The Journal of Open Source Software

aweSOM: a CPU/GPU-accelerated Self-organizing Map and Statistically Combined Ensemble Framework for Machine-learning Clustering Analysis

Trung Ha [1,2,3¶], **Joonas Nättilä** [4,5,2], **and Jordy Davelaar** [6,7,2,5]

1 Department of Astronomy, University of Massachusetts-Amherst, Amherst, MA 01003, USA **2** Center for Computational Astrophysics, Flatiron Institute, 162 Fifth Avenue, New York, NY 10010, USA **3** Department of Physics, University of North Texas, Denton, TX 76203, USA **4** Department of Physics, University of Helsinki, P.O. Box 64, University of Helsinki, FI-00014, Finland **5** Physics Department and Columbia Astrophysics Laboratory, Columbia University, 538 West 120th Street, New York, NY 10027, USA **6** Department of Astrophysical Sciences, Peyton Hall, Princeton University, Princeton, NJ 08544, USA **7** NASA Hubble Fellowship Program, Einstein Fellow ¶ Corresponding author

DOI: 10.21105/joss.07613

Software

- Review
- Repository
- Archive

Editor: George K. Thiruvathukal

Reviewers:

- @apizzuto
- @Abinashbunty

Submitted: 14 October 2024
Published: 30 March 2025

Summary

We introduce aweSOM, an open-source Python package for machine learning (ML) clustering and classification, using a Self-organizing Maps (SOM, Kohonen, 1990) algorithm that incorporates CPU/GPU acceleration to accommodate large ($N > 10^6$, where N is the number of data points), multidimensional datasets. aweSOM consists of two main modules, one that handles the initialization and training of the SOM, and another that stacks the results of multiple SOM realizations to obtain more statistically robust clusters.

Existing Python-based SOM implementations (e.g., POPSOM, Yuan (2018); MiniSom, Vettigli (2018); sklearn-som) primarily serve as proof-of-concept demonstrations, optimized for smaller datasets, but lacking scalability for large, multidimensional data. aweSOM provides a solution for this gap in capability, with good performance scaling up to $\sim 10^8$ individual points, and capable of utilizing multiple features per point. We compare the code performance against the legacy implementations it is based on, and find a $10 - 100\times$ speed up, as well as significantly improved memory efficiency, due to several built-in optimizations.

As a companion to this paper, Ha et al. (2024) demonstrates the capabilities of aweSOM in analyzing the physics of plasma turbulence. Detailed instructions on how to install, test, and replicate the results of the paper are available in the online documentation. Also included in the documentation is an example of applying aweSOM to the Iris dataset (Fisher, 1936).

Statement of need

The self-organizing map algorithm

A SOM algorithm is an unsupervised ML technique that excels at dimensionality reduction, clustering, and classification tasks. It consists of a 2-dimensional (2D) lattice of nodes. Each node contains a weight vector that matches the dimensionality of the input data. A SOM performs clustering by adapting the weight vectors of nodes, progressively reshaping the lattice's topology to match the intrinsic clustering of the input data. In this manner, a SOM lattice can capture multidimensional correlations in the input data.

SOM is commonly used in various real-world applications, such as in the financial sector (e.g., Alshantti & Rasheed, 2021; Pei et al., 2023), in environmental surveys (e.g., Alvarez-Guerra et

al., 2008; Li et al., 2020), in medical technology (e.g., Hautaniemi et al., 2003; Kawaguchi et al., 2024), among others. aweSOM is originally developed to be used in analyzing astrophysical simulations, but can be applied to a wide variety of real-world data.

POPSOM

We base the SOM module of aweSOM on POPSOM (Hamel, 2019; Yuan, 2018), a R-based SOM model. POPSOM was developed as a single-threaded, stochastic training algorithm with built-in visualization capabilities. However, due to its single-threaded nature, the algorithm does not scale well with large datasets. When $N \gtrsim 10^6$, POPSOM is often unable to complete the training process as the dimensionality of the input data increases due to its much higher memory usage. As an example, we generated a mock dataset with $N = 10^6$ and $F = 6$ dimensions, then trained it on a lattice of $X = 63$, and $Y = 32$, where X, Y are the dimensions of the lattice, using one Intel Icelake node with 64 cores and 1 TB memory. POPSOM completed the training in ≈ 2200 s and consumed ≈ 600 GB of system memory at its peak.

Rewriting POPSOM into aweSOM

To combat the long training time and excessive memory usage, we rewrite POPSOM with multiple optimizations/parallelizations. We replaced legacy code with modern NumPy functions for updating the lattice (a 3D array) and eliminated the use of pandas DataFrames (The pandas development team, 2024), which consume significantly more memory. The weight vector modifications in the DataFrame were also less efficient compared to the NumPy arrays used in aweSOM. Furthermore, for the steps where parallelization could be leveraged (such as when the cluster labels are mapped to the lattice, then to the input data), we integrate Numba (Lam et al., 2015) to take advantage of its Just-In-Time (JIT) compiler and simple parallelization of loops. In the same example as above, aweSOM took ≈ 200 s and consumed ≈ 450 MB of memory to complete the training and clustering. In addition to the $\sim 10\times$ speedup, aweSOM is also $\sim 10^3\times$ more memory-efficient.

The left hand side of Figure 1 shows a graph of the performance between aweSOM and the legacy POPSOM implementation over a range of N and F, performed on one Intel Icelake compute node with 64 CPU cores and 1 TB memory. While POPSOM initially performs slightly faster than aweSOM for $N \lesssim 10^4$, this changes when N exceeds 5×10^5, after that aweSOM consistently outperforms POPSOM by approximately a factor of 10. Critically, POPSOM fails to complete its clusters mapping for $N \gtrsim 10^6, F > 4$ because the memory buffer of the test node was exceeded.

The statistically combined ensemble method

The statistically combined ensemble (SCE) method was developed by Bussov & Nättilä (2021) to stack the result of multiple independent clustering realizations into a statically significant set of clusters. This method represents a form of ensemble learning. Additionally, SCE can also be used independently from the base SOM algorithm, and is compatible with any general unsupervised classification algorithm.

The legacy SCE implementation

In its original version, the SCE was saved as a nested dictionary of boolean arrays, each of which contains the spatial similarity index g between cluster C and cluster C'. The total number of operations scales as N_C^R, where N_C is the number of clusters in each realization, and R is the number of realizations. For example, in our use case involving plasma simulation data (Ha et al., 2024), each SOM realization produces on average 7 clusters, and the SCE analysis incorporates 36 realizations, resulting in approximately $7^{36} \sim 10^{30}$ array-to-array comparisons.

Integrating SCE into aweSOM with JAX

To mitigate this bottleneck, we rewrite the legacy SCE code with JAX (Bradbury et al., 2018) to significantly enhance the performance of array-to-array comparisons (which are matrix multiplications) by leveraging the GPU's parallel-computing advantage over the CPU. We implement this optimization by replacing the original nested dictionaries with data arrays. Then, every instance of matrix operation using NumPy is converted to `jax.numpy`. Additionally, we implement internal checks such that the SCE code automatically reverts to NumPy if GPU-accelerated JAX is not available.

Similar to the SOM implementation, the SCE implementation in aweSOM demonstrates excellent scalability as the number of data points increases. The right hand side of Figure 1 shows a graph of the performance between the two implementations given $R = 20$. At $N < 5 \times 10^4$, the legacy code is faster due to the overhead from loading JAX and the JIT compiler. However, aweSOM quickly exceeds the performance of the legacy code, and begins to approach its maximum speed-up of $\sim 100\times$ at $N \gtrsim 10^7$ (performed on one NVIDIA A100-40GB GPU). On the other hand, when running on CPU-only with NumPy, aweSOM consistently shows a $2\times$ speed improvement over the legacy code. Altogether, it is best to use aweSOM with Numpy when $N \lesssim 10^5$, and with JAX when $N \gtrsim 10^5$.

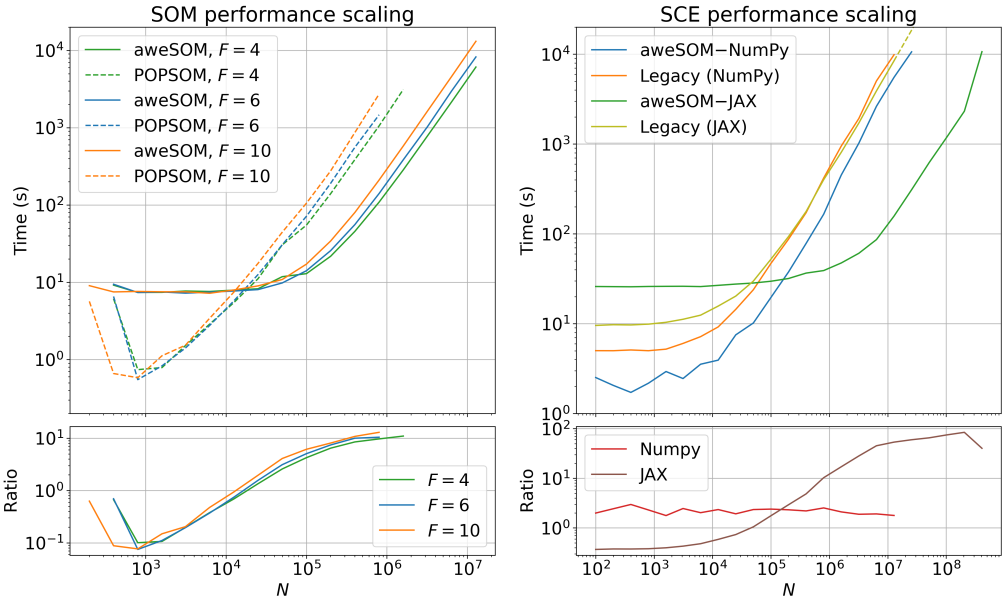

Figure 1: Performance scaling for aweSOM vs. the legacy SOM (left) and SCE (right) implementation. The top panels show the time for each implementation to complete analysis of N number of data points. On the right panel, the dotted line extending from the olive line shows linear extrapolations from the data in order to estimate the speedup. The bottom panels show the ratio between the time taken by the legacy code divided by the time taken by aweSOM. In the SOM analysis, we consider a dataset with $F = 6$ and $F = 10$ dimensions. In the SCE analysis, we test the scaling of both a GPU-accelerated implementation (with JAX) and a CPU-only implementation (with NumPy).

Acknowledgements

The authors would like to thank Kaze Wong for the valuable input in setting up JAX for the SCE analysis. The authors would also like to thank Shirley Ho and Lorenzo Sironi for the useful discussions. TH acknowledges support from a pre-doctoral program at the Center for Computational Astrophysics, which is part of the Flatiron Institute. JN is supported by an ERC grant (ILLUMINATOR, 101114623). JD is supported by NASA through the NASA Hubble Fellowship grant HST-HF2-51552.001-A, awarded by the Space Telescope Science Institute,

which is operated by the Association of Universities for Research in Astronomy, Incorporated, under NASA contract NAS5-26555. aweSOM was developed and primarily run at facilities supported by the Scientific Computing Core at the Flatiron Institute. Research at the Flatiron Institute is supported by the Simons Foundation.

References

Alshantti, A., & Rasheed, A. (2021). Self-organising map based framework for investigating accounts suspected of money laundering. *Frontiers in Artificial Intelligence*, *4*. https://doi.org/10.3389/frai.2021.761925

Alvarez-Guerra, M., González-Piñuela, C., Andrés, A., Galán, B., & Viguri, J. R. (2008). Assessment of self-organizing map artificial neural networks for the classification of sediment quality. *Environment International*, *34*(6), 782–790. https://doi.org/10.1016/j.envint.2008.01.006

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs* (Version 0.3.13). http://github.com/google/jax

Bussov, M., & Nättilä, J. (2021). Segmentation of turbulent computational fluid dynamics simulations with unsupervised ensemble learning. *Signal Processing: Image Communication*, *99*, 116450. https://doi.org/10.1016/j.image.2021.116450

Fisher, R. A. (1936). *Iris*. UCI Machine Learning Repository. https://doi.org/10.24432/C56C76

Ha, T., Nättilä, J., Davelaar, J., & Sironi, L. (2024). *Machine-learning characterization of intermittency in plasma turbulence: Single and double sheet structures*. https://arxiv.org/abs/2410.01878

Hamel, L. (2019). VSOM: Efficient, Stochastic Self-organizing Map Training. In K. Arai, S. Kapoor, & R. Bhatia (Eds.), *Intelligent Systems and Applications* (pp. 805–821). Springer International Publishing. https://doi.org/10.1007/978-3-030-01057-7_60

Hautaniemi, S., Yli-Harja, O., Astola, J., Kauraniemi, P., Kallioniemi, A., Wolf, M., Ruiz, J., Mousses, S., & Kallioniemi, O.-P. (2003). Analysis and visualization of gene expression microarray data in human cancer using self-organizing maps. *Machine Learning*, *52*(1–2). https://doi.org/10.1023/A:1023941307670

Kawaguchi, T., Ono, K., & Hikawa, H. (2024). Electroencephalogram-based facial gesture recognition using self-organizing map. *Sensors*, *24*(9). https://doi.org/10.3390/s24092741

Kohonen, T. (1990). The self-organizing map. *Proceedings of the IEEE*, *78*(9), 1464–1480. https://doi.org/10.1109/5.58325

Lam, S. K., Pitrou, A., & Seibert, S. (2015). Numba: A LLVM-based python JIT compiler. *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. https://doi.org/10.1145/2833157.2833162

Li, K., Sward, K., Deng, H., Morrison, J., Habre, R., Chiang, Y.-Y., Ambite, J.-L., Wilson, J., & Eckel, S. (2020). *Using dynamic time warping self-organizing maps to characterize diurnal patterns in environmental exposures*. https://doi.org/10.21203/rs.3.rs-87487/v1

Pei, D., Luo, C., & Liu, X. (2023). Financial trading decisions based on deep fuzzy self-organizing map. *Applied Soft Computing*, *134*, 109972. https://doi.org/10.1016/j.asoc.2022.109972

The pandas development team. (2024). *Pandas-dev/pandas: pandas* (Version v2.2.3). Zenodo. https://doi.org/10.5281/zenodo.13819579

Vettigli, G. (2018). *MiniSom: Minimalistic and NumPy-based implementation of the self organizing map*. https://github.com/JustGlowing/minisom/

Yuan, L. (2018). *Implementation of Self-Organizing Maps with Python* [University of Rhode Island]. https://doi.org/10.23860/thesis-yuan-li-2018