Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

PyRCN: A toolbox for exploration and application of Reservoir Computing Networks

Peter Steiner [a],[*],[1], Azarakhsh Jalalvand [b],[c],[1], Simon Stone [a], Peter Birkholz [a]

[a] *Institute of Acoustics and Speech Communication, Technische Universität Dresden, Dresden, Germany*
[b] *IDLab, Ghent University–imec, Ghent, Belgium*
[c] *Mechanical and Aerospace Engineering Department, Princeton University, NJ, USA*

ARTICLE INFO

ABSTRACT

Reservoir Computing Networks (RCNs) belong to a group of machine learning techniques that project the input space non-linearly into a high-dimensional feature space, where the underlying task can be solved linearly. Popular variants of RCNs are capable of solving complex tasks equivalently to widely used deep neural networks, but with a substantially simpler training paradigm based on linear regression. In this paper, we show how to uniformly describe RCNs with small and clearly defined building blocks, and we introduce the Python toolbox PyRCN (Python Reservoir Computing Networks) for optimizing, training and analyzing RCNs on arbitrarily large datasets. The tool is based on widely-used scientific packages and complies with the scikit-learn interface specification. It provides a platform for educational and exploratory analyses of RCNs, as well as a framework to apply RCNs on complex tasks including sequence processing. With a small number of building blocks, the framework allows the implementation of numerous different RCN architectures. We provide code examples on how to set up RCNs for time series prediction and for sequence classification tasks. PyRCN is around ten times faster than reference toolboxes on a benchmark task while requiring substantially less boilerplate code.

1. Introduction

Reservoir Computing Networks (RCNs) summarize a variety of machine learning techniques that use random, non-linear projections of inputs into a high-dimensional feature space. This often greatly facilitates classification tasks, because classes that are not linearly separable in the original input space, may become linearly separable in the high-dimensional space. The simplest Reservoir Computing architecture is the Extreme Learning Machine (ELM) (Huang et al., 2006), which is similar to a conventional Feed-Forward neural network. The more common RCN architectures, however, are Echo State Networks (ESNs) (Jaeger, 2001) and Liquid State Machines (LSMs) (Maass et al., 2002), which are variants of Recurrent Neural Networks (RNNs). The basic idea of RCNs is rather simple: The input nodes are randomly connected to a single hidden layer, the so-called "reservoir", consisting of non-linear neurons. Only connections from the hidden layer to the output are trained, typically using linear regression. In case of ELMs, the neurons in the hidden layer are not inter-connected and the basic ELM is thus closely related to Multilayer Perceptrons (MLPs). In case

of ESNs, the hidden layer is a pool of randomly interconnected non-linear neurons. LSMs are essentially ESNs with spiking neuron models. Based on the number of citations of their landmark papers, RCNs are not as commonly used as other types of neural networks, such as MLPs, Convolutional Neural Networks (CNNs) or networks consisting of Long-Short-Term-Memory (LSTM) cells, but they have achieved results that rival those and other popular deep-learning architectures. For example, ESNs have been successfully used for speech (Triefenbach et al., 2013), images (Jalalvand et al., 2018), radar (Jalalvand et al., 2019), plasma control (Jalalvand et al., 2021), music (Steiner et al., 2020, 2021), time-series prediction (Trierweiler Ribeiro et al., 2021; Moreno et al., 2020), electrical load and energy (consumption) forecasting (Mansoor et al., 2021; da Silva et al., 2021; Wang et al., 2018) and for robot control (Salmen and Ploger, 2005; Oubbati et al., 2006; Schrauwen et al., 2007; Antonelo et al., 2008; Antonelo and Schrauwen, 2015). ELMs have been successfully used for image classification (Huang et al., 2006), gesture recognition (Katılmış and Karakuzu, 2021), pressure prediction (Cocco Mariani et al., 2019), heartbeat classification (Ding et al., 2014), and for non-linear regression (Tang et al., 2016).

---

* Corresponding author.
E-mail addresses: peter.steiner@tu-dresden.de (P. Steiner), azarakhsh.jalalvand@ugent.be (A. Jalalvand), simon.stone@tu-dresden.de (S. Stone), peter.birkholz@tu-dresden.de (P. Birkholz).
[1] Equal contribution.

Table 1

Overview of important RCN toolboxes, download links, the year of the last major update and whether the toolboxes provide a high-level interface. The implementation of LSMs is currently under development (UD).

Toolbox	Online source	Supported RCNs			Last	High-level
		ELM	ESN	LSM	updated	interface
ESNToolbox (Jaeger, 2001)	ESNToolbox (2009)	–	✓	–	2009	–
DeepESN Toolbox (Gallicchio et al., 2017)	DeepESN (2019)	–	✓	–	2019	–
EchoTorch (Schaetti, 2018)	EchoTorch (2021)	–	✓	–	2021	–
PyTorch-ESN (Gallicchio et al., 2017)	PyTorch-ESN (2021)	–	✓	–	2021	–
ReservoirPy (Trouvain et al., 2020)	ReservoirPy (2022)	–	✓	–	2022	✓
HP-ELM (Akusok et al., 2015)	HP-ELM (2018)	✓	–	–	2018	✓
Python-ELM	Python-ELM (2019)	✓	–	–	2019	✓
LSM	LSM (2020)	–	–	✓	2020	–
OGER (Verstraeten et al., 2012)	Oger (2013)	✓	✓	✓	2013	–
PyRCN (This work)	PyRCN (2022)	✓	✓	UD	2022	✓

One important obstacle for RCNs on their way towards the mainstream could be the lack of a framework that allows a fair and straightforward comparison of RCNs with other machine learning architectures. RCNs are neither part of any widely used Machine/Deep Learning toolbox for Python, such as scikit-learn (Pedregosa et al., 2011), PyTorch (Paszke et al., 2019) or Tensorflow (Abadi et al., 2016), nor of Matlab. While specialized toolboxes have been developed in the past to optimize, train and evaluate RCNs, a side-by-side comparison of RCN models and traditional machine learning or deep-learning models was laborious. Interested users needed to change significant parts of their code or data structure, because many toolboxes neither provide a high-level interface nor utilize a common data structure.

Table 1 summarizes various RCN toolboxes and the proposed PyRCN along with important aspects, such as download links, which kind of RCNs are implemented, the year of the last major update and whether the toolboxes provide a high-level interface. In general, most of the toolboxes are developed for only one specific type of RCN. In fact, OGER (Verstraeten et al., 2012) was the only toolbox that implemented all important variants of RCNs. However, it is now out-of-date, especially because it was implemented in Python 2, which has reached its end of life in 2020. Only a few ESN toolboxes provide a high-level interface, which helps users to quickly try out ESNs. Unfortunately, even if an ESN toolbox provides a high-level interface, it is often not compatible with the commonly used Machine/Deep Learning frameworks, such as scikit-learn, PyTorch or Tensorflow. The ELM toolboxes both provide a high-level interface that is compatible with scikit-learn. However, only the HP-ELM (Akusok et al., 2015) is scalable towards large datasets and high-performance clusters, for which it provides specialized implementations. The Python-ELM (Python-ELM, 2019) can only be used for classification tasks. Compared to the other presented toolboxes in Table 1, PyRCN is the only actively maintained toolbox that efficiently implements both ESNs and ELMs. While ReservoirPy by Trouvain et al. (2020) provides a high-level interface to its ESN implementation, it does not adhere to any established specification. Therefore, PyRCN is the only toolbox that provides an RCN implementation with a high-level interface that complies with the scikit-learn specification. Herbert Jaeger,[2] Claudio Gallicchio[3] and the IEEE Task Force on Reservoir Computing[4] provide more details about these and other ESN toolboxes. Guang Bin Huang[5] also collected a coherent summary of ELM implementations and there are many more individual projects accessible on GitHub.[6]

A likely reason that RCNs are not yet officially included in leading Deep Learning toolkits, such as PyTorch or TensorFlow, is that their underlying training paradigm is very different from the one of conventional neural networks. Instead of using some kind of optimizer that requires a large number of iterations and a lot of electrical power, RCNs have only very few hyperparameters and are trained very efficiently using linear regression.

With this paper, we introduce PyRCN (Python Reservoir Computing Network), a new toolbox for RCNs that is based on widely used scientific Python packages, such as numpy or scipy, and complies with the interface specification of the popular machine learning platform scikit-learn (Pedregosa et al., 2011). As one original contribution of this work, this allows the seamless integration of PyRCN and scikit-learn its extensive list of features, such as model selection tools and alternatives to linear regression for training the output weights of RCNs. The designed models can also be easily compared with scikit-learns' built-in estimators, such as MLPs or Support Vector Machines (SVMs).

The second original contribution of this work is the definition of the "Building blocks of Reservoir Computing" as outlined in Section 2. By factoring ELMs, ESNs and LSMs in these building blocks, almost any conceivable RCN architecture can be constructed. Since the internal structure of PyRCN is based on these building blocks, a large amount of RCN architectures can be constructed in only a few lines of code. This is a significant advantage of PyRCN over other active toolboxes for RCNs, which are specifically developed to provide only one specific type of RCN.

Another unique feature of PyRCN is the support for the hyperparameter search strategy introduced in Steiner et al. (2021), which requires far fewer model fits than the conventional approach.

The remainder of this paper is structured as follows: In Section 2, we briefly review the main concepts of RCNs and show how to decompose them into basic building blocks. Section 3 introduces the proposed PyRCN toolbox with all included modules and shows a simple example of how to get started with PyRCN. In Section 4, we show how to construct different RCNs from these components. Section 5 demonstrates how to construct a custom RCN for a handwritten digit recognition task using PyRCN. Section 6 compares PyRCN with the reference toolboxes PyESN and HP-ELM by re-implementing parts of the recent work by Trierweiler Ribeiro et al. (2021). Finally, we summarize our work and outline future work in Section 7.

2. Building blocks of Reservoir Computing

A conventional Reservoir Computing Network (RCN) consists of various weight matrices that describe how input data is transported in and processed by a hidden layer, which is typically a pool of non-linear neurons. In order to provide a framework that implements existing architectures but retains the desired customizability, we need to design components that can be mixed and matched to create many different kinds of architectures instead of only offering static closed implementations of existing topologies. We postulate that all RC architectures can essentially be built by three main components (See Fig. 1):

1. "Input-to-Node": A combination of input weights and a non-linear input activation function

---

[2] https://www.ai.rug.nl/minds/research/esnresearch/.
[3] https://sites.google.com/site/cgallicch/resources.
[4] https://sites.google.com/view/reservoir-computing-tf/resources.
[5] http://www.extreme-learning-machines.org/.
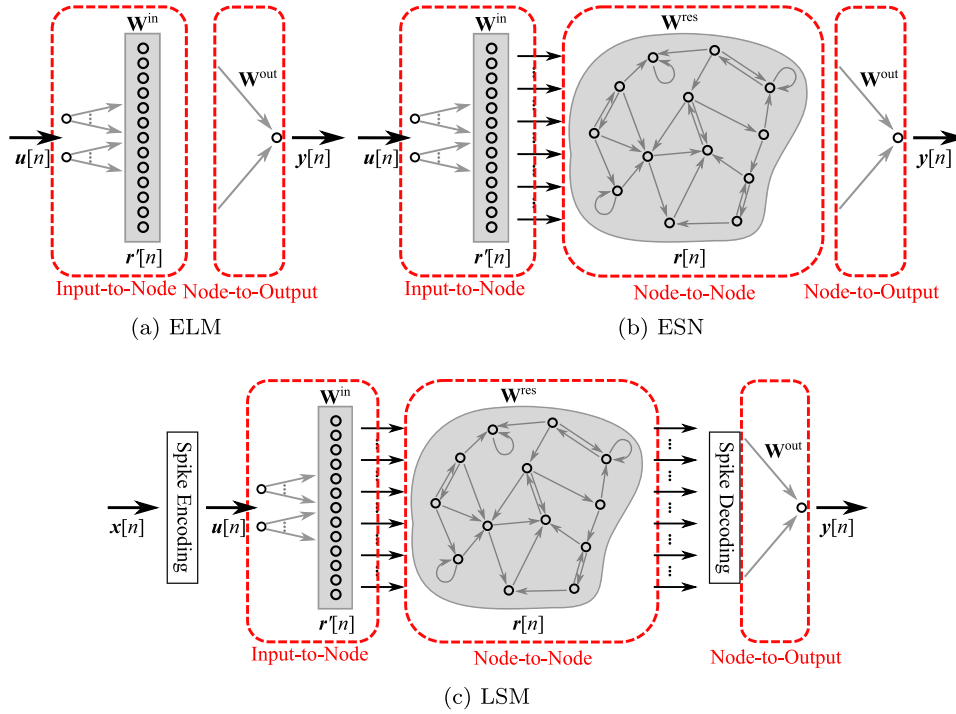[6] https://github.com/topics/reservoir-computing.

Fig. 1. Main outline of basic RCNs: In case of ELMs and ESNs, the input features are fed into a hidden layer, which is called the "reservoir" using the fixed input weight matrix $\mathbf{W}^{\text{in}}$. The reservoir consists of non-linear neurons. In case of the ESN, the neurons are inter-connected via the fixed reservoir matrix $\mathbf{W}^{\text{res}}$. For both, ESNs and ELMs, the output $\mathbf{y}[n]$ is a weighted combination of the reservoir states $\mathbf{r}[n]$ based on the output weight matrix $\mathbf{W}^{\text{out}}$. LSMs are closely related to ESNs. However, they need additional spike encoding and decoding utilities.

2. "Node-To-Node": The recurrent reservoir weights and a non-linear reservoir activation function
3. "Node-to-Output": The trainable output weights, often referred to as the linear model.

Since LSMs use spiking neural networks, additional building blocks for spike encoding and decoding are required (see Fig. 1(c))[7] and are currently under development. However, the remaining blocks are essentially the same as for the ESN.

Note that the black arrows in Figs. 1(b) and 1(c) between building blocks all indicate one-by-one connections. E.g., the first black arrow of the ESN between "Input-to-Node" and "Node-to-Node" denotes that the first neuron in "Input-to-Node" is also the first neuron of "Node-to-Node".

2.1. Input-to-Node

The "Input-to-Node" component describes the connections from the input features to the reservoir and the activation functions of the input neurons. Normally, the input weight matrix $\mathbf{W}^{\text{in}}$ has the dimension of $N^{\text{res}} \times N^{\text{in}}$, where N^{res} and N^{in} are the size of the reservoir and length of the input feature vector $\mathbf{u}[n]$ with the time index n, respectively. With

$$\mathbf{r}'[n] = f'(\mathbf{W}^{\text{in}}\mathbf{u}[n] + \mathbf{w}^{\text{bi}}) , \qquad (1)$$

we can describe the projection of the input features $\mathbf{u}[n]$ into the high-dimensional reservoir space $\mathbf{r}'[n]$ via the input activation function $f'(\cdot)$, which can be any non-linear function for the ELM (see Section 4.1) or the identity function for the standard ESN (see Section 4.2).

The values inside the input weight matrix are usually initialized randomly from a uniform distribution on the interval $[-1, 1]$ and are

afterwards scaled using the input scaling factor α_{u}. In case of a high-dimensional input feature space and/or large reservoir sizes N^{res}, this would lead to a huge input weight matrix and expensive computations to feed the feature vectors into the reservoir. As was shown in Jaeger (2001), Jalalvand et al. (2015), however, it is sufficient to have only a very small number of connections from the input nodes to the nodes inside the reservoir. Each node of the reservoir may therefore be connected to only K^{in} ($\ll N^{\text{in}}$) randomly selected input entries. This makes $\mathbf{W}^{\text{in}}$ typically very sparse and feeding the feature vectors into the reservoir potentially more efficient.

The bias weights $\mathbf{w}^{\text{bi}}$ with dimension N^{res} are typically initialized by fixed random values from a uniform distribution between ± 1 and multiplied by the hyper-parameter α_{bi}.

```
1  from pyrcn.base.blocks import InputToNode
2  from sklearn.datasets import make_blobs
3
4  # Generate a toy dataset
5  U, y = make_blobs(n_samples=100, n_features=10)
6  #        _ _ _ _ _ _ _ _
7  #       |                |
8  # ----| Input-to-Node |------
9  # u[n]|_ _ _ _ _ _ _ _ _|r'[n]
10 # U                      R_i2n
11
12 # Initialize, fit and apply an InputToNode
13 input_to_node = InputToNode(hidden_layer_size=50,
14                             k_in=5, input_activation="tanh",
15                             input_scaling=1.0, bias_scaling=0.1)
16
17 R_i2n = input_to_node.fit_transform(U)
18 print(U.shape, R_i2n.shape)
```

Listing 1: Minimal example for the "Input-to-Node" block.

Listing 1 gives an example of how to use "Input-to-Node". The toy dataset with $N^{\text{in}} = 10$ is transformed in a $N^{\text{res}} = 50$-dimensional space via sparse input weights. Due to the input scaling factor $\alpha_{\text{u}} = 1$, the non-zero weights are uniformly distributed on the interval $[-1, 1]$. The non-zero bias scaling factor α_{bi} leads to an additional constant bias input.

---

[7] For a simple notation, we assume that the sampling rate before and after spike en-/decoding is the same. In reality, the sampling rate after encoding is obviously much higher.

2.2. Node-to-Node

The "Node-to-Node" component describes the connections inside the reservoir. The output of "Input-to-Node" $\mathbf{r}'[n]$ together with the output of "Node-to-Node" from the previous time step $\mathbf{r}[n-1]$ are used to compute the new output of "Node-to-Node" $\mathbf{r}[n]$ using

$$\mathbf{r}[n] = (1 - \lambda)\mathbf{r}[n-1] + \lambda f(\mathbf{r}'[n] + \mathbf{W}^{\mathrm{res}}\mathbf{r}[n-1]) \,, \tag{2}$$

which is a leaky integration of the time-dependent reservoir states $\mathbf{r}[n]$. $f(\cdot)$ acts as the non-linear reservoir activation function of the neurons in "Node-to-Node". The leaky integration is equivalent to a first-order lowpass filter. Depending on the leakage $\lambda \in (0, 1]$, the reservoir states are globally smoothed.

The reservoir weight matrix $\mathbf{W}^{\mathrm{res}}$ is a square matrix of the size N^{res}. These weights are typically initialized from a standard normal distribution. The Echo State Property (ESP) requires that the states of all reservoir neurons need to decay in a finite time for a finite input pattern. In order to fulfil the ESP, the reservoir weight matrix is typically normalized by its largest absolute eigenvalue and rescaled to a spectral radius ρ, because it was shown in Jaeger (2001) that the ESP holds at least as long as $\rho \leq 1$ but can also hold longer for specific hyper-parameter combinations (Gallicchio, 2019). The spectral radius and the leakage together shape the temporal memory of the reservoir. Similar as for "Input-to-Node", the reservoir weight matrix gets huge in case of large reservoir sizes N^{res}, it can be sufficient to only connect each node in the reservoir only to K^{rec} ($\ll N^{\mathrm{res}}$) randomly selected other nodes in the reservoir, and to set the remaining weights to zero. Listing 2 gives an example of how to use "Node-to-Node". As before, the toy dataset with $N^{\mathrm{in}} = 10$ is transformed in a $N^{\mathrm{res}} = 50$-dimensional space via "Input-to-Node". Afterwards, the output is transformed in a dense "Node-to-Node" with recurrent connections (spectral radius $\rho = 1$, leakage $\lambda = 0.9$).

```
1  from pyrcn.base.blocks import NodeToNode
2  # Here goes the content of Listing 1
3  #      _ _ _ _ _ _ _ _        _
4  #     |               |      |
5  # ----| Input-to-Node |------| Node-to-Node |------
6  # u[n]|_ _ _ _ _ _ _ _|r'[n] |_ _ _ _ _ _ _ |r[n]
7  # U                 R_i2n                   R_n2n
8
9  # Initialize, fit and apply NodeToNode
10 node_to_node = NodeToNode(hidden_layer_size=50,
11                           reservoir_activation="tanh",
12                           spectral_radius=1.0, leakage=0.9,
13                           bidirectional=False)
14 R_n2n = node_to_node.fit_transform(R_i2n)
15 print(U.shape, R_n2n.shape)
```

Listing 2: Minimal example for the "Node-to-Node" block.

To incorporate information from the future inputs, bi-directional RCNs have been introduced, e.g. in Triefenbach and Martens (2011). The paradigm is realized when the output of "Node-to-Node" is computed in four steps:

1. $\mathbf{r}_{\mathrm{fw}}[n]$ is computed using $\mathbf{r}'[n]$ forward with Eq. (2).
2. $\mathbf{r}_{\mathrm{bw}}[n]$ is computed using $\mathbf{r}'[n]$ reversed in time with Eq. (2).
3. Reverse $\mathbf{r}_{\mathrm{bw}}[n]$ again in time to obtain the original temporal order.
4. Stack $\mathbf{r}_{\mathrm{fw}}[n]$ and $\mathbf{r}_{\mathrm{bw}}[n]$ to obtain the final reservoir state $\mathbf{r}[n]$ that includes future information.

Utilizing "Node-to-Node" in the bi-directional mode would double the size of the reservoir state vector $\mathbf{r}[n]$ to $2 \times N_{\mathrm{res}}$.

2.3. Node-to-Output

The "Node-to-Output" component is the mapping of the reservoir state $\mathbf{r}[n]$ to the output $\mathbf{y}[n]$ of the network. In conventional RCNs, this mapping is trained using (regularized) linear regression. To that end, all reservoir states $\mathbf{r}[n]$ are concatenated into the reservoir state collection matrix $\mathbf{R}$. As linear regression usually contains an intercept term, every reservoir state $\mathbf{r}[n]$ is expanded by a constant of 1. All desired outputs

$\mathbf{d}[n]$ are collected into the desired output collection matrix $\mathbf{D}$. Then, the mapping matrix $\mathbf{W}^{\mathrm{out}}$ can be computed using Eq. (3), where ϵ is a regularization parameter.

$$\mathbf{W}^{\mathrm{out}} = \left(\mathbf{R}\mathbf{R}^{\mathrm{T}} + \epsilon \mathbf{I} \right)^{-1} \left(\mathbf{D}\mathbf{R}^{\mathrm{T}} \right) \tag{3}$$

The size of the output weight matrix $N^{\mathrm{out}} \times (N^{\mathrm{res}} + 1)$ or $N^{\mathrm{out}} \times (2 \times N^{\mathrm{res}} + 1)$ in case of a bidirectional "Node-to-Node" determines the total number of free parameters to be trained in the neural network. After training, the output $\mathbf{y}[n]$ can be computed using Eq. (4).

$$\mathbf{y}[n] = \mathbf{W}^{\mathrm{out}}\mathbf{r}[n] \tag{4}$$

Note that, in general, other training methodologies could be used to compute output weights. For example, in Triefenbach and Martens (2011) it has been shown that a non-linear "Node-to-Output" can improve the performance of an RCN when limiting N^{res}. Since solving Eq. (3) is computationally expensive for large N^{res}, PyRCN provides an incremental regression as proposed in Liang et al. (2006).

```
1  from sklearn.linear_model import Ridge
2  # Here goes the content of Listing 2
3  #      _ _ _ _ _ _ _        _ _ _ _ _ _ _        _ _ _ _ _ _ _
4  #     |             |      |             |      |             |
5  # ----|Input-to-Node |-----|Node-to-Node |-----|Node-to-Output |
6  # u[n]|_ _ _ _ _ _ _|r'[n]|_ _ _ _ _ _ _|r[n] |_ _ _ _ _ _ _ |
7  # U               R_i2n               R_n2n              |
8  #                                                         |
9  #                                               y[n] | y_pred
10
11 # Initialize, fit and apply NodeToOutput
12 y_pred = Ridge().fit(R_n2n, y).predict(R_n2n)
13 print(y_pred.shape)
```

Listing 3: Minimal example for the "Node-to-Output" block.

Listing 3 gives an example of how to use "Node-to-Output". As before, the toy dataset with $N^{\mathrm{in}} = 10$ is transformed in a $N^{\mathrm{res}} = 50$-dimensional space via "Input-to-Node" and "Node-to-Node". Finally, Eq. (3) is solved using `sklearn.linear_model.Ridge` with default settings.

3. PyRCN

With PyRCN, we introduce a unified toolbox for implementing and developing different kinds of RCNs that is transparent and easy to use. PyRCN is built on scikit-learn and is fully compatible with its interface specification, allowing full interoperability between the toolkits. PyRCN is structured as shown in Figure Fig. 2.

3.1. Overview of PyRCN

The module `pyrcn.base.blocks` provides classes that implement the two building blocks "Input-to-Node" (`InputToNode`) and "Node-to-Node" (`NodeToNode`) that are introduced in Section 2. The module `pyrcn.base` provides base classes to facilitate the implementation of user-defined building blocks. In addition, we provide trainable variants of both blocks, e.g. using BIP (`BatchIntrinsicPlasticity`) or Hebbian Learning (`HebbianNodeToNode`) (Munakata and Pfaffly, 2004), and fully customizable variants (`PredefinedWeights...`), where weight matrices can be set from outside. This offers the possibility to quickly define a standard RCN or to create very custom RCNs with user-defined weight matrices such as in Rodan and Tino (2011).

As we explained in Section 2, the building block "Node-to-Output" is typically a linear regression model, although any regressor provided by scikit-learn can be used for this task. Since these cannot be trained incrementally, training on large datasets or training large networks becomes memory-expensive. To reduce the required memory, we provide a linear regression module based on Liang et al. (2006) in `pyrcn.linear_model.IncrementalRegression`. Since the correlation matrices are updated incrementally, our linear regression model is capable of online learning, i.e., with the presentation of each new training sample.
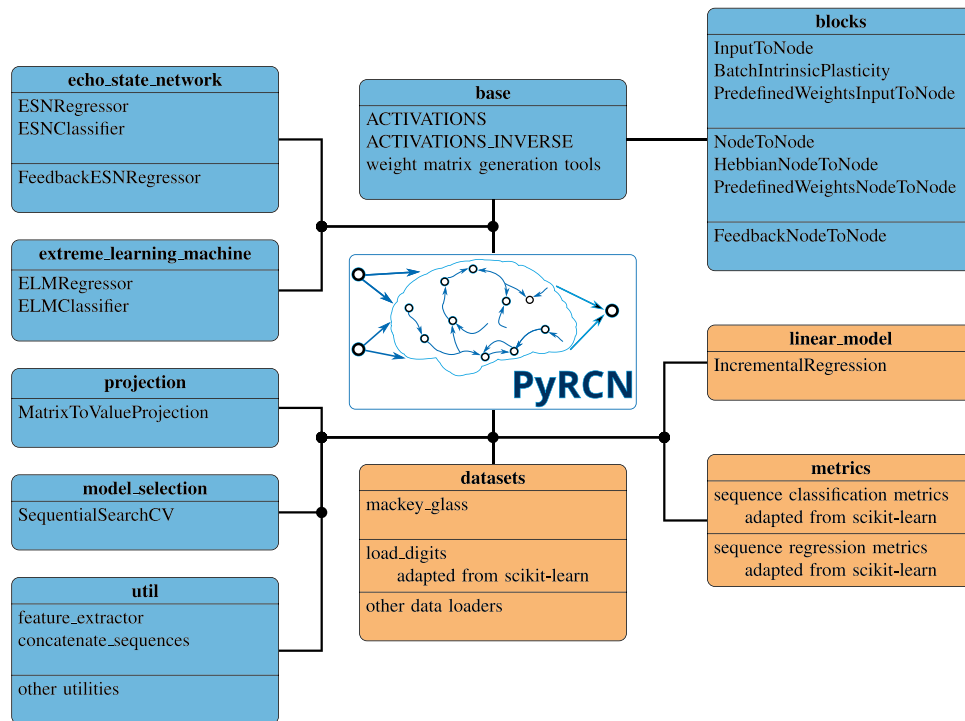
Fig. 2. Overview of the most important modules included in PyRCN. The blue modules include modules that are entirely new, whereas the orange modules can be treated as an extension of scikit-learn's built-in modules.

In `pyrcn.extreme_learning_machine` and `pyrcn.echo_state_network`, we provide classes to build ELMs and ESNs for classification and regression following the scikit-learn API. In the various listings of Sections 2 and 4, we show that these classes can be used to build a large variety of models.

To deal with sequence-to-value problems, we implemented the module `pyrcn.projection` that allows the projection of a matrix to values in different ways. This allows PyRCN to deal with sequential data while still staying as close as possible to the scikit-learn API (which does not natively support sequences).

In `pyrcn.model_selection`, we developed the estimator `SequentialSearchCV` as an extension for the existing model selection tools provided by scikit-learn.

In `pyrcn.metrics`, we have re-implemented most of the metrics from scikit-learn to handle sequence-to-sequence and sequence-to-label tasks. Using these metrics, all model selection techniques provided by scikit-learn can be applied to sequence learning tasks (not limited to using RCNs).

Thanks to the scikit-learn interface, all modules can easily be customized and integrated. Furthermore, we steadily extend the library with new modules.

3.2. Getting started with PyRCN

PyRCN can be installed with the command `pip install pyrcn` or by cloning it from our Github repository[8] and installing it with `python setup .py install`. For more information, we refer to our website.[9]

In Listing 4, we show how to set up and train vanilla RCNs for predicting the Mackey-Glass time series (Glass and Mackey, 2010) with the same settings as used in Jaeger (2001) to introduce ESNs. The `ESNRegressor` object connects the building blocks "Input-to-Node", "Node-to-Node" and "Node-to-Output", i.e., `pyrcn.linear_model. Incremental`

`Regression`, and the `ELMRegressor` object connects the building blocks "Input-to-Node" and "Node-to-Output". This minimum working example demonstrates two important aspects of PyRCN:

1. Simplicity: it needs only four lines of code to load the Mackey-Glass dataset that is part of PyRCN and only two lines to fit the different RCN models, respectively.
2. Interoperability with scikit-learn: Instead of the default incremental regression, we have customized the `ELMRegressor` by using `Ridge` from scikit-learn.

```
1  from sklearn.linear_model import Ridge as skRidge
2  from sklearn.metrics import mean_squared_error
3  from pyrcn.echo_state_network import ESNRegressor
4  from pyrcn.extreme_learning_machine import ELMRegressor
5  from pyrcn.datasets import mackey_glass
6
7  # Load the dataset
8  X, y = mackey_glass(n_timesteps=5000)
9  # Define Train/Test lengths
10 trainLen = 1900
11 X_train, y_train = X[:trainLen], y[:trainLen]
12 X_test, y_test = X[trainLen:], y[trainLen:]
13
14 # Initialize and train an ELMRegressor and an ESNRegressor
15 esn = ESNRegressor().fit(X=X_train.reshape(-1, 1), y=y_train)
16 y_pred = esn.predict(X_test.reshape(-1, 1))
17 print(mean_squared_error(y_test, y_pred))
18 elm = ELMRegressor(regressor=skRidge()).fit(X=X_train.reshape(-1,
       1), y=y_train)
19 y_pred = elm.predict(X_test.reshape(-1, 1))
20 print(mean_squared_error(y_test, y_pred))
```

Listing 4: Minimal example for setting up an ESN and an ELM to predict the Mackey-Glass time-series (Glass and Mackey, 2010)

4. Build Reservoir Computing Networks with PyRCN

By combining the building blocks introduced above, a vast number of different RCNs can be constructed. In this section, we build two important variants of RCNs, namely ELMs and ESNs.

4.1. Extreme Learning Machines

The vanilla ELM as a single-layer feedforward network consists of an "Input-to-Node" and a "Node-to-Output" module and is trained in two steps:

1. Compute the high-dimensional reservoir states $\mathbf{R}'$, which is the collection of reservoir states $\mathbf{r}'[n]$ from Eq. (1).
2. Compute the output weights $\mathbf{W}^{\text{out}}$ using Eq. (3) with $\mathbf{R}'$.

Listing 5 shows how to construct such a standard ELM with PyRCN. The `ELMRegressor` internally passes the input features through "Input-to-Node" and trains "Node-to-Output" by solving Eq. (3) via `pyrcn.linear_model.IncrementalRegression`.

```
1  # Here goes the content of Listing 3
2  from pyrcn.extreme_learning_machine import ELMRegressor
3
4  # Vanilla ELM for regression tasks with input_scaling
5  #         _ _ _ _ _ _ _             _ _ _ _ _ _ _
6  #        |             |           |             |
7  # ----|Input-to-Node |-----|Node-to-Output |------
8  # u[n]| _ _ _ _ _ _ _ |r'[n]| _ _ _ _ _ _ |y[n]
9  #                                              y_pred
10 #
11 vanilla_elm = ELMRegressor(input_scaling=0.9)
12 vanilla_elm.fit(U, y)
13 print(vanilla_elm.predict(U))
```

Listing 5: Example of how to construct a vanilla ELM with PyRCN.

When a custom "Input-to-Node" or "Node-to-Output" needs to be utilized, it is necessary to explicitly pass these to the ELM via `input_to_node` or `regressor`, respectively. This is, for example, necessary if the input weights should be pre-trained e.g. by Batch Intrinsic Plasticity (BIP) (Neumann and Steil, 2011) or other unsupervised pre-training methods, such as in Steiner et al. (2022). An example for an ELM with a BIP "Input-to-Node" and a specific "Node-to-Output" is given in Listing 6. Here, Eq. (3) is solved using `sklearn.linear_model.Ridge` instead of the built-in regression tool of PyRCN.

```
1  # Here goes the content of Listing 3
2  from pyrcn.base.blocks import BatchIntrinsicPlasticity
3  from pyrcn.extreme_learning_machine import ELMRegressor
4
5  # Custom ELM with BatchIntrinsicPlasticity
6  #         _ _ _ _ _ _ _             _ _ _ _ _ _ _
7  #        |             |           |             |
8  # ----|      BIP       |-----|Node-to-Output |------
9  # u[n]| _ _ _ _ _ _ _ |r'[n]| _ _ _ _ _ _ |y[n]
10 #                                              y_pred
11 #
12 bip_elm = ELMRegressor(input_to_node=BatchIntrinsicPlasticity(),
13                         regressor=Ridge(alpha=1e-5))
14
15 bip_elm.fit(U, y)
16 print(bip_elm.predict(U))
```

Listing 6: Example of how to construct an ELM with a BIP "Input-to-Node" ELMs with PyRCN.

Hierarchical or Ensemble ELMs can then be built using multiple "Input-to-Node" modules in parallel or in a cascade. As can be seen in Listing 7, this is possible when using scikit-learn's `sklearn.pipeline.Pipeline` (cascading) or `sklearn.pipeline.FeatureUnion` (ensemble).

```
1  # Here goes the content of Listing 3
2  from sklearn.pipeline import Pipeline, FeatureUnion
3  from pyrcn.extreme_learning_machine import ELMRegressor
4
5  # ELM with cascaded InputToNode and default regressor
6  #     _ _ _ _ _ _       _ _ _ _ _ _       _ _ _ _ _ _
7  #    |    (bip)    |   |    (base)   |   |             |
8  # ----|Input-to-Node1|-----|Input-to-Node2|-----|Node-to-Output |
9  # u[n]| _ _ _ _ _ _ |       |            |r'[n]| _ _ _ _ _ _ |
10 #                                                       |
11 #                                                       |
12 #                                              y[n] | y_pred
13 #
14 i2n = Pipeline([('bip', BatchIntrinsicPlasticity()),
15                 ('base', InputToNode(bias_scaling=0.1))])
16 casc_elm = ELMRegressor(input_to_node=i2n).fit(U, y)
17
18 # Ensemble of InputToNode with activations
19 #               _ _ _ _ _ _ _
20 #              |     (i)     |
21 #       |----|Input-to-Node1|-----|
22 #       |    | _ _ _ _ _ _ |     |
23 #       |                         _____    _ _ _ _ _ _
24 # -----o                         r'[n]|Node-to-Output |------
25 # u[n] |         _ _ _ _ _ _     |-----| _ _ _ _ _ _ _ |y[n]
26 #       |    |     (th)     |     |                          y_pred
27 #       |----|Input-to-Node2|-----|
28 #            | _ _ _ _ _ _ _ |
29 #
30 i2n = FeatureUnion([('i',InputToNode(input_activation="identity")
                      ),
31                     ('th',InputToNode(input_activation="tanh"))])
32 ens_elm = ELMRegressor(input_to_node=i2n)
33 ens_elm.fit(U, y)
34 print(casc_elm, ens_elm)
```

Listing 7: Examples for implementing cascaded or Ensemble-ELMs with PyRCN.

4.2. Echo State Networks

The Context Reverberation Network (CRN) (Kirby, 1991) is a very early variant of Reservoir Computing that was later generalized to the ESN in Jaeger (2001). CRNs and ESNs, as variants of RNNs, consequently consist of an "Input-to-Node", a "Node-to-Node" and a "Node-to-Output" module and are trained in three steps.

1. Compute the neuron input states $\mathbf{R}'$, which is the collection of reservoir states $\mathbf{r}'[n]$ from Eq. (1). Note that here the activation function $f'(\cdot)$ is typically linear.
2. Compute the reservoir states $\mathbf{R}$, which is the collection of reservoir states $\mathbf{r}[n]$ from Eq. (2). Note that here the activation function $f(\cdot)$ is typically non-linear.
3. Compute the output weights $\mathbf{W}^{\text{out}}$ using

   (a) Eq. (3) with $\mathbf{R}$ when considering an ESN.
   (b) Gradient descent or other optimization algorithms when considering a CRN or when using an ESN with non-linear outputs, e.g. in Triefenbach and Martens (2011).

An example of how to construct such a vanilla ESN with PyRCN is given in Listing 8, where the `ESNRegressor` internally passes the input features through "Input-to-Node" and "Node-to-Node", and trains "Node-to-Output" by solving Eq. (3) via `pyrcn.linear_model.IncrementalRegression`.

```
1  # Here goes the content of Listing 3
2  from pyrcn.echo_state_network import ESNRegressor
3
4  # Vanilla ESN for regression tasks with spectral_radius and
      leakage
5  #     _ _ _ _ _ _       _ _ _ _ _ _       _ _ _ _ _ _
6  #    |             |   |             |   |             |
7  # ----|Input-to-Node |-----|Node-to-Node |-----|Node-to-Output |
8  # u[n]| _ _ _ _ _ _ |r'[n]| _ _ _ _ _ _ |r[n] | _ _ _ _ _ _ |
9  #                                                       |
10 #                                                       |
11 #                                              y[n] | y_pred
12 #
13 vanilla_esn = ESNRegressor(spectral_radius=1, leakage=0.9)
14 vanilla_esn.fit(U, y)
15 print(vanilla_esn.predict(U))
```

Listing 8: Example of how to construct a vanilla ESN with PyRCN.

As for ELMs in Listing 6, various unsupervised learning techniques can be used to pre-train ESNs (Basterrech et al., 2011; Schrauwen et al., 2008; Lazar et al., 2009). For a summary, we refer to Steiner et al. (2022). An example is given in Listing 9, in which we use a BIP "Input-to-Node" and a Hebbian "Node-to-Node".

```
1  # Here goes the content of Listing 3
2  from pyrcn.base.blocks import BatchIntrinsicPlasticity,
       HebbianNodeToNode
3  from pyrcn.echo_state_network import ESNRegressor
4
5  # Custom ESN with BatchIntrinsicPlasticity and HebbianNodeToNode
6  #     _ _ _ _ _ _       _ _ _ _ _ _       _ _ _ _ _ _
7  #    |    (bip)    |   |    (hebb)   |   |             |
8  # ----|Input-to-Node |-----|Node-to-Node |-----|Node-to-Output |
9  # u[n]| _ _ _ _ _ _ |r'[n]| _ _ _ _ _ _ |r[n] | _ _ _ _ _ _ |
10 #                                                       |
11 #                                                       |
12 #                                              y[n] | y_pred
13 #
14 bip_esn = ESNRegressor(input_to_node=BatchIntrinsicPlasticity(),
15                         node_to_node=HebbianNodeToNode(),
16                         regressor=Ridge(alpha=1e-5))
17
18 bip_esn.fit(U, y)
19 print(bip_esn.predict(U))
```

Listing 9: Example of how to construct an ESN with a BIP "Input-to-Node" and a Hebbian "Node-to-Node" with PyRCN.

4.3. Deep Echo State Networks

The term "Deep ESN" can refer to different approaches of hierarchical ESN architectures:

- The Deep ESN as described in Gallicchio et al. (2017) is a sequence of "Input-to-Node" and "Node-to-Node" combinations: the states of the $m - 1$-th reservoir are the input for the mth reservoir. The states of all reservoirs are stacked and based on the stacked states, the output weights are computed.
- The stacked ESN as used in Steiner et al. (2020) is a sequence of "Input-to-Node", "Node-to-Node" and "Output-to-Node" combinations: the outputs of one reservoir serve as the inputs for the next reservoir.
- The Modular Deep ESN as described in Carmichael et al. (2018) are multiple parallel "Input-to-Node" and "Node-to-Node" combinations: the output weights are computed over all parallel reservoir states.

With the proposed building blocks, we can build different variants of deep ESNs and many more customized variants. One rather complex example is given in Listing 10, where we have a layer that mimicks the Modular Deep ESN in the first layer. On top of that, we have stacked a second reservoir that receives the output of the first layer as input.

```
1  # Here goes the content of Listing 3
2  from pyrcn.base.blocks import InputToNode, NodeToNode
3  from pyrcn.echo_state_network import ESNRegressor
4
5  # Multilayer ESN
6  #                      u[n]
7  #                       |
8  #                       |
9  #            _____o_____
10 #           |                     |
11 #  _ _ _ _ _|_ _ _ _         _ _ _|_ _ _ _
12 # |     (i)       |        |     (i)       |
13 # |Input-to-Node1|         |Input-to-Node2|
14 # |_ _ _ _ _ _ _ _|        |_ _ _ _ _ _ _ _|
15 #          |r1'[n]                  |r2'[n]
16 #  _ _ _ _ _|_ _ _ _         _ _ _|_ _ _ _
17 # |     (th)      |        |     (th)      |
18 # | Node-to-Node1 |        | Node-to-Node2 |
19 # |_ _ _ _ _ _ _ _|        |_ _ _ _ _ _ _ _|
20 #         |r1[n]                   |r2[n]
21 #         |_____           _____|
22 #                 |        |
23 #                 |        |
24 #           _ _ _|_ _ _ _|_ _
25 #          |   Node-to-Node3   |
26 #          |_ _ _ _ _ _ _ _ _ _|
27 #                    |
28 #                  r3[n]|
29 #           _ _ _ _ _|_ _ _ _ _
30 #          |                   |
31 #          |Node-to-Output     |
32 #          |_ _ _ _ _ _ _ _ _ _|
33 #                    |
34 #                  y[n]|
35
36 l1 = Pipeline([('i2n1', InputToNode(hidden_layer_size=100)),
37               ('n2n1', NodeToNode(hidden_layer_size=100))])
38
39 l2 = Pipeline([('i2n2', InputToNode(hidden_layer_size=400)),
40               ('n2n2', NodeToNode(hidden_layer_size=400))])
41
42 i2n = FeatureUnion([('l1', l1),
43                     ('l2', l2)])
44 n2n = NodeToNode(hidden_layer_size=500)
45 layered_esn = ESNRegressor(input_to_node=i2n,
46                            node_to_node=n2n)
47
48 layered_esn.fit(U, y)
49 print(layered_esn.predict(U))
```

Listing 10: Example of how to construct a rather complex ESN consisting of two layers. It is built out of two small parallel reservoirs in the first layer and a large reservoir in the second layer.

4.4. Liquid State Machines

Basic Liquid State Machines (LSMs) can be built in a similar way as ESNs. The main difference is that $\mathbf{u}[n]$ is a spike train sequence, hence, we need spiking neuronal models, and additional tools for spike coding and decoding as proposed in Schrauwen and Campenhout (2003), which are currently under development.

4.5. Complex example: Optimize the hyper-parameters of RCNs

In Lukoševičius (2012), Trouvain et al. (2020) it is discussed that the different hyper-parameters of RCNs are often tuned jointly using grid, random search or Bayesian Optimization (Mockus and Mockus, 1991). However, this requires a lot of iterations, e.g., 1000 for the ESN model in Trouvain et al. (2020). Other strategies, such as Jalalvand et al. (2015), Steiner et al. (2021, 2020) proposed a half-guided sequential optimization procedure that can significantly reduce the number of iterations.

While scikit-learn already provides utilities for performing grid and random searches, some effort is required to perform a sequential optimization. As can be seen in Listing 11, we simplify this in PyRCN by introducing `pyrcn.model_selection.SequentialSearchCV`, which is derived from the base search utilities of scikit-learn and is inspired by the `sklearn.pipeline.Pipeline`. Essentially, a model with initial parameters is defined, followed by defining the different search steps. Similar to a `sklearn.pipeline.Pipeline`, `SequentialSearchCV` receives a list of searches (optimization steps), which are then sequentially performed.

Since internally, nothing else but the standard model selection tools provided by scikit-learn are evaluated, a particular advantage of `SequentialSearchCV` is that it is not restricted to sequentially optimize PyRCN models but it can also be used to optimize any model that provides a scikit-learn interface. This shows how closely PyRCN and scikit-learn interact, and how it is possible to extend existing scikit-learn model selection tools. Note that it is furthermore possible to flexibly define an new optimization strategy, e.g., by changing the order of the optimization steps.

```
1  from sklearn.metrics import make_scorer
2  from sklearn.metrics import mean_squared_error
3  from sklearn.model_selection import TimeSeriesSplit
4  from sklearn.model_selection import RandomizedSearchCV, \
                    GridSearchCV
5  from scipy.stats import uniform
6  from pyrcn.echo_state_network import ESNRegressor
7  from pyrcn.model_selection import SequentialSearchCV
8  from pyrcn.datasets import mackey_glass
9
10
11 # Load the dataset
12 X, y = mackey_glass(n_timesteps=5000)
13 X_train, X_test = X[:1900], X[1900:]
14 y_train, y_test = y[:1900], y[1900:]
15
16 # Define initial ESN model
17 esn = ESNRegressor(bias_scaling=0, spectral_radius=0, leakage=1,
18                    requires_sequence=False)
19
20 # Define optimization workflow
21 scorer = make_scorer(mean_squared_error, greater_is_better=False)
22 step_1_params = {'input_scaling': uniform(loc=1e-2, scale=1),
23                  'spectral_radius': uniform(loc=0, scale=2)}
24 kwargs_1 = {'n_iter': 200, 'n_jobs': -1, 'scoring': scorer,
25             'cv': TimeSeriesSplit()}
26 step_2_params = {'leakage': [0.2, 0.4, 0.7, 0.9, 1.0]}
27 kwargs_2 = {'verbose': 5, 'scoring': scorer, 'n_jobs': -1,
28             'cv': TimeSeriesSplit()}
29
30 searches = [('step1', RandomizedSearchCV, step_1_params, kwargs_1
                ),
31             ('step2', GridSearchCV, step_2_params, kwargs_2)]
32
33 # Perform the search
34 esn_opti = SequentialSearchCV(esn, searches).fit(X_train.reshape
             (-1, 1), y_train)
35 print(esn_opti)
```

Listing 11: Example for a sequential parameter optimization with PyRCN. Therefore, a model with initial parameters and various search steps are defined. Internally, `SequentialSearchCV` will perform the list of optimization steps sequentially.

5. Sequence processing with PyRCN

ESNs are a variant of RNNs and, due to their recurrent connections, able to implicitly model temporal dependencies in the feature vector sequence, as it was discussed in Section 4.2. By default, scikit-learn does not support sequence handling. However, our incremental regression allows to deal with sequences of arbitrary lengths. With the following example, we show how to modify the programming pattern when dealing with sequences instead of instances.

```
1  from sklearn.base import clone
2  from sklearn.model_selection import train_test_split
3  from sklearn.model_selection import ParameterGrid
4
5  from pyrcn.echo_state_network import ESNClassifier
6  from pyrcn.metrics import accuracy_score
7  from pyrcn.datasets import load_digits
8
9  # Load the dataset
10 X, y = load_digits(return_X_y=True, as_sequence=True)
11 print("Number of digits: {0}".format(len(X)))
12 print("Shape of digits {0}".format(X[0].shape))
13 # Divide the dataset into training and test subsets
14 X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.2,
15                                           random_state=42)
16 print("Number of digits in training set: {0}".format(len(X_tr)))
17 print("Shape of the first digit: {0}".format(X_tr[0].shape))
18 print("Number of digits in test set: {0}".format(len(X_te)))
19 print("Shape of the first digit: {0}".format(X_te[0].shape))
20
21 # These parameters were optimized using SequentialSearchCV
22 esn_params = {'input_scaling': 0.1,
23               'spectral_radius': 1.2,
24               'input_activation': 'identity',
25               'k_in': 5,
26               'bias_scaling': 0.5,
27               'reservoir_activation': 'tanh',
28               'leakage': 0.1,
29               'k_rec': 10,
30               'alpha': 1e-5,
31               'decision_strategy': "winner_takes_all"}
32
33 b_esn = ESNClassifier(**esn_params)
34
35 param_grid = {'hidden_layer_size': [50, 100, 200, 400, 500],
36               'bidirectional': [False, True]}
37
38 for params in ParameterGrid(param_grid):
39     esn = clone(b_esn).set_params(**params).fit(X_tr, y_tr)
40     acc_score = accuracy_score(y_te, esn.predict(X_te))
```

Listing 12: Sequence processing with PyRCN.

5.1. Handwritten digits dataset

For this task, we utilized the test set from the small handwritten digit dataset[10] that is hosted by the UCI Machine Learning Repository (Dua and Graff, 2017) and included in scikit-learn. It consists of 1797 8x8 pixel gray-scale images of handwritten digits from 0 to 9. We normalized all pixels to fall in a range from −1 to 1. For the following experiment, we treat each image of the dataset as an independent sequence that can, in general, have an arbitrary length. Therefore, we scan the image from left to right as in Jalalvand et al. (2015, 2018) and thus present one column of eight pixels in one time step to the model.

Programming pattern for sequence processing

Listing 12 shows the script for this experiment. This complex use-case requires a serious hyper-parameter tuning. To keep the code example simple, we did not include the optimization in this paper and refer the interested readers to the Jupyter Notebook[11] that was developed to produce these results.

With `pyrcn.datasets.load_digits`, we provide a wrapper function around the scikit-learn equivalent to load the digits dataset. Specifically, we added the option `as_sequence`. If this option is set to `True`, the dataset is returned as a numpy array of objects with a length of 1797 (number of images in the dataset). Each object itself is an 8×8 numpy array representing the image as a sequence. This can be validated with the `print` statements from line 11 to 20 in Listing 12.

Two particular advantages of this data structure are:

1. We can use model selection utilities from scikit-learn to split our dataset in training and test sequences;
2. Each sequence can have an arbitrary number of samples as long as the number of features is always the same.

Table 2
Partitioning the dataset in training, validation and test time series.

Fold	Training	Validation	Test
1	CAT	EBAY	MSFT
2	EBAY	MSFT	CAT
3	MSFT	CAT	EBAY

A disadvantage of this data structure is the break with scikit-learn's paradigm. However, since the number of features in every sequence is the same, we can simply reproduce e.g. the scikit-learn-compatible feature matrix via `numpy.concatenate(X_train)`. This is important if one would compare ESNs with models provided by scikit-learn.

To ensure that the objects of PyRCN do not break with the programming pattern of scikit-learn when dealing with multiple sequences, we internally convert the new data structure in a scikit-learn compatible format and `pyrcn.echo_state_network.ESNClassifier` can take the aforementioned data structure as well as the scikit-learn compatible data structure. Since scikit-learn does not provide any metrics for evaluating sequence processing tasks, we provide metrics, such as the accuracy score, for these tasks. These metrics can easily be combined with default model selection tools provided by scikit-learn.

5.2. Final results

With the last few lines of code in Listing 12, we evaluate models with different reservoir sizes from 50 to 500 neurons and compare uni- and bidirectional ESNs. As demonstrated in Steiner et al. (2021, 2020), this is an important aspect of many experiments with RCNs, because these two steps significantly influence the final performance of RCN models. This is demonstrated in Fig. 3, where the mean validation and the test accuracy strongly increases up to a reservoir size of 200 neurons and then slowly reaches saturation areas.

6. Comparison of different toolboxes

As it was stated in the introduction, RCNs have a broad variety of applications. As an example, we reproduce a part of the results by Trierweiler Ribeiro et al. (2021), where the stock price return volatility was predicted using Echo State Networks. We compare the performance of different RCN architectures included in PyRCN with the toolboxes PyESN[12] and HP-ELM by Akusok et al. (2015) using the scikit-ELM[13] (skELM) as the interface. All required data, pre-trained models and code to repeat the experiments can be found in our repository.[14]

6.1. Dataset

We used the datasets provided by Trierweiler Ribeiro et al. (2021), namely the stock price volatilities for the three Nasdaq companies Caterpillar (CAT), eBay (EBAY) and Microsoft (MSFT), with each datasets having a length of 2745 days. The input data was the original time series and the output data the time series shifted by 1, 5 and 22 days, respectively.

The time series were split in the same 3-fold cross validation scheme as in Trierweiler Ribeiro et al. (2021) into training, validation and test sets according to Table 2. For the hyper-parameter optimization, only the training and validation sets of each fold were used. The test set of each fold was solely used to report the final results.
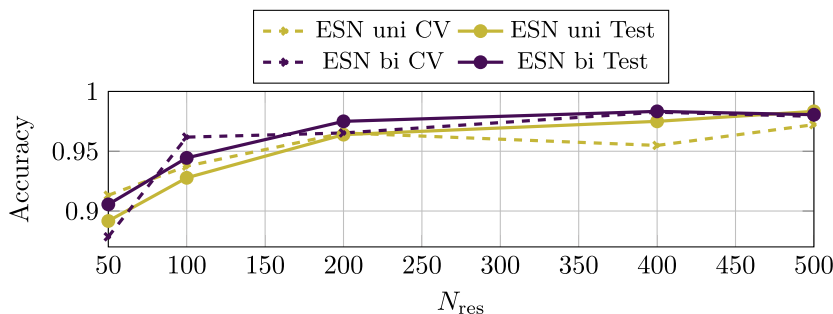

Fig. 3. Validation and test accuracy for the ESN model in uni- and bidirectional mode. The performance strongly increases until a reservoir size of 200 and remains almost constant afterwards.

6.2. Input and target preparation

As in Trierweiler Ribeiro et al. (2021), we compared two different input data: The original time series and the Heterogeneous Autoregressive (HAR) feature set (Corsi, 2009), which expands the one-dimensional time series to three dimensions by adding moving-average filtered representations with filter lengths of five and 22 days, respectively. Since the description of the HAR feature set by Corsi (2009, Eqs. (4)–(7)) and Trierweiler Ribeiro et al. (2021, Eq. (4)) have different temporal contexts for the moving average filter, we compared both implementations. We use the description by Corsi (2009) in the following, because the results were comparable to Trierweiler Ribeiro et al. (2021), even though not equal.

Each input and target time series was separately scaled to the interval $[0, 1]$ before using it to train and test models. The constants therefore were computed on the training dataset of each fold.

6.3. Hyperparameter optimization

In contrast to Trierweiler Ribeiro et al. (2021), we did not jointly optimize all hyper-parameters but used the sequential optimization from Section 4.5. When considering ELMs, we aimed to optimize input scaling α_u, then bias scaling α_{bi}, and, finally, jointly the regularization parameter β and the number of neurons N^{res}. When considering ESNs, we aimed to jointly optimize input scaling α_u and spectral radius ρ, then leakage λ, bias scaling α_{bi}, and, finally, jointly the regularization parameter β and the number of neurons N^{res}.

It is important to note that the different implementations did not always have the same hyper-parameters. The HP-ELM toolbox does neither provide a constant bias input nor support input scaling. Thus, we could not optimize the two parameters α_u and α_{bi}, which we only optimized in PyRCN. The PyESN toolbox does not support leaky integration, and regularization, and does not provide a constant bias input. Thus, we could not optimize λ and α_{bi}, which we only optimized in PyRCN. Instead of optimizing β in PyESN, we optimized the noise level added to every neuron input. This is an alternative way to regularize ESNs (Jaeger, 2001).

One advantage of PyRCN is its compliance with the scikit-learn interface specification. This becomes noticeable when we optimize the hyper-parameters with model selection tools provided by scikit-learn. Although the HP-ELM interface does not comply with scikit-learn, the scikit-elm[15] adapter is available and we were able to use that for our experiments without any difficulties.

Such an adapter is not available for PyESN. We implemented it, which, however, slightly limits the power of PyESN, such as providing input scalings and input shifts with multiple dimensions (not required for this task c.f. Trierweiler Ribeiro et al., 2021). Implementing the adapter requires (a) knowledge about the scikit-learn interface and its

Table 3
Average values of volatility prediction metrics for 1-day ahead. The results of all models marked with a star symbol (*) are taken from Trierweiler Ribeiro et al. (2021), in which the PyESN toolbox was used. The differences between the literature and the results presented here arise from different parameter optimizations. Overall, the performance of different toolboxes was similar.

Model	R^2_{train}	R^2_{test}	MSE_{train} 10^{-8}	MSE_{test} 10^{-8}	N^{res}
ESN*$_{PyESN}$	0.634	0.632	5.74	5.81	–
ESN$_{PyESN}$	0.646	0.619	5.81	5.92	–
ESN$_{PyRCN}$	**0.681**	**0.644**	**5.23**	**5.62**	50
ELM$_{skELM}$	0.598	0.571	6.58	6.69	50
ELM$_{PyRCN}$	**0.609**	**0.578**	**6.41**	**6.62**	50
HAR*	0.650	0.633	5.78	5.81	–
HAR	0.653	0.636	5.69	5.73	–
HAR-ESN*$_{PyESN}$	0.637	0.635	5.75	5.78	–
HAR-ESN$_{PyESN}$	0.657	**0.642**	5.58	5.64	50
HAR-ESN$_{PyRCN}$	**0.663**	0.639	**5.52**	5.67	50
HAR-ELM$_{skELM}$	0.654	0.636	5.68	5.72	400
HAR-ELM$_{PyRCN}$	**0.655**	0.636	**5.66**	5.72	50

underlying objects; (b) at least 64 unoptimized lines of code in accordance with the PEP-8 guidelines for Python. This is a disadvantage of PyESN (and other references in Table 1) compared to PyRCN. Alternatively, it would have been possible to implement a custom randomized search routine, which, however, would require even more effort and would be less flexible than what `sklearn.model_selection.RandomizedSearchCV` provides.

6.4. Benchmark test

In the Tables 3–5, the reference results from Trierweiler Ribeiro et al. (2021) and the reproduced results by PyRCN and reference toolboxes are summarized. Overall, the results are similar. It is worth to note that Trierweiler Ribeiro et al. (2021) used a modified variant of the PyESN toolbox for their experiments. However, in this paper, we used the publicly available PyESN version and we used the same optimization scheme as for PyRCN. For all forecasting horizons, the results of PyESN and the ESN model in PyRCN are very similar. There is hardly any difference in the results between the ELM implementation of PyRCN and the HP-ELM. However, it can still be recognized that PyRCN outperformed the references in a majority of cases. This is mostly due to missing hyper-parameters such as leaky integration, a proper regularization or the bias scaling. PyRCN supports all these features and can thus be tuned for a task more thoroughly.

Interestingly, by limiting the optimization of the PyESN toolbox to the aforementioned hyper-parameters, we slightly outperformed Trierweiler Ribeiro et al. (2021) in many cases. This suggests that our proposed way of sequential optimization from Steiner et al. (2021, 2022) is also applicable for this kind of tasks.

Finally, we compare the fit and inference (score) times during the last step of the sequential hyper-parameter optimization, where we
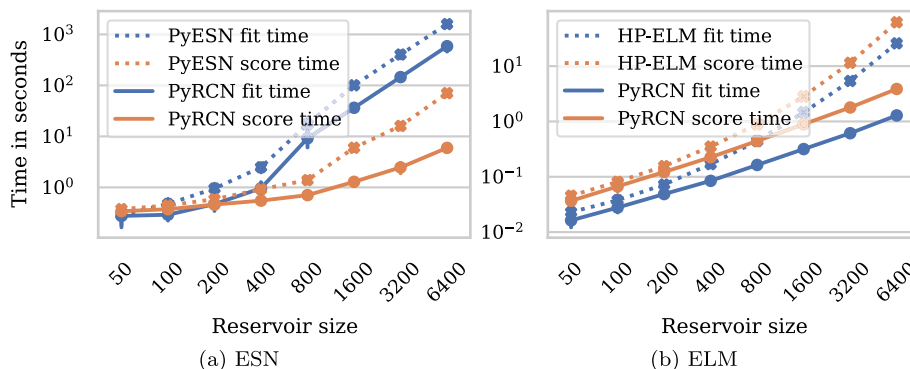
Fig. 4. Fit and score times for different RCN toolboxes. Especially for large ESNs, PyRCN requires less computational time than PyESN. Between PyRCN and HP-ELM, the difference gets smaller but is still present.

Table 4
Average values of volatility prediction metrics for 5-days ahead. The results of all models marked with a star symbol (*) are taken from Trierweiler Ribeiro et al. (2021), in which the PyESN toolbox was used. The differences between the literature and the results presented here arise from different parameter optimizations. Overall, the performance of different toolboxes was similar.

Model	R^2_{train}	R^2_{test}	MSE_{train} 10^{-8}	MSE_{test} 10^{-8}	N^{res}
ESN*$_{PyESN}$	0.485	0.480	8.27	5.81	–
ESN$_{PyESN}$	0.500	0.476	8.25	8.35	50
ESN$_{PyRCN}$	**0.519**	**0.499**	**7.93**	**8.02**	100
ELM$_{skELM}$	0.432	0.406	9.37	9.55	50
ELM$_{PyRCN}$	**0.445**	**0.411**	**9.17**	**9.51**	50
HAR*	0.499	0.481	8.33	5.81	–
HAR	0.509	0.490	8.11	8.17	–
HAR-ESN*$_{PyESN}$	0.535	0.510	7.53	5.78	–
HAR-ESN$_{PyESN}$	**0.557**	**0.513**	**7.29**	7.88	50
HAR-ESN$_{PyRCN}$	0.524	0.502	7.85	**7.96**	50
HAR-ELM$_{skELM}$	0.513	0.494	8.05	8.12	50
HAR-ELM$_{PyRCN}$	0.513	**0.495**	**8.04**	**8.11**	50

Table 5
Average values of volatility prediction metrics for 21-days ahead. The results of all models marked with a star symbol (*) are taken from Trierweiler Ribeiro et al. (2021), in which the PyESN toolbox was used. The differences between the literature and the results presented here arise from different parameter optimizations. Overall, the performance of different toolboxes was similar.

Model	R^2_{train}	R^2_{test}	MSE_{train} 10^{-7}	MSE_{test} 10^{-7}	N^{res}
ESN*$_{PyESN}$	0.268	0.264	1.18	1.19	–
ESN$_{PyESN}$	0.286	0.269	1.19	1.17	50
ESN$_{PyRCN}$	**0.362**	**0.316**	**1.06**	**1.10**	50
ELM$_{skELM}$	0.236	0.195	1.27	1.29	50
ELM$_{PyRCN}$	**0.256**	**0.218**	**1.24**	**1.26**	50
HAR*	0.256	0.224	1.25	1.27	–
HAR	0.295	0.261	1.17	1.19	–
HAR-ESN*$_{PyESN}$	0.297	0.298	1.11	1.16	–
HAR-ESN$_{PyESN}$	0.263	0.229	1.23	1.25	50
HAR-ESN$_{PyRCN}$	**0.357**	**0.310**	**1.07**	**1.11**	50
HAR-ELM$_{skELM}$	0.300	0.268	1.17	1.18	50
HAR-ELM$_{PyRCN}$	**0.306**	**0.272**	**1.16**	**1.17**	50

evaluated models with reservoir sizes between 50 and 6400 neurons and with different regularization methods. In Fig. 4, the mean values and standard deviations of the fit and score times for different reservoir sizes are summarized for the different considered toolboxes. In all cases (ESNs and ELMs), the fit and score times of small models were similar. However, in case of large models, the differences between PyRCN and the reference toolboxes get larger and partially differ by more than one order of magnitude, e.g. large ESNs (Fig. 4(a)). In fact, PyRCN is on

average ten times faster than PyESN. One possible reason is that PyESN is typically trained with feedback connections back from the output to the reservoir. Since this requires extra computations in each time step, this is computationally more expensive, in particular for larger reservoirs. In case of the HP-ELM, the differences are lower. However, the required computational time of HP-ELM still increases with a larger slope than the one of PyRCN (Fig. 4(b)). Overall, this shows that PyRCN can be used for small and for large datasets and reservoirs.

7. Conclusion and outlook

We presented PyRCN, a new Python toolbox compatible with scikit-learn that offers flexible components to design many kinds of RCNs such as ESNs and ELMs. It is relatively light-weight with structured modules. It allows for convenient hyper-parameter optimization using standard scikit-learn model selection routines. An incremental regression as implemented in the toolkit performs equivalently as default ridge regression by scikit-learn while requiring less memory and decreasing the training time in case of large reservoirs. The toolkit supports both basic ESN models and sequence-to-sequence and sequence-to-label varieties and provides functionalities to utilize basically all built-in model selection techniques provided by scikit-learn also for sequence-to-sequence and sequence-to-label handling.

We have demonstrated how to use the toolbox on two widely known tasks, namely time-series prediction with the Mackey-Glass equation and handwritten digit classification using the handwritten digits dataset. Based on a benchmark test, we have shown that we can reproduce the results by Trierweiler Ribeiro et al. (2021) using PyRCN with fewer lines of code, and that the sequential optimization scheme proposed and used by Steiner et al. (2021, 2020, 2022) is applicable to univariate time-series. By comparing PyRCN with reference toolboxes, we have shown that the required computational time of PyRCN is comparable in case of small reservoirs and outperforms the reference toolboxes by decades in case of large reservoirs as PyRCN was in the mean ten times faster than PyESN.

A current limitation is that PyRCN does not provide an interface to other established machine learning frameworks, e.g., PyTorch or Keras. In the future, we will continue to work on further optimizations of the underlying methods of the toolbox, especially parallel sequence processing, multi-layer ESN functionalities, and consider exposing additional interfaces to support other established machine learning frameworks. We strongly encourage enthusiastic reservoir computing practitioners and researchers to try out the toolbox and we welcome any feedback and contribution to improve it.

CRediT authorship contribution statement

Peter Steiner: Conceptualization, Methodology, Software, Evaluation, Writing - Original Draft, Writing - Review & Editing. **Azarakhsh Jalalvand:** Validation, Writing - Review & Editing. **Simon Stone:** Conceptualization, Methodology, Validation, Writing - Review & Editing. **Peter Birkholz:** Writing - Review & Editing, Supervision.

Declaration of competing interest

Acknowledgments

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X., 2016. TensorFlow: A system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation. OSDI 16, USENIX Association, Savannah, GA, pp. 265–283, URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

Akusok, A., Björk, K.-M., Miche, Y., Lendasse, A., 2015. High-performance extreme learning machines: A complete toolbox for big data applications. Access, IEEE 3, 1011–1025. http://dx.doi.org/10.1109/ACCESS.2015.2450498.

Antonelo, E.A., Schrauwen, B., 2015. On learning navigation behaviors for small mobile robots with reservoir computing architectures. IEEE Trans. Neural Netw. Learn. Syst. 26 (4), 763–780. http://dx.doi.org/10.1109/TNNLS.2014.2323247.

Antonelo, E.A., Schrauwen, B., Stroobandt, D., 2008. Event detection and localization for small mobile robots using reservoir computing. Neural Netw. 21 (6), 862–871. http://dx.doi.org/10.1016/j.neunet.2008.06.010, Computational and Biological Inspired Neural Networks, selected papers from ICANN 2007.

Basterrech, S., Fyfe, C., Rubino, G., 2011. Self-organizing maps and scale-invariant maps in echo state networks. In: 2011 11th International Conference on Intelligent Systems Design and Applications. pp. 94–99. http://dx.doi.org/10.1109/ISDA.2011.6121637.

Carmichael, Z., Syed, H., Burtner, S., Kudithipudi, D., 2018. Mod-DeepESN: Modular deep echo state network. CoRR, arXiv:1808.00523.

Cocco Mariani, V., Hennings Och, S., dos Santos Coelho, L., Domingues, E., 2019. Pressure prediction of a spark ignition single cylinder engine using optimized extreme learning machine models. Appl. Energy 249, 204–221. http://dx.doi.org/10.1016/j.apenergy.2019.04.126.

Corsi, F., 2009. A simple approximate long-memory model of realized volatility. J. Financ. Econom. 7 (2), 174–196. http://dx.doi.org/10.1093/jjfinec/nbp001, arXiv:https://academic.oup.com/jfec/article-pdf/7/2/174/2543795/nbp001.pdf.

DeepESN, 2019. DeepESN2019a: Deep echo state network (DeepESN) toolbox v1.1. https://www.mathworks.com/matlabcentral/fileexchange/69402-deepesn, last visited 2021-08-24.

Ding, S., Xu, X., Nie, R., 2014. Extreme learning machine and its applications. Neural Comput. Appl. 25 (3), 549–556.

Dua, D., Graff, C., 2017. UCI machine learning repository. URL http://archive.ics.uci.edu/ml.

EchoTorch, 2021. EchoTorch: Reservoir computing with PyTorch. https://github.com/nschaetti/EchoTorch, last visited 2021-08-24.

ESNToolbox, 2009 https://www.ai.rug.nl/minds/uploads/ESNToolbox.zip, last visited 2021-08-24.

Gallicchio, C., 2019. Chasing the echo state property. In: 27th European Symposium on Artificial Neural Networks, ESANN 2019, Bruges, Belgium, April 24-26, 2019. pp. 667–672, URL http://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2019-76.pdf.

Gallicchio, C., Micheli, A., Pedrelli, L., 2017. Deep reservoir computing: A critical experimental analysis. Neurocomputing 268, 87–99. http://dx.doi.org/10.1016/j.neucom.2016.12.089, URL http://www.sciencedirect.com/science/article/pii/S0925231217307567 Advances in artificial neural networks, machine learning and computational intelligence.

Glass, L., Mackey, M., 2010. Mackey-glass equation. Scholarpedia 5 (3), 6908.

HP-ELM, 2018. High performance toolbox for extreme learning machines. https://github.com/akusok/hpelm, last visited 2021-08-24.

Huang, G.-B., Zhu, Q.-Y., Siew, C.-K., 2006. Extreme learning machine: Theory and applications. Neurocomputing 70 (1), 489–501. http://dx.doi.org/10.1016/j.neucom.2005.12.126, Neural Networks.

Jaeger, H., 2001. The "Echo State" Approach to Analysing and Training Recurrent Neural Networks. Technical Report GMD Report 148, German National Research Center for Information Technology, URL http://www.faculty.iu-bremen.de/hjaeger/pubs/EchoStatesTechRep.pdf.

Jalalvand, A., Abbate, J., Conlin, R., Verdoolaege, G., Kolemen, E., 2021. Real-time and adaptive reservoir computing with application to profile prediction in fusion plasma. IEEE Trans. Neural Netw. Learn. Syst. 1–12. http://dx.doi.org/10.1109/TNNLS.2021.3085504.

Jalalvand, A., Demuynck, K., Neve, W.D., Martens, J.-P., 2018. On the application of reservoir computing networks for noisy image recognition. Neurocomputing 277, 237–248. http://dx.doi.org/10.1016/j.neucom.2016.11.100.

Jalalvand, A., Triefenbach, F., Demuynck, K., Martens, J.-P., 2015. Robust continuous digit recognition using reservoir computing. Comput. Speech Lang. 30 (1), 135–158. http://dx.doi.org/10.1016/j.csl.2014.09.006.

Jalalvand, A., Vandersmissen, B., Neve, W.D., Mannens, E., 2019. Radar signal processing for human identification by means of reservoir computing networks. In: 2019 IEEE Radar Conference (RadarConf). pp. 1–6. http://dx.doi.org/10.1109/RADAR.2019.8835753.

Katılmış, Z., Karakuzu, C., 2021. ELM based two-handed dynamic turkish sign language (TSL) word recognition. Expert Syst. Appl. 182, 115213. http://dx.doi.org/10.1016/j.eswa.2021.115213.

Kirby, K., 1991. Context dynamics in neural sequential learning. In: Proceedings of the Florida Artificial Intelligence Research Symposium FLAIRS. 1991, pp. 66–70.

Lazar, A., Pipa, G., Triesch, J., 2009. SORN: a self-organizing recurrent neural network. Front. Comput. Neurosci. 3, 23. http://dx.doi.org/10.3389/neuro.10.023.2009, URL https://www.frontiersin.org/article/10.3389/neuro.10.023.2009.

Liang, N., Huang, G., Saratchandran, P., Sundararajan, N., 2006. A fast and accurate online sequential learning algorithm for feedforward networks. IEEE Trans. Neural Netw. 17 (6), 1411–1423. http://dx.doi.org/10.1109/TNN.2006.880583.

LSM, 2020. Liquid state machine (LSM). https://github.com/IGITUGraz/LSM, last visited 2021-08-24.

Lukoševičius, M., 2012. A practical guide to applying echo state networks. In: Montavon, G., Orr, G.B., Müller, K.-R. (Eds.), Neural Networks: Tricks of the Trade: Second Edition. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 659–686. http://dx.doi.org/10.1007/978-3-642-35289-8_36.

Maass, W., Natschläger, T., Markram, H., 2002. Real-time computing without stable states: A new framework for neural computation based on perturbations. Neural Comput. 14 (11), 2531–2560. http://dx.doi.org/10.1162/089976602760407955.

Mansoor, M., Grimaccia, F., Leva, S., Mussetta, M., 2021. Comparison of echo state network and feed-forward neural networks in electrical load forecasting for demand response programs. Math. Comput. Simulation 184, 282–293. http://dx.doi.org/10.1016/j.matcom.2020.07.011, ELECTRIMACS 2019 ENGINEERING - Modelling and computational simulation for analysis and optimisation in electrical power engineering.

Mockus, J.B., Mockus, L.J., 1991. BayesIan approach to global optimization and application to multiobjective and constrained problems. J. Optim. Theory Appl. 70 (1), 157–172.

Moreno, S.R., da Silva, R.G., Mariani, V.C., dos Santos Coelho, L., 2020. Multi-step wind speed forecasting based on hybrid multi-stage decomposition model and long short-term memory neural network. Energy Convers. Manage. 213, 112869. http://dx.doi.org/10.1016/j.enconman.2020.112869.

Munakata, Y., Pfaffly, J., 2004. Hebbian learning and development. Dev. Sci. 7 (2), 141–148. http://dx.doi.org/10.1111/j.1467-7687.2004.00331.x.

Neumann, K., Steil, J.J., 2011. Batch intrinsic plasticity for extreme learning machines. In: Honkela, T., Duch, W., Girolami, M., Kaski, S. (Eds.), Artificial Neural Networks and Machine Learning – ICANN 2011. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 339–346.

Oger, 2013. Organic reservoir computing engine. https://github.com/neuronalX/Oger, last visited 2021-08-24.

Oubbati, M., Schanz, M., Buchheim, T., Levi, P., 2006. Velocity control of an omnidirectional RoboCup player with recurrent neural networks. In: Bredenfeld, A., Jacoff, A., Noda, I., Takahashi, Y. (Eds.), RoboCup 2005: Robot Soccer World Cup IX. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 691–701.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S., 2019. PyTorch: An imperative style, high-performance deep learning library. In: Wallach, H., Larochelle, H., Beygelzimer, A., dÁlché-Buc, F., Fox, E., Garnett, R. (Eds.), Advances in Neural Information Processing Systems, vol. 32. Curran Associates, Inc., pp. 1–12, URL https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E., 2011. Scikit-learn: Machine learning in Python. J. Mach. Learn. Res. 12, 2825–2830.

PyRCN, 2022. Python reservoir computing networks. https://github.com/TUD-STKS/PyRCN, last visited 2021-08-24.

Python-ELM, 2019. Python-ELM. https://github.com/masaponto/Python-ELM, last visited 2021-08-24.

PyTorch-ESN, 2021. Pytorch-esn. https://github.com/stefanonardo/pytorch-esn, last visited 2021-08-24.

ReservoirPy, 2022. Reservoirpy (v0.3). https://github.com/reservoirpy/reservoirpy, last visited 2021-08-24.

Rodan, A., Tino, P., 2011. Minimum complexity echo state network. IEEE Trans. Neural Netw. 22 (1), 131–144. http://dx.doi.org/10.1109/TNN.2010.2089641.

Salmen, M., Ploger, P., 2005. Echo state networks used for motor control. In: Proceedings of the 2005 IEEE International Conference on Robotics and Automation. pp. 1953–1958. http://dx.doi.org/10.1109/ROBOT.2005.1570399.

Schaetti, N., 2018. EchoTorch: Reservoir computing with pytorch. GitHub Repos. GitHub https://github.com/nschaetti/EchoTorch.

Schrauwen, B., Campenhout, J.V., 2003. BSA, a fast and accurate spike train encoding scheme. In: Proceedings of the International Joint Conference on Neural Networks, vol. 4. pp. 2825–2830. http://dx.doi.org/10.1109/IJCNN.2003.1224019.

Schrauwen, B., Verstraeten, D., Campenhout, J.M.V., 2007. An overview of reservoir computing: theory, applications and implementations. In: ESANN 2007, 15th European Symposium on Artificial Neural Networks, Bruges, Belgium, April 25-27, 2007, Proceedings. pp. 471–482, URL https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2007-8.pdf.

Schrauwen, B., Wardermann, M., Verstraeten, D., Steil, J.J., Stroobandt, D., 2008. Improving reservoirs using intrinsic plasticity. Neurocomputing 71 (7), 1159–1171. http://dx.doi.org/10.1016/j.neucom.2007.12.020, URL http://www.sciencedirect.com/science/article/pii/S0925231208000519 Progress in Modeling, Theory, and Application of Computational Intelligence.

da Silva, R.G., Ribeiro, M.H.D.M., Moreno, S.R., Mariani, V.C., dos Santos Coelho, L., 2021. A novel decomposition-ensemble learning framework for multi-step ahead wind energy forecasting. Energy 216, 119174. http://dx.doi.org/10.1016/j.energy.2020.119174.

Steiner, P., Jalalvand, A., Birkholz, P., 2022. Cluster-based input weight initialization for echo state networks. IEEE Trans. Neural Netw. Learn. Syst. 1–12. http://dx.doi.org/10.1109/TNNLS.2022.3145565.

Steiner, P., Jalalvand, A., Stone, S., Birkholz, P., 2020. Feature engineering and stacked echo state networks for musical onset detection. In: 2020 25th International Conference on Pattern Recognition. ICPR, pp. 9537–9544.

Steiner, P., Stone, S., Birkholz, P., Jalalvand, A., 2021. Multipitch tracking in music signals using echo state networks. In: 2020 28th European Signal Processing Conference. EUSIPCO, pp. 126–130. http://dx.doi.org/10.23919/Eusipco47968.2020.9287638.

Tang, J., Deng, C., Huang, G., 2016. Extreme learning machine for multilayer perceptron. IEEE Trans. Neural Netw. Learn. Syst. 27 (4), 809–821. http://dx.doi.org/10.1109/TNNLS.2015.2424995.

Triefenbach, F., Jalalvand, A., Demuynck, K., Martens, J.-P., 2013. Acoustic modeling with hierarchical reservoirs. IEEE Trans. Audio Speech Lang. Process. 21 (11), 2439–2450. http://dx.doi.org/10.1109/TASL.2013.2280209.

Triefenbach, F., Martens, J.-P., 2011. Can non-linear readout nodes enhance the performance of reservoir-based speech recognizers? In: 2011 First International Conference on Informatics and Computational Intelligence. pp. 262–267. http://dx.doi.org/10.1109/ICI.2011.50.

Trierweiler Ribeiro, G., Alves Portela Santos, A., Cocco Mariani, V., dos Santos Coelho, L., 2021. Novel hybrid model based on echo state neural network applied to the prediction of stock price return volatility. Expert Syst. Appl. 184, 115490. http://dx.doi.org/10.1016/j.eswa.2021.115490.

Trouvain, N., Pedrelli, L., Dinh, T.T., Hinaut, X., 2020. ReservoirPy: An efficient and user-friendly library to design echo state networks. In: Farkaš, I., Masulli, P., Wermter, S. (Eds.), Artificial Neural Networks and Machine Learning – ICANN 2020. Springer International Publishing, Cham, pp. 494–505.

Verstraeten, D., Schrauwen, B., Dieleman, S., Brakel, P., Buteneers, P., Pecevski, D., 2012. Oger: Modular learning architectures for large-scale sequential processing. J. Mach. Learn. Res. 13, 2995–2998.

Wang, L., Lv, S.-X., Zeng, Y.-R., 2018. Effective sparse adaboost method with ESN and FOA for industrial electricity consumption forecasting in China. Energy 155, 1013–1031. http://dx.doi.org/10.1016/j.energy.2018.04.175.