Replicated Focusing Belief Propagation algorithm

Release 1.0.4

Oct 21, 2020

Contents:

The **Replicated Focusing Belief Propagation** package is inspired by the original BinaryCommitteeMachineFBP verion written in Julia. In our implementation we optimize and extend the original library inclu multi-threading support and an easy-to-use interface to the main algorithm. To further improve the usage of our code, we propose also a *Python* wrap of the library with a full compatibility with the scikit-learn and scikit-optimize packages.

CHAPTER 1

Overview

The learning problem could be faced through statistical mechanic models joined with the so-called Large Deviation Theory. In general, the learning problem can be split into two sub-parts: the classification problem and the generalization one. The first aims to completely store a pattern sample, i.e a prior known ensemble of input-output associations (*perfect learning*). The second one corresponds to compute a discriminant function based on a set of features of the input which guarantees a unique association of a pattern.

From a statistical point-of-view many Neural Network models have been proposed and the most promising seems to be spin-glass models based. Starting from a balanced distribution of the system, generally based on Boltzmann distribution, and under proper conditions, we can prove that the classification problem becomes a NP-complete computational problem. A wide range of heuristic solutions to that type of problems were proposed.

In this project we show one of these algorithms developed by Zecchina et al and called *Replicated Focusing Belief Propagation* (*rFBP*). The *rFBP* algorithm is a learning algorithm developed to justify the learning process of a binary neural network framework. The model is based on a spin-glass distribution of neurons put on a fully connected neural network architecture. In this way each neuron is identified by a spin and so only binary weights (-1 and 1) can be assumed by each entry. The learning rule which controls the weight updates is given by the Belief Propagation method.

A first implementation of the algorithm was proposed in the original paper (Zecchina et al) jointly with an open-source Github repository. The original version of the code was written in *Julia* language and despite it is a quite efficient implementation the *Julia* programming language stays on difficult and far from many users. To broaden the scope and use of the method, a *C++* implementation was developed with a joint *Cython* wrap for *Python* users. The *C++* language guarantees better computational performances against the *Julia* implementation and the *Python* version enhances its usability. This implementation is optimized for parallel computing and is endowed with a custom *C++* library called scorer, which is able to compute a large number of statistical measurements based on a hierarchical graph scheme. With this optimized implementation and its scikit-learn compatibility we try to encourage researchers to approach these alternative algorithms and to use them more frequently on real context.

As the *Julia* implementation also the *C++* one provides the entire *rFBP* framework in a single library callable via a command line interface. The library widely uses template syntaxes to perform dynamic specialization of the methods between two magnetization versions of the algorithm. The main object categories needed by the algorithm are wrapped in handy *C++* objects easy to use also from the *Python* interface.

1.1 Usage example

The *rfbp* object is totally equivalent to a *scikit-learn* classifier and thus it provides the member functions *fit* (to train your model) and *predict* (to test a trained model on new samples).

```
import numpy as np
from sklearn.model_selection import train_test_split
from ReplicatedFocusingBeliefPropagation import MagT64
from ReplicatedFocusingBeliefPropagation import Pattern
from ReplicatedFocusingBeliefPropagation import ReplicatedFocusingBeliefPropagation␣
↪as rFBP

N, M = (20, 101) # M must be odd
X = np.random.choice([-1, 1], p=[.5, .5], size=(N, M))
y = np.random.choice([-1, 1], p=[.5, .5], size=(N, ))

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↪state=42)

rfbp = rFBP(mag=MagT64,
            hidden=3,
            max_iter=1000,
            seed=135,
            damping=0.5,
            accuracy=('accurate','exact'),
            randfact=0.1,
            epsil=0.5,
            protocol='pseudo_reinforcement',
            size=101,
            nth=1)

rfbp.fit(X_train, y=y_train)
y_pred = rfbp.predict(X_test)
```

The same code could be easily translated also in a pure C++ application as

```
#include <rfbp.hpp>

int main ()
{
  const int N = 20;
  const int M = 101; // M must be odd
  const int K = 3;

  FocusingProtocol fp("pseudo_reinforcement", M);
  Patterns patterns(N, M);

  long int ** bin_weights = focusingBP < MagP64 >(K,          // hidden,
                                                  patterns,   // patterns,
                                                  1000,       // max_iters,
                                                  101,        // max_steps,
                                                  42,         // seed,
                                                  0.5,        // damping,
                                                  "accurate", // accuracy1,
                                                  "exact",    // accuracy2,
                                                  0.1,        // randfact,
                                                  fp,         // fp,
```

(continues on next page)

(continued from previous page)

```
                                                    0.1,        // epsil,
                                                    1,          // nth,
                                                    "",         // outfile,
                                                    "",         // outmess,
                                                    "",         // inmess,
                                                    false       // binmess
                                                    );

  // It is clearly an overfitting! But it works as example
  long int ** y_pred = nonbayes_test(bin_weights, patterns, K);

  return 0;
}
```

1.1.1 Theory

The *rFBP* algorithm derives from an out-of-equilibrium (non-Boltzmann) model of the learning process of binary neural networks DallAsta101103. This model mimics a spin glass system whose realizations are equally likely to occur when sharing the same so-called entropy (not the same energy, i.e. out-of-equilibrium). This entropy basically counts the number of solutions (zero-energy realizations) around a realization below a fixed-distance.

Within this out-of-equilibrium framework, the objective is to maximize the entropy instead of minimizing the energy. From a machine learning standpoint, we aim at those weights sets that perfectly solve the learning process (zero-errors) and that are mathematically closed to each other. To this end, the Belief Propagation method MézardMontanari can be adopted as the underlying learning rule, although it must be properly adjusted to take into account the out-of-equilibrium nature of the model.

The *Replicated Focusing Belief Propagation* (*rFBP*) is an entropy-maximization based algorithm operating as the underlying learning rule of feed-forward binary neural networks. Here, the entropy is defined as:

$$S(\vec{w},\beta,\gamma) = \frac{1}{N} log\left(\sum_{\vec{w}'} e^{-\beta E(\vec{w}')} e^{\gamma \vec{w}' \cdot \vec{w}} \right)$$

where $\vec{w}$ is the whole weights set, N is its size and β is the standard Boltzmann term. Further, $E(\vec{w})$ is the energy of such weights set, which is equal to the number of wrong predictions on the training set produced by $\vec{w}$. When $\beta \rightarrow inf$, only those $\vec{w}$ with null energy, i.e. perfect solutions, sum up in the entropy. At the time being, the rFBP only works with $\beta \rightarrow inf$.

The realization of the rFBP is equivalent to evolve a spin-glass system composed of several interacting replicas with the Belief Propagation algorithm. The density distribution of the system is modelled by:

$$P(\vec{w}|\beta,y,\gamma) = \frac{e^{yNS(\vec{w},\beta,\gamma)}}{Z(\beta,y,\gamma)}$$

where Z is the partition function.

Such spin-glass model depends necessarily on two parameters: y and γ. The former is a temperature-alike related variable, similar to the one usually exploited by Gradient Descend approaches, but it can be also interpreted as the number of interacting replicas of the system. The latter is the penalization term associated to the distance between two weights sets. Indeed, the term $e^{\gamma \vec{w}' \cdot \vec{w}}$ in the entropy is larger, when $\vec{w}$ and $\vec{w}$ are closer.

The Belief Propagation algorithm needs to be to adjusted by adding incoming extra messages for all weights, in order to involve the interacting replicas of the system. This extra term is represented by:

$$\hat{m}^{t_1}_{\star \rightarrow w_i} = tanh\big[(y-1) artanh(m^{t_0}_{w_i \rightarrow \star} tanh\gamma)\big] tanh\gamma$$

where w_i and $\star$ stand respectively for the i-th weight and a representation of all i-th replicas.

The *rFBP* is therefore an adjusted Belief Propagation algorithm, whose general procedure can be summarized as follows:

- set $\beta \to inf$
- select protocols for y and γ
- set first values of y and γ and run the adjusted-BP method until convergence (ϵ or up to a limited-number of iterations;
- step to the next pair values of y and γ with respect to the chosen protocols and re-run the adjusted-BP method;
- keep it going until a solution is reached or protocols end.

The *rFBP* algorithm focuses the replicated system to fall step by step into weights sets extremely closed to many perfect solutions ($\vec{w}$ such that $E(\vec{w}) = 0$), which ables them to well generalize out of the training set Zecchina et al.

1.1.2 Installation guide

C++ supported compilers:

The *rFBP* project is written in *C++* using a large amount of c++17 features. To enlarge the usability of our package we provide also a retro-compatibility of all the c++17 modules reaching an usability (tested) of our code from gcc 4.8.5+. The package installation can be performed via *CMake* or *Makefile*.

If you are using the *CMake* (recommended) installer the maximum version of C++ standard is automatic detected. The *CMake* installer provides also the export of the library: after the installation you can use this library into other *CMake* projects using a simple *find_package* function. The exported *CMake* library (*rFBP::rfbp*) is installed in the *share/rFBP* directory of the current project and the relative header files are available in the *rFBP_INCLUDE_DIR* variable.

The *CMake* installer provides also a *rFBP.pc*, useful if you want link to the *rFBP* using *pkg-config*.

You can also use the *rFBP* package in *Python* using the *Cython* wrap provided inside this project. The only requirements are the following:

- numpy >= 1.15
- cython >= 0.29
- scipy >= 1.2.1
- scikit-learn >= 0.20.3
- requests >= 2.22.0

The *Cython* version can be built and installed via *CMake* enabling the *-DPYWRAP* variable. The *Python* wrap guarantees also a good integration with the other common Machine Learning tools provided by *scikit-learn Python* package; in this way you can use the *rFBP* algorithm as an equivalent alternative also in other pipelines. Like other Machine Learning algorithm also the *rFBP* one depends on many parameters, i.e its hyper-parameters, which has to be tuned according to the given problem. The *Python* wrap of the library was written according to *scikit-optimize Python* package to allow an easy hyper-parameters optimization using the already implemented classical methods.

CMake C++ Installation

We recommend to use *CMake* for the installation since it is the most automated way to reach your needs. First of all make sure you have a sufficient version of *CMake* installed (3.9 minimum version required). If you are working on a machine without root privileges and you need to upgrade your *CMake* version a valid solution to overcome your problems is provided shut.

With a valid *CMake* version installed first of all clone the project as:

```
git clone https://github.com/Nico-Curti/rFBP
cd rFBP
```

The you can build the *rFBP* package with

```
mkdir -p build
cd build && cmake .. && cmake --build . --target install
```

or more easily

```
./build.sh
```

if you are working on a Windows machine the right script to call is the build.ps1.

Note: If you want enable the OpenMP support (*4.5 version is required*) compile the library with *-DOMP=ON*.

Note: If you want enable the Scorer support compile the library with *-DSCORER=ON*. If you want use a particular installation of the Scorer library or you have manually installed the library following the *README* instructions, we suggest to add the *-DScorer_DIR=/path/to/scorer/shared/scorer* in the command line.

Note: If you want enable the Cython support compile the library with *-DPYWRAP=ON*. The Cython packages will be compiled and correctly positioned in the *rFBP* Python package **BUT** you need to run also the setup before use it.

Note: If you use MagT configuration, please download the *atanherf coefficients* file before running any executable. You can find a downloader script inside the scripts folder. Enter in that folder and just run *python dowload_atanherf.py*.

Python Installation

Python version supported :

The easiest way to install the package is using *pip*

```
python -m pip install ReplicatedFocusingBeliefPropagation
```

Warning: The setup file requires the *Cython* and *Numpy* packages, thus make sure to pre-install them! We are working on some workarounds to solve this issue.

The *Python* installation can be performed with or without the *C++* installation. The *Python* installation is always executed using setup.py script.

If you have already built the *rFBP C++* library the installation is performed faster and the *Cython* wrap was already built using the *-DPYWRAP* definition. Otherwise the full list of dependencies is build.

In both cases the installation steps are

```
python -m pip install -r ./requirements.txt
```

to install the prerequisites and then

```
python setup.py install
```

or for installing in development mode:

```
python setup.py develop --user
```

Warning: The current installation via pip has no requirements about the version of *setuptools* package. If the already installed version of *setuptools* is >= *50.** you can find some troubles during the installation of our package (ref. issue). We suggest to temporary downgrade the *setuptools* version to *49.3.0* to workaround this *setuptools* issue.

1.1.3 C++ API

FocusingProtocol

class FocusingProtocol

Abstract type representing a protocol for the focusing procedure, i.e. a way to produce successive values for the quantities γ, n_rep and β. Currently, however, only β=`Inf` is supported. To be provided as an argument to `focusingBP`.

Available protocols are: `StandardReinforcement`, `Scoping`, `PseudoReinforcement` and `FreeScoping`

Public Functions

FocusingProtocol()

Default constructor.

FocusingProtocol(**const** std::string &*prot*, **const** long int &*size*)

Constructor with protocol type and number of replicas.

Protocol types are set with default values. If you want introduce other values you must use appropriated protocol functions

Parameters

- `prot`: protocol type. Available protocols are: `StandardReinforcement`, `Scoping`, `PseudoReinforcement` and `FreeScoping`. This value is used to switch between the available protocols and the corresponding arrays are stored.
- `size`: number of step. Converted to Nrep into the class

~FocusingProtocol()

Destructor set as default.

void **StandardReinforcement**(**const** double **rho*, **const** long int &*Nrho*)

Standard reinforcement protocol, returns γ=`Inf` and `n_rep=1/(1-x)`, where `x` is taken from the given range ρ.

Parameters

- `rho`: double pointer which store the range values of x
- `Nrho`: number of step. Converted to Nrep into the class

void **StandardReinforcement**(**const** double &*drho*)

Shorthand for Standard reinforcement protocol.

Parameters

- `drho`: double related to the range increment

void **Scoping**(**const** double **gr*, **const** double &*x*, **const** long int &*ngr*)

Focusing protocol with fixed `n_rep` and a varying γ taken from the given γ `* r` range.

Parameters

- `gr`: double pointer with γ * r values
- `x`: fixed value of n_rep
- `ngr`: number of replicas

void **PseudoReinforcement**(**const** double **rho*, **const** long int &*nrho*, double *x* = .5)

A focusing protocol in which both γ and `n_rep` are progressively increased, according to the formulas.

```
γ = atanh(ρ**x)
n_rep = 1 + ρ**(1 - 2x) / (1 - ρ)
```

where ρ is taken from the given range(ngr) `r`. With `x=0`, this is basically the same as `StandardReinforcement`.

Parameters

- `rho`: double pointer with ρ values
- `nrho`: lenght of rho array
- `x`: fixed value of n_rep

void **PseudoReinforcement**(**const** double &*drho*, double *x* = .5)

Shorthand for Pseudo Reinforcement protocol.

Parameters

- `drho`: double related to the range increment
- `x`: fixed value of n_rep

void **FreeScoping**(double ***list*, **const** long int &*nlist*)

A focusing protocol which just returns the values of `(`γ`, n_rep)` from the given `list`.

Parameters

- `list`: array of lists (nlist, 3) with values
- `nlist`: lenght of list

Public Members

long int **`Nrep`**
Number of repetitions, i.e. number of focusing iterations.

std::unique_ptr<double[]> **`gamma`**
Distance parameters.

std::unique_ptr<double[]> **`n_rep`**
Number of replicas (y in the paper and original code)

std::unique_ptr<double[]> **`beta`**
1/kT (it must be infinite in the current implementation)

Mag type

MagP64

`class MagP64`
Abstract type representing magnetization type chosen for cavity messages.

Note The *`MagP64`* type allows fast executions with inexact outcomes by neglecting all `tanh` operations.

Public Functions

`MagP64``()`
Default constructor.

`MagP64``(`**`const`** double &*x*`)`
Constructor with value.

Note In *MagP64* the value is equal to the mag.

Parameters

- `x`: magnetization

`~MagP64``()`
Default destructor.

`MagP64``(`**`const`** *MagP64* &*m*`)`
Copy constructor.

Parameters

- `m`: *MagP64* object

MagP64 &**`operator=`**`(`**`const`** *MagP64* &*m*`)`
Assigment operator constructor.

Parameters

- m: *MagP64* object

std::string **magformat**() **const**
Return the mag description (plain for *MagP64*).

double **value**() **const**
Get the magnetization value.

Note In *MagP64* the value is equal to the mag.

MagP64 **operator%**(**const** *MagP64* &*m*)
Overload operator.

Add magnetization ((m1 + m2) / (1 + m1*m2)) with clamp.

See *mag* :: clamp

Parameters

- m: *MagP64* object

MagP64 **operator+**(**const** *MagP64* &*m*)
Overload operator.

Just a simple addition of the mag values.

Parameters

- m: *MagP64* object

MagP64 **operator/**(**const** double &*x*)
Overload operator.

Just a simple division as (mag / x)

Parameters

- x: double value

double **operator***(**const** double &*x*)
Overload operator.

Just a simple product as (mag * x)

Parameters

- x: double value

MagP64 **operator^**(**const** *MagP64* &*m*)
Overload operator.

Combine two mags as (mag * mag)

Parameters

- m: *MagP64* object

double **operator-**(**const** *MagP64* &*m*)
Overload operator.

Subtract values (val1 - val2)

Note In *MagP64* the values are equal to the mags.

Parameters

- m: *MagP64* object

MagP64 **operator-**() **const**
Get a magnetization with a flipped sign.

flip magnetization sign

bool **operator==**(**const** *MagP64* &*m*)
Check magnetization equality.

Compare magnetizations

Parameters

- m: *MagP64* object

bool **operator!=**(**const** *MagP64* &*m*)
Check magnetization not equality.

compare magnetizations

Parameters

- m: *MagP64* object

Public Members

double **mag**
Magnetization.

Friends

double **operator***(**const** double &*x*, **const** *MagP64* &*m*)
Overload operator.

Just a simple product as (x * mag)

Parameters

- x: double value
- m: *MagP64* object

std::ostream &**operator<<**(std::ostream &*os*, **const** *MagP64* &*m*)
Print operator with stdout/stderr.

print mag

Parameters

- `os`: ostream operator
- `m`: *MagP64* object

MagT64

class MagT64

Abstract type representing magnetization type chosen for cavity messages.

Note *MagT64* means a double type wit `tanh` application.

Public Functions

MagT64()

Default constructor.

MagT64(**const** double &*x*, double *m* = 30.0)

Constructor with value.

Note In *MagT64* the magnetization is converted to a `value` given by `tanh(x)`.

Parameters

- `x`: magnetization
- `m`: boundary value

~MagT64()

Default destructor.

MagT64(**const** *MagT64* &*m*)

Copy constructor.

Parameters

- `m`: *MagT64* object

MagT64 &**operator=**(**const** *MagT64* &*m*)

Assigment operator constructor.

Parameters

- `m`: *MagT64* object

std::string **magformat**() **const**

Return the mag description (tanh for *MagT64*).

double **value**() **const**

Get the magnetization value.

Note In *MagT64* the value is given by `tanh(mag)`.

MagT64 **operator%**(**const** *MagT64* &*m*)

Overload operator.

Add magnetization (m1 + m2)

Note The summation exclude the `tanh` evaluation.

Parameters

- `m`: *MagT64* object

MagT64 **operator+**(**const** *MagT64* &*m*)

Overload operator.

Just a simple addition of the mag values.

Parameters

- `m`: *MagT64* object

MagT64 **operator/**(**const** double &*x*)

Overload operator.

Just a simple division as (mag / x)

Parameters

- `x`: double value

MagT64 **operator^**(**const** *MagT64* &*m*)

Overload operator.

Combine two mags.

Parameters

- `m`: *MagT64* object

double **operator-**(**const** *MagT64* &*m*)

Overload operator.

Subtract values (val1 - val2)

Parameters

- `m`: *MagT64* object

MagT64 **operator-**() **const**

Get a magnetization with a flipped sign.

flip magnetization sign

bool **operator==**(**const** *MagT64* &*m*)

Check magnetization equality.

Compare magnetizations

Parameters

- `m`: *MagT64* object

bool **operator!=**(**const** *MagT64* &*m*)

Check magnetization not equality.

compare magnetizations

Parameters

- `m`: *MagT64* object

Public Members

double **mag**
Magnetization.

double **mInf**
Boundary dimension.

Friends

double **operator*** (**const** double &*x*, **const** *MagT64* &*m*)
Overload operator.

Just a simple product as (x * mag)

Parameters

- `x`: double value
- `m`: *MagT64* object

std::ostream &**operator<<** (std::ostream &*os*, **const** *MagT64* &*m*)
Print operator with stdout/stderr.

print mag

Parameters

- `os`: ostream operator
- `m`: *MagT64* object

Magnetization functions

namespace mag

Functions

double **clamp** (**const** double &*x*, **const** double &*low*, **const** double &*high*)
Clamp value between boundaries.

Return value clamped between boudaries

Parameters

- `x`: double value as argument of clamp
- `low`: lower boundary
- `high`: higher boundary

double **lr**(**const** double &*x*)

log1p for magnetizations.

Return value computed as log1p(exp(-2*abs(x)))

Parameters

- x: double value

long int **sign0**(**const** double &*x*)

Sign operation valid also for magnetizations.

Return sign evaluated as 1 - 2*signbit(x)

Parameters

- x: double value

template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :

Check if is infinite.

Return result of the check

Note The function behavior is different between *MagP64* and *MagT64*.

- *MagP64*: boolean true if is inf or -inf else false
- *MagT64*: boolean true if is nan or -nan else false

Template Parameters

- Mag: magnetization type (MagP or MagT)

Parameters

- x: double value

template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :

Get the sign of magnetization.

Return boolean sign of magnetization

Template Parameters

- Mag: magnetization type (MagP or MagT)

Parameters

- m: Mag

template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :

Fill a magnetization array with zeros.

Return void

Template Parameters

- Mag: magnetization type (MagP or MagT)

Parameters

- x: mag array
- n: array size

template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :

Set magnetization to zero.

Return void

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `x`: mag object

template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :

Abs for magnetization objects.

Return The absolute value of the input

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `a`: mag object

template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :

Flip magnetization sign if necessary.

Return The corrected mag object

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `x`: mag object
- `y`: value with desired sign

template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :

Arrow operator of original code.

Return The result of the operator.

Note The function behavior is different between *MagP64* and *MagT64*.

- *MagP64*: x * atanh(m)
- *MagT64*: m * x

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `m`: mag object
- `x`: value to multiply

template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :

Get magnetization sign.

Return sign computed as 1 - 2 * sign(x)

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `x`: mag object

`template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :`

Log operation for magnetization objects.

Return The result of the operation

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `x`: mag object

`template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :`

Convert a double to a mag value (as a constructor).

Return magnetization

Note The function behavior is different between *MagP64* and *MagT64*.

- *MagP64*: *MagP64(x)*
- *MagT64*: *MagT64*(atanh(x), -30, 30)

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `x`: double value

`template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :`

Convert a mag to double.

Return extract magnetization.

Note The function behavior is different between *MagP64* and *MagT64*.

- *MagP64*: return mag
- *MagT64*: return value

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `x`: mag object

`template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :`

Combine values to magnetizations.

Return The result of the combination.

Note The function behavior is different between *MagP64* and *MagT64*.

- *MagP64*: (x1 - x2) / (x1 + x2)

- *MagT64*: log(x1) - log(x2) * .5

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `x1`: double
- `x2`: double

`template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :`

Update magnetization.

Return The result of the update computed as newx * (1 - l) + oldx * l

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `newx`: mag object
- `oldx`: mag object
- `l`: double value

`template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :`

Perform `tanh` on magnetization value.

- **Note** The function behavior is different between *MagP64* and *MagT64*.
- *MagP64*: tanh(x)
- *MagT64*: x

Return The result of `tanh` as Mag.

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `x`: double value

`template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :`

Perform `erf` on magnetization value.

Return The result of `atanherf(x)`.

Note The function behavior is different between *MagP64* and *MagT64*.

- *MagP64*: erf(x)
- *MagT64*: atanherf(x)

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `x`: double value

template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :

Diff of magnetizations.

Return The result of the diff.

Note The function behavior is different between *MagP64* and *MagT64*.

- *MagP64*: (m1 - m2)/(1 - m1 * m2) clamped to [-1, 1]
- *MagT64*: m1 - m2

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `m1`: mag object
- `m2`: mag object

template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :

Compute the log1p for the combination of the magnetizations.

Return The result of the operation

Note The function behavior is different between *MagP64* and *MagT64*.

- *MagP64*: log((1. + (x.mag * y.mag)) * 0.5)
- *MagT64*: computation takes care of possible number overflows

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `x`: mag object
- `y`: mag object

template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :

Compute the crossentropy score for magnetization objects.

Return The resulting crossentropy score

Note The function behavior is different between *MagP64* and *MagT64*.

- *MagP64*: -x.mag * np.arctanh(y.mag) - np.log1p(- y.mag**2) * .5 + np.log(2)
- *MagT64*: -abs(y.mag) * (sign0(y.mag) * x.value - 1.) + lr(y.mag)

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `x`: mag object
- `y`: mag object

template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :

template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagT64 > ::value > :

Combine three *MagT64* variables.

Return combination of the input

Note This operation is valid only for *MagT64* variables up to now

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `H`: mag object
- `ap`: double
- `am`: double

`template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :`

Combine exactly three magnetizations.

Return The result of the mix

Note The function behavior is different between *MagP64* and *MagT64*.

- *MagP64*: H.mag * (erf(mp) - erf(mm)) / (2. + H.mag * (erf(mp) + erf(mm)))
- *MagT64*: auxmix(H, atanherf(mp), atanherf(mm))

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `H`: mag object
- `mp`: double
- `mm`: double

`template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > :`

Combine exactly three magnetizations.

Return The result of the mix

Note The function behavior is different between *MagP64* and *MagT64*.

- *MagP64*: (pp.mag - pm.mag) * H.mag / (2. + (pp.mag + pm.mag) * H.mag)
- *MagT64*: auxmix(H, pp.mag, pm.mag)

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `H`: mag object
- `pp`: mag object
- `pm`: mag object

Replicated Focusing Belief Propagation

Functions

template<class **Mag**>
double **theta_node_update_approx**(MagVec<*Mag*> *m*, *Mag* &*M*, **const** double **xi*, MagVec<*Mag*> *u*, *Mag* &*U*, **const** Params<*Mag*> &*params*, **const** long int &*nxi*, **const** long int &*nm*)

Messages update for a perceptron-like factor graph (approximated version computationally efficient in the limit of large number of weights)

Return Largest difference between new and old messages

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `m`: Total magnetization of variables nodes belonging to lower layer
- `M`: Total magnetization of variable node belonging to upper layer
- `xi`: Single input pattern
- `u`: Downward messages (cavity magnetizations) from factor node to lower variables nodes
- `U`: Upward message (cavity magnetizations) from factor node to upper variable node
- `params`: Parameters selected for the algorithm
- `nxi`: Size of input pattern
- `nm`: Number of variables node onto the lower layer

template<class **Mag**>
double **theta_node_update_accurate**(MagVec<*Mag*> *m*, *Mag* &*M*, **const** double **xi*, MagVec<*Mag*> *u*, *Mag* &*U*, **const** Params<*Mag*> &*params*, **const** long int &*nxi*, **const** long int &*nm*)

Messages update for a perceptron-like factor graph (fast approximated version)

Return Largest difference between new and old messages

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `m`: Total magnetization of variables nodes belonging to lower layer
- `M`: Total magnetization of variable node belonging to upper layer
- `xi`: Single input pattern
- `u`: Downward messages (cavity magnetizations) from factor node to lower variables nodes
- `U`: Upward message (cavity magnetizations) from factor node to upper variable node
- `params`: Parameters selected for the algorithm
- `nxi`: Size of input pattern
- `nm`: Number of variables node onto the lower layer

template<class **Mag**>
double **theta_node_update_exact**(MagVec<*Mag*> *m*, *Mag* &*M*, **const** double **xi*, MagVec<*Mag*> *u*, *Mag* &*U*, **const** Params<*Mag*> &*params*, **const** long int &*nxi*, **const** long int &*nm*)

Messages update for a perceptron-like factor graph (exact version)

Return Largest difference between new and old messages

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `m`: Total magnetization of variables nodes belonging to lower layer
- `M`: Total magnetization of variable node belonging to upper layer
- `xi`: Single input pattern
- `u`: Downward messages (cavity magnetizations) from factor node to lower variables nodes
- `U`: Upward message (cavity magnetizations) from factor node to upper variable node
- `params`: Parameters selected for the algorithm
- `nxi`: Size of input pattern
- `nm`: Number of variables node onto the lower layer

template<class **Mag**>
double **free_energy_theta**(**const** MagVec<*Mag*> *m*, **const** *Mag* &*M*, **const** double **xi*, **const** MagVec<*Mag*> *u*, **const** *Mag* &*U*, **const** long int &*nxi*, **const** long int &*nm*)

Computation of the free energy for a perceptron-like factor graph (fast approximated version)

Return Free energy for the system represented by a perceptron-like factor graph

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `m`: Total magnetization of variables nodes belonging to lower layer
- `M`: Total magnetization of variable node belonging to upper layer
- `xi`: Single input pattern
- `u`: Downward messages (cavity magnetizations) from factor node to lower variables nodes
- `U`: Upward message (cavity magnetizations) from factor node to upper variable node
- `nxi`: Size of input pattern
- `nm`: Number of variables node onto the lower layer

template<class **Mag**>
double **free_energy_theta_exact**(MagVec<*Mag*> *m*, **const** *Mag* &*M*, **const** double **xi*, MagVec<*Mag*> *u*, **const** *Mag* &*U*, **const** long int &*nm*)

Computation of the free energy for a perceptron-like factor graph (exact version)

Return Free energy for the system represented by a perceptron-like factor graph

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `m`: Total magnetization of variables nodes belonging to lower layer
- `M`: Total magnetization of variable node belonging to upper layer
- `xi`: Single input pattern
- `u`: Downward messages (cavity magnetizations) from factor node to lower variables nodes
- `U`: Upward message (cavity magnetizations) from factor node to upper variable node
- `nm`: Number of variables node onto the lower layer

template<class **Mag**>
double **m_star_update** (*Mag &m_j_star*, *Mag &m_star_j*, Params<*Mag*> *&params*)

Extra message update rule due to replicas.

Return Largest value between older maximum difference and the difference between new and old extra message

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `m_j_star`: Total magnetization of a weight node
- `m_star_j`: Extra message (cavity magnetizations) from replica node to its weight node
- `params`: Parameters selected for the algorithm

template<class **Mag**>
double **iterate** (Cavity_Message<*Mag*> *&messages*, **const** *Patterns &patterns*, Params<*Mag*> *&params*)

Management of the single iteration.

Return Largest difference between new and old messages across all updates

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `messages`: All magnetizations, both total and cavity, container
- `patterns`: All patterns, both input and output values, container
- `params`: Parameters selected for the algorithm

template<class **Mag**>
bool **converge** (Cavity_Message<*Mag*> *&messages*, **const** *Patterns &patterns*, Params<*Mag*> *&params*)

Management of all iterations within protocol step (i.e. constant focusing and replica parameteres)

Return True when convergence is reached, False otherwise

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `messages`: All magnetizations, both total and cavity, container
- `patterns`: All patterns, both input and output values, container
- `params`: Parameters selected for the algorithm

long int ***nonbayes_test**(long int ****const** *sign_m_j_star*, **const** *Patterns &patterns*, **const** long int *&K*)

Prediction of labels given weights and input patterns.

Return Predicted labels

Parameters

- `sign_m_j_star`: Total magnetization of weights nodes
- `patterns`: All patterns, both input and output values, container
- `K`: Number of nodes onto the hidden layer

template<class **Mag**>
long int **error_test**(**const** Cavity_Message<*Mag*> *&messages*, **const** *Patterns &patterns*)

Computation of number of mistaken predicted labels.

Return Number of mistaken predicted labels

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `messages`: All magnetizations, both total and cavity, container
- `patterns`: All patterns, both input and output values, container

template<class **Mag**>
double **free_energy**(**const** Cavity_Message<*Mag*> *&messages*, **const** *Patterns &patterns*, **const** Params<*Mag*> *&params*)

Computation of the free energy for the whole system.

Return Total free energy of the system

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `messages`: All magnetizations, both total and cavity, container
- `patterns`: All patterns, both input and output values, container
- `params`: Parameters selected for the algorithm

template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > ::type

Set the outcome variables nodes to training labels.

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `message`: All magnetizations, both total and cavity, container
- `output`: Output patterns (training labels)
- `beta`: Inverse of temperature (always infinite up to now)

template<class **Mag**>
long int ****focusingBP**(**const** long int *&K*, **const** *Patterns* *&patterns*, **const** long int *&max_iters*, **const** long int *&max_steps*, **const** long int *&seed*, **const** double *&damping*, **const** std::string *&accuracy1*, **const** std::string *&accuracy2*, **const** double *&randfact*, **const** *FocusingProtocol* *&fprotocol*, **const** double *&epsil*, int *nth* = 1, std::string *outfile* = "", std::string *outmessfiletmpl* = "", std::string *initmess* = "", **const** bool *&bin_mess* = false)

Management of all protocol step of the learning rule.

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `K`: Number of nodes onto the hidden layer
- `patterns`: All patterns, both input and output values, container
- `max_iters`: Highest number of iterations to run within same protocol step
- `max_steps`: Number of protocol steps
- `seed`: Seed for random generator inside *Cavity_Message* initial messages creator
- `damping`: Damping parameter for regularization of messages updates
- `accuracy1`: Accuracy level for first layer
- `accuracy2`: Accuracy level for second layer
- `randfact`: Random value used inside *Cavity_Message* initial messages creator
- `fprotocol`: Protocol type
- `epsil`: error tollerance
- `nth`: Number of cores to exploit
- `outfile`: Filename which evolution measurements can be stored in
- `outmessfiletmpl`: Filename which final messages can be written on
- `initmess`: Filename which initial messages can be taken from
- `bin_mess`: True if messages filename but me read/written as binary files, text files otherwise

template<class Mag, typename std ::enable_if< std ::is_same< Mag, MagP64 > ::value > ::type

Switch case for the right accuracy function.

Template Parameters

- `Mag`: magnetization type (MagP or MagT)

Parameters

- `acc`: accuracy name (possible values are "accurate", "exact", and "none")

Patterns

class Patterns

Abstract type used to store the input Pattern as (data, labels). To be provided as an argument to `focusingBP`.

Inputs and outputs must have the same length. Outputs entries must be {-1,1}. Intputs entries must be arrays in which each element is {-1,1}, and all the vectors must have the same length. The input pattern can be loaded from binary/ascii file, random generated or from a given matrix.

Public Functions

Patterns()

Default constructor.

Patterns(**const** std::string &*filename*, bool *bin* = false, **const** std::string &*del* = "t")

Load pattern from file.

The input file can store the values in binary or ascii format. For the **binary** format the function requires a file formatted following these instructions:

- Number of rows (long int)
- Number of columns (long int)
- labels array (long int) with a length equal to the number of rows
- data matrix (double) with a shape (number of rows x number of columns).

For the **ascii** format the function requires a file formatted following these instructions:

- labels array (as a series of separated numbers)
- matrix of data (as a series of rows separated by `\n`)

Note Outputs entries must be {-1,1}. Intputs entries must be arrays in which each element is {-1,1}.

Note In the ascii format the delimiter of the file can be set using the `del` variable.

Parameters

- `filename`: Input filename
- `bin`: switch between binary/ascii files (default = false)
- `del`: delimiter string if the file is in ascii fmt (default = "\t")

Patterns(**const** long int &*N*, **const** long int &*M*)

Generate random patter.

The pattern is generated using a Bernoulli distribution and thus it creates a data (matrix) + labels (vector) of binary values. The values are converted into the range (-1, 1) for the compatibility with the rFBP algorithm.

Parameters

- `N`: number of input vectors (samples)
- `M`: number of columns (probes)

Patterns(double **data, long int *label, **const** int &Nrow, **const** int &Ncol)

Copy pattern from arrays.

Data and labels are copied, so be careful with this constructor.

Note The copy of the arrays is performed for compatibility with the Python API of the library.

Parameters

- `data`: matrix of data in double format
- `label`: array of labels
- `Nrow`: number of rows in data matrix
- `Ncol`: number of columns in data matrix

Patterns &**operator=**(**const** *Patterns* &p)

Copy operator.

The operator performs a deep copy of the object and if there are buffers already allocated, the operatore deletes them and then re-allocates an appropriated portion of memory.

Parameters

- `p`: Pattern object

Patterns(**const** *Patterns* &p)

Copy constructor.

The copy constructor provides a deep copy of the object, i.e. all the arrays are copied and not moved.

Parameters

- `p`: Pattern object

~Patterns()

Destructor.

Completely destroy the object releasing the data/labels memory.

Public Members

long int **Nrow**

Number of input vectors (rows of input or samples)

long int **Ncol**

Number of cols in input (probes)

long int **Nout**

Lenght of output labels.

long int ***output**

Output vector.

double ****input**

Input matrix.

Params

template<class **Mag**>
class Params

Class to wrap training parameters.

This class is used by the rFBP functions to facilitate the moving of a set of training parameters along the series of functions.

Template Parameters

- `Mag`: magnetization used

Public Functions

Params(**const** int *&max_iters*, **const** double *&damping*, **const** double *&epsil*, **const** double *&beta*, **const** double *&r*, **const** double *&gamma*, **const** std::string *&accuracy1*, **const** std::string *&accuracy2*)

Parameter constructor.

Note In the constructor the value of gamma is converted to the appropriated Mag type.

Parameters

- `max_iters`: Number of iterations
- `damping`: Damping factor
- `epsil`: Error tollerance
- `beta`: 1 / KT
- `r`: Number of replicas - 1
- `gamma`: Hyperbolic tangent of distance weight between replicas (γ) as Mag object
- `accuracy1`: Updating accuracy of cavity probability (messages of hidden layers)
- `accuracy2`: Updating accuracy of cavity probability (messages of otuput node)

~Params()

Default destructor.

Public Members

long int **max_iters**

Number of iterations.

double **damping**

Damping factor.

double **epsil**

Error tollerance.

double **beta**

1/kT

double **r**

Number of replicas -1.

Mag **tan_gamma**
Hyperbolic tangent of distance weight between replicas (γ)

std::string **accuracy1**
Updating accuracy of cavity probability (messages of hidden layers)

std::string **accuracy2**
Updating accuracy of cavity probability (messages of output node)

Cavity Message

template<class **Mag**>
class Cavity_Message
Abstract type used to store weights and messages of rFBP algorithm.

The initial messages can be loaded from file and the resulting ones can be saved to file using the appropriated member functions.

Template Parameters

- `Mag`: magnetization chosen for training

Public Functions

Cavity_Message()
Default constructor.

Cavity_Message(**const** std::string *&filename*, **const** bool *&bin*)
Load messages from file.

The input file must have a very precise format.

- For the binary format we require a file with the following instructions:
  - N (long int)
  - M (long int)
  - K (long int)
  - m_star_j (K * N, double)
  - m_j_star (K * N, double)
  - m_in (M * K, double)
  - weights (M * K * N, double)
  - m_no (M * K, double)
  - m_on (M, double)
  - m_ni (M * K, double)

  All the double values are converted into the respective Mag format.
- For the ascii version the required file must have the following format:

```
fmt: plain
N,M,K: `N` `M` `K`
```

where `N` represents the numerical value of the N parameter. This header file is followed by ravel version of the required arrays (m_star_j, m_j_star, m_in, weights, m_no, m_on, m_ni) one per each line, divided by white spaces (or `\t`).

Note A valid file can be generated by the function `save_messages`.

Parameters

- `filename`: Input filename
- `bin`: switch between binary/ascii files (default = false)

Cavity_Message(**const** long int &*m*, **const** long int &*n*, **const** long int &*k*, **const** double &*x*, **const** int &*start*)

Generate random messages.

The cavity_messages' arrays are generated according a uniform distribution. The [0, 1] range is converted using the formula

```
x * (2. * dist() - 1.)
```

where `dist` represents the uniform random generator. All the values are converted to Mag types.

Parameters

- `m`: number of samples
- `n`: number of probes
- `k`: number of hidden layers
- `x`: initial value
- `start`: random seed

Cavity_Message(**const** *Cavity_Message*<Mag> &*m*)

Copy constructor.

The copy constructor provides a deep copy of the object, i.e. all the arrays are copied and not moved.

Parameters

- `m`: *Cavity_Message* object

Cavity_Message<Mag> &**operator=**(**const** Cavity_Message<Mag> &*m*)

Copy operator.

The operator performs a deep copy of the object and if there are buffers already allocated, the operatore deletes them and then re-allocates an appropriated portion of memory.

Parameters

- `m`: *Cavity_Message* object

~Cavity_Message()

Destructor.

Completely delete the object and release the memory of the arrays.

long int ****get_weights**()

Return weights matrix.

The weights are converted from double to long int, using the sign of each element, i.e.

```
1L - 2L * signbit(x)
```

This function can be used as getter member for the weight matrix used to predict new patterns.

Note The weight matrix used in the prediction is the `m_j_star` array!

void **save_weights**(**const** std::string &*filename*, Params<Mag> &*parameters*)

Save weight matrix to file.

Save the weight matrix to file and the related training parameters. This function provides a valid file for the function `read_weights` in ascii format. Only the weight matrix, i.e. `m_j_star`, is saved, since it is the only informative array for the prediction of new patterns. The training parameters are saved as header in the file.

Return The binirized format of the weight matrix, ready for the prediction.

Parameters

- `filename`: output filename
- `parameters`: *Params* object

void **save_weights**(**const** std::string &*filename*)

Save weight matrix to file.

Save **only** the weight matrix to file. This function provides a valid file for the function `read_weights` in binary format. Only the weight matrix, i.e. `m_j_star`, is saved, since it is the only informative array for the prediction of new patterns. The weight values are saved as double values and thus before use them to predict new values, it is necessary to apply the "get_weights" function.

Parameters

- `filename`: output filename

void **read_weights**(**const** std::string &*filename*, **const** bool &*bin*)

Load weight matrix from file.

This function read the weight matrix from a binary or ascii file. Its usage is in relation to the `save_weights` member function.

Parameters

- `filename`: input filename
- `bin`: switch between binary/ascii fmt

void **save_messages**(**const** std::string &*filename*, Params<Mag> &*parameters*)

Save all the messages to file.

This function dump the complete object to file with also the parameters used for the training section, according to the format required by the constructor.

Parameters

- `filename`: output filename
- `parameters`: *Params* object

void **save_messages**(**const** std::string *&filename*)
Save all the messages to a **binary** file.

This function dump the complete object to **binary** file, according to the format required by the constructor.

Parameters

- `filename`: output filename

Public Members

long int **M**
Input sample size.

long int **N**
Input layers size.

long int **K**
Number of hidden layers.

long int **seed**
Random seed.

MagVec3<Mag> **weights**
uw in the paper nomeclature

MagVec2<Mag> **m_star_j**
ux in the paper nomeclature

MagVec2<Mag> **m_j_star**
mw in the paper nomeclature

MagVec2<Mag> **m_in**
mτ1 in the paper nomeclature

MagVec2<Mag> **m_no**
Uτ1 in the paper nomeclature.

MagVec2<Mag> **m_ni**
uτ1 in the paper nomeclature

MagVec<Mag> **m_on**
mτ2 in the paper nomeclature

Atanherf

namespace AtanhErf

Functions

spline **getinp**()
Load spline data points from file.

The filename is hard coded into the function body and it must be placed in `$PWD/data/atanherf_interp.max_16.step_0.0001.first_1.dat`.

The variable PWD is defined at compile time and its value is set by the CMake file. If you want to use a file in a different location, please re-build the library setting the variable

```
-DPWD='new/path/location'
```

Return Spline object with the interpolated coordinates.

double **atanherf_largex**(**const** double &*x*)

Atanh of erf function for large values of x.

Return Approximated result of atanherf function.

Parameters

- `x`: Input variable.

double **atanherf_interp**(**const** double &*x*)

Atanh of erf function computed with the interpolation coordinates extracted by the spline.

Return Approximated result of atanherf function estimated using a pre-computed LUT. The LUT is generated using a cubic spline interpolation.

Parameters

- `x`: Input variable.

double **evalpoly**(**const** double &*x*)

Atanh of erf function evaluated as polynomial decomposition.

Return Approximated result of atanherf function.

Parameters

- `x`: Value as argument of atanherf function.

double **atanherf**(**const** double &*x*)

Atanh of erf function.

The result is evaluated with different numerical techniques according to its domain.

In particular:

- if its abs is lower than 2 -> "standard" formula
- if its abs is lower than 15 -> `atanherf_interp` formula
- if its abs is greater than 15 -> `atanherf_largex` formula

Return Approximated result of atanherf function.

Note The function automatically use the most appropriated approximation of the atanherf function to prevent possible overflows.

Parameters

- `x`: Input variable.

1.1.4 Python API

FocusingProtocol

class rfbp.FocusingProtocol.**Focusing_Protocol**(*protocol='standard_reinforcement', size=101*)

Bases: object

Focusing Protocol object. Abstract type representing a protocol for the focusing procedure, i.e. a way to produce successive values for the quantities γ, y and β. Currently, however, only β=Inf is supported. To be provided as an argument to focusingBP.

Available protocols are: StandardReinforcement, Scoping, PseudoReinforcement and FreeScoping.

- StandardReinforcement: returns γ=Inf and y=1/(1-x), where x is taken from the given range r.
- Scoping: fixed y and a varying γ taken from the given γr range.
- PseudoReinforcement: both γ and y are progressively increased, according to the formulas

```
>>> γ = atanh(ρ**x)
>>> y = 1+ρ**(1-2*x)/(1-ρ)
```

where ρ is taken from the given range(s) r. With x=0, this is basically the same as StandardReinforcement.

- FreeScoping: just returns the values of (γ,y) from the given list

Parameters

- **protocol** (*string*) – The value of string can be only one of ['scoping', 'pseudo_reinforcement', 'free_scoping', 'standard_reinforcement']
- **size** (*int (default = 101)*) – Dimension of update protocol

Example

```
>>> from ReplicatedFocusingBeliefPropagation import Focusing_Protocol
>>>
>>> fprotocol = Focusing_Protocol('scoping', 101)
>>> fprotocol
  Focusing_Protocol(protocol=scoping, size=101)
```

beta

Return the 'beta' array

Returns **beta** – The vector of the beta values

Return type array-like

fprotocol

Return the Cython object

Returns **fprotocol** – The cython object wrapped by the Pattern class

Return type Cython object

Notes

> **Warning:** We discourage the use of this property if you do not know exactly what you are doing!

gamma

Return the 'gamma' array

Returns **gamma** – The vector of the gamma values

Return type array-like

n_rep

Return the 'n_rep' array

Returns **n_rep** – The vector of the n_rep values

Return type array-like

num_of_replicas

Return the number of replicas

Returns **nrep** – The number of replicas

Return type int

Mag type

MagP64

class `rfbp.MagP64.`**`MagP64`**`(x)`

Bases: `ReplicatedFocusingBeliefPropagation.rfbp.Mag.BaseMag`

__mod__`(m)`

Clip value in [-1, 1].

Parameters **m** (`MagP64`) – The input value

Returns **m** – The MagP64 of the operation between the two mags. The clip operation is computed as np.clip((self.mag + m.mag) / (1. + self.mag * m.mag), -1., 1.)

Return type *MagP64*

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>> x = np.random.uniform(low=0., high=10)
>>> y = np.random.uniform(low=0., high=10)
>>> m1 = MagP64(x)
>>> m2 = MagP64(y)
>>> mx = m1 % m2
>>> my = m2 % m1
>>> assert np.isclose(mx.mag, my.mag)
>>> assert np.isclose(mx.value, my.value)
>>> assert -1. <= mx.mag <= 1.
>>> assert -1. <= my.mag <= 1.
```

(continues on next page)

(continued from previous page)

```
>>> assert -1. <= mx.value <= 1.
>>> assert -1. <= my.value <= 1.
```

__xor__(*m*)

Mag product

Parameters **m** (MagP64) – The input value

Returns **m** – The product of mags

Return type *MagP64*

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>> x = np.random.uniform(low=0., high=10)
>>> y = np.random.uniform(low=0., high=10)
>>> m1 = MagP64(x)
>>> m2 = MagP64(y)
>>> mx = m1 ^ m2
>>> my = m2 ^ m1
>>> assert np.isclose(mx.mag, my.mag)
>>> assert np.isclose(mx.value, my.value)
```

static convert(*x*)

Convert a float to a mag value (as a constructor)

Parameters **x** (*float*) – The number to convert

Returns **m** – Convert any-number to a MagP64 type

Return type *MagP64*

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>>
>>> x = np.random.uniform(low=0., high=10)
>>> m1 = MagP64.convert(x)
>>> m2 = MagP64(x)
>>> assert m1.mag == m2.mag
>>> assert m1.value == m2.value
```

static couple(*x1*, *x2*)

Combine two mags as diff / sum

Parameters

- **x1** (*float*) – The first element of the operation
- **x2** (*float*) – The second element of the operation

Returns **x** – In MagP64 the value is equal to the magnetization since the tanh operation is neglected

Return type float

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>>
>>> x = np.random.uniform(low=0., high=10)
>>> y = np.random.uniform(low=0., high=10)
>>> mx = MagP64.couple(x, y)
>>> my = MagP64.couple(y, x)
>>> assert np.isclose(abs(mx.mag), abs(my.mag))
>>> assert np.isclose(abs(mx.value), abs(my.value))
```

magformat

Return the mag description

Returns **plain** – The MagP64 type corresponds to a plain operation

Return type str

Example

```
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>> m = MagP64(3.14)
>>> m.magformat
  'plain'
```

static merf(*x*)

Perform erf on magnetization value (MagP64(erf(x)) in this case)

Parameters **x** (*float*) – The input value

Returns **m** – The MagP64 version of the erf(x)

Return type *MagP64*

Example

```
>>> import numpy as np
>>> from scipy.special import erf
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>>
>>> x = np.random.uniform(low=0., high=10)
>>> mx = MagP64.merf(x)
>>> assert 0 <= mx.mag <= 1
>>> assert np.isclose(mx.mag, erf(x))
```

static mtanh(*x*)

Perform tanh on magnetization value (MagP64(tanh(x)) in this case)

Parameters **x** (*float*) – The input value

Returns **m** – The MagP64 version of the tanh(x)

Return type *MagP64*

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>>
>>> x = np.random.uniform(low=0., high=10)
>>> mx = MagP64.mtanh(x)
>>> assert 0 <= mx.mag <= 1
>>> assert np.isclose(mx.mag, np.tanh(x))
```

value

Return the mag value

Returns **x** – In MagP64 the value is equal to the magnetization since the tanh operation is neglected

Return type float

Example

```
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>> x = np.random.uniform(low=0, high=1.)
>>> m = MagP64(x)
>>> assert np.isclose(m.mag, x)
>>> assert np.isclose(m.value, x)
```

MagT64

class rfbp.MagT64.**MagT64**(*x*, *mInf=30.0*)

Bases: ReplicatedFocusingBeliefPropagation.rfbp.Mag.BaseMag

__mod__(*m*)

In this case the mod operation corresponds to a sum of mags

Parameters **m** (MagT64) – The input value

Returns **m** – The MagT64 of the operation between the two mags. The mod operation corresponds to self.mag + m.mag

Return type *MagT64*

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagT64
>>>
>>> x = np.random.uniform(low=0., high=10)
>>> y = np.random.uniform(low=0., high=1)
>>> m1 = MagT64(x)
>>> m2 = MagT64(y)
>>>
>>> mx = m1 % m2
>>> my = m2 % m1
>>> assert np.isclose(mx.mag, my.mag)
```

(continues on next page)

(continued from previous page)

```
>>> assert np.isclose(mx.value, my.value)
>>>
>>> null = MagT64(0.)
>>> mx = m1 % null
>>> assert np.isclose(mx.mag, m1.mag)
>>> assert np.isclose(mx.value, m1.value)
```

__xor__(*m*)

Mag product

Parameters **m** (MagT64) – The input value

Returns **m** – The product of mags

Return type *MagT64*

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagT64
>>>
>>> x = np.random.uniform(low=0., high=10)
>>> y = np.random.uniform(low=0., high=1)
>>> m1 = MagT64(x)
>>> m2 = MagT64(y)
>>>
>>>
>>> mx = m1 ^ m2
>>> my = m2 ^ m1
>>> assert np.isclose(mx.mag, my.mag)
>>> assert np.isclose(mx.value, my.value)
>>>
>>> mx = (-m1) ^ (-m2)
>>> my = (-m2) ^ (-m1)
>>> assert not np.isclose(mx.mag, my.mag)
>>> assert not np.isclose(mx.value, my.value)
>>>
>>> null = MagT64(0.)
>>> mx = m1 ^ null
>>> assert np.isclose(mx.mag, 0.)
>>> assert np.isclose(mx.value, 0.)
>>>
>>> mx = m1 ^ MagT64(float('inf'))
>>> assert np.isclose(mx.mag, m1.mag)
>>> assert np.isclose(mx.value, m1.value)
```

static convert(*x*)

Convert a float to a mag value (as a constructor)

Parameters **x** (*float*) – The number to convert

Returns **m** – Convert any-number to a MagT64 type

Return type *MagT64*

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagT64
>>>
>>> x = np.random.uniform(low=0., high=10)
>>> mx = MagT64.convert(x)
>>> assert -30. <= mx.mag <= 30.
>>> assert np.isclose(mx.mag, np.arctanh(x))
>>> assert np.isclose(mx.value, x)
```

static couple(*x1*, *x2*)

Combine two mags

Parameters

- **x1** (*float*) – The first element of the operation
- **x2** (*float*) – The second element of the operation

Returns **x** – Mags combination as np.log(x1 / x2) * .5

Return type float

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagT64
>>>
>>> x = np.random.uniform(low=0., high=10)
>>> y = np.random.uniform(low=0., high=10)
>>> mx = MagT64.couple(x, y)
>>> my = MagT64.couple(y, x)
>>> assert np.isclose(abs(mx.mag), abs(my.mag))
>>> assert np.isclose(abs(mx.value), abs(my.value))
```

magformat

Return the mag description

Returns **tanh** – The MagT64 type corresponds to a tanh operation

Return type str

Example

```
>>> from ReplicatedFocusingBeliefPropagation import MagT64
>>> m = MagT64(3.14)
>>> m.magformat
  'tanh'
```

static merf(*x*)

Perform erf on magnetization value (MagT64(erf(x)) in this case)

Parameters **x** (*float*) – The input value

Returns **m** – The MagT64 version of the erf(x)

Return type *MagT64*

Example

```
>>> import numpy as np
>>> from scipy.special import erf
>>> from ReplicatedFocusingBeliefPropagation import MagT64
>>>
>>> x = np.random.uniform(low=0., high=10)
>>> mx = MagT64.merf(x)
>>> assert np.isclose(mx.mag, np.arctanh(erf(x)))
```

static mtanh(*x*)

Perform tanh on magnetization value (MagT64(x) in this case)

Parameters **x** (*float*) – The input value

Returns **m** – The MagT64 version of the tanh(x)

Return type *MagT64*

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagT64
>>>
>>> x = np.random.uniform(low=0., high=10)
>>> mx = MagT64.mtanh(x)
>>> assert np.isclose(mx.mag, x)
>>> assert np.isclose(mx.value, np.tanh(x))
```

value

Return the mag value

Returns **x** – In MagT64 the value is equal to the tanh(x)

Return type float

Example

```
>>> from ReplicatedFocusingBeliefPropagation import MagT64
>>> x = np.random.uniform(low=0, high=1.)
>>> m = MagT64(x)
>>> assert np.isclose(m.mag, x)
>>> assert np.isclose(m.value, np.tanh(x))
```

Magnetization functions

rfbp.magnetization.**arrow**(*m*, *x*)

Arrow operator of original code

Parameters

- **x** (*Mag object*) – Input variable
- **y** (*float*) – Input variable

Returns **m** – The result of the operator

Return type Mag object

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>> from ReplicatedFocusingBeliefPropagation import magnetization as mag
>>> x = np.random.uniform(low=0., high=1.)
>>> m = MagP64(np.random.uniform(low=0., high=1.))
>>>
>>> x = mag.arrow(m, x)
>>> assert isinstance(x, MagP64)
>>>
>>>
>>> y = np.random.uniform(low=0., high=1.)
>>> mag.arrow(x, y)
  ValueError('m must be MagP64 or MagT64')
```

Notes

Note: The computation of the arrow operator is different from MagP64 and MagT64.

- In MagP64 the computation is equivalent to

  mtanh(x * arctanh(m)).

- In MagT64 the computation is equivalent to

  mtanh(x * m)

`rfbp.magnetization.`**`auxmix`**(*H*, *ap*, *am*)

Combine three MagT64 magnetizations

Parameters

- **H** (*MagT64 object*) – Input variable
- **ap** (*float*) – Input variable
- **am** (*float*) – Input variable

Returns **m** – The result of the mix

Return type MagT64 object

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>> from ReplicatedFocusingBeliefPropagation import magnetization as mag
>>>
>>> x = np.random.uniform(low=0., high=1.)
>>> y = np.random.uniform(low=0., high=1.)
>>>
>>> mx = mag.auxmix(MagT64(0.), x, y)
```

(continues on next page)

(continued from previous page)

```
>>> assert mx.mag == 0.
>>>
>>> m = MagP64(np.random.uniform(low=0., high=1.))
>>> mag.auxmix(m, y, x)
  ValueError('H must be a MagT64 magnetization type')
```

Notes

Note: This operation is valid only for MagT64 variables up to now

`rfbp.magnetization.bar`(*m1*, *m2*)

Diff of magnetizations

Parameters

- **m1** (*Mag object*) – Input variable
- **m2** (*Mag object*) – Input variable

Returns **m** – The result of the diff as (m1 - m2)/(1 - m1 * m2) clamped to [-1, 1] if MagP else m1 - m2

Return type Mag object

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>> from ReplicatedFocusingBeliefPropagation import magnetization as mag
>>>
>>> m = MagP64(np.random.uniform(low=0., high=1.))
>>> n = MagP64(np.random.uniform(low=0., high=1.))
>>>
>>> mx = mag.bar(m, n)
>>> my = mag.bar(n, m)
>>> assert -1 <= mx.mag <= 1.
>>> assert -1 <= my.mag <= 1.
>>> assert np.isclose(abs(mx.mag), abs(my.mag))
```

`rfbp.magnetization.copysign`(*x*, *y*)

Flip magnetization sign if necessary

Parameters

- **x** (*Mag object*) – Input variable to check
- **y** (*float*) – Varibale from which copy the sign

Returns **m** – The corrected mag object

Return type Mag object

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>> from ReplicatedFocusingBeliefPropagation import magnetization as mag
>>> x = np.random.uniform(low=0., high=1.)
>>> m = MagP64(np.random.uniform(low=0., high=1.))
>>>
>>> x = mag.copysign(m, x)
>>> assert x.mag == m.mag
>>>
>>> x = mag.copysign(-m, x)
>>> assert x.mag == m.mag
>>>
>>> x = mag.copysign(m, -x)
>>> assert x.mag == -m.mag
>>>
>>> x = mag.copysign(-m, -x)
>>> assert x.mag == -m.mag
```

`rfbp.magnetization.`**`damp`**(*newx*, *oldx*, *l*)

Update magnetization

Parameters

- **newx** (*Mag object*) – Update magnetization value
- **oldx** (*Mag object*) – Old magnetization value
- **l** (*float*) – Scale factor

Returns **m** – The result of the update computed as newx * (1 - l) + oldx * l

Return type Mag object

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>> from ReplicatedFocusingBeliefPropagation import magnetization as mag
>>>
>>> x = np.random.uniform(low=0., high=1.)
>>> m = MagP64(np.random.uniform(low=0., high=1.))
>>> n = MagP64(np.random.uniform(low=0., high=1.))
>>>
>>> mx = mag.damp(m, n, x)
>>> my = mag.damp(n, m, 1. - x)
>>> assert np.isclose(mx.mag, my.mag)
>>> assert np.isclose(mx.value,m y.value)
>>>
>>> mx = mag.damp(m, n, 0.)
>>> assert np.isclose(mx.mag, m.mag)
```

`rfbp.magnetization.`**`erfmix`**(*H*, *mp*, *mm*)

Combine three magnetizations with the erf

Parameters

- **H** (*Mag object*) – Input variable

- **mp** (*float*) – Input variable
- **mm** (*float*) – Input variable

Returns **m** – The result of the mix

Return type Mag object

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>> from ReplicatedFocusingBeliefPropagation import magnetization as mag
>>>
>>> x = np.random.uniform(low=0., high=1.)
>>> m = MagP64(np.random.uniform(low=0., high=1.))
>>>
>>> mx = mag.erfmix(MagP64(0.), x, x)
>>> assert np.isclose(mx.mag, 0.)
>>>
>>> mx = mag.erfmix(m, 0., 0.)
>>> assert np.isclose(mx.mag, 0.)
```

Notes

Note: The computation of the erfmix is different from MagP64 and MagT64.

- In MagP64 the computation is equivalent to

  H.mag * (erf(mp) - erf(mm)) / (2. + H.mag * (erf(mp) + erf(mm)))

- In MagT64 the computation is equivalent to

  auxmix(H, atanherf(mp), atanherf(mm))

`rfbp.magnetization.`**`exactmix`**(*H*, *pp*, *pm*)

Combine exactly three magnetizations

Parameters

- **H** (*Mag object*) – Input variable
- **pp** (*Mag object*) – Input variable
- **pm** (*Mag object*) – Input variable

Returns **m** – The result of the mix

Return type Mag object

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>> from ReplicatedFocusingBeliefPropagation import magnetization as mag
>>>
```

(continues on next page)

(continued from previous page)

```
>>> x = np.random.uniform(low=0., high=1.)
>>> m = MagP64(np.random.uniform(low=0., high=1.))
>>>
>>> mag.exactmix(m, [x], [m])
  ValueError('Input variables must magnetizations (MagP64 or MagT64)')
>>>
>>> mx = mag.exactmix(MagP64(0.), m, m)
>>> assert np.isclose(x.mag, 0.)
>>>
>>> mx = mag.exactmix(m, m, m)
>>> assert np.isclose(mx.mag, 0.)
```

Notes

Note: The computation of the exactmix is different from MagP64 and MagT64.

- In MagP64 the computation is equivalent to

  (pp.mag - pm.mag) * H.mag / (2. + (pp.mag + pm.mag) * H.mag)

- In MagT64 the computation is equivalent to

  auxmix(H, pp.mag, pm.mag)

`rfbp.magnetization.`**`log1pxy`**(*x*, *y*)

Compute the log1p for the combination of the magnetizations

Parameters

- **x** (*Mag object*) – The input variable
- **y** (*Mag object*) – The input variable

Returns **res** – The result of the operation

Return type float

Notes

Note: The computation of the function is different from MagP64 and MagT64.

- In MagP64 the computation is equivalent to

  np.log((1. + (x.mag * y.mag)) * 0.5)

- In MagT64 the computation takes care of possible number overflows

`rfbp.magnetization.`**`logZ`**(*u0*, *u*)

`rfbp.magnetization.`**`logmag2p`**(*x*)

Log operation for magnetization objects

Parameters **x** (*Mag object*) – Input variable

Returns **m** – The result of the operation

Return type Mag object

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>> from ReplicatedFocusingBeliefPropagation import magnetization as mag
>>>
>>> x = mag.logmag2p(MagP64(0.))
>>> assert np.isclose(x, np.log(.5))
>>>
>>> x = mag.logmag2p(MagT64(-1.))
>>> assert np.isinf(x)
```

`rfbp.magnetization.`**`lr`**(*x*)

log1p for magnetizations

Parameters **x** (*float*) – Input variable

Returns **res** – value computed as log1p(exp(-2*abs(x)))

Return type float

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import magnetization as mag
>>> x = np.random.uniform(low=0., high=1.)
>>> assert mag.lr(x) >= 0.
```

`rfbp.magnetization.`**`mabs`**(*x*)

Abs for magnetization objects

Parameters **x** (*Mag object*) – Input variable

Returns **abs** – The absolute value of the input

Return type float

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>> from ReplicatedFocusingBeliefPropagation import magnetization as mag
>>> x = np.random.uniform(low=0., high=1.)
>>> assert mag.mabs(MagP64(x)) >= 0.
>>> assert mag.mabs(MagP64(-x)) >= 0.
>>> assert mag.mabs(MagP64(-x)) == mag.mabs(MagP64(x))
>>>
>>> mag.mabs(x)
  ValueError('Incompatible type found. x must be a Mag')
```

`rfbp.magnetization.`**`mcrossentropy`**(*x*, *y*)

Compute the crossentropy score for magnetization objects

Parameters

- **x** (*Mag objects*) – Input variable
- **y** (*Mag objects*) – Input variable

Returns **res** – The resulting crossentropy score

Return type float

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>> from ReplicatedFocusingBeliefPropagation import MagT64
>>> from ReplicatedFocusingBeliefPropagation import magnetization as mag
>>>
>>> x = mag.mcrossentropy(MagT64(-float('Inf')), MagT64(float('Inf')))
>>> assert np.isinf(x)
>>>
>>> x = mag.mcrossentropy(MagP64(0.), MagP64(0.))
>>> assert np.isclose(x, np.log(2))
>>>
>>> x = mag.mcrossentropy(MagP64(0.), MagP64(1.))
>>> assert np.isnan(x)
>>>
>>> x = mag.mcrossentropy(MagP64(1.), MagP64(1.))
>>> assert np.isnan(x)
>>>
>>> x = mag.mcrossentropy(MagT64(0.), MagT64(0.))
>>> y = mag.mcrossentropy(MagT64(1.), MagT64(0.))
>>> assert np.isclose(x, y)
>>>
>>> x = mag.mcrossentropy(MagT64(float('Inf')), MagT64(float('Inf')))
>>> assert np.isclose(x, 0.)
>>>
>>> x = np.random.uniform(low=0., high=1.)
>>> y = np.random.uniform(low=0., high=1.)
>>> mag.mcrossentropy(x, y)
  ValueError('Both magnetizations must be the same')
```

Notes

Note: The computation of the mcrossentropy is different from MagP64 and MagT64.

- In MagP64 the computation is equivalent to

  -x.mag * np.arctanh(y.mag) - np.log1p(- y.mag**2) * .5 + np.log(2)

- In MagT64 the computation is equivalent to

  -abs(y.mag) * (sign0(y.mag) * x.value - 1.) + lr(y.mag)

`rfbp.magnetization.`**`sign0`**`(x)`

Sign operation valid also for magnetizations

Parameters **x** (*float*) – Input variable

Returns **res** – sign evaluated as 1 - 2*signbit(x)

Return type float

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>> from ReplicatedFocusingBeliefPropagation import magnetization as mag
>>> x = np.random.uniform(low=0., high=1.)
>>> m = MagP64(x)
>>> assert mag.sign0(x) in (-1, 1)
>>> assert mag.sign0(-x) == 1
>>> assert mag.sign0(m) in (-1, 1)
```

`rfbp.magnetization.zero(x)`

Set magnetization to zero

Parameters **x** (*Mag object*) – Input variable

Returns

Return type None

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>> from ReplicatedFocusingBeliefPropagation import magnetization as mag
>>> x = np.random.uniform(low=0., high=1.)
>>> m = MagP64(3.14)
>>> mag.zero(m)
>>>
>>> assert m.mag == 0.
>>> assert m.value == 0.
>>> mag.zero(x)
  ValueError('Incompatible type found. x must be a Mag')
```

`rfbp.magnetization.zeros(x)`

Fill array of magnetizations with zeros

Parameters **x** (*array-like*) – Input array or list of Mag objects

Returns

Return type None

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import MagP64
>>> from ReplicatedFocusingBeliefPropagation import magnetization as mag
>>> x = np.random.uniform(low=0., high=1.)
>>> mags = [MagP64(_) for _ in range(10)]
>>> mag.zeros(mags)
>>> assert all((i.mag == 0 for i in mags))
>>>
>>> l = 3.14
>>> mag.zeros(3.14)
  ValueError('zeros takes an iterable object in input')
```

(continues on next page)

(continued from previous page)

```
>>>
>>> l = [MagP64(3.14), x]
>>> mag.zeros(l)
  ValueError('Incompatible type found. x must be an iterable of Mags')
```

Mag

class rfbp.Mag.**BaseMag**(*x*)

Bases: object

__add__(*m*)

Sum of mags (overload operator +)

Parameters **m** (*mag-like object*) – A mag data type is required.

Returns **mag** – The result of the operation

Return type *BaseMag*

__eq__(*m*)

Check mag equality

Parameters **m** (*mag-like object*) – A mag data type is required.

Returns **res** – The result of the operation

Return type bool

__mul__(*x*)

Overload operator * between mag and number

Parameters **x** (*float*) – This function takes any type of object

Returns **res** – The result of the operation

Return type float

__ne__(*m*)

Check mag difference

Parameters **m** (*mag-like object*) – A mag data type is required.

Returns **res** – The result of the operation

Return type bool

__neg__()

Overload operator -mag

Returns **mag** – The result of the operation

Return type *BaseMag*

__sub__(*m*)

Overload operator - between mags

Parameters **m** (*mag-like object*) – A mag data type is required.

Returns **x** – The result of the operation

Return type float

Notes

Note: In the operation are involved the "values" of the magnetizations

__truediv__(*x*)

Overload operator /

Parameters **x** (*float*) – This function takes any type of object

magformat

The name of the specialization used

Returns **name** – Name of the specialization

Return type str

value

The value of the magnetization (it could be equal or processed according to the specialization function)

Returns **x** – The value of the magnetization

Return type float

ReplicatedFocusingBeliefPropagation

class rfbp.ReplicatedFocusingBeliefPropagation.**ReplicatedFocusingBeliefPropagation**(*mag=<cla 'Replicated-Focusing-Belief-Propagation.rfbp.M hidden=3, max_iter= seed=135, damping=0.5, accuracy=('acc 'exact'), randfact=0.1, epsil=0.1, protocol='pseua size=101, nth=2, verbose=False*)

Bases: `sklearn.base.BaseEstimator`, `sklearn.base.ClassifierMixin`

ReplicatedFocusingBeliefPropagation classifier

Parameters

- **mag** (*Enum Mag (default = MagP64)*) – Switch magnetization type
- **hidden** (*int (default = 3)*) – Number of hidden layers
- **max_iters** (*int (default = 1000)*) – Number of iterations
- **seed** (*int (default = 135)*) – Random seed
- **damping** (*float (default = 0.5)*) – Damping parameter
- **accuracy** (*pair of string (default : ('accurate', 'exact'))*) – Accuracy of the messages computation at the hidden units level. Possible values are ('exact', 'accurate', 'approx', 'none')
- **randfact** (*float (default = 0.1)*) – Seed random generator of Cavity Messages

- **epsil** (*float (default = 0.1)*) – Threshold for convergence
- **protocol** (*string (default = 'pseudo_reinforcement')*) – Updating protocol. Possible values are ["scoping", "pseudo_reinforcement", "free_scoping", "standard_reinforcement"]
- **size** (*int (default = 101)*) – Number of updates
- **nth** (*int (default = max_num_of_cores)*) – Number of thread to use in the computation
- **verbose** (*bool (default = False)*) – Enable or disable stdout on shell

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import␣
↪ReplicatedFocusingBeliefPropagation as rFBP
>>>
>>> N, M = (20, 101) # M must be odd
>>> data = np.random.choice([-1, 1], p=[.5, .5], size=(N, M))
>>> label = np.random.choice([-1, 1], p=[.5, .5], size=(N, ))
>>>
>>> rfbp = rFBP()
>>> rfbp.fit(data, label)
  ReplicatedFocusingBeliefPropagation(randfact=0.1, damping=0.5, accuracy=(
↪'accurate', 'exact'), nth=1, epsil=0.1, seed=135, size=101, hidden=3,␣
↪verbose=False, protocol=pseudo_reinforcement, mag=<class
↪'ReplicatedFocusingBeliefPropagation.rfbp.MagP64.MagP64'>, max_iter=1000)
>>> predicted_labels = rfbp.predict(data)
```

Notes

Note: The input data must be composed by binary variables codified as *[-1, 1]*, since the model works only with spin-like variables.

References

- C. Baldassi, C. Borgs, J. T. Chayes, A. Ingrosso, C. Lucibello, L. Saglietti, and R. Zecchina. "Unreasonable effectiveness of learning neural networks: From accessible states and robust ensembles to basic algorithmic schemes", Proceedings of the National Academy of Sciences, 113(48):E7655-E7662, 2016.
- C. Baldassi, A. Braunstein, N. Brunel, R. Zecchina. "Efficient supervised learning in networks with binary synapses", Proceedings of the National Academy of Sciences, 104(26):11079-11084, 2007.
- C. Baldassi, F. Gerace, C. Lucibello, L. Saglietti, R. Zecchina. "Learning may need only a few bits of synaptic precision", Physical Review E, 93, 2016
- D. Dall'Olio, N. Curti, G. Castellani, A. Bazzani, D. Remondini. "Classification of Genome Wide Association data by Belief Propagation Neural network", CCS Italy, 2019.

fit (*X*, *y=None*)

Fit the ReplicatedFocusingBeliefPropagation model meta-transformer

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – The training input samples.
- **y** (*array-like, shape (n_samples,)*) – The target values (integers that correspond to classes in classification, real numbers in regression).

Returns self

Return type ReplicatedFocusingBeliefPropagation object

load_weights (*weightfile*, *delimiter='\t'*, *binary=False*)

Load weights from file

Parameters

- **weightfile** (*string*) – Filename of weights
- **delimiter** (*char*) – Separator for ascii loading
- **binary** (*bool*) – Switch between binary and ascii loading style

Returns self

Return type ReplicatedFocusingBeliefPropagation object

Example

```
>>> from ReplicatedFocusingBeliefPropagation import␣
↪ReplicatedFocusingBeliefPropagation as rFBP
>>>
>>> clf = rFBP()
>>> clf.load_weights('path/to/weights_filename.csv', delimiter=',',␣
↪binary=False)
 ReplicatedFocusingBeliefPropagation(randfact=0.1, damping=0.5, accuracy=(
↪'accurate', 'exact'), nth=1, epsil=0.1, seed=135, size=101, hidden=3,␣
↪verbose=False, protocol=pseudo_reinforcement, mag=<class
↪'ReplicatedFocusingBeliefPropagation.rfbp.MagP64.MagP64'>, max_iter=1000)
```

predict (*X*)

Predict the new labels computed by ReplicatedFocusingBeliefPropagation model

Parameters **X** (*array of shape [n_samples, n_features]*) – The input samples.

Returns **y** – The predicted target values.

Return type array of shape [n_samples]

save_weights (*weightfile*, *delimiter='\t'*, *binary=False*)

Load weights from file

Parameters

- **weightfile** (*string*) – Filename to dump the weights
- **delimiter** (*char*) – Separator for ascii dump
- **binary** (*bool*) – Switch between binary and ascii dumping style

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import␣
↪ReplicatedFocusingBeliefPropagation as rFBP
>>>
>>> N, M = (20, 101) # M must be odd
>>> data = np.random.choice([-1, 1], p=[.5, .5], size=(N, M))
>>> label = np.random.choice([-1, 1], p=[.5, .5], size=(N, ))
>>>
>>> rfbp = rFBP()
>>> rfbp.fit(data, label)
>>> rfbp.save_weights('path/to/weights_filename.csv', delimiter=',',␣
↪binary=False)
```

Patterns

class rfbp.Patterns.**Pattern**(*X=None*, *y=None*)

Bases: object

Pattern object for C++ compatibility. The Pattern object is just a simple wrap of a data (matrix) + labels (vector). This object type provide a compatibility with the rFBP functions in C++ and it provides also a series of checks for the input validity.

Parameters

- **X** (*None or 2D array-like or string*) – Input matrix of variables as (Nsample, Nfeatures) or filename with the input stored in the same way
- **y** (*None or 1D array-like*) – Input labels. The label can be given or read from the input filename as first row in the file.

Example

```
>>> import numpy as np
>>> from ReplicatedFocusingBeliefPropagation import Pattern
>>>
>>> n_sample, n_feature = (20, 101) # n_feature must be odd
>>> data = np.random.choice(a=(-1, 1), p=(.5, .5), size=(n_sample, n_feature))
>>> labels = np.random.choice(a=(-1, 1), p=(.5, .5), size=(n_sample, ))
>>>
>>> pt = Pattern(X=data, y=labels)
>>> # dimensions
>>> assert pt.shape == (n_sample, n_feature)
>>> # data
>>> np.testing.assert_allclose(pt.data, data)
>>> # labels
>>> np.testing.assert_allclose(pt.labels, labels)
```

data

Return the data matrix

Returns **data** – The data matrix as (n_sample, n_features) casted to integers.

Return type array-like

labels

Return the label array

Returns **labels** – The labels vector as (n_sample,) casted to integers.

Return type array-like

load(*filename*, *binary=False*, *delimiter='\t'*)

Load pattern from file. This is the main utility of the Pattern object. You can use this function to load data from csv-like files OR from a binary file.

Parameters

- **filename** (*str*) – Filename/Path to the Pattern file
- **binary** (*bool*) – True if the filename is in binary fmt; False for ASCII fmt
- **delimiter** (*str*) – Separator of input file (valid if binary is False)

Example

```
>>> from ReplicatedFocusingBeliefPropagation import Pattern
>>>
>>> data = Pattern().load(filename='path/to/datafile.csv', delimiter=',',␣
↪binary=False)
>>> data
  Pattern[shapes=(10, 20)]
```

pattern

Return the pattern Cython object

Returns **pattern** – The cython object wrapped by the Pattern class

Return type Cython object

Notes

> **Warning:** We discourage the use of this property if you do not know exactly what you are doing!

random(*shape*)

Generate Random pattern. The pattern is generated using a Bernoulli distribution and thus it creates a data (matrix) + labels (vector) of binary values. The values are converted into the range (-1, 1) for the compatibility with the rFBP algorithm.

Parameters **shapes** (*tuple*) – a 2-D tuple with (M, N) where M is the number of samples and N the number of probes

Example

```
>>> from ReplicatedFocusingBeliefPropagation import Pattern
>>>
>>> n_sample = 10
>>> n_feature = 20
>>> data = Pattern().random(shape=(n_sample, n_feature))
>>> assert data.shape == (n_sample, n_feature)
```

(continues on next page)

(continued from previous page)

```
>>> data
  Pattern[shapes=(10, 20)]
```

shape

Return the shape of the data matrix

Returns **shape** – The tuple related to the data dimensions (n_sample, n_features)

Return type tuple

Atanherf

`rfbp.atanherf.`**`atanherf`**`(x)`

Compute atanh(erf) for general values of x

Parameters **x** (*float*) – Input variable

Returns **atanh(erf(x))** – atanh(erf(x)) for any value of x

Return type float

Example

```
>>> x = 3.14
>>> atanherf(x)
  6.157408006068702
```

```
>>> from scipy.special import erf
>>> import numpy as np
>>> np.arctanh(erf(x))
  6.157408006066962
```

```
>>> x = 100
>>> atanherf(x)
  5002.9353661516125
>>> np.arctanh(erf(x))
  inf
```

Notes

Note: We encourage to use this function since it is faster than a possible pure-Python counterpart **and** it implements a series of computational tricks to solve possible numerical instability or precision losses.

1.1.5 References

- D. Dall'Olio, N. Curti, G. Castellani, A. Bazzani, D. Remondini. "Classification of Genome Wide Association data by Belief Propagation Neural network", CCS Italy, 2019.
- C. Baldassi, C. Borgs, J. T. Chayes, A. Ingrosso, C. Lucibello, L. Saglietti, and R. Zecchina. "Unreasonable effectiveness of learning neural networks: From accessible states and robust ensembles to basic algorithmic schemes", Proceedings of the National Academy of Sciences, 113(48):E7655-E7662, 2016

- C. Baldassi, A. Braunstein, N. Brunel, R. Zecchina. "Efficient supervised learning in networks with binary synapses", Proceedings of the National Academy of Sciences, 104(26):11079-11084, 2007.
- A., Braunstein, R. Zecchina. "Learning by message passing in networks of discrete synapses". Physical Review Letters 96(3), 2006.
- C. Baldassi, F. Gerace, C. Lucibello, L. Saglietti, R. Zecchina. "Learning may need only a few bits of synaptic precision", Physical Review E, 93, 2016
- A. Blum, R. L. Rivest. "Training a 3-node neural network is NP-complete", Neural Networks, 1992
- W. Krauth, M. Mezard. "Storage capacity of memory networks with binary coupling", Journal of Physics (France), 1989
- H. Huang, Y. Kabashima. "Origin of the computational hardness for learning with binary synapses", Physical Review E - Statistical, Nonlinear, and Soft Matter Physics, 2014
- C. Baldassi, A. Ingrosso, C. Lucibello, L. Saglietti, R. Zecchina. "Local entropy as a measure for sampling solutions in constraint satisfaction problems", Journal of Statistical Mechanics: Theory and Experiment, 2016
- R. Monasson, R. Zecchina. "Learning and Generalization Theories of Large Committee Machines", Modern Physics Letters B, 1995
- R. Monasson, R. Zecchina. "Weight space structure and internal representations: A direct approach to learning and generalization in multilayer neural networks", Physical Review Letters, 1995
- C. Baldassi, A. Braunstein. "A Max-Sum algorithm for training discrete neural networks", Journal of Statistical Mechanics: Theory and Experiment, 2015
- G. Parisi. "Mean field theory of spin glasses: statics and dynamics", arXiv, 2007
- L. Dall'Asta, A. Ramezanpour, R. Zecchina. "Entropy landscape and non-Gibbs solutions in constraint satisfaction problem", Physical Review E, 2008
- M. Mézard, A. Montanari. "Information, Physics and Computation", Oxford Graduate Texts, 2009
- C. Baldassi, A. Ingrosso, C. Lucibello, L. Saglietti, R. Zecchina. "Subdominant Dense Clusters Allow for Simple Learning and High Computational Performance in Neural Networks with Discrete Synapses", Physical Review Letters, 2015.

Python Module Index

r

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

V

Z