most of the calculations are unit located, and thus independent.

- *Data partitioning* dispatches the input data among processors; in this case the complete network needs to be duplicated (or shared).

(Lawrence et al., 1999) points out that the first approach introduces a latency constant and is therefore less attractive. With true partitioning, which is often communication bound (e.g. with distinct machines on a cluster using message passing), it is difficult to mix both schemes, altough some authors (Yang and Ahuja, 1999; Silva and Marques, 2007) proposed such hybrid approaches. This is less problematic with shared memory systems.

By the serial nature of the standard SOM version, data partitioning is irrelevant, and it turns out that it is hard to parallelize efficiently. The main reason is primarily due to the high frequency of thread synchronization that prevents it to take a real advantage from parallelism.

That does not apply to the batch version. While implementing the batch algorithm, we noticed that the memory access latency is a key performance issue on modern CPUs, even without parallelism and much more so in the shared-memory multiprocessing paradigm[5]. In our experiments to parallelize the batch SOM algorithm with OpenMP, we gained considerable speed improvement with the outer loops on the network and the inner loops on the data, which lead us to mix the two approaches by using data partitioning for eq. (6) and network partitioning for eq. (5).

## 4.2 OpenMP

OpenMP (Dagum and Menon, 1998) provides a shared-memory multiprocessing paradigm easily applicable to C/C++ or Fortran code with special directives, without modifying the code semantics. Thanks to this simplicity, we were able to parallelize our batch version without significant changes in the source code.

A major issue we have encountered is to find a proper management of the processor cache, which has a very significant impact to performances on modern processors, by avoiding multiple access to memory. For this reason, the loop order has been modified for certain portions of code, without changing the underlying algorithm. The resulting algorithm is shown in Algorithm 2.

[5]This is specific to sparse vector operations, because of the unpredictable pattern of memory access, which cannot take advantage of the processor cache, and makes it challenging to split the workload evenly across processors.

The outer loops (lines 5 and 12) are set on the codebook, and the inner loops (lines 7 and 15) on the data. With OpenMP, using the `omp for` directive on the outer loop is equivalent to use network partitioning, and on the inner loop, it is equivalent to use data partitioning.

In order to simplify the underlying code and prevent shared variables from concurrent writes, our parallel version uses outer parallel loop for best match units search (line 7) and inner parallel loop for updates (line 12).

**Algorithm 2:** Batch Sparse BSOM.

**Input:** $\mathbf{x}$ a set of $N$ sparse vectors of $D$ components.
**Data:** $\mathbf{w}$ initialized codebook of $M$ dense vectors.
**Data:** $\chi$ an array of $N$ reals, satisfying $\chi_i = \sum_j \mathbf{x}_{ij}^2$
**Data:** *dst* array of $N$ reals to store best distances.
**Data:** *bmu* array of $N$ integers to store best match units.
**Data:** *num* array of $D$ reals to accumulate numerator values.

1 **for** $i \leftarrow 1$ **to** $N$ **do** $\chi_i \leftarrow \sum_j \mathbf{x}_{ij}^2$ ; init $\chi$
2 **for** $e \leftarrow 1$ **to** $e_{\max}$ **do** train one epoch
3  interpolate $\sigma$ ;
4  **for** $i \leftarrow 1$ **to** $N$ **do** $dst_i \leftarrow \infty$ ; initialize *dst*
5  **for** $k \leftarrow 1$ **to** $M$ **do** find all bmus
6  $\omega \leftarrow \sum_j \mathbf{w}_{kj}^2$ ;
7  **forall** $i \in 1,\ldots,N$ **do**
8  $d \leftarrow \omega + \chi_i - 2(\mathbf{x}_i \cdot \mathbf{w}_k)$ ;
9  **if** $d < dst_i$ **then** store best match unit
10  $dst_i \leftarrow d$ ;
11  $bmu_i \leftarrow k$ ;
12  **forall** $k \in 1,\ldots,M$ **do**
13  $den \leftarrow 0$ ; init denominator
14  **for** $j \leftarrow 1$ **to** $D$ **do** $num_j \leftarrow 0$ ; init numerator
15  **for** $i \leftarrow 1$ **to** $N$ **do** accumulate *num* and *den*
16  $c \leftarrow bmu_i$ ;
17  $h \leftarrow \exp(\|r_k - r_c\|^2 / 2\sigma^2)$ ;
18  $den \leftarrow den + h$ ;
19  **for** $j \leftarrow 1$ **to** $D$ **do**
20  $num_j \leftarrow num_j + h\,\mathbf{x}_{ij}$
21  **for** $j \leftarrow 1$ **to** $D$ **do** update $\mathbf{w}_k$
22  $\mathbf{w}_{kj} \leftarrow \frac{num_j}{den}$

# 5 PERFORMANCE EVALUATION

To evaluate the performance of our implementations, we have trained several networks with the same configuration parameters on various datasets and measured their relative performance, using the following parameters:

- $30 \times 40$ rectangular unit grids for all the networks.
- $t_{\max} = 10 \times N_{\text{samples}}$ (or $K_{\text{epochs}} = 10$ for the batch version).

- rectangular neighborhood limits with the radius $r(t)$ decreasing linearly from 15 to 0.5.
- Gaussian neighborhood function, with $\sigma(t) = 0.3 \times r(t)$.
- $\alpha(t) = 1 - (t/t_{max})$ if applicable.

## 5.1 Datasets

We have selected several large datasets of sparse format from (Chang and Lin, 2006) to evaluate the performance of the two approaches on true examples.

**rcv1:** Reuters corpus dataset, multiclass (Lewis et al., 2004).

**news20:** netnews dataset, normalized (Lang, 1995).

**sector:** text categorization dataset, normalized (McCallum and Nigam, 1998).

**mnist:** MNIST database of handwritten digits (LeCun et al., 1998).

**usps:** subset of CEDAR handwritten database (Hull, 1994).

**protein:** bioinformatic dataset (Wang, 2002).

**dna:** recognizing splice-junction of primate gene sequences (Noordewier et al., 1991).

**satimage:** classification of satellite images (King et al., 1995).

**letter:** character recognition dataset (Frey and Slate, 1991).

The detailed properties of these datasets are given in Table 1. 'Features' denote the number of values inside the vectors, 'density' gives the percentage of non zero values; double rows in columns 'samples' and 'density' separate the specific values for the training set and the test set.

## 5.2 Speed Benchmark

As a comparison baseline we have used the open-source tool **Somoclu** (Wittek et al., 2017) whose characteristics are the following :

- supports both dense and sparse vectors as input;
- is designed for performance (though no specific optimization was used on sparse inputs);
- uses the batch algorithm for training;
- can be parallelized using OpenMP and/or MPI.

### 5.2.1 Parallel Comparison on Batch Algorithm

We measured the performance of the parallel implementations of the batch algorithm in terms of execution time, with various levels of parallelism.

Table 1: Characteristics of the datasets.

| | classes | features | samples | density |
|---|---|---|---|---|
| rcv1 | 53 | 47236 | 15564<br>518571 | 0.14<br>0.14 |
| news20 | 20 | 62061 | 15933<br>3993 | 0.13<br>0.13 |
| sector | 105 | 55197 | 6412<br>3207 | 0.29<br>0.30 |
| mnist | 10 | 780 | 60000<br>10000 | 19.22<br>19.37 |
| usps | 10 | 256 | 7291<br>2007 | 100.00<br>100.00 |
| protein | 3 | 357 | 17766<br>6621 | 29.00<br>26.06 |
| dna | 3 | 180 | 2000<br>1186 | 25.34<br>25.14 |
| satimage | 6 | 36 | 4435<br>2000 | 98.99<br>98.96 |
| letter | 26 | 16 | 15000<br>5000 | 100.00<br>100.00 |

For this test, we have used the sector, news20, mnist and usps training datasets. The first two are very sparse and sufficiently large to evaluate the optimization effect on sparse data, and the last two are intended to observe the implementation behavior on mostly dense data.

Tests were conducted on a multicore computer with 4 sockets of 6 cores Intel Xeon E5-4610 at 2.40 GHz (2 threads at 1.2 GHz per core). Several runs were made with different number of cores assigned to the computation.

Results shown in Figure 1 demonstrate that: **Somoclu** speed is correlated with the total input vectors dimension, while **Sparse-BSom** speed is closely correlated with the number of non-zero values. Notably, **Sparse-BSom** is several order of magnitude faster than **Somoclu** in case of very sparse data, and stays faster in all four cases.

For both implementations, execution time decreases almost linearly when the number of cores grows (the dotted lines represent the theoretical speed-up linearly based on the number of cores).

### 5.2.2 Serial Comparison of Optimized Versions

We carried out experiments to compare our optimized approaches to each other. To this end, we have selected datasets with various densities and executed our implementations using the same parameters. As stated before the standard version cannot be parallelized efficiently, so we compared single threaded versions in these tests.

Results are shown in Figure 2. **Sparse-BSom** performs better than **Sparse-Som** on very sparse data,

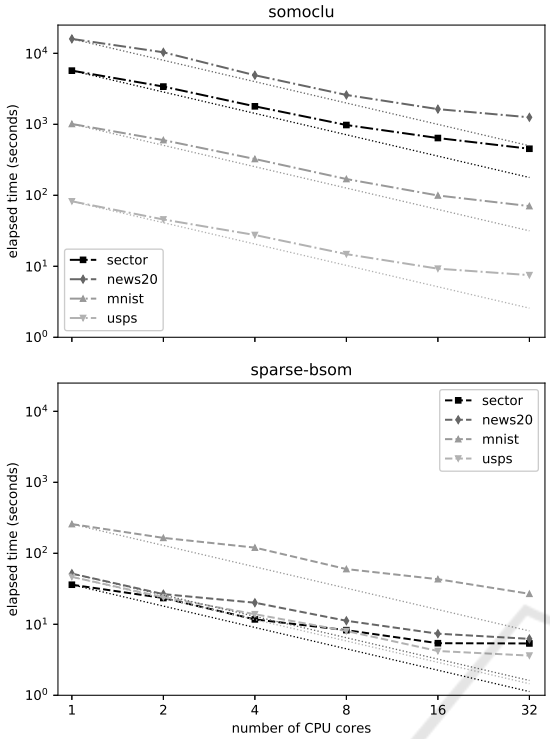


Figure 1: Parallel speed benchmark.

which is easy to explain, because this last algorithm involves more calculations, and for this reason has a larger constant factor in its time complexity. Less clear is the reason why **Sparse-Som** performs better on dense data. One possible interpretation is that it is related to the different memory access management in both algorithms.

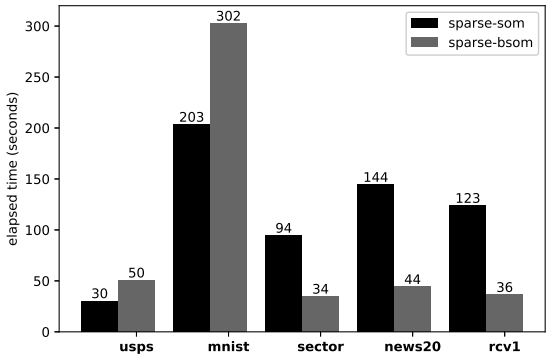


Figure 2: Serial speed benchmark (lower is faster).

## 5.3 Quality Evaluation

Some authors have reported degradations in the resulting maps using the batch algorithm (Fort et al., 2002; Nöcker et al., 2006). Here we look for such effects with our implementations.

### 5.3.1 Methodology

Various metrics can be used to analyze the maps without human labelling, the most common one is the average quantization error (Kohonen et al., 1996) defined as

$$Q = \frac{\sum_{i=1}^{N} \|\mathbf{x}_i - \mathbf{w}_c\|}{N} \quad (9)$$

where $\mathbf{w}_c$ is the best match unit for $\mathbf{x}_i$.

Since SOM can be used in a supervised manner to classify input vectors, one can also use standard evaluation metrics (recall, precision). Because our datasets are all multi-class, we calculate metrics for each label, and find their average, weighted by support (the number of true instances for each label).

We have used the following evaluation method for all datasets:

1. train the SOM network with the training part of the dataset.
2. perform units calibration with the associated labels (each unit is labeled according to the majority of the data it matches).
3. predict the labels of the train data according to the label attributed to their best match units.
4. do the same as step 3 on the test data.

If a unit has not attracted data in the training stage, it is not labeled; if in test stage it attracts some input data, we assign it a non-existent class. Though this strategy can significantly decrease the overall recall score (it is possible to use more sophisticated approaches to deal with such cases), this simple method is in general enough to analyze the clustering quality.

### 5.3.2 Results

Detailed results are shown in Table 2: Quantization error and Table 3: Prediction evaluation (best F-score highlighted). The experiments were run five times, and we report mean values and standard deviation for each system.

It should be emphasized that no parameters optimization per dataset was performed, and that it is certainly possible to obtain better results with careful parameter tuning. For example, the network we have used (1200 units) is too large for small training datasets, which probably explains the low recall rate for the dna dataset. It seems, however, that the standard SOM version is more robust against such type of difficulty, indicating that data samples are better distributed over the network with this algorithm.

The first observation we can make is that, though not exactly identical, results of both batch versions

Table 2: Quantization error.

| | **Sparse-Som** | **Sparse-BSom** | **Somoclu** |
|---|---|---|---|
| rcv1 | $0.825 \pm 0.001$ | $\mathbf{0.816 \pm 0.001}$ | $0.817 \pm 0.004$ |
| news20 | $0.905 \pm 0.000$ | $\mathbf{0.901 \pm 0.001}$ | $0.904 \pm 0.001$ |
| sector | $0.814 \pm 0.001$ | $\mathbf{0.772 \pm 0.003}$ | $0.780 \pm 0.011$ |
| mnist | $\mathbf{4.400 \pm 0.001}$ | $4.500 \pm 0.008$ | $4.512 \pm 0.005$ |
| usps | $3.333 \pm 0.002$ | $\mathbf{3.086 \pm 0.006}$ | $3.117 \pm 0.010$ |
| protein | $2.451 \pm 0.000$ | $\mathbf{2.450 \pm 0.001}$ | $2.452 \pm 0.001$ |
| dna | $4.452 \pm 0.006$ | $\mathbf{3.267 \pm 0.042}$ | $3.272 \pm 0.013$ |
| satimage | $0.439 \pm 0.001$ | $\mathbf{0.377 \pm 0.001}$ | $0.378 \pm 0.002$ |
| letter | $0.357 \pm 0.001$ | $\mathbf{0.345 \pm 0.002}$ | $0.349 \pm 0.002$ |

Table 3: Prediction evaluation.

| | | **Sparse-Som** | | **Sparse-BSom** | | **Somoclu** | |
|---|---|---|---|---|---|---|---|
| | | precision | recall | precision | recall | precision | recall |
| rcv1 | *train* | $79.2 \pm 0.5$ | $79.3 \pm 0.6$ | $\mathbf{81.3 \pm 0.4}$ | $\mathbf{82.1 \pm 0.3}$ | $81.2 \pm 0.4$ | $81.9 \pm 0.5$ |
| | *test* | $73.7 \pm 0.4$ | $70.6 \pm 0.5$ | $\mathbf{76.6 \pm 0.4}$ | $\mathbf{72.6 \pm 0.5}$ | $75.6 \pm 0.5$ | $71.2 \pm 0.8$ |
| news20 | *train* | $\mathbf{64.2 \pm 0.5}$ | $\mathbf{62.8 \pm 0.5}$ | $50.3 \pm 0.9$ | $49.6 \pm 0.8$ | $50.8 \pm 1.6$ | $50.3 \pm 1.6$ |
| | *test* | $\mathbf{60.0 \pm 1.7}$ | $\mathbf{55.4 \pm 1.3}$ | $47.8 \pm 1.2$ | $43.6 \pm 1.2$ | $47.0 \pm 1.9$ | $42.8 \pm 1.4$ |
| sector | *train* | $\mathbf{77.2 \pm 0.9}$ | $\mathbf{73.2 \pm 0.9}$ | $58.4 \pm 0.5$ | $56.0 \pm 1.0$ | $57.3 \pm 1.4$ | $54.3 \pm 3.1$ |
| | *test* | $\mathbf{73.3 \pm 0.8}$ | $\mathbf{61.3 \pm 1.8}$ | $60.9 \pm 1.3$ | $44.8 \pm 1.0$ | $60.5 \pm 3.3$ | $41.3 \pm 3.6$ |
| mnist | *train* | $\mathbf{93.5 \pm 0.2}$ | $\mathbf{93.5 \pm 0.2}$ | $91.5 \pm 0.2$ | $91.5 \pm 0.2$ | $91.3 \pm 0.3$ | $91.3 \pm 0.3$ |
| | *test* | $\mathbf{93.4 \pm 0.2}$ | $\mathbf{93.4 \pm 0.2}$ | $91.7 \pm 0.2$ | $91.7 \pm 0.2$ | $91.7 \pm 0.4$ | $91.7 \pm 0.4$ |
| usps | *train* | $\mathbf{95.9 \pm 0.2}$ | $\mathbf{95.9 \pm 0.2}$ | $95.6 \pm 0.2$ | $95.6 \pm 0.2$ | $95.7 \pm 0.2$ | $95.7 \pm 0.2$ |
| | *test* | $91.4 \pm 0.3$ | $90.7 \pm 0.3$ | $\mathbf{92.4 \pm 0.5}$ | $\mathbf{91.5 \pm 0.4}$ | $92.1 \pm 0.5$ | $91.3 \pm 0.4$ |
| protein | *train* | $\mathbf{56.7 \pm 0.2}$ | $\mathbf{57.5 \pm 0.2}$ | $56.7 \pm 0.4$ | $57.6 \pm 0.3$ | $56.3 \pm 0.3$ | $57.2 \pm 0.2$ |
| | *test* | $49.8 \pm 0.7$ | $51.2 \pm 0.6$ | $\mathbf{50.7 \pm 0.7}$ | $\mathbf{52.1 \pm 0.6}$ | $50.5 \pm 1.0$ | $51.6 \pm 0.5$ |
| dna | *train* | $\mathbf{90.9 \pm 0.6}$ | $\mathbf{90.8 \pm 0.5}$ | $88.5 \pm 0.6$ | $88.5 \pm 0.5$ | $89.3 \pm 0.6$ | $89.3 \pm 0.6$ |
| | *test* | $\mathbf{77.7 \pm 1.5}$ | $\mathbf{69.6 \pm 2.1}$ | $81.9 \pm 2.9$ | $30.3 \pm 1.7$ | $83.9 \pm 2.7$ | $25.1 \pm 1.1$ |
| satimage | *train* | $92.3 \pm 0.4$ | $92.4 \pm 0.3$ | $92.5 \pm 0.4$ | $92.6 \pm 0.4$ | $\mathbf{93.0 \pm 0.2}$ | $\mathbf{93.1 \pm 0.1}$ |
| | *test* | $87.6 \pm 0.3$ | $85.4 \pm 0.4$ | $\mathbf{88.7 \pm 0.5}$ | $\mathbf{86.3 \pm 0.5}$ | $88.9 \pm 0.3$ | $85.5 \pm 0.7$ |
| letter | *train* | $\mathbf{83.8 \pm 0.3}$ | $\mathbf{83.7 \pm 0.3}$ | $81.9 \pm 0.3$ | $81.7 \pm 0.4$ | $81.3 \pm 0.8$ | $81.2 \pm 0.8$ |
| | *test* | $\mathbf{81.5 \pm 0.5}$ | $\mathbf{81.1 \pm 0.5}$ | $80.2 \pm 0.3$ | $79.8 \pm 0.5$ | $78.9 \pm 0.8$ | $78.6 \pm 0.8$ |

(**Somoclu** and **Sparse-BSom**) are perfectly consistent. Therefore, we focused our analysis on the differences between our standard version and our batch version.

With regard to the quantization error, it is clear that the batch version performs better than the standard version, but this has no effect on the predictive performance. The predictive benchmark results are globally better with the standard version than with the batch version. Furthermore, the results of the **Sparse-Som** also seem to be more stable, and never fall much lower than the **Sparse-BSom** results.

A significant gap occurs between the two versions for the news20 and sector datasets, which are both very sparse. However, we cannot generalize a negative impact of sparseness with the batch version, because of the counterexample with rcv1 results.

# 6 CONCLUSIONS

We have shown that, in case of the SOM algorithm, the sparse nature of many data models can be effectively tackled using an appropriate formulation of the calculations. The time required to train such a network was reduced proportionally to the data sparseness, and the input data can be used directly in compressed form, which saves memory requirements. This holds for both **Sparse-Som** and **Sparse-BSom**.

**Sparse-BSom** can also be parallelized efficiently on multi-core CPUs, as demonstrated by our experiments with OpenMP. This leads us to plan further experiments on a cluster computing implementation, potentially using MPI.

Unfortunately, due to the amount of synchronization required, **Sparse-Som** is much harder to paral-

lelize, and we found no way to significantly improve its performance compared to serial execution.

As regards the maps obtained with both our versions, we carried out an empirical qualitative analysis using various datasets. Our results confirm the current assumption that the behavior of the standard version is more stable and generally produces overall better results than the batch version.

In order to ensure reliable reproducibility of our results, our complete implementation is freely available online for the research community, with its documentation, on GitHub, under the terms of the GNU General Public License (https://github.com/yoch/sparse-som).

## ACKNOWLEDGEMENTS

We thank Gilles Bernard and Nourredine Aliane for their valuable comments.

## REFERENCES

Bandeira, N., Lobo, V., and Moura-Pires, F. (1998). Training a self-organizing map distributed on a pvm network. In *Neural Networks Proceedings, 1998. IEEE World Congress on Computational Intelligence. The 1998 IEEE International Joint Conference on*, volume 1, pages 457–461. IEEE.

Bernard, G., Aliane, N., and Manad, O. (2015). An experimentation line for underlying graphemic properties - acquiring knowledge from text data with self organizing maps. In *ICINCO 2015 - Proceedings of the 12th International Conference on Informatics in Control, Automation and Robotics, Volume 1, Colmar, Alsace, France, 21-23 July, 2015.*, pages 659–666.

Chang, C.-C. and Lin, C.-J. (2006). Libsvm data: Classification (multi class). https://www.csie.ntu.edu.tw/∼cjlin/libsvmtools/datasets/multiclass.html.

Cheng, Y. (1997). Convergence and ordering of kohonen's batch map. *Neural Computation*, 9(8):1667–1676.

Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55.

Fort, J.-C., Letremy, P., and Cottrell, M. (2002). Advantages and drawbacks of the batch kohonen algorithm. In *ESANN*, volume 2, pages 223–230.

Frey, P. W. and Slate, D. J. (1991). Letter recognition using holland-style adaptive classifiers. *Machine learning*, 6(2):161–182.

Guan, H., Li, C.-k., Cheung, T.-y., and Yu, S. (1997). Parallel design and implementation of som neural computing model in pvm environment of a distributed system. In *Advances in Parallel and Distributed Computing, 1997. Proceedings*, pages 26–31. IEEE.

Hämäläinen, T. D. (2002). Parallel implementation of self-organizing maps. In Seiffert, U. and Jain, L. C., editors, *Self-Organizing Neural Networks*, pages 245–278. Springer-Verlag, Inc., New York, USA.

Hull, J. J. (1994). A database for handwritten text recognition research. *IEEE Transactions on pattern analysis and machine intelligence*, 16(5):550–554.

Ienne, P., Thiran, P., and Vassilas, N. (1997). Modified self-organizing feature map algorithms for efficient digital hardware implementation. *IEEE Transactions on Neural Networks*, 8(2):315–330.

King, R. D., Feng, C., and Sutherland, A. (1995). Statlog: comparison of classification algorithms on large real-world problems. *Applied Artificial Intelligence an International Journal*, 9(3):289–333.

Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological cybernetics*, 43(1):59–69.

Kohonen, T. (1993). Things you haven't heard about the self-organizing map. In *1993 IEEE International Conference on Neural Networks*, pages 1147–1156. IEEE.

Kohonen, T. (1997). *Self-Organizing Maps*. Number 30 in Springer Series in Information Sciences. Springer, second edition.

Kohonen, T. (2013). Essentials of the self-organizing map. *Neural Networks*, 37:52–65.

Kohonen, T., Hynninen, J., Kangas, J., and Laaksonen, J. (1996). Som pak: The self-organizing map program package. *Report A31, Helsinki University of Technology, Laboratory of Computer and Information Science*.

Kohonen, T., Kaski, S., Lagus, K., Salojarvi, J., Honkela, J., Paatero, V., and Saarela, A. (2000). Self organization of a massive document collection. *IEEE transactions on neural networks*, 11(3):574–585.

Lagus, K., Kaski, S., and Kohonen, T. (2004). Mining massive document collections by the websom method. *Information Sciences*, 163(1):135–156.

Lang, K. (1995). Newsweeder: Learning to filter netnews. In *Proceedings of the 12th international conference on machine learning*, pages 331–339.

Lawrence, R. D., Almasi, G. S., and Rushmeier, H. E. (1999). A scalable parallel algorithm for self-organizing maps with applications to sparse data mining problems. *Data Mining and Knowledge Discovery*, 3(2):171–195.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Lewis, D. D., Yang, Y., Rose, T. G., and Li, F. (2004). Rcv1: A new benchmark collection for text categorization research. *Journal of machine learning research*, 5(Apr):361–397.

Maiorana, F. (2008). Performance improvements of a kohonen self organizing classification algorithm on sparse data sets. In *Proceedings of the 10th WSEAS International Conference on Mathematical Methods, Computational Techniques and Intelligent Systems*, MAMECTIS'08, pages 347–352. World Scientific and Engineering Academy and Society (WSEAS).

McCallum, A. and Nigam, K. (1998). A comparison of event models for naive bayes text classification. In *AAAI/ICML-98 Workshop on Learning for Text Categorization*, Technical Report WS-98-05, pages 41–48.

Mulier, F. and Cherkassky, V. (1995). Self-organization as an iterative kernel smoothing process. *Neural computation*, 7(6):1165–1177.

Natarajan, R. (1997). Exploratory data analysis in large, sparse datasets. Technical report, IBM Thomas J. Watson Research Division.

Nöcker, M., Mörchen, F., and Ultsch, A. (2006). An algorithm for fast and reliable esom learning. In *ESANN, 14th European Symposium on Artificial Neural Networks*, pages 131–136.

Noordewier, M. O., Towell, G. G., and Shavlik, J. W. (1991). Training knowledge-based neural networks to recognize genes in dna sequences. *Advances in neural information processing systems*, 3:530–536.

Nordström, T. (1992). Designing parallel computers for self organizing maps. In *Proceedings of the 4th Swedish Workshop on Computer System Architecture (DSA-92)*, pages 13–15.

Seiffert, U. and Michaelis, B. (2001). Multi-dimensional self-organizing maps on massively parallel hardware. In *Advances in Self-Organising Maps*, pages 160–166. Springer.

Silva, B. and Marques, N. (2007). A hybrid parallel som algorithm for large maps in data-mining. *New Trends in Artificial Intelligence*.

Tomsich, P., Rauber, A., and Merkl, D. (2000). Optimizing the parsom neural network implementation for data mining with distributed memory systems and cluster computing. In *Database and Expert Systems Applications, 2000. Proceedings. 11th International Workshop on*, pages 661–665. IEEE.

Ultsch, A. (1999). Data mining and knowledge discovery with emergent self-organizing feature maps for multivariate time series. *Kohonen maps*, 46:33–46.

Wang, J.-Y. (2002). *Application of support vector machines in bioinformatics*. PhD thesis, National Taiwan University.

Wittek, P. (2013). Training emergent self-organizing maps on sparse data with Somoclu. http://peterwittek.com/training-emergent-self-organizing-maps-with-somoclu.html.

Wittek, P., Gao, S. C., Lim, I. S., and Zhao, L. (2017). Somoclu: An efficient parallel library for self-organizing maps. *Journal of Statistical Software*, 78(9):1–21.

Wu, C.-H., Hodges, R. E., and Wang, C.-J. (1991). Parallelizing the self-organizing feature map on multiprocessor systems. *Parallel Computing*, 17(6-7):821–832.

Yang, M.-H. and Ahuja, N. (1999). A data partition method for parallel self-organizing map. In *Neural Networks, 1999. IJCNN'99. International Joint Conference on*, volume 3, pages 1929–1933. IEEE.