Package 'nnlib2Rcpp'

April 25, 2024

Type Package

Title A Tool for Creating Custom Neural Networks in C++ and using Them in R

Version 0.2.6

Author Vasilis Nikolaidis [aut, cph, cre] (<https://orcid.org/0000-0003-1471-8788>)

Maintainer Vasilis Nikolaidis <v.nikolaidis@uop.gr>

Description Contains a module to define neural networks from custom components and versions of Autoencoder, BP, LVQ, MAM NN.

LinkingTo Rcpp

Imports Rcpp , methods

License MIT + file LICENSE

URL https://github.com/VNNikolaidis/nnlib2Rcpp

BugReports https://github.com/VNNikolaidis/nnlib2Rcpp/issues

Encoding UTF-8

Suggests R.rsp

VignetteBuilder R.rsp

NeedsCompilation yes

Repository CRAN

Date/Publication 2024-04-25 12:00:03 UTC

R topics documented:

`nnlib2Rcpp-package` *A collection of Neural Networks and tools to create custom models*

Description

This package provides a module (NN module) to define and control neural networks containing predefined or custom components (layers, sets of connnections etc.). These components may have been derived from nnlib2 NN components (written in C++) or defined using R.

It also contains a small collection of ready-to-use Neural Networks (NN), i.e. versions of Autoencoder, Back-Propagation, Learning Vector Quantization and Matrix Associative Memory NN. More information and examples for each of the above can be found in its documentation (see below).

Ready-to-use Neural Networks:

- Plain Back-Propagation (BP-supervised) (`BP`)
- Learning Vector Quantization (LVQ-supervised) (`LVQs`)
- Learning Vector Quantization (LVQ-unsupervised) (`LVQu`)
- Matrix Associative Memory (MAM-supervised) (`MAM`)
- Autoencoder (unsupervised) (`Autoencoder`)

Custom Neural Networks:

- NN module (`NN`)

Author(s)

Author/Maintainer:

- Vasilis Nikolaidis <vnnikolaidis@gmail.com>

Contributors:

- Arfon Smith [contributor]
- Dirk Eddelbuettel [contributor]

References

- Nikolaidis, V. N., (2021). The nnlib2 library and nnlib2Rcpp R package for implementing neural networks. Journal of Open Source Software, 6(61), 2876, doi:10.21105/joss.02876.

References for the ready-to-use NN models (can also be found in related documentation):

- Kohonen, T (1988). Self-Organization and Associative Memory, Springer-Verlag.; Simpson, P. K. (1991). Artificial neural systems: Foundations, paradigms, applications, and implementations. New York: Pergamon Press.
- Pao Y (1989). Adaptive Pattern Recognition and Neural Networks. Reading, MA (US); Addison-Wesley Publishing Co., Inc.

- Simpson, P. K. (1991). Artificial neural systems: Foundations, paradigms, applications, and implementations. New York: Pergamon Press.
- Philippidis, TP & Nikolaidis, VN & Kolaxis, JG. (1999). Unsupervised pattern recognition techniques for the prediction of composite failure. Journal of acoustic emission. 17. 69-81.
- Nikolaidis V.N., Makris I.A, Stavroyiannis S, "ANS-based preprocessing of company performance indicators." Global Business and Economics Review 15.1 (2013): 49-58, doi:10.1504/GBER.2013.050667.

See Also

More information and examples on using the package can be found in the following vignette:

```
vignette("manual", package='nnlib2Rcpp')
```

Related links:

- https://github.com/VNNikolaidis/nnlib2Rcpp
- Package manual in PDF format at https://github.com/VNNikolaidis/nnlib2Rcpp/blob/master/support/manual.pdf)
- Report bugs, issues and suggestions at https://github.com/VNNikolaidis/nnlib2Rcpp/issues

---

`Autoencoder` *Autoencoder NN*

Description

A neural network for autoencoding data, projects data to a new set of variables.

Usage

```
Autoencoder(
  data_in,
  desired_new_dimension,
  number_of_training_epochs,
  learning_rate,
  num_hidden_layers = 1L,
  hidden_layer_size = 5L,
  show_nn = FALSE,
  error_type = "MAE",
  acceptable_error_level = 0,
  display_rate = 1000)
```

Arguments

`data_in`	data to be autoencoded, a numeric matrix, (2d, cases in rows, variables in columns). It is recommended to be in [0 1] range.
`desired_new_dimension`	number of new variables to be produced. This is effectively the size (length) of the special hidden layer that outputs the new variable values, thus the dimension of the output vector space.
`number_of_training_epochs`	number of training epochs, aka presentations of all training data to ANN during training.
`learning_rate`	the learning rate parameter of the Back-Propagation (BP) NN.
`num_hidden_layers`	number of hidden layers on each side of the special layer.
`hidden_layer_size`	number of nodes (processing elements or PEs) in each of the hidden layers. In this implementation of Autoencoder all hidden layers are of the same length (defined here), except for the special hidden layer (whose size is defined by `desired_new_dimension` above).
`show_nn`	boolean, option to display the (trained) ANN internal structure.
`error_type`	string, error to display and possibly use to stop training (must be 'MSE' or 'MAE').
`acceptable_error_level`	stops training when error is below this level.
`display_rate`	number of epochs that pass before current error level is displayed (0 = never display current error).

Value

Returns a numeric matrix containing the projected data.

Note

This Autoencoder NN employs a BP-type NN to perform a data pre-processing step baring similarities to PCA since it too can be used for dimensionality reduction (Kramer 1991)(DeMers and Cottrell 1993)(Hinton and Salakhutdinov 2006). Unlike PCA, an autoencoding NN can also expand the feature-space dimensions (as feature expansion methods do). The NN maps input vectors to themselves via a special hidden layer (the coding layer, usually of different size than the input vector length) from which the new data vectors are produced. Note: The internal BP PEs in computing layers apply the logistic sigmoid threshold function, and their output is in [0 1] range. It is recommended to use this range in your data as well. More for this particular autoencoder implementation can be found in (Nikolaidis, Makris, and Stavroyiannis 2013). The method is not deterministic and the mappings may be non-linear, depending on the NN topology.

(This function uses Rcpp to employ 'bpu_autoencoder_nn' class in nnlib2.)

Author(s)

Vasilis N. Nikolaidis <vnnikolaidis@gmail.com>

References

Nikolaidis V.N., Makris I.A, Stavroyiannis S, "ANS-based preprocessing of company performance indicators." Global Business and Economics Review 15.1 (2013): 49-58.

See Also

BP.

Examples

```
iris_s <- as.matrix(scale(iris[1:4]))
output_dim <- 2
epochs <- 100
learning_rate <- 0.73
num_hidden_layers <-2
hidden_layer_size <- 5

out_data <-  Autoencoder( iris_s, output_dim,
                          epochs, learning_rate,
                          num_hidden_layers, hidden_layer_size, FALSE)

plot( out_data,pch=21,
      bg=c("red","green3","blue")[unclass(iris$Species)],
      main="Randomly autoencoded Iris data")
```

BP-class *Class* "BP"

Description

Supervised Back-Propagation (BP) NN module, for encoding input-output mappings.

Extends

Class "RcppClass", directly.

All reference classes extend and inherit methods from "envRefClass".

Fields

.CppObject: Object of class C++Object ~~

.CppClassDef: Object of class activeBindingFunction ~~

.CppGenerator: Object of class activeBindingFunction ~~

Methods

`encode( data_in, data_out, learning_rate, training_epochs, hidden_layers, hidden_layer_size ):` Setup a new BP NN and encode input-output data pairs. Parameters are:

- `data_in`: numeric matrix, containing input vectors as rows. . It is recommended that these values are in 0 to 1 range.
- `data_out`: numeric matrix, containing corresponding (desired) output vectors. It is recommended that these values are in 0 to 1 range.
- `learning_rate`: a number (preferably greater than 0 and less than 1) used in training.
- `training_epochs`: number of training epochs, aka single presentation iterations of all training data pairs to the NN during training.
- `hidden_layers`: number of hidden layers to be created between input and output layers.
- `hidden_layer_size`: number of nodes (processing elements or PEs) in each of the hidden layers (all hidden layers are of the same length in this implementation of BP).

Note: to encode additional input-output vector pairs in an existing BP, use `train_single` or `train_multiple` methods (see below).

`recall(data_in):` Get output for a dataset (numeric matrix `data_in`) from the (trained) BP NN.

`setup(input_dim, output_dim, learning_rate, hidden_layers, hidden_layer_size):` Setup the BP NN so it can be trained and used. Note: this is not needed if using `encode`. Parameters are:

- `input_dim`: integer length of input vectors.
- `output_dim`: integer length of output vectors.
- `learning_rate`: a number (preferably greater than 0 and less than 1) used in training.
- `hidden_layers`: number of hidden layers to be created between input and output layers.
- `hidden_layer_size`: number of nodes (processing elements or PEs) in each of the hidden layers (all hidden layers are of the same length in this implementation of BP).

`train_single (data_in, data_out):` Encode an input-output vector pair in the BP NN. Only performs a single training iteration (multiple may be required for proper encoding). Vector sizes should be compatible to the current NN (as resulted from the `encode` or `setup` methods). Returns error level indicator value.

`train_multiple (data_in, data_out, training_epochs):` Encode multiple input-output vector pairs stored in corresponding datasets. Performs multiple iterations in epochs (see `encode`). Vector sizes should be compatible to the current NN (as resulted from the `encode` or `setup` methods). Returns error level indicator value.

`set_error_level(error_type, acceptable_error_level):` Set options that stop training when an acceptable error level has been reached (when a subsequent `encode` or `train_multiple` is performed). Parameters are:

- `error_type`: string, error type to display and use to stop training (must be 'MSE' or 'MAE').
- `acceptable_error_level`: training stops when error is below this level.

`mute(on):` Disable output of current error level when training (if parameter `on` is TRUE).

`print():` Print NN structure.

`show():` Print NN structure.

`load(filename):` Retrieve the NN from specified file.

`save(filename)`: Save the NN to specified file.

The following methods are inherited (from the corresponding class): objectPointer ("RcppClass"), initialize ("RcppClass"), show ("RcppClass")

Note

This R module maintains an internal Back-Propagation (BP) multilayer perceptron NN (described in Simpson (1991) as the vanilla back-propagation algorithm), which can be used to store input-output vector pairs. Since the nodes (PEs) in computing layers of this BP implementation apply the logistic sigmoid threshold function, their output is in [0 1] range (and so should the desired output vector values).

(This object uses Rcpp to employ 'bp_nn' class in nnlib2.)

Author(s)

Vasilis N. Nikolaidis <vnnikolaidis@gmail.com>

References

Simpson, P. K. (1991). Artificial neural systems: Foundations, paradigms, applications, and implementations. New York: Pergamon Press.

See Also

Autoencoder.

Examples

```
# create some data...
iris_s                  <- as.matrix(scale(iris[1:4]))

# use a randomly picked subset of (scaled) iris data for training.
training_cases          <- sample(1:nrow(iris_s), nrow(iris_s)/2,replace=FALSE)
train_set               <- iris_s[training_cases,]
train_class_ids         <- as.integer(iris$Species[training_cases])
train_num_cases         <- nrow(train_set)
train_num_variables     <- ncol(train_set)
train_num_classes       <- max(train_class_ids)

# create output dataset to be used for training.
# Here we encode class as 0s and 1s (one-hot encoding).

train_set_data_out <- matrix(
          data = 0,
          nrow = train_num_cases,
          ncol = train_num_classes)

# now for each case, assign a 1 to the column corresponding to its class, 0 otherwise
# (note: there are better R ways to do this in R)
for(r in 1:train_num_cases) train_set_data_out[r,train_class_ids[r]]=1
```

```
# done with data, let's use BP...
bp<-new("BP")

bp$encode(train_set,train_set_data_out,0.8,10000,2,4)

# let's test by recalling the original training set...
bp_output <- bp$recall(train_set)

cat("- Using this demo's encoding, recalled class is:\n")
print(apply(bp_output,1,which.max))
cat("- BP success in recalling correct class is: ",
  sum(apply(bp_output,1,which.max)==train_class_ids)," out of ",
  train_num_cases, "\n")

# Let's see how well it recalls the entire Iris set:
bp_output <- bp$recall(iris_s)

# show output
cat("\n- Recalling entire Iris set returns:\n")
print(bp_output)
cat("- Using this demo's encoding, original class is:\n")
print(as.integer(iris$Species))
cat("- Using this demo's encoding, recalled class is:\n")
bp_classification <- apply(bp_output,1,which.max)
print(bp_classification)
cat("- BP success in recalling correct class is: ",
  sum(apply(bp_output,1,which.max)==as.integer(iris$Species)),
  "out of ", nrow(iris_s), "\n")
plot(iris_s, pch=bp_classification, main="Iris classified by a partialy trained BP (module)")
```

`LVQs-class` *Class* `"LVQs"`

Description

Supervised Learning Vector Quantization (LVQ) NN module, for data classification.

Extends

Class `"RcppClass"`, directly.

All reference classes extend and inherit methods from `"envRefClass"`.

Fields

`.CppObject:` Object of class `C++Object` ~~

`.CppClassDef:` Object of class `activeBindingFunction` ~~

`.CppGenerator:` Object of class `activeBindingFunction` ~~

Methods

`encode(data, desired_class_ids, training_epochs)`: Encode input and output (classification) for a dataset using a LVQ NN (which sets up accordingly if required). Parameters are:

- `data`: training data, a numeric matrix, (2d, cases in rows, variables in columns). Data should be in 0 to 1 range.
- `desired_class_ids` : vector of integers containing a desired class id for each training data case (row). Should contain integers in 0 to n-1 range, where n is the number of classes.
- `training_epochs`: integer, number of training epochs, aka presentations of all training data to the NN during training.

`recall(data_in, min_rewards)`: Get output (classification) for a dataset (numeric matrix `data_in`) from the (trained) LVQ NN. The `data_in` dataset should be 2-d containing data cases (rows) to be presented to the NN and is expected to have same number or columns as the original training data. Returns a vector of integers containing a class id for each case (row).Parameters are:

- `data_in`: numeric 2-d matrix containing data cases (as rows).
- `min_rewards`: (optional) integer, ignore output nodes that (during encoding/training) were rewarded less times that this number (default is 0, i.e. use all nodes).

`setup( input_length, int number_of_classes, number_of_nodes_per_class )`: Setup an untrained supervised LVQ for given input data vector dimension and number of classes. Parameters are:

- `input_length`: integer, dimension (length) of input data vectors.
- `number_of_classes`: integer, number of classes in data (including empty ones).
- `number_of_nodes_per_class`: (optional) integer, number of output nodes (PE) to be used per class. Default is 1.

`train_single (data_in, class_id, epoch)`: Encode a single [input vector,class] pair in the LVQ NN. Only performs a single training iteration (multiple may be required for proper encoding). Vector length and class id should be compatible to the current NN (as resulted from the `encode`, `setup` or `load` methods). Returns `TRUE` if succesfull, `FALSE` otherwise. Parameters are:

- `data_in`: numeric, data vector to be encoded.
- `class_id`: integer, id of class corresponding to the data vector.(ids start from 0).
- `epoch`: integer, presumed epoch during which this encoding occurs (learning rate decreases with epochs in supervised LVQ).

`get_weights()`: Get the current weights (codebook vector coordinates) of the 2nd component (`connection_set`). If successful, returns NumericVector of connection weights (otherwise vector of zero length).

`set_weights( data_in )`: Set the weights of the 2nd component (`connection_set`), i.e. directly define the LVQ's codebook vectors. If successful, returns TRUE. Parameters are:

- `data_in`: NumericVector, data to be used for new values in `weight` registers of connections (sizes must match).

`set_number_of_nodes_per_class( n )`: Set the number of nodes in the output layer (and thus incoming connections whose weights form codebook vectors) that will be used per class. Default is 1, i.e. each class in the data to be encoded in the NN corresponds to a single

node (PE) in it's output layer. This method affects how the new NN topology will be created, therefore this method should be used before the NN has been set up (either by encode or setup) or after a NN topology (and NN state) has been loaded from file via load). Returns number of nodes to be used per class. Parameters are:

- n: integer, number of nodes to be used per each class.

get_number_of_nodes_per_class( ): Get the number of nodes in the output layer that are used per class.

enable_punishment( ): Enables negative reinfoncement. During encoding incorrect winner nodes will be notified and incoming weights will be adjusted accordingly. Returns TRUE if punishment is enabled, FALSE otherwise.

disable_punishment( ): Disables negative reinfoncement. During encoding incorrect winner nodes will not be notified, thus incoming weights will not be adjusted accordingly. Adjustments will only occur in correct winning nodes. Returns TRUE if punishment is enabled, FALSE otherwise.

get_number_of_rewards( ): Get the number of times an output node was positively reinforced during data encoding. Returns NumericVector containing results per output node.

set_weight_limits(min, max): Define the minimum and maximum values that will be allowed in connection weights during encoding (limiting results of punishment). The NN must have been set up before using this method (either by encode, setup or load). Parameters are:

- min: numeric, minimum weight allowed.
- max: numeric, maximum weight allowed.

set_encoding_coefficients(reward, punish): Define coefficients used for reward and punishment during encoding. In this version, the actual learning rate a(t) also depends on the epoch t, i.e. a(t) = coefficient * (1 - (t/10000)). The NN must have been set up before using this method (either by encode, setup or load). Parameters are:

- reward: numeric, coefficient used to reward a node that classified data correctly (usually positive, e.g. 0.2).
- punish: numeric, coefficient used to punish a node that classified data incorrectly (usually negative, e.g. -0.2).

print(): print NN structure.

show(): print NN structure.

load(filename): Retrieve the state of the NN from specified file. Note: parameters such as number of nodes per class or reward/punish coefficients are not retrieved.

save(filename): Store the state of the current NN to specified file. Note: parameters such as number of nodes per class or reward/punish coefficients are not stored.

The following methods are inherited (from the corresponding class): objectPointer ("RcppClass"), initialize ("RcppClass"), show ("RcppClass")

Note

This module uses Rcpp to employ 'lvq_nn' class in nnlib2. The NN used in this module uses supervised training for data classification (described as Supervised Learning LVQ in Simpson (1991)). Initial weights are random (uniform distribution) in 0 to 1 range. As these weights represent vector coordinates (forming the class reference, prototype or codebook vector(s)), it is important that input data is also scaled to 0 to 1 range.

Author(s)

Vasilis N. Nikolaidis <vnnikolaidis@gmail.com>

References

Simpson, P. K. (1991). Artificial neural systems: Foundations, paradigms, applications, and implementations. New York: Pergamon Press. p.88.

See Also

LVQu (unsupervised LVQ function).

Examples

```
# Create some compatible data (here, from iris data set):

# LVQ expects data in 0 to 1 range, so scale some numeric data...
DATA <- as.matrix(iris[1:4])
c_min <- apply(DATA, 2, FUN = "min")
c_max <- apply(DATA, 2, FUN = "max")
c_rng <- c_max - c_min
DATA <- sweep(DATA, 2, FUN = "-", c_min)
DATA <- sweep(DATA, 2, FUN = "/", c_rng)
NUM_VARIABLES <- ncol(DATA)

# create a vector of desired class ids (consecutive ids, starting from 0):
CLASS <- as.integer(iris$Species) - 1
NUM_CLASSES <- length(unique(CLASS))

# avoid using data with NA or other special values:
if (sum(is.na(DATA)) > 0)
  stop("NAs found in DATA")
if (sum(is.na(CLASS)) > 0)
  stop("NAs found in CLASS")

# Example 1:
# (Note: the example uses DATA and CLASS variables defined earlier).

  # use half of the data to train, the other half to evaluate how well LVQ was
  # trained (interlaced half is used to select members of these data sets):

  l1_train_dataset <- DATA[c(TRUE, FALSE),]
  l1_train_class   <- CLASS[c(TRUE, FALSE)]
  l1_test_dataset <- DATA[c(FALSE, TRUE),]
  l1_test_class   <- CLASS[c(FALSE, TRUE)]

  # now create the NN:
  l1 <- new("LVQs")

  # train it:
  l1$encode(l1_train_dataset, l1_train_class, 100)
```

```
  # recall the same data (a simple check of how well the LVQ was trained):
  l1_recalled_class_ids <- l1$recall(l1_test_dataset)

  # show results:
  cat(
    "Example 1 results: Correct ",
    sum(l1_recalled_class_ids == l1_test_class),
    "out of",
    nrow(l1_test_dataset),
    ".\n"
  )

# Example 2: (playing around with some optional settings)
# (Note: the example uses DATA, CLASS, NUM_CLASSES variables defined earlier).

  # create the NN:
  l2 <- new("LVQs")

  # Optionally, the output layer could be expanded, e.g. use 2 nodes per each class:
  l2$set_number_of_nodes_per_class(2)

  # Optionally, for experimentation negative reinforcement can be disabled:
  l2$disable_punishment()

  # train it:
  l2$encode(DATA, CLASS, 100)

  # recall the same data (a simple check of how well the LVQ was trained):
  l2_recalled_class_ids <- l2$recall(DATA)

  # Done. Optional part for further examining results of training:

  # collect the connection weights (codebook vector coordinates), number
  # of rewards per node and corresponding class:

  l2_codebook_vector_info <-
    cbind(
      matrix(l2$get_weights(),
             ncol = ncol(DATA),
             byrow = TRUE),
      l2$get_number_of_rewards(),
      rep(
        0:(NUM_CLASSES - 1),
        rep(l2$get_number_of_nodes_per_class(),
            NUM_CLASSES)
      )
    )

  colnames(l2_codebook_vector_info) <-
    c(colnames(DATA), "Rewarded", "Class")

  print(l2_codebook_vector_info)
```

```
  # plot recalled classification:

  plot(
    DATA,
    pch = l2_recalled_class_ids,
    main = "LVQ recalled clusters (LVQs module)",
    xlim = c(-0.2, 1.2),
    ylim = c(-0.2, 1.2)
  )

  # plot connection weights (a.k.a codebook vectors):
  # the big circles are codebook vectors, (crossed-out if they were never used
  # to assign a training data point to the correct class, i.e. never rewarded)

  points(
    l2_codebook_vector_info[, 1:2],
    cex = 4,
    pch = ifelse(l2_codebook_vector_info[, "Rewarded"] > 0,1, 13),
    col  = l2_codebook_vector_info[, "Class"] + 10
  )

  # show results:
  cat(
    "Example 2 results: Correct ",
    sum(l2_recalled_class_ids == CLASS),
    "out of",
    nrow(DATA),
    ".\n"
  )

# Example 3 (demonstrate 'setup' and some other methods it allows):
# (Note: uses DATA, CLASS, NUM_VARIABLES, NUM_CLASSES defined earlier).

  # create the NN:
  l3 <- new("LVQs")

  l3_number_of_output_nodes_per_class <- 3

  # setup the LVQ:
  l3$setup(NUM_VARIABLES,
           NUM_CLASSES,
           l3_number_of_output_nodes_per_class)
  l3$set_weight_limits(-0.5 , 1.5)
  l3$set_encoding_coefficients(0.2,-sum(CLASS == 0) / length(CLASS))

  # experiment with setting initial weights (codebook vectors) per output node;
  # here, weights are set to the mean vector of the training set data for the
  # class the output node corresponds to:

  class_means <- aggregate(DATA, list(CLASS), FUN = mean)
  class_means <- t(class_means)[-1,]
  l3_initial_weights <- NULL
  for (i in 1:l3_number_of_output_nodes_per_class)
```

```
  l3_initial_weights <- rbind(l3_initial_weights, class_means)

l3$set_weights(as.vector(l3_initial_weights))

# now train it:
l3$encode(DATA, CLASS, 100)

# recall the same data (a simple check of how well the LVQ was trained):
l3_recalled_class_ids <- l3$recall(DATA, 0)

# show results:
cat(
  "Example 3 results: Correct ",
  sum(l3_recalled_class_ids == CLASS),
  "out of",
  nrow(DATA),
  ".\n"
)
```

LVQu *Unsupervised LVQ*

Description

Unsupervised (clustering) Learning Vector Quantization (LVQ) NN.

Usage

```
LVQu(
  data,
  max_number_of_desired_clusters,
  number_of_training_epochs,
  neighborhood_size,
  show_nn )
```

Arguments

data	data to be clustered, a numeric matrix, (2d, cases in rows, variables in columns). Data should be in 0 to 1 range.
max_number_of_desired_clusters	clusters to be produced (at most)
number_of_training_epochs	number of training epochs, aka presentations of all training data to ANN during training.
neighborhood_size	integer >=1, specifies affected neighbor output nodes during training. if 1 (Single Winner) the ANN is somewhat similar to k-means.
show_nn	boolean, option to display the (trained) ANN internal structure.

Value

Returns a vector of integers containing a cluster id for each data case (row).

Note

Function LVQu employs an unsupervised LVQ for clustering data (Kohonen 1988). This LVQ variant is described as Unsupervised Learning LVQ in Simpson (1991) and is a simplified 1-D version of Self-Organizing-Map (SOM). Its parameter `neighborhood_size` controls the encoding mode (where `neighborhood_size=1` is Single-Winner Unsupervised encoding, similar to k-means, while an odd valued `neighborhood_size > 1` means Multiple-Winner Unsupervised encoding mode). Initial weights are random (uniform distribution) in 0 to 1 range. As these weights represent cluster center coordinates (the class reference vector), it is important that input data is also scaled to this range.

(This function uses Rcpp to employ 'som_nn' class in nnlib2.)

Author(s)

Vasilis N. Nikolaidis <vnnikolaidis@gmail.com>

References

Kohonen, T (1988). Self-Organization and Associative Memory, Springer-Verlag.; Simpson, P. K. (1991). Artificial neural systems: Foundations, paradigms, applications, and implementations. New York: Pergamon Press.

Philippidis, TP & Nikolaidis, VN & Kolaxis, JG. (1999). Unsupervised pattern recognition techniques for the prediction of composite failure. Journal of acoustic emission. 17. 69-81.

See Also

`LVQs` (supervised LVQ module),

Examples

```
# LVQ expects data in 0 to 1 range, so scale...
iris_s<-as.matrix(iris[1:4])
c_min<-apply(iris_s,2,FUN = "min")
c_max<-apply(iris_s,2,FUN = "max")
c_rng<-c_max-c_min
iris_s<-sweep(iris_s,2,FUN="-",c_min)
iris_s<-sweep(iris_s,2,FUN="/",c_rng)

cluster_ids<-LVQu(iris_s,5,100)
plot(iris_s, pch=cluster_ids, main="LVQ-clustered Iris data")
```

MAM-class *Class* "MAM"

Description

A single Matrix Associative Memory (MAM) implemented as a (supervised) NN.

Extends

Class "RcppClass", directly.

All reference classes extend and inherit methods from "envRefClass".

Fields

`.CppObject:` Object of class `C++Object` ~~

`.CppClassDef:` Object of class `activeBindingFunction` ~~

`.CppGenerator:` Object of class `activeBindingFunction` ~~

Methods

`encode( data_in, data_out ):` Setup a new MAM NN and encode input-output data pairs. Parameters are:

- `data_in`: numeric matrix, input data to be encoded in MAM, a numeric matrix (2d, of n rows). Each row will be paired to the corresponding data_out row, forming an input-output vector pair.
- `data_out`: numeric matrix, output data to be encoded in MAM, a numeric matrix (2d, also of n rows). Each row will be paired to the corresponding data_in row, forming an input-output vector pair.

Note: to encode additional input-output vector pairs in an existing MAM, use `train_single` method (see below).

`recall(data):` Get output for a dataset (numeric matrix `data`) from the (trained) MAM NN.

`train_single (data_in, data_out):` Encode an input-output vector pair in the MAM NN. Vector sizes should be compatible to the current NN (as resulted from the `encode` method).

`print():` print NN structure.

`show():` print NN structure.

`load(filename):` retrieve the NN from specified file.

`save(filename):` save the NN to specified file.

The following methods are inherited (from the corresponding class): objectPointer ("RcppClass"), initialize ("RcppClass"), show ("RcppClass")

Note

The NN in this module uses supervised training to store input-output vector pairs.

(This function uses Rcpp to employ 'mam_nn' class in nnlib2.)

Author(s)

Vasilis N. Nikolaidis <vnnikolaidis@gmail.com>

References

Pao Y (1989). Adaptive Pattern Recognition and Neural Networks. Reading, MA (US); Addison-Wesley Publishing Co., Inc.

See Also

BP, LVQs.

Examples

```
iris_s            <- as.matrix(scale(iris[1:4]))
class_ids         <- as.integer(iris$Species)
num_classes       <- max(class_ids)

# create output dataset to be used for training, Here we encode class as -1s and 1s
iris_data_out <- matrix( data = -1, nrow = nrow(iris_s), ncol = num_classes)

# now for each case, assign a 1 to the column corresponding to its class
for(r in 1:nrow(iris_data_out)) iris_data_out[r,class_ids[r]]=1

# Finally apply MAM:
# Encode train pairs in MAM and then get output dataset by recalling the test data.

mam <- new("MAM")

mam$encode(iris_s,iris_data_out)

# test the encoding by recalling the original input data...
mam_data_out <- mam$recall(iris_s)

# find which MAM output has the largest value and use this as the final cluster tag.
mam_recalled_cluster_ids = apply(mam_data_out,1,which.max)

plot(iris_s, pch=mam_recalled_cluster_ids, main="MAM recalled Iris data classes")

cat("MAM recalled these IDs:\n")
print(mam_recalled_cluster_ids)
```

NN-class *Class* "NN"

Description

NN module, for defining and manipulating custom neural networks.

Extends

Class "RcppClass", directly.

All reference classes extend and inherit methods from "envRefClass".

Fields

.CppObject: Object of class C++Object ~~

.CppClassDef: Object of class activeBindingFunction ~~

.CppGenerator: Object of class activeBindingFunction ~~

Methods

add_layer(name, size, optional_parameter):

add_layer(parameters): Setup a new layer component (a layer of processing nodes) and append it to the NN topology. Returns TRUE if successful. Parameters are:

- name: string, containing name (that also Specifies type) of new layer. Names of predefined layers currently include 'pe'(same as 'generic'), 'pass-through', 'which-max', 'MAM', 'LVQ-input', 'LVQ-output', 'BP-hidden', 'BP-output', 'perceptron' (Some of the available names are listed in NN_component_names, additional names for user-defined components may be added, see note.)
- size: integer, layer size i.e. number of pe (Processing Elements or nodes) to create in the layer.
- optional_parameter: (optional) double, parameter to be used by specific layer implementations (for example, BP layer implementations 'BP-hidden' and 'BP-output' interpret it is as the layer's learning rate). Note: for more optional parameters use parameters below.
- parameters: list, containing named parameters to be used in creating the layer. Must include an element named name and an element called size (similar to the corresponding standalone parameters described above).

add_connection_set(name, optional_parameter):

add_connection_set(parameters): Create a new empty connection_set component (a set of connections between two layers). It does not connect any layers nor contain any connections between specific layer nodes. The set is appended to the NN topology. Returns TRUE if successful. Parameters are:

- name: string, containing name (that also specifies type) of new empty connection set. Names of predefined connection sets currently include 'generic', 'pass-through'(which does not multiply weights), 'wpass-through'(which does multiply weights), 'MAM', 'LVQ', 'BP', 'perceptron' (Some of the available names are listed in NN_component_names, additional names for user-defined components may be added, see note.).
- optional_parameter: (optional) double, parameter to be used by specific connection set implementations (for example, 'BP' connection sets interpret it is as the learning rate to be used when adjusting weights, 'LVQ' connection sets use it to count iterations for decreasing weight adjustments, etc). Note: for more optional parameters use parameters below.

- parameters: list, containing named parameters to be used in creating the connection set. Must include an element named name which contains the name (that also specifies type) of new empty connection set (similar to the corresponding standalone parameter described above).

create_connections_in_sets(min_random_weight, max_random_weight): Find empty, unconnected connection_set components that are between two layers in the topology, and set them up to connect the adjacent layers, adding connections to fully connect their nodes (n x m connections created, n and m the number of nodes at each layer respectively). Assumes top layer is source and bottom layer is destination. Returns TRUE if successful. Parameters are:

- min_random_weight: double, minimum value for random initial connection weights.
- max_random_weight: double, maximum value for random initial connection weights.

connect_layers_at(source_pos, destin_pos, name, optional_parameter):

connect_layers_at(source_pos, destin_pos, parameters): Insert a new empty connection_set component (a set of connections between two layers) between the layers at specified topology positions, and prepare it to be filled with connections between them. No actual connections between any layer nodes are created. Returns TRUE if successful. Parameters are:

- source_pos: integer, position in topology of source layer.
- destin_pos: integer, position in topology of destination layer.
- name: string, containing name (that also specifies type) of new connection set (see above).
- optional_parameter: (optional) double, parameter to be used by specific connection set implementations (for example, 'BP' connection sets interpret it is as the learning rate to be used when adjusting weights, 'LVQ' connection sets use it to count iterations for decreasing weight adjustments, etc). Note: for more optional parameters use parameters below.
- parameters: list, containing named parameters to be used in creating the connection set. Must include an element named name which contains the name (that also specifies type) of new empty connection set (similar to the corresponding standalone parameter described above).

fully_connect_layers_at(source_pos, destin_pos, name, min_random_weight, max_random_weight, optional_

fully_connect_layers_at(source_pos, destin_pos, parameters, min_random_weight, max_random_weight): Same as connect_layers_at but also fills the new connection_set with connections between the nodes of the two layers, fully connecting the layers (n x m connections are created, with n and m the number of nodes at each layer respectively). Returns TRUE if successful. Parameters are:

- source_pos: integer, position in topology of source layer.
- destin_pos: integer, position in topology of destination layer.
- name: string, containing name (that also specifies type) of new connection set (see above).
- min_random_weight: double, minimum value for random initial connection weights.
- max_random_weight: double, maximum value for random initial connection weights.
- optional_parameter: (optional) double, parameter to be used by specific connection set implementations (for example, 'BP' connection sets interpret it is as the learning rate to be used when adjusting weights, 'LVQ' connection sets use it to count iterations for decreasing weight adjustments, etc). Note: for more optional parameters use parameters below.

- parameters: list, containing named parameters to be used in creating the connection set. Must include an element named name which contains the name (that also specifies type) of new empty connection set (similar to the corresponding standalone parameter described above).

add_single_connection(pos, source_pe, destin_pe, weight): Add a connection to a connection_set that already connects two layers. Parameters are:

- pos: integer, position in topology of connection_set to which the new connection will be added.
- source_pe: integer, pe in source layer to connect.
- destin_pe: integer, pe in destination layer to connect.
- weight: double, value for initial connection weight.

remove_single_connection(pos, con): Remove a connection from a connection_set. Parameters are:

- pos: integer, position in topology of connection_set.
- con: integer, connection to remove (note: numbering starts from 0).

size(): Returns neural network size, i.e. the number of components its topology.

sizes(): Returns sizes of components in topology.

component_ids(): Returns an integer vector containing the ids of the components in topology (these ids are created at run-time and identify each NN component).

encode_at(pos): Trigger the encoding operation of the component at specified topology index (note: depending on implementation, an 'encode' operation usually collects inputs, processes the data, adjusts internal state variables and/or weights, and possibly produces output). Returns TRUE if successful. Parameters are:

- pos: integer, position (in NN's topology) of component to perform encoding.

encode_all(fwd): Trigger the encoding operation of all the components in the NN topology. Returns TRUE if successful. Parameters are:

- fwd: logical, set to TRUE to trigger encoding forwards (first-to-last component), FALSE to trigger encoding backwards (last-to-first component).

encode_all_fwd( ): Trigger the encoding operation of all the components in the NN topology following a forward (top-to-bottom) direction. Returns TRUE if successful.

encode_all_bwd( ): Trigger the encoding operation of all the components in the NN topology following a backward (bottom-to-top) direction. Returns TRUE if successful.

encode_dataset_unsupervised(data, pos, epochs, fwd): Encode a dataset using unsupervised training. A faster method to encode a data set. All the components in the NN topology will perform 'encode' in specified direction. Returns TRUE if successful. Parameters are:

- data: numeric matrix, containing input vectors as rows.
- pos: integer, position (in NN's topology) of component to receive input vectors.
- epochs: integer, number of training epochs (encoding repetitions of the entire dataset).
- fwd: logical, indicates direction, TRUE to trigger encoding forwards (first-to-last component), FALSE to encode backwards (last-to-first component).

encode_datasets_supervised(i_data, i_pos, j_data, j_pos, j_destination_register, epochs, fwd): Encode multiple (i,j) vector pairs stored in two corresponding data sets, using supervised training. A faster method to encode the data. All the components in the NN topology will perform 'encode' in specified direction. Returns TRUE if successful. Parameters are:

- i_data: numeric matrix, data set, each row is a vector i of vector-pair (i,j).
- i_pos: integer, position (in NN's topology) of component to receive i vectors.
- j_data: numeric matrix, data set, each row is a corresponding vector j of vector-pair (i,j).
- j_pos: integer, position (in NN's topology) of component to receive j vectors.
- j_destination_selector: integer, selects which internal node (pe) registers will receive vector j, i.e. if 0 internal node register 'input' will be used (j will become the layer's input), if 1 register 'output' will be used (j will become the layer's output), if 2 register 'misc' will be used (implementations may use this as an alternative way to transfer data to nodes without altering current input or output).
- epochs: integer, number of training epochs (encoding repetitions of the entire data).
- fwd: logical, indicates direction, TRUE to trigger encoding forwards (first-to-last component), FALSE to encode backwards (last-to-first component).

recall_at(pos): Trigger the recall (mapping, data retrieval) operation of the component at specified topology index (note: depending on implementation, a 'recall' operation usually collects input(s), processes the data, produces output and resets input to 0). Returns TRUE if successful. Parameters are:

- pos: integer, position (in NN's topology) of component to perform recall.

recall_all(fwd): Trigger the recall (mapping, data retrieval) operation of all the components in the NN topology. Returns TRUE if successful. Parameters are:

- fwd: logical, set to TRUE to trigger recall forwards (first-to-last component), FALSE to trigger recall backwards (last-to-first component).

recall_all_fwd( ): Trigger the recall (mapping, data retrieval) operation of all the components in the NN topology following a forward (top-to-bottom) direction. Returns TRUE if successful.

recall_all_bwd( ): Trigger the recall (mapping, data retrieval) operation of all the components in the NN topology following a backward (bottom-to-top) direction. Returns TRUE if successful.

recall_dataset(data_in, input_pos, output_pos, fwd): Recall (map, retrieve output for) a dataset. A faster method to recall an entire data set. All the components in the NN topology will perform 'recall' in specified direction. Returns numeric matrix containing corresponding output. Parameters are:

- data_in: numeric matrix, containing input vectors as rows.
- input_pos: integer, position (in NN's topology) of component to receive input vectors.
- output_pos: integer, position (in NN's topology) of component to produce output.
- fwd: logical, indicates direction, TRUE to trigger 'recall' (mapping) forwards (first-to-last component), FALSE to recall backwards (last-to-first component).

input_at(pos, data_in): Input a data vector to the component (layer) at specified topology index. Returns TRUE if successful. Parameters are:

- pos: integer, position (in NN's topology) of component to receive input.
- data_in: NumericVector, data to be sent as input to component (sizes must match).

set_input_at(pos, data_in): Same as input_at (see above)

get_input_at(pos): Get the current input for the component at specified topology index. Currently applicable to connection_set (returning for each connection the output of corresponding source PE), or layer (returning a preview of current PE inputs; note: many PE implementations clear their inputs once they have processed them and produced the corresponding output). If successful, returns NumericVector, otherwise vector of zero length. Parameters are:

- pos: integer, position (in NN's topology) of component to use.

`get_output_from( pos ):` Get the current output of the component at specified topology index. If successful, returns NumericVector of output values (otherwise vector of zero length). Parameters are:

- pos: integer, position (in NN's topology) of component to use.

`get_output_at( pos ):` Same as `get_output_from`, see above.

`set_output_at( pos, data_in ):` Set the values in the `output` data register that pe objects maintain, for `layer` at specified topology index (currently only `layer` components are supported). If successful, returns TRUE. Parameters are:

- pos: integer, position (in NN's topology) of component to use.
- data_in: NumericVector, data to be used for new values in `misc` registers (sizes must match).

`get_weights_at( pos ):` Get the current weights of the component (`connection_set`) at specified topology index. If successful, returns NumericVector of connection weights (otherwise vector of zero length). Parameters are:

- pos: integer, position (in NN's topology) of component to use.

`set_weights_at( pos ):` Set the weights of the component (`connection_set`) at specified topology index. If successful, returns TRUE. Parameters are:

- pos: integer, position (in NN's topology) of component to use.
- data_in: NumericVector, data to be used for new values in `weight` registers of connections (sizes must match).

`get_weight_at( pos, connection ):` Get the current weight of a connection in component (`connection_set`) at specified topology index. If successful, returns weight, otherwise 0 (note: this might change in future versions). Parameters are:

- pos: integer, position (in NN's topology) of component to use.
- connection: integer, connection to use (note: numbering starts from 0).

`set_weight_at( pos, connection, value ):` Set the weight of a connection in component (`connection_set`) at specified topology index. If successful, returns TRUE. Parameters are:

- pos: integer, position (in NN's topology) of component to use.
- connection: integer, connection to use (note: numbering starts from 0).
- value: new weight for connection.

`get_misc_values_at( pos ):` Get the values in the `misc` data register that pe and `connection` objects maintain, for objects at specified topology index. If successful, returns NumericVector of the values (otherwise vector of zero length). Parameters are:

- pos: integer, position (in NN's topology) of component to use.

`set_misc_values_at( pos, data_in ):` Set the values in the `misc` data register that pe and `connection` objects maintain, for objects at specified topology index. If successful, returns TRUE. Parameters are:

- pos: integer, position (in NN's topology) of component to use.
- data_in: NumericVector, data to be used for new values in `misc` registers (sizes must match).

`get_biases_at( pos ):` Get the values in the `bias` register that `pe` (Processing Elements or nodes) maintain, for `layer` at specified topology index (only `layer` components are supported). If successful, returns NumericVector of bias values (otherwise vector of zero length). Parameters are:

- `pos`: integer, position (in NN's topology) of component to use.

`set_biases_at( pos, data_in ):` Set the values in the `bias` register that `pe` (Processing Elements or nodes) maintain, for `layer` at specified topology index (only `layer` components are supported). If successful, returns TRUE. Parameters are:

- `pos`: integer, position (in NN's topology) of component to use.
- `data_in`: NumericVector, data to be used for new values in `bias` registers (sizes must match).

`get_bias_at( pos, pe ):` Get the bias of a `pe` (Processing Element or node) in component (`layer`) at specified topology index. If successful, returns bias otherwise 0 (note: this might change in future versions). Parameters are:

- `pos`: integer, position (in NN's topology) of component to use.
- `pe`: integer, Processing Element (node) in `layer` to use (note: numbering starts from 0).

`set_bias_at( pos, pe, value ):` Set the bias of a `pe` (Processing Element or node) in component (`layer`) at specified topology index. If successful, returns TRUE. Parameters are:

- `pos`: integer, position (in NN's topology) of component to use.
- `pe`: integer, Processing Element (node) in `layer` to use (note: numbering starts from 0).
- `value`: new value for bias at the specified `pe`.

`add_R_forwarding( trigger, FUN ):` Adds a control component which will invoke an R function. The R function will receive (as its first argument) a vector of values containing the output of the previous component in the topology. The object returned by the function will be fed as input to the next (in forward direction) component in the topology. Notes: (a) once the R function is invoked, its result will be maintained as this component's output; (b) the component will fail to perform processing if the R function's result cannot be converted to a numeric vector. If successful, returns TRUE. Parameters are:

- `trigger`: string, specifies when to invoke the R function. Valid options are `"on encode"`, `"on recall"`, `"always"` or `"never"`.
- `FUN`: string, the R function to be invoked. If `""`, no R function is invoked and data is transferred unmodified.

`add_R_pipelining( trigger, FUN, fwd ):` Adds a control component which will invoke an R function. The R function will process (as its first argument) a vector of values which are the output of a neighboring component in the topology; The result of invoking the function will be fed as input to the other neighboring component in the topology. The components are selected according to the value of parameter `fwd` (see below). Notes: (a) once the R function is invoked, its result will be maintained as this component's output; (b) the component will fail to perform processing if the R function's result cannot be converted to a numeric vector. If successful, returns TRUE. Parameters are:

- `trigger`: string, specifies when to invoke the R function. Valid options are `"on encode"`, `"on recall"`, `"always"` or `"never"`.
- `FUN`: string, the R function to be invoked. If `""`, no R function is invoked and data is transferred unmodified.

- fwd: logical, set to TRUE if encoding or recalling in forward, top-to-bottom, direction and need to read from previous component in the topology feeding the result as input to the next (same as add_R_forwarding). If FALSE, reads from next component in the topology and feeds the result as input to the previous (useful when encoding/recalling in backward, bottom-to-top, direction).

add_R_ignoring(trigger, FUN, i_mode, input_from): Adds a control component which will invoke an R function ignoring its result. The R function will process (as its first argument) a vector of values taken from a specified component in the topology, but the function's result will be ignored. This is suitable for invoking functions such as print, plot etc. Note: the component maintains the original values as its output values but does not send to any other component neither these original values nor the result of the R function. If successful, returns TRUE. Parameters are:

- trigger: string, specifies when to invoke the R function. Valid options are "on encode", "on recall", "always" or "never".
- FUN: string, the R function to be invoked. If "", no R function is invoked and data is transferred unmodified.
- i_mode: string, specifies the source of data to be retrieved and processed by the R function. Valid options are "none", "input of", "output of", "weights at", "biases at" and "misc at".
- input_from: integer, position (in NN's topology) of component to retrieve data from.

add_R_function(trigger, FUN, i_mode, input_from, o_mode, output_to, ignore_result): Adds a control component which will invoke an R function. The R function will process process (as its first argument) a vector of values taken from a specified component and feed the results to another component. Notes: (a) once the R function is invoked, its result will be maintained as this component's output (unless ignore_result is set to TRUE, in which case the original values will be maintained); (b) the component will fail to perform processing if the R function's result cannot be converted to a numeric vector and ignore_result is FALSE. If successful, returns TRUE. Parameters are:

- trigger: string, specifies when to invoke the R function. Valid options are "on encode", "on recall", "always" or "never".
- FUN: string, the R function to be invoked. If "", no R function is invoked and data is transferred unmodified.
- i_mode: string, specifies the source of data to be retrieved and processed by the R function. Valid options are "none", "input of", "output of", "weights at", "biases at" and "misc at".
- input_from: integer, position (in NN's topology) of component to retrieve data from.
- o_mode: string, specifies the destination for the result returned by the R function. Valid options are "none", "to input", "to output", "to weights", "to biases" and "to misc".
- output_to: integer, position (in NN's topology) of component to receive the resulting data.
- ignore_result: logical, if TRUE, the R function's results are ignored and original (incoming) values are maintained and (possibly) sent to the output_to component. If FALSE, the values used are those returned by the R function.

outline(): Print a summary description of all components in topology.

print(): Print internal NN state, including all components in topology.

`show()`: Print summary description and internal NN state.

`get_topology_info()`: Returns `data.frame` with topology information.

The following methods are inherited (from the corresponding class): objectPointer ("RcppClass"), initialize ("RcppClass"), show ("RcppClass").

Note

This R module maintains a generic neural network that can be manipulated using the provided methods. In addition to predefined components already existing in the package, new neural network components can be defined and then employed by the `"NN"` module. In doing so, it is recommended to use the provided C++ base classes and class-templates. This requires the package source code (which includes the **nnlib2** C++ library of neural network base classes) and the ability to compile the package. The steps for defining new types of components using C++ are outlined below:

- Any new component type or class definition should be added to the header file called `"additional_parts.h"` which is included in the package source (src) directory, or in files accessible by the functions in `"additional_parts.h"`. Therefore, all new components to be employed by the `NN` module must be defined in `"additional_parts.h"` or in files that this file includes via `#include`.
- New `layer`, `connection_set`, `pe` or `connection` definitions must comply (at least loosely) to the **nnlib2** base class hierarchy and structure and follow the related guidelines. Note: some minimal examples of class and type definitions can be found in the `"additional_parts.h"` file itself.
- A textual name must be assigned to any new `layer` or `connection_set`, to be used as parameter in `NN` module methods that require a name to create a component. This can be as simple as a single line of code where given the textual name the corresponding component object is created and returned. This code must be added (as appropriate) to either `generate_custom_layer()` or `generate_custom_connection_set()` functions found in the same `"additional_parts.h"` header file. Note: example entries can be found in these functions at the `"additional_parts.h"` file. Some of the available names are listed in NN_component_names.

Alternatively, NN components can also be defined using only R code (see NN_R_components). More information on expanding the library with new types of NN components (nodes, layers, connections etc) and models, can be found in the package's vignette as well as the related repository on Github). Please consider submitting any useful components you create, to enrich future versions of the package.

Author(s)

Vasilis N. Nikolaidis <vnnikolaidis@gmail.com>

See Also

BP, LVQs, MAM, NN_component_names, NN_R_components.

Examples

```
# Example 1:
```

```
# (1.A) create new 'NN' object:

n <- new("NN")

# (1.B) Add topology components:

# 1. add a layer of 4 generic nodes:
n$add_layer("generic",4)
# 2. add a set for connections that pass data unmodified:
n$add_connection_set("pass-through")
# 3. add another layer of 2 generic nodes:
n$add_layer("generic",2)
# 4. add a set for connections that pass data x weight:
n$add_connection_set("wpass-through")
# 5. add a layer of 1 generic node:
n$add_layer("generic",1)
# Create actual full connections in sets, random initial weights in [0,1]:
n$create_connections_in_sets(0,1)
# Optionaly, show an outline of the topology:
n$outline()

# (1.C) use the network.

# input some data, and create output for it:
n$input_at(1,c(10,20,30,40))
n$recall_all(TRUE)
# the final output:
n$get_output_from(5)

# (1.D) optionally, examine the network:

# the input for set of connections at position 2:
n$get_input_at(2)
# Data is passed unmodified through connections at position 2,
# and (by default) summed together at each node of layer at position 3.
# Final output from layer in position 3:
n$get_output_from(3)
# Data is then passed multiplied by the random weights through
# connections at position 4. The weights of these connections:
n$get_weights_at(4)
# Data is finally summed together at the node of layer at position 5,
# producing the final output, which (again) is:
n$get_output_from(5)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Example 2: A simple MAM NN

# (2.A) Preparation:

# Create data pairs

iris_data    <- as.matrix( scale( iris[1:4] ) )
iris_species <- matrix(data=-1, nrow=nrow(iris_data), ncol=3)
```

```
for(r in 1:nrow(iris_data))
 iris_species[r ,as.integer( iris$Species )[r]]=1

# Create the NN and its components:

m <- new( "NN" )
m$add_layer( "generic" , 4 )
m$add_layer( "generic" , 3 )
m$fully_connect_layers_at(1, 2, "MAM", 0, 0)

# (2.B) Use the NN to store iris (data,species) pair:

# encode pairs in NN:

m$encode_datasets_supervised(
iris_data,1,
iris_species,3,0,
1,TRUE)

# (2.C) Recall iris species from NN:

recalled_data <- m$recall_dataset(iris_data,1,3,TRUE)

# (2.D) Convert recalled data to ids and plot results:

recalled_ids <- apply(recalled_data, 1, which.max)
plot(iris_data, pch=recalled_ids)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Example 3: Using add_R_... methods in a NN:

# (3.A) add_R_ignoring, for functions whose result will be ignored by the NN:

a<-new("NN")
a$add_layer("pass-through",4)
a$add_R_ignoring("on recall","print","output of",1)
a$add_connection_set("pass-through")
a$add_R_ignoring("on recall","print","input of",3)
a$add_layer("pass-through",2)
a$add_R_ignoring("on recall","print","output of",5)
a$create_connections_in_sets(0,0)

# below a fwd recall. During it, the NN will print the output
# of layer @1, then print the input of connections @3, and
# finally print the output of layer @5:

a$set_input_at(1,1:4)
a$recall_all(TRUE)

# (3.B) add_R_forwarding is used to read output of component above,
# apply an R function and send result as input to component below.
# (Due to current limitations of various component types, place the
# add_R_forwarding between two layers and connect other components
```

```
# two those layers)

a<-new("NN")
a$add_layer("pass-through",4)
a$add_R_forwarding("on recall","sin")
a$add_layer("pass-through",4)

# during a fwd recall, the R component @2 will get the output
# of layer @1, apply an R function (here function sin) and send
# the result as input to layer @3.

a$set_input_at(1,1:4)
a$recall_all(TRUE)
a$get_output_from(3)

# (3.C) add_R_pipelining is similar to add_R_forwarding but allows reading
# the output of component below, and feed result to component above
# (for encode/recalls in backwards direction)

a<-new("NN")
a$add_layer("pass-through",4)
a$add_R_pipelining("on recall","sin",FALSE)
a$add_layer("pass-through",4)

# below is a recall backwards, the R component @2 will get the output
# of layer @3, apply R function and send the its as input to layer @1.

a$set_input_at(3,1:4)
a$recall_all(FALSE)
a$get_output_from(1)

# (3.D) add_R_function allows us to define the destination for the function's
# results. This may include destinations such as PE biases, connection
# weights etc.

a<-new("NN")
a$add_layer("pass-through",4)
a$add_R_function("on recall","sum","output of",1,"to input",3, FALSE)
a$add_layer("pass-through",1)

# below, in a typical forward recall, the R component @2 will get the output
# of layer @1, apply an R function (here function sum) and send it as
# input of layer @3.

a$set_input_at(1,1:4)
a$recall_all(TRUE)
a$get_output_from(3)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Example 4: A more complete example where a NN similar to that of help(LVQs)
# is implemented via 'NN'. It is a (supervised) LVQ. This version
# also allows using multiple output nodes per class.
# Note: while this is similar to LVQs, learning rate is NOT affected by epoch.
```

```
# Obviously (as goes for most NN, especially simple ones like this), one could
# easily create the model using just a matrix and some R code processing it;
# more elaborately, it could be implemented via R components (see help(NN_R_components));
# but how could one then be able to use all that fancy NN terminology? :)

# some options:

# define how many output nodes will be implicitly assigned for each class,
# i.e. groups of connections / prototype vectors / codebook vectors per class:

number_of_output_pes_per_class <- 3

# plot results?

plot_result = FALSE

# also use a mechanism to store weights (so we can plot them later)?

record_weights_at_each_iteration <- FALSE

# Next, prepare some data (based on iris).
# LVQ expects data in 0 to 1 range, so scale some numeric data...

DATA <- as.matrix(iris[1:4])
c_min <- apply(DATA, 2, FUN = "min")
c_max <- apply(DATA, 2, FUN = "max")
c_rng <- c_max - c_min
DATA <- sweep(DATA, 2, FUN = "-", c_min)
DATA <- sweep(DATA, 2, FUN = "/", c_rng)

# create a vector of desired class ids:

desired_class_ids <- as.integer(iris$Species)

# defined just to make names more general (independent from iris):

input_length      <- ncol(DATA)
number_of_classes <- length(unique(desired_class_ids))

# Next, setup the LVQ NN.
# output layer may be expanded to accommodate multiple PEs per class:

output_layer_size <-
number_of_classes * number_of_output_pes_per_class

# next, implement a supervised LVQ using NN module:

LVQ_PUNISH_PE <- 10   # as defined in the C++ LVQ code.
LVQ_DEACTI_PE <- 20   # as defined in the C++ LVQ code.
LVQ_REWARD_PE <- 30   # as defined in the C++ LVQ code.
LVQ_RND_MIN <-    0   # as defined in the C++ LVQ code.
LVQ_RND_MAX <-   +1   # as defined in the C++ LVQ code.
```

```
# create a typical LVQ topology for this problem:

n <- new('NN')
n$add_layer('pass-through', input_length)
n$add_connection_set('LVQ', 0)
n$add_layer('LVQ-output', output_layer_size)
n$create_connections_in_sets(LVQ_RND_MIN, LVQ_RND_MAX)

# optional, store current weights (so we can plot them later):

if (record_weights_at_each_iteration)
cvs <- n$get_weights_at(2)

# an ugly (nested loop) encoding code:

for (epoch in 1:5)
for (i in 1:nrow(DATA))
{
# recall a data vector:

n$input_at(1, DATA[i, ])
n$recall_all_fwd()

# find which output node is best for input vector (has smallest distance)
current_winner_pe <- which.min(n$get_output_at(3))

# translate winning node to class id:
returned_class <-
ceiling(current_winner_pe / number_of_output_pes_per_class)

# now check if the correct class was recalled (and reward)
# or an incorrect (and punish):

# in LVQ layers, the 'bias' node (PE) register is used to indicate if
# positive (reward) or negative (punishment) should be applied.

new_output_flags <- rep(LVQ_DEACTI_PE, output_layer_size)
new_output_flags[current_winner_pe] <- LVQ_PUNISH_PE
if (returned_class == desired_class_ids[i])
new_output_flags[current_winner_pe] <- LVQ_REWARD_PE
n$set_biases_at(3, new_output_flags)

# note: for this example (and unlike LVQs) learning rate is constant,
# NOT dicreasing as epochs increase.

n$encode_at(2)

# optional, store current weights (so we can plot them later):

if (record_weights_at_each_iteration)
cvs <- rbind(cvs, n$get_weights_at(2))
}
```

```
# done encoding.

# recall all data:

lvq_recalled_winning_nodes <-
apply(n$recall_dataset(DATA, 1, 3, TRUE), 1, which.min)

# translate winning node to class id:
lvq_recalled_class_ids <-
ceiling(lvq_recalled_winning_nodes / number_of_output_pes_per_class)

correct <- lvq_recalled_class_ids == desired_class_ids
cat("Correct:", sum(correct), "\n")
cat("Number of produced classes:", length(unique(lvq_recalled_class_ids)), "\n")

# plot results if requested (here only columns 1 and 2 are displayed):

if (plot_result)
{
plot(data, pch = lvq_recalled_class_ids,
 main = "LVQ recalled clusters (module)")

# optional, if weights were stored, plot them later:

if (record_weights_at_each_iteration)
{
for (cv in 0:(output_layer_size - 1))
lines(cvs[, (cv * input_length + 1):(cv * input_length + 2)],
  lwd = 2, col = cv + 1)
}
}
```

NN_component_names *Names of available NN components*

Description

A quick summary of names that can be used for adding NN components in a NN module. These names are available in the current package version. More components can be defined by the user or may be added in future versions.

Current names for layers:

Layer names currently available include:

- generic: a layer of generic Processing Elements (PEs).
- generic_d: same as above.
- pe: same as above.

- pass-through: a layer with PEs that simply pass input to output.
- which-max: a layer with PEs that return the index of one of their inputs whose value is maximum.
- MAM: a layer with PEs for Matrix-Associative-Memory NNs (see vignette).
- LVQ-input: LVQ input layer (see vignette).
- LVQ-output: LVQ output layer (see vignette).
- BP-hidden: Back-Propagation hidden layer (see vignette).
- BP-output: Back-Propagation output layer (see vignette).
- R-layer: A layer whose encode and recall (map) functionality is defined in R (see NN_R_components).

Additional (user-defined) layers currently available include:

- JustAdd10: a layer where PEs output the sum of their inputs plus 10 (created for use as example in vingnette).
- perceptron: a classic perceptron layer (created for use as example in in this post).
- MEX: a layer created for use as example in vingnette.
- example_layer_0: a layer created to be used as a simple code example for users creating custom layers.
- example_layer_1: as above.
- example_layer_2: as above.
- BP-hidden-softmax: Back-Propagation hidden layer that performs softmax on its output (when recalling).
- BP-output-softmax: Back-Propagation output layer that performs softmax on its output (when recalling).
- softmax: a layer that (during recall) sums its inputs and outputs the softmax values.
- R-connections: A set of connections whose encode and recall (map) functionality is defined in R (see NN_R_components).

Current names for sets of connections:

Names for connection sets that are currently available include:

- generic: a set of generic connections.
- pass-through: connections that pass data through with no modification.
- wpass-through: connections that pass data multiplied by weight.
- MAM: connections for Matrix-Associative-Memory NNs (see vignette).
- LVQ: connections for LVQ NNs (see vignette).
- BP: connections for Back-Propagation (see vignette).

Additional (user-defined) connection sets currently available include:

- perceptron: connections for perceptron (created for use as example in in this post).
- MEX: a connection set created for use as example in vingnette.
- example_connection_set_0: a connection set created to be used as a simple code example for users creating custom types of connection sets.
- example_connection_set_1: as above.
- example_connection_set_2: as above.

Note

These are component names that can be currently used to add components to a NN using the methods provided by NN module. Such methods include `add_layer`, `add_connection_set`, `connect_layers_at`, `fully_connect_layers_at` etc. Some of these components may be experimental or created for use in examples and may change or be removed in future versions, while other components may be added.

More information on expanding the library with new, user-defined types of NN components (nodes, layers, connections etc) and models, can be found in the package's vignette as well as the related repository on Github). A quick example can also be found in this post. Please consider submitting any useful components you create, to enrich future versions of the package.

Author(s)

Vasilis N. Nikolaidis <vnnikolaidis@gmail.com>

See Also

NN, NN_R_components.

`NN_R_components` *Custom NN components defined using R*

Description

Custom NN components (to be employed in NN module neural networks) usually have their functionality defined using corresponding **nnlib2** C++ classes. Alternatively, custom NN components can be defined using only R code.

Introduction

In addition to NN components defined using the provided **nnlib2** C++ classes and class-templates (see Notes in NN), custom, user-defined NN components also can be created in R code (without any need for C++). Regardless of how they are defined, such components can be added to neural networks created in R via NN module and cooperate with each other.

1. Layers

Layers of nodes (aka Processing Elements or PEs) whose encode/recall behavior is to be defined using R can be added to the NN via the `add_layer` NN method. The call to `add_layer` should have a single parameter, a `list` containing four named elements:

- `name`: always equal to `"R-layer"`.
- `size`: the number of nodes in the new layer.
- `encode_FUN`: the name of the R function to be called when the layer is encoding data (or `""` if none).

- recall_FUN: the name of the R function to be called when the layer is recalling data (or "" if none).

For example:

```
p$add_layer(list(name="R-layer", size=100, encode_FUN="", recall_FUN="rfun"))
```

adds a layer to a NN topology (here the NN is named p). The new layer will contain 100 nodes, no R function will be used when the layer is encoding, some R function (here named rfun) will be used when the layer is recalling (mapping) data.

The R functions specified as encode_FUN and recall_FUN for layers will have to be defined so that they accept (zero or more) of the following parameters:

- INPUT: vector of the current incoming numeric values (one per node, length equals size of the layer).
- INPUT_Q: matrix where each column contains the numeric values that have been sent to the corresponding node.
- BIAS: vector with the numeric value stored as 'bias' of each node (length equals size of the layer).
- MISC: vector with the numeric value stored in each node's 'misc' register (length equals size of the layer).
- OUTPUT: vector with the current output of the layer (length equals size of the layer).

In particular, for layers the R functions should have the following characteristics:

- Encode function: the R function to be called when the layer is encoding data may use any of the parameters listed above and must return a list containing named items with the new (adjusted) values for BIAS, MISC and / or OUTPUT. If no changes are made by the R function, it may return an empty list.
- Recall function: the R function to be called when the layer is recalling (mapping) data may use any of the parameters listed above and must return a vector containing the layer's new OUTPUT.

Note: The two variations of input (INPUT and INPUT_Q) are provided for flexibility in various implementations. Some connection set implementations may only send a single (final) input value to each node. These values are found in INPUT. Other connection set types may send the individual values from each individual connection, so that they can be processed by the node's input_function; these values will be found in INPUT_Q. Furthermore, there may be designs where a combination of the two is used (not recommended), or several different connection sets are connected and sending data to the same destination layer, etc. Also note that direct access to INPUT may be removed is future versions.

2. Set of connections

Connection sets whose encode/recall behavior is to be defined using R can be added to the NN via the add_connection_set or fully_connect_layers_at NN methods. The call to add_connection_set should have a single parameter, a list containing named elements:

- name: always equal to "R-connections".

- encode_FUN: the name of the R function to be called when the connection set is encoding data (or "" if none).
- recall_FUN: the name of the R function to be called when the connection set is recalling data (or "" if none).
- requires_misc: (optional) logical, if TRUE each connection will be provided with an extra 'misc' data register.

For example:

```
p$add_connection_set(list(name="R-connections",encode_FUN="ef",recall_FUN="rf"))
```

adds a set of connections to the NN topology (here the NN is named p). The new connection set will use some R function (here named ef) when encoding data and another R function (here named rf) when recalling (mapping) data.

Note that for sets of connections defined using R (as described here), each set maintains the connection weights and (if required) misc values in matrices. During encode or recall (map) operations, the connection weights matrix, misc values matrix (if any) and other data from the connected layers are sent for processing to the two R functions. No **nnlib2** C++ classes (connection_set and connection) are employed in this process, and all processing is done in R. The R functions specified as encode_FUN and recall_FUN for connection sets will have to be defined so that they accept (zero or more) of the following parameters:

- WEIGHTS: numeric matrix (s rows, d columns). This matrix contains the current connection weights..
- SOURCE_INPUT: numeric vector (length s) containing the current input values of the nodes in the source layer (note: nodes often reset this values after they have processed them).
- SOURCE_OUTPUT: numeric vector (length s) containing the current output values of the nodes in the source layer.
- SOURCE_MISC: numeric vector (length s) with the numeric value stored in 'misc' registers of each node in the source layer.
- DESTINATION_INPUT: numeric vector (length d) containing the current input values of the nodes in the destination layer (note: nodes often reset this values after they have processed them).
- DESTINATION_OUTPUT: numeric vector (length d) containing the current output values of the nodes in the destination layer.
- DESTINATION_MISC: numeric vector (length d) with the numeric value stored in 'misc' registers of each node in the destination layer.
- MISC: numeric matrix (s rows, d columns). If not used, this is a matrix of 0 rows and 0 columns, otherwise it contains the values of the 'misc' register in each connection.

where s is the number of nodes (length) of the source layer and d the number nodes in the destination layer.

The R functions for connection sets should have the following characteristics:

- Encode function: the R function to be called when the connection set is encoding data may use any of the parameters listed above, but must return a `list` containing the new (adjusted) connection weights (named `WEIGHTS`, numeric matrix of s rows, d columns) and possibly the new connection 'misc' values (named `MISC`, numeric matrix of s rows, d columns). If no changes were made by the R function, it may return an empty list.
- Recall function: the R function to be called when the connection set is recalling or mapping data may use any of the parameters listed above and must return a numeric matrix of d columns. Each column of this matrix should contain the data values to be sent to the corresponding node (PE) in the destination layer. (Note: this matrix is similar to the `INPUT_Q` used in layers, see above).

3. Control components

Other special control or processing components can be added using NN module's `add_R_forwarding`, `add_R_pipelining`, `add_R_ignoring`, and `add_R_function` methods. See NN module.

Note

Defining NN components with custom behavior in R does have a cost in terms of run-time performance. It also, to a certain degree, defies some of the reasons for using C++ classes. However, it may be useful for experimentation, prototyping, education purposes etc.

Author(s)

Vasilis N. Nikolaidis <vnnikolaidis@gmail.com>

See Also

NN, NN_component_names.

Examples

```
## Not run:
#-------------------------------------------------------------------------------
# 1. LAYER EXAMPLE:

# Example R function to be used when the layer is encoding:
# Version for when the final input (a single value per PE) is directly sent to
# the layer (by set_input or some connection set).
# Outputs difference from current bias values, stores current input as new bias:

LAYERenc1 <- function(INPUT,BIAS,...)
{
i <- INPUT # get values directly injected as input to the PE.
o <- i-BIAS # subtract old bias from input.
# update layer's output and biases:
return(list(OUTPUT=o, BIAS=INPUT))
}

# Example R function to be used when the layer is recalling (mapping):
# Version for when the final input (a single value per PE) is directly sent to
```

```
# the layer (by set_input or some connection set).
# Outputs difference from current bias values:

LAYERrec1 <- function(INPUT,BIAS,...)
{
i <- INPUT # get values directly injected as input to the PE.
o <- i-BIAS # subtract old bias from input.
return(o) # return this as output.
}

# Example R function to be used when the layer is encoding (same as above):
# Version for cases where a connection set is designed to send multiple
# values (one for each incoming connection) to each PE in the layer so that
# the PE can process them as needed. - typically via its 'input_function'.
# (also works when set_input is used)
# INPUT_Q is a matrix where each column contains the values that have been sent
# to the corresponding node (PE).
# Outputs difference from current bias values, stores current input as new bias:

LAYERenc2 <- function(INPUT_Q,BIAS,...)
{
i <- colSums(INPUT_Q) # summate incoming values to produce final input.
o <- i-BIAS # subtract old bias from that input.
# update layer's output and biases:
return(list(OUTPUT=o, BIAS=i))
}

# Example R function to be used when the layer is recalling/mapping (same as above):
# version for cases where a connection set is designed to send multiple
# values (one for each incoming connection) to each PE in the layer so that
# the PE can process them as needed - typically via its 'input_function'.
# (also works when set_input is used)
# INPUT_Q is a matrix where each column contains the values that have been sent
# to the corresponding node (PE).
# Outputs difference from current bias values:

LAYERrec2 <- function(INPUT_Q,BIAS,...)
{
i <- colSums(INPUT_Q) # summate incoming values to produce final input.
o <- i-BIAS # subtract old bias from that input.
return(o) # return this as output.
}

# create and setup a "NN".

n<-new("NN")
n$add_layer(list(name="R-layer", size=4,
 encode_FUN="LAYERenc1", recall_FUN="LAYERrec1"))

# test the layer:

n$set_input_at(1,c(1,0,5,5))
n$encode_at(1)
```

```
print(n$get_biases_at(1))

n$set_input_at(1,c(20,20,20,20))
n$recall_at(1)
print(n$get_output_at(1))
n$set_input_at(1,c(10,0,10,0))
n$recall_at(1)
print(n$get_output_at(1))

#-------------------------------------------------------------------------------
# 2. CONNECTION SET EXAMPLE:

# This simple connection set will encode data by adding to each connection
# weight the output of the source node.

CSenc <- function(WEIGHTS, SOURCE_OUTPUT,...)
{
x <- WEIGHTS + SOURCE_OUTPUT
return(list(WEIGHTS=x))
}

# When recalling, this simple connection set multiplies source data by weights.
# this version sends multiple values (the products) to each destination node.
# Typical (s.a. generic) nodes add these values to process them.

CSrec1 <- function(WEIGHTS, SOURCE_OUTPUT,...)
{
x <- WEIGHTS * SOURCE_OUTPUT
return(x)
}

# When recalling, this simple connection set multiplies source data by weights.
# this version sends a single value (the sum of the products) to each
# destination node.

CSrec2 <- function(WEIGHTS, SOURCE_OUTPUT,...)
{
x <-  SOURCE_OUTPUT %*% WEIGHTS
return(x)
}

# create and setup a "NN".

n<-new("NN")
n$add_layer("generic",4)
n$add_connection_set(list(name="R-connections",encode_FUN="CSenc",recall_FUN="CSrec2"))
n$add_layer("generic",2)
n$create_connections_in_sets(0,0)

# test the NN:

n$set_input_at(1,c(0,1,5,10))
n$encode_all_fwd()
```

```
n$set_input_at(1,c(1,1,1,1))
n$encode_all_fwd()

# see if weights were modified:
print(n$get_weights_at(2))

n$set_input_at(1,c(20,20,20,20))
n$recall_all_fwd()
print(n$get_output_at(3))

#-------------------------------------------------------------------------------
# 3. A COMPLETE EXAMPLE (simple single layer perceptron-like NN):

# Function for connections, when recalling/mapping:
# Use any one of the two functions below.
# Each column of the returned matrix contains the data that will be sent to the
# corresponding destination node.

# version 1: sends multiple values (product) for destination nodes to summate.

CSmap1 <- function(WEIGHTS, SOURCE_OUTPUT,...)  WEIGHTS * SOURCE_OUTPUT

# version 2: sends corresponding value (dot product) to destination node.

CSmap2 <- function(WEIGHTS, SOURCE_OUTPUT,...) SOURCE_OUTPUT %*% WEIGHTS

#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Function for connections, when encoding data:

learning_rate <- 0.3

CSenc <- function(WEIGHTS, SOURCE_OUTPUT, DESTINATION_MISC, DESTINATION_OUTPUT, ...)
{
  a <- learning_rate *
         (DESTINATION_MISC - DESTINATION_OUTPUT)   # desired output is in misc registers.
  a <- outer( SOURCE_OUTPUT, a , "*" )              # compute weight adjustments.
  w <- WEIGHTS + a                                  # compute adjusted weights.
  return(list(WEIGHTS=w))                           # return new (adjusted) weights.
}

#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Function for layer, when recalling/mapping:
# (note: no encode function is used for the layer in this example)

LAmap <- function(INPUT_Q,...)
{
x <- colSums(INPUT_Q) # input function is summation.
x <- ifelse(x>0,1,0) # threshold function is step.
return(x)
}

#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# prepare some data based on iris data set:
```

```
data_in <- as.matrix(iris[1:4])
iris_cases <- nrow((data_in))
# make a "one-hot" encoding matrix for iris species
desired_data_out <- matrix(data=0, nrow=iris_cases, ncol=3)
desired_data_out[cbind(1:iris_cases,unclass(iris[,5]))]=1

# create the NN and define its components:
# (first generic layer simply accepts input and transfers it to the connections)

p <- new("NN")

p$add_layer("generic",4)

p$add_connection_set(list(name="R-connections",
                          encode_FUN="CSenc",
                          recall_FUN="CSmap2"))

p$add_layer(list(name="R-layer",
                 size=3,
                 encode_FUN="",
                 recall_FUN="LAmap"))

p$create_connections_in_sets(0,0)

# encode data and desired output (for 50 training epochs):

for(i in 1:50)
for(c in 1:iris_cases)
{
p$input_at(1,data_in[c,])
p$set_misc_values_at(3,desired_data_out[c,])  # put desired output in misc registers
p$recall_all_fwd();
p$encode_at(2)
}

# Recall the data and show NN's output:

for(c in 1:iris_cases)
{
p$input_at(1,data_in[c,])
p$recall_all_fwd()
cat("iris case ",c,", desired = ", desired_data_out[c,],
" returned = ", p$get_output_from(3),"\n")
}


## End(Not run)
```

Index