aqua_blue

aqua-blue

Lightweight and basic reservoir computing library

gh-pages docs

pypi v0.2.17 python 3.9 | 3.10 | 3.11 | 3.12

What is aqua-blue?

`aqua-blue` is a lightweight `python` library for reservoir computing (specifically echo state networks) depending only on `numpy`. `aqua-blue`'s namesake comes from:

- A blue ocean of data, aka a reservoir
- A very fancy cat named Blue

Found a bug?

Please open an issue here if you found a bug! The easier it is to reproduce the bug, the faster we will find a solution to the problem. Please consider including the following info in your issue:

- Steps to reproduce
- Expected and actual behavior
- Version info, OS, etc.

Contributing

Please see CONTRIBUTING.md for instructions on how to contribute to `aqua-blue`

Installation

`aqua-blue` is on PyPI, and can therefore be installed with `pip`:

```
pip install aqua-blue
```

Quickstart

```
import numpy as np
import aqua_blue


# generate arbitrary two-dimensional time series
```

```
# y_1(t) = cos(t), y_2(t) = sin(t)
# resulting dependent variable has shape (number of timesteps, 2)
t = np.linspace(0, 4.0 * np.pi, 10_000)
y = np.vstack((2.0 * np.cos(t) + 1, 5.0 * np.sin(t) - 1)).T

# create time series object to feed into echo state network
time_series = aqua_blue.time_series.TimeSeries(dependent_variable=y, times=t)

# normalize
normalizer = aqua_blue.utilities.Normalizer()
time_series = normalizer.normalize(time_series)

# make model and train
model = aqua_blue.models.Model(
    reservoir=aqua_blue.reservoirs.DynamicalReservoir(
        reservoir_dimensionality=100,
        input_dimensionality=2
    ),
    readout=aqua_blue.readouts.LinearReadout()
)
model.train(time_series)

# predict and denormalize
prediction = model.predict(horizon=1_000)
prediction = normalizer.denormalize(prediction)
```

License

`aqua-blue` is released under the MIT License.

Blue, the cat behind `aqua-blue`.

Examples

Basic Lotka-Volterra example

Below is an example of using `aqua-blue` to predict the predator-prey [Lotka-Volterra equations]:

$$\dot{x} = \alpha x - \beta xy$$

$$\dot{y} = -\gamma y + \delta xy$$

with parameters $\alpha = 0.1$, $\beta = 0.02$, $\gamma = 0.3$, and $\delta = 0.01$, and initial conditions $(x_0, y_0) = (20, 9)$. We train a reservoir computer with a reservoir dimensionality of 1000 over $0 \leq t \leq 10$, with 1000 timesteps. Then, we predict the next 1000 timesteps.

Here, we use `scipy.integrate.solve_ivp` to integrate the system of differential equations.

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt
import aqua_blue


def lotka_volterra(t, z, alpha, beta, delta, gamma):
    x, y = z
    dxdt = alpha * x - beta * x * y
    dydt = delta * x * y - gamma * y
    return [dxdt, dydt]


def solve_lv(t_start, t_end, no, alpha=0.1, beta=0.02, gamma=0.3, delta=0.01, x0=20,
 y0=9):

    t_eval = np.linspace(t_start, t_end, no)
    solution = solve_ivp(lotka_volterra, [t_start, t_end], [x0, y0], t_eval=t_eval,
 args=(alpha, beta, delta, gamma))
    x, y = solution.y
    lotka_volterra_array = np.vstack((x, y)).T
    return lotka_volterra_array


def main():

    y = solve_lv(0, 10, 1000)
    t = np.linspace(0, 10, 1000)

    time_series = aqua_blue.time_series.TimeSeries(dependent_variable=y, times=t)
```

```python
    normalizer = aqua_blue.utilities.Normalizer()
    time_series = normalizer.normalize(time_series)

    model = aqua_blue.models.Model(
        reservoir=aqua_blue.reservoirs.DynamicalReservoir(
            reservoir_dimensionality=100,
            input_dimensionality=2
        ),
        readout=aqua_blue.readouts.LinearReadout()
    )
    model.train(time_series)

    prediction = model.predict(horizon=1_000)
    prediction = normalizer.denormalize(prediction)

    actual_future = solve_lv(prediction.times[0], prediction.times[-1], 1_000)

    plt.plot(prediction.times, actual_future)
    plt.xlabel('t')
    plt.plot(prediction.times, prediction.dependent_variable)
    plt.legend(['actual x', 'actual y', 'predicted x', 'predicted y'], shadow=True)
    plt.title('Lotka-Volterra System')
    plt.show()


if __name__ == "__main__":

    main()
```

Using datetime objects

Below is an example of a simple sine-cosine task similar to above, using `datetime.datetime` objects as times.

```python
import numpy as np
import matplotlib.pyplot as plt

import datetime
from zoneinfo import ZoneInfo

import aqua_blue


def main():
```

```
    start_date = datetime.datetime.now().astimezone(ZoneInfo("Indian/Maldives"))

    # Generate 10,000 datetime objects, each 1 minute apart
    t = [start_date + datetime.timedelta(minutes=i) for i in range(10000)]
    a = np.arange(len(t)) / 100
    y = np.vstack((np.cos(a)+1, np.sin(a)-1)).T
    time_series = aqua_blue.time_series.TimeSeries(dependent_variable=y, times=t)
    normalizer = aqua_blue.utilities.Normalizer()
    time_series = normalizer.normalize(time_series)

    model = aqua_blue.models.Model(
        reservoir=aqua_blue.reservoirs.DynamicalReservoir(
            reservoir_dimensionality=100,
            input_dimensionality=2
        ),
        readout=aqua_blue.readouts.LinearReadout()
    )
    model.train(time_series)

    horizon = 1_000
    prediction = model.predict(horizon=horizon)
    prediction = normalizer.denormalize(prediction)

    dt = np.diff(a)[0]
    actual_future = np.vstack((
        (np.cos(a[-1] + dt * np.arange(horizon)) + 1, np.sin(a[-1] + dt *
np.arange(horizon)) - 1)
    )).T

    root_mean_square_error = np.sqrt(np.mean((actual_future -
prediction.dependent_variable) ** 2))

    print(root_mean_square_error)
    plt.plot(prediction.times, actual_future)
    plt.plot(prediction.times, prediction.dependent_variable)
    plt.legend(['actual x', 'actual y', 'predicted x', 'predicted y'])
    plt.show()


if __name__ == "__main__":

    main()
```

Load and output a JSON string

Below is an example of inputting a `json` string as the training data, and outputting a `json` string for the prediction. This is particularly useful for interfacing `aqua-blue` with already-existing systems.

```
import json

import aqua_blue


def main():

    # some string that is valid json
    json_str = """
{
"dependent_variable": [
[2.0, -1.0],
[1.5403023058681398, -0.1585290151921035],
[0.5838531634528576, -0.09070257317431829],
[0.010007503399554585, -0.8588799919401328],
[0.34635637913638806, -1.7568024953079282],
[1.2836621854632262, -1.9589242746631386],
[1.9601702866503659, -1.2794154981989259],
[1.7539022543433047, -0.34301340128121094],
[0.8544999661913865, -0.010641753376618213],
[0.08886973811532306, -0.5878815147582435]
],
"times": [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
}
"""

    # turn into a TimeSeries instance
    time_series = aqua_blue.time_series.TimeSeries(**json.loads(json_str))

    # normalize and feed into model
    normalizer = aqua_blue.utilities.Normalizer()
    normalized_time_series = normalizer.normalize(time_series)

    model = aqua_blue.models.Model(
        reservoir=aqua_blue.reservoirs.DynamicalReservoir(
            reservoir_dimensionality=100,
            input_dimensionality=2
        ),
```

```
        readout=aqua_blue.readouts.LinearReadout()
    )
    model.train(normalized_time_series)

    # predict 5 more steps
    horizon = 5
    prediction = model.predict(horizon=horizon)
    prediction = normalizer.denormalize(prediction)

    # concatenate prediction and original time series, and print out a new json
    concatenated = time_series >> prediction

    # turn into a dictionary and json dump it
    print(json.dumps(concatenated.to_dict()))


if __name__ == "__main__":

    main()
```

Explicit weights

Below is an example of generating explicit matrices for W_{in} and W_{res}. Here, `sparsity=0.99` and `spectral_radius=1.2` respectively zero-out 99% of W_{res}'s elements and force W_{res} to have a spectral radius of 1.2. We also showcase the `>>` operator, which concatenates instances of `aqua_blue.time_series.TimeSeries`.

```
import numpy as np
import matplotlib.pyplot as plt

import aqua_blue


def main():
    t = np.arange(5_000) / 100
    y = np.vstack((np.cos(t) ** 2, np.sin(t))).T

    time_series = aqua_blue.time_series.TimeSeries(dependent_variable=y, times=t)
    normalizer = aqua_blue.utilities.Normalizer()
    normalized_time_series = normalizer.normalize(time_series)
    generator = np.random.default_rng(seed=0)

    w_res = generator.uniform(
        low=-0.5,
```

```
        high=0.5,
        size=(100, 100)
    )
    w_in = generator.uniform(
        low=-0.5,
        high=0.5,
        size=(100, 2)
    )

    model = aqua_blue.models.Model(
        reservoir=aqua_blue.reservoirs.DynamicalReservoir(
            reservoir_dimensionality=100,
            input_dimensionality=2,
            w_res=w_res,
            w_in=w_in,
            spectral_radius=1.2,
            sparsity=0.99
        ),
        readout=aqua_blue.readouts.LinearReadout()
    )
    model.train(normalized_time_series)

    horizon = 1_000
    prediction = model.predict(horizon=horizon)
    prediction = normalizer.denormalize(prediction)

    actual_future = aqua_blue.time_series.TimeSeries(
        dependent_variable=np.vstack((np.cos(prediction.times) ** 2,
 np.sin(prediction.times))).T,
        times=prediction.times
    )

    ground_truth = time_series >> actual_future
    predicted = time_series >> prediction

    plt.plot(ground_truth.times, ground_truth.dependent_variable)
    plt.plot(predicted.times, predicted.dependent_variable)
    plt.axvline(time_series.times[-1], color="black", linestyle="--")
    plt.legend(['actual x', 'actual y', 'predicted x', 'predicted y', 'knowledge
horizon'])
    plt.title("Explicit Weight Matrix Example")

    plt.show()
```

```
if __name__ == "__main__":

    main()
```

Explicit activation function

Below is an example of using a different activation function to map from the input state to the reservoir. Here, we use both hyperbolic tangent (`tanh`) and the error function (`erf`), and compare the results.

```
from typing import Dict, Callable

import numpy as np
import matplotlib.pyplot as plt
import scipy

import aqua_blue


def main():

    fig, axs = plt.subplots(nrows=1, ncols=2, sharex=True, sharey=True)
    activation_functions: Dict[str, Callable] = {"tanh": np.tanh, "erf":
 scipy.special.erf}

    t = np.arange(10_000) / 100
    y = np.vstack((np.cos(t) ** 2, np.sin(t))).T

    for ax, activation_function in zip(axs, activation_functions):
        time_series = aqua_blue.time_series.TimeSeries(dependent_variable=y, times=t)
        normalizer = aqua_blue.utilities.Normalizer()
        time_series = normalizer.normalize(time_series)

        model = aqua_blue.models.Model(
            reservoir=aqua_blue.reservoirs.DynamicalReservoir(
                reservoir_dimensionality=100,
                input_dimensionality=2,
                activation_function=activation_functions[activation_function]
            ),
            readout=aqua_blue.readouts.LinearReadout()
        )
        model.train(time_series)
```

```
        prediction = model.predict(horizon=1_000)
        prediction = normalizer.denormalize(prediction)

        actual_future = np.vstack((np.cos(prediction.times) ** 2,
 np.sin(prediction.times))).T

        ax.plot(prediction.times, actual_future)
        ax.plot(prediction.times, prediction.dependent_variable)
        ax.legend(['actual_x', 'actual_y', 'predicted_x', 'predicted_y'])
        ax.set_title(activation_function)

    plt.show()


if __name__ == "__main__":

    main()
```

HTTP Requests

Below is an example of pulling csv file data from a resource URL using the requests library. Here, we retrieve a time series of temperature data from NCEI NOAA and use it for training and predicting temperatures.

```
 import requests
import aqua_blue
from io import BytesIO
import matplotlib.pyplot as plt
import numpy as np
import time

def main():
    req = requests.get("https://www.ncei.noaa.gov/data/global-summary-of-the-
day/access/2024/01001099999.csv")

    time_col = "DATE"
    dependent_var_cols = ["TEMP"]

    with BytesIO() as file:
        txt = req.text
        file.write(txt.encode('utf-8'))
        file.seek(0)

        DATA = aqua_blue.time_series.TimeSeries.from_csv(
```

```
        fp=file,
        time_col=time_col,
        dependent_var_cols=dependent_var_cols,
        times_dtype='datetime64[s]',
        max_rows=87,
    )

TRAIN_DATA = aqua_blue.time_series.TimeSeries(
    times=DATA.times[:82],
    dependent_variable=DATA.dependent_variable[:82, :]
)

normalizer = aqua_blue.utilities.Normalizer()
normalized_time_series = normalizer.normalize(TRAIN_DATA)

seed = int(time.time())
generator = np.random.default_rng(seed)

w_res = generator.uniform(
    low=-0.5,
    high=0.5,
    size=(100, 100)
)
w_in = generator.uniform(
    low=-0.5,
    high=0.5,
    size=(100, 1)
)

model = aqua_blue.models.Model(
    reservoir=aqua_blue.reservoirs.DynamicalReservoir(
        reservoir_dimensionality=100,
        w_in = w_in,
        w_res = w_res,
        input_dimensionality=1,
    ),
    readout=aqua_blue.readouts.LinearReadout(1e-1)
)

model.train(normalized_time_series)

horizon = 5
prediction = model.predict(horizon=horizon)
prediction = normalizer.denormalize(prediction)
```

```
    concatenated = TRAIN_DATA >> prediction

    plt.plot(concatenated.times, concatenated.dependent_variable, label='Predicted
Future')
    plt.plot(DATA.times, DATA.dependent_variable, label='Actual Future')
    plt.legend()
    plt.show()


if __name__ == '__main__':
    main()
```

Read from and write to CSV files

Below is an example of parsing data from a csv file (`goldstocks.csv`) and writing it to a `TimeSeries` object, which is used for training and predictions. The predictions are written to a new csv file (`predicted-goldstocks.csv`).

```
import aqua_blue
from pathlib import Path

def main():

    goldstocks = aqua_blue.time_series.TimeSeries.from_csv(
        fp=Path('examples/goldstocks.csv'),
        time_col='DATE',
        times_dtype='datetime64[s]',
        dependent_var_cols=['X', 'Y', 'Z', 'A', 'B'],
    )

    normalizer = aqua_blue.utilities.Normalizer()
    normalized_time_series = normalizer.normalize(goldstocks)

    model = aqua_blue.models.Model(
        reservoir=aqua_blue.reservoirs.DynamicalReservoir(
            reservoir_dimensionality=100,
            input_dimensionality=5
        ),
        readout=aqua_blue.readouts.LinearReadout()
    )

    model.train(normalized_time_series)
```

```
    horizon = 100
    prediction = model.predict(horizon=horizon)
    prediction = normalizer.denormalize(prediction)

    concatenated = goldstocks >> prediction

    concatenated.save(
        fp=Path('examples/predicted_goldstocks.csv'),
        header='DATE,X,Y,Z,A,B',
        fmt=('%s', '%.2f', '%.2f', '%.2f', '%.2f', '%.2f')
    )

if __name__ == '__main__':
    main()
```

Logging

`aqua-blue` utilizes the native `logging` library to do some additional logging. An example of this is below:

```
import logging
import sys

import numpy as np

import aqua_blue


def main():

    logging.basicConfig(stream=sys.stdout, level=logging.DEBUG)

    t = np.arange(5_000) / 100
    y = np.vstack((np.cos(t) ** 2, np.sin(t))).T

    time_series = aqua_blue.time_series.TimeSeries(dependent_variable=y, times=t)
    normalizer = aqua_blue.utilities.Normalizer()
    normalized_time_series = normalizer.normalize(time_series)

    model = aqua_blue.models.Model(
        reservoir=aqua_blue.reservoirs.DynamicalReservoir(
            reservoir_dimensionality=100,
            input_dimensionality=2,
            sparsity=0.5,
```

```
            spectral_radius=0.95
        ),
        readout=aqua_blue.readouts.LinearReadout()
    )
    model.train(normalized_time_series)


if __name__ == "__main__":

    main()
```

which prints:

```
INFO:root:times dtype set to float64
INFO:root:times dtype set to float64
DEBUG:root:DynamicalReservoir.w_res sparsity set to 50.67%
DEBUG:root:DynamicalReservoir.w_res spectral radius set to 4.7707258199919655
INFO:root:LinearReadout layer trained. Inaccuracy = 5.025374978118052e-09 and pcc = 1.0
DEBUG:root:Model.timestep set to 0.01
DEBUG:root:Model.final_time set to 49.99
DEBUG:root:Model.tz set to None
DEBUG:root:Model.times_detype set to float64
```

For my favorite video about logging in Python, see a wonderful video below by mCoding:

aqua_blue.datetimelikearray

Module providing a timezone-aware wrapper for NumPy arrays.

Timezone awareness is a deprecated NumPy feature due to the deprecation of pytz. This module provides a workaround by storing the timezone information separately in the array. The datetime objects are stored in UTC and converted to the specified timezone when accessed.

This implementation is designed specifically for one-dimensional arrays and is intended to satisfy the datetime processing requirements of the project, rather than general NumPy timezone integration.

DatetimeLike = ~DatetimeLike

Datetime like, representing either dates or numerical values

TimeDeltaLike = ~TimeDeltaLike

Corresponding object representing timesteps, which are either float if the two times are floats, or a timedelta

```
class DatetimeLikeArray(numpy.ndarray):
```

A subclass of NumPy ndarray that provides timezone awareness for datetime arrays.

The timezone information is stored separately since NumPy does not natively support timezone-aware datetime objects. All datetime values are stored in UTC and converted back to the specified timezone when accessed.

```
DatetimeLikeArray( input_array: Sequence[~DatetimeLike], dtype, buffer=None, offset=0, strides=None, order=None)
```

Create a new instance of DatetimeLikeArray.

Arguments:

- **input_array (Sequence[DatetimeLike]):** List of datetime-like objects to be stored in the array.
- **dtype:** Data type for the NumPy array.
- **buffer:** Optional buffer for the array.
- **offset:** Offset for the array.
- **strides:** Strides for the array.
- **order:** Memory layout order.

Returns:

> DatetimeLikeArray: A new instance of the class.

```
tz: Optional[datetime.tzinfo] = None
```

The timezone associated with the array. Defaults to None (assumed UTC).

```
tz_offset: Optional[datetime.timedelta] = None
```

The timezone offset from UTC for the stored datetime values. Defaults to None.

```
def to_list(self) -> List[~DatetimeLike]:
```

Convert the array back to a list of datetime-like objects with timezone information.

Returns:

> List[DatetimeLike]: A list of datetime objects with their original timezone restored.

```
def to_file( self, fp: Union[IO, str, pathlib.Path], tz: Optional[datetime.tzinfo] = None):
```

Save a DatetimeLikeArray instance to a text file.

Arguments:

- **fp (Union[IO, str, Path]):** File path or file-like object to write to.
- **tz (datetime.tzinfo, optional):** Timezone in which to write the data.

```
@classmethod
def from_array( cls, input_array: numpy.ndarray[typing.Any, numpy.dtype[typing.Union[numpy.number,
numpy.datetime64]]], tz: Optional[datetime.tzinfo] = None):
```

Convert a numpy array to a DatetimeLikeArray instance.

Arguments:

- **input_array (np.ndarray):** NumPy array containing datetime values.
- **tz (datetime.tzinfo, optional):** Timezone of the input datetime values.

Returns:

> DatetimeLikeArray: A new instance with timezone awareness.

```
@classmethod
def from_fp( cls, fp: Union[IO, str, pathlib.Path], dtype: Type, tz: Optional[datetime.tzinfo] = None):
```

Load a text file and convert it to a DatetimeLikeArray instance.

Arguments:

- **fp (Union[IO, str, Path]):** File path or file-like object to read from.
- **dtype (Type):** Data type of the values in the file.
- **tz (datetime.tzinfo, optional):** Timezone to assign to the loaded data.

Returns:

> DatetimeLikeArray: A new instance with timezone awareness.

```
@classmethod
def from_iter( cls, gen: Generator[~DatetimeLike, NoneType, NoneType], dtype: Type, tz:
Optional[datetime.tzinfo] = None):
```

Create a DatetimeLikeArray object from an Iterable with DatetimeLike yields

Arguments:

- **gen (Generator[DatetimeLike, None, None]):** A generator that yields DatetimeLike values
- **dtype (Type):** Data type of the values in the file.
- **tz (datetime.tzinfo, optional):** Timezone to assign to the loaded data.

Returns:

DatetimeLikeArray: A new instance with timezone awareness.

aqua_blue.models

Module defining models, i.e., compositions of reservoir(s) and readout layers.

This module implements the `Model` class, which integrates a reservoir and a readout layer to process time series data. The model enables training using input time series data and forecasting future values based on learned patterns.

Classes:

- Model: Represents an Echo State Network (ESN)-based model that learns from input time series data and makes future predictions.

```
logger = <Logger aqua_blue.models (WARNING)>
@dataclass
class Model(typing.Generic[~DatetimeLike, ~TimeDeltaLike]):
```

A machine learning model that integrates a reservoir with a readout layer for time series forecasting.

This class implements an Echo State Network (ESN) approach, where the reservoir serves as a high-dimensional dynamic system, and the readout layer maps reservoir states to output values.

Attributes:

- **reservoir (Reservoir):** The reservoir component, defining the input-to-reservoir mapping.
- **readout (Readout):** The readout layer, mapping reservoir states to output values.
- **final_time (float):** The last timestamp seen during training. This is set automatically after training.
- **timestep (float):** The fixed time interval between consecutive steps in the input time series, set during training.
- **initial_guess (np.ndarray):** The last observed state of the system during training, used as an initial condition for predictions.
- **tz (Union[datetime.tzinfo, None]):** The timezone associated with the time series. Set to `None` if the `DatetimeLikeArray` is incompatible.

```
Model( reservoir: aqua_blue.reservoirs.Reservoir, readout: aqua_blue.readouts.Readout)
reservoir: aqua_blue.reservoirs.Reservoir
```

The reservoir component that defines the input-to-reservoir mapping.

readout: aqua_blue.readouts.Readout

The readout component that defines the reservoir-to-output mapping.

final_time: ~DatetimeLike

The final timestamp encountered in the training dataset (set during training).

timestep: ~TimeDeltaLike

The fixed time step interval of the training dataset (set during training).

initial_guess: numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]]

The last observed state of the system, used for future predictions (set during training).

tz: Optional[datetime.tzinfo]

The timezone associated with the independent variable. Set to `None` if unsupported.

def train(self, input_time_series: aqua_blue.time_series.TimeSeries, warmup: int = 0):

Trains the model on the provided time series data.

This method fits the readout layer using reservoir states obtained from the input time series data. A warmup period can be specified to exclude initial steps from training.

Arguments:

- **input_time_series (TimeSeries):** The time series instance used for training.
- **warmup (int):** The number of initial steps to ignore in training (default: 0).

Raises:

- **ValueError:** If `warmup` is greater than or equal to the number of timesteps in the input time series.

def predict(self, horizon: int) -> aqua_blue.time_series.TimeSeries:

Generates future predictions for a specified time horizon.

This method uses the trained model to generate future values based on the learned dynamics of the input time series.

Arguments:

- **horizon (int):** The number of steps to forecast into the future.

Returns:

TimeSeries: A `TimeSeries` instance containing the predicted values and corresponding timestamps.

aqua_blue.readouts

Module defining readout layers.

This module provides the abstract `Readout` class and its concrete implementation, `LinearReadout`. Readout layers map the internal reservoir states of an Echo State Network (ESN) to output values.

Classes:

- Readout: Abstract base class defining the interface for readout layers.
- LinearReadout: A linear mapping readout layer that transforms reservoir states into output values using learned coefficients.

```
logger = <Logger aqua_blue.readouts (WARNING)>
@dataclass
class Readout(abc.ABC):
```

Abstract base class for readout layers in Echo State Networks (ESNs).

Readout layers transform the high-dimensional reservoir states into output predictions. The transformation is typically learned during training.

Attributes:

- **coefficients (np.ndarray):** The learned weights for mapping reservoir states to output values. This is set after training.

```
coefficients: numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]]
```

The learned weight matrix for the readout layer, initialized during training.

```
@abstractmethod
def train( self, independent_variables: numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]],
dependent_variables: numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]]):
```

Trains the readout layer by learning the mapping from reservoir states to output values.

This method takes independent input variables (reservoir states) and corresponding dependent variables (target outputs) to compute the optimal readout weights.

Arguments:

- **independent_variables (np.ndarray):** The reservoir state matrix used as input for training.
- **dependent_variables (np.ndarray):** The expected output values corresponding to the input states.

```
@abstractmethod
def reservoir_to_output( self, reservoir_state: numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]]) -> numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]]:
```

Maps a given reservoir state to an output value.

Arguments:

- **reservoir_state (np.ndarray):** The current state of the reservoir.

Returns:

> np.ndarray: The predicted output corresponding to the given reservoir state.

```
@dataclass
class LinearReadout(Readout):
```

A linear readout layer that applies a learned linear transformation to reservoir states.

This readout layer learns a set of coefficients during training and applies a simple linear mapping to transform reservoir states into output predictions.

Attributes:

- **rcond (float):** A regularization parameter used in the pseudo-inverse calculation to prevent numerical instability in the least squares solution.

```
LinearReadout(rcond: float = 1e-10)
rcond: float = 1e-10
```

Regularization parameter for pseudo-inverse computation.

This controls the minimum singular value considered for the pseudo-inverse computation. A lower value ensures more stable training.

```
def train( self, independent_variables: numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]], dependent_variables: numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]]):
```

Trains the linear readout layer by solving the least-squares optimization problem.

The training process determines the optimal readout coefficients W^* by solving the optimization problem below:

$$W^* = \lim_{\lambda \to 0^+} \arg\min_W \|XW - Y\|_F^2 + \lambda \|W\|_F^2$$

where X is the matrix of reservoir states (independent variables) and Y is the matrix of target output values (dependent variables).

Arguments:

- **independent_variables (np.ndarray):** The reservoir state matrix used for training.
- **dependent_variables (np.ndarray):** The target output values corresponding to the reservoir states.

```
def reservoir_to_output( self, reservoir_state: numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]]) -> numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]]:
```

Computes the output from a given reservoir state using a learned linear mapping.

This method applies the learned weight matrix (`self.coefficients`) to map the reservoir state to an output value.

Arguments:

- **reservoir_state (np.ndarray):** The reservoir state to be mapped to an output value.

Returns:

> np.ndarray: The predicted output value.

Raises:

- **ValueError:** If the readout layer has not been trained (i.e., coefficients are not set).

Inherited Members

Readout
    coefficients

aqua_blue.reservoirs

Module defining reservoirs.

This module contains the base `Reservoir` class and its concrete implementation, `DynamicalReservoir`. Reservoirs serve as dynamic memory structures in Echo State Networks (ESNs) by transforming input signals into high-dimensional representations.

Classes:

- Reservoir: Abstract base class defining the structure of a reservoir.
- DynamicalReservoir: A specific implementation of a reservoir with tunable dynamics and activation functions.

```
ActivationFunction = typing.Callable[[numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]]], numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]]]
```

activation function, taking in a numpy array and returning a numpy array of the same shape

```
logger = <Logger aqua_blue.reservoirs (WARNING)>
@dataclass
class Reservoir(abc.ABC):
```

Abstract base class defining a reservoir in an Echo State Network (ESN).

Reservoirs are responsible for transforming input signals into high-dimensional representations, which are then used by the readout layer for predictions.

Attributes:

- **input_dimensionality (int):** The number of input features.
- **reservoir_dimensionality (int):** The number of reservoir neurons (i.e., the size of the reservoir).
- **res_state (np.ndarray):** The current state of the reservoir, which is updated at each time step.

```
input_dimensionality: int
```

Dimensionality of the input state.

```
reservoir_dimensionality: int
```

Dimensionality of the reservoir state, equivalently the reservoir size.

```
res_state: numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]]
```

Reservoir state, necessary property when performing training loop.

```
@abstractmethod
def update_reservoir( self, input_state: numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]]) -> numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]]:
```

Updates the reservoir state given an input state.

This method defines the transformation applied to an input vector when passed through the reservoir.

```
@dataclass
class DynamicalReservoir(Reservoir):
```

A dynamical reservoir with tunable properties.

This reservoir is defined by the equation:

$$y_t = (1 - \alpha) y_{t-1} + \alpha f(W_{\text{in}} x_t + W_{\text{res}} y_{t-1})$$

where x_t is the input at time step t, y_t is the reservoir state at time t, W_{in} is the input weight matrix, W_{res} is the reservoir weight matrix, α (leaking_rate) controls how much of the previous state influences the next state, and f is a nonlinear activation function.

Attributes:

- **generator (Optional[np.random.Generator]):** Random number generator for weight initialization.
- **w_in (Optional[np.ndarray]):** Input weight matrix of shape `(reservoir_dimensionality, input_dimensionality)`. Auto-generated if not provided.
- **w_res (Optional[np.ndarray]):** Reservoir weight matrix of shape `(reservoir_dimensionality, reservoir_dimensionality)`. Auto-generated if not provided.
- **activation_function (ActivationFunction):** Activation function applied to the reservoir state. Defaults to `np.tanh`.
- **leaking_rate (float):** Leaking rate that controls the contribution of the previous state.

```
DynamicalReservoir( input_dimensionality: int, reservoir_dimensionality: int, generator: Optional[numpy.random._generator.Generator] = None, w_in: Optional[numpy.ndarray[Any, numpy.dtype[numpy.floating]]] = None, w_res: Optional[numpy.ndarray[Any, numpy.dtype[numpy.floating]]] = None, activation_function: Callable[[numpy.ndarray[Any, numpy.dtype[numpy.floating]]], numpy.ndarray[Any, numpy.dtype[numpy.floating]]] = <ufunc 'tanh'>, leaking_rate: float = 1.0, sparsity: Optional[float] = None, spectral_radius: Optional[float] = 0.95)
generator: Optional[numpy.random._generator.Generator] = None
```

Random generator for initializing weights. Defaults to `np.random.default_rng(seed=0)` if not specified.

```
w_in: Optional[numpy.ndarray[Any, numpy.dtype[numpy.floating]]] = None
```

Input weight matrix. Must have shape `(reservoir_dimensionality, input_dimensionality)`. If not provided, it is auto-generated with values in `[-0.5, 0.5]`.

w_res: Optional[numpy.ndarray[Any, numpy.dtype[numpy.floating]]] = None

Reservoir weight matrix. Must have shape `(reservoir_dimensionality, reservoir_dimensionality)`. If not provided, it is auto-generated and normalized to have a spectral radius of 0.95.

activation_function: Callable[[numpy.ndarray[Any, numpy.dtype[numpy.floating]]], numpy.ndarray[Any, numpy.dtype[numpy.floating]]] = <ufunc 'tanh'>

Nonlinear activation function applied to the reservoir state. Defaults to `np.tanh`, but can be replaced with other functions like ReLU.

leaking_rate: float = 1.0

Leaking rate ((\alpha)) that controls how much of the previous state contributes to the next. Defaults to `1.0`, meaning the state is fully updated at each time step.

sparsity: Optional[float] = None

sparsity of the reservoir weight matrix. (0, 1]

spectral_radius: Optional[float] = 0.95

spectral radius of reservoir weight matrix. Recommended values - [0.9, 1.2]

def update_reservoir(self, input_state: numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]]) -> numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]]:

Updates the reservoir state given an input.

This method applies the state update equation:

$$y_t = (1 - \alpha) y_{t-1} + \alpha f(W_{\text{in}} x_t + W_{\text{res}} y_{t-1})$$

Arguments:

- **input_state (np.ndarray):** The input state vector.

Returns:

> np.ndarray: The updated reservoir state.

Inherited Members

Reservoir
- input_dimensionality
- reservoir_dimensionality
- res_state

aqua_blue.time_series

Module defining the TimeSeries object

```
logger = <Logger aqua_blue.time_series (WARNING)>
def parse_time(s: str):
class ShapeChangedWarning(builtins.Warning):
```

Warning for cases where `TimeSeries.__post_init__` alters the shape of the dependent variable.

```
class TimeSeriesTypedDict(typing.TypedDict):
```

TypedDict form of a TimeSeries object, useful for turning into json

```
dependent_variable: Sequence[Sequence[float]]
times: Sequence[Union[float, str]]
@dataclass
class TimeSeries(typing.Generic[~TimeDeltaLike]):
```

A class representing a time series, encapsulating dependent variables and corresponding timestamps.

```
TimeSeries( dependent_variable: numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]], times:
aqua_blue.datetimelikearray.DatetimeLikeArray)
dependent_variable: numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]]
```

Array of dependent variables representing the time series values.

```
times: aqua_blue.datetimelikearray.DatetimeLikeArray
```

Array of time values associated with the dependent variable.

```
def save( self, fp: Union[IO, str, pathlib.Path], header: str = '', delimiter=',', fmt: Union[str, Tuple[str]] = '%s'):
```

Saves the time series data to a file.

Arguments:

- **fp (Union[IO, str, Path]):** File path or object where the TimeSeries instance will be saved.
- **header (str, optional):** An optional header. Defaults to an empty string.
- **delimiter (str, optional):** The delimiter used in the output file. Defaults to a comma.
- **fmt (Tuple(str), optional):** Format specifier used for saving the data to fp. Defaults to '%s' (String representation)

```
num_dims: int
```

Returns the dimensionality of the dependent variable.

Returns:

int: Number of dimensions of the time series.

```
@classmethod
def from_csv( cls, fp: Union[IO, str, pathlib.Path], time_col: str, times_dtype: Type, dependent_var_cols: List[str], times_conversion: Callable[[str], ~DatetimeLike] = <function parse_time>, dep_var_conversion: Callable[[str], float] = <class 'float'>, max_rows: Optional[int] = 0):
```

Loads time series data from a CSV file.

Arguments:

- **fp (Union[IO, str, Path]):** File path or object to read from.
- **time_col (str):** Name of the times column.
- **times_dtype (dtype):** Type of the times column
- **dependent_var_cols (List[str]):** Names of the dependent variable columns
- **times_conversion (Callable[[str], DatetimeLike]):** Function determining how to parse elements of the times column. Defaults to parse_time
- **dep_var_conversion (Callable[[str], float]):** Function determining how to parse elements of the dependent variable columns. Defaults to float
- **max_rows (float):** Maximum number of rows that should be parsed. If set to zero, all rows are parsed. Defaults to 0

Returns:

> TimeSeries: A TimeSeries instance populated by data from the csv file.

```
def to_dict(self) -> TimeSeriesTypedDict:
```

convert to a typed dictionary

```
timestep: ~TimeDeltaLike
```

Returns the time step between consecutive observations.

Returns:

> TimeDeltaLike: The timestep of the time series.

aqua_blue.utilities

This module provides simple utilities for processing TimeSeries instances.

```
@dataclass
class Normalizer:
```

A utility class for normalizing and denormalizing TimeSeries instances.

This class computes and stores the mean and standard deviation of the dependent variable during normalization. These statistics are later used to restore the original scale of the data when denormalizing.

means: numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]]

Mean values of the dependent variable, computed during normalization.

standard_deviations: numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]]

Standard deviation values of the dependent variable, computed during normalization.

def normalize(self, time_series: aqua_blue.time_series.TimeSeries) -> aqua_blue.time_series.TimeSeries:

Normalize a TimeSeries instance by adjusting its values to have zero mean and unit variance.

Arguments:

- **time_series (TimeSeries):** The time series to be normalized.

Returns:

> TimeSeries: A new TimeSeries instance with normalized values.

Raises:

- **ValueError:** If the normalizer has already been used, since it is intended for one-time use.

def denormalize(self, time_series: aqua_blue.time_series.TimeSeries) -> aqua_blue.time_series.TimeSeries:

Denormalize a previously normalized TimeSeries instance, restoring it to its original scale.

Arguments:

- **time_series (TimeSeries):** Time series to denormalize

Returns:

> TimeSeries: The denormalized time series

Raises:

- **ValueError:** If normalization has not been performed before calling this method.

def make_sparse(weight_matrix: numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]], sparsity: float, generator: Optional[numpy.random._generator.Generator] = None) -> numpy.ndarray[typing.Any,

```
numpy.dtype[numpy.floating]]:
```

Make a weight matrix sparse

Arguments:

- **weight_matrix (np.typing.NDArray[np.floating]):** Weight matrix to be made sparse
- **sparsity (float):** Extent of how sparse to make the weight matrix. Ranges from 0 to 1.
- **generator (np.random.Generator):** NumPy Generator to create random numbers

```
def set_spectral( weight_matrix: numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]], spectral_radius:
float) -> numpy.ndarray[typing.Any, numpy.dtype[numpy.floating]]:
```

Set the spectral radius of the weight matrix

Arguments:

- **weight_matrix (np.typing.NDArray[np.floating]):** Weight matrix whose spectral radius is to be set
- **spectral_radius (float):** The largest absolute singular value of the weight matrix. Values less than 1.0 are recommended for tasks that require significant memory fading. Values between 1-1.5 are recommended for tasks that are memory dependent.