XPySom: High-Performance Self-Organizing Maps

Riccardo Mancini
Scuola Superiore Sant'Anna
Pisa, Italy

Antonio Ritacco
Scuola Superiore Sant'Anna
Pisa, Italy

Giacomo Lanciano
Scuola Normale Superiore
Scuola Superiore Sant'Anna
Pisa, Italy

Tommaso Cucinotta
Scuola Superiore Sant'Anna
Pisa, Italy

***Abstract*—In this paper, we introduce *XPySom*, a new open-source Python implementation of the well-known Self-Organizing Maps (SOM) technique. It is designed to achieve high performance on a single node, exploiting widely available Python libraries for vector processing on multi-core CPUs and GP-GPUs. We present results from an extensive experimental evaluation of *XPySom* in comparison to widely used open-source SOM implementations, showing that it outperforms the other available alternatives. Indeed, our experimentation carried out using the Extended MNIST open data set shows a speed-up of about 7x and 100x when compared to the best open-source multi-core implementations we could find with multi-core and GP-GPU acceleration, respectively, achieving the same accuracy levels in terms of quantization error.**

***Index Terms*—self-organizing maps (SOMs), performance comparison, experimental evaluation, GP-GPU acceleration**

I. Introduction

There is a growing interest in deploying data-driven techniques in a number of decision-support systems. Indeed, during recent years, we have been witnessing an exponential growth in the amount of data that is made available for decision making, and a corresponding increase of the complexity of the computations carried out on such data, in order to squeeze the maximum "value" out of it. This required the development of ad-hoc software tools realizing big-data processing pipelines that make use of advanced techniques going beyond traditional statistics, relying more and more on complex computations employing Machine Learning (ML), Artificial Neural Networks (ANNs), Deep Neural Networks (DNNs), and others. Therefore, ML/ANN has become a fundamental component of the software development life-cycle, allowing for building software that does not require to be explicitly programmed in order to accomplish a task.

However, for these methods to be usable and effective in practice, an efficient implementation of the algorithms is needed, exhibiting good scalability when provided with massive real-world data sets. For instance, reducing the processing time required to deliver results is not only highly desirable to increase the efficiency of production workloads, but also allows data scientists to be quicker at implementing and evaluating new ideas. General-Purpose Graphics Processing Units (GP-GPUs) have established themselves as the go-to computing platform when it comes to accelerating ML algorithms, due to their extremely parallel and high-precision computing capabilities. It is well known that the advances in the technology behind such hardware accelerators have enabled researchers to show the disruptive effectiveness of Deep Learning (DL) in fields like Computer Vision [1], [2].

This work has received funding from the European Commission through the EU H2020 research project AMPERE (A Model-driven development framework for highly Parallel and EneRgy-Efficient computation supporting multi-criteria optimization) under the grant agreement no. 871669.

In this work, we focus on a particular ANN technology: Self-Organizing Maps (SOMs) [3]. SOMs are a kind of unsupervised, shallow, artificial neural networks, built on top of the competitive learning principle and typically employed for clustering, dimensionality reduction and high-dimensional data visualization. Indeed, they are designed for mapping high-dimensional data into a lower-dimensional space (e.g., 2D) that is better interpretable by human perception and easier to treat computation-wise, while preserving the *topology* and *distribution* of the original data at cluster-level. Given their ability to yield a data distribution in the target domain that faithfully reflects the observed relationships in the original space, SOMs have achieved remarkable results in many application fields like: image processing [4], [5], industrial data processing [6], [7], data visualization [8]–[10], pattern recognition [11], [12], anomaly detection in NFV infrastructures [13], [14].

Thanks to their simplicity, a wide variety of open-source implementations of SOMs is available. In this work, we have focused on those exposing an API to the *Python* programming language, due to its raising popularity in the ML community. Such implementations differ significantly in their performance under various scenarios. The differences can be caused by several factors like: their reliance on native vector processing and linear algebra libraries, that carry out most of the computations in *C/C++*, e.g., through *NumPy*[1]; their internal parallel architecture and exploitation of the underlying multi-core hardware; their capability to exploit GP-GPU acceleration, e.g., through CUDA[2]; or, the way several input samples are batch-processed so to parallelize computations and minimize the execution of slow Python for loops.

A. Contributions

In this paper, we focus on the parallelization and acceleration architecture of SOM implementations, performing an extensive performance comparison of widely available open-source libraries for SOM computations, namely *MiniSom* [15], *Somoclu* [16] and TensorFlow SOM [17], under various configuration options, leveraging both multi-core and GP-GPU

[1]More information at: https://numpy.org.

[2]More information at: https://developer.nvidia.com/about-cuda.

acceleration. Also, we present for the first time *XPySom*, a novel open-source SOM implementation designed to leverage existing and widely available frameworks for accelerated processing like NumPy, particularly effective when coupled with the Intel Math Kernel Library (MKL)[3] or other BLAS libraries[4] for parallel processing on multi-core CPUs, and CuPy[5] for GP-GPUs. We show that a proper design of the data processing operations, arranged so to perform a relatively small number of calls from Python to the natively accelerated libraries just mentioned, may result in a high-performance implementation that outperforms the others. Indeed, an experimental comparison carried out processing the Extended MNIST [18] data set, highlights that *XPySom* exhibits a performance that is an order of magnitude better than the other evaluated SOM implementations, both for multi-core and GPU-accelerated platforms.

B. Paper Organization

This paper is organized as follows. In Section II, we provide an overview of relevant approaches found in the research literature aiming at increasing SOM performance, mostly by parallelizing the training algorithm. In Section III, we present our own implementation, called *XPySom*, discussing the most important choices in its design regarding optimized execution and parallelization. Section IV reports the results of the benchmark we have run to compare our implementation to other widely used open-source ones. Section V includes our final remarks and possible ideas for further work on the topic.

II. Related Work

Virtually all ML algorithms can benefit – at least partially – from an implementation that exploits the parallelization capabilities of modern hardware architectures, being them classical statistical learning algorithms [19], visualization methods for high-dimensional spaces [20] or clustering techniques [21]. This work focuses on SOMs, an instance of the latter class of ML-based approaches.

SOMs make extensive use of vector operations, which can be accelerated at various levels, from leveraging the SIMD vector instructions of high-end processors like the Streaming SIMD Extensions (SSE) and Advanced Vector eXtensions (AVX) [22], to multi-core processing widely available on basically all computing platforms, to the use of general-purpose graphics processing units (GP-GPUs) featuring thousands of processing units able to compute in parallel the same kernel at high speeds exploiting the fast local memory on the GPU.

Several works appeared in the research literature dealing with SOM performance and optimizations. In [23], a parallel SOM implementation for interactive high-performance data analysis is proposed. Indeed, the classical (serial) training procedure for SOMs, that consists in determining the best-matching unit (BMU) and then updating the units weights accordingly for each training sample, is not feasible in contexts where massive datasets must be processed and results are expected to be returned in near real-time (e.g., interactive web searches). The approach is based on *(i)* partitioning the map over multiple processors, each one responsible for solving a local (minimization) problem of finding the BMU in its partition, *(ii)* aggregating the partial results to compute the global BMU and *(iii)* propagating the solution to allow for weights updates. The authors show that the introduced synchronization overhead is negligible with respect to the floating-point operations involved in the training process, that can be further optimized to leverage better L2 caches.

In [16], [24], attempts at distributing the SOM training algorithm using the MapReduce framework are described. Whereas [24] relies on a pure Spark implementation to effectively scale on massive datasets, [16] also allows for accelerating *map* and *reduce* jobs on GPUs, by leveraging on the MapReduce-MPI [25] framework[6].

In [26], a thorough scalability analysis of SOMs on a GPU cluster is reported. In particular, OpenCL- and CUDA-based single-GPU approaches, as well as a multi-GPU implementation combining CUDA and MPI, are evaluated with respect to an MPI-only baseline, considering also two different types of graphic cards. Results show that the CUDA-based implementation outperforms the OpenCL-based one, mainly due to the fact that the latter framework is designed to be compatible with a heterogeneous set of devices, and that the multi-GPU approach allows for a relatively small speed-up, because of the synchronization requirements of the training procedure.

Xiao et al. [27] highlight that the performance of pure CUDA implementations of SOMs are poor when dealing with large neighborhoods, since many weights update operations end up being serialized. To address this limitation, the authors propose an approach in which the computation of the distances between training samples and map units is implemented as a matrix multiplication with compute shader, and the weights updates are treated as a vertex rendering problem.

In [28], a heterogeneous parallel implementation based on MPI and CUDA is proposed, which is highly scalable on multiple GPUs and multiple hosts, thanks to the employed process-level and thread-level (data) parallelism. In particular, the approach leverages on the batch version of the SOM training algorithm that performs a single weights update per epoch (i.e., after all training samples are consumed by the model), whereas the original one performs a weight update after each step of an epoch (i.e., after a single training sample is consumed). The former training strategy requires fewer computations and results in faster convergence. In addition, such version of the algorithm is a perfect candidate for highly parallel implementations, because the most computationally-intensive parts can be turned into matrix operations that can be performed with cuBLAS, fully exploiting the computational capacity of GPUs.

[3]More information at: https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html

[4]More information at: http://www.netlib.org/blas/.

[5]More information at: https://cupy.chainer.org.

[6]More information at: https://mapreduce.sandia.gov/index.html.

In [29], a SOM implementation that combines both data and model parallelism is described. This implementation pushes the parallelization capability of the batch training algorithm to the extreme, as not only the training set is split in chunks to be processed independently by copies of the map (data parallelism) but each copy is also partitioned at unit weights-level, so that a separated GPU thread handles the updates of a single dimension of each neuron. According to the experiments conducted by the authors, this method performs best when dealing with large data sets but its effectiveness starts decreasing when increasing the number of features per data point. Also, by profiling the GPU usage, the authors discovered that, even though the thread processor occupancy was very close to the theoretical limit, their throughput was relatively low. This was most likely due to the time spent waiting for a CUDA core to be available, a problem that can be certainly mitigated by using a more powerful graphics card.

In [30], an efficient parallel SOM library – called *Somoclu* – is presented, which improves on [16]. Such implementation offers high flexibility, as it can be effectively executed on a single machine as well as on a cluster, both on CPUs and GPUs, and exposes interfaces compatible with widely-used data analysis ecosystems (e.g. *Python*, *R* and *MATLAB*). The *Somoclu* core implementation is based on OpenMP to achieve efficient single-node parallelism, and MPI, which replaces the MapReduce framework employed in [16], to distribute the computation among multiple nodes. Furthermore, a GPU kernel implemented with CUDA Thrust[7] and cuBLAS is available. Notably, due to its remarkable performance and versatility, we decided to use *Somoclu* as a reference for the evaluation of our approach.

Lachmair et al. [31] compared several different SOM implementations, targeting many diverse computing platforms ranging from general-purpose CPUs to FPGAs, both in terms of performance and energy efficiency. With respect to a baseline provided by the *MATLAB* SOM toolbox, a multi-threaded CPU implementation is able to achieve a speed-up of $\tilde{2}00$x with small networks. Although, when dealing with large, high-dimensional data, GPU and FPGA implementations performs best, with the latter being the most energy-efficient while reaching a speed-up of 200 with respect to the multi-threaded CPU implementation.

Among the approaches described above, only a few of them have been made widely available as open-source projects exhibiting a Python API, actively maintained and well documented with on-line examples. In the present paper, we chose to compare our proposed *XPySom* with a few projects we found with the mentioned characteristics, which include: *Somoclu* [16], due to its versatility (besides multi-core parallelism, it can also exploit GP-GPUs as well as multiple nodes in a cluster) and promising performance (at the core, it is implemented in C/C++ and exposes bindings for higher-level languages such as Python); *MiniSom* [15], due to the fact that, thanks to its simplicity, our code base was realized as a modification to it; and TensorFlow SOM [17], due to its seemingly promising approach based on the well-known *TensorFlow* framework. *MiniSom* implements the on-line algorithm described in Section III-A, exploiting exclusively vector-vector and vector-matrix operations (i.e., Euclidean distance, vector additions and scalar multiplications) which, albeit implemented efficiently in *Numpy* and – if possible – leveraging on multiple cores, are invoked several times from the Python language. Specifically, Python makes a number of calls to *Numpy* vector operations for each input sample, updating the neurons after each sample. Notice that such implementation pattern – with everything explicitly coded in Python – is among the easiest to implement and makes the code readable and easily modifiable. Unfortunately, this also results in a very poor implementation in terms of performance, as shown in Section IV. *XPySom*, on the other hand, implements the batch algorithm described in Section III-B – the same used in *Somoclu* and the compared *TensorFlow* implementation – with the improvements described in Section III-C, heavily exploiting higher-dimensional operations (e.g., matrix-matrix) to work on batches of input samples for each native call, resulting in a number of native calls proportional to the number of batches. Furthermore, neurons are updated once per epoch, with just the numerator/denominator accumulation required for each batch, which results in a much lower number of overall operations. Thanks to the additional implementation details provided in Section III-D, *XPySom* is able to exploit either *Numpy* or *CuPy* interchangeably, leveraging on either multiple CPU cores or GPU.

III. Proposed Approach

This section introduces our new *XPySom* library for Python, and it is further divided into four subsections: Section III-A introduces the classical SOM algorithm in the *on-line* formulation, used by *MiniSom* [15]; Section III-B explains how the embarrassingly parallelizable *batch* algorithm is usually derived, with reference to the *Somoclu* [16] implementation[8]; Section III-C shows the reformulated *batch* algorithm which has been adopted in *XPySom* – presented in Section III-D – that makes use of matrix-based operations to take advantage of existing *BLAS* libraries (both *CuBLAS* and *Intel MKL BLAS*).

A. On-line SOM

The SOM training algorithm aims at building a non-linear topology-preserving mapping of an input data set of N $P-$dimensional vectors $\mathbb{X} = \{x_0, x_1, ..., x_{N-1}\}, x_i \in \mathbb{R}^P, \forall i = 0..N-1$ onto a set of M neurons characterized by their $P-$dimensional weight vectors $\mathbb{W} = \{w_0, w_1, ..., w_{M-1}\}, w_k \in \mathbb{R}^P, \forall k = 0..M-1$, where the $M = G_W G_H$ neurons are usually arranged as a $G_W \times G_H$ grid (in the following, a rectangular grid is assumed). Therefore each neuron w_k has also its associated coordinates $r_k = (k \, div \, G_W, k \, mod \, G_W)$ in the 2D grid (where div and mod represent the quotient and modulus integer operations, respectively).

[7]More information at: https://docs.nvidia.com/cuda/thrust/index.html.

[8]Code available at: https://github.com/peterwittek/somoclu

The $\mathbb{W}$ neurons are first initialized in some way, usually randomly sampling the $\mathbb{X}$ data set or using the well-known Principal Components Analysis (PCA) technique. Then, for each training epoch t, an update to the SOM weights is performed for each input sample as follows. At each iteration t', an *input data* x_i is fetched (using either a random or a sequential scheduling), its associated best matching neuron b_i (usually referred to as BMU, Best Matching Unit) is found (Equation (7)), i.e. the neuron which is closer to the data point (in the following we will assume L2 distance is used). Then, the weights of *all neurons* are updated with Equation (2), where h (Equation (3)) is called *neighborhood function* and is assumed to be a Gaussian in the following.

$$b_i = argmin_k |x_i - w_k|_2 \tag{1}$$

$$\forall k,\, w_k(t'+1) = w_k(t') + \alpha(t)h(b_i,k,t)(x_i - w_k(t')) \tag{2}$$

$$h(b,k,t) = -exp\left(\frac{\|r_b - r_k\|^2}{\delta(t)}\right) \tag{3}$$

Here, $\alpha(t')$ and $\delta(t')$ are respectively the learning rate and the radius of the neighborhood function, which depend on the current epoch t ($\alpha, \delta : \mathbb{N} \to \mathbb{R}$), and decrease across epochs either linearly or exponentially, to make the algorithm converge.

B. Batch SOM

It is clear that the formulation in Section III-A is not suitable for a parallel implementation since each iteration directly depends on the one immediately before and only processes a single data sample at a time. Therefore, an embarrassingly parallel implementation has been proposed: instead of updating the neuron weights for each data sample, they are updated after a batch of N' data samples (in the following we will assume $N' = N$ for simplicity). Essentially, the term of Eq. (2) that depends on the input sample $h(b_i,k,t)x_i$ is replaced by a weighted sum of the same terms computed in parallel for all samples in the batch, using the formula:

$$\frac{\sum_i h(b_i,k,t)x_i}{\sum_i h(b_i,k,t)} \tag{4}$$

This way, one can compute in parallel all numerator and denominator parts (i.e. $h(b_i,j,t)x_i$ and $h(b_i,j,t)$, respectively) for each sample in each batch and then sum up all numerator and denominator parts and finally compute the weight update.

C. Matrix-based batch SOM

In order to take advantage of BLAS libraries which are highly optimized for execution in both GPU and CPU (in the latter making use of vector instructions), the formulas in sections Section III-A and Section III-B can be rewritten to operate on a batch of B data samples at a time arranged in a $B \times P$ matrix, $X \in \mathbb{R}^{B \times P}$. In the following the weights are assumed as arranged in a $M \times P$ matrix $W \in \mathbb{R}^{M \times P}$ We will now go through each algorithm step providing the corresponding matrix implementation: first (Section III-C1) the distance matrix $D \in \mathbb{R}^{B \times M}$ is computed, consisting of all pair-wise distances between X and W; second (Section III-C2) the BMUs are found for each sample, i.e. the index of the smallest value in each row of D as a vector $BMU \in \mathbb{N}^B$; then (Section III-C1) the neighborhood function is computed for each element of BMU, yielding a matrix $H \in \mathbb{R}^{B \times M}$; successively (Section III-C4) numerator and denominator updates are computed yielding matrix $\mathbb{NUM} \in \mathbb{R}^{M \times P}$ and vector $\mathbb{DEN} \in \mathbb{R}^M$; finally (Section III-C5), after all batches of the epochs are processed, the weights of the neurons are updated.

1) Distance matrix: There is en efficient formula for computing pair-wise distances between two matrices:

$$D^2 = X^2 - 2XW^T + W^2 \tag{5}$$

$$d^2_{i,k} = \sum_j x^2_{i,j} - \sum_j 2x_{i,j}wk,j + \sum_j w^2_{k,j} \tag{6}$$

First, we note that since we are interested only in finding the minimum over a row (i.e. for each sample in the batch) we do not need to compute the square root of D^2. Furthermore, if we express Equation (5) in terms of element-wise operations, as in Equation (6), we can note that: (1) X^2 is constant on a row (i.e. for same values of i), hence we can skip its computation since we are only interested in finding the BMU; (2) W^2 is constant over an epoch and therefore it may be computed only once and the same value reused.

2) BMU: BMUs are trivially computed element-wise: $BMU_i = argmin_k D_{i,k}$

3) Neighborhood function: By unraveling the BMU_i indices we can obtain directly the coordinates of the BMU in the rectangular grid (Equation (7)). Calling $U \in \mathbb{N}^{B \times 2}$ the matrix with the unraveled indices of the BMUs and $P \in \mathbb{N}^{M \times 2}$ the matrix with the positions r_k of the M neurons, we can compute the H matrix as follows:

$$U_{i,1} = BMU_i \, mod \, G_X,\, U_{i,2} = BMU_i \, div \, G_Y \tag{7}$$

$$h_{i,k} = -exp\left(\frac{(U_{i,1} - P_{k,1})^2 + (U_{i,2} - P_{k,2})^2}{\delta(t)}\right) \tag{8}$$

4) Numerator and denominator update: The updates N' and D' can now be computed as:

$$n'_{k,j} = \sum_i h_{i,k}x_{i,j},\, d'_k = \sum_i h_{i,k} \tag{9}$$

The corresponding matrix operation for the numerator in Equation (9) is: $N' = H^T X$ where N' and D' are accumulated in N and D.

5) Weights update: The new weights $W(t+1)$ are calculated as: $w_{k,j}(t+1) = n_{k,j}/d_k$.

D. XPySom

XPySom is our implementation of Self Organizing Maps in Python that uses the matrix-oriented formulation of the algorithm of Section III-C. *XPySom* has been obtained as a (quite disruptive) modification to *MiniSom*, which has been chosen as a starting point due to its simplicity of implementation and richness of features. In *XPySom*, the sequential sample-by-sample operations originally in *MiniSom* have been replaced with matrix operations as detailed above. These are executed using the APIs provided by either *NumPy* or *CuPy*, which

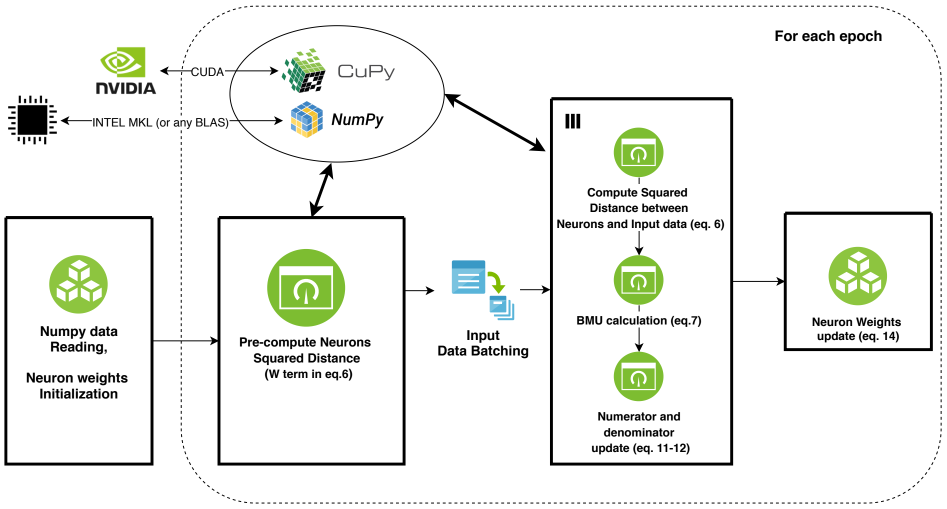

For each epoch
CUDA
CuPy
NumPy
INTEL MKL (or any BLAS)
Numpy data Reading,
Neuron weights Initialization
Pre-compute Neurons Squared Distance (W term in eq.6)
Input Data Batching
Compute Squared Distance between Neurons and Input data (eq. 6)
BMU calculation (eq.7)
Numerator and denominator update (eq. 11-12)
Neuron Weights update (eq. 14)

Fig. 1: Visualization of the data flow for SOM training in *XPySom*.

are able to exploit CPU and GPU processing, respectively. We exploit the interchangeability among *NumPy* and *CuPy*, as *CuPy* implements the same APIs as *NumPy* but it executes its operations on GPUs through cuBLAS calls or raw kernels.

The obtained data flow is exemplified in Figure 1, where the various steps XPySom goes through in order to perform a SOM training operation are visualized, in connection with the corresponding equations defined in the previous subsections. Core features of the *XPySom* architecture are its capability to perform SOM training operations in batches of input samples, arranged as matrix/vector operations that are executed very efficiently through relatively few calls to the underlying NumPy or CuPy libraries. *XPySom* is able to use either of them interchangeably, thanks to their compatible APIs.

XPySom is an open-source project and the code is available under a Creative Commons license[9]. For the moment, *XPySom* does not support execution on multiple nodes nor multiple GPUs. Moreover, when in GPU mode, *XPySom* cannot make use of CPUs to accelerate further the processing. However, the implementation of these features is planned in a short future.

IV. Experimental Results

In this section we present the results we obtained, both in terms of quantization error (QE)[10] and training time, from an extensive experimental comparison among our proposed *XPySom* and a few other commonly available SOM implementations: *MiniSom*, *SomoClu* and *TensorFlow SOM*. For the experiments, we have relied on a workstation equipped with 16GB of DDR4 memory, an Intel(R) Core(TM) i7-4790K quad-core CPU (8 hyper-threads) with base frequency 4.00 GHz (turbo-boosting to 4.40 GHz) and an Nvidia GeForce(R) GTX 1080 Ti with 11 GB of on-board memory and 3584 CUDA cores, running Ubuntu 18.04 LTS. The installed libraries and packages were Python 3.6.5, NumPy 1.18.1, *Somoclu* 1.7.5 (built from source at commit `c0f40ed`), CuPy-cuda92 7.4.0, CUDA 10.2, *TFSom*[11] and TensorFlow 2.1.0.

We have performed five different tests each comparing different training environments. To have a fair quantization error evaluation, we have chosen to run each experiment five times with a 30 minutes timeout and to use the mean quantization error as the *error metric* and mean training time as the *performance metric*. The input data used in the following experiments is the *Extended MNIST* (EMNIST) dataset [18] that contains 240000 data samples, each composed of 784 features. In all the experiments the input data values are divided by 255 so that all the features are scaled in the $[0, 1]$ range. In the *TFSom* implementation the batch size is set to 128, which shows a nice compromise in terms of memory usage and computation time. The learning rate starts every time with a value of 0.5 and decays exponentially over epochs in all the tested implementations.

Note that we have also run *XPySom* on an industrial dataset – provided by Vodafone – regarding data-center metrics for network function virtualization, that has been mentioned in our previous works [13], [14]. From a preliminary experimentation, the conclusions in terms of performance gain of *XPySom* compared to *SomoClu* are fundamentally the same as shown in this paper. Although, due to space constraints and unavailability of the aforementioned dataset to the public, we preferred not to include such results in this work.

A. Quantization error vs execution time

The first test aimed to exclude from further performance experiments all the SOM implementations that did not reach an acceptable quantization error after a fixed amount of training time or to exclude SOM initialization techniques that did not

[9]Code available at: https://github.com/Manciukic/xpysom

[10]The quantization error is defined as the average distance between each input vector and the weight vector of its associated BMU neuron.

[11]Code available at: https://github.com/cgorman/tensorflow-som

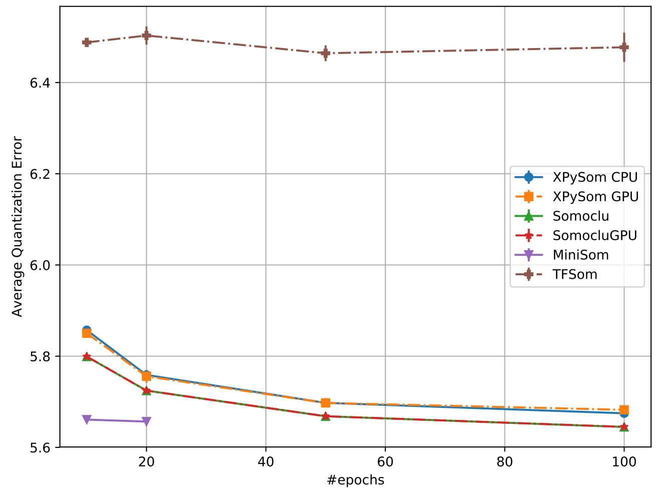


Fig. 2: Evolution of the quantization error for a 10x10 SOM throughout training epochs (EMNIST, 240000 samples, 784 features).

bring performance improvements. The quantization error is averaged over 5 training sessions and measured after each epoch for a total of 100 epochs. The number of neurons is fixed to 100 and arranged in a 10×10 rectangular grid. Since *MiniSom* offers the possibility to initialize the weights using the PCA or the random initialization, we have chosen to run the experiment with both initialization techniques while in the *Somoclu* implementation the initialization is the random one. The training update rule is the one described in Equation (4) used in both *XPySom* and *Somoclu* implementations, while in the *MiniSom* implementation the training update rule is the online update rule described in Equation (2).

Figure 2 shows how both *Somoclu* and all *XPySom* implementations reach similar mean QE through all the 100 epochs. *MiniSom* reaches a lower QE in fewer epochs just because of the online update rule that updates all the SOM weights for each sample, resulting in more updates within the same number of epochs. After 20 epochs the *MiniSom* experiment is quitted since it reaches the time limit of 30 minutes. The TensorFlow SOM implementation (*TFSom*) seems to be unable to lower the QE even after 100 epochs as it remains with a QE between the 15% and 20% higher than the other SOM implementations. PCA initialization of both *MiniSom* and *XPySom* seems to not help to lower the QE with respect to the random initialization, and the only difference seems to be in the slightly higher initialization time due to the initial PCA decomposition. For this reason, since the following experiments will focus only on the time performance, the faster random initialization will be used in all SOM implementations.

B. Execution time with different SOM grid sizes

The results obtained in Section IV-A suggest that the batch update rule used in all SOM implementations tested except the original *MiniSom* implementation does not worsen the quantization error compared to the online one. In this section, we focus on how the execution time is affected when increasing the number of SOM neurons instead. Figure 3 shows that, increasing the number of SOM neurons (on the X axis), the training time increases (in seconds on the Y axis, averaged on 5 different training sessions, in linear and logarithmic scale in Figure 3a and Figure 3b, respectively). The number of epochs is fixed to 10, the number of training samples is fixed to 240000 and the number of features is fixed to 784.

The *XPySom* implementation presented in this paper outperforms the *Somoclu* implementation in both OpenMP (CPU, just labelled as Somoclu in the figure) and CUDA (SomocluGPU label in the figure) compiled versions by two and three orders of magnitude. It is worth noticing how *XPySom* effectively leverages the GPU parallelization while *Somoclu* seems to perform worse when using the GPU. A quick investigation revealed that in this case *Somoclu* makes a non-intensive use of the GPU, while still keeping a significant amount of computations on the CPU (on a related note, the GPU and CPU kernels are kept in different files, in the source code). The *MiniSom* original implementation is not using any sort of explicit parallelization and the training time grows uncontrolled when increasing the number of neurons. The *Somoclu* GPU compiled version starts to become unusable when the number of neurons approaches the size of 500, while the TensorFlow based implementation *TFSom* starts to be faster than the *Somoclu* one when the number of neurons approaches the size of 500, but if remains two orders of magnitude slower than the *XPySom* GPU implementation.

C. Execution time with increasing number of training epochs

Figure 2 shows how increasing the number of training epochs leads to a decrease in the quantization error. Figure 4 shows how the training time increases linearly with the number of epochs. It is worth to notice that even after 100 epochs the *XPySom* GPU implementation ends the training session in less than 10 seconds, more than two orders of magnitude faster than the second faster GPU SOM implementation (*TFSom*).

D. Execution time with increasing number of training samples

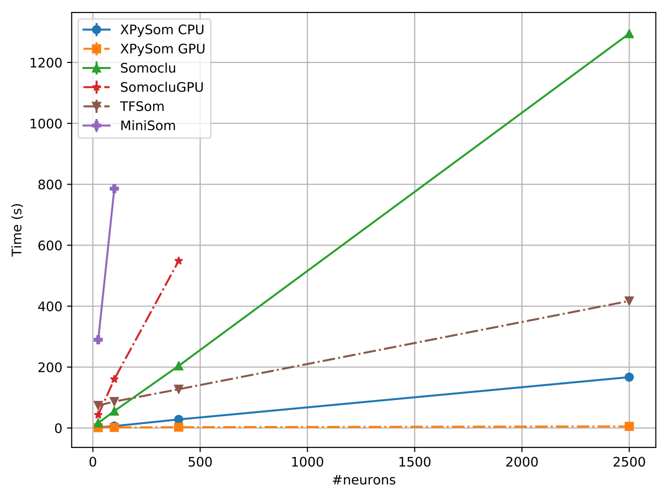
The training time increases linearly with the data set size and the log-plot in Figure 5 follows the expected behavior in all the six tested SOM implementations. Again, the *XPySom* implementation shows the best performance with both the NumPy (CPU) and CuPy (GPU) backends.

E. Execution time with increasing number of input features

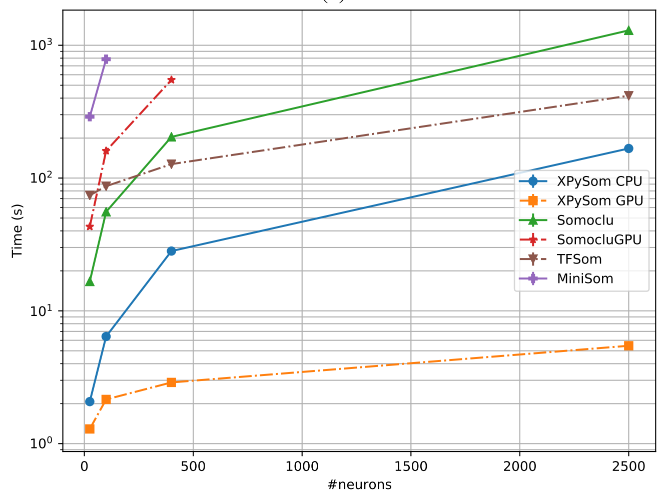
To check how the execution time is impacted when using more input features, we have scaled the original input samples to the following sizes using a bilinear interpolation over the pixel neighborhood:

- 7×7, resulting in a input dataset of size 240000×49
- 14×14, resulting in a input dataset of size 240000×196
- 28×28, resulting in a input dataset of size 240000×784
- 56×56, resulting in a input dataset of size 240000×3136

The log-plot in Figure 6 shows that the *XPySom* implementation outperforms the other SOM implementations by two or



(a)



(b)

Fig. 3: Training time as a function of the number of neurons (EMNIST, 240000 samples, 784 features).

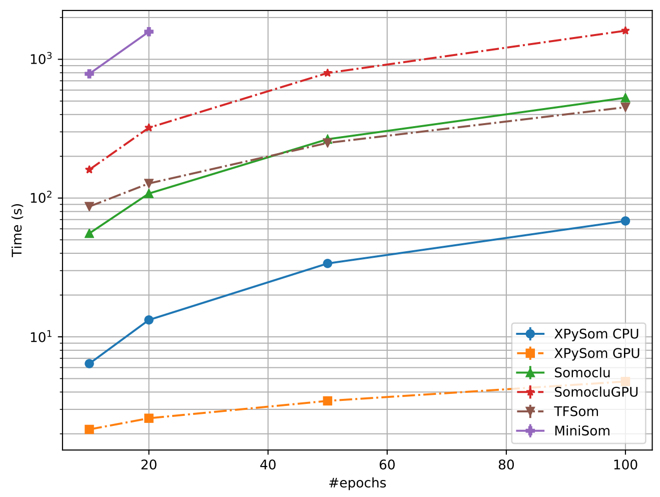


Fig. 4: Training time as a function of the number of training epochs (EMNIST, 240000 samples, 784 features)

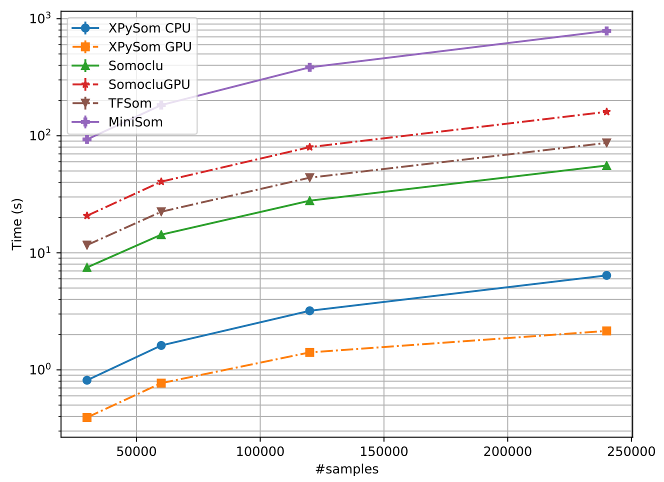


Fig. 5: Training time as a function of the number of training samples (EMNIST, 240000 samples, 784 features)

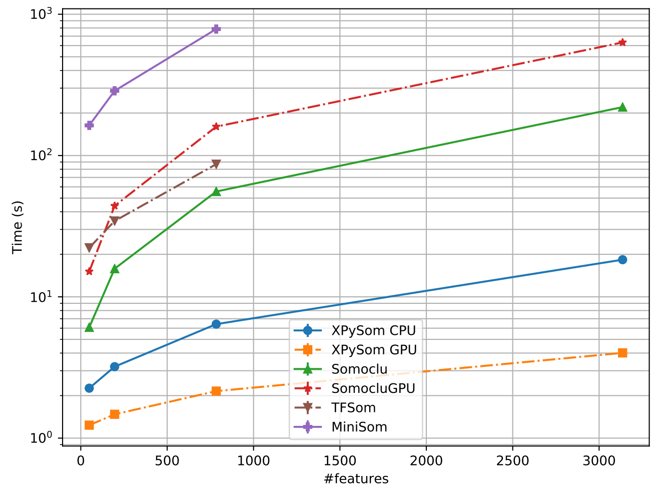


Fig. 6: Execution time vs Increasing number of training features (EMNIST, 240000 samples, 784 features)

three orders of magnitude. The number of epochs is fixed to 10 while the number of training samples is fixed to 240000. The TensorFlow implementation *TFSom* cannot start the training session with 3136 features due to a memory error (exceeded available 11Gbyte GPU memory) using a batch size of 128. On the other hand, the *MiniSom* implementation cannot reach the training session ends since it exceeded the maximum conceded training time (30 minutes) for our experiments.

V. Conclusions and Future Work

We presented *XPySom*, a variant of the popular *MiniSom* package for Python that effectively leverages the parallelization of the batch update rule for training SOMs, recurring to a massive use of matrix/vector operations optimized through the use of the well-known NumPy and CuPy libraries. We have tested our implementation on a single GP-GPU, multi-core CPU machine using different training settings. Extensive experimental results demonstrate that, even when increasing the number of neurons, the number of training samples or the number of training features, our implementation outperforms other popular open-source implementations for Python (including *Somoclu* that has a native C/C++ implementation), with a training time two or three orders of magnitude lower and a practically identical accuracy (quantization error).

Our proposed *XPySom* implementation is certainly an interesting choice when the SOM training can be run on a single machine, being probably among the fastest SOM implementations available for Python. However, a more extended evaluation of other existing open-source SOM implementations for Python is planned in the future, including for example *susi* [32], *SOMvec* [33] or others, also including in the comparison the richness of features made available by the various software packages.

Moreover, the algorithm is extremely easy to be adapted to custom implementations since most of the functions are inherited from the *MiniSom* minimalistic package.

References

[1] D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, "Flexible, High Performance Convolutional Neural Networks for Image Classification," in *Proc. 22nd International Joint Conference on Artificial Intelligence (IJCAI), Barcelona, Catalonia, Spain, July 16-22, 2011*, T. Walsh, Ed. IJCAI/AAAI, 2011, pp. 1237–1242.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[3] T. Kohonen, "The Self-Organizing Map," *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1464–1480, 1990.

[4] Y. Jiang and Z.-H. Zhou, "SOM Ensemble-Based Image Segmentation," *Neural Processing Letters*, vol. 20, no. 3, pp. 171–178, nov 2004.

[5] A. De, Y. Zhang, and C. Guo, "A parallel adaptive segmentation method based on SOM and GPU with application to MRI image processing," *Neurocomputing*, vol. 198, pp. 180–189, jul 2016.

[6] I. Díaz, M. Domínguez, A. A. Cuadrado, and J. J. Fuertes, "A new approach to exploratory analysis of system dynamics using som. applications to industrial processes," *Expert Systems with Applications*, vol. 34, no. 4, pp. 2953 – 2965, 2008.

[7] L. Canetta, N. Cheikhrouhou, and R. Glardon, "Applying two-stage SOM-based clustering approaches to industrial data analysis," *Production Planning & Control*, vol. 16, no. 8, pp. 774–784, 2005.

[8] L. Xu, Y. Xu, and T. W. Chow, "PolSOM: A new method for multidimensional data visualization," *Pattern Recognition*, vol. 43, no. 4, pp. 1668–1675, apr 2010.

[9] E. Corchado and B. Baruque, "WeVoS-ViSOM: An ensemble summarization algorithm for enhanced data visualization," *Neurocomputing*, vol. 75, no. 1, pp. 171–184, jan 2012.

[10] E. Palomo, J. North, D. Elizondo, R. Luque, and T. Watson, "Application of growing hierarchical SOM for visualisation of network forensics traffic data," *Neural Networks*, vol. 32, pp. 275–284, aug 2012.

[11] T. Yamagutchi, K. Nagata, and P. Q. Truong, "Pattern Recognition of EEG Signal during Motor Imagery by Using SOM," in *Second International Conference on Innovative Computing, Informatio and Control (ICICIC 2007)*. IEEE, sep 2007, pp. 121–121.

[12] T.-S. Li and C.-L. Huang, "Defect spatial pattern recognition using a hybrid SOM–SVM approach in semiconductor manufacturing," *Expert Systems with Applications*, vol. 36, no. 1, pp. 374–385, jan 2009.

[13] G. Lanciano, A. Ritacco, T. Cucinotta, M. Vannucci, A. Artale, L. Basili, E. Sposato, and J. Barata, "SOM-based behavioral analysis for virtualized network functions," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, ser. SAC '20. New York, NY, USA: ACM, mar 2020, pp. 1204–1206.

[14] T. Cucinotta, G. Lanciano, A. Ritacco, M. Vannucci, A. Artale, J. Barata, E. Sposato, and L. Basili, "Behavioral Analysis for Virtualized Network Functions: A SOM-based Approach," in *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2020, pp. 150–160.

[15] G. Vettigli, "MiniSom," 2019. [Online]. Available: https://github.com/JustGlowing/minisom

[16] P. Wittek and S. Daranyi, "A GPU-Accelerated Algorithm for Self-Organizing Maps in a Distributed Environment," in *Proceedings of ESANN-12, 20th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, 2012.

[17] C. Gorman, "TensorFlow Self-Organizing Map," 2019.

[18] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik, "EMNIST: an extension of MNIST to handwritten letters," *CoRR*, vol. abs/1702.05373, 2017. [Online]. Available: http://arxiv.org/abs/1702.05373

[19] T. V. Sang, R. Kobayashi, R. S. Yamaguchi, and T. Nakata, "Accelerating Solution of Generalized Linear Models by Solving Normal Equation Using GPGPU on a Large Real-World Tall-Skinny Data Set," in *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, vol. 2019-Octob. IEEE, oct 2019, pp. 112–119.

[20] D. M. Chan, R. Rao, F. Huang, and J. F. Canny, "T-SNE-CUDA: GPU-Accelerated T-SNE and its Applications to Modern Data," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, sep 2018, pp. 330–338.

[21] M. A. Souza, L. A. Maciel, P. H. Penna, and H. C. Freitas, "Energy Efficient Parallel K-Means Clustering for an Intel® Hybrid Multi-Chip Package," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, sep 2018, pp. 372–379.

[22] "Intel Whitepaper: Optimizing Performance with Intel Advanced Vector Extensions," 2014. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/performance-xeon-e5-v3-advanced-vector-extensions-paper.pdf

[23] A. Rauber, P. Tomsich, and D. Merkl, "parSOM: a parallel implementation of the self-organizing map exploiting cache effects: making the SOM fit for interactive high-performance data analysis," in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, vol. 6. IEEE, 2000, pp. 177–182 vol.6.

[24] T. Sarazin, H. Azzag, and M. Lebbah, "SOM Clustering Using Spark-MapReduce," in *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE, may 2014, pp. 1727–1734.

[25] S. J. Plimpton and K. D. Devine, "Mapreduce in mpi for large-scale graph algorithms," *Parallel Comput.*, vol. 37, no. 9, p. 610–632, Sep. 2011.

[26] S. McConnell, R. Sturgeon, G. Henry, A. Mayne, and R. Hurley, "Scalability of Self-organizing Maps on a GPU cluster using OpenCL and CUDA," *Journal of Physics: Conference Series*, vol. 341, p. 012018, feb 2012.

[27] Y. Xiao, R.-B. Feng, Z.-F. Han, and C.-S. Leung, "GPU Accelerated Self-Organizing Map for High Dimensional Data," *Neural Processing Letters*, vol. 41, no. 3, pp. 341–355, jun 2015.

[28] Y. Liu, J. Sun, Q. Yao, S. Wang, K. Zheng, and Y. Liu, "A Scalable Heterogeneous Parallel SOM Based on MPI/CUDA," in *Proceedings of The 10th Asian Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Zhu and I. Takeuchi, Eds., vol. 95. PMLR, may 2018, pp. 264–279.

[29] T. Richardson and E. Winer, "Extending parallelization of the self-organizing map by combining data and network partitioned methods," *Advances in Engineering Software*, vol. 88, pp. 1–7, oct 2015.

[30] P. Wittek, S. C. Gao, I. S. Lim, and L. Zhao, "somoclu : An Efficient Parallel Library for Self-Organizing Maps," *Journal of Statistical Software*, vol. 78, no. 9, jun 2017.

[31] J. Lachmair, T. Mieth, R. Griessl, J. Hagemeyer, and M. Porrmann, "From CPU to FPGA — Acceleration of self-organizing maps for data mining," in *2017 International Joint Conference on Neural Networks (IJCNN)*, vol. 2017-May. IEEE, may 2017, pp. 4299–4308.

[32] F. M. Riese, "SuSi: SUpervised Self-organIzing maps in Python," 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2609130

[33] G. Clark, "SOMvec," 2020. [Online]. Available: https://bitbucket.org/GeoffreyClark/somvec/src/master/