Journal of Machine Learning Research 27 (2026) 1-17 Submitted 10/25; Revised 7/26; Published 7/26

torchsom: The Reference PyTorch Library for Self-Organizing Maps

Louis Berthier[1,2] LOUIS-DESIRE-ROMEO.BERTHIER@MICHELIN.COM
Ahmed Shokry[1] AHMED.SHOKRY@POLYTECHNIQUE.EDU
Maxime Moreaud[2] MAXIME.MOREAUD@MICHELIN.COM
Guillaume Ramelet[2] GUILLAUME.RAMELET@MICHELIN.COM
Eric Moulines[1] ERIC.MOULINES@POLYTECHNIQUE.EDU

[1] *Centre de Mathématiques Appliquées, Ecole Polytechnique, 91120 Palaiseau, France*

[2] *Manufacture Française des Pneumatiques Michelin, 63100 Clermont-Ferrand, France*

Editor: Alexandre Gramfort

Abstract

This paper introduces `torchsom`, an open-source Python library that provides a reference implementation of the Self-Organizing Map (SOM) in `PyTorch`. This package offers three main features: (i) dimensionality reduction, (ii) clustering, and (iii) friendly data visualization. It relies on a `PyTorch` backend, enabling (i) fast and efficient training of SOMs through GPU acceleration, and (ii) easy and scalable integration with the `PyTorch` ecosystem. `torchsom` also follows the `scikit-learn` API for ease of use and extensibility. The library is released under the `Apache 2.0` license with 90% test coverage, and its source code and documentation are available at `https://github.com/michelin/TorchSOM`.

Keywords: self-organizing maps, pytorch, unsupervised learning, dimensionality reduction, clustering

1. Introduction

Self-Organizing Maps (SOMs) remain a valuable and enduring technique in modern machine learning (ML) and data analytics, despite being introduced decades ago (Kohonen, 1982, 1990, 2001). This is because SOMs integrate key mechanisms at low computational cost, including: (i) abstraction of high-dimensional data with dimensionality reduction, (ii) preservation of latent nonlinear topological structures, and (iii) visual interpretability. Such properties make SOMs a valuable asset for exploratory analysis, explainable AI workflows, and resource-constrained environments.

These properties have driven the extensive adoption of SOMs in a broad range of domains, such as energy industry (Raj et al., 2023; Concetti et al., 2023; Dash et al., 2024), biology and health (Hao et al., 2021; Farzamnia et al., 2023; Weber et al., 2023), IoT systems (Khan and Mailewa, 2023; Gad et al., 2024), chemical and environmental applications (Maltarollo et al., 2013; Feng et al., 2023; Xiang et al., 2022; Mia et al., 2023; Zhang et al., 2023; Licen et al., 2023), and business cases (Bloom, 2005; Bowen and Siegler, 2024). SOMs can also be periodically updated with new data, progressively improving their representa-

	torchsom (Berthier, 2025)	MiniSom (Vettigli, 2018)	SimpSOM (Comitani, 2017)	SOMPY (Moosavi et al., 2014)	somoclu (Wittek, 2017)	som-pbc (Müller, 2018)
Technical Architecture						
Framework	PyTorch	NumPy	NumPy	NumPy	C++	NumPy
GPU Acceleration	CUDA PyTorch	✗	CuPy/cuML	✗	CUDA C++	✗
JIT Compilation	via PyTorch	Numba[1]	✗	✗	✗ (AOT)[2]	✗
API Design	scikit-learn	Custom	Custom	MATLAB	Custom	Custom
Development						
Maintenance	Active	Active	Minimal	Minimal	Minimal	✗
Documentation	Rich	Basic[3]	Basic	✗	Basic	Basic
Test Coverage	90%	98%	53%	✗	Minimal	✗
PyPI Distribution	✓	✓	✓	✗	✓	✗
Functional Capabilities (built-in)						
Visualization	Advanced	✗	Moderate	Moderate	Basic	Basic
Clustering	✓	Examples only[4]	✓	✗	✗	✗
Just-In-Time Learning (JITL) support	✓	✗	✗	✗	✗	✗
SOM Variants	Multiple	✗	PBC	✗	PBC	PBC
Extensibility	High	Moderate	Low	Low	Low	Low

Table 1: Comparative analysis of Python SOM libraries available on GitHub.

tions and supporting applications such as industrial monitoring and control (Zhang et al., 2018; Zheng et al., 2018; Jin et al., 2020; Urhan and Alakent, 2020; Zhang et al., 2022).

Despite this relevance, only a few Python libraries attempt to bridge SOMs with modern ML workflows, such as integration with `PyTorch` (Ansel et al., 2024) or a `scikit-learn` interface (Pedregosa et al., 2011). However, these limited implementations are often outdated and poorly maintained, and they lack (i) GPU acceleration, (ii) integration with modern Deep Learning (DL) frameworks, and (iii) user-friendly APIs with visualization capabilities. Consequently, the Python SOM ecosystem still suffers from significant gaps, hindering reproducible and scalable SOM-based analyses.

To overcome these challenges, we introduce `torchsom`, to the best of our knowledge the most complete library for SOMs built on the `PyTorch` ecosystem. `torchsom` integrates GPU acceleration, SOM variants, clustering tools, and a familiar `scikit-learn`-style API, complemented by user-friendly visualization tools. Our library is designed to integrate into ML workflows, combining modern DL frameworks with sound software engineering practices, while improving efficiency and strengthening the visual interpretability that distinguishes SOMs, whose underlying principles are summarized in Appendix B.

2. Related Work

In the Python ecosystem, several libraries provide SOM implementations. However, they differ in their technical architecture, development, and capabilities as shown in Table 1.

While existing libraries address specific use cases (`MiniSom` offers a minimalist `NumPy`-based implementation suited to education and prototyping, and `somoclu` targets HPC environments through CUDA C++), `torchsom` is the only library in this comparison that combines a native `PyTorch` backend with GPU acceleration, a `scikit-learn`-compatible

1. Opt-in Numba (Lam et al., 2015) JIT (`train_batch_offline_fast`); distinct from *Just-In-Time Learning* (JITL, below), which is a supervised local-modeling capability rather than a compilation technique.
2. `somoclu`'s kernels are ahead-of-time compiled with `nvcc` at build time.
3. Example notebooks and partial in-code docstrings; no narrative documentation site (e.g. comparable to `https://opensource.michelin.io/TorchSOM`).
4. Clustering is not a built-in `MiniSom` feature; it requires user-supplied code on top of `MiniSom` primitives.

API, an advanced built-in visualization suite, a built-in clustering interface, JITL support, and multiple grid topologies with configurable neighborhood retrieval modes within a single modular codebase. It is further supported by a published narrative documentation site (`https://opensource.michelin.io/TorchSOM`) and a community-oriented development process, making `torchsom` a complete and scalable reference for both research and production.

3. Package Architecture

The `torchsom` library follows a modular design built around three core components that provide a complete SOM implementation with native `PyTorch` integration, namely:

1. `torchsom.core`:
   This module implements classical SOM algorithms (Kohonen, 1990) in the `PyTorch` ecosystem for integration into DL workflows. The core classes provide (i) a `fit()` supporting automatic GPU acceleration for model training, (ii) a `cluster()` for partitioning the trained map, (iii) a `build_map()` for generating maps suitable for visualization, and (iv) a `collect_samples()` for identifying informative samples from grid topology and latent-space distances. The last of these offers three configurable retrieval modes: `bmu_only`, `bmu_neighborhood`, and `bmu_neighborhood_knn`. Best Matching Unit (BMU) selection is delegated to a configurable backend (the `search_backend` argument), which by default selects automatically between a `PyTorch` brute-force implementation and an optional `FAISS` backend that accelerates nearest-neighbor search for large maps and high-dimensional inputs.

2. `torchsom.utils`:
   This module provides essential components for SOM parameterization and training, including decay functions for learning rate and neighborhood width scheduling. It supports multiple distance metrics, with Euclidean and cosine distances commonly used for latent space calculations and BMU selection. Multiple neighborhood kernels are implemented, with the Gaussian kernel serving as the primary example for weight updates around the BMU. Both rectangular and hexagonal grid topologies are supported, each optionally combined with periodic boundary conditions (the `pbc` flag) that wrap the lattice into a toroidal structure to eliminate edge effects. Finally, three clustering methods are available: (i) `K-means` (Ikotun et al., 2023), (ii) `GMM` (Li and Barron, 1999; Figueiredo and Jain, 2002), and (iii) `HDBSCAN` (Campello et al., 2013; McInnes et al., 2017).

3. `torchsom.visualization`:
   This module fills a significant gap in existing SOM implementations by providing visualizations for both rectangular and hexagonal topologies. It includes seven visualization types: (i) U-matrix for map topology and cluster structure, (ii) hit maps showing neuron activation patterns, (iii) component planes for feature-wise analysis, (iv) classification and metric maps for target statistics, (v) score and rank maps for quality assessment, (vi) training curves to monitor convergence, and (vii) cluster maps with associated quality metrics. All visualizations are `matplotlib`-based, supporting customizable styling and automatic figure generation (see Appendix C).

torchsom is designed to integrate with modern DL workflows using the PyTorch ecosystem, enabling GPU acceleration and efficient batch processing of large data sets. Comprehensive documentation provides implementation details, API references, and user guides covering regression, classification, and clustering tasks.[5]

The modular and extensible architecture of torchsom allows users to add new visualizations, customize functionalities, and extend core components. This flexibility facilitates rapid experimentation and adaptation, while also encouraging contributions from the open-source community. By supporting integration and customization, torchsom serves as a reference implementation for both research and production environments, enabling users to extend and tailor SOM applications to their specific requirements.

4. Benchmarks

torchsom's computational performance and fidelity are evaluated against MiniSom, the most widely adopted and actively maintained SOM library, and against somoclu, a massively parallel C++/CUDA implementation (see Table 1). To keep every comparison fair, backends are grouped by execution device: CPU backends are compared with one another in Table 2, and GPU backends in Table 3.

Synthetic data sets are generated using scikit-learn's make_blobs(), varying both sample size and feature dimensionality to assess scaling behavior. We consider data sets with sample sizes of {240, 4000, 16000} and feature counts of {4, 50, 300}. All implementations use identical, commonly adopted hyperparameters to ensure a fair comparison: a 25×15 grid, PCA initialization, rectangular topology, 100 training iterations, Gaussian neighborhood function, and Euclidean distance. All scripts, configuration files, and seed-controlled run wrappers used to produce Tables 2 and 3 are publicly released under the benchmark/ directory of the repository; the exact state of this revision is reproducible from the Git tag jmlr-revision-v2.

Both tables report (i) test Quantization Error (QE), (ii) test Topographic Error (TE), and (iii) wall-clock time (initialization + training + compilation/device setup) averaged over 10 runs. For readability, standard deviations are shown only when they are non-zero. This convention applies to every benchmark table, here and in Appendix D. The CPU comparison (Table 2) considers (a) MiniSom with its standard online training, (b) MiniSom-JIT, its optional Numba-accelerated batch-offline routine, and (c) torchsom. The GPU comparison (Table 3) considers the two backends with a native GPU kernel: (a) somoclu (CUDA C++) and (b) torchsom (CUDA PyTorch). Extensive benchmarking results, together with the metric definitions and the per-backend measurement details, are provided in Appendix D.[6]

torchsom's most consistent advantage is topology preservation: it attains the lowest TE in every rectangular configuration, on both CPU and GPU, reducing TE by 62% to 100% relative to the standard MiniSom baseline and remaining below MiniSom-JIT and somoclu throughout (e.g. at 16000×300, TE of 6% versus 76%, 31%, and 37%). On hexagonal maps it leads in nearly all configurations, the only exceptions being the low-dimensional

5. The documentation is accessible at https://opensource.michelin.io/TorchSOM.
6. All benchmarks were run on an Intel Xeon Gold 6134 CPU (16 cores at 3.20 GHz, 187 GB RAM) and an NVIDIA Tesla V100-PCIE GPU (5120 CUDA cores at 1.38 GHz, 32 GB RAM).

Data set		MiniSom (CPU)			MiniSom-JIT (CPU)			torchsom (CPU)		
Samples	Features	QE ↓	TE (%) ↓	Time (s) ↓	QE ↓	TE (%) ↓	Time (s) ↓	QE ↓	TE (%) ↓	Time (s) ↓
240	4	**0.17**	26 ± 4	1.45 ± 0.03	0.18	23	4.31 ± 0.01	0.24	**2**	$\mathbf{0.27 \pm 0.04}$
240	50	1.80	48 ± 7	2.74 ± 0.05	**1.78**	40	2.38 ± 0.01	1.79	**5**	$\mathbf{0.34 \pm 0.07}$
240	300	5.44	71 ± 10	13.84 ± 0.19	5.34	52	5.87 ± 0.01	**5.21**	**27**	$\mathbf{0.52 \pm 0.04}$
4000	4	**0.16**	32 ± 1	25.08 ± 0.35	**0.16**	20	5.91 ± 0.03	0.23	**0**	$\mathbf{3.26 \pm 0.05}$
4000	50	1.66	55 ± 2	49.03 ± 1.10	**1.64**	28	16.33 ± 0.58	1.77	**4**	$\mathbf{3.50 \pm 0.09}$
4000	300	5.14	74 ± 2	226 ± 2	**5.02**	37	69.47 ± 0.35	5.13	**6**	$\mathbf{5.25 \pm 0.10}$
16000	4	**0.15**	32 ± 1	99.35 ± 1.53	**0.15**	19	17.58 ± 0.05	0.23	**1**	$\mathbf{14.55 \pm 0.20}$
16000	50	1.64	56 ± 1	186 ± 2	**1.60**	25	51.35 ± 0.20	1.75	**3**	$\mathbf{15.63 \pm 0.20}$
16000	300	5.15	76 ± 1	900 ± 10	**4.98**	31	276 ± 1	5.14	**6**	$\mathbf{22.68 \pm 0.08}$

Table 2: CPU benchmark results with mean ± standard deviation (shown only when non-zero) across 10 runs for a 25×15 rectangular map. **Bold** marks the best (lowest) value per metric and row.

Data set		somoclu (GPU)			torchsom (GPU)		
Samples	Features	QE ↓	TE (%) ↓	Time (s) ↓	QE ↓	TE (%) ↓	Time (s) ↓
240	4	**0.19**	23	$\mathbf{0.15 \pm 0.01}$	0.24	**2**	1.29 ± 0.04
240	50	**1.80**	57	$\mathbf{0.51 \pm 0.01}$	1.83	**3**	1.37 ± 0.04
240	300	5.34	53	2.37 ± 0.01	**5.21**	$\mathbf{14 \pm 2}$	$\mathbf{1.34 \pm 0.03}$
4000	4	**0.16**	24	$\mathbf{1.77 \pm 0.02}$	0.23	**1**	3.32 ± 0.01
4000	50	**1.61**	37	8.05 ± 0.05	1.74	**4**	$\mathbf{3.44 \pm 0.04}$
4000	300	**5.02**	47	40.92 ± 1.30	5.13	**7**	$\mathbf{3.33 \pm 0.03}$
16000	4	**0.15**	22	$\mathbf{10.48 \pm 0.27}$	0.23	**0**	12.71 ± 0.03
16000	50	**1.60**	30	31.82 ± 0.16	1.75	**4**	$\mathbf{13.29 \pm 0.10}$
16000	300	**4.98**	37	161	5.14	**5**	$\mathbf{13.08 \pm 0.04}$

Table 3: GPU benchmark results with mean ± standard deviation (shown only when non-zero) across 10 runs for a 25×15 rectangular map. **Bold** marks the best (lowest) value per metric and row.

(4-feature) sets (Appendix D). QE is comparable across backends, with `torchsom` being marginally higher.

On CPU (Table 2), `torchsom` is 81% to 98% faster than standard `MiniSom`: `MiniSom` updates one sample at a time and cannot exploit batched linear algebra, whereas `torchsom` processes the whole batch with vectorized tensor operations, so its advantage grows with both sample size and feature dimensionality. Against the optimized `MiniSom-JIT` baseline the result is workload-dependent: `torchsom` matches or outperforms it in every rectangular configuration (up to $\sim$16×), with the gap smallest on the low-dimensional (4-feature) sets, where there is little batched work to amortize.

On GPU (Table 3), this batching effect combines with a startup overhead: `torchsom`'s reported GPU time carries a one-time initialization (independent of feature count), whereas on small problems `somoclu`'s compiled kernel starts up almost instantly. On the 4-feature and smallest sets there is too little compute to hide this overhead, so `somoclu` is faster; as the data grows it is amortized and `torchsom`'s parallelism dominates, reaching up to $\sim$12× faster than `somoclu` on high-dimensional data. The same fixed-cost-amortized-at-scale effect appears on CPU for `MiniSom-JIT`, whose one-time Numba compilation is only worthwhile once the workload is large enough to absorb it. Device choice is therefore itself workload-dependent: GPU execution is most beneficial for large, high-dimensional data (e.g. cutting the 16000×300 time by roughly 40% relative to CPU), while the multi-core CPU is competitive or faster for small or low-dimensional data.

These advantages compound with map size. On the larger 90×70 grid (Appendix D, roughly 17× the neurons), torchsom's CPU speedup over standard MiniSom grows to ∼200× and over MiniSom-JIT to ∼50×, while its GPU speedup over somoclu reaches ∼66×. TE stays markedly lower than every baseline, while QE remains on par, as in the main comparison. The device trade-off also tips further toward the GPU: at this scale torchsom's GPU run is ∼8× faster than its CPU run, confirming that the parallel backend is most valuable on large, high-dimensional problems.

All reported torchsom times are conservative: the library additionally evaluates both metrics (QE and TE) over the full batch at every epoch, an $\mathcal{O}(2 \times \text{batch} \times \text{epochs})$ cost that MiniSom and somoclu avoid by training only and measuring once at the end. Overall, torchsom combines the strongest topology preservation with competitive or superior speed (dominant against standard MiniSom, and leading the optimized baselines on high-dimensional and GPU workloads) within a single modular library.

5. Conclusion

We introduced torchsom, a PyTorch-native reference implementation of the SOM that combines a scikit-learn-compatible API with native GPU acceleration, an advanced built-in visualization suite, a built-in clustering interface, and JITL support within a single modular and openly licensed codebase. Beyond the classical algorithm, torchsom provides a configurable BMU search backend with optional FAISS compatibility, flexible neighborhood retrieval modes, and rectangular and hexagonal grid topologies that can be wrapped into toroidal structures through periodic boundary conditions. Across nine synthetic benchmark configurations spanning sample sizes from 240 to 16,000 and feature dimensionalities from 4 to 300, and against fair per-device baselines, torchsom attains QE parity with MiniSom and the lowest TE in every rectangular configuration (62% to 100% below standard MiniSom) and nearly all hexagonal ones. It is 81% to 98% faster than standard MiniSom on CPU and, against the optimized MiniSom-JIT and somoclu baselines, is fastest on high-dimensional and GPU workloads while staying competitive elsewhere. These advantages become more pronounced as the map size increases, with torchsom achieving speedups of up to $\sim 200\times$ over MiniSom on CPU and $\sim 66\times$ over somoclu on GPU (Appendix D).

Several directions remain open for future work: (i) integration of Growing and Hierarchical SOM variants; (ii) broader benchmarking, both against domain-specific data (e.g., time series, images, industrial sensor streams) beyond scikit-learn's make_blobs(), and against established dimensionality-reduction and manifold-learning baselines such as PCA, t-SNE, and UMAP; and (iii) extension of the visualization suite with interactive widgets for exploratory analysis.

torchsom is distributed on PyPI under the Apache 2.0 license, with narrative documentation at https://opensource.michelin.io/TorchSOM and source at https://github.com/michelin/TorchSOM. The benchmark scripts and configuration files are released under the benchmark/ directory of the repository for full reproducibility.

Appendix A. Glossary

BMU Best Matching Unit.

DL Deep Learning.

JITL Just-In-Time Learning.

ML Machine Learning.

QE Quantization Error.

SOM Self-Organizing Map.

TE Topographic Error.

Appendix B. SOM Overview

This appendix gives a self-contained, notation-complete description of the SOM as implemented in `torchsom`. The same material, kept in sync with the code, is mirrored in the online documentation at `https://opensource.michelin.io/TorchSOM/getting_started/basic_concepts.html`.

Setup and notation. A SOM approximates a distribution over an input space $\mathcal{X} \subseteq \mathbb{R}^d$ by a two-dimensional lattice of $I \times J$ neurons. Each neuron at grid position (i, j), with $i \in \{1, \dots, I\}$ and $j \in \{1, \dots, J\}$, carries a *codebook* (weight) vector $\mathbf{w}_{ij} \in \mathbb{R}^l$ with $l = d$, and the full parameter set is the tensor

$$\mathbf{W} \coloneqq [\mathbf{w}_{ij}]_{i \leq I,\, j \leq J} \in \mathbb{R}^{I \times J \times l}. \tag{1}$$

Training uses a data set $\{\mathbf{x}_k\}_{k=1}^{N} \subset \mathbb{R}^d$ over epochs $t \in \{0, 1, \dots, T\}$.

Best-matching unit and projection. Similarity in feature space is measured by a distance $\delta : \mathbb{R}^d \times \mathbb{R}^l \to \mathbb{R}_{\geq 0}$ (Eq. 7). For an input $\mathbf{x}$, the BMU is the neuron whose codebook minimizes δ,

$$\mathrm{BMU}(\mathbf{x}) \coloneqq \underset{(i,j)}{\arg\min}\ \delta(\mathbf{x}, \mathbf{w}_{ij}), \tag{2}$$

which induces a projection onto grid coordinates and a latent codebook retrieval,

$$\psi : \mathbb{R}^d \to \{1, \dots, I\} \times \{1, \dots, J\}, \qquad \psi(\mathbf{x}) \coloneqq \mathrm{BMU}(\mathbf{x}), \qquad \mathbf{z} \coloneqq \mathbf{w}_{\psi(\mathbf{x})} \in \mathbb{R}^l. \tag{3}$$

The latent vector $\mathbf{z}$ is the representation used for clustering, visualization, and JITL retrieval.

Competitive update. A SOM learns by a neighborhood-weighted competitive rule rather than gradient descent: at each step the BMU for the presented sample $\mathbf{x}$ is found, and each neuron is moved toward $\mathbf{x}$ by a step scaled by its grid proximity to the BMU,

$$\mathbf{w}_{ij}(t+1) \coloneqq \mathbf{w}_{ij}(t) + \alpha(t)\, h_{ij}(t) \left(\mathbf{x} - \mathbf{w}_{ij}(t)\right), \tag{4}$$

with learning rate $\alpha(t) \in \mathbb{R}_{\geq 0}$ (Eq. 8) and neighborhood weight $h_{ij}(t) \in \mathbb{R}$ (Eq. 5).

Neighborhood function and order. Let $\rho \coloneqq d_{\text{grid}}\big((i,j), \text{BMU}\big)$ denote the *grid-space* distance from neuron (i,j) to the BMU, where the grid metric is induced by the lattice geometry (Eq. 9). `torchsom` provides four neighborhood kernels of width $\sigma(t)$,

$$\begin{aligned} h_{ij}^{\text{gaussian}}(t) &\coloneqq \exp\left(-\frac{\rho^2}{2\,\sigma(t)^2}\right), & h_{ij}^{\text{mexican}}(t) &\coloneqq \left(1-\frac{\rho^2}{4\,\sigma(t)^2}\right)\exp\left(-\frac{\rho^2}{2\,\sigma(t)^2}\right), \\ h_{ij}^{\text{bubble}}(t) &\coloneqq \mathbb{I}\big(\rho \le \sigma(t)\big), & h_{ij}^{\text{triangle}}(t) &\coloneqq \max\left(0,\ 1-\frac{\rho}{\sigma(t)}\right). \end{aligned} \tag{5}$$

Discrete neighborhoods are controlled by an integer *order* $o \in \mathbb{N}^+$. On a rectangular grid the order-o neighborhood of the BMU at (i,j) is the Chebyshev ball

$$\mathcal{N}_o\big((i,j)\big) \coloneqq \big\{(i',j') \in \{1,\dots,I\} \times \{1,\dots,J\} : \max(|i'-i|,\, |j'-j|) \le o\big\}, \tag{6}$$

at most a $(2o{+}1) \times (2o{+}1)$ block of neurons; the hexagonal grid uses the analogous hop-distance rings (Figure 1). The order o sets both the support of the discrete weight update and the sample-retrieval neighborhoods used for JITL (the `neighborhood_order` parameter and the `bmu_neighborhood` retrieval modes).

Feature-space distances. The distance δ in Eq. 2 is configurable; writing x_a and w_a for the a-th components,

$$\begin{aligned} \delta_{\text{euclidean}}(\mathbf{x},\mathbf{w}) &\coloneqq \sqrt{\textstyle\sum_{a=1}^{d}(x_a-w_a)^2}, & \delta_{\text{manhattan}}(\mathbf{x},\mathbf{w}) &\coloneqq \textstyle\sum_{a=1}^{d}|x_a-w_a|, \\ \delta_{\text{cosine}}(\mathbf{x},\mathbf{w}) &\coloneqq 1-\frac{\mathbf{x}\cdot\mathbf{w}}{\|\mathbf{x}\|_2\,\|\mathbf{w}\|_2}, & \delta_{\text{chebyshev}}(\mathbf{x},\mathbf{w}) &\coloneqq \max_{1\le a\le d}|x_a-w_a|. \end{aligned} \tag{7}$$

Decay schedules. `torchsom` offers an inverse, a linear, and a general asymptotic schedule; one is selected per parameter, and each is a closed-form function of the epoch t and of the

(a) Rectangular: Chebyshev blocks. (b) Hexagonal: hop-distance rings.

Figure 1: Neighborhood orders around the BMU (black) for $o = 1$ (blue), $o = 2$ (teal), and $o = 3$ (pink), on the two grid topologies supported by `torchsom`, illustrating $\mathcal{N}_o$ in Eq. 6.

initial value,

$$\begin{aligned}
\alpha_{\mathrm{inv}}(t) &\coloneqq \alpha(0)\,\frac{\gamma}{\gamma+t}, & \alpha_{\mathrm{lin}}(t) &\coloneqq \alpha(0)\left(1-\frac{t}{T}\right), \\
\sigma_{\mathrm{inv}}(t) &\coloneqq \frac{\sigma(0)}{1+t\,(\sigma(0)-1)/T}, & \sigma_{\mathrm{lin}}(t) &\coloneqq \sigma(0)+t\,\frac{1-\sigma(0)}{T}, \\
\theta_{\mathrm{asym}}(t) &\coloneqq \frac{\theta(0)}{1+2t/T},
\end{aligned} \tag{8}$$

with total epochs $T \in \mathbb{N}$, current epoch $t \in \{0, \ldots, T\}$, inverse-decay constant $\gamma \coloneqq T/100$, and $\theta \in \{\alpha, \sigma\}$ for the asymptotic schedule. The linear schedules reach $\alpha_{\mathrm{lin}}(T) = 0$ and $\sigma_{\mathrm{lin}}(T) = 1$ exactly, as does σ_{inv}.

Grid topologies and boundary conditions. `torchsom` supports rectangular and hexagonal grids. On a hexagonal grid every neuron is equidistant from its six neighbors, whereas on a rectangular grid diagonal neighbors are farther away than orthogonal ones. The lattice geometry fixes a planar embedding φ of the grid positions: the identity on a rectangular grid, and the even-r offset layout on a hexagonal grid, in which odd rows are shifted by half a cell and consecutive rows are spaced by $\sqrt{3}/2$. Either grid may enable periodic boundary conditions (the `pbc` flag), identifying opposite edges so the lattice becomes a torus. Then, the grid metric is

$$d_{\mathrm{grid}}\big((i,j),(i',j')\big) \coloneqq \min_{\mathbf{s}\in\mathcal{S}} \big\|\varphi(i,j) - \varphi(i',j') + \mathbf{s}\big\|_2, \tag{9}$$

where $\mathcal{S}$ enumerates translations by the grid periods, so neighborhoods wrap across boundaries and corner neurons are not penalized. Without PBC, $\mathcal{S} = \{\mathbf{0}\}$ and d_{grid} is the plain Euclidean distance in the embedding.

Initialization. Codebooks are initialized either (i) at random, sampling each $\mathbf{w}_{ij}$ uniformly over the per-feature data range, or (ii) by PCA, placing the $\mathbf{w}_{ij}$ on the plane spanned by the two leading principal components of $\{\mathbf{x}_k\}$. PCA initialization speeds convergence and improves reproducibility.

Quality metrics. Map fidelity and topology preservation are quantified by the QE and TE (reported in Tables 2 and 3),

$$\mathrm{QE} \coloneqq \frac{1}{N}\sum_{k=1}^{N} \big\|\mathbf{x}_k - \mathbf{w}_{\mathrm{BMU}(\mathbf{x}_k)}\big\|_2, \tag{10}$$

$$\mathrm{TE} \coloneqq \frac{1}{N}\sum_{k=1}^{N} \mathbb{I}\big(\mathrm{BMU}_2(\mathbf{x}_k) \notin \mathcal{N}_1\big(\mathrm{BMU}(\mathbf{x}_k)\big)\big), \tag{11}$$

where $\mathcal{N}_1(\cdot)$ is the order-one neighborhood of the lattice (Eq. 6 with $o = 1$): a neuron together with its immediate neighbors, the eight orthogonal and diagonal ones on a rectangular grid and the six at hop distance one on a hexagonal grid. Every backend is scored with the same definition, so TE values are comparable within a table. They are not comparable between the rectangular and hexagonal tables, whose adjacency sets differ in size. QE measures quantization fidelity, while TE measures the fraction of samples whose two best neurons are non-adjacent; lower is better for both.

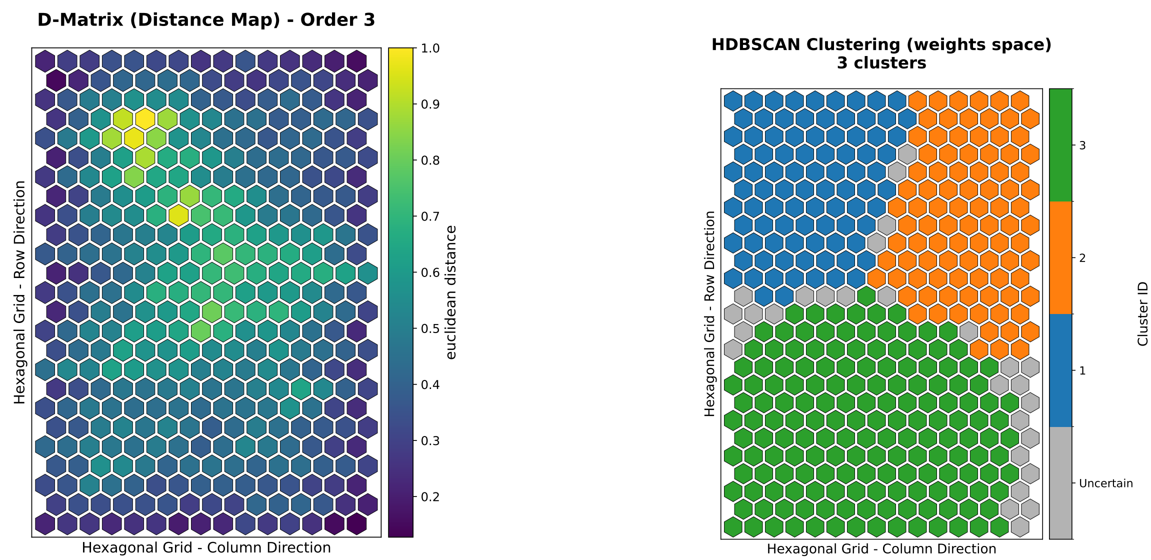
The package structure, class hierarchy, BMU-search backends, and end-to-end training flow are documented at `https://opensource.michelin.io/TorchSOM/user_guide/architecture.html`.

Appendix C. `torchsom` Visualization Examples

`torchsom` ships an integrated, `matplotlib`-based visualization suite covering seven categories of maps for both training diagnostics and post-hoc analysis (Table 4); each category supports both rectangular and hexagonal topologies and integrates with the `scikit-learn`-style API. Two representative outputs are shown in Figure 2: a distance map (U-matrix) and a cluster map on two data sets.

Category	**Purpose**
Learning curves	Per-epoch QE/TE traces during training
Distance map (U-matrix)	Inter-neuron distance landscape; reveals cluster boundaries
Hit map	Sample density / neuron utilization
Component plane	Per-feature weight surface across the grid
Classification & metric maps	Majority-class assignment (classification) and aggregated target statistics (regression)
Score & rank maps	Per-neuron reliability (variance, density, significance) and predicted-value ordering (regression)
Cluster analysis	Cluster map with silhouette, elbow, and clustering-algorithm comparison diagnostics

Table 4: Visualization categories provided by `torchsom`'s built-in suite.



(a) Distance map (U-matrix), wine data set.

(b) Cluster map, `make_blobs` data set.

Figure 2: Two outputs from `torchsom`'s integrated visualization suite (hexagonal topology): (a) a distance map (U-matrix) exposing cluster boundaries through inter-neuron distances; and (b) a cluster map showing the spatial assignment of automatically discovered clusters. The remaining visualizations are available at `https://opensource.michelin.io/TorchSOM/user_guide/visualization_help.html`.

Appendix D. Benchmark Results

The tables below extend the main comparison (Tables 2 and 3) to three additional configurations: a hexagonal 25×15 map across every sample/feature combination, and the larger 90×70 map at 300 features in both rectangular and hexagonal topologies. Each configuration follows the same device-matched layout as the main text: a CPU table with `MiniSom`, `MiniSom`-JIT, and `torchsom`, and a GPU table with `somoclu` and `torchsom`.

On both CPU and GPU the hexagonal results mirror the rectangular trends: `torchsom` again attains the lowest TE in nearly all configurations, the exception being the 4-feature sets, where `MiniSom`-JIT (CPU) and `somoclu` (GPU) are marginally lower. `torchsom`'s hexagonal wall-clock time is markedly higher than on the rectangular map, which reflects a current implementation limitation rather than an algorithmic cost: the per-epoch TE evaluation is fully vectorized for rectangular grids but, for hexagonal grids, still iterates over the batch and copies each sample's best-matching units from GPU to CPU, and this host synchronization dominates the hexagonal runtime. Therefore, the reported hexagonal times are conservative, and vectorizing this evaluation is a target for future optimization.

These extended tables confirm and sharpen the main-text findings at scale. On the 90×70 map `torchsom`'s CPU speed advantage over standard `MiniSom` reaches $\sim$200×, and $\sim$66× over `somoclu` on GPU, while TE stays lowest and QE remains on par. Rectangular and hexagonal maps agree, with one deviation: hexagonal GPU time is inflated by the unvectorized per-epoch TE evaluation noted after Table 6 (e.g. 94.8 s versus 15.6 s rectangular at 16000×300), so those times remain conservative.

Data set		`MiniSom` (CPU)			`MiniSom-JIT` (CPU)			`torchsom` (CPU)		
Samples	Features	QE ↓	TE (%) ↓	Time (s) ↓	QE ↓	TE (%) ↓	Time (s) ↓	QE ↓	TE (%) ↓	Time (s) ↓
240	4	**0.18**	25 ± 6	1.47 ± 0.02	**0.18**	**22**	1.98	0.24	32	$\mathbf{0.50 \pm 0.03}$
240	50	**1.78**	56 ± 5	2.98 ± 0.07	1.79	62	2.39 ± 0.01	1.80	**15**	$\mathbf{0.52 \pm 0.04}$
240	300	5.39	86 ± 3	12.95 ± 0.03	5.32	53	5.72 ± 0.02	**5.22**	**10**	$\mathbf{0.72 \pm 0.04}$
4000	4	**0.16**	31 ± 1	23.47 ± 0.38	**0.16**	**20**	$\mathbf{5.64 \pm 0.04}$	0.23	26	7.30 ± 0.08
4000	50	1.66	51 ± 1	45.86 ± 2.13	**1.64**	27	14.58 ± 0.76	1.78	**17**	$\mathbf{7.64 \pm 0.11}$
4000	300	5.12	71 ± 1	233 ± 1	**5.02**	34	68.86 ± 0.51	5.15	$\mathbf{12 \pm 1}$	$\mathbf{12.65 \pm 0.17}$
16000	4	**0.16**	31 ± 1	95.89 ± 2.21	**0.16**	**19**	$\mathbf{17.76 \pm 0.16}$	0.22	26	29.57 ± 0.37
16000	50	1.64	54 ± 1	184 ± 2	**1.61**	24	51.02 ± 0.22	1.75	**13**	$\mathbf{30.03 \pm 0.18}$
16000	300	5.13	72 ± 1	885 ± 3	**4.98**	28	271 ± 3	5.15	**11**	$\mathbf{36.85 \pm 0.15}$

Table 5: CPU benchmark results with mean ± standard deviation (shown only when non-zero) across 10 runs for a 25×15 hexagonal map. **Bold** marks the best (lowest) value per metric and row.

Data set		`somoclu` (GPU)			`torchsom` (GPU)		
Samples	Features	QE ↓	TE (%) ↓	Time (s) ↓	QE ↓	TE (%) ↓	Time (s) ↓
240	4	**0.19**	43	**0.18**	0.22	**27**	2.31 ± 0.03
240	50	**1.78**	45	$\mathbf{0.54 \pm 0.02}$	1.84	**18**	2.89 ± 0.03
240	300	5.29	53	$\mathbf{2.46 \pm 0.02}$	**5.21**	**12**	2.91 ± 0.04
4000	4	**0.17**	**26**	$\mathbf{1.77 \pm 0.02}$	0.22	29	20.65 ± 0.04
4000	50	**1.62**	34	$\mathbf{7.71 \pm 0.19}$	1.75	**14**	20.56 ± 0.11
4000	300	**5.01**	47	43.69 ± 2.87	5.15	**13**	$\mathbf{20.39 \pm 0.22}$
16000	4	**0.16**	$\mathbf{26 \pm 1}$	$\mathbf{8.34 \pm 0.07}$	0.22	30	115
16000	50	**1.61**	32 ± 1	$\mathbf{33.23 \pm 0.13}$	1.75	**14**	81.03 ± 0.04
16000	300	**4.99**	38	155	5.15	**13**	$\mathbf{117 \pm 10}$

Table 6: GPU benchmark results with mean ± standard deviation (shown only when non-zero) across 10 runs for a 25×15 hexagonal map. **Bold** marks the best (lowest) value per metric and row.

Data set		`MiniSom` (CPU)			`MiniSom-JIT` (CPU)			`torchsom` (CPU)		
Samples	Features	QE ↓	TE (%) ↓	Time (s) ↓	QE ↓	TE (%) ↓	Time (s) ↓	QE ↓	TE (%) ↓	Time (s) ↓
240	300	5.56	78 ± 6	547 ± 6	5.51	77	88.73 ± 0.37	**5.19**	$\mathbf{11 \pm 1}$	$\mathbf{2.36 \pm 0.10}$
4000	300	5.18	84 ± 1	5526 ± 572	**5.09**	67	1607 ± 35	5.11	**7**	$\mathbf{36.56 \pm 0.17}$
16000	300	5.08	85 ± 1	25860 ± 613	**4.96**	62	6365 ± 49	5.12	**7**	$\mathbf{130 \pm 2}$

Table 7: CPU benchmark results with mean ± standard deviation (shown only when non-zero) across 10 runs for a 90×70 rectangular map. **Bold** marks the best (lowest) value per metric and row.

Data set		`somoclu` (GPU)			`torchsom` (GPU)		
Samples	Features	QE ↓	TE (%) ↓	Time (s) ↓	QE ↓	TE (%) ↓	Time (s) ↓
240	300	5.77	82	38.46 ± 4.71	**5.20**	**20**	$\mathbf{3.20 \pm 0.04}$
4000	300	**5.04**	60	471 ± 10	5.11	**7**	$\mathbf{7.27 \pm 0.04}$
16000	300	**4.95**	56	1038 ± 15	5.12	**7**	$\mathbf{15.64 \pm 0.04}$

Table 8: GPU benchmark results with mean ± standard deviation (shown only when non-zero) across 10 runs for a 90×70 rectangular map. **Bold** marks the best (lowest) value per metric and row.

Data set		`MiniSom` (CPU)			`MiniSom-JIT` (CPU)			`torchsom` (CPU)		
Samples	Features	QE ↓	TE (%) ↓	Time (s) ↓	QE ↓	TE (%) ↓	Time (s) ↓	QE ↓	TE (%) ↓	Time (s) ↓
240	300	5.56	82 ± 6	315 ± 2	5.53	70	87.30 ± 0.30	**5.21**	**38**	$\mathbf{2.62 \pm 0.06}$
4000	300	5.14	93 ± 1	5672 ± 309	**5.06**	67	1426 ± 16	5.12	**20**	$\mathbf{48.93 \pm 1.44}$
16000	300	5.04	83 ± 1	25250 ± 2268	**4.95**	62	6410 ± 107	5.11	**18**	$\mathbf{148 \pm 6}$

Table 9: CPU benchmark results with mean ± standard deviation (shown only when non-zero) across 10 runs for a 90×70 hexagonal map. **Bold** marks the best (lowest) value per metric and row.

Data set		`somoclu` (GPU)			`torchsom` (GPU)		
Samples	Features	QE ↓	TE (%) ↓	Time (s) ↓	QE ↓	TE (%) ↓	Time (s) ↓
240	300	5.69	92	36.73 ± 1.24	**5.21**	**35**	$\mathbf{2.97 \pm 0.04}$
4000	300	**5.01**	60	479 ± 12	5.11	**19**	$\mathbf{21.74 \pm 0.04}$
16000	300	**4.94**	55	1189 ± 30	5.12	**19**	$\mathbf{94.77 \pm 7.31}$

Table 10: GPU benchmark results with mean ± standard deviation (shown only when non-zero) across 10 runs for a 90×70 hexagonal map. **Bold** marks the best (lowest) value per metric and row.

References

J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, CK. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, M. Suo, P. Tillet, E. Wang, X. Wang, W. Wen, S. Zhang, X. Zhao, K. Zhou, R. Zou, A. Mathews, G. Chanan, P. Wu, and S. Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. ACM, Apr. 2024. URL https://docs.pytorch.org/assets/pytorch2-2.pdf.

L. Berthier. torchsom: The Reference PyTorch Library for Self-Organizing Maps, 2025. URL https://github.com/michelin/TorchSOM.

J. Z. Bloom. MARKET SEGMENTATION: A Neural Network Application. *Annals of Tourism Research*, 32(1):93–111, Jan. 2005. ISSN 0160-7383. doi: 10.1016/j.annals.2004.05.001. URL https://www.sciencedirect.com/science/article/pii/S0160738304001033.

F. Bowen and J. Siegler. Self-organizing maps: A novel approach to identify and map business clusters. *Journal of Management Analytics*, 11(2):228–246, Apr. 2024. ISSN 2327-0012. doi: 10.1080/23270012.2024.2306628. URL https://doi.org/10.1080/23270012.2024.2306628.

R. J. G. B. Campello, D. Moulavi, and J. Sander. Density-Based Clustering Based on Hierarchical Density Estimates. In *Advances in Knowledge Discovery and Data Mining*, pages 160–172, Berlin, Heidelberg, 2013. Springer. ISBN 978-3-642-37456-2. doi: 10.1007/978-3-642-37456-2_14.

F. Comitani. SimpSOM (Simple Self-Organizing Maps), 2017. URL https://github.com/fcomitani/simpsom.

L. Concetti, G. Mazzuto, F. E. Ciarapica, and M. Bevilacqua. An Unsupervised Anomaly Detection Based on Self-Organizing Map for the Oil and Gas Sector. *Applied Sciences*, 13(6):3725, Jan. 2023. ISSN 2076-3417. doi: 10.3390/app13063725. URL https://www.mdpi.com/2076-3417/13/6/3725.

S. Dash, S. Chakravarty, N. C. Giri, U. Ghugar, and G. Fotis. Performance Assessment of Different Sustainable Energy Systems Using Multiple-Criteria Decision-Making Model and Self-Organizing Maps. *Technologies*, 12(3):42, Mar. 2024. ISSN 2227-7080. doi: 10.3390/technologies12030042. URL https://www.mdpi.com/2227-7080/12/3/42.

A. Farzamnia, S. H. Hazaveh, S. S. Siadat, and E. G. Moung. MRI Brain Tumor Detection Methods Using Contourlet Transform Based on Time Adaptive Self-Organizing Map. *IEEE Access*, 11:113480–113492, 2023. ISSN 2169-3536. doi: 10.1109/ACCESS.2023.3322450. URL https://ieeexplore.ieee.org/document/10273724/.

Z. Feng, C. Xu, Y. Zuo, X. Luo, L. Wang, H. Chen, X. Xie, D. Yan, and T. Liang. Analysis of water quality indexes and their relationships with vegetation using self-organizing map and geographically and temporally weighted regression. *Environmental Research*, 216: 114587, Jan. 2023. ISSN 0013-9351. doi: 10.1016/j.envres.2022.114587. URL https://www.sciencedirect.com/science/article/pii/S0013935122019144.

M. Figueiredo and A. Jain. Unsupervised learning of finite mixture models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(3):381–396, Mar. 2002. ISSN 1939-3539. doi: 10.1109/34.990138. URL https://ieeexplore.ieee.org/document/990138.

G. Gad, A. Farrag, A. Aboulfotouh, K. Bedda, Z. M. Fadlullah, and M. M. Fouda. Joint Self-Organizing Maps and Knowledge-Distillation-Based Communication-Efficient Federated Learning for Resource-Constrained UAV-IoT Systems. *IEEE Internet of Things Journal*, 11(9):15504–15522, May 2024. ISSN 2327-4662. doi: 10.1109/JIOT.2023.3349295. URL https://ieeexplore.ieee.org/document/10379499.

M. Hao, K. Hua, and X. Zhang. SOMDE: A scalable method for identifying spatially variable genes with self-organizing map. *Bioinformatics*, 37(23):4392–4398, Dec. 2021. ISSN 1367-4803. doi: 10.1093/bioinformatics/btab471. URL https://doi.org/10.1093/bioinformatics/btab471.

A. M. Ikotun, A. E. Ezugwu, L. Abualigah, B. Abuhaija, and J. Heming. K-means clustering algorithms: A comprehensive review, variants analysis, and advances in the era of big data. *Information Sciences*, 622:178–210, Apr. 2023. ISSN 0020-0255. doi: 10.1016/j.ins.2022.11.139. URL https://www.sciencedirect.com/science/article/pii/S0020025522014633.

H. Jin, J. Li, M. Wang, B. Qian, B. Yang, Z. Li, and L. Shi. Ensemble Just-In-Time Learning-Based Soft Sensor for Mooney Viscosity Prediction in an Industrial Rubber Mixing Process. *Advances in Polymer Technology*, 2020(1):6575326, 2020. ISSN 1098-2329. doi: 10.1155/2020/6575326. URL https://onlinelibrary.wiley.com/doi/abs/10.1155/2020/6575326.

S. Khan and A. B. Mailewa. Discover botnets in IoT sensor networks: A lightweight deep learning framework with hybrid self-organizing maps. *Microprocessors and Microsystems*, 97:104753, Mar. 2023. ISSN 0141-9331. doi: 10.1016/j.micpro.2022.104753. URL https://www.sciencedirect.com/science/article/pii/S0141933122002824.

T. Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(1):59–69, Jan. 1982. ISSN 1432-0770. doi: 10.1007/BF00337288. URL https://doi.org/10.1007/BF00337288.

T. Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, Sept. 1990. ISSN 1558-2256. doi: 10.1109/5.58325. URL https://ieeexplore.ieee.org/document/58325/.

T. Kohonen. *Self-Organizing Maps*, volume 30 of *Springer Series in Information Sciences*. Springer, Berlin, Heidelberg, 2001. ISBN 978-3-540-67921-9 978-3-642-56927-2. doi: 10.1007/978-3-642-56927-2. URL http://link.springer.com/10.1007/978-3-642-56927-2.

S. K. Lam, A. Pitrou, and S. Seibert. Numba: a llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340052. doi: 10.1145/2833157.2833162. URL https://doi.org/10.1145/2833157.2833162.

J. Li and A. Barron. Mixture Density Estimation. In *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999. URL https://papers.nips.cc/paper_files/paper/1999/hash/a0f3601dc682036423013a5d965db9aa-Abstract.html.

S. Licen, A. Astel, and S. Tsakovski. Self-organizing map algorithm for assessing spatial and temporal patterns of pollutants in environmental compartments: A review. *Science of The Total Environment*, 878:163084, June 2023. ISSN 0048-9697. doi: 10.1016/j.scitotenv.2023.163084. URL https://www.sciencedirect.com/science/article/pii/S0048969723017035.

V. G. Maltarollo, K. M. Honório, A. B. F. da Silva, V. G. Maltarollo, K. M. Honório, and A. B. F. da Silva. Applications of Artificial Neural Networks in Chemical Problems. In *Artificial Neural Networks - Architectures and Applications*. IntechOpen, Jan. 2013. ISBN 978-953-51-0935-8. doi: 10.5772/51275. URL https://www.intechopen.com/chapters/39067.

L. McInnes, J. Healy, and S. Astels. Hdbscan: Hierarchical density based clustering. *Journal of Open Source Software*, 2(11):205, Mar. 2017. ISSN 2475-9066. doi: 10.21105/joss.00205. URL https://joss.theoj.org/papers/10.21105/joss.00205.

M. Y. Mia, M. E. Haque, A. R. M. T. Islam, J. N. Jannat, M. M. M. F. Jion, M. S. Islam, M. A. B. Siddique, A. M. Idris, V. Senapathi, S. Talukdar, and A. Rahman. Analysis of self-organizing maps and explainable artificial intelligence to identify hydrochemical factors that drive drinking water quality in Haor region. *Science of The Total Environment*, 904:166927, Dec. 2023. ISSN 0048-9697. doi: 10.1016/j.scitotenv.2023.166927. URL https://www.sciencedirect.com/science/article/pii/S0048969723055523.

V. Moosavi, S. Packmann, and I. Valles. A Python Library for Self Organizing Map (SOM), 2014. URL https://github.com/sevamoo/SOMPY.

A. Müller. Som-pbc: A simple self-organizing map implementation in Python with periodic boundary conditions, 2018. URL https://github.com/alexarnimueller/som.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay. Scikit-learn: Machine Learning in Python, 2011. URL https://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html.

R. A. Raj, D. Sarathkumar, L. J. B. Andrews, and S. K. Venkatachary. Key Gases in Transformer Oil – An Analysis using Self Organizing Map (SOM) Neural Networks. In *2023 IEEE 12th International Conference on Communication Systems and Network Technologies (CSNT)*, pages 642–647, Apr. 2023. doi: 10.1109/CSNT57126.2023.10134597. URL `https://ieeexplore.ieee.org/document/10134597`.

A. Urhan and B. Alakent. Integrating adaptive moving window and just-in-time learning paradigms for soft-sensor design. *Neurocomputing*, 392:23–37, June 2020. ISSN 0925-2312. doi: 10.1016/j.neucom.2020.01.083. URL `https://www.sciencedirect.com/science/article/pii/S0925231220301417`.

G. Vettigli. MiniSom: Minimalistic and NumPy-based implementation of the Self Organizing Map, 2018. URL `https://github.com/JustGlowing/minisom`.

A. Weber, M. dM Vivanco, and J. L. Toca-Herrera. Application of self-organizing maps to AFM-based viscoelastic characterization of breast cancer cell mechanics. *Scientific Reports*, 13(1):3087, Feb. 2023. ISSN 2045-2322. doi: 10.1038/s41598-023-30156-3. URL `https://www.nature.com/articles/s41598-023-30156-3`.

P. Wittek. Somoclu: Massively parallel self-organizing maps, 2017. URL `https://github.com/peterwittek/somoclu`.

Q. Xiang, H. Yu, H. Chu, M. Hu, T. Xu, X. Xu, and Z. He. The potential ecological risk assessment of soil heavy metals using self-organizing map. *Science of The Total Environment*, 843:156978, Oct. 2022. ISSN 0048-9697. doi: 10.1016/j.scitotenv.2022.156978. URL `https://www.sciencedirect.com/science/article/pii/S004896972204075X`.

M. Zhang, B. Chen, Y. Wu, W. Deng, X. Zhang, and Y. Liu. Online Semi-supervised Quality Prediction Model for Batch Mixing Process. In *2018 IEEE 7th Data Driven Control and Learning Systems Conference (DDCLS)*, pages 893–898, May 2018. doi: 10.1109/DDCLS.2018.8516014. URL `https://ieeexplore.ieee.org/document/8516014`.

Y. Zhang, H. Jin, H. Liu, B. Yang, and S. Dong. Deep Semi-Supervised Just-in-Time Learning Based Soft Sensor for Mooney Viscosity Estimation in Industrial Rubber Mixing Process. *Polymers*, 14(5):1018, Mar. 2022. ISSN 2073-4360. doi: 10.3390/polym14051018. URL `https://www.mdpi.com/2073-4360/14/5/1018`.

Y. Zhang, Q. Zhang, W. Chen, W. Shi, Y. Cui, L. Chen, and J. Shao. Hydrogeochemical analysis and groundwater pollution source identification based on self-organizing map at a contaminated site. *Journal of Hydrology*, 616:128839, Jan. 2023. ISSN 0022-1694. doi: 10.1016/j.jhydrol.2022.128839. URL `https://www.sciencedirect.com/science/article/pii/S0022169422014093`.

W. Zheng, Y. Liu, Z. Gao, and J. Yang. Just-in-time semi-supervised soft sensor for quality prediction in industrial rubber mixers. *Chemometrics and Intelligent Laboratory Systems*, 180:36–41, Sept. 2018. ISSN 0169-7439. doi: 10.1016/j.chemolab.2018.07.002. URL `https://www.sciencedirect.com/science/article/pii/S0169743918300844`.