








# xesn: Echo state networks powered by xarray and dask

Timothy A. Smith <sup>1¶</sup>, Stephen G. Penny <sup>2,3</sup>, Jason A. Platt <sup>4</sup>, and Tse-Chun Chen <sup>5</sup>

<sup>1</sup> Physical Sciences Laboratory (PSL), National Oceanic and Atmospheric Administration (NOAA), Boulder, CO, USA <sup>2</sup> Sofar Ocean, San Francisco, CA, USA <sup>3</sup> Cooperative Institute for Research in Environmental Sciences (CIRES) at the University of Colorado Boulder, Boulder, CO, USA <sup>4</sup> University of California San Diego (UCSD), La Jolla, CA, USA <sup>5</sup> Pacific Northwest National Laboratory, Richland, WA, USA ¶ Corresponding author

DOI: [10.21105/joss.07286](https://doi.org/10.21105/joss.07286)

## Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: [Jonny Saunders](#) 

## Reviewers:

- [@Arcomano1234](#)
- [@wiljnich](#)

Submitted: 24 June 2024

Published: 01 November 2024

## License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](#)).

## Summary

Xesn is a python package that allows scientists to easily design Echo State Networks (ESNs) for forecasting problems. ESNs are a Recurrent Neural Network architecture introduced by Jaeger (2001) that are part of a class of techniques termed Reservoir Computing. One defining characteristic of these techniques is that all internal weights are determined by a handful of global, scalar parameters, thereby avoiding problems during backpropagation and reducing training time significantly. Because this architecture is conceptually simple, many scientists implement ESNs from scratch, leading to questions about computational performance. Xesn offers a straightforward, standard implementation of ESNs that operates efficiently on CPU and GPU hardware. The package leverages optimization tools to automate the parameter selection process, so that scientists can reduce the time finding a good architecture and focus on using ESNs for their domain application. Importantly, the package flexibly handles forecasting tasks for out-of-core, multi-dimensional datasets, eliminating the need to write parallel programming code. Xesn was initially developed to handle the problem of forecasting weather dynamics, and so it integrates naturally with Python packages that have become familiar to weather and climate scientists such as xarray (Hoyer & Hamman, 2017). However, the software is ultimately general enough to be utilized in other domains where ESNs have been useful, such as in signal processing (Jaeger & Haas, 2004).

## Statement of Need

ESNs are a conceptually simple Recurrent Neural Network architecture, leading many scientists who use ESNs to implement them from scratch. While this approach can work well for low dimensional problems, the situation quickly becomes more complicated when:

1. deploying the code on GPUs,
2. interacting with a parameter optimization algorithm in order to tune the model, and
3. parallelizing the architecture for higher dimensional applications.

Xesn is designed to address all of these points. Additionally, while there are some design flexibilities for the ESN architectures, the overall interface is streamlined based on the parameter and design impact study shown by Platt et al. (2022).

## GPU Deployment

At its core, xesn uses NumPy (Harris et al., 2020) and SciPy (Virtanen et al., 2020) to perform array based operations on CPUs. The package then harnesses the CPU/GPU agnostic code capabilities afforded by CuPy (Okuta et al., 2017) to operate on GPUs.

## Parameter Optimization

Although ESNs do not employ backpropagation to train internal weights, their behavior and performance is highly sensitive to a set of 5 scalar parameters. Moreover, the interaction of these parameters is often not straightforward, and it is therefore advantageous to optimize these parameter values (Platt et al., 2022). Additionally, Platt et al. (2023) and Smith et al. (2023) showed that adding invariant metrics to the loss function, like the leading Lyapunov exponent or the Kinetic Energy spectrum, improved generalizability. As a generic implementation of these metrics, xesn offers the capability to constrain the system's Power Spectral Density during parameter optimization in addition to a more traditional mean squared error loss function.

Xesn enables parameter optimization by integrating with the Surrogate Modeling Toolbox (Bouhlef et al., 2020), which has a Bayesian optimization implementation. Xesn provides a simple interface so that the user can specify all of the settings for training, parameter optimization, and testing with a single YAML file. By doing so, all parts of the experiment are more easily reproducible and easier to manage with scheduling systems like SLURM on HPC environments or in the cloud.

## Scaling to Higher Dimensions

It is typical for ESNs to use a hidden layer that is  $\mathcal{O}(10 - 100)$  times larger than the input and target space, so forecasting large target spaces quickly becomes intractable with a single reservoir. To address this, Pathak et al. (2018) developed a parallelization strategy so that multiple, semi-independent reservoirs make predictions of a single, high dimensional system. This parallelization was generalized for multiple dimensions by Arcomano et al. (2020) and Smith et al. (2023), the latter of which serves as the basis for xesn.

Xesn enables prediction for multi-dimensional systems by integrating its high level operations with xarray (Hoyer & Hamman, 2017). As with xarray, users refer to dimensions based on their named axes. Xesn parallelizes the core array based operations by using dask (Dask Development Team, 2016; Rocklin, 2015) to map them across available resources, from a laptop to a distributed HPC or cloud cluster.

## Existing Reservoir Computing Software

It is important to note that there is already an existing software package in Python for Reservoir Computing, called ReservoirPy (Trouvain et al., 2020). To our knowledge, the purpose of this package is distinctly different. The focus of ReservoirPy is to develop a highly generic framework for Reservoir Computing, for example, allowing one to change the network node type and graph structure underlying the reservoir, and allowing for delayed connections. On the other hand, xesn is focused specifically on implementing ESN architectures that can scale to multi-dimensional forecasting tasks. Additionally, while ReservoirPy enables hyperparameter grid search capabilities via Hyperopt (Bergstra et al., 2013), xesn enables Bayesian optimization as noted above.

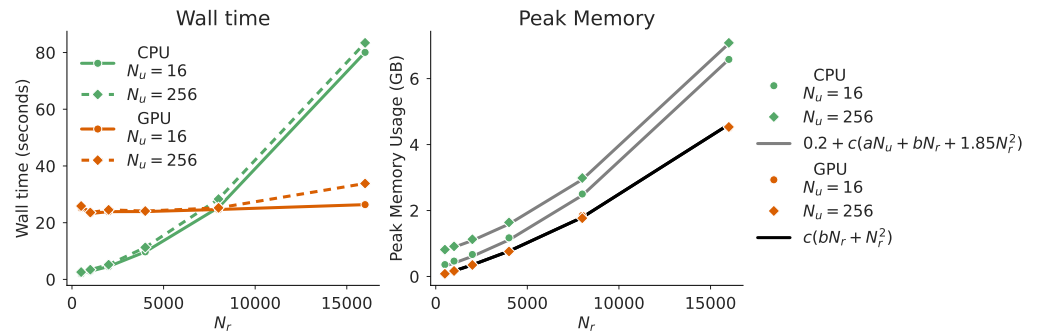
Another ESN implementation is that of (Arcomano et al., 2020, 2022, 2023), available at (Arcomano, 2023). The code implements ESNs in Fortran, and focuses on using ESNs for hybrid physics-ML modeling.

## Computational Performance

Here we show brief scaling results in order to show how the standard (eager) `xesn.ESN` scales with increasing hidden and input dimensions. Additionally, we provide some baseline results to serve as guidance when configuring dask to use the parallelized `xesn.LazyESN` architecture. The scripts used to setup, execute, and visualize these scaling tests can be found [here](#). For

methodological details on these two architectures, please refer to [the methods section of the documentation](#).

## Standard (Eager) ESN Performance



**Figure 1:** Wall time and peak memory usage for the standard ESN architecture for two different system sizes ( $N_u$ ) and a variety of reservoir sizes ( $N_r$ ). Wall time is captured with Python's time module, and peak memory usage is captured with [memory-profiler](#) for the CPU runs and with [NVIDIA Nsight Systems](#) for the GPU runs. Note that the peak memory usage for the GPU runs indicates GPU memory usage only, since this is a typical bottleneck. The gray and black lines indicate the general trend in memory usage during the CPU and GPU simulations, respectively. The empirically derived gray and black curves are a function of the problem size, and are provided so users can estimate how much memory might be required for their applications. The constants are as follows:  $a = 250,000$  is  $\sim 3$  times the total number of samples used,  $b = 20,000$  is the batch size, and  $c = 8 \cdot 10^9$  is a conversion to GB.

For reference, in [Figure 1](#) we show the wall time and peak memory usage required to train the standard (eager) ESN architecture as a function of the input dimension  $N_u$  and reservoir size  $N_r$ . We ran the scaling tests in the us-central-1c zone on Google Cloud Platform (GCP), using a single c2-standard-60 instance to test the CPU (NumPy) implementation and a single a2-highgpu-8g (i.e., with 8 A100 cards) instance to test the GPU (CuPy) implementation. The training data was generated from the Lorenz96 model ([Lorenz, 1996](#)) with dimensions  $N_u = \{16, 256\}$ , and we generated 80,000 total samples in the training dataset.

In the CPU tests, wall time scales quadratically with the reservoir size, while it is mostly constant on a GPU. For this problem, it becomes advantageous to use GPUs once the reservoir size is approximately  $N_r = 8,000$  or greater. In both the CPU and GPU tests, memory scales quadratically with reservoir size, although the increasing memory usage with reservoir size is more dramatic on the CPU than GPU. This result serves as a motivation for our parallelized architecture.

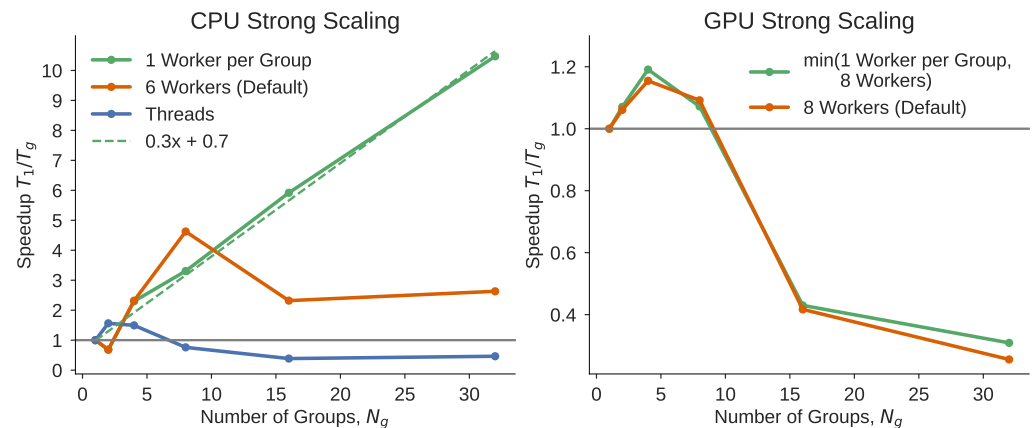
## Parallel (Lazy) Architecture Strong Scaling Results

In order to evaluate the performance of the parallelized architecture, we take the Lorenz96 system with dimension  $N_u = 256$  and subdivide the domain into  $N_g = \{2, 4, 8, 16, 32\}$  groups. We then fix the problem size such that  $N_r \cdot N_g = 16,000$ , so that the timing results reflect strong scaling. That is, the results show how the code performs with increasing resources on a fixed problem size, which in theory correspond to Amdahl's Law ([Amdahl, 1967](#)). The training task and resources used are otherwise the same as for the standard ESN results shown in [Figure 1](#). We then create 3 different `dask.distributed` Clusters, testing:

1. Purely threaded mode (CPU only).
2. The relevant default "LocalCluster" (i.e., single node) configuration for our resources. On the CPU resource, a GCP c2-standard-60 instance, the default `dask.distributed.LocalCluster` has 6 workers, each with 5 threads. On the GPU

resource, a GCP a2-highgpu-8g instance, the default `dask_cuda.LocalCUDACluster` has 8 workers, each with 1 thread.

3. A `LocalCluster` with 1 dask worker per group. On GPUs, this assumes 1 GPU per worker and we are able to use a maximum of 8 workers due to our available resources.



**Figure 2:** Strong scaling results, showing speedup as a ratio of serial training time to parallel training time as a function of number of groups or subdomains of the Lorenz96 system. Serial training time is evaluated with  $N_u = 256$  and  $N_r = 16,000$  with `xesn.ESN` from Figure 1, and parallel training time uses `xesn.LazyESN` with the number of groups as shown. See text for a description of the different schedulers used.

Figure 2 shows the strong scaling results of `xesn.LazyESN` for each of these cluster configurations, where each point shows the ratio of the wall time with the standard (serial) architecture to the lazy (parallel) architecture with  $N_g$  groups. On CPUs, using 1 dask worker process per ESN group generally scales well, which makes sense because each group is trained entirely independently.

On GPUs, the timing is largely determined by how many workers (GPUs) there are relative to the number of groups. When the number of workers is less than the number of groups, performance is detrimental. However, when there is at least one worker per group, the timing is almost the same as for the single worker case, only improving performance by 10-20%. While the strong scaling is somewhat muted, the invariance of wall time to reservoir size in Figure 1 and number of groups in Figure 2 means that the distributed GPU implementation is able to tackle larger problems at roughly the same computational cost.

## Acknowledgements

T.A. Smith and S.G. Penny acknowledge support from NOAA Grant NA20OAR4600277. S.G. Penny and J.A. Platt acknowledge support from the Office of Naval Research (ONR) Grants N00014-19-1-2522 and N00014-20-1-2580. T.A. Smith acknowledges support from the Cooperative Institute for Research in Environmental Sciences (CIRES) at the University of Colorado Boulder. The authors thank the editor Jonny Saunders for comments that significantly improved the manuscript, and the reviewers Troy Arcomano and William Nicholas.

## References

- Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, 483–485. <https://doi.org/10.1145/1465482.1465560>

- Arcomano, T. (2023). *Arcomano1234/SPEEDY-ML: V1 - GRL Paper*. Zenodo. <https://doi.org/10.5281/zenodo.7508156>
- Arcomano, T., Szunyogh, I., Pathak, J., Wikner, A., Hunt, B. R., & Ott, E. (2020). A Machine Learning-Based Global Atmospheric Forecast Model. *Geophysical Research Letters*, 47(9), e2020GL087776. <https://doi.org/10.1029/2020GL087776>
- Arcomano, T., Szunyogh, I., Wikner, A., Hunt, B. R., & Ott, E. (2023). A Hybrid Atmospheric Model Incorporating Machine Learning Can Capture Dynamical Processes Not Captured by Its Physics-Based Component. *Geophysical Research Letters*, 50(8), e2022GL102649. <https://doi.org/10.1029/2022GL102649>
- Arcomano, T., Szunyogh, I., Wikner, A., Pathak, J., Hunt, B. R., & Ott, E. (2022). A Hybrid Approach to Atmospheric Modeling That Combines Machine Learning With a Physics-Based Numerical Model. *Journal of Advances in Modeling Earth Systems*, 14(3), e2021MS002712. <https://doi.org/10.1029/2021MS002712>
- Bergstra, J., Yamins, D., & Cox, D. (2013). Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In S. Dasgupta & D. McAllester (Eds.), *Proceedings of the 30th international conference on machine learning* (Vol. 28, pp. 115–123). PMLR. <https://proceedings.mlr.press/v28/bergstra13.html>
- Bouhlel, M. A., He, S., & Martins, J. R. R. A. (2020). Scalable gradient-enhanced artificial neural networks for airfoil shape design in the subsonic and transonic regimes. *Structural and Multidisciplinary Optimization*, 61(4), 1363–1376. <https://doi.org/10.1007/s00158-020-02488-5>
- Dask Development Team. (2016). *Dask: Library for dynamic task scheduling*. <https://dask.org>
- Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H. van, Brett, M., Haldane, A., Río, J. F. del, Wiebe, M., Peterson, P., ... Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825), 357–362. <https://doi.org/10.1038/s41586-020-2649-2>
- Hoyer, S., & Hamman, J. (2017). Xarray: N-D labeled Arrays and Datasets in Python. *Journal of Open Research Software*, 5(1). <https://doi.org/10.5334/jors.148>
- Jaeger, H. (2001). The "echo state" approach to analysing and training recurrent neural networks – with an Erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, 148(34), 13.
- Jaeger, H., & Haas, H. (2004). Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication. *Science*, 304(5667), 78–80. <https://doi.org/10.1126/science.1091277>
- Lorenz, E. (1996). Predictability - a problem partly solved. *Proceedings of a Seminar Held at ECMWF on Predictability*.
- Okuta, R., Unno, Y., Nishino, D., Hido, S., & Loomis, C. (2017). CuPy: A NumPy-compatible library for NVIDIA GPU calculations. *Proceedings of Workshop on Machine Learning Systems (LearningSys) in the Thirty-First Annual Conference on Neural Information Processing Systems (NIPS)*. [http://learningsys.org/nips17/assets/papers/paper\\_16.pdf](http://learningsys.org/nips17/assets/papers/paper_16.pdf)
- Pathak, J., Hunt, B., Girvan, M., Lu, Z., & Ott, E. (2018). Model-Free Prediction of Large Spatiotemporally Chaotic Systems from Data: A Reservoir Computing Approach. *Physical Review Letters*, 120(2), 024102. <https://doi.org/10.1103/PhysRevLett.120.024102>
- Platt, J. A., Penny, S. G., Smith, T. A., Chen, T.-C., & Abarbanel, H. D. I. (2022). A systematic exploration of reservoir computing for forecasting complex spatiotemporal dynamics. *Neural Networks*, 153, 530–552. <https://doi.org/10.1016/j.neunet.2022.06.025>

- Platt, J. A., Penny, S. G., Smith, T. A., Chen, T.-C., & Abarbanel, H. D. I. (2023). *Constraining Chaos: Enforcing dynamical invariants in the training of recurrent neural networks*. arXiv. <https://doi.org/10.48550/arXiv.2304.12865>
- Rocklin, Matthew. (2015). Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In Kathryn Huff & James Bergstra (Eds.), *Proceedings of the 14th Python in Science Conference* (pp. 126–132). <https://doi.org/10.25080/Majora-7b98e3ed-013>
- Smith, T. A., Penny, S. G., Platt, J. A., & Chen, T.-C. (2023). Temporal Subsampling Diminishes Small Spatial Scales in Recurrent Neural Network Emulators of Geophysical Turbulence. *Journal of Advances in Modeling Earth Systems*, 15(12), e2023MS003792. <https://doi.org/10.1029/2023MS003792>
- Trouvain, N., Pedrelli, L., Dinh, T. T., & Hinaut, X. (2020). ReservoirPy: An efficient and user-friendly library to design echo state networks. In *Artificial neural networks and machine learning – ICANN 2020* (pp. 494–505). Springer International Publishing. [https://doi.org/10.1007/978-3-030-61616-8\\_40](https://doi.org/10.1007/978-3-030-61616-8_40)
- Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., ... SciPy 1.0 Contributors. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17, 261–272. <https://doi.org/10.1038/s41592-019-0686-2>