A fast algorithm to find Best Matching Units in Self-Organizing Maps

Yann Bernard, Nicolas Hueber, Bernard Girau

A fast algorithm to find Best Matching Units in Self-Organizing Maps

Yann Bernard[1,2], Nicolas Hueber[1], and Bernard Girau[2]

[1] French-German Research Institute of Saint Louis, F-68300 Saint-Louis, France
[2] Université de Lorraine, CNRS, LORIA, F-54000 Nancy, France
yann.bernard@loria.fr, nicolas.hueber@isl.eu, bernard.girau@loria.fr

Abstract. Self-Organizing Maps (SOM) are well-known unsupervised neural networks able to perform vector quantization while mapping an underlying regular neighbourhood structure onto the codebook. They are used in a wide range of applications. As with most properly trained neural networks models, increasing the number of neurons in a SOM leads to better results or new emerging properties. Therefore highly efficient algorithms for learning and evaluation are key to improve the performance of such models. In this paper, we propose a faster alternative to compute the Winner Takes All component of SOM that scales better with a large number of neurons. We present our algorithm to find the so-called best matching unit (BMU) in a SOM, and we theoretically analyze its computational complexity. Statistical results on various synthetic and real-world datasets confirm this analysis and show an even more significant improvement in computing time with a minimal degradation of performance. With our method, we explore a new approach for optimizing SOM that can be combined with other optimization methods commonly used in these models for an even faster computation in both learning and recall phases.

Keywords: Self-Organizing Maps · Vector Quantization · Dimensionality Reduction.

1 Introduction

Self-organizing maps (SOM) are widely used algorithms that feature vector quantization with dimensionality reduction properties. An explanation of how they work can be found in [11]. They are used in numerous fields like image processing, automatic text and language processing, and for visualization, analysis and classification of all kinds of highly dimensional datasets. Many applications examples are depicted in [3]. However, the amount of computations required by SOMs linearly increases with the number of neurons, the number of elements in the dataset and the dimensionality of the input, in both learning and recall phases. Therefore applying SOMs on datasets with huge numbers of elements and with a high number of neurons to precisely represent the input induces a significant computational cost that may exceed some constraints such as real-time computation or low power consumption.

With the goal of reducing the required computational time of SOMs in mind, variants of the classical SOM algorithm have been developed. The most well-known SOM modification is the Batch Learning algorithm, as explained in [3]. Contrary to the classical online learning, the batch learning averages the modifications over multiple training vectors before updating the neurons weights. Similar efforts have been made in [4] or in [14]. However, all those variants are only focusing on reducing the convergence time of the SOM training. To the best of our knowledge, no work has been carried out to reduce the time required for each iteration. This can be partially explained by the highly parallel nature of the computations inside each iterations, in so far as when a fast real world implementation is required, parallel solutions are proposed, like the use of an FPGA substrate with each neuron having its own circuitry, as in [1] and in [6]. However, parallel solutions should not lead to a lack of effort in optimizing the algorithms, as the majority of SOM training is performed on CPU, and parallel hardware can be costly and difficult to program. Furthermore one can parallelise multiple iterations within an epoch instead of inside the iteration itself, and therefore can benefit from our improvements on parallel hardware.

A SOM training iteration consists of two major parts, a competitive part which searches the Best Matching Unit (or winner neuron), and a cooperative part which updates all the neurons weights proportionally to their distance with the BMU. In the classic SOM, both steps have the same algorithmic complexity (number of neurons multiplied by the dimensionality of the data) and take roughly the same time (depending on implementation details). In this paper we focus on improving the competitive part by reducing the number of neurons evaluated to find the BMU. This optimization also applies to the recall phase.

After a brief description of the standard SOM model in section 2, section 3 defines the proposed method to speed up the computation of BMU, before analyzing its computational complexity. The experimental setup is described in section 4, and the corresponding results are discussed in section 5.

2 Self-Organizing Maps

2.1 Vector quantization

Vector quantization (VQ) is a lossy source coding technique in which blocks of samples are quantized together [15]. It consists in approximating the probability density of the input space (which is split in blocks of samples) with a finite set of prototype vectors. A prototype vector is often referred to as a codeword and the set of codewords as the codebook. Many VQ techniques exist such as k-means [12], self-organizing maps (SOM) [8], neural gas (NG) [13], growing neural gas (GNG) [5]. Algorithms such as NG or GNG present good performance in terms of minimization of the quantization error which is measured by the mean squared error. However this performance is often related to the creation in the network of a very large number of new prototypes and/or connections between prototypes, thus inducing a significant increase in computational cost. On the contrary SOM are based on a static underlying topology and a fixed number of

codewords (neuron weight vectors). Moreover, in a SOM each prototype has an associated position in a map of a predefined topology (usually a 2D lattice). This spatial arrangement of the prototypes in the map makes it able to capture the topographic relationships (i.e. the similarities) between the inputs, in such a way that similar blocks of samples tend to be represented by spatially close neurons in the map. As a matter of fact, the SOM is a well-known and biologically plausible model of the topographical mapping of the visual sensors onto the cortex [16].

2.2 Kohonen SOM

The neurons of a Kohonen SOM [10] are spatially arranged in a discrete map that usually consists of a two-dimensional grid (see figure 1), or with hexagonal tiling. Each neuron n is connected the input and has a weight vector w_n, or codeword, whose dimension is equal to the size of the input vectors (or input dimension).

At the beginning of the learning algorithm, all codewords are initialized with random weights. The training of the SOM lasts several epochs. One epoch includes enough training iterations such that the whole training dataset has been used once for learning. For each training iteration, a training vector v is picked among the inputs. The best matching unit (BMU) g is then found, it corresponds to the neuron with the minimal distance (usually L^2) between w_g and v. Then the weights of all neurons are updated according to the following equation:

$$w_i(t+1) = w_i(t) + \epsilon(t) \cdot \Theta(\sigma(t), d_{i,g}) \cdot (v - w_i(t)) \tag{1}$$

where ϵ and σ are time functions (see below for details), and $d_{i,g}$ is the normalized distance between neuron i and the BMU in the map (not in the input space). Θ is a normalized centered Gaussian function with standard deviation σ.

Figure 1 illustrates how a SOM unfolds in the input space (here simply with 2D vector sizes): the codewords that are learned are shown as red points in the input space (from which random inputs are drawn, see blue points), and red links are showing the connections between the direct neighbouring neurons in the map.

$\sigma(t)$ is a parameter that influences the neighbourhood function. The higher it is, the more the BMU influences other neurons. In our experiments, we have set it to start at 0.5 and linearly decrease to a final value of 0.001, so that at the beginning of the training all neurons are significantly influenced by the BMU (unfolding the SOM), and at the end, nearly none except the BMU are (optimizing the quantization). $\epsilon(t)$ is the learning parameter, it starts at 0.6 and linearly decreases to a final value of 0.05. In our tests, we ran the SOM for 10 epochs.

An iterative batch version of SOM learning exists. Instead of using a single input vector at a time, the whole dataset (batch) is presented to the map before updating any weight [9]. This algorithm is deterministic, and the limit states of the prototypes depend only on the initial choices. However, this batch version is far less used than the above "on-line" learning version, and it can not be used

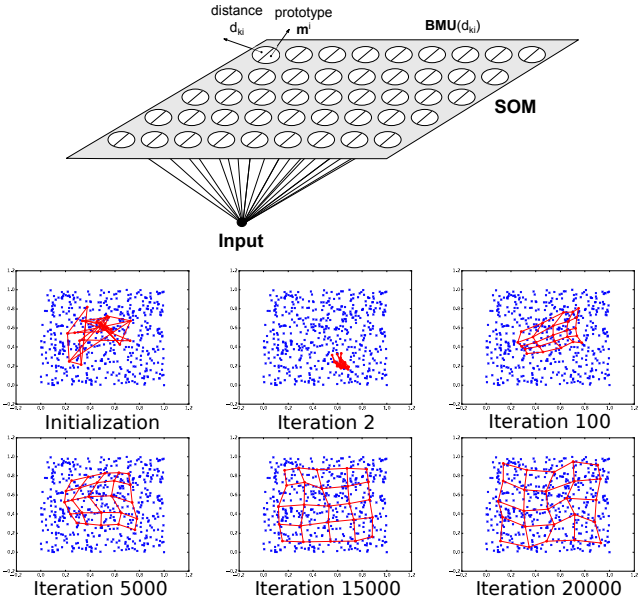

Fig. 1. Top : Typical SOM architecture. Bottom : A Kohonen SOM learning uniformly distributed random data in $[0, 1]^2$.

in the applications where the dataset dynamically evolves. Therefore we will describe and discuss our efficient method to compute the BMU in the context of the standard on-line learning algorithm.

3 Fast-BMU

3.1 Algorithm

We present here our new algorithm for searching the Best Matching Unit (BMU) in the SOM. It is traditionally determined by means of an exhaustive search by comparing all the distances between the input vector and the neurons weights. This method, while able to always find the correct minimum of the map, is computationally inefficient. The dimensionality reduction property of the SOM means that close neurons in the SOM topology represent close vectors in the input space. Therefore when an input vector is presented to the SOM, a pseudo-continuous gradient appears in the SOM when considering the distances between the neuron weights and the input vector whose minimum is located at the BMU.

In our algorithm we use this progressive reduction of distances to perform a discretized gradient descent to find the Best Matching Unit. The pseudo-code is shown in Algorithm 1, and illustrated in figure 2 (left). The algorithm starts at an arbitrarily selected position in the map. For this example let us consider that it starts at coordinates $(0, 0)$ (top left corner). This neuron has two neighbours, one

Algorithm 1 Particle

Input: *pos* : current position
Parameters: *values*: global table of distances, *eval*: global history of evaluations
Output: *bmu* : best matching unit's index

1: **if** *eval*(*pos*) is True **then**
2: **return** *pos*
3: **end if**
4: *eval*[*pos*] ← True
5: Let *new_pos* take the index of the closest not evaluated neighbour to the input vector (ignoring neighbours p such that *eval*(p) =True)
6: **if** *values*[*new_pos*] ≤ *values*[*pos*] **then**
7: **return** Particle(*new_pos*)
8: **end if**
9: **return** The index of the smallest value between *pos* and the returned indexes by execution of Particle() on all direct neighbours of *pos*.

to the east (1,0) and one to the south (0,1). We evaluate the distances to the input vector of these three neurons to find the next step in the gradient descent. If the smallest distance is found where we are currently positioned, then we consider this is a local minimum. On the contrary if one of the neighbouring neurons gives the smallest distance, then the gradient descent goes in the direction of this particular neuron, that becomes the next selected position from which we repeat this process until we find a local minimum. In our example the smallest distance is measured for the eastern neuron in (1,0). Thus the search process moves one step to the east, and this neuron has three neighbours, one to the south (1,1), one to the east (2,0) and one to the west that we come from (and thus we ignore it). We compare the three distances again, and move towards the lowest distance (south).

This process iterates until it reaches a local minimum at position (6,5), where all neighbouring neurons have a higher distance to the input vector than the selected neuron. In order to ensure that the local minimum that has been found is the best one in the local neighbourhood, we then perform a search of the local space by continuing the gradient descent from all directly neighbouring neurons that are still unexplored. If this search finds a better local minimum, then this local minimum is considered as the BMU. This part of the algorithm aims at circumventing problems that may arise from the topology. In a grid topology for instance, we can only look at orthogonal axes. But sometimes, the gradient is oriented towards a diagonal direction, and by extending the search from a local minimum, we are able to still follow this gradient towards a potential BMU.

Another problem that can arise with the particle algorithm is edge effects. When mapping high dimensional data onto a 2D map, the map sometimes become twisted no gradient between the top left neuron and the bottom right neuron. Imagine a net that is thrown onto a ball and nearly completely enveloping it: the shortest path between one corner of the net and the opposite corner of the net does not follow the mesh of the net. If in the learning phase, the

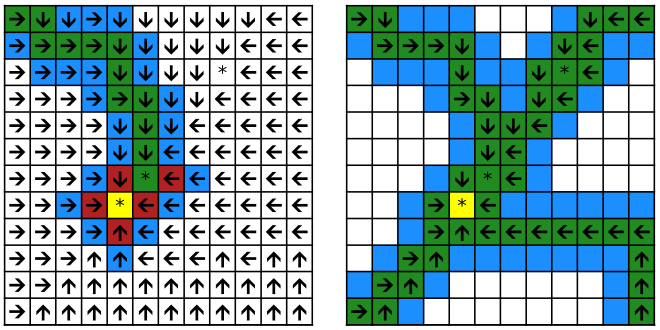

Fig. 2. Left: Example of the execution of one particle. Each cell represents one neuron. The arrow in each cell points towards the best neighbour (where a lower distance can be found). A star represents a local minimum. The green colored cells are the neurons who have been explored during the search, and the blue cells are the neurons whose distance to the input has been computed but that have not been explored by the algorithm. After having found a local minimum, new particles are created (in red) and continue the gradient descent by exploring the local neighbourhood. The cell in Yellow is the BMU, and the global minimum of the map. Right: Execution of the four particles.

particle ends up in the wrong corner of the SOM, it will find a terrible BMU and the following weight update will locally break the neighbourhood continuity, and create a negative feedback loop that will remove all dimensionality reduction properties from the SOM. Thankfully there is an easy way to avoid this, by simply starting 4 particles, one in each corner of the SOM, and selecting the smallest BMU found. This technique preserves the continuity of the SOM and makes the Fast-BMU algorithm more robust to local discontinuities in the gradient, especially at the beginning of the training process, when the map is not yet well unfolded. In the case where there is no clear gradient present in the SOM (typically when the SOM is randomly initialized before training), this algorithm will fail to find the correct BMU. However it is not a problem at the start of training as the neighborhood function will create a gradient by unfolding the SOM regardless of which neuron is initially selected. The result of a full Fast-BMU execution is shown in figure 2 (right).

3.2 Complexity analysis

The execution time of the Fast-BMU algorithm depends on the shape and size of the neural map, and on the input vector. As our algorithm relies on the dimensionality reduction property of the SOM, we will hypothesise that there is always some kind of continuity in the neural map in our analysis, because if not, our algorithm would not work anyways. In order to estimate the complexity of our algorithm, we have to estimate the number of steps that each of the four particles takes before finding a potential Best Matching Unit (BMU). We assume that: 1) all particles follow paths that do not significantly differ from shortest

XY paths until their destination (thus minimizing the number of steps), and 2) they do not stop early in some local minimum (thus maximizing the number of steps). We will call this kind of best-worst-case complexity: *Expected Complexity*. Let us consider that the BMU is located at position x, y of the SOM :

- The top left particle with coordinates $(0, 0)$ needs x steps in the width dimension, and y steps in the height dimension in order to get to the BMU.
- Similarly, the top right particle with coordinates $(w, 0)$ will take $w - x$ steps for the width and y steps for the height.
- For the bottom left $(0, h)$, it will be x and $h - y$ steps.
- For the bottom right (w, h), it will be $w - x$ and $h - y$ steps.

To get the total number of steps for one execution, we sum the steps from all particles together as shown in equation 2:

$$\text{NbSteps} = (x + y) + (w - x + y) + (x + h - y) + (w - x + h - y) \quad (2)$$

The x and y cancel out, and the resulting number of steps only depends on the width and the height of the map. The total number of steps taken by our algorithm is consequently equal to $(2w + 2h)$. We can therefore derive equation 3, which defines the expected complexity of our algorithm.

$$\mathcal{EC}(w, h) = 2 \times (w + h) \times NbrEvalPerStep \quad (3)$$

with w, h the width and height of the SOM respectively and $NbrEvalPerStep$ the number of new distances to compute on each step of a particle. Its value depends on the topology. It is at most 3 with a grid (4 neighbours minus the neuron from the previous step) and also 3 with a hexagonal topology (6 neighbours minus the neuron from the previous step and 2 neurons that were neighbours of the previous neuron and consequently have already been evaluated).

From an analytical point of view, we can estimate that our Fast-BMU algorithm ($\mathcal{EC}(w, h) = 6(w+h)$ in the worst case in a standard grid configuration) is significantly faster than the current exhaustive search algorithm ($\mathrm{O}(w, h) = wh$) when the number of neurons in the map is substantial. For instance, it is twice as fast with 24 by 24 SOMs, and 10 times faster with 120 by 120 SOMs. An experimental evaluation of the speed difference is reported in section 5.

4 Experimental Setup

To evaluate the robustness of our algorithm with various kinds of data, we have selected 6 datasets aimed at being representative of the kind of data SOMs are usually trained on. For the 2D case, we have generated data with different properties (uniform distribution and a highly non-convex shape), for 3D data we use a uniformly distributed cube shape and the pixel color values of an image. For the high dimensional case, we consider an image compression application [2] with 10 by 10 sub-images as training vectors (100 pixels with 3 colors each,

resulting in 300 dimensions), and the Free Spoken Digits Dataset [7] which uses soundwaves that we have reduced to 1000 dimensions.

Our algorithm is mostly designed for a 2 dimensional SOM, although adaptations to higher dimensions can be easily defined by just adapting the neighbourhood and the number of initial particles, even leading to possibly greater gains in computational cost. We tested it with a grid and a hexagonal shaped 2D topology, which are the most commonly used in SOM applications.

In order to evaluate the differences between all tested models, we used three metrics. The first one is the standard *Mean Squared Quantization Error* (MSQE) which estimates the Vector Quantization quality of the tested algorithm. The *Mean Squared Distance to Neurons* (MSDtN) which computes the average squared codeword distance between neurons and all of their direct neighbours. The lower this value is, the more closely related neighbouring neurons are, and the better the dimensional reduction property is. Numerous metrics exist in the SOM literature to estimate the topographical mapping of the SOM, but MSDtN has the advantage of being easy to compute, without parameters and only dependent on the neurons weights. For all metrics, lower is better.

$$\text{MSQE} = \frac{1}{V}\sum_{i=1}^{V} ||v_i - w_{\text{bmu}(i)}||^2 \tag{4}$$

$$\text{MSDtN} = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{N} \begin{cases} ||w_i - w_j||^2, & \text{if } \text{dist}(i,j) = 1 \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

With V the number of vectors in the dataset, v_i the weights of the i^{th} vector of the dataset, and $\text{bmu}(i)$ is the index of the neuron which is the best matching unit for the i^{th} vector of the dataset. Similarly N is the number of neurons and w_i the weights of the i^{th} neuron, while $\text{dist}(i,j)$ is the number of links in the shortest path between neurons i and j in the neural map.

5 Results

In this section, we explore the differences in quality of learning and recall between the standard version and our fast version for the computation of the BMU in a SOM. We also look at the practical differences in the amount of computations required for the two versions and we compare it with the previously presented complexity analysis. Quality tests were performed on a 32×32 SOM (1024 neurons). For each combination of dataset and model (choice of topology and BMU algorithm), we ran 50 executions with different random seeds which affect the datasets that are generated, the initialization of the neuron weights (who are all randomly initialised with no pre-existing gradient) and the order in which the training vectors are presented. Results are shown in table 1.

The algorithm column of the table shows the combination of BMU finding algorithm and topology that was used for training. The MSDtN (see section 4) is calculated on the trained neurons weights. The MSQE_S (Standard) is the

Table 1. Results with a 32x32 neurons SOM, averaged over 50 executions. The Algorithm column specifies with which algorithm and topology the SOM was trained, FastG and FastH stand respectively for Fast Grid and Fast Hexagonal. MSQE_S is the MSQE calculated with the standard (exhaustive) BMU algorithm whereas MSQE_F uses the Fast-BMU version. Differences of MSQE_S between different algorithms reflect the quality of the training phase. The mismatch is the proportion of BMU that are differently selected by the two algorithms.

Data	Algorithm	MSDtN	MSQE_S	MSQE_F	Mismatch
Square	Grid	1.94e−4	2.22e−4	2.22e−4	0%
	FastG	**1.93e−4**	2.23e−4	2.23e−4	0%
	Hex	2.39e−4	**2.12e−4**	**2.12e−4**	0%
	FastH	2.38e−4	2.15e−4	2.15e−4	0%
Shape	Grid	**1.38e−4**	1.40e−4	1.40e−4	>0%
	FastG	**1.38e−4**	1.40e−4	1.40e−4	>0%
	Hex	1.65e−4	**1.31e−4**	**1.31e−4**	>0%
	FastH	1.65e−4	**1.31e−4**	**1.31e−4**	>0%
Cube	Grid	**4.48e−4**	2.21e−3	2.50e−3	4.8%
	FastG	4.61e−4	2.25e−3	3.21e−3	9.8%
	Hex	5.29e−4	**2.09e−3**	**2.34e−3**	3.1%
	FastH	5.38e−4	2.11e−3	2.79e−3	7.6%
Colors	Grid	**1.15e−4**	8.64e−5	8.80e−5	4.4%
	FastG	1.19e−4	8.91e−5	9.08e−5	5.4%
	Hex	1.33e−4	8.29e−5	8.30e−5	0.4%
	FastH	1.35e−4	**8.26e−5**	**8.29e−5**	0.7%
Digits	Grid	2.02e−4	1.42e−2	1.49e−2	31.3%
	FastG	**1.93e−4**	1.44e−2	1.51e−2	32.2%
	Hex	2.29e−4	**1.41e−2**	**1.45e−2**	19.8%
	FastH	2.25e−4	1.42e−2	**1.45e−2**	13.3%
Image	Grid	**1.64e−4**	1.80e−3	1.83e−3	4.2%
	FastG	1.65e−4	1.82e−3	1.85e−3	4.4%
	Hex	1.97e−4	**1.75e−3**	1.77e−3	1.2%
	FastH	1.99e−4	**1.75e−3**	**1.76e−3**	1.2%

quantization error in the recall phase (after learning) when using the standard BMU algorithm. Comparing the different MSQE_S values for a standard version and a fast version gives an indication of the influence of the Fast-BMU algorithm on training quality only, as it always selects the real BMU in the recall phase. The MSQE_F (Fast) metric measures the vector quantization error with the BMU selection done by the Fast-BMU algorithm. If the training was performed on the standard SOM, it gives an indication of the influence of the Fast-BMU algorithm on recall accuracy only; if it was trained on a Fast version, it represents the MSQE result of a SOM that only uses the Fast-BMU algorithm. The mismatch column gives the proportion of BMU that are selected differently by the two algorithms.

5.1 Experimental results

We first observe that the distance between neurons after learning (MSDtN) is lower with the grid topology than with the hexagonal one, but this difference could be attributed to the different number of neighbours between the two topologies and therefore should only be used to compare the standard and Fast-BMU algorithms, and not topologies. We also remark that the Fast algorithm does not make any significant mismatch on the Square and Shape datasets, and therefore MSQE_S and MSQE_F have similar recall results on these datasets.

The mismatch percentages vary greatly between the datasets. From 0 to 6% for the Images and Colors Datasets, 3 to 10% for the Cube and 13 to 33% for the Spoken Digits dataset. Such differences could be explained by the distribution of the data in the datasets, as Images and Colors feature data that are closely related together. In pictures for instance, there are usually a few dominant colors with a lot of color gradients that make the continuities in the distribution easier to learn for the SOM, thus improving the performance of our Fast-BMU algorithm. The Spoken Digits dataset on the other hand has high mismatch values, which seems to indicate that a strong continuity in the neurons weights is not present after learning the SOM. The hexagonal topology also performs better with the Fast-BMU algorithm than the grid topology as mismatches are significantly lower with it. Finally the dimensionality of the dataset does not seem to play a key role here, as the Image dataset (300 dimensions) has lower mismatches than the Cube dataset (3 dimensions).

For the vector quantization part, the hexagonal topology leads again to the lowest error values. What is more surprising is that the Fast-BMU version has quantization results that are very similar to the standard version. Even with high mismatches (30% with Digits using a grid-based SOM) the MSQE is only around 5% higher, and even less when only training is compared. The only significantly higher MSQE values with Fast-BMU is with the Cube dataset where the algorithm selects quite bad BMU choices in the recall phase while being able to correctly train the SOM. In most cases, a difference in the topology of the SOM has more impact on the resulting MSQE than the use of the Fast-BMU algorithm.

5.2 Computational gains

To evaluate the computational gains that are induced by the use of the Fast-BMU algorithm independently from implementation techniques, we compared the percentage of neurons that must be evaluated in order to find the BMU. The results are shown in figure 3. The standard SOM is evaluating all neurons by definition, so the percentage is always 100%. The complexity curve for Fast-BMU plots the function defined in section 3.2. To obtain the Fast-measured curve, we ran tests with $n \times n$ SOM, where n is every even number between 10 and 50 (21 tests in total). Each test featured all datasets and all topologies (so 12 executions per test). With these results, we can observe significant improvements in the required computational time. Our algorithm is twice as fast with (16×16)

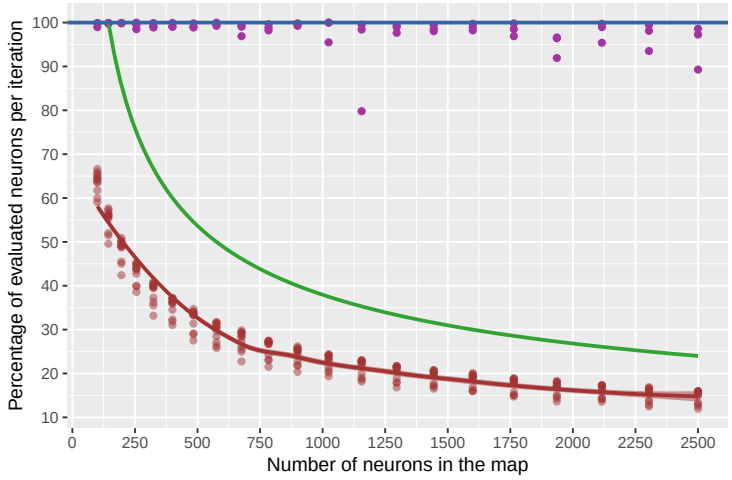

Fig. 3. Evaluation of performance gains with the number of neurons. All results were calculated on square maps. The blue line is the standard SOM, in green is the analytical value and in red the measured value. Additionally, the purple dots are the percentage of correct BMU in an execution on the Image dataset.

SOMs, four times faster with 1000 neurons (32×32). The (50×50) SOM evaluates approximately 375 neurons per iteration, which is similar to the 400 neurons a standard (20×20) SOM has to evaluate. We can also observe that the complexity curve follows a similar shape to the measured curve, while overestimating the required number of evaluated neurons by approximately 75%.

6 Conclusion

We have presented a novel method to find the Best Matching Unit in Self-Organizing Maps by taking advantage of their ability to preserve their underlying topology within the codebook. Our algorithm is significantly faster to compute while performing barely worse than the standard approach in vector quantization. This result makes SOM with a high number of neurons a more viable solution for many applications. We expect future improvements of this algorithm, or new approaches to further reduce the computational cost of SOM by modifying the BMU searching algorithm. We also study how the Fast-BMU approach can reduce the bandwidth demands in a fully parallel implementation on a manycore substrate, where all neurons are simultaneously evaluated but the BMU selection uses the simple unicast propagation of particles instead of full broadcast-reduce schemes. Finally it must be pointed out that the computational gains offered by our Fast-BMU algorithm specifically rely on preservation of neighbourhood relations when mapping the input space onto the neural map.

This property is not present when using more conventional VQ models such as k-means, so that the use of SOM could be extended to more applications where their specific mapping properties would not be useful for the application itself, but would induce potential computational gains out of reach for other models.

Acknowledgements : The authors thank the French AID agency (Agence de l'Innovation pour la Défense) for funding the DGA-2018 60 0017 contract. The code is available at github.com/yabernar/FastBMU.

References

1. Abadi, M., Jovanovic, S., Khalifa, K.B., Weber, S., Bedoui, M.H.: A scalable and adaptable hardware noc-based self organizing map. Microprocessors and Microsystems **57**, 1–14 (2018)
2. Amerijckx, C., Legat, J.D., Verleysen, M.: Image compression using self-organizing maps. Systems Analysis Modelling Simulation **43**(11), 1529–1543 (2003)
3. Cottrell, M., Olteanu, M., Rossi, F., Villa-Vialaneix, N.N.: Self-OrganizingMaps, theory and applications. Revista de Investigacion Operacional **39**(1), 1–22 (Jan 2018)
4. Fiannaca, A., Di Fatta, G., Rizzo, R., Urso, A., Gaglio, S.: Simulated annealing technique for fast learning of som networks. Neural Computing and Applications **22**(5), 889–899 (2013)
5. Fritzke, B.: A Growing Neural Gas Network Learns Topologies. In: Advances in Neural Information Processing Systems 7. pp. 625–632. MIT Press (1995)
6. Huang, Z., Zhang, X., Chen, L., Zhu, Y., An, F., Wang, H., Feng, S.: A hardware-efficient vector quantizer based on self-organizing map for high-speed image compression. Applied Sciences **7**(11), 1106 (2017)
7. Jackson, Z., Souza, C., Flaks, J., Pan, Y., Nicolas, H., Thite, A.: Jakobovski/free-spoken-digit-dataset: v1.0.8 (Aug 2018)
8. Kohonen, T.: Self-organized formation of topologically correct feature maps. Biological Cybernetics **43**(1), 59–69 (Jan 1982)
9. Kohonen, T.: The self-organizing map. Neurocomputing **21**(1–3), 1 – 6 (1998)
10. Kohonen, T.: Essentials of the self-organizing map. Neural Networks **37**, 52–65 (Jan 2013)
11. Kohonen, T., Honkela, T.: Kohonen network. Scholarpedia **2**(1), 1568 (2007)
12. MacQueen, J.: Some methods for classification and analysis of multivariate observations. The Regents of the University of California (1967)
13. Martinetz, T.M., Berkovich, S.G., Schulten, K.J.: Neural-gas network for vector quantization and its application to time-series prediction. IEEE Trans. on Neural Networks **4**(4) (1993)
14. Oyana, T.J., Achenie, L.E., Heo, J.: The new and computationally efficient mil-som algorithm: potential benefits for visualization and analysis of a large-scale high-dimensional clinically acquired geographic data. Computational and mathematical methods in medicine (2012)
15. Vasuki, A., Vanathi, P.: A review of vector quantization techniques. IEEE Potentials **25**(4), 39–47 (Jul 2006)
16. Yin, H.: The Self-Organizing Maps: Background, Theories, Extensions and Applications. In: Kacprzyk, J., Fulcher, J., Jain, L. (eds.) Computational Intelligence: A Compendium, vol. 115. Springer (2008)