ReservoirPy: an Efficient and User-Friendly Library to Design Echo State Networks

Nathan Trouvain, Luca Pedrelli, Thanh Trung Dinh, Xavier Hinaut

ReservoirPy: an Efficient and User-Friendly Library to Design Echo State Networks

Nathan Trouvain[1,2,3][0000−0003−2121−7826], Luca Pedrelli[1,2,3][0000−0002−4752−7622], Thanh Trung Dinh[1,2,3][0000−0003−0249−2080], and Xavier Hinaut[1,2,3,*][0000−0002−1924−1184]

[1] INRIA Bordeaux Sud-Ouest, France.
[2] LaBRI, Bordeaux INP, CNRS, UMR 5800.
[3] Institut des Maladies Neurodégénératives, Université de Bordeaux, CNRS, UMR 5293.
*Corresponding author: xavier.hinaut@inria.fr

Abstract. We present a simple user-friendly library called *ReservoirPy* based on Python scientific modules. It provides a flexible interface to implement efficient Reservoir Computing (RC) architectures with a particular focus on Echo State Networks (ESN). Advanced features of *ReservoirPy* allow to improve up to 87.9% of computation time efficiency on a simple laptop compared to basic Python implementation. Overall, we provide tutorials for hyperparameters tuning, offline and online training, fast spectral initialization, parallel and sparse matrix computation on various tasks (MackeyGlass and audio recognition tasks). In particular, we provide graphical tools to easily explore hyperparameters using random search with the help of the *hyperopt* library.

Keywords: Reservoir Computing · Echo State Networks · Offline Learning · Online Learning · Hyperparameter Optimization · Parallel Computing · Sparse Matrix Computation · Toolbox.

1 Introduction

Reservoir Computing (RC) [16,12] is a paradigm to train Recurrent Neural Networks (RNN), while not as popular as fully trained neural networks typically used in Deep Learning. It is attractive given the good performance/computation cost ratio, and it is even at the state-of-the-art for several timeseries tasks [12]. Echo State Networks (ESN) [8] is the most well known instance of Reservoir Computing paradigm. While programming a basic ESN is relatively easy – requiring a hundred lines of code for the MackeyGlass timeseries prediction task[4] – having a complete customizable ESN framework error-prone and including hyperparameter optimization requires more effort. Therefore, we want to provide new users or regular ones an easy to handle and flexible library for Echo

[4] See for instance the minimal version of Mantas Lukoševičius saved at `https://mantas.info/code/simple_esn` or reproduced in *examples* directory of *ReservoirPy*: `https://github.com/neuronalX/reservoirpy/tree/master/examples`.

State Networks, and more generally extensible to Random RNN-based methods. While it is still in active development, it already includes several useful and advanced features, such as various methods for offline and online learning, parallel computation, and efficient sparse matrix computation. Importantly, we provide integrated graphical tools to easily perform what is usually time-consumming for each new task: explore the influence of various hyperparameters (e.g. spectral radius, input scaling, ...) on the performance of a given task. Moreover, we would like to emphasize the educational aspects of *ReservoirPy*, simple to manage for beginners, and for experts it is easy to build more complex architectures like deep or hierarchical reservoirs [4,13].

A decade ago, some integrated libraries were available, like *Oger* [5] in Python language, or *aureservoir* [7] in C++. Several projects on ESNs can be found on Github[6]. However, there is currently no equivalent library to Oger. Existing Python libraries either use specific frameworks such as PyTorch, or custom implementations. In order to have a general, flexible and easily extendable programming library for RC, which encourages collaboration and educational purposes, we developed *ReservoirPy*. Indeed, reservoir computing is an intuitive way to dive into the processing of timeseries with RNNs; compared to less intuitive training methods used in Long Short Term Memory (LSTM) for instance.

Moreover, we provide visualisation methods for hyperparameter exploration that ease this dive into reservoirs for newcommers, and which is insightful for experts. Several members of our team and students already used it for different tasks and purposes (e.g. to build Computational Neuroscience models and Human-Robot Interaction modules [6,11,14]), it is now time to share it more extensively.

2 The *ReservoirPy* library

2.1 Features summary

ReservoirPy can be accessed here: `https://github.com/neuronalX/reservoirpy`
The library provides several features:

- general features: **washout**, **input bias**, **readout feedback**, **regularization coefficient**, ...;
- custom or general **offline or online training methods** (e.g. other methods available in *scikit-learn*)
- save and load of ESNs in a readable structure;
- **parallel computation** of reservoir states for independent timeseries (with *jolib* library);
- **sparse matrix** computation (using scipy.sparse);
- **fast spectral initialization** [5];
- tools for easy **hyperparameter exploration** (with *hyperopt* [3]).

[5] Oger is no longer maintained; archived at `https://github.com/neuronalX/Oger`
[6] See for instance `https://github.com/topics/echo-state-networks`

Several tutorials and demos are provided (see section 5.4), along with a documentation. *Nota Bene*: In the following when we say "*train the reservoir*" we mean "*train the readout (i.e. output weights) of the reservoir*"; the internal recurrent connections of the reservoir are always kept fixed throughout the paper.

2.2 Precisions on Online learning feature

Alongside with offline learning, *ReservoirPy* also provides the ability to perform online (incremental) learning. Given a sequence of inputs, online learning allows to train the reservoir sequentially on each time step, avoiding storing all data in memory and making matrix inversion on large matrices. Thus, online learning proposes a lighter approach to train reservoir with less computational demand while still achieving compatible level of accuracy. More importantly perhaps, online incremental learning methods are crucial for computational neuroscience models [14] and developmental experiments in cognitive science (developmental psychology, robotics, ...) [6,11]. Current implementation of online learning in *ReservoirPy* is based on FORCE learning method [15], and resides in a separate class: `ESNOnline`. More details on FORCE can be found in Appendix 8.3.

3 Getting Started with *ReservoirPy*

In this section, we introduce how to use basic features of *ReservoirPy*.

3.1 Requirements

Basic ReservoirPy (requirements.txt): numpy, joblib, scipy, tqdm. Advanced features to use notebooks and hyperperameters optimization (examples.txt, requirements.txt): hyperopt, pandas, matplotlib, seaborn, scikit-learn. Installation instructions are given in appendix 8.1.

3.2 Prepare your dataset

```
data = np.loadtxt('MackeyGlass_t17.txt').reshape(-1, 1)
# inputs and teachers for training and testing
x_train, y_train = data[0:train].T, data[1:train+1]
x_test, y_test = data[train:train+test], data[train+1:train+test+1]
```

3.3 Generate random matrices

The `mat_gen` module contains functions to create new input, feedback and internal weights matrices, control spectral radius, modify sparsity and add bias.

```
from reservoirpy import mat_gen
W = mat_gen.generate_internal_weights(...)
Win = mat_gen.generate_input_weights(...)
# optionnaly, generate a feedback matrix Wfb
```

3.4 Offline training

Set a custom offline reservoir ESN can be created using various parameters, allowing to set the leaking rate `leak_rate`, the regularization coefficient value `regularization_coef`, feedback between outputs and the reservoir, an activation function for feedback, or a reference to a Scikit-Learn linear regression model (respectively `lr`, `ridge`, `Wfb`, `fbfunc` and `reg_model` arguments).

```
from reservoirpy import ESN
esn = ESN(leak_rate, W, Win, input_bias, regularization_coef, ...)
# Additional parameters: Wfb, fbfunc, reg_model, use_raw_input
```

Train and test the reservoir The `train` method can handle a sequence of inputs to train a readout matrix `Wout`, using various linear regression methods. The `run` method can then output the readout values from any sequence of inputs. Internal states generated by the reservoir during both processes are returned by all functions. `wash_nr_timesteps` argument also allows to consider only the states generated after a warmup phase for training, ensuring to use only dynamics generated from the input itself and not the initial zero state.

Inputs should be lists of time series. Each time series will be used to compute the corresponding internal states. Between each time series, the internal states of the reservoir are reinitialized, or can be reset to particular values.

```
# training
states_train = esn.train(inputs=[x_train,], teachers=[y_train,],
                         wash_nr_timesteps=100)
# testing
out_pred, states_pred = esn.run(inputs=[test_in,], reset_state=False)
print("Root Mean Squared error:")
print(np.sqrt(np.mean((out_pred[0] - y_test)**2)) / test_len)
```

3.5 Online learning

A custom reservoir needs to be instantiated for online learning. Then, the reservoir can be trained and tested in the same way as for offline learning.

`alpha_coef` is needed to initialize P(0), where P is used in equations (1) and (2) (see *8.3*), more information on `alpha_coef` can be found in the FORCE learning paper. `Wout` needs to be initialized in the online version, because the modification of the weights starts since the beginning of the training. `Wout` could be initialized with null matrix.

```
from reservoirpy import ESNOnline
Wout = ... #initializaton of Wout
esn = ESNOnline(... alpha_coef, Wout, ...) # other parameters are the same
```

4 A tutorial to explore visually hyperparameters

4.1 Random-search vs. Grid-Search

Setting a reservoir is easy, but training it optimally (or with good enough performance) requires some expertise. Novices and experts' first reaction is to tune parameters by hand, in order to get "some insights" on the influence of parameters. Many users will try grid-search to find which hyperparameters produce a good performance. Indeed, grid-search can be useful to have a global understanding on the influence of hyperparameters. However, in the following we show that this can be done with random exploration as well, especially if you get help from some graphical tools, such as the one we provide in *ReservoirPy*.

More importantly, as Bergstra et al. show [2], grid-search is *suboptimal* compared to random search. Indeed, as shown in Figure 1, grid-search undersamples the hyperparameter space compared to random-search. This undersampling comes from the fact that grid-search repeatedly tests the same values for one given hyperparameter while changing other hyperparameters. Therefore, grid-search "looses" time (i.e. useful samples) when changing values of unimportant hyperparameters while keeping fixed important hyperparameters. Consequently, random-search obtains better results by sampling more values of important hyperparameters.

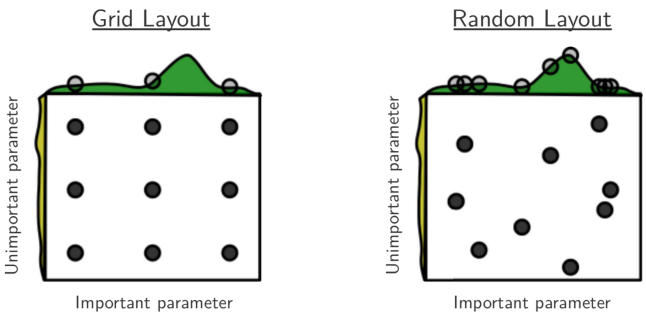

Fig. 1. Why *random search* is better than *grid search*? With *random search* one is able to obtain more samples from the *important parameters*, because with *grid search* one undersamples the space by repeatedly sampling the same values for *important parameters*. Image from [2].

4.2 Integrated graphical toolbox

In order to make hyperparameters optimisation more efficient and less time-consuming, we integrate a simple toolbox in *ReservoiPy*. It relies on the widely used *hyperopt* [3] and *Matplotlib* Python libraries. This toolbox provides users

with research management and visual tools adapted to the exploration of hyperparameters spaces of ESN. We will present these tools through a minimalist experiment with random search, over a regression task with Mackey-Glass time series (see also subsection 5.4). For this tutorial we will use the famous Mackey-Glass task in the RC community: the aim is to perform chaotic timeseries prediction.

4.3 Set the experiment

The first step consists of defining an *objective* function, based on the parameters we want to explore. This function describes the experiment the user wants to perform. Within this function, the model is instantiated, trained and tested using the parameters yielded by the optimization algorithm. The function should then return a quantitative evaluation of the subsequent model performances when receiving a combination of parameters and input data.

In this example, the objective function returns to *hyperopt* the mean-squared error (MSE) over the testing set, which is the required loss metric for *hyperopt* optimisation. In addition to the loss metric, any other metric can be added in the returned dictionary, by inserting another named item storing its value. Additional metrics can give significant insights on how hyperparameters influence the final results of the model. For the sake of the example we added the root mean-squared error (RMSE) as an additional metric. The codomain of loss functions should preferentially be $\mathbb{R}_+$ and they should have a reachable local or global minimum within the range of explored parameters. If these conditions are not reached, the trials results may be hard to analyse, and show no interesting properties. In this case, the range of parameters defined should be considered as sub-optimal. Additional metrics functions codomain should be $[0; 1]$. Otherwise, the visualisation tool will normalize the results by default to ensure that the figure is readable, which may cause substantial loss of information and add bias when interpretation the figure.

We call one *hyperparameter (hp) combination* (or *hp configuration*) the set of hyperparameters passed to the objective function: the result returned will be represented by one point of data in Figure 2. During a hp search, the hp combinations should preferably be computed and averaged on several reservoir instances: i.e. it is preferable that the objective function returns the average loss obtained from different reservoir instances for the same set of hyperparameters instead of the loss for one single reservoir instance. As the performance varies from one reservoir instance to another, averaging over 5 or 10 instances is a good compromise between representativeness of results and overall computation time. In the example depicted in Figure 2, we set `instances_per_trial` to 10. In case only one instance is used, the resulting performance of the hp configuration could not be trusted. In any case, one should not trust blindly to the best hp combination found by *hyperopt*, but rather take the 1 or 2% best configurations and think of it as a range of values for which hyperparameters are optimal. Additionally, this procedure provides more robustness to the parameters found.

```
def objective(train_d, test_d, config, *, iss, N, sr, leak, ridge):
# the empty starred expression is mandatory in objective function
# definition. It separates hyperopt keyword arguments (right)
# and required arguments (left).

    # unpack train and test data, with target values.
    x_train, y_train = train_d # proprocessing could be done here
    x_test, y_test = test_d    # if parametric

    # train and test an 'insts' number of ESN
    # This value can be extrated from the config
    insts = config["instances_per_trial"] # = 10

    mses = []; rmses = [];
    for i in range(insts):
        W = ...; Win = ...; reservoir = ...;
        reservoir.train(inputs=[x_train], teachers=[y_train])
        outputs, _ = reservoir.run(inputs=[x_test])
        mses.append(mse(outputs[0], y_test))
        rmses.append(sqrt(mse(outputs[0], y_test)))

    # return a dictionary of averaged metrics.
    # The 'loss' key is mandatory when using hyperopt.
    return {'loss': np.mean(mses)
            'rmse': np.mean(rmses)}
```

4.4 Define the parameters spaces

The next step is to declare the exploration space of parameters inside a JSON structured file, named *configuration file*. This convention allows to keep track of all the parameters used for each exploration, and to uniquely identify every experiments. It is important when doing random search explorations that all choices of parameter ranges are detailed and saved to make the experiments fully reproducible (e.g. define parameters that are kept constant like the number of neurons N in our example). The configuration file has the following structure, and should always begin with an unique experiment name:

```
{   "exp": "hyperopt-mackeyglass-1",
    "hp_max_evals": 1000,
    "hp_method": "random",
    "instances_per_trial": 10,
    "hp_space": {
        "N": ["choice", 300],
        "sr": ["loguniform", 1e-6, 10],
        "leak": ["loguniform", 1e-3, 1],
        "iss": ["choice", 1.0],
        "ridge": ["loguniform", 1e-8, 1] }    }
```

Not all parameters are tested at the same time. To maximize the chance to obtain interesting results, we advise to keep some parameters constant. This will minimize the number of covariant interactions, which are difficult to analyse

(e.g. spectral radius `sr`, leak-rate `leak` and input scaling `iss` are often interdependent). In this example, only spectral radius, leaking rate and regularization coefficient (respectively `sr`, `leak` and `ridge`) are set with an active exploration space. Other fields are used to configure the *hyperopt* module, setting the optimization algorithm – random search in this case – and the number of trials – one thousand. For example, these parameters could also set the number of initial random trials of *hyperopt* (`n_startup_jobs`) when using the TPE (Tree-Parzen Estimator) Bayesian optimizer (see [3] for more details). All these parameters are defined accordingly to *hyperopt* conventions.

4.5 Launch the trials

Then, we call the `research` function to run the experiment. The function will call *hyperopt* algorithm and automatically save the results of each trial as JSON structured files in a report directory. Objective function is passed as argument, with the dataset and the paths to configuration files and report directory.

```
best = research(loss_ESN, dataset,
                config="examples/mackeyglass-config.json",
                report="examples/report")
```

4.6 Display the results

After the end of the random search, all results can be retrieved from the report directory and displayed on a scatter plot, using the `plot_opt_results` function (fig. 2). This function will load the results and extract the parameters and additional metric the user wants to display, specified with the `params` and `metric` arguments. Other parameters can be used to adjust the figure rendering, for instance by removing outliers or switching scales from logarithmic to linear.

```
fig = plot_opt_results("examples/report/hpt-mg"),
params=["sr", "leak", "ridge"],  metric="rmse")
```

In this example, we use the MSE as loss metric and the RMSE as additional metric for display. The default behaviour of the function is to use loss as metric. Every plot in the figure show the interaction between each couple of parameters, weighted by the normalized loss value (gradient of color) and the normalized additional metric (size of dots). The plots displayed on the diagonal of the figure display the relation between the loss function and the parameters, with the top five percent of trials, regarding to the additional metric, displayed in shades of green. The plot given as example display interesting results: the loss function have a convex profile, and variations in dots density in cross parameters scatter plots indicate acceptable ranges of parameters, for both spectral radius and leaking rate. The regularization coefficient does not seem to play an important

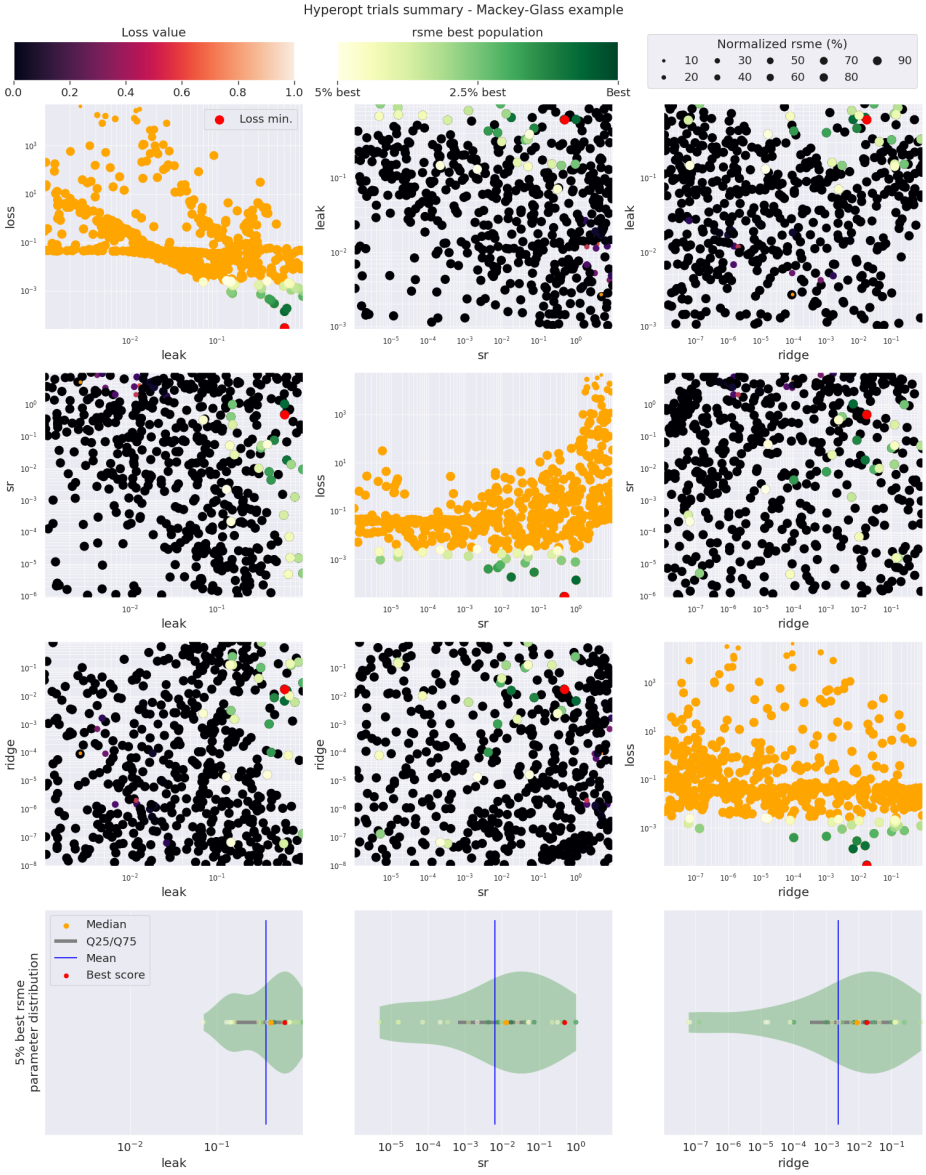

Fig. 2. An example of figure obtained after 1000 trials over Mackey-Glass time series. The random search was performed on spectral radius (sr), leaking rate (leak) and regularization parameter (ridge). MSE and RMSE are displayed as evaluation metrics. Each trial point represent the averaged evaluation metrics over 10 sub-trials. Each sub-trial was performed on the same parameters combination within each trial, but with different ESN instances (e.g. different random weights initialization).

role in model performance, and can therefore be fixed to a constant value for further explorations. Violin plots at the bottom of the figure can then help with choosing a range of acceptable parameters, by displaying the distribution of the top 5 percent of trials parameters regarding to the additional metric.

A more complex example of random search visualization can be found in Appendix 8.4.

5 Demo experiments

In this section, we provide applications on three tasks to showcase a selection of features. The first task is the well-known Mackey-Glass task: chaotic timeseries prediction (used in subsections 5.4). For the other tasks, we chose more computationally expensive tasks (bird and human audio recognition tasks) in order to better demonstrate the gain in computation time (used in subsections 5.1, 5.2 and 5.3). We used a canary song annotation corpus (we call it *Canary* dataset in the following) which contains about 3.5 hours of annotated canary songs (i.e. 1,043,042 MFCC frames), with 41 target classes. The speech recognition corpus TIMIT [1] is composed by 5.4 hours of spoken audio characterized by 5040 multidimensional time series with a total of 1,944,000 time steps.

5.1 Parallel computations

If the ESN is provided with a sequence of independent inputs during training or running (for example for an independent sequence classification task), the reservoir internal states can be computed in parallel. The parallel computation can be enabled by setting the `workers` parameters to a value > 1 in the `train` and `run` methods. The `backend` parameter also allows to seamlessly control the module used for parallel computation by the *joblib* package. To ensure minimal performance overhead across all hardware environments, we recommend users to keep the default *threading* backend.

```
# setting workers at -1 will use all available threads/processes
# for computation. Backend can be switched to any value proposed
# by joblib ("threading","multiprocessing", "loki"...)
states_train = esn.train(..., workers=-1, backend="threading")
```

5.2 Sparse matrix computation

In order to address applications characterized by medium/big datasets, the state computation of the network is implemented considering sparse matrix operations. Here, we show the improvement in terms of efficiency obtained by the sparse computation of the network's state on two audio datasets, the *Canary* and *TIMIT* datasets. Table 1 shows the time spent (in seconds) by the network in the state computation on Canary and TIMIT datasets by using parallelization

Task	Dense – Serial	Dense – Parallel	Sparse – Serial	Sparse – Parallel
Canary	621 sec. (-)	442 sec. (28.82 %)	503 sec. (19.00 %)	380 sec. (38.81 %)
TIMIT	849 sec. (-)	627 sec. (26.15 %)	191 sec. (77.50 %)	103 sec. (87.87 %)

Table 1. Comparison in terms of efficiency considering parallelization and sparse recurrent matrices for the state computation of the network on Canary and TIMIT datasets by using 1000 units and 10% of sparsity, with and without parallel computation enabled. Performance was measured with an *Intel Core i7-8650U*, 1.90GHz with 8 cores using the Canary dataset, and with an *Intel Core i5*, 2,7 GHz with 2 cores using TIMIT dataset. The percentage of improvement is indicated by taking the *Dense – Serial* case as baseline.

and sparse recurrent matrices with 1000 units and 10% of sparsity. Interestingly, the sparse computation allows the network to significantly improve the efficiency. In particular, it obtains an improvement of 19.00% and 77.50% in terms efficiency w.r.t. the dense computation on Canary and TIMIT tasks, respectively. Overall, by combining the parallel and the sparse approach, the network obtains a very good improvement spending of 38.81% and 87.87% in terms of efficiency w.r.t. the baseline case on Canary and TIMIT tasks, respectively.

Units	FSI	Eigen – Sparse	Eigen – Dense
1000	0.042 sec.	0.319 sec.	1.341 sec.
2000	0.226 sec.	1.475 sec.	7.584 sec.
5000	1.754 sec.	21.238 sec.	128.419 sec.

Table 2. Comparison in terms of efficiency among FSI, eigen-sparse and eigen-dense by using 1000, 2000 and 5000 recurrent units and 10% of sparsity. Performance was measured with an *Intel Core i5*, 2,7 GHz with 2 cores.

5.3 Fast Spectral Initialization

In the RC context, the recurrent weights are typically initialized by performing the spectral radius through eigenvalues computation. This can be expensive when the application needs large reservoirs or a wide model selection of hyperparameters. A very efficient initialization approach to address these cases is called Fast Spectral Initialization (FSI) [5]. Here, we compare the Python implementation of the FSI approach integrated in this library with the typical methods based on eigenvalues computation in sparse (**eigen − sparse**) and dense (**eigen − dense**) cases typically used to initialize recurrent weights. Table 2 shows the time (in seconds) spent by FSI, eigen-sparse and eigen-dense considering 1000, 2000 and 5000 recurrent units and 10% of sparsity. As expected, FSI obtains an extremely better efficiency w.r.t. the typical initialization approaches which is progressively enhanced when the number of units increases.

5.4 Online learning

To demonstrate that online training with FORCE learning method is competitive, we trained a reservoir and evaluated it on the Mackey-Glass task with FORCE learning (with and without feedback). In addition, results are compared with the offline learning case. Surprisingly, online learning method obtains slightly better result than offline learning.

Method	**NRMSE** (10^{-3})
Online learning (with feedback)	3.47 ($\pm$0.09)
Online learning (without feedback)	4.39 ($\pm$0.26)
Offline learning	6.06 ($\pm$1.67)

Table 3. Comparison of online learning and offline learning on Mackey-Glass task. For each cell: mean ($\pm$ standard deviation) averaged on 30 reservoir instances. Hyperparameters are the same as the best results for the experiment performed in section 4.6 with `sr` = 0.5, `leak` = 0.6 and `ridge` = 0.02. Normalized Root Mean Square Error (NRMSE).

6 Ongoing and Future Work

In the future, there is several other features we want to include in *ReservoirPy*: more use-case examples (e.g. generative mode, hyperparameter search for more tasks, ...); more online learning methods (e.g. LMS); GPU computations (e.g. CuPy, JAX, ...); offline batch computation; batch direct approach for ridge regression[7]; framework to build layers of reservoirs (e.g. deep reservoirs [4], hierarchical-task reservoirs [13]); Conceptors [9]; scikit-learn API compatibility.

7 Conclusion

We presented the *ReservoirPy*: a simple and user-friendly library for training Echo State Networks, and soon more models of Random Recurrent Neural Networks. It provides a balance between a flexible tool, based on pure Python library using only scientific libraries, and a computational effective one (parallel implementation, sparse matrix computations, ...), without the burden of a complex framework such as TensorFlow or PyTorch.

The library includes several features that enables to computations more efficient. By using sparse and parallel computations we showed computation time

[7] An approach to incrementally compute the normal equations matrices in ridge regression. This allows the learning algorithm to compute the readout weights by saving memory in the case of large datasets.

improvement from 38.8% to 87.9% depending on the dataset and the CPU. Moreover, we provided a tutorial to explore efficiently hyperparameters with a graphical tools.

References

1. Garofolo et al., J.: Timit acoustic-phonetic continuous speech corpus. Linguistic Data Consortium LDC93S1 (1993)
2. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. Journal of machine learning research **13**(Feb), 281–305 (2012)
3. Bergstra, J., Yamins, D., Cox, D.D.: Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In: Proceedings of the 12th Python in Science Conference. pp. 13–20 (2013)
4. Gallicchio, C., Micheli, A., Pedrelli, L.: Deep reservoir computing: a critical experimental analysis. Neurocomputing **268**, 87–99 (2017)
5. Gallicchio, C., Micheli, A., Pedrelli, L.: Fast spectral radius initialization for recurrent neural networks. In: INNS BDDL (2020)
6. Hinaut, X., Spranger, M.: Learning to parse grounded language using reservoir computing. In: 2019 Joint IEEE 9th International Conference on Development and Learning and Epigenetic Robotics (ICDL-EpiRob) (Aug 2019)
7. Holzmann, G.: Efficient c++ library for analog reservoir computing neural networks (echo state networks). http://aureservoir.sourceforge.net (2007-2008)
8. Jaeger, H.: The "echo state" approach to analysing and training recurrent neural networks. Tech. Rep. 148, German National Research Center for Information Technology GMD, Bonn, Germany (2001)
9. Jaeger, H.: Controlling recurrent neural networks by conceptors. arXiv preprint arXiv:1403.3369 (2014)
10. Jaeger, H., Lukoševičius, M., Popovici, D., Siewert, U.: Optimization and applications of echo state networks with leaky-integrator neurons. Neural Networks **20**(3), 335–352 (Apr 2007)
11. Juven, A., Hinaut, X.: Cross-situational learning with reservoircomputing for language acquisition modelling. In: IJCNN (2020)
12. Lukoševičius, M., Jaeger, H.: Reservoir computing approaches to recurrent neural network training. Computer Science Review **3**(3), 127–149 (2009)
13. Pedrelli, L., Hinaut, X.: Hierarchical-task reservoir for anytime POS tagging from continuous speech. In: IJCNN (2020)
14. Strock, A., Hinaut, X., Rougier, N.P.: A robust model of gated working memory. Neural Computation **32**(1), 153–181 (Jan 2020)
15. Sussillo, D., Abbott, L.: Generating coherent patterns of activity from chaotic neural networks. Neuron **63**(4), 544–557 (Aug 2009)
16. Verstraeten, D., Schrauwen, B., d'Haene, M., Stroobandt, D.: An experimental unification of reservoir computing methods. Neural Networks **20**(3), 391–403 (2007)

8 Appendices

8.1 *ReservoirPy* installation

ReservoirPy can be installed easily with `pip`[8]:

[8] If you want to be able to modify the code of reservoirpy, debug it (hopefully it would not be necessary), or extend it, use the '-e' option. You will be able to inspect the

```
git clone https://github.com/neuronalX/reservoirpy
# Simple installation
pip install reservoirpy
# Developper installation
pip install -e <path/to/reservoipy/root/directory>
```

8.2 Echo State Network architecture

Echo State Networks (ESNs) are a class of Recurrent Neural Networks (RNNs) implemented according to Reservoir Computing (RC) paradigm. Figure 3 shows an example of ESN architecture. It is composed by a recurrent layer called *reservoir* and an output layer called *readout*. The reservoir is randomly initialized and left untrained, while, the readout weights are trained through *offline* learning or *online* learning. Please, see [8,10] for more information about ESN model.

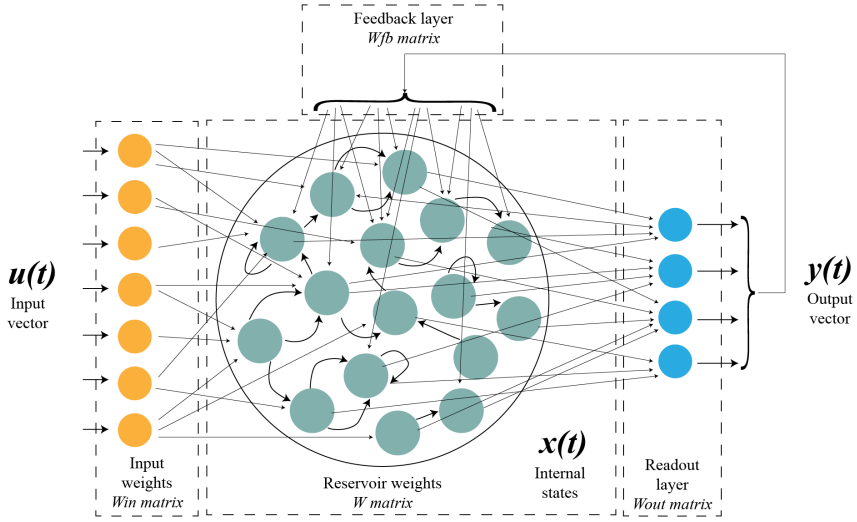

Fig. 3. Schematic representation of an ESN. For each time step t, an input vector $u(t)$ is fed to the model through the input matrix Win. The internal states vector $x(t)$ results from the inner dynamics of the reservoir and their reaction with the input data. A matrix $Wout$ then compute a readout from the state vector, to produce the $\hat{y}(t)$ output vector. Optionally, this vector can then be fed back to the reservoir, as a feedback vector for the next update of internal states with the vector $u(t+1)$.

code during execution with any debugger, and changes will be applied without any need to reinstall the package.

8.3 Online learning: Details on FORCE learning

With FORCE learning[9], the output weights matrix (W_{out}) is updated for each time step, so as to keep prediction error as small as possible. The update of W_{out} is governed by equation (1), where $e\_(t)$ is the difference between the prediction output and ground truth at time t (i.e. prediction error), $r(t)$ is the state vector (of the reservoir) and $P(t)$ is computed via equation (2).

$$W_{out}(t) = W_{out}(t - \Delta t) - e\_(t)P(t)r(t) \tag{1}$$

$$P(t) = P(t - \Delta t) - \frac{P(t - \Delta t)r(t)r^T(t)P(t - \Delta t)}{1 + r^T(t)P(t - \Delta t)r(t)} \tag{2}$$

8.4 Example of random search visualization on the Canary dataset

The following plot was made using the same tool presented in 4.4. A random search is performed to find optimal ranges of parameters for a classification task over acoustic data representing canary songs. This data is fed to the ESN as two different vectors of features: a first vector represents the first order derivatives of an MFCC signal extracted from the acoustic data, and the second vector the second order derivatives of this MFCC signal. Each of these feature vectors have their own input scaling parameter, respectively `isd` and `isd2`. This visualization allows to quickly distinguish the best parameters range to use. Importantly, it gives insights on interactions between the two input scaling parameters used for the task and the leaking rate.

[9] FORCE is a 2^{nd} order learning method similar to RLS (Recursive Least Squares), contrary to LMS (Least Mean Squares) which is 1^{nd} order.

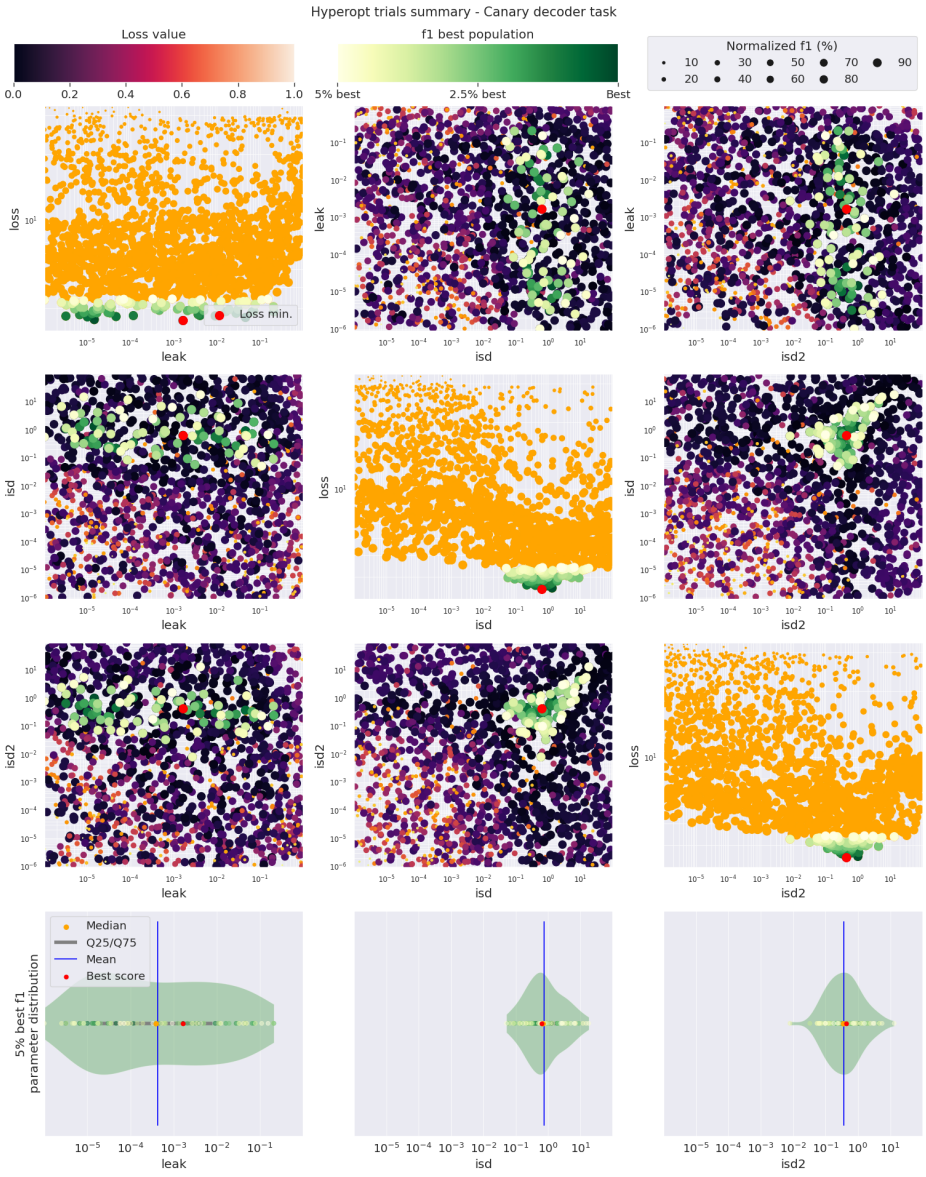

Fig. 4. An example of figure obtained after 1000 trials over the Canary dataset. The random search was performed on leaking rate (leak) and input scaling coefficients (isd and isd2), used to adjust two different sets of features. Cross-entropy and F1-score are displayed as evaluation metrics. For this experiment, the number of units was constant (N = 300), as the spectral radius (sr = 0.4) and the regularization coefficient (ridge = 1.10^{-7}).