Create Efficient and Complex Reservoir Computing Architectures with ReservoirPy ⋆

Nathan Trouvain[1,2,3][0000−0003−2121−7826], Nicolas Rougier[1,2,3][0000−0002−6972−589X], 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. Reservoir Computing (RC) is a type of recurrent neural network (RNNs) where learning is restricted to the output weights. RCs are often considered as temporal Support Vector Machines (SVMs) for the way they project inputs onto dynamic non-linear high-dimensional representations. This paradigm, mainly represented by Echo State Networks (ESNs), has been successfully applied on a wide variety of tasks, from time series forecasting to sequence generation. They offer de facto a fast, simple yet efficient way to train RNNs.
We present in this paper a library that facilitates the creation of RC architectures, from simplest to most complex, based on the Python scientific stack (NumPy, Scipy). This library offers memory and time efficient implementations for both online and offline training paradigms, such as FORCE learning or parallel ridge regression. The flexibility of the API allows to quickly design ESNs including re-usable and customizable components. It enables to build models such as DeepESNs as well as other advanced architectures with complex connectivity between multiple reservoirs with feedback loops. Extensive documentation and tutorials both for newcomers and experts are provided through GitHub and ReadTheDocs websites.
The paper introduces the main concepts supporting the library, illustrated with code examples covering popular RC techniques from the literature. We argue that such flexible dedicated library will ease the creation of more advanced architectures while guarantying their correct implementation and reproducibility across the RC community.

Keywords: Reservoir Computing · Echo State Network · Recurrent Neural Networks · Python · Online Learning · Offline Learning · Toolbox.

1 Introduction

Within the field of Recurrent Neural Networks (RNNs), Reservoir Computing (RC) is an interesting paradigm of timeseries and sequence processing. Most of

⋆ Supported by Inria.

RC techniques rely on a *reservoir*, a pool of randomly – and recurrently – connected neurons, in charge of projecting data into a high dimensional space able to encode temporal information. This reservoir is connected to an output layer called *readout* whose role is to extract information from the reservoir activity. In a more formal way, the readout neurons act as a linear layer which can be used to perform regression or classification on high dimensional representations of any timeseries processed by the reservoir. As opposed to more popular Deep Learning strategies, Reservoir Computing techniques do not require gradient error backpropagation algorithm to work. The only trained connections being the readout connections, a simple linear regression can be computed between the activations of the reservoir and the desired target values to obtain a functional model.

Reservoir Computing is mostly known through its two first and most widely used instances: Echo State Networks (ESNs) [13] and Liquid State Machines (LSMs) [15], their spiking neural networks counterpart. Although existing since the beginning of the 2000s, RC techniques are less well-known compared to other RNN-based Deep Learning architectures like Long Short-Term Memory networks (LSTMs). In the meantime, they have been successfully applied to various tasks and problems (some are listed in this review by [25]) and even demonstrates state of the art performances for tasks such as chaotic timeseries forecasting [30] or sound processing [27]. It was shown that ESNs needed less data than LSTMs to obtain good performances while being trained in much less time (e.g. see [27]).

Several code implementations of RC, in particular for ESNs, can be found online, but these implementations are often isolated scripts written in Python or Matlab. They often provide reusable objects intended to allow reproduction of specific results and techniques, but do not offer any way to re-use, re-combine or extend their code. Whereas Deep Learning architectures have thrived, supported by complete, user-friendly toolboxes enabling such flexibility and re-usability, RC may remain an underground technique without this kind of off the shelf, ready to use and yet permissive programming frameworks and libraries. Libraries like *Oger* were successful attempts of creating a rather complete RC tool. However, *Oger* was originally written in Python 2 (whose support have ended in 2020) and its maintenance has not been continued.

ReservoirPy provides an implementation only relying on general scientific librairies like *Numpy* and *Scipy*, in order to be more versatile than specific frameworks (e.g. *TensorFlow*, *PyTorch*) and provide more flexibilty when building custom architectures. On the one hand, *TensorFlow* and *PyTorch* were mostly developed for gradient descent based learning algorithms, and most of their features are useless, if not cumbersome, to develop RC techniques. For instance, they are usually optimized to perform several epochs of training on batches of independent samples, or to differentiate any kind of operations happening under their scope, which is not often useful for RC. On the other hand, libraries such as *Scikit-Learn* [17] are geared towards static data processing and do not integrate much timeseries processing or online learning tools. Most importantly, we designed ReservoirPy as a highly flexible tool offering the possibility to design

several kinds of RC architectures while promoting reusable components. Indeed, numerous Reservoir Computing extensions and derivatives have been developed (see [23] for a recent review): they generally include modified learning methods and architectures and some of them allow for the composition of several reservoirs (decoupled-ESNs [31], tree ESNs [7], deep reservoirs [8], hierarchical-task reservoirs[18], and more exotic architectures like Reservoir-of-Reservoirs (RoR) [4] or self-supervised pairs of reservoirs [2]).

In this paper, we detail the backbone of ReservoirPy, its major components and how to combine them to design complex architectures, with different learning rules and feedback loops. We then introduce the library usage with a simple example of timeseries forecasting using online and offline trained ESNs. In Appendices[4], we give more details about more advanced features and minimal implementations of advanced or exotic RC architectures, such as Deep ESNs or Extreme Learning Machines (ELMs) [12], as a proof of concept of ReservoirPy capacities. Finally, we present future features and conclude about the potential of ReservoirPy on the RC community.

2 Flexible Reservoir Computing

Most RC techniques rely on interconnected and interchangeable building blocks: an ESN can be defined as a network connecting a reservoir of neurons to a layer of readout neurons. Inside an ESN, a reservoir can be connected to several independent readouts, and to several input sources. In more complex architectures, like DeepESNs [6] or Hierarchical ESNs [26,**?**], reservoirs can be layered, and readouts inserted between them, with feedback connections possibly connecting all these blocks in different ways. Some architectures might also use other additional blocks and operators than reservoirs and readouts.

Our tool enables such flexibility using 4 components.

2.1 Functional nodes

The minimal component is the `Node`. A Node is a Python class that can be equipped with several functions and parameters in order to operate on a timeseries.

All Nodes are recurrent operators, meaning that they at least carry two elements: (1) an internal state vector x_t storing their last output, accessible anytime through the `state()` method, and (2) a generic *forward* function defined as $\text{forward}(x_t, u_t) = x_{t+1}$. This function takes as input a single timestep of input data u_t and the current value of the Node's internal state x_t, and outputs an updated value x_{t+1} for this state. A new Node can either be created by passing a forward function and its parameters as argument to the Node class constructor, or by defining a new specialized class that inherits from the Node class. An

[4] Appendices are available at `https://github.com/reservoirpy/publications/SAB2022.pdf`

initialization function may also be added, to allow for the dynamic initialization of the Node's parameters as well as the inference of input and internal state vectors dimensions directly from data. Hence, the minimal creation of a new Node requires the declaration of two functions and an optional dictionary mapping parameters and hyperparameters names to their values, without enforcing any inheritance from the class itself.

For instance, our `Reservoir` Node implementation of a reservoir is a subclass of Node holding several parameters and hyperparameters, among which:

- a matrix $\mathbf{W}$, defining the connection weights between the reservoir neurons,
- a matrix $\mathbf{W}_{in}$ defining the connection weights between the input neurons and the reservoir neurons,
- a coefficient lr, called *leak rate*, which defines the time constant of the reservoir neurons.
- a function f, used as an activation function for the reservoir neurons, usually the hyperbolic tangent applied element-wise on the activation vector.

These parameters are used in the Reservoir *forward* function definition:

$$\text{forward}(x_t, u_t) = (1 - lr)x_t + lr f(\mathbf{W}x_t + \mathbf{W}_{in}u_t) = x_{t+1} \tag{1}$$

Nodes *forward* function can be triggered by calling the Node on a single data point like a Python function. Since Nodes are mainly designed to process timeseries or sequential data, it is also possible to use a *forward* function on several points of a timeseries, updating its Node internal state several times and gathering temporal information. This can be done using the `run()` method of a Node.

```
import numpy as np
from reservoirpy.nodes import Reservoir

# A Reservoir with 100 neurons and lr=0.1
res = Reservoir(units=100, lr=0.1) # Activation is tanh by default.

u = np.array([[1.0, 0.0]]) # 1 timestep of 2D data.
U = np.array([[1.0, 0.0],  # 2 timesteps of 2D data.
              [0.0, 1.0]])
s1 = res(u) # Update reservoir state on 1 timestep.
s = res.state() # Current state of the reservoir.

# Update reservoir state on a sequence of 3 timesteps.
S = res.run(U)

# Parameters can be accessed as attributes.
print(res.lr, res.Win)
```

In the code above, as the `Reservoir` class is already implemented as a subclass of Node within the library, internal code machinery like Reservoir's *forward* function is hidden. Only necessary hyperparameters, like the number of neurons inside the reservoir, were given to the Reservoir constructor. All other parameters were initialized when the Node was first used, i.e. when `s1 = res(u)` was executed. This allows the Node to infer the shape of all other parameters like the input matrix $\mathbf{W}_{in}$ based on data dimension and to build them using initialization functions.

2.2 Learning rules

The second major component of the library are the learning rules. A learning rule can be declared as a function that takes sequences of data as arguments and updates the Node parameters. Once learning rules functions have been loaded into a Node, it is possible to use the same mechanisms described in the previous section about *forward* function definition. There is is two main ways of using them:

Offline learning A learning rule is said to be *offline* if the parameters estimation of the learned model is performed only once, on a single corpus of data, and cannot be modified later. This is for instance the case for the L2-regularized linear regression, also called *ridge regression* or Tikhonov regression, widely used in Reservoir Computing to train ESNs, and implemented by our library in the `Ridge` Node. Nodes equipped with offline learning rules can be trained using the `fit()` method.

```
import numpy as np
from reservoirpy.nodes import Ridge

readout = Ridge()

# Update parameters of the Node.
readout.fit(X_train, y_train)
```

Additionally, offline learning can be performed incrementally or using batches of data to pre-compute some parts of the learning process. A `partial_fit()` method can be defined in order to perform such operations on chunks of the dataset.

Online learning *Online* learning rules describe continuous learning processes, during which the learned parameters are updated as soon as new data is fed to the model. This learning procedure is more biologically relevant since the model does not learn its parameters in one single step but rather in small successive steps, trying to improve its predictions to minimize a cost function. Taking inspiration from this idea of progressive training, Nodes carrying an online learning rule can update their parameters using a method named `train()`. The `FORCE` Node is an example of Node trained using the online learning rule described by [24].

```
import numpy as np
from reservoirpy.nodes import FORCE

readout = FORCE()

# Update parameters of the Node once.
s0 = readout.train(one_x, one_y)

# Update parameters of the Node
# on a sequence (several times in a row)
S = readout.train(X_train, Y_train)
```

Note that the `train()` method can be called on single timesteps of data, triggering a single step of learning. Calling this methods returns an array holding the output of the Node *forward* function, before learning was applied. This

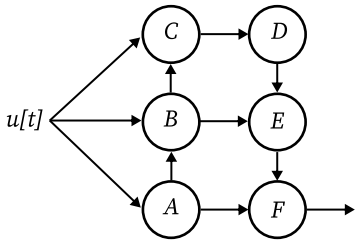

Fig. 1. An advanced example of Model graph, involving 6 interconnected Nodes.

method can also be called on sequences of data. In such case, the Node parameters evolve at each timestep of data in the sequence, and the method call will return the sequence of responses obtained from each training step.

2.3 Models as computational graphs

While Nodes offer a way to quickly define reusable operations and estimators, *Models* allow to compose these operations and create complex architectures involving several Nodes in interaction. We define Models as a simple mean of managing computational graphs, without any heavy pre-existing framework.

Models are subclasses of Node, and therefore expose the same interface: function-like calls, `run`, `fit` and the `train` method of a Model triggers one-timestep update of all Nodes in the Model, sequential update of all Nodes in the Model, fitting of all offline learner Nodes and training of all online learner Nodes, respectively.

Models are created by linking Nodes using the "`>>`" Python operator. The *forward* function of a Model created by linking a Node A to a Node B will be defined as the composition $f_{Model} = f_B \circ f_A$ of A and B forward functions. However, Nodes involved in a Model do not store any reference to other Nodes in the Model. That is, Nodes can be shared between different Models without requiring to copy or to reinitialize them. This feature provides a flexible way of defining Reservoir Computing architectures involving several readouts or several pathways, allowing to train Nodes using a Model and to run them using another.

```
model1 = nodeA >> nodeB  # A Model with two Nodes.

# Another Model. nodeA state and parameters are
# shared with model1.
model2 = nodeA >> nodeC

# The result of nodeB(nodeA(u)):
s = model1(u)
```

Since Models are a Node subclass, they can also be linked to other Nodes, allowing to chain the linking operator. Models can also be merged using the "`&`" Python operator. Merging a Model A and a Model B will create a third model containing an unique version of all the Nodes present in A and B, along with the union of all their connections. This allows to design architectures with

several pathways, as shown in the code below, defining the graph of Nodes in fig:complex-graph. Note the usage of `Input()` and `Output()` Nodes in order to clearly define the input and output of the graph. These Nodes have no effect on the Model behavior other than forcing Nodes A, B and C to receive inputs from the input source, and F to be the final output of the Model. Note also that many-to-one and one-to-many connections can be declared by connecting Nodes to lists of Nodes and vice-versa.

```
from reservoirpy.nodes import Input, Output

path1 = A >> F
path2 = B >> E

# One-to-many connection using a list
path3 = Input() >> [nodeA, nodeB, nodeC]

# Chain of connections
path4 = A >> B >> C >> D >> E >> F >> Output()

# Merge all pathways to create Fig. 1 graph:
model = path1 & path2 & path3 & path4
```

One limitation of the Model object is the necessity for the declared computational graph to be a directed acyclic graph of Nodes. If this condition is not met, Nodes cannot be topologically sorted and operation order within the Model *forward* function is undefined. Nevertheless, this condition can be skirted using feedback connections between Nodes in the graph as explained in sec:feedback.

Once a Model has been defined, it can be either fitted offline or trained online using the same methods as for a Node. Models training procedure is identified using "duck-typing". If all trainable Nodes (e.g. all readout Nodes such as `Ridge` or `FORCE`) in the Model exposes an offline learning interface, then the Model is considered to be an offline learner. Similarly, if all trainable Nodes are online learners, then the Model is considered to be an online learner. For now, mixing different learning procedure is not an allowed behavior, although it can be achieved by splitting the Model into different pathways with different learning rules.

2.4 Feedback loops

RC models might require time-delayed connection between different components of neural networks or computational models. Usually, these connections are used to connect readout layer of neurons to reservoir neurons in a feedback loop, in order to tame reservoir neurons activities using the output signal of the network.

Our library differentiates itself from most other RC tools by providing a rather simple and flexible interface to define such delayed connections, using a similar operation than the Model definition in sec:model. A feedback connection between two Nodes can be defined using the "`<<`" Python operator. This operation must be performed on the receiver Node, and will create a copy of the Node storing the feedback link. This mechanism is different from the linking mechanism used to define Models. Feedback connections are stored in the

form of a reference to the signal sender within the signal receiver. This implies that feedback connections can only be defined once on a receiver Node, instead of being decoupled from the Node object like regular connections stored in a Model.

```
nodeA = nodeA << nodeB # Copying nodeA.
nodeA <<= nodeB # Using in place modification.
```

Feedback connections may need an initialization step. Reservoirs, for instance, receive the feedback signal through neuronal connections whose weights are stored in a $\mathbf{W}_{fb}$ matrix. Feedback connections initialization functions can be created and loaded into Nodes to tackle these situations. Plus, feedback signal may come from any type of Node, including Models. It is therefore possible to design complex feedback graphs to transform the feedback signal before it reaches the receiver Node, or to get feedback from several Nodes gathered in a Model at once.

Once a feedback connection has been defined, a feedback receiver Node can retrieve the current feedback signal sent by the connected Node using the method `feedback()`. This method will fetch the state of the feedback sender Node, or the state of the outputs Nodes of the feedback sender Model. When running or training a Model over a sequence of data, and if the feedback sender and receiver Nodes are part of the Model, then feedback will be sent through the connection while respecting a one timestep delay between the sender and the receiver. For instance, if a reservoir and a readout are connected to form a Model, and if the reservoir receives feedback from the readout, then the reservoir will receive at t inputs u_t and feedback signal y_{t-1}, y being the state (or output) of the readout.

In addition to setup regular feedback connections as defined in RC, feedback connections mechanism can be hijacked to build teacher Nodes or reward Nodes, and help building architectures based on online learning such as the model from [2] (see sec:asabuki), or help implementing reward-modulated online learning rule like the 3-factor Hebbian learning rule proposed by [11] (ongoing work). These reward or teaching Nodes can be used to provide a connected Node with some target values for training at runtime, even if these targets values are not available *before* runtime, e.g. if these values are computed by some part of a Model and used to train some other part.

3 Getting started: ESN for timeseries forecasting with ReservoirPy

This section introduces how to use ReservoirPy to define, train and run Echo State Networks (ESN) for some classic literature benchmarks. Since ESN is among the most used techniques of Reservoir Computing, ReservoirPy introduces special optimizations to increase their performances and leverage larger corpus of data. Reservoirs and readouts objects also provide users with many options to precisely tune and adapt their behavior to different needs. For example, it is possible to switch the `Reservoir` *forward* function between two different

definitions, the first one applying leaky integration to neurons states after applying the activation function, like in [6] or [13], the second one before like in [2] or [5]. Other tunable parameters include spectral radius of the recurrent matrix $\mathbf{W}$, input and feedback scaling, random weights distribution, leaking rate, additive noise in the input, internal states and feedback, and many more. On top of these included features, users may also define their own Nodes and Models using the interface described in sec:flexible-rc

3.1 Step 1: Choose a timeseries for one timestep ahead prediction

ReservoirPy contains 7 timeseries generators, and this number is steadily increased with each new release. Currently available timeseries generators are listed in sec:datasets.

For this tutorial, we draw 2000 points of the well-known Mackey-Glass timeseries, used in many RC benchmarks. We then split this timeseries in order to create two series shifted in time by one timestep. The goal of our task will thus be to predict u_{t+1} knowing u_t.

```
from reservoirpy.datasets import mackey_glass

# tau=17 series is chaotic
X = mackey_glass(2000, tau=17)

test_len = 500 # Split for training/testing.
X_train, y_train, X_test, y_test = ...
```

3.2 Step 2: Define your ESN

An ESN can be defined as a simple Model with two Nodes: a *reservoir*, connected to a *readout*. We will create two Models for this tutorial: an ESN geared with an offline learning rule, and an ESN equiped with an online learning rule. Both can share the same reservoir. Inside the 100 neurons reservoir, spectral radius of matrix $\mathbf{W}$ is set to 0.9, input scaling of $\mathbf{W}_{in}$ to 0.1, leaking rate to 0.3, and a Gaussian noise with a gain of 0.01 is added to the inputs. These hyperparameters may be suboptimal, and are just provided for the sake of example. We also set the offline readout regularization parameter to 10^{-6} (also called *ridge*).

```
from reservoirpy.nodes import Reservoir, Ridge, FORCE

reservoir = Reservoir(100, sr=0.9, lr=0.3,
                      input_scaling=0.1,
                      noise_in=0.01)

off_readout = Ridge(ridge=1e-6) # Offline readout
on_readout  = FORCE()           # Online readout

esn_off = reservoir >> off_readout # Offline ESN
esn_on  = reservoir >> on_readout  # Online ESN
```

3.3 Step 3.1: Train the offline model

The offline learner ESN can be trained using the `fit()` method of the Model. Because this method call will also be the first use of our ESN on our dataset,

`fit()` will also trigger all initialization functions available in the Nodes, building random parameters matrices using the previously defined spectral radius and input scaling for instance. These functions will also infer the input and output dimension of all Nodes in the Model, i.e. an input dimension of 1 and an output dimension of 1 as the input of our Model is the 1-dimensional Mackey-Glass timeseries and the output is a 1-dimensional forecast of this timeseries.

```
esn_off.fit(X_train, y_train)
```

3.4 Step 3.2: Train the online model

Whereas offline learning can only be performed once *via* the `fit()` method, online learning may happen several times in the life cycle of a Model. Hence, the `train()` method can be invoked several times in a row to sequentially train the online learner ESN. For the sake of example, let's create a simple for-loop updating the Model's parameters several time on only one timestep of data at a time.

```
Y = []
for x, y in zip(X_train, y_train):
    # Nodes and Models only accept 2D arrays
    # as parameters, so reshaping is important.
    y_hat = esn_on.train(x.reshape(1, -1),
                         y.reshape(1, -1)
    Y.append(y_hat)
```

Models can also be trained in one line of code be providing the `train()` method with a sequence of inputs and targets values.

```
Y = esn_on.train(X_train, y_train)
```

3.5 Step 4: Evaluate the model

Once trained, Models or Nodes can be run to outputs predictions. ReservoirPy exposes some common metrics in the `obersvables` module to evaluate this kind of tasks, such as Root-Mean Square Error (RMSE).

```
from reservoirpy.observables import rmse

y_pred_off = esn_offline.run(X_test)
y_pred_on  = esn_online.run(X_test)

print("RMSE offline: ", rmse(y_test, y_pred_off))
>>> RMSE offline: ...
print("RMSE online: ", rmse(y_test, y_pred_on))
>>> RMSE offline: ...
```

4 Discussion

ReservoirPy is a Python library for Reservoir Computing architectures, from ESNs to deep ESNs, providing users with online and offline learning rules, complete feedback loop support, and a powerful syntax to quickly develop any kind

of model using reusable building blocks. We demonstrated its ability to handle exotic architectures of reservoirs where learning could be performed in unusual ways. Such architectures provide new ideas "to think how learning could be performed", which is particularly interesting in computational neuroscience were people try to understand how different learning and memory mechanisms interact in the brain [1]. Some of these works studied how a couple of reservoirs could learn to train one another to find chunks in a sequence [2] or how to model working memory in reservoirs [16][22]. More generally, we are aiming at replication and implementation of additional tools emerging from literature, like new learning rules such as Intrinsic Plasticity (unsupervised training of reservoir units) [19] or the three-factor Hebbian learning rule from [11], and new results on non-linear vector autoregressive machines equivalence with reservoirs from [10]. Other ongoing and future work will also focus on improving general performance and usability of the tool and add spiking version of reservoirs. We thus plan on integrating parallel computation procedures to any kind of model, and study the possibility to perform some computations on graphical processing units (GPU) to speed up linear algebra operations for large sized reservoirs.

ReservoirPy has been built without strongly enforcing design principles from other classical libraries (e.g. *TensorFlow* or *Scikit-Learn*). This choice has been motivated by the will to offer a high degree of flexibility, tailored to RC techniques. Such flexibility and design choice allow for rapid and efficient prototyping of original reservoir architectures. We believe this is an important step towards the development of a new family of more complex reservoir architectures, such as Deep Learning ones (e.g. Transformers [28]). Indeed, future trends in machine learning are probably in-between Reservoir Computing and Deep Learning approaches where parts of advanced models are kept untrained [20][21].

ReservoirPy is a community oriented project: we provide tutorials and extensive documentation. We welcome any feedback or contribution, from improvement of the code base to implementation of new tools, or publication of new examples and use cases.

References

1. Alexandre, F., Hinaut, X., Rougier, N., Viéville, T.: Higher cognitive functions in bio-inspired artificial intelligence. ERCIM News **125** (2021)
2. Asabuki, T., Hiratani, N., Fukai, T.: Interactive reservoir computing for chunking information streams. PLoS computational biology **14**(10), e1006400 (2018)
3. Bergstra, J., et al.: Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In: SciPy. pp. 13–20 (2013)
4. Dale, M.: Neuroevolution of hierarchical reservoir computers. In: Proceedings of the Genetic and Evolutionary Computation Conference. pp. 410–417 (2018)
5. Enel, P., Procyk, E., Quilodran, R., Dominey, P.: Reservoir computing properties of neural dynamics in prefrontal cortex. PLoS Comput Biol **12**(6), e1004967 (2016)
6. Gallicchio, C., Micheli, A., Pedrelli, L.: Deep reservoir computing: a critical experimental analysis. Neurocomputing **268**, 87–99 (2017)
7. Gallicchio, C., Micheli, A.: Tree echo state networks. Neurocomputing **101**, 319–337 (2013)

8. Gallicchio, C., Micheli, A., Pedrelli, L.: Deep reservoir computing: A critical experimental analysis. Neurocomputing **268**, 87–99 (2017)
9. Gallicchio, C., Micheli, A., Pedrelli, L.: Fast spectral radius initialization for recurrent neural networks. In: INNS Big Data and Deep Learning conference. pp. 380–390. Springer (2019)
10. Gauthier, D.J., Bollt, E., Griffith, A., Barbosa, W.A.S.: Next generation reservoir computing **12**(1), 5564
11. Hoerzer, G.M., Legenstein, R., Maass, W.: Emergence of Complex Computational Structures From Chaotic Neural Networks Through Reward-Modulated Hebbian Learning **24**(3), 677–690
12. Huang, G.B., Wang, D.H., Lan, Y.: Extreme learning machines: A survey. Int. J. Mach. Learn. & Cyber. **2**(2), 107–122 (Jun 2011)
13. Jaeger, H.: The "echo state" approach to analysing and training recurrent neural networks. Bonn, Germany: German National Research Center for Information Technology GMD Tech. Report **148**, 34 (2001)
14. Lukoševičius, M.: A practical guide to applying echo state networks. In: Neural Networks: Tricks of the Trade, pp. 659–686. Springer (2012)
15. Maass, W., Natschläger, T., Markram, H.: Real-time computing without stable states: A new framework for neural computation based on perturbations. Neural computation **14**(11), 2531–2560 (2002)
16. Pascanu, R., Jaeger, H.: A neurodynamical model for working memory. Neural networks **24**(2), 199–207 (2011)
17. Pedregosa, F., et al.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
18. Pedrelli, L., Hinaut, X.: Hierarchical-task reservoir for online semantic analysis from continuous speech. IEEE TNNLS pp. 1–10 (2021)
19. Schrauwen, B., Wardermann, M., Verstraeten, D., Steil, J.J., Stroobandt, D.: Improving reservoirs using intrinsic plasticity **71**(7), 1159–1171
20. Shen, S., Baevski, A., Morcos, A.S., Keutzer, K., Auli, M., Kiela, D.: Reservoir transformers. arXiv preprint arXiv:2012.15045 (2020)
21. Shrivastava, H., Garg, A., Cao, Y., Zhang, Y., Sainath, T.: Echo state speech recognition. In: ICASSP. pp. 5669–5673. IEEE (2021)
22. Strock, A., Hinaut, X., Rougier, N.P.: A robust model of gated working memory. Neural Computation **32**(1), 153–181 (2020)
23. Sun, C., Song, M., Hong, S., Li, H.: A review of designs and applications of echo state networks. arXiv preprint arXiv:2012.02974 (2020)
24. Sussillo, D., Abbott, L.F.: Generating Coherent Patterns of Activity from Chaotic Neural Networks **63**(4), 544–557
25. Tanaka, G., et al.review **115**, 100–123
26. Triefenbach, F., Jalalvand, A., Schrauwen, B., Martens, J.: Phoneme recognition with large hierarchical reservoirs. In: NIPS. pp. 2307–2315 (2010)
27. Trouvain, N., Hinaut, X.: Canary song decoder: Transduction and implicit segmentation with esns and ltsms. In: ICANN. pp. 71–82. Springer (2021)
28. Vaswani, A., et al.: Attention is all you need. In: NIPS. pp. 5998–6008 (2017)
29. Virtanen, P., et al.: SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods **17**, 261–272 (2020)
30. Vlachas, P.R., et al.: Backpropagation algorithms and Reservoir Computing in Recurrent Neural Networks for the forecasting of complex spatiotemporal dynamics. Neural Networks **126**, 191–217
31. Xue, Y., Yang, L., Haykin, S.: Decoupled echo state networks with lateral inhibition. Neural Networks **20**(3), 365–376 (2007)

5 Appendix: Advanced features

In addition to the interface described in sec:flexible-rc, that encompasses most of RC enthusiasts needs from "classic" RC techniques to "deep" architectures training, ReservoirPy offers some more advanced tools, listed in this section.

Parallel execution A special Node `ESN` can be loaded with a reservoir and an offline readout such as the `Ridge` Node to speed up both training and inference of ESNs on large corpus of data. This is typically useful when dealing with language data or batched timeseries in general, where each independent sentence in the corpus can be processed in parallel by the reservoir while the readout gathers all internal states emitted and performs linear regression. Parallel execution is based on the *joblib* library, allowing to easily switch between different execution backends.

Memory efficient linear regression Linear regression for offline learning or readout weights is performed in ReservoirPy using the following equation:

$$\mathbf{W}_{out} = \mathbf{Y}\mathbf{X}^\top(\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{Id})^{-1} \tag{2}$$

where $\mathbf{X}$ is the *design matrix*, which stores all the reservoir activations computed over the training data in an ESN; $\mathbf{Y}$ is a matrix storing all the target vector for all the activation vectors in $\mathbf{X}$; $\mathbf{W}_{out}$ is the learned weight matrix; λ is a regularization parameter and $|\cdot|^{-1}$ stands for Moore-Penrose pseudo inversion of a matrix. The `Ridge` Node is optimized as proposed by [14], to avoid storing the full design matrix $\mathbf{X}$, which can occupy a huge space in memory if the dataset is long and/or if the states dimension is high. This optimization involves computing the $\mathbf{X}\mathbf{X}^\top$ and $\mathbf{Y}\mathbf{X}^\top$ terms "on the fly", using a summation of $\mathbf{X}_i\mathbf{X}_i^\top$ and $\mathbf{Y}_i\mathbf{X}_i^\top$ computed on batches (or single timesteps) of data i. These matrices have a maximum dimension of $N \times N$ and $O \times N$ respectively, where N is the dimension of the reservoir states (or more generally of the `Ridge` Node inputs) and O is the dimension of the target vectors $\mathbf{Y}_i$. This can be significantly more memory efficient than storing design matrices of dimension $N \times L$ or $O \times L$ where L is the length of the training dataset.

Custom matrix initialization Most Nodes accepts parameters functions or arrays to define their parameters, like the reservoir weight matrices. This allows to use ReservoirPy with handcrafted initialization rules. The module `mat_gen` also provides some specific initialization techniques, like fast scaling of spectral radius as presented by [9].

Hyperparameter tuning utilities We rely on *hyperopt* [3], a general purpose optimization library, to perform efficient searches of optimal hyperparameters. ReservoirPy provides users with helpers to handle *hyperopt* machinery, and some visualization tools, along with tutorials and examples.

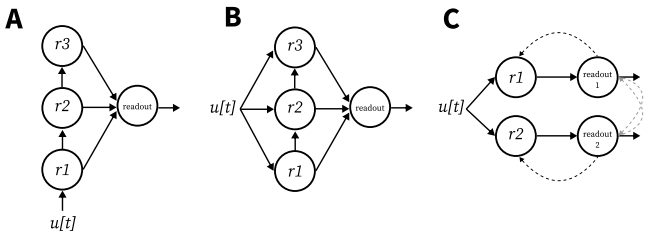

Fig. 2. Some advanced RC models architectures. Nodes $r1, r2$ and $r3$ are reservoirs. Plain arrows are direct connections at time step t. Dashed arrows are feedback connections from $t-1$ states to t. Gray dashed arrows are teacher connections. **A** DeepESN [6] **B** DeepESN with inputs-to-all connections [6] **C** Mutually supervised reservoirs [2].

Timeseries generators These generators can solve on demand the set of differential equations describing chaotic attractors using *Scipy* [29] tools such as `solve_ivp`, or yield values from a discrete timeseries defined by a recurrent relation. Users can refine their parameters, initial conditions, integration time step, and solving method when applicable. We provide:

- Lorenz attractor timeseries;
- Mackey-Glass timeseries;
- Doublescroll attractor timeseries;
- Rabinovich-Fabrikant attractor timeseries;
- Logistic map;
- Hénon map;
- NARMA discrete timeseries.

6 Appendix: Advanced architectures

This section presents some examples of advanced model architectures implemented with ReservoirPy. These examples are not meant to be full reproductions of any research paper, but rather seek to give clues on how to achieve some possibly advanced design patterns. The hereafter selected models have been encountered in the RC literature, and come with a minimal ReservoirPy architecture.

6.1 Deep Echo State Networks

Deep Echo State Networks (DeepESNs) have been first introduced by [6] in 2017 and have been since a quite trendy research topic in the RC community. They consist in several reservoirs connected together in a sequential way, and a readout taking as input all the reservoirs in the model. Assuming that we want to create a DeepESN with three reservoirs, and that all Nodes have been constructed with some relevant parameters, a minimal ReservoirPy infused script for a DeepESN Model would be:

```
deep_esn = r1 >> r2 >> r3  \ # Without input-to-all.
         & [r1, r2, r3] >> readout

inputs = Input() # Adding input-to-all.
deep_esn_ia = inputs >> [r1, r2, r3] & deep_esn
```

where `r1`, `r2` and `r3` are three `Reservoir` Nodes and `readout` might be a `Ridge` or a `FORCE` Node. This DeepESN Model can then be trained offline or online, using either its `fit()` or `train()` method. Corresponding graphs are displayed in fig:advancedA and fig:advancedB.

6.2 Extreme Learning Machines

While ReservoirPy project was not initially including Extreme Learning Machines (ELMs)[12], it appears that they can be designed using the tools described in this paper with a few tricks. Indeed, ELMs can be seen as a restricted version of ESNs, where the state recurrence and the leaky integration are removed from the reservoir. In that case, neurons in the reservoir respond without integrating temporal context, as if each timesteps of inputs was *i.i.d.* Recurrence can be shut down by setting the spectral radius of the reservoir recurrence matrix $\mathbf{W}$ to 0, and the leaking rate to 1. We also change the weight distribution in matrix $\mathbf{W}_{in}$, which are by default randomly chosen between 1 and -1, for a Gaussian distribution. Finally, we augment connectivity of the $\mathbf{W}_{in}$ matrix to 1 to depict a fully connected layer of neurons.

```
from reservoirpy.mat_gen import normal

# Create an 'i.i.d states' reservoir (sr=0.0)
static_res = Reservoir(100, lr=1.0, sr=0.0,
                       input_connectivity=1.0,
                       Win=normal)

# Connect with an offline readout.
readout = Ridge()
elm = static_res >> readout
```

6.3 Mutually supervised reservoirs

Asabuki et al. [2] proposed in 2018 a model able to perform chunking tasks on sequences, without explicit supervision [2]. This model is composed of two ESNs equiped with an online learning rule, and feedback connections going from the readouts to their respective reservoirs. The readouts were trained by *mutual supervision*: readout 1 has to learn how to predict a normalized response of readout 2, and readout 2 learns the opposite mapping (see fig:advancedC). In the following example, we propose an implementation of this model using *teacher Nodes*. Online learners may invoke their `train()` method using a Node as target. Each time the online learning Nodes train themselves, they first fetch a target value from this Node, using a mechanism similar to feedback connections. We will not detail the exact implementation of this model. In the following code, we assume that we have properly parametrized reservoir Nodes and online readout Nodes and that we have defined an additional `Normalize` subclass of Node able to normalize the responses of the readouts as explained in [2].

```
reservoir1 = reservoir1 << readout1 # Feedbacks.
reservoir2 = reservoir2 << readout2

model = reservoir1 >> readout1 \ # Model.
      & reservoir2 >> readout2

teacher1 = read1 >> Normalize() # Teacher Nodes,
teacher2 = read2 >> Normalize() # can be Models.

# Training Model. We assume that readout were
# defined with names, here "readout1" and "readout2".
model.train(X, Y={"readout1": teacher2,
                  "readout2": teacher1}
```