Package ‘RSNNS’

November 30, 2023

Maintainer Christoph Bergmeir <c.bergmeir@decsai.ugr.es>

License LGPL (>= 2) | file LICENSE

Title Neural Networks using the Stuttgart Neural Network Simulator (SNNS)

LinkingTo Rcpp

Type Package

LazyLoad yes

Copyright Original SNNS software Copyright (C) 1990-1995 SNNS Group, IPVR, Univ. Stuttgart, FRG; 1996-1998 SNNS Group, WSI, Univ. Tuebingen, FRG. R interface Copyright (C) DiCITS Lab, Sci2s group, DECSAI, University of Granada.

Description The Stuttgart Neural Network Simulator (SNNS) is a library containing many standard implementations of neural networks. This package wraps the SNNS functionality to make it available from within R. Using the 'RSNNS' low-level interface, all of the algorithmic functionality and flexibility of SNNS can be accessed. Furthermore, the package contains a convenient high-level interface, so that the most common neural network topologies and learning algorithms integrate seamlessly into R.

Version 0.4-17

URL https://github.com/cbergmeir/RSNNS

BugReports https://github.com/cbergmeir/RSNNS/issues

MailingList rsnns@googlegroups.com

Date 2023-11-30

Depends R (>= 2.10.0), methods, Rcpp (>= 0.8.5)

Suggests scatterplot3d,NeuralNetTools

Encoding UTF-8

RoxygenNote 7.2.3

NeedsCompilation yes

Author Christoph Bergmeir [aut, cre, cph],
José M. Benítez [ths],
Andreas Zell [ctb] (Part of original SNNS development team),
Niels Mache [ctb] (Part of original SNNS development team),
Günter Mamier [ctb] (Part of original SNNS development team),
Michael Vogt [ctb] (Part of original SNNS development team),
Sven Döring [ctb] (Part of original SNNS development team),
Ralf Hübner [ctb] (Part of original SNNS development team),
Kai-Uwe Herrmann [ctb] (Part of original SNNS development team),
Tobias Soyez [ctb] (Part of original SNNS development team),
Michael Schmalzl [ctb] (Part of original SNNS development team),
Tilman Sommer [ctb] (Part of original SNNS development team),
Artemis Hatzigeorgiou [ctb] (Part of original SNNS development team),
Dietmar Posselt [ctb] (Part of original SNNS development team),
Tobias Schreiner [ctb] (Part of original SNNS development team),
Bernward Kett [ctb] (Part of original SNNS development team),
Martin Reczko [ctb] (Part of original SNNS external contributors),
Martin Riedmiller [ctb] (Part of original SNNS external contributors),
Mark Seemann [ctb] (Part of original SNNS external contributors),
Marcus Ritt [ctb] (Part of original SNNS external contributors),
Jamie DeCoster [ctb] (Part of original SNNS external contributors),
Jochen Biedermann [ctb] (Part of original SNNS external contributors),
Joachim Danz [ctb] (Part of original SNNS development team),
Christian Wehrfritz [ctb] (Part of original SNNS development team),
Patrick Kursawe [ctb] (Contributors to SNNS Version 4.3),
Andre El-Ama [ctb] (Contributors to SNNS Version 4.3)

Repository CRAN

Date/Publication 2023-11-30 05:50:07 UTC

R topics documented:

RSNNS-package . . . 4
analyzeClassification . . . 6
art1 . . . 7
art2 . . . 10
artmap . . . 12
assoz . . . 15
confusionMatrix . . . 17
decodeClassLabels . . . 17
denormalizeData . . . 18
dlvq . . . 19
elman . . . 21
encodeClassLabels . . . 23
exportToSnnsNetFile . . . 24
extractNetInfo . . . 24
getNormParameters . . . 25
getSnnsRDefine . . . 26

getSnnsRFunctionTable . . . 26
inputColumns . . . 27
jordan . . . 27
matrixToActMapList . . . 30
mlp . . . 31
normalizeData . . . 33
normTrainingAndTestSet . . . 34
outputColumns . . . 35
plotActMap . . . 36
plotIterativeError . . . 36
plotRegressionError . . . 37
plotROC . . . 37
predict.rsnns . . . 38
print.rsnns . . . 38
rbf . . . 39
rbfDDA . . . 41
readPatFile . . . 43
readResFile . . . 43
resolveSnnsRDefine . . . 44
rsnnsObjectFactory . . . 44
savePatFile . . . 46
setSnnsRSeedValue . . . 46
snnsData . . . 47
SnnsR-class . . . 47
SnnsRObjectFactory . . . 49
SnnsRObjectMethodCaller . . . 50
SnnsRObject$createNet . . . 51
SnnsRObject$createPatSet . . . 52
SnnsRObject$extractNetInfo . . . 52
SnnsRObject$extractPatterns . . . 53
SnnsRObject$getAllHiddenUnits . . . 53
SnnsRObject$getAllInputUnits . . . 54
SnnsRObject$getAllOutputUnits . . . 54
SnnsRObject$getAllUnits . . . 55
SnnsRObject$getAllUnitsTType . . . 55
SnnsRObject$getCompleteWeightMatrix . . . 56
SnnsRObject$getInfoHeader . . . 56
SnnsRObject$getSiteDefinitions . . . 57
SnnsRObject$getTypeDefinitions . . . 57
SnnsRObject$getUnitDefinitions . . . 58
SnnsRObject$getUnitsByName . . . 58
SnnsRObject$getWeightMatrix . . . 59
SnnsRObject$initializeNet . . . 59
SnnsRObject$predictCurrPatSet . . . 60
SnnsRObject$resetRSNNS . . . 60
SnnsRObject$setTTypeUnitsActFunc . . . 61
SnnsRObject$setUnitDefaults . . . 61
SnnsRObject$somPredictComponentMaps . . . 62

SnnsRObject$somPredictCurrPatSetWinners . . . 63
SnnsRObject$somPredictCurrPatSetWinnersSpanTree . . . 64
SnnsRObject$train . . . 64
SnnsRObject$whereAreResults . . . 66
som . . . 66
splitForTrainingAndTest . . . 69
summary.rsnns . . . 70
toNumericClassLabels . . . 71
train . . . 71
vectorToActMap . . . 72
weightMatrix . . . 73

Index **74**

`RSNNS-package` *Getting started with the RSNNS package*

Description

The Stuttgart Neural Network Simulator (SNNS) is a library containing many standard implementations of neural networks. This package wraps the SNNS functionality to make it available from within R.

Details

If you have problems using RSNNS, find a bug, or have suggestions, please do not write to the general R lists or contact the authors of the original SNNS software. Instead, you should: File an issue on github (bugs/suggestions), Ask your question on Stackoverflow under the tag RSNNS, or write to the mailing list (rsnns@googlegroups.com). If all that fails, then you can also contact the maintainer directly by email.

If you use the package, please cite the following work in your publications:

Bergmeir, C. and Benítez, J.M. (2012), Neural Networks in R Using the Stuttgart Neural Network Simulator: RSNNS. Journal of Statistical Software, 46(7), 1-26.

The package has a hierarchical architecture with three levels:

- RSNNS high-level api (rsnns)
- RSNNS low-level api (SnnsR)
- The api of our C++ port of SNNS (SnnsCLib)

Many demos for using both low-level and high-level api of the package are available. To get a list of them, type:

`library(RSNNS)`

`demo()`

It is a good idea to start with the demos of the high-level api (which is much more convenient to use). E.g., to access the iris classification demo type:

`demo(iris)`

or for the laser regression demo type:

```
demo(laser)
```

As the high-level api is already quite powerful and flexible, you'll most probably normally end up using one of the functions: mlp, dlvq, rbf, rbfDDA, elman, jordan, som, art1, art2, artmap, or assoz, with some pre- and postprocessing. These S3 classes are all subclasses of rsnns.

You might also want to have a look at the original SNNS program and the SNNS User Manual 4.2, especially pp 67-87 for explications on all the parameters of the learning functions, and pp 145-215 for detailed (theoretical) explications of the methods and advice on their use. And, there is also the javaNNS, the sucessor of SNNS from the original authors. It makes the C core functionality available from a Java GUI.

Demos ending with "SnnsR" show the use of the low-level api. If you want to do special things with neural networks that are currently not implemented in the high-level api, you can see in this demos how to do it. Many demos are present both as high-level and low-level versions.

The low-level api consists mainly of the class SnnsR-class, which internally holds a pointer to a C++ object of the class SnnsCLib, i.e., an instance of the SNNS kernel. The class furthermore implements a calling mechanism for methods of the SnnsCLib object, so that they can be called conveniently using the "$"-operator. This calling mechanism also allows for transparent masking of methods or extending the kernel with new methods from within R. See $,SnnsR-method. R-functions that are added by RSNNS to the kernel are documented in this manual under topics beginning with SnnsRObject$. Documentation of the original SNNS kernel user interface functions can be found in the SNNS User Manual 4.2 pp 290-314. A call to, e.g., the SNNS kernel function krui_getNoOfUnits(...) can be done with SnnsRObject$getNoOfUnits(...). However, a few functions were excluded from the wrapping for various reasons. Fur more details and other known issues see the file /inst/doc/KnownIssues.

Another nice tool is the NeuralNetTools package, that can be used to visualize and analyse the networks generated with RSNNS.

Most of the example data included in SNNS is also present in this package, see snnsData.

A comprehensive report with many examples showing the usage of RSNNS, developed by Seymour Shlien, is available here:

https://ifdo.ca/~seymour/R/

Author(s)

Christoph Bergmeir <c.bergmeir@decsai.ugr.es>

and José M. Benítez <j.m.benitez@decsai.ugr.es>

DiCITS Lab, Sci2s group, DECSAI, University of Granada.

http://dicits.ugr.es, https://sci2s.ugr.es

References

Bergmeir, C. and Benítez, J.M. (2012), 'Neural Networks in R Using the Stuttgart Neural Network Simulator: RSNNS', Journal of Statistical Software, 46(7), 1-26.

General neural network literature:

Bishop, C. M. (2003), Neural networks for pattern recognition, University Press, Oxford.

Haykin, S. S. (1999), Neural networks :a comprehensive foundation, Prentice Hall, Upper Saddle River, NJ.

Kriesel, D. (2007), A Brief Introduction to Neural Networks. http://www.dkriesel.com

Ripley, B. D. (2007), Pattern recognition and neural networks, Cambridge University Press, Cambridge.

Rojas, R. (1996), Neural networks :a systematic introduction, Springer-Verlag, Berlin.

Rumelhart, D. E.; Clelland, J. L. M. & Group, P. R. (1986), Parallel distributed processing :explorations in the microstructure of cognition, Mit, Cambridge, MA etc..

Literature on the original SNNS software:

Zell, A. et al. (1998), 'SNNS Stuttgart Neural Network Simulator User Manual, Version 4.2', IPVR, University of Stuttgart and WSI, University of Tübingen. https://www.ra.cs.uni-tuebingen.de/SNNS/welcome.html

javaNNS, the sucessor of the original SNNS with a Java GUI: https://github.com/mwri/javanns

Zell, A. (1994), Simulation Neuronaler Netze, Addison-Wesley.

Other resources:

A function to plot networks from the `mlp` function: https://beckmw.wordpress.com/2013/11/14/visualizing-neural-networks-in-r-update/

See Also

`mlp`, `dlvq`, `rbf`, `rbfDDA`, `elman`, `jordan`, `som`, `art1`, `art2`, `artmap`, `assoz`

`analyzeClassification` *Converts continuous outputs to class labels*

Description

This function converts the continuous outputs to binary outputs that can be used for classification. The two methods 402040, and winner-takes-all (WTA), are implemented as described in the SNNS User Manual 4.2.

Usage

```
analyzeClassification(y, method = "WTA", l = 0, h = 0)
```

Arguments

`y`	inputs
`method`	"WTA" or "402040"
`l`	lower bound, e.g. in 402040: l=0.4
`h`	upper bound, e.g. in 402040: h=0.6

Details

The following text is an edited citation from the SNNS User Manual 4.2 (pp 269):

402040 A pattern is recognized as classified correctly, if (i) the output of exactly one output unit is >= h (ii) the teaching output of this unit is the maximum teaching output (> 0) of the pattern (iii) the output of all other output units is <= l.

A pattern is recognized as classified incorrectly, if (i) and (iii) hold as above, but for (ii) holds that the teaching output is *not* the maximum teaching output of the pattern or there is no teaching output > 0.

A pattern is recognized as unclassified in all other cases.

The method derives its name from the commonly used default values l = 0.4, h = 0.6.

WTA A pattern is recognized as classified correctly, if (i) there is an output unit with the value greater than the output value of all other output units (this output value is supposed to be a) (ii) a > h (iii) the teaching output of this unit is the maximum teaching output of the pattern (> 0) (iv) the output of all other units is < a - l.

A pattern is recognized as classified incorrectly, if (i), (ii), and (iv) hold as above, but for (iii) holds that the teaching output of this unit is *not* the maximum teaching output of the pattern or there is no teaching output > 0.

A pattern is recognized as unclassified in all other cases.

Commonly used default values for this method are: l = 0.0, h = 0.0.

Value

the position of the winning unit (i.e., the winning class), or zero, if no classification was done.

References

Zell, A. et al. (1998), 'SNNS Stuttgart Neural Network Simulator User Manual, Version 4.2', IPVR, University of Stuttgart and WSI, University of Tübingen. https://www.ra.cs.uni-tuebingen.de/SNNS/welcome.html

See Also

encodeClassLabels

`art1` *Create and train an art1 network*

Description

Adaptive resonance theory (ART) networks perform clustering by finding prototypes. They are mainly designed to solve the stability/plasticity dilemma (which is one of the central problems in neural networks) in the following way: new input patterns may generate new prototypes (plasticity), but patterns already present in the net (represented by their prototypes) are only altered by similar new patterns, not by others (stability). ART1 is for binary inputs only, if you have real-valued input, use art2 instead.

Learning in an ART network works as follows: A new input is intended to be classified according to the prototypes already present in the net. The similarity between the input and all prototypes is calculated. The most similar prototype is the *winner*. If the similarity between the input and the winner is high enough (defined by a *vigilance parameter*), the winner is adapted to make it more similar to the input. If similarity is not high enough, a new prototype is created. So, at most the winner is adapted, all other prototypes remain unchanged.

Usage

```
art1(x, ...)

## Default S3 method:
art1(
  x,
  dimX,
  dimY,
  f2Units = nrow(x),
  maxit = 100,
  initFunc = "ART1_Weights",
  initFuncParams = c(1, 1),
  learnFunc = "ART1",
  learnFuncParams = c(0.9, 0, 0),
  updateFunc = "ART1_Stable",
  updateFuncParams = c(0),
  shufflePatterns = TRUE,
  ...
)
```

Arguments

`x`	a matrix with training inputs for the network
`...`	additional function parameters (currently not used)
`dimX`	x dimension of inputs and outputs
`dimY`	y dimension of inputs and outputs
`f2Units`	controls the number of clusters assumed to be present
`maxit`	maximum of iterations to learn
`initFunc`	the initialization function to use
`initFuncParams`	the parameters for the initialization function
`learnFunc`	the learning function to use
`learnFuncParams`	the parameters for the learning function
`updateFunc`	the update function to use
`updateFuncParams`	the parameters for the update function
`shufflePatterns`	should the patterns be shuffled?

Details

The architecture of an ART network is the following: ART is based on the more general concept of *competitive learning*. The networks have two fully connected layers (in both directions), the input/comparison layer and the recognition layer. They propagate activation back and forth (resonance). The units in the recognition layer have lateral inhibition, so that they show a winner-takes-all behaviour, i.e., the unit that has the highest activation inhibits activation of other units, so that after a few cycles its activation will converge to one, whereas the other units activations converge to zero. ART stabilizes this general learning mechanism by the presence of some special units. For details refer to the referenced literature.

The default initialization function, `ART1_Weights`, is the only one suitable for ART1 networks. It has two parameters, which are explained in the SNNS User Manual pp.189. A default of 1.0 for both is usually fine. The only learning function suitable for ART1 is `ART1`. Update functions are `ART1_Stable` and `ART1_Synchronous`. The difference between the two is that the first one updates until the network is in a stable state, and the latter one only performs one update step. Both the learning function and the update functions have one parameter, the vigilance parameter.

In its current implementation, the network has two-dimensional input. The matrix `x` contains all (one dimensional) input patterns. Internally, every one of these patterns is converted to a two-dimensional pattern using parameters `dimX` and `dimY`. The parameter `f2Units` controls the number of units in the recognition layer, and therewith the maximal amount of clusters that are assumed to be present in the input patterns.

A detailed description of the theory and the parameters is available from the SNNS documentation and the other referenced literature.

Value

an `rsnns` object. The `fitted.values` member of the object contains a list of two-dimensional activation patterns.

References

Carpenter, G. A. & Grossberg, S. (1987), 'A massively parallel architecture for a self-organizing neural pattern recognition machine', Comput. Vision Graph. Image Process. 37, 54–115.

Grossberg, S. (1988), Adaptive pattern classification and universal recoding. I.: parallel development and coding of neural feature detectors, MIT Press, Cambridge, MA, USA, chapter I, pp. 243–258.

Herrmann, K.-U. (1992), 'ART – Adaptive Resonance Theory – Architekturen, Implementierung und Anwendung', Master's thesis, IPVR, University of Stuttgart. (in German)

Zell, A. et al. (1998), 'SNNS Stuttgart Neural Network Simulator User Manual, Version 4.2', IPVR, University of Stuttgart and WSI, University of Tübingen. https://www.ra.cs.uni-tuebingen.de/SNNS/welcome.html

Zell, A. (1994), Simulation Neuronaler Netze, Addison-Wesley. (in German)

See Also

`art2`, `artmap`

Examples

```
## Not run: demo(art1_letters)
## Not run: demo(art1_lettersSnnsR)


data(snnsData)
patterns <- snnsData$art1_letters.pat

inputMaps <- matrixToActMapList(patterns, nrow=7)
par(mfrow=c(3,3))
for (i in 1:9) plotActMap(inputMaps[[i]])

model <- art1(patterns, dimX=7, dimY=5)
encodeClassLabels(model$fitted.values)
```

---

art2 *Create and train an art2 network*

Description

ART2 is very similar to ART1, but for real-valued input. See art1 for more information. Opposed to the ART1 implementation, the ART2 implementation does not assume two-dimensional input.

Usage

```
art2(x, ...)

## Default S3 method:
art2(
  x,
  f2Units = 5,
  maxit = 100,
  initFunc = "ART2_Weights",
  initFuncParams = c(0.9, 2),
  learnFunc = "ART2",
  learnFuncParams = c(0.98, 10, 10, 0.1, 0),
  updateFunc = "ART2_Stable",
  updateFuncParams = c(0.98, 10, 10, 0.1, 0),
  shufflePatterns = TRUE,
  ...
)
```

Arguments

x	a matrix with training inputs for the network
...	additional function parameters (currently not used)
f2Units	controls the number of clusters assumed to be present

`maxit`	maximum of iterations to learn
`initFunc`	the initialization function to use
`initFuncParams`	the parameters for the initialization function
`learnFunc`	the learning function to use
`learnFuncParams`	the parameters for the learning function
`updateFunc`	the update function to use
`updateFuncParams`	the parameters for the update function
`shufflePatterns`	should the patterns be shuffled?

Details

As comparison of real-valued vectors is more difficult than comparison of binary vectors, the comparison layer is more complex in ART2, and actually consists of three layers. With a more complex comparison layer, also other parts of the network enhance their complexity. In SNNS, this enhanced complexity is reflected by the presence of more parameters in initialization-, learning-, and update function.

In analogy to the implementation of ART1, there are one initialization function, one learning function and two update functions suitable for ART2. The learning and update functions have five parameters, the initialization function has two parameters. For details see the SNNS User Manual, p. 67 and pp. 192.

Value

an `rsnns` object. The `fitted.values` member contains the activation patterns for all inputs.

References

Carpenter, G. A. & Grossberg, S. (1987), 'ART 2: self-organization of stable category recognition codes for analog input patterns', Appl. Opt. 26(23), 4919–4930.

Grossberg, S. (1988), Adaptive pattern classification and universal recoding. I.: parallel development and coding of neural feature detectors, MIT Press, Cambridge, MA, USA, chapter I, pp. 243–258.

Herrmann, K.-U. (1992), 'ART – Adaptive Resonance Theory – Architekturen, Implementierung und Anwendung', Master's thesis, IPVR, University of Stuttgart. (in German)

Zell, A. et al. (1998), 'SNNS Stuttgart Neural Network Simulator User Manual, Version 4.2', IPVR, University of Stuttgart and WSI, University of Tübingen. `https://www.ra.cs.uni-tuebingen.de/SNNS/welcome.html`

Zell, A. (1994), Simulation Neuronaler Netze, Addison-Wesley. (in German)

See Also

`art1`, `artmap`

Examples

```
## Not run: demo(art2_tetra)
## Not run: demo(art2_tetraSnnsR)


data(snnsData)
patterns <- snnsData$art2_tetra_med.pat

model <- art2(patterns, f2Units=5, learnFuncParams=c(0.99, 20, 20, 0.1, 0),
                  updateFuncParams=c(0.99, 20, 20, 0.1, 0))
model

testPatterns <- snnsData$art2_tetra_high.pat
predictions <- predict(model, testPatterns)

## Not run: library(scatterplot3d)

## Not run: par(mfrow=c(2,2))
## Not run: scatterplot3d(patterns, pch=encodeClassLabels(model$fitted.values))
## Not run: scatterplot3d(testPatterns, pch=encodeClassLabels(predictions))
```

`artmap` *Create and train an artmap network*

Description

An ARTMAP performs supervised learning. It consists of two coupled ART networks. In theory, these could be ART1, ART2, or others. However, in SNNS ARTMAP is implemented for ART1 only. So, this function is to be used with binary input. As explained in the description of `art1`, ART aims at solving the stability/plasticity dilemma. So the advantage of ARTMAP is that it is a supervised learning mechanism that guarantees stability.

Usage

```
artmap(x, ...)

## Default S3 method:
artmap(
  x,
  nInputsTrain,
  nInputsTargets,
  nUnitsRecLayerTrain,
  nUnitsRecLayerTargets,
  maxit = 1,
  nRowInputsTrain = 1,
  nRowInputsTargets = 1,
  nRowUnitsRecLayerTrain = 1,
  nRowUnitsRecLayerTargets = 1,
```

```
  initFunc = "ARTMAP_Weights",
  initFuncParams = c(1, 1, 1, 1, 0),
  learnFunc = "ARTMAP",
  learnFuncParams = c(0.8, 1, 1, 0, 0),
  updateFunc = "ARTMAP_Stable",
  updateFuncParams = c(0.8, 1, 1, 0, 0),
  shufflePatterns = TRUE,
  ...
)
```

Arguments

`x`	a matrix with training inputs and targets for the network
`...`	additional function parameters (currently not used)
`nInputsTrain`	the number of columns of the matrix that are training input
`nInputsTargets`	the number of columns that are target values
`nUnitsRecLayerTrain`	number of units in the recognition layer of the training data ART network
`nUnitsRecLayerTargets`	number of units in the recognition layer of the target data ART network
`maxit`	maximum of iterations to perform
`nRowInputsTrain`	number of rows the training input units are to be organized in (only for visualization purposes of the net in the original SNNS software)
`nRowInputsTargets`	same, but for the target value input units
`nRowUnitsRecLayerTrain`	same, but for the recognition layer of the training data ART network
`nRowUnitsRecLayerTargets`	same, but for the recognition layer of the target data ART network
`initFunc`	the initialization function to use
`initFuncParams`	the parameters for the initialization function
`learnFunc`	the learning function to use
`learnFuncParams`	the parameters for the learning function
`updateFunc`	the update function to use
`updateFuncParams`	the parameters for the update function
`shufflePatterns`	should the patterns be shuffled?

Details

See also the details section of art1. The two ART1 networks are connected by a *map field*. The input of the first ART1 network is the training input, the input of the second network are the target values, the teacher signals. The two networks are often called ARTa and ARTb, we call them here training data network and target data network.

In analogy to the ART1 and ART2 implementations, there are one initialization function, one learning function, and two update functions present that are suitable for ARTMAP. The parameters are basically as in ART1, but for two networks. The learning function and the update functions have 3 parameters, the vigilance parameters of the two ART1 networks and an additional vigilance parameter for inter ART reset control. The initialization function has four parameters, two for every ART1 network.

A detailed description of the theory and the parameters is available from the SNNS documentation and the other referenced literature.

Value

an rsnns object. The `fitted.values` member of the object contains a list of two-dimensional activation patterns.

References

Carpenter, G. A.; Grossberg, S. & Reynolds, J. H. (1991), 'ARTMAP: Supervised real-time learning and classification of nonstationary data by a self-organizing neural network', Neural Networks 4(5), 565–588.

Grossberg, S. (1988), Adaptive pattern classification and universal recoding. I.: parallel development and coding of neural feature detectors, MIT Press, Cambridge, MA, USA, chapter I, pp. 243–258.

Herrmann, K.-U. (1992), 'ART – Adaptive Resonance Theory – Architekturen, Implementierung und Anwendung', Master's thesis, IPVR, University of Stuttgart. (in German)

Zell, A. et al. (1998), 'SNNS Stuttgart Neural Network Simulator User Manual, Version 4.2', IPVR, University of Stuttgart and WSI, University of Tübingen. https://www.ra.cs.uni-tuebingen.de/SNNS/welcome.html

Zell, A. (1994), Simulation Neuronaler Netze, Addison-Wesley. (in German)

See Also

art1, art2

Examples

```
## Not run: demo(artmap_letters)
## Not run: demo(artmap_lettersSnnsR)


data(snnsData)
trainData <- snnsData$artmap_train.pat
testData <- snnsData$artmap_test.pat
```

```
model <- artmap(trainData, nInputsTrain=70, nInputsTargets=5,
                nUnitsRecLayerTrain=50, nUnitsRecLayerTargets=26)
model$fitted.values

predict(model, testData)
```

---

assoz *Create and train an (auto-)associative memory*

Description

The autoassociative memory performs clustering by finding a prototype to the given input. The implementation assumes two-dimensional input and output (cf. art1).

Usage

```
assoz(x, ...)

## Default S3 method:
assoz(
  x,
  dimX,
  dimY,
  maxit = 100,
  initFunc = "RM_Random_Weights",
  initFuncParams = c(1, -1),
  learnFunc = "RM_delta",
  learnFuncParams = c(0.01, 100, 0, 0, 0),
  updateFunc = "Auto_Synchronous",
  updateFuncParams = c(50),
  shufflePatterns = TRUE,
  ...
)
```

Arguments

x	a matrix with training inputs for the network
...	additional function parameters (currently not used)
dimX	x dimension of inputs and outputs
dimY	y dimension of inputs and outputs
maxit	maximum of iterations to learn
initFunc	the initialization function to use
initFuncParams	the parameters for the initialization function
learnFunc	the learning function to use

`learnFuncParams`
: the parameters for the learning function

`updateFunc`
: the update function to use

`updateFuncParams`
: the parameters for the update function

`shufflePatterns`
: should the patterns be shuffled?

Details

The default initialization and update functions are the only ones suitable for this kind of network. The update function takes one parameter, which is the number of iterations that will be performed. The default of 50 usually does not have to be modified. For learning, `RM_delta` and `Hebbian` functions can be used, though the first one usually performs better.

A more detailed description of the theory and the parameters is available from the SNNS documentation and the other referenced literature.

Value

an `rsnns` object. The `fitted.values` member contains the activation patterns for all inputs.

References

Palm, G. (1980), 'On associative memory', Biological Cybernetics 36, 19-31.

Rojas, R. (1996), Neural networks :a systematic introduction, Springer-Verlag, Berlin.

Zell, A. et al. (1998), 'SNNS Stuttgart Neural Network Simulator User Manual, Version 4.2', IPVR, University of Stuttgart and WSI, University of Tübingen. https://www.ra.cs.uni-tuebingen.de/SNNS/welcome.html

See Also

`art1`, `art2`

Examples

```
## Not run: demo(assoz_letters)
## Not run: demo(assoz_lettersSnnsR)


data(snnsData)
patterns <- snnsData$art1_letters.pat

model <- assoz(patterns, dimX=7, dimY=5)

actMaps <- matrixToActMapList(model$fitted.values, nrow=7)

par(mfrow=c(3,3))
for (i in 1:9) plotActMap(actMaps[[i]])
```

`confusionMatrix` *Computes a confusion matrix*

Description

The confusion matrix shows how many times a pattern with the real class x was classified as class y. A perfect method should result in a diagonal matrix. All values not on the diagonal are errors of the method.

Usage

```
confusionMatrix(targets, predictions)
```

Arguments

`targets`	the known, correct target values
`predictions`	the corresponding predictions of a method for the targets

Details

If the class labels are not already encoded, they are encoded using `encodeClassLabels` (with default values).

Value

the confusion matrix

`decodeClassLabels` *Decode class labels to a binary matrix*

Description

This method decodes class labels from a numerical or levels vector to a binary matrix, i.e., it converts the input vector to a binary matrix.

Usage

```
decodeClassLabels(x, valTrue = 1, valFalse = 0)
```

Arguments

`x`	class label vector
`valTrue`	see Details paragraph
`valFalse`	see Details paragraph

Details

In the matrix, the value `valTrue` (e.g. 1) is present exactly in the column given by the value in the input vector, and the value `valFalse` (e.g. 0) in the other columns. The number of columns of the resulting matrix depends on the number of unique labels found in the vector. E.g. the input c(1, 3, 2, 3) will result in an output matrix with rows: 100 001 010 001

Value

a matrix containing the decoded class labels

Author(s)

The implementation is a slightly modified version of the function `class.ind` from the `nnet` package of Brian Ripley.

References

Venables, W. N. and Ripley, B. D. (2002), 'Modern Applied Statistics with S', Springer-Verlag.

Examples

```
decodeClassLabels(c(1,3,2,3))
decodeClassLabels(c("r","b","b","r", "g", "g"))

data(iris)
decodeClassLabels(iris[,5])
```

`denormalizeData` *Revert data normalization*

Description

Column-wise normalization of the input matrix is reverted, using the given parameters.

Usage

```
denormalizeData(x, normParams)
```

Arguments

`x`	input data
`normParams`	the parameters generated by an earlier call to `normalizeData` that will be used for reverting normalization

Details

The input matrix is column-wise denormalized using the parameters given by `normParams`. E.g., if `normParams` contains mean and sd for every column, the values are multiplied by sd and the mean is added

Value

column-wise denormalized input

See Also

normalizeData, getNormParameters

Examples

```
data(iris)
values <- normalizeData(iris[,1:4])
denormalizeData(values, getNormParameters(values))
```

---

dlvq *Create and train a dlvq network*

Description

Dynamic learning vector quantization (DLVQ) networks are similar to self-organizing maps (SOM, som). But they perform supervised learning and lack a neighborhood relationship between the prototypes.

Usage

```
dlvq(x, ...)

## Default S3 method:
dlvq(
  x,
  y,
  initFunc = "DLVQ_Weights",
  initFuncParams = c(1, -1),
  learnFunc = "Dynamic_LVQ",
  learnFuncParams = c(0.03, 0.03, 10),
  updateFunc = "Dynamic_LVQ",
  updateFuncParams = c(0),
  shufflePatterns = TRUE,
  ...
)
```

Arguments

x	a matrix with training inputs for the network
...	additional function parameters (currently not used)
y	the corresponding target values
initFunc	the initialization function to use

initFuncParams the parameters for the initialization function

learnFunc the learning function to use

learnFuncParams
the parameters for the learning function

updateFunc the update function to use

updateFuncParams
the parameters for the update function

shufflePatterns
should the patterns be shuffled?

Details

The input data has to be normalized in order to use DLVQ.

Learning in DLVQ: For each class, a mean vector (prototype) is calculated and stored in a (newly generated) hidden unit. Then, the net is used to classify every pattern by using the nearest prototype. If a pattern gets misclassified as class y instead of class x, the prototype of class y is moved away from the pattern, and the prototype of class x is moved towards the pattern. This procedure is repeated iteratively until no more changes in classification take place. Then, new prototypes are introduced in the net per class as new hidden units, and initialized by the mean vector of misclassified patterns in that class.

Network architecture: The network only has one hidden layer, containing one unit for each prototype. The prototypes/hidden units are also called codebook vectors. Because SNNS generates the units automatically, and does not need their number to be specified in advance, the procedure is called *dynamic* LVQ in SNNS.

The default initialization, learning, and update functions are the only ones suitable for this kind of network. The three parameters of the learning function specify two learning rates (for the cases correctly/uncorrectly classified), and the number of cycles the net is trained before mean vectors are calculated.

A detailed description of the theory and the parameters is available, as always, from the SNNS documentation and the other referenced literature.

Value

an rsnns object. The fitted.values member contains the activation patterns for all inputs.

References

Kohonen, T. (1988), Self-organization and associative memory, Vol. 8, Springer-Verlag.

Zell, A. et al. (1998), 'SNNS Stuttgart Neural Network Simulator User Manual, Version 4.2', IPVR, University of Stuttgart and WSI, University of Tübingen. https://www.ra.cs.uni-tuebingen.de/SNNS/welcome.html

Zell, A. (1994), Simulation Neuronaler Netze, Addison-Wesley. (in German)

Examples

```
## Not run: demo(dlvq_ziff)
## Not run: demo(dlvq_ziffSnnsR)


data(snnsData)
dataset <- snnsData$dlvq_ziff_100.pat

inputs <- dataset[,inputColumns(dataset)]
outputs <- dataset[,outputColumns(dataset)]

model <- dlvq(inputs, outputs)

fitted(model) == outputs
mean(fitted(model) - outputs)
```

---

elman *Create and train an Elman network*

Description

Elman networks are partially recurrent networks and similar to Jordan networks (function `jordan`). For details, see explanations there.

Usage

```
elman(x, ...)

## Default S3 method:
elman(
  x,
  y,
  size = c(5),
  maxit = 100,
  initFunc = "JE_Weights",
  initFuncParams = c(1, -1, 0.3, 1, 0.5),
  learnFunc = "JE_BP",
  learnFuncParams = c(0.2),
  updateFunc = "JE_Order",
  updateFuncParams = c(0),
  shufflePatterns = FALSE,
  linOut = TRUE,
  outContext = FALSE,
  inputsTest = NULL,
  targetsTest = NULL,
  ...
)
```

Arguments

`x`	a matrix with training inputs for the network
`...`	additional function parameters (currently not used)
`y`	the corresponding targets values
`size`	number of units in the hidden layer(s)
`maxit`	maximum of iterations to learn
`initFunc`	the initialization function to use
`initFuncParams`	the parameters for the initialization function
`learnFunc`	the learning function to use
`learnFuncParams`	the parameters for the learning function
`updateFunc`	the update function to use
`updateFuncParams`	the parameters for the update function
`shufflePatterns`	should the patterns be shuffled?
`linOut`	sets the activation function of the output units to linear or logistic
`outContext`	if TRUE, the context units are also output units (untested)
`inputsTest`	a matrix with inputs to test the network
`targetsTest`	the corresponding targets for the test input

Details

Learning in Elman networks: Same as in Jordan networks (see `jordan`).

Network architecture: The difference between Elman and Jordan networks is that in an Elman network the context units get input not from the output units, but from the hidden units. Furthermore, there is no direct feedback in the context units. In an Elman net, the number of context units and hidden units has to be the same. The main advantage of Elman nets is that the number of context units is not directly determined by the output dimension (as in Jordan nets), but by the number of hidden units, which is more flexible, as it is easy to add/remove hidden units, but not output units.

A detailed description of the theory and the parameters is available, as always, from the SNNS documentation and the other referenced literature.

Value

an `rsnns` object.

References

Elman, J. L. (1990), 'Finding structure in time', Cognitive Science 14(2), 179–211.

Zell, A. et al. (1998), 'SNNS Stuttgart Neural Network Simulator User Manual, Version 4.2', IPVR, University of Stuttgart and WSI, University of Tübingen. https://www.ra.cs.uni-tuebingen.de/SNNS/welcome.html

Zell, A. (1994), Simulation Neuronaler Netze, Addison-Wesley. (in German)

See Also

jordan

Examples

```
## Not run: demo(iris)
## Not run: demo(laser)
## Not run: demo(eight_elman)
## Not run: demo(eight_elmanSnnsR)


data(snnsData)
inputs <- snnsData$eight_016.pat[,inputColumns(snnsData$eight_016.pat)]
outputs <- snnsData$eight_016.pat[,outputColumns(snnsData$eight_016.pat)]

par(mfrow=c(1,2))

modelElman <- elman(inputs, outputs, size=8, learnFuncParams=c(0.1), maxit=1000)
modelElman
modelJordan <- jordan(inputs, outputs, size=8, learnFuncParams=c(0.1), maxit=1000)
modelJordan

plotIterativeError(modelElman)
plotIterativeError(modelJordan)

summary(modelElman)
summary(modelJordan)
```

---

encodeClassLabels *Encode a matrix of (decoded) class labels*

Description

Applies `analyzeClassification` row-wise to a matrix.

Usage

```
encodeClassLabels(x, method = "WTA", l = 0, h = 0)
```

Arguments

`x`	inputs
`method`	see `analyzeClassification`
`l`	idem
`h`	idem

Value

a numeric vector, each number represents a different class. A zero means that no class was assigned to the pattern.

See Also

`analyzeClassification`

Examples

```
data(iris)
labels <- decodeClassLabels(iris[,5])
encodeClassLabels(labels)
```

`exportToSnnsNetFile` *Export the net to a file in the original SNNS file format*

Description

Export the net that is present in the `rsnns` object in the original (.net) SNNS file format.

Usage

```
exportToSnnsNetFile(object, filename, netname = "RSNNS_untitled")
```

Arguments

`object`	the `rsnns` object
`filename`	path and filename to be written to
`netname`	name that is given to the network in the file

`extractNetInfo` *Extract information from a network*

Description

This function generates a list of data.frames containing the most important information that defines a network, in a format that is easy to use. To get the full definition in the original SNNS format, use `summary.rsnns` or `exportToSnnsNetFile` instead.

Usage

```
extractNetInfo(object)
```

Arguments

`object`	the `rsnns` object

Details

Internally, a call to `SnnsRObject$extractNetInfo` is done, and the results of this call are returned.

Value

a list containing information extracted from the network (see `SnnsRObject$extractNetInfo`).

See Also

`SnnsRObject$extractNetInfo`

`getNormParameters` *Get normalization parameters of the input data*

Description

Get the normalization parameters that are appended by `normalizeData` as attributes to the input data.

Usage

```
getNormParameters(x)
```

Arguments

`x`	input data

Details

This function is equivalent to calling `attr(x, "normParams")`.

Value

the parameters generated by an earlier call to `normalizeData`

See Also

`normalizeData`, `denormalizeData`

getSnnsRDefine — *Get a define of the SNNS kernel*

Description

Get a define of the SNNS kernel from a defines-list. All defines-lists present can be shown with `RSNNS:::SnnsDefines`.

Usage

```
getSnnsRDefine(defList, defValue)
```

Arguments

`defList`	the defines-list from which to get the define from
`defValue`	the value in the list

Value

a string with the name of the define

See Also

`resolveSnnsRDefine`

Examples

```
getSnnsRDefine("topologicalUnitTypes",3)
getSnnsRDefine("errorCodes",-50)
```

getSnnsRFunctionTable — *Get SnnsR function table*

Description

Get the function table of available SNNS functions.

Usage

```
getSnnsRFunctionTable()
```

Value

a data.frame with columns:

`name`	name of the function
`type`	the type of the function (learning, init, update,...)
`#inParams`	the number of input parameters of the function
`#outParams`	the number of output parameters of the function

`inputColumns` *Get the columns that are inputs*

Description

This function extracts all columns from a matrix whose column names begin with "in". The example data of this package follows this naming convention.

Usage

```
inputColumns(patterns)
```

Arguments

`patterns`	matrix or data.frame containing the patterns

`jordan` *Create and train a Jordan network*

Description

Jordan networks are partially recurrent networks and similar to Elman networks (see `elman`). Partially recurrent networks are useful when working with time series data. I.e., when the output of the network not only should depend on the current pattern, but also on the patterns presented before.

Usage

```
jordan(x, ...)

## Default S3 method:
jordan(
  x,
  y,
  size = c(5),
  maxit = 100,
  initFunc = "JE_Weights",
```

```
  initFuncParams = c(1, -1, 0.3, 1, 0.5),
  learnFunc = "JE_BP",
  learnFuncParams = c(0.2),
  updateFunc = "JE_Order",
  updateFuncParams = c(0),
  shufflePatterns = FALSE,
  linOut = TRUE,
  inputsTest = NULL,
  targetsTest = NULL,
  ...
)
```

Arguments

`x`	a matrix with training inputs for the network
`...`	additional function parameters (currently not used)
`y`	the corresponding targets values
`size`	number of units in the hidden layer(s)
`maxit`	maximum of iterations to learn
`initFunc`	the initialization function to use
`initFuncParams`	the parameters for the initialization function
`learnFunc`	the learning function to use
`learnFuncParams`	the parameters for the learning function
`updateFunc`	the update function to use
`updateFuncParams`	the parameters for the update function
`shufflePatterns`	should the patterns be shuffled?
`linOut`	sets the activation function of the output units to linear or logistic
`inputsTest`	a matrix with inputs to test the network
`targetsTest`	the corresponding targets for the test input

Details

Learning on Jordan networks: Backpropagation algorithms for feed-forward networks can be adapted for their use with this type of networks. In SNNS, there exist adapted versions of several backpropagation-type algorithms for Jordan and Elman networks.

Network architecture: A Jordan network can be seen as a feed-forward network with additional context units in the input layer. These context units take input from themselves (direct feedback), and from the output units. The context units save the current state of the net. In a Jordan net, the number of context units and output units has to be the same.

Initialization of Jordan and Elman nets should be done with the default init function `JE_Weights`, which has five parameters. The first two parameters define an interval from which the forward connections are randomly chosen. The third parameter gives the self-excitation weights of the context units. The fourth parameter gives the weights of context units between them, and the fifth parameter gives the initial activation of context units.

Learning functions are `JE_BP`, `JE_BP_Momentum`, `JE_Quickprop`, and `JE_Rprop`, which are all adapted versions of their standard-procedure counterparts. Update functions that can be used are `JE_Order` and `JE_Special`.

A detailed description of the theory and the parameters is available, as always, from the SNNS documentation and the other referenced literature.

Value

an `rsnns` object.

References

Jordan, M. I. (1986), 'Serial Order: A Parallel, Distributed Processing Approach', Advances in Connectionist Theory Speech 121(ICS-8604), 471-495.

Zell, A. et al. (1998), 'SNNS Stuttgart Neural Network Simulator User Manual, Version 4.2', IPVR, University of Stuttgart and WSI, University of Tübingen. https://www.ra.cs.uni-tuebingen.de/SNNS/welcome.html

Zell, A. (1994), Simulation Neuronaler Netze, Addison-Wesley. (in German)

See Also

`elman`

Examples

```
## Not run: demo(iris)
## Not run: demo(laser)
## Not run: demo(eight_elman)
## Not run: demo(eight_elmanSnnsR)


data(snnsData)
inputs <- snnsData$laser_1000.pat[,inputColumns(snnsData$laser_1000.pat)]
outputs <- snnsData$laser_1000.pat[,outputColumns(snnsData$laser_1000.pat)]

patterns <- splitForTrainingAndTest(inputs, outputs, ratio=0.15)

modelJordan <- jordan(patterns$inputsTrain, patterns$targetsTrain,
                      size=c(8), learnFuncParams=c(0.1), maxit=100,
                      inputsTest=patterns$inputsTest,
                      targetsTest=patterns$targetsTest, linOut=FALSE)

names(modelJordan)

par(mfrow=c(3,3))
plotIterativeError(modelJordan)
```

```
plotRegressionError(patterns$targetsTrain, modelJordan$fitted.values)
plotRegressionError(patterns$targetsTest, modelJordan$fittedTestValues)
hist(modelJordan$fitted.values - patterns$targetsTrain, col="lightblue")

plot(inputs, type="l")
plot(inputs[1:100], type="l")
lines(outputs[1:100], col="red")
lines(modelJordan$fitted.values[1:100], col="green")
```

`matrixToActMapList` *Convert matrix of activations to activation map list*

Description

Organize a matrix containing 1d vectors of network activations as 2d maps.

Usage

```
matrixToActMapList(m, nrow = 0, ncol = 0)
```

Arguments

`m`	the matrix containing one activation pattern in every row
`nrow`	number of rows the resulting matrices will have
`ncol`	number of columns the resulting matrices will have

Details

The input to this function is a matrix containing in each row an activation pattern/output of a neural network. This function uses vectorToActMap to reorganize the matrix to a list of matrices, whereby each row of the input matrix is converted to a matrix in the output list.

Value

a list containing the activation map matrices

See Also

vectorToActMap plotActMap

mlp *Create and train a multi-layer perceptron (MLP)*

Description

This function creates a multilayer perceptron (MLP) and trains it. MLPs are fully connected feed-forward networks, and probably the most common network architecture in use. Training is usually performed by error backpropagation or a related procedure.

There are a lot of different learning functions present in SNNS that can be used together with this function, e.g., `Std_Backpropagation`, `BackpropBatch`, `BackpropChunk`, `BackpropMomentum`, `BackpropWeightDecay`, `Rprop`, `Quickprop`, `SCG` (scaled conjugate gradient), ...

Usage

```
mlp(x, ...)

## Default S3 method:
mlp(
  x,
  y,
  size = c(5),
  maxit = 100,
  initFunc = "Randomize_Weights",
  initFuncParams = c(-0.3, 0.3),
  learnFunc = "Std_Backpropagation",
  learnFuncParams = c(0.2, 0),
  updateFunc = "Topological_Order",
  updateFuncParams = c(0),
  hiddenActFunc = "Act_Logistic",
  shufflePatterns = TRUE,
  linOut = FALSE,
  outputActFunc = if (linOut) "Act_Identity" else "Act_Logistic",
  inputsTest = NULL,
  targetsTest = NULL,
  pruneFunc = NULL,
  pruneFuncParams = NULL,
  ...
)
```

Arguments

`x`	a matrix with training inputs for the network
`...`	additional function parameters (currently not used)
`y`	the corresponding targets values
`size`	number of units in the hidden layer(s)

maxit	maximum of iterations to learn
initFunc	the initialization function to use
initFuncParams	the parameters for the initialization function
learnFunc	the learning function to use
learnFuncParams	the parameters for the learning function
updateFunc	the update function to use
updateFuncParams	the parameters for the update function
hiddenActFunc	the activation function of all hidden units
shufflePatterns	should the patterns be shuffled?
linOut	sets the activation function of the output units to linear or logistic (ignored if outputActFunc is given)
outputActFunc	the activation function of all output units
inputsTest	a matrix with inputs to test the network
targetsTest	the corresponding targets for the test input
pruneFunc	the pruning function to use
pruneFuncParams	the parameters for the pruning function. Unlike the other functions, these have to be given in a named list. See the pruning demos for further explanation.

Details

Std_Backpropagation, BackpropBatch, e.g., have two parameters, the learning rate and the maximum output difference. The learning rate is usually a value between 0.1 and 1. It specifies the gradient descent step width. The maximum difference defines, how much difference between output and target value is treated as zero error, and not backpropagated. This parameter is used to prevent overtraining. For a complete list of the parameters of all the learning functions, see the SNNS User Manual, pp. 67.

The defaults that are set for initialization and update functions usually don't have to be changed.

Value

an rsnns object.

References

Rosenblatt, F. (1958), 'The perceptron: A probabilistic model for information storage and organization in the brain', Psychological Review 65(6), 386–408.

Rumelhart, D. E.; Clelland, J. L. M. & Group, P. R. (1986), Parallel distributed processing :explorations in the microstructure of cognition, Mit, Cambridge, MA etc.

Zell, A. et al. (1998), 'SNNS Stuttgart Neural Network Simulator User Manual, Version 4.2', IPVR, University of Stuttgart and WSI, University of Tübingen. https://www.ra.cs.uni-tuebingen.de/SNNS/welcome.html

Zell, A. (1994), Simulation Neuronaler Netze, Addison-Wesley. (in German)

Examples

```
## Not run: demo(iris)
## Not run: demo(laser)
## Not run: demo(encoderSnnsCLib)


data(iris)

#shuffle the vector
iris <- iris[sample(1:nrow(iris),length(1:nrow(iris))),1:ncol(iris)]

irisValues <- iris[,1:4]
irisTargets <- decodeClassLabels(iris[,5])
#irisTargets <- decodeClassLabels(iris[,5], valTrue=0.9, valFalse=0.1)

iris <- splitForTrainingAndTest(irisValues, irisTargets, ratio=0.15)
iris <- normTrainingAndTestSet(iris)

model <- mlp(iris$inputsTrain, iris$targetsTrain, size=5, learnFuncParams=c(0.1),
              maxit=50, inputsTest=iris$inputsTest, targetsTest=iris$targetsTest)

summary(model)
model
weightMatrix(model)
extractNetInfo(model)

par(mfrow=c(2,2))
plotIterativeError(model)

predictions <- predict(model,iris$inputsTest)

plotRegressionError(predictions[,2], iris$targetsTest[,2])

confusionMatrix(iris$targetsTrain,fitted.values(model))
confusionMatrix(iris$targetsTest,predictions)

plotROC(fitted.values(model)[,2], iris$targetsTrain[,2])
plotROC(predictions[,2], iris$targetsTest[,2])

#confusion matrix with 402040-method
confusionMatrix(iris$targetsTrain, encodeClassLabels(fitted.values(model),
                                                       method="402040", l=0.4, h=0.6))
```

---

normalizeData *Data normalization*

Description

The input matrix is column-wise normalized.

Usage

```
normalizeData(x, type = "norm")
```

Arguments

x	input data
type	**either** type string specifying the type of normalization. Implemented are "0_1", "center", and "norm" **or** attribute list of a former call to this method to apply e.g. normalization of the training data to the test data

Details

The parameter `type` specifies, how normalization takes place:

0_1 values are normalized to the [0,1]-interval. The minimum in the data is mapped to zero, the maximum to one.

center the data is centered, i.e. the mean is substracted

norm the data is normalized to mean zero, variance one

Value

column-wise normalized input. The normalization parameters that were used for the normalization are present as attributes of the output. They can be obtained with `getNormParameters`.

See Also

`denormalizeData`, `getNormParameters`

`normTrainingAndTestSet` *Function to normalize training and test set*

Description

Normalize training and test set as obtained by `splitForTrainingAndTest` in the following way: The `inputsTrain` member is normalized using `normalizeData` with the parameters given in `type`. The normalization parameters obtained during this normalization are then used to normalize the `inputsTest` member. if `dontNormTargets` is not set, then the targets are normalized in the same way. In classification problems, normalizing the targets normally makes no sense. For regression, normalizing also the targets is usually a good idea. The default is to not normalize targets values.

Usage

```
normTrainingAndTestSet(x, dontNormTargets = TRUE, type = "norm")
```

Arguments

`x`	a list containing training and test data. Usually the output of `splitForTrainingAndTest`.
`dontNormTargets`	should the target values also be normalized?
`type`	type of the normalization. This parameter is passed to `normalizeData`.

Value

a named list with the same elements as `splitForTrainingAndTest`, but with normalized values. The normalization parameters are appended to each member of the list as attributes, as in `normalizeData`.

See Also

`splitForTrainingAndTest`, `normalizeData`, `denormalizeData`, `getNormParameters`

Examples

```
data(iris)
#shuffle the vector
iris <- iris[sample(1:nrow(iris),length(1:nrow(iris))),1:ncol(iris)]

irisValues <- iris[,1:4]
irisTargets <- decodeClassLabels(iris[,5])

iris <- splitForTrainingAndTest(irisValues, irisTargets, ratio=0.15)
normTrainingAndTestSet(iris)
```

`outputColumns` *Get the columns that are targets*

Description

This function extracts all columns from a matrix whose column names begin with "out". The example data of this package follows this naming convention.

Usage

```
outputColumns(patterns)
```

Arguments

`patterns`	matrix or data.frame containing the patterns

plotActMap *Plot activation map*

Description

Plot an activation map as a heatmap.

Usage

```
plotActMap(x, ...)
```

Arguments

x	the input data matrix
...	parameters passed to image

See Also

vectorToActMap matrixToActMapList

plotIterativeError *Plot iterative errors of an rsnns object*

Description

Plot the iterative training and test error of the net of this rsnns object.

Usage

```
plotIterativeError(object, ...)

## S3 method for class 'rsnns'
plotIterativeError(object, ...)
```

Arguments

object	a rsnns object
...	parameters passed to plot

Details

Plots (if present) the class members IterativeFitError (as black line) and IterativeTestError (as red line).

plotRegressionError — *Plot a regression error plot*

Description

The plot shows target values on the x-axis and fitted/predicted values on the y-axis. The optimal fit would yield a line through zero with gradient one. This optimal line is shown in black color. A linear fit to the actual data is shown in red color.

Usage

```
plotRegressionError(targets, fits, ...)
```

Arguments

`targets`	the target values
`fits`	the values predicted/fitted by the model
`...`	parameters passed to `plot`

plotROC — *Plot a ROC curve*

Description

This function plots a receiver operating characteristic (ROC) curve.

Usage

```
plotROC(T, D, ...)
```

Arguments

`T`	predictions
`D`	targets
`...`	parameters passed to plot

Author(s)

Code is taken from R news Volume 4/1, June 2004.

References

R news Volume 4/1, June 2004

`predict.rsnns` *Generic predict function for rsnns object*

Description

Predict values using the given network.

Usage

```
## S3 method for class 'rsnns'
predict(object, newdata, ...)
```

Arguments

`object`	the `rsnns` object
`newdata`	the new input data which is used for prediction
`...`	additional function parameters (currently not used)

Value

the predicted values

`print.rsnns` *Generic print function for rsnns objects*

Description

Print out some characteristics of an `rsnns` object.

Usage

```
## S3 method for class 'rsnns'
print(x, ...)
```

Arguments

`x`	the `rsnns` object
`...`	additional function parameters (currently not used)

rbf *Create and train a radial basis function (RBF) network*

Description

The use of an RBF network is similar to that of an mlp. The idea of radial basis function networks comes from function interpolation theory. The RBF performs a linear combination of n basis functions that are radially symmetric around a center/prototype.

Usage

```
rbf(x, ...)

## Default S3 method:
rbf(
  x,
  y,
  size = c(5),
  maxit = 100,
  initFunc = "RBF_Weights",
  initFuncParams = c(0, 1, 0, 0.02, 0.04),
  learnFunc = "RadialBasisLearning",
  learnFuncParams = c(1e-05, 0, 1e-05, 0.1, 0.8),
  updateFunc = "Topological_Order",
  updateFuncParams = c(0),
  shufflePatterns = TRUE,
  linOut = TRUE,
  inputsTest = NULL,
  targetsTest = NULL,
  ...
)
```

Arguments

`x`	a matrix with training inputs for the network
`...`	additional function parameters (currently not used)
`y`	the corresponding targets values
`size`	number of units in the hidden layer(s)
`maxit`	maximum of iterations to learn
`initFunc`	the initialization function to use
`initFuncParams`	the parameters for the initialization function
`learnFunc`	the learning function to use
`learnFuncParams`	the parameters for the learning function

`updateFunc` the update function to use

`updateFuncParams` the parameters for the update function

`shufflePatterns` should the patterns be shuffled?

`linOut` sets the activation function of the output units to linear or logistic

`inputsTest` a matrix with inputs to test the network

`targetsTest` the corresponding targets for the test input

Details

RBF networks are feed-forward networks with one hidden layer. Their activation is not sigmoid (as in MLP), but radially symmetric (often gaussian). Thereby, information is represented locally in the network (in contrast to MLP, where it is globally represented). Advantages of RBF networks in comparison to MLPs are mainly, that the networks are more interpretable, training ought to be easier and faster, and the network only activates in areas of the feature space where it was actually trained, and has therewith the possibility to indicate that it "just doesn't know".

Initialization of an RBF network can be difficult and require prior knowledge. Before use of this function, you might want to read pp 172-183 of the SNNS User Manual 4.2. The initialization is performed in the current implementation by a call to `RBF_Weights_Kohonen(0,0,0,0,0)` and a successive call to the given `initFunc` (usually `RBF_Weights`). If this initialization doesn't fit your needs, you should use the RSNNS low-level interface to implement your own one. Have a look then at the demos/examples. Also, we note that depending on whether linear or logistic output is chosen, the initialization parameters have to be different (normally `c(0,1,...)` for linear and `c(-4,4,...)` for logistic output).

Value

an `rsnns` object.

References

Poggio, T. & Girosi, F. (1989), 'A Theory of Networks for Approximation and Learning'(A.I. Memo No.1140, C.B.I.P. Paper No. 31), Technical report, MIT ARTIFICIAL INTELLIGENCE LABORATORY.

Vogt, M. (1992), 'Implementierung und Anwendung von Generalized Radial Basis Functions in einem Simulator neuronaler Netze', Master's thesis, IPVR, University of Stuttgart. (in German)

Zell, A. et al. (1998), 'SNNS Stuttgart Neural Network Simulator User Manual, Version 4.2', IPVR, University of Stuttgart and WSI, University of Tübingen. https://www.ra.cs.uni-tuebingen.de/SNNS/welcome.html

Zell, A. (1994), Simulation Neuronaler Netze, Addison-Wesley. (in German)

Examples

```
## Not run: demo(rbf_irisSnnsR)
## Not run: demo(rbf_sin)
## Not run: demo(rbf_sinSnnsR)
```

```
inputs <- as.matrix(seq(0,10,0.1))
outputs <- as.matrix(sin(inputs) + runif(inputs*0.2))
outputs <- normalizeData(outputs, "0_1")

model <- rbf(inputs, outputs, size=40, maxit=1000,
                     initFuncParams=c(0, 1, 0, 0.01, 0.01),
                     learnFuncParams=c(1e-8, 0, 1e-8, 0.1, 0.8), linOut=TRUE)

par(mfrow=c(2,1))
plotIterativeError(model)
plot(inputs, outputs)
lines(inputs, fitted(model), col="green")
```

`rbfDDA` *Create and train an RBF network with the DDA algorithm*

Description

Create and train an RBF network with the dynamic decay adjustment (DDA) algorithm. This type of network can only be used for classification. The training typically begins with an empty network, i.e., a network only consisting of input and output units, and adds new units successively. It is a lot easier to use than normal RBF, because it only requires two quite uncritical parameters.

Usage

```
rbfDDA(x, ...)

## Default S3 method:
rbfDDA(
  x,
  y,
  maxit = 1,
  initFunc = "Randomize_Weights",
  initFuncParams = c(-0.3, 0.3),
  learnFunc = "RBF-DDA",
  learnFuncParams = c(0.4, 0.2, 5),
  updateFunc = "Topological_Order",
  updateFuncParams = c(0),
  shufflePatterns = TRUE,
  linOut = FALSE,
  ...
)
```

Arguments

`x` a matrix with training inputs for the network

`...`	additional function parameters (currently not used)
`y`	the corresponding targets values
`maxit`	maximum of iterations to learn
`initFunc`	the initialization function to use
`initFuncParams`	the parameters for the initialization function
`learnFunc`	the learning function to use
`learnFuncParams`	the parameters for the learning function
`updateFunc`	the update function to use
`updateFuncParams`	the parameters for the update function
`shufflePatterns`	should the patterns be shuffled?
`linOut`	sets the activation function of the output units to linear or logistic

Details

The default functions do not have to be altered. The learning function `RBF-DDA` has three parameters: a positive threshold, and a negative threshold, that controls adding units to the network, and a parameter for display purposes in the original SNNS. This parameter has no effect in RSNNS. See p 74 of the original SNNS User Manual for details.

Value

an `rsnns` object.

References

Berthold, M. R. & Diamond, J. (1995), Boosting the Performance of RBF Networks with Dynamic Decay Adjustment, in 'Advances in Neural Information Processing Systems', MIT Press, , pp. 521–528.

Hudak, M. (1993), 'RCE classifiers: theory and practice', Cybernetics and Systems 23(5), 483–515.

Zell, A. et al. (1998), 'SNNS Stuttgart Neural Network Simulator User Manual, Version 4.2', IPVR, University of Stuttgart and WSI, University of Tübingen. https://www.ra.cs.uni-tuebingen.de/SNNS/welcome.html

Examples

```
## Not run: demo(iris)
## Not run: demo(rbfDDA_spiralsSnnsR)


data(iris)
iris <- iris[sample(1:nrow(iris),length(1:nrow(iris))),1:ncol(iris)]
irisValues <- iris[,1:4]
irisTargets <- decodeClassLabels(iris[,5])
iris <- splitForTrainingAndTest(irisValues, irisTargets, ratio=0.15)
```

```
iris <- normTrainingAndTestSet(iris)

model <- rbfDDA(iris$inputsTrain, iris$targetsTrain)

summary(model)
plotIterativeError(model)
```

---

readPatFile — *Load data from a pat file*

Description

This function generates an SnnsR-class object, loads the given .pat file there as a pattern set and then extracts the patterns to a matrix, using SnnsRObject$extractPatterns.

Usage

```
readPatFile(filename)
```

Arguments

`filename`	the name of the .pat file

Value

a matrix containing the data loaded from the .pat file.

---

readResFile — *Rudimentary parser for res files.*

Description

This function contains a rudimentary parser for SNNS .res files. It is completely implemented in R and doesn't make use of SNNS functionality.

Usage

```
readResFile(filename)
```

Arguments

`filename`	the name of the .res file

Value

a matrix containing the predicted values that were found in the .res file

resolveSnnsRDefine *Resolve a define of the SNNS kernel*

Description

Resolve a define of the SNNS kernel using a defines-list. All defines-lists present can be shown with `RSNNS:::SnnsDefines`.

Usage

```
resolveSnnsRDefine(defList, def)
```

Arguments

`defList`	the defines-list from which to resolve the define from
`def`	the name of the define

Value

the value of the define

See Also

`getSnnsRDefine`

Examples

```
resolveSnnsRDefine("topologicalUnitTypes","UNIT_HIDDEN")
```

rsnnsObjectFactory *Object factory for generating rsnns objects*

Description

The object factory generates an `rsnns` object and initializes its member variables with the values given as parameters. Furthermore, it generates an object of `SnnsR-class`. Later, this information is to be used to train the network.

Usage

```
rsnnsObjectFactory(
  subclass,
  nInputs,
  maxit,
  initFunc,
  initFuncParams,
  learnFunc,
  learnFuncParams,
  updateFunc,
  updateFuncParams,
  shufflePatterns = TRUE,
  computeIterativeError = TRUE,
  pruneFunc = NULL,
  pruneFuncParams = NULL
)
```

Arguments

`subclass`	the subclass of rsnns to generate (vector of strings)
`nInputs`	the number of inputs the network will have
`maxit`	maximum of iterations to learn
`initFunc`	the initialization function to use
`initFuncParams`	the parameters for the initialization function
`learnFunc`	the learning function to use
`learnFuncParams`	the parameters for the learning function
`updateFunc`	the update function to use
`updateFuncParams`	the parameters for the update function
`shufflePatterns`	should the patterns be shuffled?
`computeIterativeError`	should the error be computed in every iteration?
`pruneFunc`	the pruning function to use
`pruneFuncParams`	the parameters for the pruning function. Unlike the other functions, these have to be given in a named list. See the pruning demos for further explanation.

Details

The typical procedure implemented in `rsnns` subclasses is the following:

- generate the `rsnns` object with this object factory
- generate the network according to the architecture needed

- train the network (with train)

In every rsnns object, the iterative error is the summed squared error (SSE) of all patterns. If the SSE is computed on the test set, then it is weighted to take care of the different amount of patterns in the sets.

Value

a partly initialized rsnns object

See Also

mlp, dlvq, rbf, rbfDDA, elman, jordan, som, art1, art2, artmap, assoz

---

savePatFile *Save data to a pat file*

Description

This function generates an SnnsR-class object, loads the given data there as a pattern set and then uses the functionality of SNNS to save the data as a .pat file.

Usage

```
savePatFile(inputs, targets, filename)
```

Arguments

inputs	a matrix with input values
targets	a matrix with target values
filename	the name of the .pat file

---

setSnnsRSeedValue *DEPRECATED, Set the SnnsR seed value*

Description

DEPRECATED, now just calls R's set.seed(), that should be used instead.

Usage

```
setSnnsRSeedValue(seed)
```

Arguments

seed	the seed to use. If 0, a seed based on the system time is generated.

snnsData — *Example data of the package*

Description

This is data from the original SNNS examples directory ported to R and stored as one list. The function readPatFile was used to parse all pattern files (.pat) from the original SNNS examples directory. Due to limitations of that function, pattern files containing patterns with variable size were omitted.

Examples

```
data(snnsData)
names(snnsData)
```

SnnsR-class — *The main class of the package*

Description

An S4 class that is the main class of RSNNS. Each instance of this class contains a pointer to a C++ object of type SnnsCLib, i.e. an instance of the SNNS kernel.

Details

The only slot `variables` holds an environment with all member variables. Currently, there are two members (constructed by the object factory):

snnsCLibPointer A pointer to the corresponding C++ object

serialization a serialization of the C++ object, in SNNS .net format

The member variables are not directly present as slots but wrapped in an environment to allow for changing the serialization (by call by reference).

An object of this class is used internally by all the models in the package. The object is always accessible by `model$snnsObject$...`

To make full use of the SNNS functionalities, you might want to use this class directly. Always use the object factory SnnsRObjectFactory to construct an object, and the calling mechanism $ to call functions. Through the calling mechanism, many functions of SnnsCLib are present that are not documented here, but in the SNNS User Manual. So, if you choose to use the low-level interface, it is highly recommended to have a look at the demos and at the SNNS User Manual.

References

Zell, A. et al. (1998), 'SNNS Stuttgart Neural Network Simulator User Manual, Version 4.2', IPVR, University of Stuttgart and WSI, University of Tübingen. https://www.ra.cs.uni-tuebingen.de/SNNS/welcome.html

See Also

$, SnnsRObjectFactory

Examples

```
## Not run: demo(encoderSnnsCLib)
## Not run: demo(art1_lettersSnnsR)
## Not run: demo(art2_tetraSnnsR)
## Not run: demo(artmap_lettersSnnsR)
## Not run: demo(eight_elmanSnnsR)
## Not run: demo(rbf_irisSnnsR)
## Not run: demo(rbf_sinSnnsR)
## Not run: demo(rbfDDA_spiralsSnnsR)
## Not run: demo(som_cubeSnnsR)


#This is the demo eight_elmanSnnsR
#Here, we train an Elman network
#and save a trained and an untrained version
#to disk, as well as the used training data

basePath <- ("./")

data(snnsData)

inputs <- snnsData$eight_016.pat[,inputColumns(snnsData$eight_016.pat)]
outputs <- snnsData$eight_016.pat[,outputColumns(snnsData$eight_016.pat)]

snnsObject <- SnnsRObjectFactory()

snnsObject$setLearnFunc('JE_BP')
snnsObject$setUpdateFunc('JE_Order')
snnsObject$setUnitDefaults(1,0,1,0,1,'Act_Logistic','Out_Identity')

snnsObject$elman_createNet(c(2,8,2),c(1,1,1),FALSE)


patset <- snnsObject$createPatSet(inputs, outputs)
snnsObject$setCurrPatSet(patset$set_no)

snnsObject$initializeNet(c(1.0,  -1.0,  0.3,  1.0,  0.5) )
snnsObject$shufflePatterns(TRUE)
snnsObject$DefTrainSubPat()

## Not run: snnsObject$saveNet(paste(basePath,"eight_elmanSnnsR_untrained.net",sep=""),
                                          "eight_elmanSnnsR_untrained")
## End(Not run)

parameters <- c(0.2, 0, 0, 0, 0)
maxit <- 1000

error <- vector()
```

```
for(i in 1:maxit) {
  res <- snnsObject$learnAllPatterns(parameters)
  if(res[[1]] != 0) print(paste("Error at iteration ", i, " : ", res, sep=""))
  error[i] <- res[[2]]
}

error[1:500]
plot(error, type="l")

## Not run: snnsObject$saveNet(paste(basePath,"eight_elmanSnnsR.net",sep=""),
                                              "eight_elmanSnnsR")
## End(Not run)
## Not run: snnsObject$saveNewPatterns(paste(basePath,"eight_elmanSnnsR.pat",sep=""),
                                                          patset$set_no)
## End(Not run)
```

`SnnsRObjectFactory` *SnnsR object factory*

Description

Object factory to create a new object of type `SnnsR-class`.

Usage

```
SnnsRObjectFactory(x = NULL)
```

Arguments

`x`	(optional) object of class SnnsR-class, to be deep-copied

Details

This function creates a new object of class `SnnsR-class`, initializes its only slot `variables` with a new environment, generates a new C++ object of class SnnsCLib, and an empty object serialization.

See Also

`$`, `SnnsR-class`

Examples

```
mySnnsObject <- SnnsRObjectFactory()
mySnnsObject$setLearnFunc('Quickprop')
mySnnsObject$setUpdateFunc('Topological_Order')
```

SnnsRObjectMethodCaller

Method caller for SnnsR objects

Description

Enable calling of C++ functions as methods of `SnnsR-class` objects.

Usage

```
## S4 method for signature 'SnnsR'
x$name
```

Arguments

`x`	object of class SnnsR-class
`name`	function to call

Details

This function makes methods of SnnsR__ and SnnsCLib__ accessible via "$". If no SnnsR__ method is present, then the according SnnsCLib__ method is called. This enables a very flexible method handling. To mask a method from SnnsCLib, e.g. to do some parameter checking or postprocessing, only a method with the same name, but beginning with SnnsR__ has to be present in R. See e.g. `SnnsRObject$initializeNet` for such an implementation.

Error handling is also done within the method caller. If the result of a function is a list with a member `err`, then `SnnsCLib__error` is called to use the SNNS kernel function to get the corresponding error message code and an R warning is thrown containing this message.

Furthermore, a serialization mechanism is implemented which all models present in the package use to be able to be saved and loaded by R's normal save/load mechanism (as RData files).

The completely trained object can be serialized with

`s <- snnsObject$serializeNet("RSNNS_untitled")`

`snnsObject@variables$serialization <- s$serialization`

For the models implemented, this is done in `SnnsRObject$train`. If the S4 object is then saved and loaded, the calling mechanism will notice on the next use of a function that the pointer to the C++ SnnsCLib object is `nil`, and if a serialization is present, the object is restored from this serialization before the method is called.

SnnsRObject$createNet *Create a layered network*

Description

This function creates a layered network in the given SnnsR object. This is an SnnsR low-level function. You may want to have a look at SnnsR-class to find out how to properly use it.

Usage

```
## S4 method for signature 'SnnsR'
createNet(unitsPerLayer, fullyConnectedFeedForward = TRUE, iNames = NULL, oNames = NULL)
```

Arguments

- `unitsPerLayer` a vector of integers that represents the number of units in each layer, including input and output layer
- `fullyConnectedFeedForward` if `TRUE`, the network is fully connected as a feed-forward network. If `FALSE`, no connections are created
- `iNames` names of input units
- `oNames` names of output units

See Also

SnnsR-class

Examples

```
obj1 <- SnnsRObjectFactory()
obj1$createNet(c(2,2), FALSE)
obj1$getUnitDefinitions()

obj2 <- SnnsRObjectFactory()
obj2$createNet(c(8,5,5,2), TRUE)
obj2$getUnitDefinitions()
```

`SnnsRObject$createPatSet`
Create a pattern set

Description

SnnsR low-level function to create a pattern set in the SNNS kernel from the values given, so that they are available in the SNNS kernel for use.

Usage

```
## S4 method for signature 'SnnsR'
createPatSet(inputs, targets)
```

Arguments

`inputs`	the input values
`targets`	the target values

Value

a list with elements `err` and `set_no`. The latter one identifies the pattern set within the `SnnsR-class` object

---

`SnnsRObject$extractNetInfo`
Get characteristics of the network.

Description

The returned list has three members:

- infoHeader general information about the network
- unitDefinitions information about the units
- fullWeightMatrix weight matrix of the connections

Usage

```
## S4 method for signature 'SnnsR'
extractNetInfo()
```

Value

a list of data frames containing information extracted from the network.

SnnsRObject$extractPatterns *Extract the current pattern set to a matrix*

Description

SnnsR low-level function that extracts all patterns of the current pattern set and returns them as a matrix. Columns are named with the prefix "in" or "out", respectively.

Usage

```
## S4 method for signature 'SnnsR'
extractPatterns()
```

Value

a matrix containing the patterns of the currently loaded patern set.

SnnsRObject$getAllHiddenUnits *Get all hidden units of the net*

Description

SnnsR low-level function to get all units from the net with the ttype "UNIT_HIDDEN". This function calls SnnsRObject$getAllUnitsTType with the parameter "UNIT_HIDDEN".

Usage

```
## S4 method for signature 'SnnsR'
getAllHiddenUnits()
```

Value

a vector with integer numbers identifying the units.

See Also

SnnsRObject$getAllUnitsTType

SnnsRObject$getAllInputUnits
Get all input units of the net

Description

SnnsR low-level function to get all units from the net with the ttype "UNIT_INPUT". This function calls SnnsRObject$getAllUnitsTType with the parameter "UNIT_INPUT".

Usage

```
## S4 method for signature 'SnnsR'
getAllInputUnits()
```

Value

a vector with integer numbers identifying the units.

See Also

SnnsRObject$getAllUnitsTType

SnnsRObject$getAllOutputUnits
Get all output units of the net.

Description

SnnsR low-level function to get all units from the net with the ttype "UNIT_OUTPUT". This function calls SnnsRObject$getAllUnitsTType with the parameter "UNIT_OUTPUT".

Usage

```
## S4 method for signature 'SnnsR'
getAllOutputUnits()
```

Value

a vector with integer numbers identifying the units.

See Also

SnnsRObject$getAllUnitsTType

SnnsRObject$getAllUnits

Get all units present in the net.

Description

Get all units present in the net.

Usage

```
## S4 method for signature 'SnnsR'
getAllUnits()
```

Value

a vector with integer numbers identifying the units.

SnnsRObject$getAllUnitsTType

Get all units in the net of a certain `ttype`.

Description

SnnsR low-level function to get all units in the net of a certain `ttype`. Possible `ttype` defined by SNNS are, among others: "UNIT_OUTPUT", "UNIT_INPUT", and "UNIT_HIDDEN". For a full list, call `RSNNS:::SnnsDefines$topologicalUnitTypes` As this is an SnnsR low-level function, you may want to have a look at `SnnsR-class` to find out how to properly use it.

Usage

```
## S4 method for signature 'SnnsR'
getAllUnitsTType(ttype)
```

Arguments

`ttype`	a string containing the `ttype`.

Value

a vector with integer numbers identifying the units.

See Also

SnnsRObject$getAllOutputUnits, SnnsRObject$getAllInputUnits, SnnsRObject$getAllHiddenUnits

---

`SnnsRObject$getCompleteWeightMatrix`
Get the complete weight matrix.

Description

Get a weight matrix containing all weights of all neurons present in the net.

Usage

```
## S4 method for signature 'SnnsR'
getCompleteWeightMatrix(setDimNames)
```

Arguments

`setDimNames`	indicates, whether names of units are extracted and set as row/col names in the weight matrix

Value

the complete weight matrix

`SnnsRObject$getInfoHeader`
Get an info header of the network.

Description

Get an info header of the network.

Usage

```
## S4 method for signature 'SnnsR'
getInfoHeader()
```

Value

a data frame containing some general characteristics of the network.

SnnsRObject$getSiteDefinitions

Get the sites definitions of the network.

Description

Get the sites definitions of the network.

Usage

```
## S4 method for signature 'SnnsR'
getSiteDefinitions()
```

Value

a data frame containing information about all sites present in the network.

SnnsRObject$getTypeDefinitions

Get the FType definitions of the network.

Description

Get the FType definitions of the network.

Usage

```
## S4 method for signature 'SnnsR'
getTypeDefinitions()
```

Value

a data frame containing information about FType units present in the network.

SnnsRObject$getUnitDefinitions

Get the unit definitions of the network.

Description

Get the unit definitions of the network.

Usage

```
## S4 method for signature 'SnnsR'
getUnitDefinitions()
```

Value

a data frame containing information about all units present in the network.

SnnsRObject$getUnitsByName

Find all units whose name begins with a given prefix.

Description

Find all units whose name begins with a given prefix.

Usage

```
## S4 method for signature 'SnnsR'
getUnitsByName(prefix)
```

Arguments

`prefix`	a prefix that the names of the units to find have.

Value

a vector with integer numbers identifying the units.

`SnnsRObject$getWeightMatrix` *Get the weight matrix between two sets of units*

Description

SnnsR low-level function to get the weight matrix between two sets of units.

Usage

```
## S4 method for signature 'SnnsR'
getWeightMatrix(unitsSource, unitsTarget, setDimNames)
```

Arguments

`unitsSource`	a vector with numbers identifying the source units
`unitsTarget`	a vector with numbers identifying the target units
`setDimNames`	indicates, whether names of units are extracted and set as row/col names in the weight matrix

Value

the weight matrix between the two sets of neurons

See Also

SnnsRObject$getAllUnitsTType

`SnnsRObject$initializeNet` *Initialize the network*

Description

This SnnsR low-level function masks the SNNS kernel function of the same name to allow for both giving the initialization function directly in the call or to use the one that is currently set.

Usage

```
## S4 method for signature 'SnnsR'
initializeNet(parameterInArray, initFunc)
```

Arguments

`parameterInArray`	the parameters of the initialization function
`initFunc`	the name of the initialization function

SnnsRObject$predictCurrPatSet

Predict values with a trained net

Description

SnnsR low-level function to predict values with a trained net.

Usage

```
## S4 method for signature 'SnnsR'
predictCurrPatSet(outputMethod="reg_class", updateFuncParams=c(0.0))
```

Arguments

`outputMethod`	is passed to SnnsRObject$whereAreResults
`updateFuncParams`	parameters passed to the networks update function

Details

This function has to be used embedded in a step of loading and afterwards removing the patterns into the `SnnsR-class` object. As SNNS only supports 2 pattern sets in parallel, removing unneeded pattern sets is quite important.

Value

the predicted values

SnnsRObject$resetRSNNS

Reset the SnnsR object.

Description

SnnsR low-level function to delete all pattern sets and delete the current net in the `SnnsR-class` object.

Usage

```
## S4 method for signature 'SnnsR'
resetRSNNS()
```

SnnsRObject$setTTypeUnitsActFunc

Set the activation function for all units of a certain ttype.

Description

The function uses the function SnnsRObject$getAllUnitsTType to find all units of a certain `ttype`, and sets the activation function of all these units to the given activation function.

Usage

```
## S4 method for signature 'SnnsR'
setTTypeUnitsActFunc(ttype, act_func)
```

Arguments

`ttype`	a string containing the `ttype`.
`act_func`	the name of the activation function to set.

See Also

SnnsRObject$getAllUnitsTType

Examples

```
## Not run: SnnsRObject$setTTypeUnitsActFunc("UNIT_HIDDEN", "Act_Logistic")
```

SnnsRObject$setUnitDefaults

Set the unit defaults

Description

This SnnsR low-level function masks the SNNS kernel function of the same name to allow both for giving the parameters directly or as a vector. If the second parameter, `bias`, is missing, it is assumed that the first parameter should be interpreted as a vector containing all parameters.

Usage

```
## S4 method for signature 'SnnsR'
setUnitDefaults(act, bias, st, subnet_no, layer_no, act_func, out_func)
```

Arguments

act	same as SNNS kernel function
bias	idem
st	idem
subnet_no	idem
layer_no	idem
act_func	idem
out_func	idem

SnnsRObject$somPredictComponentMaps
Calculate the som component maps

Description

SnnsR low-level function to calculate the som component maps.

Usage

```
## S4 method for signature 'SnnsR'
somPredictComponentMaps(updateFuncParams=c(0.0, 0.0, 1.0))
```

Arguments

updateFuncParams
: parameters passed to the networks update function

Value

a matrix containing all componant maps as 1d vectors

See Also

som

SnnsRObject$somPredictCurrPatSetWinners
Get most of the relevant results from a som

Description

SnnsR low-level function to get most of the relevant results from a SOM.

Usage

```
## S4 method for signature 'SnnsR'
somPredictCurrPatSetWinners(updateFuncParams=c(0.0, 0.0, 1.0),
saveWinnersPerPattern=TRUE, targets=NULL)
```

Arguments

`updateFuncParams`
: parameters passed to the networks update function

`saveWinnersPerPattern`
: should a list with the winners for every pattern be saved?

`targets`
: optional target classes of the patterns

Value

a list with three elements:

`nWinnersPerUnit`
: For each unit, the amount of patterns where this unit won is given. So, this is a 1d vector representing the normal version of the som.

`winnersPerPattern`
: a vector where for each pattern the number of the winning unit is given. This is an intermediary result that normally won't be saved.

`labeledUnits`
: a matrix which – if the `targets` parameter is given – contains for each unit (rows) and each class present in the `targets` (columns), the amount of patterns of the class where the unit has won. From the `labeledUnits`, the `labeledMap` can be computed, e.g. by voting of the class labels for the final label of the unit.

See Also

som

SnnsRObject$somPredictCurrPatSetWinnersSpanTree *Get the spanning tree of the SOM*

Description

SnnsR low-level function to get the spanning tree of the SOM, This function calls directly the corresponding SNNS kernel function (the only one available for SOM). Advantage are faster computation, disadvantage is somewhat limited information in the output.

Usage

```
## S4 method for signature 'SnnsR'
somPredictCurrPatSetWinnersSpanTree()
```

Value

the spanning tree, which is the som, showing for each unit a number identifying the last pattern for which this unit won. (We note that, also if there are more than one patterns, only the last one is saved)

See Also

som

SnnsRObject$train *Train a network and test it in every training iteration*

Description

SnnsR low-level function to train a network and test it in every training iteration.

Usage

```
## S4 method for signature 'SnnsR'
train(inputsTrain, targetsTrain=NULL,
    initFunc="Randomize_Weights", initFuncParams=c(1.0, -1.0),
    learnFunc="Std_Backpropagation", learnFuncParams=c(0.2, 0),
    updateFunc="Topological_Order", updateFuncParams=c(0.0),
    outputMethod="reg_class", maxit=100, shufflePatterns=TRUE,
    computeError=TRUE, inputsTest=NULL, targetsTest=NULL,
    pruneFunc=NULL, pruneFuncParams=NULL)
```

Arguments

`inputsTrain` a matrix with inputs for the network

`targetsTrain` the corresponding targets

`initFunc` the initialization function to use

`initFuncParams` the parameters for the initialization function

`learnFunc` the learning function to use

`learnFuncParams` the parameters for the learning function

`updateFunc` the update function to use

`updateFuncParams` the parameters for the update function

`outputMethod` the output method of the net

`maxit` maximum of iterations to learn

`shufflePatterns` should the patterns be shuffled?

`computeError` should the error be computed in every iteration?

`inputsTest` a matrix with inputs to test the network

`targetsTest` the corresponding targets for the test input

`pruneFunc` the pruning function to use

`pruneFuncParams` the parameters for the pruning function. Unlike the other functions, these have to be given in a named list. See the pruning demos for further explanation.

Value

a list containing:

`fitValues` the fitted values, i.e. outputs of the training inputs

`IterativeFitError` The SSE in every iteration/epoch on the training set

`testValues` the predicted values, i.e. outputs of the test inputs

`IterativeTestError` The SSE in every iteration/epoch on the test set

`SnnsRObject$whereAreResults` *Get a list of output units of a net*

Description

SnnsR low-level function to get a list of output units of a net.

Usage

```
## S4 method for signature 'SnnsR'
whereAreResults(outputMethod="output")
```

Arguments

`outputMethod`	a string defining the output method of the net. Possible values are: "art1", "art2", "artmap", "assoz", "som", "output".

Details

Depending on the network architecture, output is present in hidden units, in output units, etc. In some network types, the output units have a certain name prefix in SNNS. This function finds the output units according to certain network types. The type is specified by `outputMethod`. If the given `outputMethod` is unknown, the function defaults to "output".

Value

a list of numbers identifying the units

`som` *Create and train a self-organizing map (SOM)*

Description

This function creates and trains a self-organizing map (SOM). SOMs are neural networks with one hidden layer. The network structure is similar to LVQ, but the method is unsupervised and uses a notion of neighborhood between the units. The general idea is that the map develops by itself a notion of similarity among the input and represents this as spatial nearness on the map. Every hidden unit represents a prototype. The goal of learning is to distribute the prototypes in the feature space such that the (probability density of the) input is represented well. SOMs are usually built with 1d, 2d quadratic, 2d hexagonal, or 3d neighborhood, so that they can be visualized straightforwardly. The SOM implemented in SNNS has a 2d quadratic neighborhood.

As the computation of this function might be slow if many patterns are involved, much of its output is made switchable (see comments on return values).

Usage

```
som(x, ...)

## Default S3 method:
som(
  x,
  mapX = 16,
  mapY = 16,
  maxit = 100,
  initFuncParams = c(1, -1),
  learnFuncParams = c(0.5, mapX/2, 0.8, 0.8, mapX),
  updateFuncParams = c(0, 0, 1),
  shufflePatterns = TRUE,
  calculateMap = TRUE,
  calculateActMaps = FALSE,
  calculateSpanningTree = FALSE,
  saveWinnersPerPattern = FALSE,
  targets = NULL,
  ...
)
```

Arguments

`x`	a matrix with training inputs for the network
`...`	additional function parameters (currently not used)
`mapX`	the x dimension of the som
`mapY`	the y dimension of the som
`maxit`	maximum of iterations to learn
`initFuncParams`	the parameters for the initialization function
`learnFuncParams`	the parameters for the learning function
`updateFuncParams`	the parameters for the update function
`shufflePatterns`	should the patterns be shuffled?
`calculateMap`	should the som be calculated?
`calculateActMaps`	should the activation maps be calculated?
`calculateSpanningTree`	should the SNNS kernel algorithm for generating a spanning tree be applied?
`saveWinnersPerPattern`	should a list with the winners for every pattern be saved?
`targets`	optional target classes of the patterns

Details

Internally, this function uses the initialization function `Kohonen_Weights_v3.2`, the learning function `Kohonen`, and the update function `Kohonen_Order` of SNNS.

Value

an `rsnns` object. Depending on which calculation flags are switched on, the som generates some special members:

- `map` — the som. For each unit, the amount of patterns where this unit won is given.
- `componentMaps` — a map for every input component, showing where in the map this component leads to high activation.
- `actMaps` — a list containing for each pattern its activation map, i.e. all unit activations. The `actMaps` are an intermediary result, from which all other results can be computed. This list can be very long, so normally it won't be saved.
- `winnersPerPattern` — a vector where for each pattern the number of the winning unit is given. Also, an intermediary result that normally won't be saved.
- `labeledUnits` — a matrix which – if the `targets` parameter is given – contains for each unit (rows) and each class present in the `targets` (columns), the amount of patterns of the class where the unit has won. From the `labeledUnits`, the `labeledMap` can be computed, e.g. by voting of the class labels for the final label of the unit.
- `labeledMap` — a labeled som that is computed from `labeledUnits` using `decodeClassLabels`.
- `spanningTree` — the result of the original SNNS function to calculate the map. For each unit, the last pattern where this unit won is present. As the other results are more informative, the spanning tree is only interesting, if the other functions are too slow or if the original SNNS implementation is needed.

References

Kohonen, T. (1988), Self-organization and associative memory, Vol. 8, Springer-Verlag.

Zell, A. et al. (1998), 'SNNS Stuttgart Neural Network Simulator User Manual, Version 4.2', IPVR, University of Stuttgart and WSI, University of Tübingen. https://www.ra.cs.uni-tuebingen.de/SNNS/welcome.html

Zell, A. (1994), Simulation Neuronaler Netze, Addison-Wesley. (in German)

Examples

```
## Not run: demo(som_iris)
## Not run: demo(som_cubeSnnsR)


data(iris)
inputs <- normalizeData(iris[,1:4], "norm")

model <- som(inputs, mapX=16, mapY=16, maxit=500,
                calculateActMaps=TRUE, targets=iris[,5])
```

```
par(mfrow=c(3,3))
for(i in 1:ncol(inputs)) plotActMap(model$componentMaps[[i]],
                                    col=rev(topo.colors(12)))

plotActMap(model$map, col=rev(heat.colors(12)))
plotActMap(log(model$map+1), col=rev(heat.colors(12)))
persp(1:model$archParams$mapX, 1:model$archParams$mapY, log(model$map+1),
     theta = 30, phi = 30, expand = 0.5, col = "lightblue")

plotActMap(model$labeledMap)

model$componentMaps
model$labeledUnits
model$map

names(model)
```

---

splitForTrainingAndTest

Function to split data into training and test set

Description

Split the input and target values to a training and a test set. Test set is taken from the end of the data. If the data is to be shuffled, this should be done before calling this function.

Usage

```
splitForTrainingAndTest(x, y, ratio = 0.15)
```

Arguments

`x`	inputs
`y`	targets
`ratio`	ratio of training and test sets (default: 15% of the data is used for testing)

Value

a named list with the following elements:

`inputsTrain`	a matrix containing the training inputs
`targetsTrain`	a matrix containing the training targets
`inputsTest`	a matrix containing the test inputs
`targetsTest`	a matrix containing the test targets

Examples

```
data(iris)
#shuffle the vector
iris <- iris[sample(1:nrow(iris),length(1:nrow(iris))),1:ncol(iris)]

irisValues <- iris[,1:4]
irisTargets <- decodeClassLabels(iris[,5])

splitForTrainingAndTest(irisValues, irisTargets, ratio=0.15)
```

---

summary.rsnns *Generic summary function for rsnns objects*

Description

Prints out a summary of the network. The printed information can be either all information of the network in the original SNNS file format, or the information given by `extractNetInfo`. This behaviour is controlled with the parameter `origSnnsFormat`.

Usage

```
## S3 method for class 'rsnns'
summary(object, origSnnsFormat = TRUE, ...)
```

Arguments

`object`	the `rsnns` object
`origSnnsFormat`	show data in SNNS's original format in which networks are saved, or show output of `extractNetInfo`
`...`	additional function parameters (currently not used)

Value

Either the contents of the .net file that SNNS would generate from the object, as a string. Or the output of `extractNetInfo`.

See Also

`extractNetInfo`

toNumericClassLabels *Convert a vector (of class labels) to a numeric vector*

Description

This function converts a vector (of class labels) to a numeric vector.

Usage

```
toNumericClassLabels(x)
```

Arguments

x	inputs

Value

the vector converted to a numeric vector

Examples

```
data(iris)
toNumericClassLabels(iris[,5])
```

train *Internal generic train function for rsnns objects*

Description

The function calls `SnnsRObject$train` and saves the result in the current `rsnns` object. This function is used internally by the models (e.g. `mlp`) for training. Unless you are not about to implement a new model on the S3 layer you most probably don't want to use this function.

Usage

```
train(object, ...)

## S3 method for class 'rsnns'
train(
  object,
  inputsTrain,
  targetsTrain = NULL,
  inputsTest = NULL,
  targetsTest = NULL,
  serializeTrainedObject = TRUE,
  ...
)
```

Arguments

`object`	the `rsnns` object
`...`	additional function parameters (currently not used)
`inputsTrain`	training input
`targetsTrain`	training targets
`inputsTest`	test input
`targetsTest`	test targets
`serializeTrainedObject`	parameter passed to `SnnsRObject$train`

Value

an `rsnns` object, to which the results of training have been added.

`vectorToActMap` *Convert a vector to an activation map*

Description

Organize network activation as 2d map.

Usage

```
vectorToActMap(v, nrow = 0, ncol = 0)
```

Arguments

`v`	the vector containing the activation pattern
`nrow`	number of rows the resulting matrices will have
`ncol`	number of columns the resulting matrices will have

Details

The input to this function is a vector containing in each row an activation pattern/output of a neural network. This function reorganizes the vector to a matrix. Normally, only the number of rows `nrow` will be used.

Value

a matrix containing the 2d reorganized input

See Also

matrixToActMapList plotActMap

weightMatrix — *Function to extract the weight matrix of an rsnns object*

Description

The function calls SnnsRObject$getCompleteWeightMatrix and returns its result.

Usage

```
weightMatrix(object, ...)

## S3 method for class 'rsnns'
weightMatrix(object, ...)
```

Arguments

object	the rsnns object
...	additional function parameters (currently not used)

Value

a matrix with all weights from all neurons present in the net.

Index

∗ **SNNS**
RSNNS-package, 4
∗ **data**
snnsData, 47
∗ **networks**
RSNNS-package, 4
∗ **neural**
RSNNS-package, 4
∗ **package**
RSNNS-package, 4
$, *47–49*
$ *(SnnsRObjectMethodCaller)*, 50
$,SnnsR-method
(SnnsRObjectMethodCaller), 50

analyzeClassification, 6, *24*
art1, *5, 6*, 7, *10–12*, *14–16*, *46*
art2, *5–7*, *9*, 10, *14*, *16*, *46*
artmap, *5, 6*, *9*, *11*, 12, *46*
assoz, *5, 6*, 15, *46*

confusionMatrix, 17
createNet,SnnsR-method
(SnnsRObject$createNet), 51
createPatSet,SnnsR-method
(SnnsRObject$createPatSet), 52

decodeClassLabels, 17, *68*
denormalizeData, 18, *25*, *34, 35*
dlvq, *5, 6*, 19, *46*

elman, *5, 6*, 21, *27*, *29*, *46*
encodeClassLabels, *7*, *17*, 23
exportToSnnsNetFile, *24*, 24
extractNetInfo, 24, *70*
extractNetInfo,SnnsR-method
(SnnsRObject$extractNetInfo),
52
extractPatterns,SnnsR-method
(SnnsRObject$extractPatterns),
53

getAllHiddenUnits,SnnsR-method
(SnnsRObject$getAllHiddenUnits),
53
getAllInputUnits,SnnsR-method
(SnnsRObject$getAllInputUnits),
54
getAllOutputUnits,SnnsR-method
(SnnsRObject$getAllOutputUnits),
54
getAllUnits,SnnsR-method
(SnnsRObject$getAllUnits), 55
getAllUnitsTType,SnnsR-method
(SnnsRObject$getAllUnitsTType),
55
getCompleteWeightMatrix,SnnsR-method
(SnnsRObject$getCompleteWeightMatrix),
56
getInfoHeader,SnnsR-method
(SnnsRObject$getInfoHeader), 56
getNormParameters, *19*, 25, *34, 35*
getSiteDefinitions,SnnsR-method
(SnnsRObject$getSiteDefinitions),
57
getSnnsRDefine, 26, *44*
getSnnsRFunctionTable, 26
getTypeDefinitions,SnnsR-method
(SnnsRObject$getTypeDefinitions),
57
getUnitDefinitions,SnnsR-method
(SnnsRObject$getUnitDefinitions),
58
getUnitsByName,SnnsR-method
(SnnsRObject$getUnitsByName),
58
getWeightMatrix,SnnsR-method
(SnnsRObject$getWeightMatrix),
59

initializeNet,SnnsR-method
(SnnsRObject$initializeNet), 59

inputColumns, 27

jordan, *5, 6*, *21–23*, 27, *46*

matrixToActMapList, 30, *36*, *72*
mlp, *5, 6*, 31, *39*, *46*, *71*

normalizeData, *18, 19*, *25*, 33, *34, 35*
normTrainingAndTestSet, 34

outputColumns, 35

plotActMap, *30*, 36, *72*
plotIterativeError, 36
plotRegressionError, 37
plotROC, 37
predict.rsnns, 38
predictCurrPatSet,SnnsR-method
(SnnsRObject$predictCurrPatSet), 60
print.rsnns, 38

rbf, *5, 6*, 39, *46*
rbfDDA, *5, 6*, 41, *46*
readPatFile, 43, *47*
readResFile, 43
resetRSNNS,SnnsR-method
(SnnsRObject$resetRSNNS), 60
resolveSnnsRDefine, *26*, 44
RSNNS (RSNNS-package), 4
rsnns, *5*, *9*, *11*, *14*, *16*, *20*, *22*, *24, 25*, *29*, *32*, *36*, *38*, *40*, *42*, *68*, *70–73*
rsnns (rsnnsObjectFactory), 44
RSNNS-package, 4
rsnnsObjectFactory, 44

savePatFile, 46
setSnnsRSeedValue, 46
setTTypeUnitsActFunc,SnnsR-method
(SnnsRObject$setTTypeUnitsActFunc), 61
setUnitDefaults,SnnsR-method
(SnnsRObject$setUnitDefaults), 61
snnsData, *5*, 47
SnnsR-class, *43*, *46*, 47, *49, 50*
SnnsR__createNet
(SnnsRObject$createNet), 51
SnnsR__createPatSet
(SnnsRObject$createPatSet), 52
SnnsR__extractNetInfo
(SnnsRObject$extractNetInfo), 52
SnnsR__extractPatterns
(SnnsRObject$extractPatterns), 53
SnnsR__getAllHiddenUnits
(SnnsRObject$getAllHiddenUnits), 53
SnnsR__getAllInputUnits
(SnnsRObject$getAllInputUnits), 54
SnnsR__getAllOutputUnits
(SnnsRObject$getAllOutputUnits), 54
SnnsR__getAllUnits
(SnnsRObject$getAllUnits), 55
SnnsR__getAllUnitsTType
(SnnsRObject$getAllUnitsTType), 55
SnnsR__getCompleteWeightMatrix
(SnnsRObject$getCompleteWeightMatrix), 56
SnnsR__getInfoHeader
(SnnsRObject$getInfoHeader), 56
SnnsR__getSiteDefinitions
(SnnsRObject$getSiteDefinitions), 57
SnnsR__getTypeDefinitions
(SnnsRObject$getTypeDefinitions), 57
SnnsR__getUnitDefinitions
(SnnsRObject$getUnitDefinitions), 58
SnnsR__getUnitsByName
(SnnsRObject$getUnitsByName), 58
SnnsR__getWeightMatrix
(SnnsRObject$getWeightMatrix), 59
SnnsR__initializeNet
(SnnsRObject$initializeNet), 59
SnnsR__predictCurrPatSet
(SnnsRObject$predictCurrPatSet), 60
SnnsR__resetRSNNS
(SnnsRObject$resetRSNNS), 60
SnnsR__setTTypeUnitsActFunc

(SnnsRObject$setTTypeUnitsActFunc), 61
SnnsR__setUnitDefaults *(SnnsRObject$setUnitDefaults)*, 61
SnnsR__somPredictComponentMaps *(SnnsRObject$somPredictComponentMaps)*, 62
SnnsR__somPredictCurrPatSetWinners *(SnnsRObject$somPredictCurrPatSetWinners)*, 63
SnnsR__somPredictCurrPatSetWinnersSpanTree *(SnnsRObject$somPredictCurrPatSetWinnersSpanTree)*, 64
SnnsR__train *(SnnsRObject$train)*, 64
SnnsR__whereAreResults *(SnnsRObject$whereAreResults)*, 66
SnnsRObject$createNet, 51
SnnsRObject$createPatSet, 52
SnnsRObject$extractNetInfo, *25*, 52
SnnsRObject$extractPatterns, *43*, 53
SnnsRObject$getAllHiddenUnits, 53, *55*
SnnsRObject$getAllInputUnits, 54, *55*
SnnsRObject$getAllOutputUnits, 54, *55*
SnnsRObject$getAllUnits, 55
SnnsRObject$getAllUnitsTType, *53, 54*, 55, *59*, *61*
SnnsRObject$getCompleteWeightMatrix, 56, *73*
SnnsRObject$getInfoHeader, 56
SnnsRObject$getSiteDefinitions, 57
SnnsRObject$getTypeDefinitions, 57
SnnsRObject$getUnitDefinitions, 58
SnnsRObject$getUnitsByName, 58
SnnsRObject$getWeightMatrix, 59
SnnsRObject$initializeNet, *50*, 59
SnnsRObject$predictCurrPatSet, 60
SnnsRObject$resetRSNNS, 60
SnnsRObject$setTTypeUnitsActFunc, 61
SnnsRObject$setUnitDefaults, 61
SnnsRObject$somPredictComponentMaps, 62
SnnsRObject$somPredictCurrPatSetWinners, 63
SnnsRObject$somPredictCurrPatSetWinnersSpanTree, 64
SnnsRObject$train, *50*, 64, *71, 72*
SnnsRObject$whereAreResults, *60*, 66
SnnsRObjectFactory, *47, 48*, 49
SnnsRObjectMethodCaller, 50
som, *5, 6*, *19*, *46*, *62–64*, 66
somPredictComponentMaps,SnnsR-method *(SnnsRObject$somPredictComponentMaps)*, 62
somPredictCurrPatSetWinners,SnnsR-method *(SnnsRObject$somPredictCurrPatSetWinners)*, 63
somPredictCurrPatSetWinnersSpanTree,SnnsR-method *(SnnsRObject$somPredictCurrPatSetWinnersSpanTree)*, 64
splitForTrainingAndTest, *34, 35*, 69
summary.rsnns, *24*, 70

toNumericClassLabels, 71
train, *46*, 71
train,SnnsR-method *(SnnsRObject$train)*, 64

vectorToActMap, *30*, *36*, 72

weightMatrix, 73
whereAreResults,SnnsR-method *(SnnsRObject$whereAreResults)*, 66