Package 'NeuralEstimators'

January 20, 2025

Title Likelihood-Free Parameter Estimation using Neural Networks

Version 0.1.3

Description An 'R' interface to the 'Julia' package 'NeuralEstimators.jl'. The package facilitates the user-friendly development of neural point estimators, which are neural networks that map data to a point summary of the posterior distribution. These estimators are likelihood-free and amortised, in the sense that, after an initial setup cost, inference from observed data can be made in a fraction of the time required by conventional approaches; see Sainsbury-Dale, Zammit-Mangion, and Huser (2024) <doi:10.1080/00031305.2023.2249522> for further details and an accessible introduction. The package also enables the construction of neural networks that approximate the likelihood-to-evidence ratio in an amortised manner, allowing one to perform inference based on the likelihood function or the entire posterior distribution; see Zammit-Mangion, Sainsbury-Dale, and Huser (2024, Sec. 5.2) <doi:10.48550/arXiv.2404.12484>, and the references therein. The package accommodates any model for which simulation is feasible by allowing the user to implicitly define their model through simulated data.

Maintainer Matthew Sainsbury-Dale <msainsburydale@gmail.com>

License GPL (>= 2)

Encoding UTF-8

RoxygenNote 7.3.2

Imports JuliaConnectoR, magrittr

Suggests dplyr, ggplot2, ggplotify, ggpubr, gridExtra, knitr, rmarkdown, markdown, R.rsp, testthat (>= 3.0.0)

Config/testthat/edition 3

SystemRequirements Julia (>= 1.10)

VignetteBuilder R.rsp

NeedsCompilation no

Author Matthew Sainsbury-Dale [aut, cre]

Repository CRAN

Date/Publication 2025-01-14 13:40:01 UTC

Contents

`NeuralEstimators-package`

NeuralEstimators: Likelihood-Free Parameter Estimation using Neural Networks

---

Description

An 'R' interface to the 'Julia' package 'NeuralEstimators.jl'. The package facilitates the user-friendly development of neural point estimators, which are neural networks that map data to a point summary of the posterior distribution. These estimators are likelihood-free and amortised, in the sense that, after an initial setup cost, inference from observed data can be made in a fraction of the time required by conventional approaches; see Sainsbury-Dale, Zammit-Mangion, and Huser (2024) doi:10.1080/00031305.2023.2249522 for further details and an accessible introduction. The package also enables the construction of neural networks that approximate the likelihood-to-evidence ratio in an amortised manner, allowing one to perform inference based on the likelihood function or the entire posterior distribution; see Zammit-Mangion, Sainsbury-Dale, and Huser (2024, Sec. 5.2) doi:10.48550/arXiv.2404.12484, and the references therein. The package accommodates any model for which simulation is feasible by allowing the user to implicitly define their model through simulated data.

Author(s)

Maintainer: Matthew Sainsbury-Dale <msainsburydale@gmail.com>

`assess` *assess a neural estimator*

Description

assess a neural estimator

Usage

```
assess(
  estimators,
  parameters,
  Z,
  estimator_names = NULL,
  parameter_names = NULL,
  use_gpu = TRUE,
  verbose = TRUE
)
```

Arguments

`estimators`	a list of (neural) estimators
`parameters`	true parameters, stored as a pxK matrix, where p is the number of parameters in the statistical model and K is the number of sampled parameter vectors
`Z`	data simulated conditionally on the `parameters`. If `Z` contains more data sets than parameter vectors, the parameter matrix will be recycled by horizontal concatenation.
`estimator_names`	list of names of the estimators (sensible defaults provided)
`parameter_names`	list of names of the parameters (sensible defaults provided)
`use_gpu`	a boolean indicating whether to use the GPU if it is available (default true)
`verbose`	a boolean indicating whether information should be printed to the console

Value

a list of two data frames: `runtimes`, contains the total time taken for each estimator, while `estimates` is a long-form data frame with columns:

- "estimator"; the name of the estimator
- "parameter"; the name of the parameter
- "truth"; the true value of the parameter

- "estimate"; the estimated value of the parameter
- "m"; the sample size (number of iid replicates)
- "k"; the index of the parameter vector in the test set
- "j"; the index of the data set

See Also

`risk()`, `rmse()`, `bias()`, `plotestimates()`, and `plotdistribution()` for computing various empirical diagnostics and visualisations based on an assessment object

---

`bias` *computes a Monte Carlo approximation of an estimator's bias*

Description

computes a Monte Carlo approximation of an estimator's bias

Usage

```
bias(assessment, ...)
```

Arguments

`assessment`	an object returned by `assess()` (or the `estimates` data frame of this object)
`...`	optional arguments inherited from `risk` (excluding the argument `loss`)

Value

a dataframe giving the estimated bias

See Also

`assess()`, `risk()`, `rmse()`

bootstrap *bootstrap*

Description

Compute bootstrap estimates from a neural estimator

Usage

```
bootstrap(estimator, Z, B = 400, blocks = NULL, use_gpu = TRUE)
```

Arguments

estimator	a neural estimator
Z	either a list of data sets simulated conditionally on the fitted parameters (parametric bootstrap); or a single observed data set containing independent replicates, which will be sampled with replacement B (non-parametric bootstrap)
B	number of non-parametric bootstrap estimates (default 400)
blocks	integer vector specifying the blocks in non-parameteric bootstrap (default NULL). For example, with 5 replicates, the first two corresponding to block 1 and the remaining three corresponding to block 2, blocks should be c(1,1,2,2,2). The bootstrap sampling algorithm aims to produce bootstrap data sets that are of a similar size to Z, but this can only be achieved exactly if all blocks are equal in length.
use_gpu	a boolean indicating whether to use the GPU if it is available (default TRUE)

Value

p × B matrix, where p is the number of parameters in the model and B is the number of bootstrap samples

Examples

```
## Not run:
library("NeuralEstimators")
library("JuliaConnectoR")

## Observed data: m independent replicates of a N(0, 1) random variable
m = 100
Z = t(rnorm(m))

## Construct an (un-trained) neural point estimator
estimator <- initialise_estimator(1, architecture = "MLP")

## Non-parametric bootstrap
bootstrap(estimator, Z = Z)
bootstrap(estimator, Z = Z, blocks = rep(1:5, each = m/5))
```

```
## End(Not run)
```

`encodedata` *encodedata*

Description

for data `Z` with missing (`NA`) entries, returns an augmented data set (U, W) where W encodes the missingness pattern as an indicator vector and U is the original data Z with missing entries replaced by a fixed constant `c`.

The indicator vector W is stored in the second-to-last dimension of `Z`, which should be singleton. If the second-to-last dimension is not singleton, then two singleton dimensions will be added to the array, and W will be stored in the new second-to-last dimension.

Usage

```
encodedata(Z, c = 0)
```

Arguments

`Z`	data containing `NA` entries
`c`	fixed constant with which to replace `NA` entries

Value

Augmented data set (U, W). If `Z` is provided as a list, the return type will be a `JuliaProxy` object; these objects can be indexed in the usual manner (e.g., using `[[`), or converted to an R object using `juliaGet()` (note however that `juliaGet()` can be slow for large data sets).

Examples

```
## Not run:
library("NeuralEstimators")
Z <- matrix(c(1, 2, NA, NA, 5, 6, 7, NA, 9), nrow = 3)
encodedata(Z)
encodedata(list(Z, Z))
## End(Not run)
```

estimate *estimate*

Description

estimate parameters from observed data using a neural estimator

Usage

```
estimate(estimator, Z, theta = NULL, use_gpu = TRUE)
```

Arguments

estimator	a neural estimator
Z	data; format should be amenable to the architecture of estimator
theta	parameter vectors (only for neural estimators that take both the data and parameters as input, e.g., neural ratio estimators)
use_gpu	a boolean indicating whether to use the GPU if it is available (default true)

Value

a matrix of parameter estimates (i.e., estimator applied to Z)

Examples

```
## Not run:
library(NeuralEstimators)
library(JuliaConnectoR)

## Observed data: 100 replicates of a univariate random variable
Z <- matrix(rnorm(100), nrow = 1)

## Construct an (un-trained) point estimator
estimator <- initialise_estimator(1, architecture = "MLP")

## Apply the estimator
estimate(estimator, Z)

## End(Not run)
```

`initialise_estimator` *Initialise a neural estimator*

Description

Helper function for initialising a neural estimator.

The estimator is couched in the DeepSets framework so that it can be applied to data with an arbitrary number of independent replicates (including the special case of a single replicate).

Usage

```
initialise_estimator(
  p,
  architecture,
  d = 1,
  estimator_type = "point",
  depth = 3,
  width = 32,
  activation = "relu",
  activation_output = "identity",
  variance_stabiliser = NULL,
  kernel_size = NULL,
  weight_by_distance = TRUE,
  probs = c(0.025, 0.975)
)
```

Arguments

`p`	number of unknown parameters in the statistical model
`architecture`	a string: for unstructured data, one may use a fully-connected MLP ("MLP"); for data collected over a grid, a convolutional neural network ("CNN"); and for graphical or irregular spatial data, a graphical neural network ("GNN").
`d`	for unstructured multivariate data (i.e., when `architecture = "MLP"`), the dimension of the data (e.g., `d = 3` for trivariate data); otherwise, if `architecture` is `"CNN"` or `"GNN"`, the argument `d` controls the number of input channels (e.g., `d = 1` for univariate spatial processes).
`estimator_type`	the type of estimator; either "point" or "interval".
`depth`	the number of hidden layers. Either a single integer or an integer vector of length two specifying the depth of inner (summary) and outer (inference) network of the DeepSets framework. Since there is an input and an output layer, the total number of layers is `sum(depth) + 2`.
`width`	a single integer or an integer vector of length `sum(depth)` specifying the width (or number of convolutional filters/channels) in each layer.
`activation`	the (non-linear) activation function of each hidden layer. Accepts a string of Julia code (default `"relu"`).

`activation_output`
: the activation function of the output layer layer. Accepts a string of Julia code (default `"identity"`).

`variance_stabiliser`
: a function that will be applied directly to the input, usually to stabilise the variance.: a string ('log' for the natural logarithm, or 'cbrt' for the cube-root function), or a string of Julia code that will be converted to a Julia function using `juliaEval()`.

`kernel_size`
: (applicable only to CNNs) a list of length `depth[1]` containing lists of integers of length D, where D is the dimension of the convolution (e.g., D = 2 for two-dimensional convolution).

`weight_by_distance`
: (applicable only to GNNs) flag indicating whether the estimator will weight by spatial distance; if true (default), a `WeightedGraphConv` layer is used in the propagation module; otherwise, a regular `GraphConv` layer is used.

`probs`
: (applicable only if `estimator_type = "interval"`) probability levels defining the lower and upper endpoints of the posterior credible interval.

Value

the initialised neural estimator, a JuliaProxy object

Examples

```
## Not run:
library("NeuralEstimators")
p = 2
initialise_estimator(p, architecture = "MLP")
initialise_estimator(p, architecture = "GNN")

## 1D convolution
initialise_estimator(p, architecture = "CNN", kernel_size = list(10, 5, 3))

## 2D convolution
initialise_estimator(p, architecture = "CNN",
                     kernel_size = list(c(10, 10), c(5, 5), c(3, 3)))
## End(Not run)
```

`loadstate` *load a saved state of a neural estimator*

Description

Load a saved state of a neural estimator, including both weights and additional model parameters.

Usage

```
loadstate(estimator, filename)
```

Arguments

`estimator`	the neural estimator that we wish to load the state into
`filename`	file (including absolute path) of the neural-network state in a `bson` file

Value

`estimator` updated with the saved state

`loadweights` *load a collection of saved weights of a neural estimator*

Description

This function is deprecated and will be removed in a future version. Please use `loadstate()` instead, which provides more comprehensive functionality.

Usage

```
loadweights(estimator, filename)
```

Arguments

`estimator`	the neural estimator that we wish to load weights into
`filename`	file (including absolute path) of the neural-network weights saved as a `bson` file

Value

`estimator` updated with the saved weights

See Also

`loadstate()`

`mapestimate` *Maximum a posteriori estimation*

Description

Given data `Z`, a neural likelihood-to-evidence-ratio `estimator`, and a `prior`, computes the implied approximate maximum a posteriori (MAP) estimate

If a vector `theta0` of initial parameter estimates is given, the approximate posterior density is maximised by gradient descent. Otherwise, if a matrix of parameters `theta_grid` is given, the approximate posterior density is maximised by grid search.

Usage

```
mapestimate(
  estimator,
  Z,
  prior = NULL,
  theta_grid = NULL,
  theta0 = NULL,
  use_gpu = TRUE
)
```

Arguments

estimator	a neural likelihood-to-evidence-ratio estimator
Z	data; it's format should be amenable to the architecture of estimator
prior	the prior (default uniform), specified as a Julia or R function
theta_grid	a (fine) gridding of the parameter space, given as a matrix with p rows, where p is the number of parameters in the model
theta0	a vector of initial parameter estimates
use_gpu	a boolean indicating whether to use the GPU if it is available (default true)

Value

a p × K matrix of MAP estimates, where p is the number of parameters in the statistical model and K is the number of data sets provided in Z

See Also

sampleposterior(), mlestimate()

---

mlestimate *Maximum likelihood estimation*

Description

Given data Z and a neural likelihood-to-evidence-ratio estimator, computes the implied approximate maximum-likelihood estimate

If a vector theta0 of initial parameter estimates is given, the approximate likelihood is maximised by gradient descent. Otherwise, if a matrix of parameters theta_grid is given, the approximate likelihood is maximised by grid search.

Usage

```
mlestimate(estimator, Z, theta_grid = NULL, theta0 = NULL, use_gpu = TRUE)
```

Arguments

estimator	a neural likelihood-to-evidence-ratio estimator
Z	data; it's format should be amenable to the architecture of estimator
theta_grid	a (fine) gridding of the parameter space, given as a matrix with p rows, where p is the number of parameters in the model
theta0	a vector of initial parameter estimates
use_gpu	a boolean indicating whether to use the GPU if it is available (default true)

Value

a p × K matrix of maximum-likelihood estimates, where p is the number of parameters in the statistical model and K is the number of data sets provided in Z

See Also

sampleposterior(), mapestimate()

---

plotdistribution *Plot the empirical sampling distribution of an estimator.*

Description

Plot the empirical sampling distribution of an estimator.

Usage

```
plotdistribution(
  df,
  type = c("box", "density", "scatter"),
  parameter_labels = NULL,
  estimator_labels = ggplot2::waiver(),
  truth_colour = "black",
  truth_size = 8,
  truth_line_size = NULL,
  pairs = FALSE,
  upper_triangle_plots = NULL,
  legend = TRUE,
  return_list = FALSE,
  flip = FALSE
)
```

Arguments

df	a long form data frame containing fields estimator, parameter, estimate, truth, and a column (e.g., replicate) to uniquely identify each observation.
type	string indicating whether to plot kernel density estimates for each individual parameter (type = "density") or scatter plots for all parameter pairs (type = "scatter").
parameter_labels	a named vector containing parameter labels.
estimator_labels	a named vector containing estimator labels.
truth_colour	the colour used to denote the true parameter value.
truth_size	the size of the point used to denote the true parameter value (applicable only for type = "scatter").
truth_line_size	the size of the cross-hairs used to denote the true parameter value. If NULL (default), the cross-hairs are not plotted. (applicable only for type = "scatter").
pairs	logical; should we combine the scatter plots into a single pairs plot (applicable only for type = "scatter")?
upper_triangle_plots	an optional list of plots to include in the uppertriangle of the pairs plot.
legend	Flag; should we include the legend (only applies when constructing a pairs plot)
return_list	Flag; should the parameters be split into a list?
flip	Flag; should the boxplots be "flipped" using coord_flip() (default FALSE)?

Value

a list of 'ggplot' objects or, if pairs = TRUE, a single 'ggplot'.

Examples

```
## Not run:
# In the following, we have two estimators and, for each parameter, 50 estimates
# from each estimator.

estimators <- c("Estimator 1", "Estimator 2")
estimator_labels <- c("Estimator 1" = expression(hat(theta)[1]("·")),
                      "Estimator 2" = expression(hat(theta)[2]("·")))

# Single parameter:
df <- data.frame(
  estimator = estimators, truth = 0, parameter = "mu",
  estimate  = rnorm(2*50),
  replicate = rep(1:50, each = 2)
)
parameter_labels <- c("mu" = expression(mu))
plotdistribution(df)
plotdistribution(df, type = "density")
```

```
plotdistribution(df, parameter_labels = parameter_labels, estimator_labels = estimator_labels)

# Two parameters:
df <- rbind(df, data.frame(
  estimator = estimators, truth = 1, parameter = "sigma",
  estimate  = rgamma(2*50, shape = 1, rate = 1),
  replicate = rep(1:50, each = 2)
))
parameter_labels <- c(parameter_labels, "sigma" = expression(sigma))
plotdistribution(df, parameter_labels = parameter_labels)
plotdistribution(df, parameter_labels = parameter_labels, type = "density")
plotdistribution(df, parameter_labels = parameter_labels, type = "scatter")

# Three parameters:
df <- rbind(df, data.frame(
  estimator = estimators, truth = 0.25, parameter = "alpha",
  estimate  = 0.5 * runif(2*50),
  replicate = rep(1:50, each = 2)
))
parameter_labels <- c(parameter_labels, "alpha" = expression(alpha))
plotdistribution(df, parameter_labels = parameter_labels)
plotdistribution(df, parameter_labels = parameter_labels, type = "density")
plotdistribution(df, parameter_labels = parameter_labels, type = "scatter")
plotdistribution(df, parameter_labels = parameter_labels, type = "scatter", pairs = TRUE)

# Pairs plot with user-specified plots in the upper triangle:
upper_triangle_plots <- lapply(1:3, function(i) {
  x = rnorm(10)
  y = rnorm(10)
  shape = sample(c("Class 1", "Class 2"), 10, replace = TRUE)
  ggplot() +
    geom_point(aes(x = x, y = y, shape = shape)) +
    labs(shape = "") +
    theme_bw()
})
plotdistribution(
    df,
    parameter_labels = parameter_labels, estimator_labels = estimator_labels,
    type = "scatter", pairs = TRUE, upper_triangle_plots = upper_triangle_plots
    )
## End(Not run)
```

`plotestimates` *Plot estimates vs. true values.*

Description

Plot estimates vs. true values.

Usage

```
plotestimates(
  df,
  estimator_labels = ggplot2::waiver(),
  parameter_labels = NULL
)
```

Arguments

df	a long form data frame containing fields `estimator`, `parameter`, `estimate`, and `truth`.
estimator_labels	a named vector containing estimator labels.
parameter_labels	a named vector containing parameter labels.

Value

a `'ggplot'` of the estimates for each parameter against the true value.

Examples

```
## Not run:
K <- 50
df <- data.frame(
  estimator = c("Estimator 1", "Estimator 2"),
  parameter = rep(c("mu", "sigma"), each = K),
  truth = 1:(2*K),
  estimate = 1:(2*K) + rnorm(4*K)
)
estimator_labels <- c("Estimator 1" = expression(hat(theta)[1]("·")),
                      "Estimator 2" = expression(hat(theta)[2]("·")))
parameter_labels <- c("mu" = expression(mu), "sigma" = expression(sigma))

plotestimates(df,  parameter_labels = parameter_labels, estimator_labels)
## End(Not run)
```

---

risk *computes a Monte Carlo approximation of an estimator's Bayes risk*

Description

computes a Monte Carlo approximation of an estimator's Bayes risk

Usage

```
risk(
  assessment,
  loss = function(x, y) abs(x - y),
  average_over_parameters = FALSE,
  average_over_sample_sizes = TRUE
)
```

Arguments

assessment	an object returned by `assess()` (or the `estimates` data frame of this object)
loss	a binary operator defining the loss function (default absolute-error loss)
average_over_parameters	if `TRUE`, the loss is averaged over all parameters; otherwise (default), the loss is averaged over each parameter separately
average_over_sample_sizes	if `TRUE` (default), the loss is averaged over all sample sizes (the column `m` in `df`); otherwise, the loss is averaged over each sample size separately

Value

a dataframe giving an estimate of the Bayes risk

See Also

`assess()`, `bias()`, `rmse()`

---

rmse *computes a Monte Carlo approximation of an estimator's root-mean-square error (RMSE)*

Description

computes a Monte Carlo approximation of an estimator's root-mean-square error (RMSE)

Usage

```
rmse(assessment, ...)
```

Arguments

assessment	an object returned by `assess()` (or the `estimates` data frame of this object)
...	optional arguments inherited from `risk` (excluding the argument `loss`)

Value

a dataframe giving the estimated RMSE

See Also

assess(), bias(), risk()

---

sampleposterior *sampleposterior*

Description

Given data Z, a neural likelihood-to-evidence-ratio estimator, and a prior, draws samples from the implied approximate posterior distribution

Currently, the sampling algorithm is based on a fine-gridding theta_grid of the parameter space. The approximate posterior density is evaluated over this grid, which is then used to draw samples. This is very effective when making inference with a small number of parameters. For models with a large number of parameters, other sampling algorithms may be needed (please feel free to contact the package maintainer for discussion).

Usage

```
sampleposterior(
  estimator,
  Z,
  theta_grid,
  N = 1000,
  prior = NULL,
  use_gpu = TRUE
)
```

Arguments

estimator	a neural likelihood-to-evidence-ratio estimator
Z	data; it's format should be amenable to the architecture of estimator
theta_grid	a (fine) gridding of the parameter space, given as a matrix with p rows, where p is the number of parameters in the model
N	number of samples to draw (default 1000)
prior	the prior (default uniform), specified as a Julia or R function
use_gpu	a boolean indicating whether to use the GPU if it is available (default true)

Value

a p × N matrix of posterior samples, where p is the number of parameters in the model. If multiple data sets are given in Z, a list of posterior samples will be returned

See Also

mlestimate(), mapestimate()

savestate *save the state of a neural estimator*

Description

save the state of a neural estimator

Usage

```
savestate(estimator, filename)
```

Arguments

estimator	the neural estimator that we wish to save
filename	file in which to save the neural-network state as a bson file

Value

No return value, called for side effects

spatialgraph *spatialgraph*

Description

Constructs a graph object for use in graph neural networks.

Usage

```
spatialgraph(S, Z, isotropic = TRUE, stationary = TRUE, ...)
```

Arguments

S	Spatial locations, provided as: • An n x d matrix when locations are fixed across replicates, where n is the number of spatial locations and d is the spatial dimension (typically d=2). • A list of n_i x d matrices when locations vary across replicates, where each matrix corresponds to the spatial locations of a single replicate.
Z	Spatial data, provided as: • An n x m matrix when locations are fixed, where m is the number of replicates. • A list of n_i vectors when locations vary across replicates, where each vector corresponds to the data for a single replicate.

isotropic	Logical. If TRUE, edge features store the spatial distance (magnitude) between nodes. If FALSE, the spatial displacement or spatial location is stored, depending on the value of stationary.
stationary	Logical. If TRUE, edge features store the spatial displacement (vector difference) between nodes, capturing both magnitude and direction. If FALSE, edge features include the full spatial locations of both nodes.
...	Additional keyword arguments from the Julia function adjacencymatrix() determine the neighborhood of each node, with the default being a randomly selected set of k=30 neighbors within a radius of r=0.15 units.

Value

A GNNGraph (a JuliaProxy object) ready for use in a graph neural network.

See Also

spatialgraphlist() for the vectorised version of this function.

Examples

```
## Not run:
library("NeuralEstimators")

# Number of replicates and spatial dimension
m <- 5
d <- 2

# Spatial locations fixed for all replicates
n <- 100
S <- matrix(runif(n * d), n, d)
Z <- matrix(runif(n * m), n, m)
g <- spatialgraph(S, Z)

# Spatial locations varying between replicates
n <- sample(50:100, m, replace = TRUE)
S <- lapply(n, function(ni) matrix(runif(ni * d), ni, d))
Z <- lapply(n, function(ni) runif(ni))
g <- spatialgraph(S, Z)

## End(Not run)
```

---

spatialgraphlist *spatialgraphlist*

Description

Constructs a list of graph objects for use in a graph neural network.

Usage

```
spatialgraphlist(S, Z, isotropic = TRUE, stationary = TRUE, ...)
```

Arguments

S	A list of spatial locations, where each element of the list is: • An n x d matrix when locations are fixed across replicates in a given data set, where n is the number of spatial locations and d is the spatial dimension (typically d=2). • A list of n_i x d matrices when locations vary across replicates, where each matrix corresponds to the spatial locations of a single replicate.
Z	A list of spatial data, where each element of the list is: • An n x m matrix when locations are fixed, where m is the number of replicates. • A list of n_i vectors when locations vary across replicates, where each vector corresponds to the data for a single replicate.
isotropic	Logical. If TRUE, edge features store the spatial distance (magnitude) between nodes. If FALSE, the spatial displacement or spatial location is stored, depending on the value of stationary.
stationary	Logical. If TRUE, edge features store the spatial displacement (vector difference) between nodes, capturing both magnitude and direction. If FALSE, edge features include the full spatial locations of both nodes.
...	Additional keyword arguments from the Julia function adjacencymatrix() determine the neighborhood of each node, with the default being a randomly selected set of k=30 neighbors within a radius of r=0.15 units.

Details

Additional keyword arguments from the Julia function adjacencymatrix() determine the neighborhood of each node, with the default being a randomly selected set of k=30 neighbors within a radius of r=0.15 units.

Value

A list of GNNGraph objects stored as a JuliaProxy object; this object can be indexed in the usual manner (e.g., using [[), and converted to an R list using a combination of indexing and lapply.

See Also

spatialgraph() for the non-vectorised version of this function.

Examples

```
## Not run:
library("NeuralEstimators")

# Number of data sets, number of replicates in each data set, and spatial dimension
```

```
K <- 15
m <- 5
d <- 2

# Spatial locations fixed for all replicates within a given data set
n <- 100
S <- lapply(1:K, function(k) matrix(runif(n * d), n, d))
Z <- lapply(1:K, function(k) runif(n))
g <- spatialgraphlist(S, Z)

# Spatial locations varying between replicates within a given data set
S <- lapply(1:K, function(k) {
  lapply(1:m, function(i) {
   ni <- sample(50:100, 1)     # Randomly generate the number of locations for each replicate
    matrix(runif(ni * d), ni, d)  # Generate the spatial locations
  })
})
Z <- lapply(1:K, function(k) {
  lapply(1:m, function(i) {
    n <- nrow(S[[k]][[i]])
    runif(n)
  })
})
g <- spatialgraphlist(S, Z)

## End(Not run)
```

`tanhloss` *tanhloss*

Description

For k > 0, defines Julia code that defines the loss function,

$$L(\hat{\theta}, \theta) = \tanh\left(\frac{|\hat{\theta} - \theta|}{k}\right),$$

which approximates the 0-1 loss as k tends to zero.

The resulting string is intended to be used in the function `train`, but can also be converted to a callable function using `juliaEval`.

Usage

```
tanhloss(k)
```

Arguments

k	Positive numeric value that controls the smoothness of the approximation.

Value

String defining the tanh loss function in Julia code.

`train` *Train a neural estimator*

Description

The function caters for different variants of "on-the-fly" simulation. Specifically, a `sampler` can be provided to continuously sample new parameter vectors from the prior, and a `simulator` can be provided to continuously simulate new data conditional on the parameters. If provided with specific sets of parameters (`theta_train` and `theta_val`) and/or data (`Z_train` and `Z_val`), they will be held fixed during training.

Note that using `R` functions to perform "on-the-fly" simulation requires the user to have installed the Julia package `RCall`.

Usage

```
train(
  estimator,
  sampler = NULL,
  simulator = NULL,
  theta_train = NULL,
  theta_val = NULL,
  Z_train = NULL,
  Z_val = NULL,
  m = NULL,
  M = NULL,
  K = 10000,
  xi = NULL,
  loss = "absolute-error",
  learning_rate = 1e-04,
  epochs = 100,
  batchsize = 32,
  savepath = "",
  stopping_epochs = 5,
  epochs_per_Z_refresh = 1,
  epochs_per_theta_refresh = 1,
  simulate_just_in_time = FALSE,
  use_gpu = TRUE,
  verbose = TRUE
)
```

Arguments

estimator	a neural estimator
sampler	a function that takes an integer K, samples K parameter vectors from the prior, and returns them as a pxK matrix
simulator	a function that takes a pxK matrix of parameters and an integer m, and returns K simulated data sets each containing m independent replicates
theta_train	a set of parameters used for updating the estimator using stochastic gradient descent
theta_val	a set of parameters used for monitoring the performance of the estimator during training
Z_train	a simulated data set used for updating the estimator using stochastic gradient descent
Z_val	a simulated data set used for monitoring the performance of the estimator during training
m	vector of sample sizes. If NULL (default), a single neural estimator is trained, with the sample size inferred from Z_val. If m is a vector of integers, a sequence of neural estimators is constructed for each sample size; see the Julia documentation for trainx() for further details
M	deprecated; use m
K	the number of parameter vectors sampled in the training set at each epoch; the size of the validation set is set to K/5.
xi	a list of objects used for data simulation (e.g., distance matrices); if it is provided, the parameter sampler is called as sampler(K, xi).
loss	the loss function: a string ('absolute-error' for mean-absolute-error loss or 'squared-error' for mean-squared-error loss), or a string of Julia code defining the loss function. For some classes of estimators (e.g., QuantileEstimator and RatioEstimator), the loss function does not need to be specified.
learning_rate	the learning rate for the optimiser ADAM (default 1e-3)
epochs	the number of epochs to train the neural network. An epoch is one complete pass through the entire training data set when doing stochastic gradient descent.
batchsize	the batchsize to use when performing stochastic gradient descent, that is, the number of training samples processed between each update of the neural-network parameters.
savepath	path to save the trained estimator and other information; if null (default), nothing is saved. Otherwise, the neural-network parameters (i.e., the weights and biases) will be saved during training as bson files; the risk function evaluated over the training and validation sets will also be saved, in the first and second columns of loss_per_epoch.csv, respectively; the best parameters (as measured by validation risk) will be saved as best_network.bson.
stopping_epochs	cease training if the risk doesn't improve in this number of epochs (default 5).
epochs_per_Z_refresh	integer indicating how often to refresh the training data

epochs_per_theta_refresh
: integer indicating how often to refresh the training parameters; must be a multiple of epochs_per_Z_refresh

simulate_just_in_time
: flag indicating whether we should simulate "just-in-time", in the sense that only a batchsize number of parameter vectors and corresponding data are in memory at a given time

use_gpu
: a boolean indicating whether to use the GPU if one is available

verbose
: a boolean indicating whether information, including empirical risk values and timings, should be printed to the console during training.

Value

a trained neural estimator or, if m is a vector, a list of trained neural estimators

See Also

assess() for assessing an estimator post training, and estimate() for applying an estimator to observed data

Examples

```
## Not run:
# Construct a neural Bayes estimator for replicated univariate Gaussian
# data with unknown mean and standard deviation.

# Load R and Julia packages
library("NeuralEstimators")
library("JuliaConnectoR")
juliaEval("using NeuralEstimators, Flux, Distributions")

# Define the neural-network architecture
estimator <- juliaEval('
 d = 1    # dimension of each replicate
 p = 2    # number of parameters in the model
 w = 32   # width of each layer
 psi = Chain(Dense(d, w, relu), Dense(w, w, relu))
 phi = Chain(Dense(w, w, relu), Dense(w, p))
 deepset = DeepSet(psi, phi)
 estimator = PointEstimator(deepset)
')

# Sampler from the prior
sampler <- function(K) {
  mu    <- rnorm(K)      # Gaussian prior for the mean
  sigma <- rgamma(K, 1)  # Gamma prior for the standard deviation
  theta <- matrix(c(mu, sigma), byrow = TRUE, ncol = K)
  return(theta)
}

# Data simulator
```

```
simulator <- function(theta_set, m) {
  apply(theta_set, 2, function(theta) {
    t(rnorm(m, theta[1], theta[2]))
  }, simplify = FALSE)
}

# Train using fixed parameter and data sets
theta_train <- sampler(10000)
theta_val   <- sampler(2000)
m <- 30 # number of iid replicates
Z_train <- simulator(theta_train, m)
Z_val   <- simulator(theta_val, m)
estimator <- train(estimator,
                   theta_train = theta_train,
                   theta_val = theta_val,
                   Z_train = Z_train,
                   Z_val = Z_val)

# Train using simulation on-the-fly (requires Julia package RCall)
estimator <- train(estimator, sampler = sampler, simulator = simulator, m = m)

##### Simulation on-the-fly using Julia functions ####

# Defining the sampler and simulator in Julia can improve computational
# efficiency by avoiding the overhead of communicating between R and Julia.
# Julia is also fast (comparable to C) and so it can be useful to define
# these functions in Julia when they involve for-loops.

# Parameter sampler
sampler <- juliaEval("
      function sampler(K)
       mu = rand(Normal(0, 1), K)
       sigma = rand(Gamma(1), K)
       theta = hcat(mu, sigma)'
       return theta
      end")

# Data simulator
simulator <- juliaEval("
      function simulator(theta_matrix, m)
       Z = [rand(Normal(theta[1], theta[2]), 1, m) for theta in eachcol(theta_matrix)]
       return Z
      end")

# Train the estimator
estimator <- train(estimator, sampler = sampler, simulator = simulator, m = m)
## End(Not run)
```

Index