# pyrcn_tutorial

December 12, 2022

## 1 Building blocks of Reservoir Computing

[ ]:
```
from sklearn.datasets import make_blobs


# Generate a toy dataset
U, y = make_blobs(n_samples=100, n_features=10)
```

### 1.1 Input-to-Node

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

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

we can describe the non-linear projection of the input features $\mathbf{u}[n]$ into the high-dimensional reservoir space $\mathbf{r}'[n]$ via the non-linear input activation function $f'(\cdot)$.

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 $\alpha_{\text{u}}$. Since in case of a high dimensional input feature space and/or large reservoir sizes $N^{\text{res}}$, this leads to a huge input weight matrix and expensive computations to feed the feature vectors into the reservoir, it was shown that 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 $K^{\text{in}}$ ($\ll N^{\text{in}}$) 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 $N^{\text{res}}$ are typically initialized by fixed random values from a uniform distribution between $\pm 1$ and multiplied by the hyper-parameter $\alpha_{\text{bi}}$.

[ ]:
```
from pyrcn.base.blocks import InputToNode
```

[ ]:
```
#      _ _ _ _ _ _ _ _
#     |               |
# ----| Input-to-Node |------
# u[n]|_ _ _ _ _ _ _ _|r'[n]
```

```
# U                     R_i2n

input_to_node = InputToNode(hidden_layer_size=50,
                            k_in=5, input_activation="tanh",
                            input_scaling=1.0, bias_scaling=0.1)
```

[ ]:
```
%%time

R_i2n = input_to_node.fit_transform(U)
print(U.shape, R_i2n.shape)
```

[ ]:
```
%%timeit

R_i2n = input_to_node.fit_transform(U)
```

### 1.2 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

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

which is a leaky integration of the time-dependent reservoir states $\mathbf{r}[n]$. $f(\cdot)$ acts as the non-linear reservoir activation functions 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 $\mathbf{W}^{\text{res}}$ is a square matrix of the size $N^{\text{res}}$. 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 fulfill the ESP, the reservoir weight matrix is typically normalized by its largest absolute eigenvalue and rescaled to a spectral radius $\rho$, because it was shown that the ESP holds as long as $\rho \leq 1$. 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 $N^{\text{res}}$, it can be sufficient to only connect each node in the reservoir only to $K^{\text{rec}}$ ($\ll N^{\text{res}}$) randomly selected other nodes in the reservoir, and to set the remaining weights to zero.

To incorporate some information from the future inputs, bidirectional RCNs have been introduced.

[ ]:
```
from pyrcn.base.blocks import NodeToNode
```

[ ]:
```
#      _ _ _ _ _ _ _ _        _ _ _ _ _ _ _
#     |               |      |             |
# ----| Input-to-Node |------| Node-to-Node |------
# u[n]|_ _ _ _ _ _ _ _|r'[n] |_ _ _ _ _ _ _ |r[n]
# U                    R_i2n                R_n2n
```

```
# Initialize, fit and apply NodeToNode
node_to_node = NodeToNode(hidden_layer_size=50,
                          reservoir_activation="tanh",
                          spectral_radius=1.0, leakage=0.9,
                          bidirectional=False)
```

[ ]:
```
%%time

R_n2n = node_to_node.fit_transform(R_i2n)
print(U.shape, R_n2n.shape)
```

[ ]:
```
%%timeit

R_n2n = node_to_node.fit_transform(R_i2n)
```

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 $\mathbf{W}^{\text{out}}$ can be computed using

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

where $\epsilon$ is a regularization parameter.

The size of the output weight matrix $N^{\text{out}} \times (N^{\text{res}} + 1)$ or $N^{\text{out}} \times (2 \times N^{\text{res}} + 1)$ 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 Equation

$$\mathbf{y}[n] = \mathbf{W}^{\text{out}}\mathbf{r}[n] \ .$$

Note that, in general, other training methodologies could be used to compute output weights.

[ ]:
```
from sklearn.linear_model import Ridge
```

[ ]:
```
#       _ _ _ _ _ _ _       _ _ _ _ _ _ _       _ _ _ _ _ _ _
#     |               |     |             |     |               |
# ----|Input-to-Node |-----|Node-to-Node |-----|Node-to-Output |
# u[n]| _ _ _ _ _ _ _|r'[n]|_ _ _ _ _ _ _|r[n] | _ _ _ _ _ _ _ |
# U                   R_i2n               R_n2n         |
#                                                       |
#                                                 y[n]  | y_pred

# Initialize, fit and apply NodeToOutput
y_pred = Ridge().fit(R_n2n, y).predict(R_n2n)
```

```
print(y_pred.shape)
```

### 1.3 Predict the Mackey-Glass equation

Set up and train vanilla RCNs for predicting the Mackey-Glass time series with the same settings as used to introduce ESNs. The minimum working example shows the simplicity of implementing a model with PyRCN and the inter-operability with scikit-learn; 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. Instead of the default incremental regression, we have customized the `ELMRegressor()` by using `Ridge` from scikit-learn.

[ ]:
```
from sklearn.linear_model import Ridge as skRidge
from pyrcn.echo_state_network import ESNRegressor
from pyrcn.extreme_learning_machine import ELMRegressor
from pyrcn.datasets import mackey_glass
```

[ ]:
```
# Load the dataset
X, y = mackey_glass(n_timesteps=5000)
# Define Train/Test lengths
trainLen = 1900
X_train, y_train = X[:trainLen], y[:trainLen]
X_test, y_test = X[trainLen:], y[trainLen:]

# Initialize and train an ELMRegressor and an ESNRegressor
esn = ESNRegressor().fit(X=X_train.reshape(-1, 1), y=y_train)
elm = ELMRegressor(regressor=skRidge()).fit(X=X_train.reshape(-1, 1), y=y_train)

print("Fitted models")
```

[ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
```

[ ]:
```
fig, axs = plt.subplots()

sns.lineplot(x=list(range(len(y_test))), y=y_test, ax=axs)
sns.lineplot(x=list(range(len(y_test))), y=esn.predict(X_test.reshape(-1, 1)),␣
 ↪ax=axs)
sns.lineplot(x=list(range(len(y_test))), y=elm.predict(X_test.reshape(-1, 1)),␣
 ↪ax=axs)
```

## 2 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.

## 3 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 $\mathbf{r}'[n]$.
2. Compute the output weights $\mathbf{W}^{\text{out}}$ with $\mathbf{R}'$.

[ ]:
```
U, y = make_blobs(n_samples=100, n_features=10)
from pyrcn.extreme_learning_machine import ELMRegressor
```

[ ]:
```
# Vanilla ELM for regression tasks with input_scaling
#       _ _ _ _ _ _ _        _ _ _ _ _ _ _
#     |               |     |               |
# ----|Input-to-Node |-----|Node-to-Output |------
# u[n]| _ _ _ _ _ _ _|r'[n]| _ _ _ _ _ _ _ |y[n]
#                                          y_pred
#
vanilla_elm = ELMRegressor(input_scaling=0.9)
vanilla_elm.fit(U, y)
print(vanilla_elm.predict(U))
```

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

[ ]:
```
from pyrcn.base.blocks import BatchIntrinsicPlasticity
```

[ ]:
```
# Custom ELM with BatchIntrinsicPlasticity
#       _ _ _ _ _ _ _        _ _ _ _ _ _ _
#     |               |     |               |
# ----|     BIP       |-----|Node-to-Output |------
# u[n]| _ _ _ _ _ _ _|r'[n]| _ _ _ _ _ _ _ |y[n]
#                                          y_pred
#
bip_elm = ELMRegressor(input_to_node=BatchIntrinsicPlasticity(),
                       regressor=Ridge(alpha=1e-5))

bip_elm.fit(U, y)
print(bip_elm.predict(U))
```

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

[ ]:
```
from sklearn.pipeline import Pipeline, FeatureUnion
```

[ ]:
```
# ELM with cascaded InputToNode and default regressor
#       _ _ _ _ _ _ _        _ _ _ _ _ _ _        _ _ _ _ _ _ _
#     |     (bip)     |     |    (base)     |     |               |
```

```
# ----|Input-to-Node1|-----|Input-to-Node2|-----|Node-to-Output |
# u[n]| _ _ _ _ _ _ _|     | _ _ _ _ _ _ _|r'[n]| _ _ _ _ _ _ _ |
#                                                 |
#                                                 |
#                                             y[n] | y_pred
#
i2n = Pipeline([('bip', BatchIntrinsicPlasticity()),
                ('base', InputToNode(bias_scaling=0.1))])
casc_elm = ELMRegressor(input_to_node=i2n).fit(U, y)

# Ensemble of InputToNode with activations
#             _ _ _ _ _ _ _
#           |      (i)     |
#      |----|Input-to-Node1|-----|
#      |    | _ _ _ _ _ _ _|     |       _ _ _ _ _ _ _
#      |                          -----|               |
# -----o                           r'[n]|Node-to-Output |------
# u[n] |      _ _ _ _ _ _ _      |-----| _ _ _ _ _ _ _ |y[n]
#      |    |     (th)     |     |                      y_pred
#      |----|Input-to-Node2|-----|
#           | _ _ _ _ _ _ _|
#
i2n = FeatureUnion([('i', InputToNode(input_activation="identity")),
                    ('th', InputToNode(input_activation="tanh"))])
ens_elm = ELMRegressor(input_to_node=i2n)
ens_elm.fit(U, y)
print(casc_elm, ens_elm)
```

### 3.1 Echo State Networks

ESNs, as variants of RNNs, consist of an "Input-to-Node", a "Node-to-Node" and a "Node-to-Output" block and are trained in three steps:

1. Compute the neuron input states $\mathbf{R}'$, which is the collection of reservoir states $\mathbf{r}'[n]$. 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 $\mathbf{r}[n]$. Note that here the activation function $f(\cdot)$ is typically non-linear.
3. Compute the output weights $\mathbf{W}^{\text{out}}$ using
    1. Linear regression with $\mathbf{R}$ when considering an ESN.
    2. Backpropagation or other optimization algorithm when considering a CRN or when using an ESN with non-linear outputs.

What follows is an example of how to construct such a vanilla ESN with PyRCN, where the `ESNRegressor` internally passes the input features through "Input-to-Node" and "Node-to-Node", and fits "Node-to-Output" using `pyrcn.linear_model.IncrementalRegression`.

[ ]:
```
from pyrcn.echo_state_network import ESNRegressor
```

[ ]:
```
# Vanilla ESN for regression tasks with spectral_radius and leakage
#       _ _ _ _ _ _ _       _ _ _ _ _ _ _       _ _ _ _ _ _ _
#     |             |     |             |     |               |
# ----|Input-to-Node |-----|Node-to-Node |-----|Node-to-Output |
# u[n]| _ _ _ _ _ _ _|r'[n]|_ _ _ _ _ _ _|r[n] | _ _ _ _ _ _ _ |
#                                                     |
#                                                     |
#                                                y[n] | y_pred
#
vanilla_esn = ESNRegressor(spectral_radius=1, leakage=0.9)
vanilla_esn.fit(U, y)
print(vanilla_esn.predict(U))
```

As for ELMs, various unsupervised learning techniques can be used to pre-train "Input-to-Node" and "Node-to-Node".

[ ]:
```
from pyrcn.base.blocks import HebbianNodeToNode
```

[ ]:
```
# Custom ESN with BatchIntrinsicPlasticity and HebbianNodeToNode
#       _ _ _ _ _ _ _       _ _ _ _ _ _ _       _ _ _ _ _ _ _
#     |     (bip)   |     |   (hebb)    |     |               |
# ----|Input-to-Node |-----|Node-to-Node |-----|Node-to-Output |
# u[n]| _ _ _ _ _ _ _|r'[n]|_ _ _ _ _ _ _|r[n] | _ _ _ _ _ _ _ |
#                                                     |
#                                                     |
#                                                y[n] | y_pred
#
bip_esn = ESNRegressor(input_to_node=BatchIntrinsicPlasticity(),
                       node_to_node=HebbianNodeToNode(),
                       regressor=Ridge(alpha=1e-5))

bip_esn.fit(U, y)
print(bip_esn.predict(U))
```

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

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.

[ ]:
```
# Multilayer ESN
#                   u[n]
#                    |
#                    |
#           _________o_________
#          |                   |
#    _ _ _ | _ _ _       _ _ _ | _ _ _
# |      (i)     |     |      (i)     |
# |Input-to-Node1|     |Input-to-Node2|
```

```
# | _ _ _ _ _ _ _ _|    | _ _ _ _ _ _ _ _|
#         |r1'[n]              | r2'[n]
#   _ _ _ | _ _ _        _ _ _ | _ _ _
# |     (th)     |    |     (th)     |
# | Node-to-Node1|    | Node-to-Node2|
# | _ _ _ _ _ _ _ |    | _ _ _ _ _ _ _ |
#         |r1[n]               | r2[n]
#         |_____          _____|
#               |       |
#             _ | _ _ _ | _
#           |                 |
#           | Node-to-Node3   |
#           | _ _ _ _ _ _ _ _ |
#                   |
#             r3[n]|
#             _ _ _ | _ _ _
#           |                 |
#           |Node-to-Output   |
#           | _ _ _ _ _ _ _ _ |
#                   |
#              y[n]|

l1 = Pipeline([('i2n1', InputToNode(hidden_layer_size=100)),
               ('n2n1', NodeToNode(hidden_layer_size=100))])

l2 = Pipeline([('i2n2', InputToNode(hidden_layer_size=400)),
               ('n2n2', NodeToNode(hidden_layer_size=400))])

i2n = FeatureUnion([('l1', l1),
                    ('l2', l2)])
n2n = NodeToNode(hidden_layer_size=500)
layered_esn = ESNRegressor(input_to_node=i2n,
                           node_to_node=n2n)

layered_esn.fit(U, y)
print(layered_esn.predict(U))
```

[ ]:
```
import numpy as np
from sklearn.decomposition import PCA


# Multiple small reservoirs with different leakages in parallel
res1 = FeatureUnion([
    ("lambda_0.1",
     Pipeline([('i2n', InputToNode(hidden_layer_size=10)),
               ('n2n', NodeToNode(hidden_layer_size=10,
                                  leakage=0.1))])),
```

```
    ("lambda_0.2",
     Pipeline([('i2n', InputToNode(hidden_layer_size=10)),
               ('n2n', NodeToNode(hidden_layer_size=10,
                                  leakage=0.2))])),
    ("lambda_0.3",
     Pipeline([('i2n', InputToNode(hidden_layer_size=10)),
               ('n2n', NodeToNode(hidden_layer_size=10,
                                  leakage=0.3))])),
    ("lambda_0.4",
     Pipeline([('i2n', InputToNode(hidden_layer_size=10)),
               ('n2n', NodeToNode(hidden_layer_size=10,
                                  leakage=0.4))])),])

pca = PCA(n_components=10)

res2 = Pipeline([("i2n", InputToNode(hidden_layer_size=100)),
                 ("n2n", NodeToNode(hidden_layer_size=100))])

i2n = FeatureUnion([("path1",
                     Pipeline([("res1", res1), ("pca", pca),
                               ("res2", res2)])),
                    ("path2", res1)])

n2n = NodeToNode(spectral_radius=0., leakage=1., hidden_layer_size=100+40,
                 predefined_recurrent_weights=np.eye(40+100))

deep_esn = ESNRegressor(input_to_node=i2n, node_to_node=n2n)
deep_esn.fit(U, y)
print(deep_esn.predict(U))
```

### 3.2 Complex example: Optimize the hyper-parameters of RCNs

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.

[ ]:
```
from sklearn.metrics import make_scorer
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import TimeSeriesSplit
from sklearn.model_selection import (RandomizedSearchCV,
                                     GridSearchCV)
from scipy.stats import uniform
from pyrcn.model_selection import SequentialSearchCV
from pyrcn.datasets import mackey_glass
```

[ ]:
```
# Load the dataset
X, y = mackey_glass(n_timesteps=5000)
```

```
X_train, X_test = X[:1900], X[1900:]
y_train, y_test = y[:1900], y[1900:]

# Define initial ESN model
esn = ESNRegressor(bias_scaling=0, spectral_radius=0, leakage=1)

# Define optimization workflow
scorer = make_scorer(mean_squared_error, greater_is_better=False)
step_1_params = {'input_scaling': uniform(loc=1e-2, scale=1),
                 'spectral_radius': uniform(loc=0, scale=2)}
kwargs_1 = {'n_iter': 200, 'n_jobs': -1, 'scoring': scorer,
            'cv': TimeSeriesSplit()}
step_2_params = {'leakage': [0.2, 0.4, 0.7, 0.9, 1.0]}
kwargs_2 = {'verbose': 5, 'scoring': scorer, 'n_jobs': -1,
            'cv': TimeSeriesSplit()}

searches = [('step1', RandomizedSearchCV, step_1_params, kwargs_1),
            ('step2', GridSearchCV, step_2_params, kwargs_2)]

# Perform the search
esn_opti = SequentialSearchCV(esn, searches).fit(X_train.reshape(-1, 1),␣
 ↪y_train)
print(esn_opti)
```

### 3.3 Programming pattern for sequence processing

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 [1] that was developed to produce these results.

[ ]:
```
from sklearn.base import clone
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_validate
from sklearn.model_selection import ParameterGrid
from sklearn.metrics import make_scorer

from pyrcn.echo_state_network import ESNClassifier
from pyrcn.metrics import accuracy_score
from pyrcn.datasets import load_digits
```

[ ]:
```
# Load the dataset
X, y = load_digits(return_X_y=True, as_sequence=True)
print("Number of digits: {0}".format(len(X)))
print("Shape of digits {0}".format(X[0].shape))
# Divide the dataset into training and test subsets
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.2,
```

[1] https://github.com/TUD-STKS/PyRCN/blob/main/examples/digits.ipynb

```
                                            random_state=42)
print("Number of digits in training set: {0}".format(len(X_tr)))
print("Shape of the first digit: {0}".format(X_tr[0].shape))
print("Number of digits in test set: {0}".format(len(X_te)))
print("Shape of the first digit: {0}".format(X_te[0].shape))

# These parameters were optimized using SequentialSearchCV
esn_params = {'input_scaling': 0.05077514155476392,
              'spectral_radius': 1.1817858863764836,
              'input_activation': 'identity',
              'k_in': 5,
              'bias_scaling': 1.6045393364745582,
              'reservoir_activation': 'tanh',
              'leakage': 0.03470266988650412,
              'k_rec': 10,
              'alpha': 3.0786517836196185e-05,
              'decision_strategy': "winner_takes_all"}

b_esn = ESNClassifier(**esn_params)

param_grid = {'hidden_layer_size': [50, 100, 200, 400, 500],
              'bidirectional': [False, True]}

for params in ParameterGrid(param_grid):
    esn_cv = cross_validate(clone(b_esn).set_params(**params),
                            X=X_tr, y=y_tr,
                            scoring=make_scorer(accuracy_score))
    esn = clone(b_esn).set_params(**params).fit(X_tr, y_tr, n_jobs=1)
    acc_score = accuracy_score(y_te, esn.predict(X_te))
    print(acc_score)
```

[ ]: