Speeding Up the Dissimilarity Self-Organizing Maps by Branch and Bound

Brieuc Conan-Guez[1] and Fabrice Rossi[2]

[1] LITA EA3097, Université de Metz, Ile du Saulcy, F-57045, Metz, France
Brieuc.Conan-Guez@univ-metz.fr
[2] Projet AxIS, INRIA, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France – Fabrice.Rossi@inria.fr

Abstract. This paper proposes to apply the branch and bound principle from combinatorial optimization to the Dissimilarity Self-Organizing Map (DSOM), a variant of the SOM that can handle dissimilarity data. A new reference model optimization method is derived from this principle. Its results are strictly identical to those of the original DSOM algorithm by Kohonen and Somervuo, while its running time is reduced by a factor up to 2.5 compared to the one of the previously proposed optimized implementation.

1 Introduction

The Dissimilarity Self Organizing Map (DSOM, also referred to as the Median SOM) is one of the adaptation of Kohonen's SOM [1] to dissimilarity data [2], i.e., to data that are only described by the results of pairwise comparisons of all observations via a dissimilarity measure.

Previous optimization works [3] have showed that the complexity of this algorithm can be greatly reduced by factorizing some calculations. The actual running time of the implementation was also reduced with the help of some heuristics (that don't modify the final result). The present paper proposes to replace and complement those *ad hoc* heuristics by simpler solutions based on the branch and bound principle [4]. The new implementation gives exactly the same results as the original algorithm. Moreover, as shown in the proposed experiments, this new approach leads to very short running times.

Sections 2 and 3 recall the DSOM algorithm based on partial sum precalculation proposed in our previous works. Section 4 presents the new algorithm based on the branch and bound principle. Sections 5 and 6 show how to complement this principle with previously proposed heuristics. The final section 7 is devoted to experiments on artificial and real world data.

2 Self-Organizing Maps for dissimilarity data

The DSOM algorithm is based on the batch version of the SOM. Let us consider N input data, $\mathcal{D} = (\mathbf{x}_i)_{1 \leq i \leq N}$, from an arbitrary input space $\mathcal{X}$. We assume

given all pairwise dissimilarities between the data, denoted $d(\mathbf{x}_i, \mathbf{x}_k)_{i,k}$. The dissimilarity is symmetric and positive (with $d(\mathbf{x}_i, \mathbf{x}_i) = 0$).

As in the standard SOM, the DSOM maps input data from an input space to a low dimensional organized set of M units (or neurons) arranged via a prior structure (generally a graph or a grid structure). However, as we make no assumption on the structure of the data, we can no longer associate an arbitrary vector model to each unit. The main difference between the DSOM and the standard SOM lies in this limitation of the former. There are two main solutions to overcome this problem. One possibility is to completely avoid using reference models, as proposed in [5]. In this paper we focus on a simpler solution, proposed in [6, 2], in which the reference model for each unit is chosen among the input data: for each unit j, it exists i such that the model associated to unit j, denoted $\mathbf{m}_j$, is located on $\mathbf{x}_i$.

As in a standard DSOM, the prior structure of the units leads to the definition of a neighborhood function, h^l, such that $h^l(u, v)$ measures the influence of unit u on unit v (and *vice versa*). One usual requirement in Kohonen algorithms is that $h^l(u, v)$ is a decreasing function of the distance between u and v in the prior structure. The DSOM is based on the batch paradigm: the superscript l used in the neighborhood function corresponds to the epoch number in the batch training, and as in the standard batch SOM, the influence on u over v tends to decrease over time.

For each epoch, the DSOM performs two operations. The first operation is the affectation of each input data to its best matching unit (BMU). The second operation is the model update. An elementary solution for the affectation is to minimize the dissimilarity between an input data and the models, i.e., to define the affectation function c^l (which maps, at epoch l, an input data index to a unit index) as follows

$$c^l(i) = \arg \min_{j \in \{1,\dots,M\}} d(\mathbf{x}_i, \mathbf{m}_j^{l-1}), \tag{1}$$

However, this criterion has a major drawback when used for the DSOM. As models are chosen among the input data, different units can share the same models [7] and there is no more a unique BMU for each input data. In order two avoid such map folding, our implementation relies on a tie breaking rule from [8]: if there are several candidate for the BMU of an input data, the comparison is extended to the models of neighbor units. While this approach gives satisfactory results, it also induces an increased running time (see section 7).

The cluster associated to unit j is denoted $\mathcal{C}_j^l = \{1 \leq i \leq N | c^l(i) = j\}$. We denote $\mathcal{P}^l$ the partition induced by the affectation phase at epoch l.

Model update is done through the principle of the generalized median exposed in [2], which is summarized in the following equation

$$\mathbf{m}_j^l = \arg \min_{\mathbf{m} \in \mathcal{D}} \sum_{i=1}^{N} h^l(c^l(i), j) d(\mathbf{x}_i, \mathbf{m}). \tag{2}$$

The new model for unit j, $\mathbf{m}_j^l$ is chosen such as to minimize the weighted sum of its dissimilarity to all the input data. Weight values reflect the prior structure of the map. In [2], variants of Equation 2 are considered: for instance, the search for the model $\mathbf{m}_j^l$ can be restricted to observations affected to units close to unit j in the map. We restrict ourselves to the presented version, but our analysis applies also to its variants.

3 Partial sum precalculation

As show in [3], the cost of the model update is $O(N^2M)$ when it is implemented with an exhaustive search over the input data. The affectation phase cost is $O(NM)$ for the criterion of Equation 1 and $O(NM^2)$ in the worst case with the solution proposed in [8]. This is smaller than $O(N^2M)$ as M is assumed to be smaller than N.

As show in [3], equation 2 can be rewritten in order to reduce the cost of the exhaustive search. We denote $D^l(u,k) = \sum_{i \in \mathcal{C}_u^l} d(\mathbf{x}_i, \mathbf{x}_k)$. We call these quantities "partial sums". The function which appears in the right hand side of equation 2 can now be re-expressed as follows:

$$S^l(j,k) = \arg\min_{\mathbf{m} \in \mathcal{D}} \sum_{i=1}^{N} h^l(c^l(i), j) d(\mathbf{x}_i, \mathbf{x}_k) = \sum_{u=1}^{M} h^l(u,j) D^l(u,k) \qquad (3)$$

As all partial sums can be computed once and for all before the computation of $S^l(j,k)$, the cost associated to the model update reduces to $O(N^2 + NM^2)$. It should be noted that this new algorithm leads to strictly identical results as those obtained by the original algorithm.

4 DSOM implementation based on the branch and bound principle

4.1 Branch and bound principle

The branch and bound principle [4] reduces the expected cost of the resolution of a minimization problem by avoiding an exhaustive search, with the help of two procedures the first one is a partition method for the solution space. This corresponds to the branching part. The second is an approximation method capable that can give quickly a lower bound for the function to be minimized on a cluster obtained by the first procedure. This corresponds to the bounding part. The algorithm proceeds by first optimizing the function on an initial cluster with an exhaustive search, obtaining this way an upper bound for the minimum. For each of the other clusters, it starts by calculating its lower bound via the approximation method and carries an exhaustive search in the cluster only if the lower bound is lower than the current upper bound. The efficiency of this approach depends on the partition, on the quality of lower bounds and on the access order of clusters.

In the DSOM context, the goal is to optimize $S^l(j,.)$, and the solution space $\mathcal{D}$ is split according to the partition $\mathcal{P}^l$ produced during the affectation phase. Because of the organization induced by the DSOM algorithm, this partition tend to have homogeneous and separated clusters after a few epoch. Therefore, lower bounds tend to be relatively high for numerous clusters and relatively low for few clusters. Exhaustive searches should therefore be avoided quite frequently.

Moreover, the prior structure of the DSOM helps to define an efficient cluster order. Indeed, good models for unit j are likely to belong to the corresponding cluster $\mathcal{C}_j^l$ or to clusters associated to units close to j in the prior structure. By browsing the clusters in increasing distance order from unit j, the algorithm will obtain quickly a good estimate of the minimum and will avoid exhaustive search in the farthest clusters (this intuition is confirmed by the experiments).

The potential impact on computation time is quite interesting, as the model update cost can be reduced from $O(NM^2)$ to $O(NM + M^2)$ in the best case (without taking into account the partial sum computation and the cost of the approximation method). Let us assume indeed an equi-distribution of observations in clusters (N/M observations per cluster) and a perfect behavior: the exhaustive search is conducted only in the first cluster and avoided in all the other (because the minimum belongs to the first cluster and the lower bound for the remaining clusters is always higher than the minimal value). Under such assumption, the update cost for unit j is $O(M(N/M))$ for the exhaustive search in the first cluster and $O(M-1)$ for the comparison with the lower bounds of the other clusters. This leads to a total model update cost of $O(MN + M^2)$.

4.2 Lower bound computation

As explained in the previous section, we need to compute a lower bound of $S^l(j,.)$ on each clusters. A very efficient strategy relies on the structure of equation 3:

$$\eta^l(j,u) = \sum_{v=1}^{M} h^l(v,j) \min_{k \in \mathcal{C}_u^l} D^l(v,k) \leq \min_{k \in \mathcal{C}_u^l} S^l(j,k) \tag{4}$$

Computing all quantities $\lambda^l(v,u) = \min_{k \in \mathcal{C}_u^l} D^l(v,k)$ costs $O(NM)$ ($O(|\mathcal{C}_u^l|)$ for one of them). This is compatible with the ideal running time of the model update phase with branch and bound ($O(MN + M^2)$ see previous section).

However the computation cost associated to lower bounds η^l is $O(M^3)$. This is a quite high overhead compared to ideal running time ($O(MN+M^2)$). Therefore we propose the following generalization:

$$\eta^l(j,u,\Theta) = \sum_{v \in \Theta} h^l(v,j)\lambda^l(v,u) \tag{5}$$

where Θ is a subset of $\{1,\ldots,M\}$. If Θ is maximal we get back to equation 4. When Θ is reduced to a singleton, the cost reduces to $O(M^2)$, which is compatible with the ideal running time of the model update phase. In practice, we focused on the particular case of $\eta^l(j,u,\{j\})$. This heuristic is motivated by the following

remark: $h^l(v,j)$ gets smaller as the distance between units v and j increases in the prior structure. Furthermore, functions used to compute $h^l(v,j)$ decrease very quickly. Therefore, $\lambda^l(j,u)$ has a large influence on $\eta^l(j,u)$.

5 Early stopping

An early stopping approach was already used in our previous works [3]. In this paper, it is combined to the branch and bound principle. The lower bound computation proposed in the previous section can be described by the following recurrent scheme:

$$\eta^l(j,u,\Theta)_1 = h^l(\theta_1,j)\lambda^l(v_1,u),$$
$$\eta^l(j,u,\Theta)_t = \eta^l(j,u,\Theta)_{t-1} + h^l(\theta_t,j)\lambda^l(\theta_t,u),$$

where $\Theta = \{\theta_1,\ldots,\theta_{|\Theta|}\}$. Each $\eta^l(j,u,\Theta)_t$ is a lower bound of $S^l(j,.)$. As the complete computation of $\eta^l(j,u,\Theta)$ is quite expensive (and sometimes useless), it can be replaced by a lazy strategy: at each step of the iteration calculation $\eta^l(j,u,\Theta)_t$ is compared to the upper bound find so far. If the upper bound is lower than $\eta^l(j,u,\Theta)_t$, the loop is stopped before termination (early stopping) and no exhaustive search is performed in the current cluster.

As in the previous section, the computation order has a major impact on the algorithm efficiency. Indeed $\eta^l(j,u,\Theta)_t$ must increase as quickly as possible in order to avoid useless comparisons. A naive approach would be to sort $h^l(\theta_t,j)\lambda^l(\theta_t,u)$ in decreasing order, and to sum them accordingly. However the overhead induced by the sorting algorithm renders this optimal order useless, and a sub-optimal order must therefore be used. We propose to sum the terms according to the decreasing order of $h^l(\theta_t,j)$. After some epoch the neighborhood function dominates the product (the neighborhood function decreases quickly) and this order is a reasonable approximation of the optimal order. Moreover, as the neighborhood function is a decreasing function of the distance in the prior structure, such order is independent of the current epoch, and can be computed during the algorithm initialization: the overhead associated to this order is therefore negligible.

6 Memorization

The last optimization presented in this paper was already applied in our previous works [3]. It applies to the computation of $D^l(u,k)$ and $\lambda^l(v,u)$, and is based on the fact that when the DSOM algorithm proceeds, clusters tend to stabilize: it is quite common for one (or more) cluster to remain identical from one epoch to the next one. In such cases, the N partial sums $D^l(u,k)$ associated to this cluster remain unchanged. In a similar way, the quantity $\lambda^l(v,u) = \min_{k\in\mathcal{C}_u^l} D^l(v,k)$ changes only if cluster $\mathcal{C}_u^l$ or cluster $\mathcal{C}_v^l$ are modified. Therefore, a lazy computation strategy can be used: we just have to monitor cluster modifications in the affectation phase, and recompute corresponding quantities accordingly.

7 Experiments

The proposed optimized algorithms have been evaluated on two data sets. The first one is a simple benchmark: it consists in a set of N vectors in $\mathbb{R}^2$ chosen randomly and uniformly in the unit square. The square euclidean distance was used to construct the dissimilarity matrix. Five values for N the number of observations, 500, 1 000, 1 500, 2 000 and 3 000 where tested.

The second data set is a real world one: it consists in a small English word list, extracted from the SCOWL [9]. The smallest list in this collection contains 4 946 very common English words. After removing plural forms and possessive forms, the word list reduces to 3 200 words. This is the first set. From this first set, a second one is constructed by applying Porter's *stemming* algorithm [10]. This reduces the word list to 2 277 words. In both cases a normalized version of the Levenshtein distance is used to compare the words [11] (string edit distance).

A DSOM with a hexagonal grid of size $M = m \times m$ models is applied to those data. We always used $L = 100$ iterations and a Gaussian kernel for the neighborhood function.

We report first in Table 1 some reference performances[3] obtained with the DSOM algorithm based on partial sum computation. We tested three different sizes for the grid, $M = 49 = 7 \times 7$, $M = 100 = 10 \times 10$, $M = 225 = 15 \times 15$ and $M = 400 = 20 \times 20$. To avoid too small clusters, high values of M were used only with high values of N.

	Artificial data					SCOWL	
N (observations) / M (clusters)	500	1 000	1 500	2 000	3 000	2 277	3 200
$49 = 7 \times 7$	0.7	1.5	2.5	3.7	6.6	4.6	8.6
$100 = 10 \times 10$	2.6	4.5	7.3	10.0	16.9	12.6	19.4
$225 = 15 \times 15$		26.6	40.9	54.4	83.2	62.5	86.4
$400 = 20 \times 20$			133.2	174.2	242.8	185.9	318.0

Table 1. Running times in seconds of partial sum algorithm

We notice first an important difference between results on artificial data and those on SCOWL data: running times on SCOWL data are slower than those on artificial data with comparable size. Such behavior can be explained by the fact that SCOWL data are much more sensitive to model collisions. As explained in Section 2, model collisions increase the cost associated to the affectation phase, which impacts overall running times. Such behavior will be identical for subsequent experiments.

Table 2 reports running time ratio between algorithms proposed in this work and the reference algorithm (the partial sum algorithm without branch and

[3] algorithms have been implemented in Java (JRE 5.0 of Sun) and tested on a workstation equipped with a 3.00 GHz Pentium IV under a Linux operating system

bound or other heuristics). The results of our previous algorithm [3] is also reported. The first part of the table focuses on the branch and bound algorithm. Lower bound computations are done according to strategies described in section 4.2: the first result in each cell corresponds to lower bounds calculated with only one term $\eta^l(j, u, \{j\})$ (see Equation 5), while the second result corresponds to lower bounds calculated according to Equation 4 (all terms are used).

		Artificial data					SCOWL	
	NM	500	1 000	1 500	2 000	3 000	2 277	3 200
Branch and bound	49	1.2 | 1.5	1.1 | 1.4	1.1 | 1.4	1.1 | 1.3	1.0 | 1.3	1.0 | 1.2	0.9 | 1.1
algorithm.	100	1.4 | 1.7	1.4 | 2.1	1.3 | 2.2	1.2 | 2.2	1.2 | 2.1	1.0 | 1.5	1.0 | 1.3
left: one term	225		2.0 | 2.7	2.0 | 3.4	2.0 | 4.0	2.0 | 4.3	1.2 | 1.8	1.2 | 2.0
right: all the terms	400			2.5 | 2.8	2.5 | 3.4	2.4 | 4.2	1.3 | 2.0	1.2 | 1.7
Early stopping	49	1.4 | 1.5	1.4 | 1.4	1.3 | 1.4	1.3 | 1.4	1.2 | 1.2	1.2 | 1.2	1.1 | 1.1
algorithm.	100	1.7 | 1.9	2.1 | 2.3	2.2 | 2.3	2.1 | 2.3	2.0 | 2.1	1.5 | 1.5	1.4 | 1.3
left: no order	225		3.1 | 4.4	3.8 | 5.0	4.2 | 5.3	4.5 | 5.3	2.0 | 2.0	2.1 | 2.2
right: order	400			3.6 | 6.2	4.1 | 6.5	5.0 | 7.1	2.4 | 2.6	1.8 | 1.9
left: memorization	49	1.9 | 1.4	1.9 | 1.4	1.9 | 1.4	1.8 | 0.9	1.8 | 1.3	1.4 | 1.2	1.3 | 1.1
algorithm.	100	2.2 | 1.7	2.9 | 1.9	2.9 | 1.7	2.9 | 1.6	2.8 | 1.5	1.6 | 1.2	1.5 | 1.1
right: previous	225		5.1 | 2.7	5.8 | 2.8	6.3 | 2.7	6.4 | 2.5	2.1 | 1.6	2.3 | 1.5
algorithm.	400			6.8 | 3.6	7.3 | 3.5	8.2 | 3.2	2.6 | 1.9	1.9 | 1.5

Table 2. Speed up of proposed algorithms (reference: partial sum algorithm)

The running times obtained with the simplest lower bound estimator (only one term) are almost always lower than those of the partial sum algorithm. This shows that even with a very optimistic lower bound, the branch and bound principle reduces a lot the search burden. However a closer analysis of the results shows that the speed up decreases as N increases. This a consequence of the domination of the $O(N^2)$ term in the total cost of the algorithm: a reduction of the actual cost of the other term $O(NM^2)$ has only a marginal effect in this situation. For the very same reasons, results get better when M increases.

Results based on a more accurate estimation of the lower bound (based on Equation 4) show a greater improvement. Despite the higher calculation cost for those bounds ($O(M^3)$ compared to $O(M^2)$), its improved accuracy is sufficient to reduce even further the need of exhaustive search and therefore the running times.

The second part of Table 2 reports running time ratio between the early stopping algorithm (when early stopping is included in the lower bound calculation) and the reference algorithm (partial sum algorithm). Left results correspond to the algorithm where summations are done according to the natural order, whereas right results correspond to the order induced by the prior structure. Both series of results show that early stopping has always a positive impact on the running time. Moreover, the ordering strategy proposed in section 5 appears to lead to significant improvements.

The last part of Table 2 reports results obtained with memorization added to the other heuristics (on the left) and results obtained by our previous algorithm (see [3]). It appears clearly that memorization reduces the running time in all cases. Benefits of memorization are also more noticeable as N increases. This can be explained by the fact that memorization reduces the precalculation phase, which is very dependent of the N term. On the other hand, memorization efficiency decreases with M. This can be explained by two reasons: firstly the representation phase is not improved by the memorization strategy, which gets more and more important in the global cost. Secondly, as the number of clusters M increases, cluster modifications increase which impair the memorization strategy. It is also clear that the newly proposed algorithm is always faster than our previous attempt.

8 Conclusion

Overall, the proposed algorithm, which combines the branch and bound principle with heuristics proposed earlier (early stopping and memorization), reduces the running time of the DSOM by a factor up to 2.5 compared to our previous solution [3], under favorable circumstances. Even on more demanding data (the SCOWL data), the gain is significant (1.5). The implementation of both algorithms are quite similar in term of code complexity and there is therefore no reason to retain the previous version. Moreover, results obtained with this new version are strictly identical to the one obtained with the original algorithm.

References

1. Kohonen, T.: Self-Organizing Maps. Third edn. Volume 30 of Springer Series in Information Sciences. Springer (1995) Last edition published in 2001.
2. Kohonen, T., Somervuo, P.J.: Self-organizing maps of symbol strings. Neurocomputing **21** (1998) 19–30
3. Conan-Guez, B., Rossi, F., El Golli, A.: Fast algorithm and implementation of dissimilarity self-organizing maps. Neural Networks **19**(6–7) (2006) 855–863
4. Land, A.H., Doig, A.G.: An automatic method for solving discrete programming problems. Econometrica **28** (1960) 497–520
5. Graepel, T., Burger, M., Obermayer, K.: Self-organizing maps: Generalizations and new optimization techniques. Neurocomputing **21** (1998) 173–190
6. Ambroise, C., Govaert, G.: Analyzing dissimilarity matrices via Kohonen maps. In: Proceedings of 5th Conference of the International Federation of Classification Societies (IFCS 1996). Volume 2., Kobe (Japan) (1996) 96–99
7. Rossi, F.: Model collisions in the dissimilarity som. In: Proc. of ESANN 2007, Bruges (Belgium) (2007)
8. Kohonen, T., Somervuo, P.J.: How to make large self-organizing maps for nonvectorial data. Neural Networks **15**(8) (2002) 945–952
9. Atkinson, K.: Spell checking oriented word lists (SCOWL). Available at URL http://wordlist.sourceforge.net/ (2004) Revision 6.
10. Porter, M.F.: An algorithm for suffix stripping. Program **14**(3) (1980) 130–137
11. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions and reversals. Sov. Phys. Dokl. **6** (1966) 707–710