Neural Networks in **R** Using the Stuttgart Neural Network Simulator: RSNNS

Christoph Bergmeir
University of Granada

José M. Benítez
University of Granada

Abstract

Neural networks are important standard machine learning procedures for classification and regression. We describe the R package **RSNNS** that provides a convenient interface to the popular Stuttgart Neural Network Simulator **SNNS**. The main features are (a) encapsulation of the relevant **SNNS** parts in a C++ class, for sequential and parallel usage of different networks, (b) accessibility of all of the **SNNS** algorithmic functionality from R using a low-level interface, and (c) a high-level interface for convenient, R-style usage of many standard neural network procedures. The package also includes functions for visualization and analysis of the models and the training procedures, as well as functions for data input/output from/to the original **SNNS** file formats.

Keywords: neural networks, **SNNS**, R, **RSNNS**.

1. Introduction

This paper presents the package **RSNNS** (Bergmeir and Benítez 2012) that implements an R (R Development Core Team 2011) interface to the Stuttgart Neural Network Simulator (**SNNS**, Zell *et al.* 1998). The **SNNS** is a comprehensive application for neural network model building, training, and testing. Today it is still one of the most complete, most reliable, and fastest implementations of neural network standard procedures. The main advantages of **RSNNS**, rendering it a general purpose comprehensive neural network package for R, are threefold. (a) The functionality and flexibility of the **SNNS** kernel is provided within R. (b) Convenient interfaces to the most common tasks are provided, so that the methods of **SNNS** integrate seamlessly into R, especially with respect to the scripting, automation, and parallelization capabilities of R. Finally, (c) enhanced tools for visualization and analysis of training and testing of the networks are provided.

The **RSNNS** is available from the Comprehensive R Archive Network (CRAN) at http:

//CRAN.R-project.org/package=RSNNS. Moreover, a web page including fully detailed examples of usage at different abstraction levels and further information is available at `http://sci2s.ugr.es/dicits/software/RSNNS`.

The remainder of this paper is structured as follows. Section 2 describes the main features of the **SNNS** software, together with references to general introductions to neural networks and to original publications. Section 3 presents the general software architecture and implementation details of the package. Section 4 presents the high-level interface of the package. Section 5 gives an overview of the included example datasets, and Section 6 shows some examples for the usage of the package. Section 7 compares the presented package with packages and solutions yet available in R, and discusses strengths and limitations. Finally, Section 8 concludes the paper.

2. Features of the original SNNS software

SNNS consists of three main components: a simulation kernel, a graphical user interface (GUI), and a set of command line tools. It was written for Unix in C and the GUI uses X11. Windows ports exist. **SNNS** was developed at University of Stuttgart and is now maintained at University of Tübingen. The last version where the authors added new functionality is version 4.2 that was released in 1998. In 2008, version 4.3 was released, which includes some patches contributed by the community (`http://developer.berlios.de/projects/snns-dev/`) that mainly add a Python wrapping. Furthermore, in this version a license change was performed from a more restrictive, academic license to the Library General Public License (LGPL).

To the best of our knowledge **SNNS** is the neural network software that supports the highest number of models. The neural network types implemented differ greatly in their manner of operation, their architecture, and their type of learning. As giving a comprehensive introduction to neural networks in general or detailed descriptions of the methods' theories is beyond the scope of this paper, we give a brief overview of the methods present in **SNNS** in the following, together with references to the original literature. Various comprehensive introductions to neural networks exist, to which the interested reader may refer, e.g., Rojas (1996) covers many of the neural networks implemented in **SNNS**. Haykin (1999) and Bishop (2003) also give comprehensive introductions, and Ripley (2007) gives a good introduction from a statistical point of view.

Network architectures implemented in **SNNS** contain multi-layer perceptrons (MLP, Rosenblatt 1958), recurrent Elman-, and Jordan networks (Elman 1990; Jordan 1986), radial basis function networks (RBF, Poggio and Girosi 1989), RBF with dynamic decay adjustment (Berthold and Diamond 1995; Hudak 1993), Hopfield networks (Hopfield 1982), time-delay neural networks (Lang *et al.* 1990), self-organizing maps (SOM), associative memories, learning vector quantization networks (LVQ, Kohonen 1988), and different types of adaptive resonance theory networks (ART, Grossberg 1988), namely ART1 (Carpenter and Grossberg 1987b), ART2 (Carpenter and Grossberg 1987a), and ARTMAP (Carpenter *et al.* 1991) nets. It implements a wide variety of initialization, learning, update, activation, and pruning functions, adding up to more than 270 in total. For example, update functions allow serial, randomized, topologically ordered, or synchronous update of the units. The adaptive resonance theory networks can be trained step-wise or directly until they reach a stable state. Learning functions include standard backpropagation, backpropagation with momentum term, back-

propagation through time (BPTT, Rumelhart *et al.* 1986), Quickprop (Fahlman 1988), resilient backpropagation (Riedmiller and Braun 1993; Riedmiller 1994), backpercolation (Jurik 1994), (recurrent) cascade-correlation (Fahlman and Lebiere 1990; Fahlman 1991), counterpropagation (Hecht-Nielsen 1987), scaled conjugate gradient (Møller 1993), and TACOMA learning (Lange *et al.* 1994). Activation functions include many activation functions common in neural networks, such as different step functions, the logistic and tanh functions, the linear function, the softmax function, etc. An example of a pruning function implemented is optimal brain damage (OBD, Cun *et al.* 1990). For a comprehensive overview of all functions along with explanations of their parameters, as well as some theoretical background, we refer to Zell *et al.* (1998), and Zell (1994).

The GUI offers tools for easy development and precise control of topologies, learning and analyzing processes. Finally, some additional functionality is scattered along a set of command line tools which for example allow to create networks, convert networks to standalone C code, or to perform some analysis tasks.

3. Package architecture and implementation details

RSNNS provides convenient access to the major part of the **SNNS** application. To achieve this, the **SNNS** kernel and some parts of the GUI code for network generation (see Section 3.1) were ported to C++ and encapsulated in one main class. All code is included in a library, which ships with **RSNNS**. We call this fork of **SNNS** (as well as its main class) SnnsCLib throughout this paper and the software. To make the SnnsCLib functionality accessible from within R, the package **Rcpp** (Eddelbuettel and François 2011) is used. The SnnsCLib class has an equivalent S4 class in R, called SnnsR: the low-level interface of the package. As SnnsR can be used to access directly the kernel API (called "krui" within **SNNS**), all of the functionality and flexibility of the **SNNS** kernel can be taken advantage of. But, as it maps C++ functions directly, its usage might seem unintuitive and laborious to R programmers. Therefore, a high-level R interface is present by the S3 class rsnns and its subclasses. This interface provides an intuitive, easy to use, and still fairly flexible interface so that the most common network topologies and learning algorithms integrate seamlessly into R. All of these components are detailed in the remainder of this section.

3.1. The SNNS fork SnnsCLib

SNNS is written as pure C code. It can only manage one neural network at a time that is represented as a global state of the program. Porting it to C++ and encapsulating it in a class bears the advantage that various instances can be used sequentially or in parallel. The code basis of SnnsCLib is the **SNNS** version 4.3 with a reverse-applied Python patch[1]. All code from the **SNNS** kernel and some code taken from the GUI to create networks of different architectures (within **SNNS** known as the "bignet" tool)[2] was copied to the src folder of the

---

[1] The patch prepares **SNNS** for Python wrapping, which is not needed in our context. Furthermore, it caused some problems during the porting to C++. So we decided to reverse-apply it, so that everything included in the patch was removed from the source.

[2] There is a tool in **SNNS** that assists in network generation, the so-called bignet tool. Though this tool is located within the source code of the GUI, there are a lot of important functions for network generation that depend strongly on kernel functionality and whose only connection to the GUI is, that they receive some input from there. These functions were also included in SnnsCLib.

package. **SNNS** has header files with a file name extension of `.h`, containing public function declarations, and others with a file name extension of `.ph`, the private headers. These were merged into one `.h` file, each. The body file extensions were renamed from `.c` to `.cpp`. As all code is to be encapsulated in one C++ class, the merged `.h` files are included (with `#include`) in the private section of the `SnnsCLib` class definition. This procedure has the advantage that in the original code relatively few things had to be changed: In the header files, the changes mainly include removal of `static` keywords and the moving of variable initializations to the constructor of `SnnsCLib`. As these variables are not anymore initialized automatically to zero, variable initializations for all variables of this type were added. In the body files, (a) **SNNS**-internal includes were substituted with an include of `SnnsCLib.h`, (b) `static` variables within functions were turned into member variables of `SnnsCLib`, (c) all function declarations were changed to declarations of member functions of `SnnsCLib`, and (d) calls to the function table are now done using the `this` pointer and C++ -style function pointers.

3.2. The **R** low-level interface class and **C++** wrapping

To use the `SnnsCLib` functionality from R we employ **Rcpp**. Every function from `SnnsCLib` has a corresponding wrapper function, e.g., the wrapper for the function to set the current learning function `setLearnFunc` is implemented as follows:

```
RcppExport SEXP SnnsCLib__setLearnFunc( SEXP xp, SEXP learning_func ) {
 Rcpp::XPtr<SnnsCLib> snnsCLib( xp );

  std::string p1 = Rcpp::as<std::string>( learning_func );

  int err = snnsCLib->krui_setLearnFunc( const_cast<char*>( p1.c_str() ) );
  return Rcpp::List::create( Rcpp::Named( "err" ) = err );
}
```

A pointer to the current `SnnsCLib` object is in all such functions present as the first parameter. The other parameters are converted and passed to the corresponding **SNNS** function. In this example, the only parameter `learning_func` gives the name of the function to be set as the current learning function. The result of the wrapper function is typically a named list. If a return value for `err` is present, then an error handling is implemented (see below).

The corresponding part of `SnnsCLib` on the R side is the S4 class `SnnsR`. Each `SnnsR` object has an external pointer to its corresponding `SnnsCLib` object. If the object factory is used, a `SnnsCLib` object is generated automatically and the pointer is initialized with this object:

```
snnsObject <- SnnsRObjectFactory()
```

`SnnsR` contains a convenient calling mechanism for member functions of its instances based on the calling mechanism suggested by **Rcpp**. The `$` operator is overloaded and can be used for function calling as illustrated by the following example. The function used earlier as an example, `SnnsCLib__setLearnFunc`, can be called like this:

```
snnsObject$setLearnFunc("Quickprop")
```

The `$` operator first searches an R function with the name `SnnsR__setLearnFunc`. If no such function is present, R's calling mechanism is used to call the corresponding C++ function from

R class	Description
`mlp`	Multi-layer perceptron
`dlvq`	Dynamic learning vector quantization network
`rbf`	Radial basis function network
`rbfDDA`	RBF network using the dynamic decay adjustment algorithm
`elman`	Recurrent Elman network
`jordan`	Recurrent Jordan network
`som`	Self-organizing map
`art1`	ART network for binary patterns
`art2`	ART network for real-valued patterns
`artmap`	ART network for supervised learning of binary patterns
`assoz`	Autoassociative memory

Table 1: Models directly accessible from the R high-level interface.

SnnsCLib. This has the advantage that a function from `SnnsCLib` can be replaced completely transparently by a function implemented in R, which typically might call the original `SnnsCLib` function after some error checking and preprocessing, or perform some postprocessing. Also, functions not present in the original **SNNS** can be added in this way, either implemented in R or in C++. A preprocessing currently implemented is automatic loading of an object serialization of the `SnnsCLib` object, if it is not present. This mechanism is used by the high-level interface objects to allow them to be saved and loaded by the usual R mechanisms for saving and loading. A postprocessing currently implemented in the `$` operator is an error handling. If in the result list the member `err` is present and not equal to zero, then the **SNNS** function `error` is called, which translates the error code to a text message. This text message then is displayed as an R warning.

3.3. R high-level interface classes

The most convenient way to use the **RSNNS** is through its high-level R interface composed of the S3 class `rsnns` and its subclasses. The classes currently implemented are shown in Table 1.

The subclasses typically implement a process of five steps:

1. Check and preprocess the input.

2. Generate an `rsnns` object using the object factory `rsnnsObjectFactory`, setting training, update, and initialization function and their parameters.

3. Create/load a network architecture by directly accessing functions of the `SnnsR` object created within the `rsnns` object: `rsnnsObject$snnsObject$...`

4. Train the network, using either the function `train.rsnns` or direct access again.

5. Postprocess the output.

The `train.rsnns` function directly saves all training results in the `rsnns` object, e.g., in `rsnnsObject$fitted.values`.

4. Using the **R** high-level interface

The high-level interface of the **RSNNS** package provides a comfortable way to develop and deploy the most common models of neural nets. The classes present in this interface enable convenient use of the models in R, as a model can be built with a single command, and many standard methods known from other models in R are present, such as `predict` to apply the models to new data, or `print` and `summary` to show characteristics of the models.

The interface is very similar for all models. As an example of use, we show the interface for the multi-layer perceptron `mlp`:

```
R> mlp(x, y, size = 5, maxit = 100, initFunc = "Randomize_Weights",
+    initFuncParams = c(-0.3, 0.3), learnFunc = "Std_Backpropagation",
+    learnFuncParams = c(0.2, 0.0), updateFunc = "Topological_Order",
+    updateFuncParams = 0.0, hiddenActFunc = "Act_Logistic",
+    shufflePatterns = TRUE, linOut = FALSE, inputsTest = NULL,
+    targetsTest = NULL)
```

The parameters reflect the five processing steps discussed in Section 3.3:

- *Training data:* `x`, `y`. `x` is a matrix or vector containing the training inputs. If the method is a supervised learning method, also training targets `y` have to be supplied.

- *The network architecture parameters:* `size`. The `mlp` has one architecture parameter, `size`, which defines the amount of neurons in the hidden layers.

- *Number of iterations:* `maxit`. The parameter `maxit` defines the number of iterations, i.e., the number of training epochs to perform.

- *Initialization function:* `initFunc`, `initFuncParams`. The initialization function initializes the components of the network, i.e., mainly the weights of the connections between the units. Depending on the network architecture and the learning method to use, an appropriate initialization function has to be chosen, e.g., there are learning procedures where weights have to be initialized with a determined value such as 0.0 or 1.0, in other procedures, random initialization may be best, or in methods where the weights represent prototypes they are usually initialized in a way to cover the whole input space. As the initialization function has a close connection to the architecture of the network, it is usually not necessary to alter the defaults that are set in the high-level functions.

- *Learning function:* `learnFunc`, `learnFuncParams`. The learning function defines how learning takes place in the network. It is a central characteristic of the network. For some network architectures, e.g., the ART networks, there is essentially only one type of learning function, which has to be used then. For other networks, such as the MLP, a host of choices exists. Depending on the function, the parameters have different meanings, e.g., for `Std_Backpropagation`, there are two parameters, the step width of the gradient descent η, and the maximal difference d_{max} between target value and output that is tolerated, i.e., that is propagated as error of value zero.

- *Update function:* `updateFunc`, `updateFuncParams`. The update function defines how activation is propagated through the network. As it is the case for the initialization

function, an update function that suits the network architecture has to be chosen, so that it usually is not necessary to alter the defaults. E.g., in feed-forward networks normally the function `Topological_Order` is used, which calculates unit activations in the topological order of the units, which means especially that activation of the units in the input layer is calculated before activations in the first hidden layer etc., so that the activation is propagated through the network from the input to the output layer. In many architectures, the same parameters that are used for the learning function can be used for the update function.

- *Special parameters for the method:* `hiddenActFunc`, `shufflePatterns`, `linOut`. Often, parameters particular to the method are present. In the `mlp`, the `hiddenActFunc` parameter specifies the activation function of the hidden units, `shufflePatterns` defines, whether patterns will be internally shuffled by **SNNS** before training or not, and `linOut` defines, whether the activation function of the output units will be the linear or the logistic function, making it suitable for classification or regression.

- *Optional test data:* `inputsTest`, `targetsTest`. Providing test data directly at the model building stage has the advantage that after each epoch the error on the test set is computed, and can later be used to analyze the training process. As the data at this stage is exclusively used to compute this iterative error, providing test data here is only interesting for supervised learning methods. In any case, the `predict` function can be used later to obtain results on test data, both in supervised and unsupervised learning scenarios.

Initialization, learning, and update function names together with their parameters are directly passed to the **SNNS** kernel. For their documentation, we refer to Zell *et al.* (1998). In the **SNNS** kernel, each of the parameter sets has a maximal length of five. A list of all available functions in the **SNNS** kernel can be obtained by `getSnnsRFunctionTable`. Their name, type, number of input- and output parameters are shown:

```
R> getSnnsRFunctionTable()[196:202, ]
```

```
                 name type #inParams #outParams
196 Counterpropagation    4         3          3
197        Dynamic_LVQ    4         5          5
198            Hebbian    4         3          3
199              JE_BP    4         3          3
200      JE_BP_Momentum    4         5          5
201       JE_Quickprop    4         5          5
202           JE_Rprop    4         4          4
```

As a short example, the easiest way to train a model is using the default parameters, only setting the training inputs, and – if required by the model – the training outputs. The generic `print` function implemented for `rsnns` objects can then be used to get information about the architecture, the functions, and the parameters used by the model:

```
R> encoder <- mlp(inputs, targets)
R> encoder
```

```
Class: mlp->rsnns
Number of inputs: 8
Number of outputs: 8
Maximal iterations: 100
Initialization function: Randomize_Weights
Initialization function parameters: -0.3 0.3
Learning function: Std_Backpropagation
Learning function parameters: 0.2 0
Update function:Topological_Order
Update function parameters: 0
Patterns are shuffled internally: TRUE
Compute error in every iteration: TRUE
Architecture Parameters:
$size
[1] 5

All members of model:
 [1] "nInputs"               "maxit"
 [3] "initFunc"              "initFuncParams"
 [5] "learnFunc"             "learnFuncParams"
 [7] "updateFunc"            "updateFuncParams"
 [9] "shufflePatterns"       "computeIterativeError"
[11] "snnsObject"            "archParams"
[13] "IterativeFitError"     "fitted.values"
[15] "nOutputs"
```

The `summary` function that is implemented in the `rsnns` class generates a textual represen-
tation of the net in the original **SNNS** file format, and displays it (output of the function is
omitted here):

```
R> summary(encoder)
```

After training, the model can be used for prediction on new data, or the visualization capabil-
ities of R can be used to analyze the training process and the models performance on training
and test data. Examples are given in Section 6.

5. Included datasets

A fairly convenient feature in R is the inclusion of datasets along with software packages.
This is important for standardized tests as well as for examples of the usage of the package
(as presented in Section 6). Various datasets are present in the original **SNNS** examples,
covering a wide variety of network types and possible applications. The functions presented
in Section 6.5 were used to convert all pattern files with fixed-size patterns included as example
data in **SNNS** to data accessible within R. All the data is available in **RSNNS** through the
list `snnsData`:

```
R> data("snnsData")
R> names(snnsData)
```

```
 [1] "art1_letters.pat"        "artmap_test.pat"
 [3] "eight_016.pat"           "laser_999.pat"
 [5] "letters_with_classes.pat" "spirals.pat"
 [7] "trainValidMAP.pat"       "xor_rec1.pat"
 [9] "art2_tetra_high.pat"     "artmap_train.pat"
[11] "eight_160.pat"           "letseq_test.pat"
[13] "nettalk.pat"             "sprach_test.pat"
[15] "validMAP.pat"            "xor_rec2.pat"
[17] "art2_tetra_low.pat"      "bdg_TDNN.pat"
[19] "encoder.pat"             "letseq_train.pat"
[21] "patmat.pat"              "sprach_train.pat"
[23] "art2_tetra_med.pat"      "font.pat"
[25] "letters_auto.pat"        "som_cube_norm.pat"
[27] "testMAP.pat"             "art2_tetra.pat"
[29] "dlvq_ziff_100.pat"       "laser_1000.pat"
[31] "letters.pat"             "som_cube.pat"
[33] "trainMAP.pat"            "xor.pat"
```

The columns of the datasets are named according to whether they are input or output to the net. The convenience functions `inputColumns` and `outputColumns` can be used to pick the right columns according to their names. Furthermore, the function `splitForTrainingAndTest` can be used to split the data in a training and a test set. So, for example the `laser` dataset can be loaded and preprocessed with:

```
R> laser <- snnsData$laser_1000.pat
R> inputs <- laser[, inputColumns(laser)]
R> targets <- laser[, outputColumns(laser)]
R> patterns <- splitForTrainingAndTest(inputs, targets, ratio = 0.15)
```

Also, use of datasets from the KEEL dataset repository (http://sci2s.ugr.es/keel/datasets.php) is possible. Reference results for various models are provided at the package homepage at http://sci2s.ugr.es/dicits/software/RSNNS/.

6. Examples

In this section we illustrate the use of the package with examples for regression, classification, and clustering. Furthermore, we comment on some other useful functionality and on the benefits of using both **RSNNS** and a separate installation of the original **SNNS** software. All examples shown here and various other ones that illustrate both the use of the high-level and the low-level interface are included in the package as demos. In R the command `demo()` gives a list of available demos, e.g., the demo `"laser"` can be started with `demo("laser")`. The examples can also be found at the package web page referred to above.

6.1. Recurrent neural networks for regression

In this example, we show the use of an Elman network (Elman 1990) for time series regression. After loading the data as seen in Section 5, the model is trained in a way similar to the one

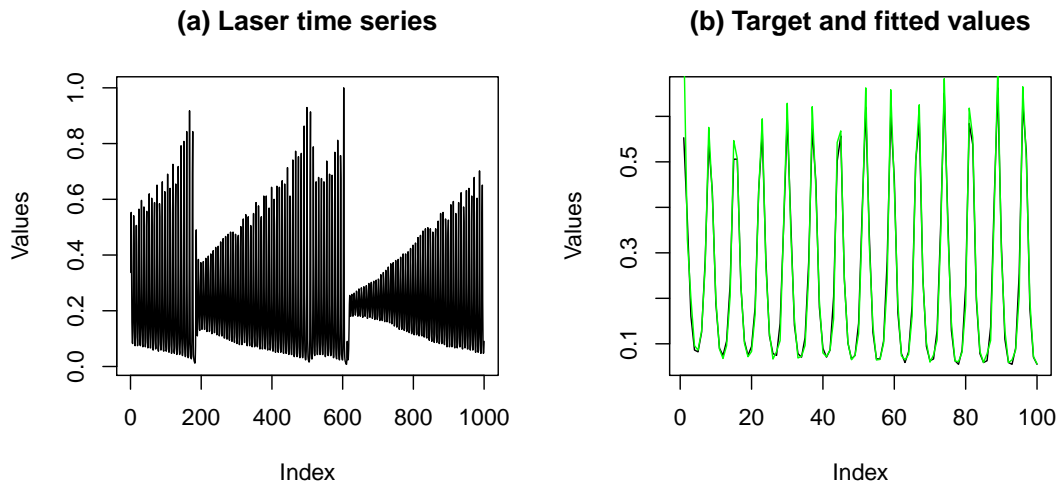**(a) Laser time series**   **(b) Target and fitted values**

Figure 1: (a) The laser example time series. (b) The first 100 values of the series (black), and the corresponding fits (green).

shown in Section 4. Meaningful default values (especially for the function to use) are already given, so it is often sufficient just to adjust the learning parameters:

```
R> model <- elman(patterns$inputsTrain, patterns$targetsTrain,
+    size = c(8, 8), learnFuncParams = c(0.1), maxit = 500,
+    inputsTest = patterns$inputsTest, targetsTest = patterns$targetsTest,
+    linOut = FALSE)
```

The powerful tools for data visualization in R come handy for neural net modeling. For example, input data and fitted values can be visualized in the following way (the plots are shown in Figure 1):

```
R> plot(inputs, type = "l")
R> plot(targets[1:100], type = "l")
R> lines(model$fitted.values[1:100], col = "green")
```

In addition to the visualization tools already available in R, various other methods for visualization and analysis are offered by the package. The function `plotIterativeError` generates an iterative error plot that shows the summed squared error (SSE), i.e., the sum of the squared errors of all patterns for every epoch. If a test set is provided, its SSE is also shown in the plot, normalized by dividing the SSE through the test set ratio (which is the amount of patterns in the test set divided by the amount of patterns in the training set). The function `plotRegressionError` can be used to generate a regression plot which illustrates the quality of the regression. It has 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, as well as a linear fit to the actual data. Using standard methods of R, also other evaluation techniques can be implemented straightforwardly, e.g., an error histogram (the plots are shown in Figure 2):
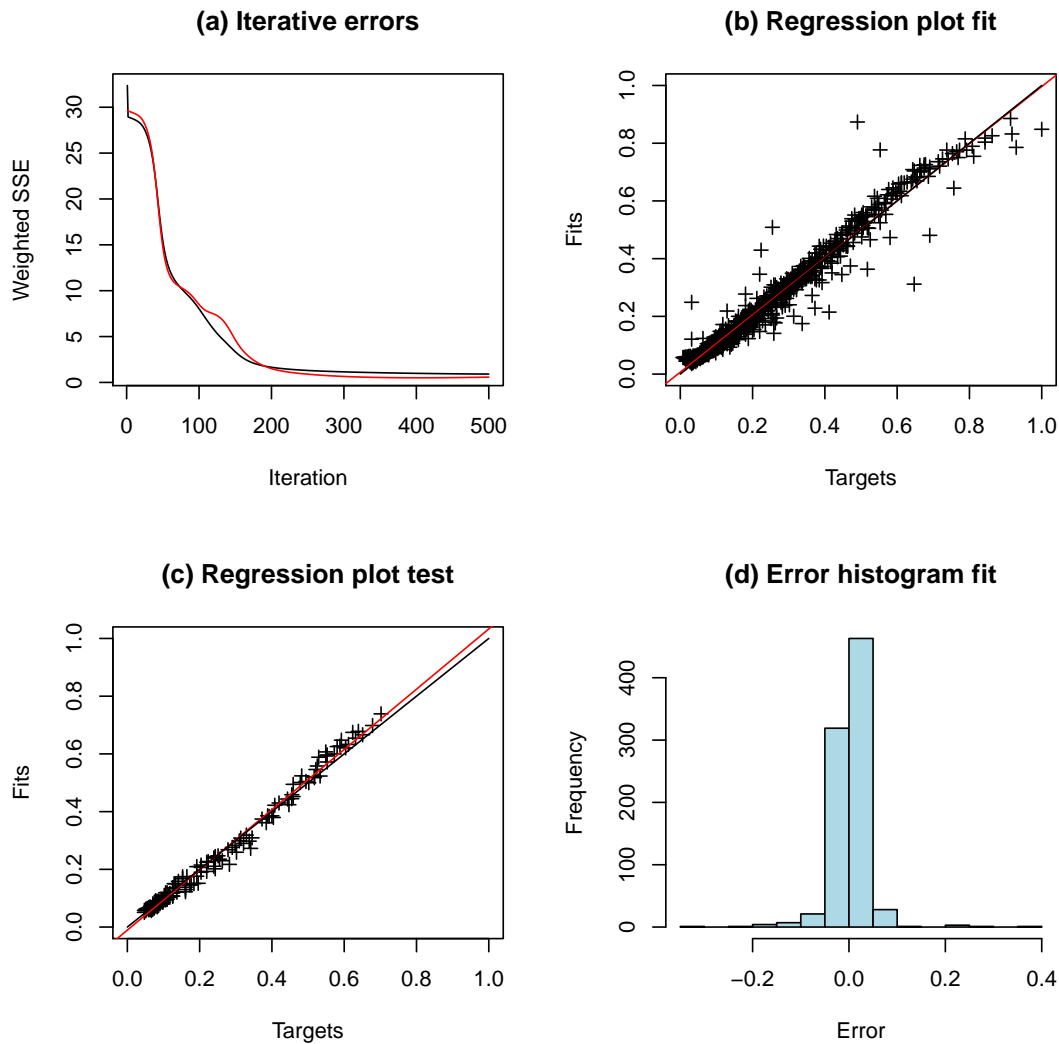
Figure 2: An Elman net trained with the laser example dataset. (a) The iterative error plot of both training (black) and test (red) error. (b) Regression plot for the training data, showing a linear fit in the optimal case (black), and to the current data (red). (c) Regression plot for the test data. (d) An error histogram of the training error.

```
R> plotIterativeError(model)
R> plotRegressionError(patterns$targetsTrain, model$fitted.values)
R> plotRegressionError(patterns$targetsTest, model$fittedTestValues)
R> hist(model$fitted.values - patterns$targetsTrain)
```

6.2. Multi-layer perceptron for classification

Performing classification or regression is very similar with the package. The neural output is typically set to the logistic function instead of the linear function, and an output neuron is used for each possible class. Training targets force the activation of the neuron representing

the correct class, i.e., using the logistic function its output should be close to one. The other neurons should output values close to zero. Methods of pre- and postprocessing to facilitate such a procedure are present in **RSNNS**. In the following, we present an example of how to train a multi-layer perceptron using the standard backpropagation algorithm (Rumelhart *et al.* 1986). We use the well-known `iris` dataset included in R for this example.

The data is loaded, shuffled, and preprocessed. The function `decodeClassLabels` generates a binary matrix from an integer-valued input vector representing class labels. With `splitForTrainingAndTest`, the data is split into training and test set, that can then be normalized using the function `normTrainingAndTestSet`, which has different normalization types implemented. We use a normalization to zero mean and variance one, which is the default setting:

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

The training data of this structure can then be used for training the multi-layer perceptron (or any other supervised learning method):

```
R> model <- mlp(iris$inputsTrain, iris$targetsTrain, size = 5,
+    learnFuncParams = 0.1, maxit = 60, inputsTest = iris$inputsTest,
+    targetsTest = iris$targetsTest)
R> predictions <- predict(model, iris$inputsTest)
```

Again, iterative and regression error plots can be used for analysis, but the regression error plot is less informative than for a regression problem (see Figure 3). Also, a function for displaying receiver operating characteristics (ROC) is included in the package. ROC plots are usually used for the analysis of classification problems with two classes. With more classes, for every class a ROC plot can be generated by combining all other classes to one class, and using the output of only the output of the corresponding neuron for drawing the ROC plot (see Figure 3 for examples):

```
R> plotIterativeError(model)
R> plotRegressionError(predictions[, 2], iris$targetsTest[, 2], pch = 3)
R> plotROC(fitted.values(model)[, 2], iris$targetsTrain[, 2])
R> plotROC(predictions[, 2], iris$targetsTest[, 2])
```

However, using ROC plots in this way might be confusing, especially if many classes are present in the data. Yet in the given example with three classes it is probably more informative to analyze confusion matrices, using the function `confusionMatrix`. A confusion matrix shows the amount of times the network erroneously classified a pattern of class X to be a member of class Y. If the class labels are given as a matrix to the function `confusionMatrix`, it encodes them using the standard setting. Currently, this standard setting is a strict winner-takes-all (WTA) algorithm that classifies each pattern to the class that is represented by the
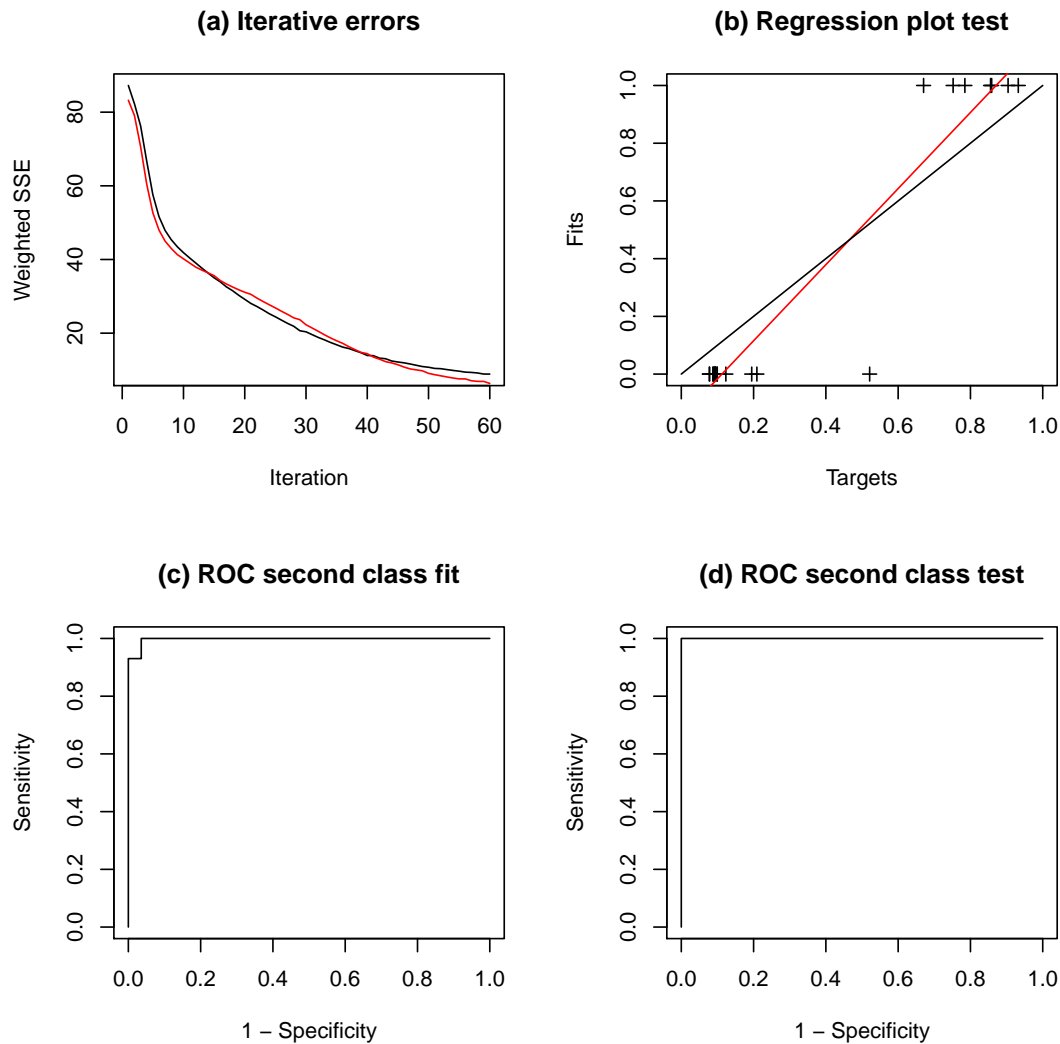
Figure 3: A multi-layer perceptron trained with the `iris` dataset. (a) The iterative error plot of both training (black) and test (red) error. (b) The regression plot for the test data. As a classification is performed, ideally only the points $(0,0)$ and $(1,1)$ would be populated. (c) ROC plot for the second class against all other classes, on the training set. (d) Same as (c), but for the test data.

neuron having maximal output activation, regardless of the activation of other units. For other encoding algorithms, the class labels can be encoded manually. In the following example, besides the default, the `402040` method is used, which is named after its default parameters, the two thresholds $l = 0.4$, and $h = 0.6$. In this configuration, these two thresholds divide the $[0, 1]$-interval into a lower part with 40% of the values, a middle part of 20% of the values and an upper part containing 40% of the values. The method classifies a pattern to the corresponding class of an output neuron, if this output neuron has an activation in the upper part, and all other neurons have an activation in the lower part. Otherwise, the pattern is

treated as unclassified. In the current implementation, unclassified patterns are represented by a zero as the class label. If `WTA` is used with standard settings, no unclassified patterns occur. Both `402040` and `WTA` are implemented as described in Zell *et al.* (1998). In the following, we show the calculation of confusion matrices for the training and the test dataset:

```
R> confusionMatrix(iris$targetsTrain, fitted.values(model))

       predictions
targets  1  2  3
      1 42  0  0
      2  0 40  3
      3  0  1 41

R> confusionMatrix(iris$targetsTest, predictions)

       predictions
targets 1 2 3
      1 8 0 0
      2 0 7 0
      3 0 0 8

R> confusionMatrix(iris$targetsTrain, encodeClassLabels(fitted.values(model),
+    method = "402040", l = 0.4, h = 0.6))

       predictions
targets  0  1  2  3
      1  0 42  0  0
      2  5  0 38  0
      3  3  0  0 39
```

Finally, we can have a look at the weights of the newly trained network, using the function `weightMatrix` (output is omitted):

```
R> weightMatrix(model)
```

6.3. Self-organizing map for clustering

A self-organizing map is an unsupervised learning method for clustering (Kohonen 1988). Similar input patterns result in spatially near outputs on the map. For example, a SOM can be trained with the `iris` data by:

```
R> model <- som(irisValues, mapX = 16, mapY = 16, maxit = 500,
+    targets = irisTargets)
```

The `targets` parameter is optional. If given, a labeled version of the SOM is calculated, to see if patterns from the same class are present as groups in the map. As for large pattern sets calculation of the outputs can take a long time, the parameters `calculateMap`, `calculateActMaps`, `calculateSpanningTree`, and `saveWinnersPerPattern` can be used to control which results are computed. Component maps are always computed. The results in more detail are:

- `model$actMaps`: An activation map is simply the network activation for one pattern. The activation maps list `actMaps` contains a list of activation maps for each pattern. If there are many patterns, this list can be very large. All the other results can be computed from this list. So, being an intermediary result, with limited use especially if many patterns are present, it is not saved if not explicitly requested by the parameter `calculateActMaps`.

- `model$map`: The most common representation of the self-organizing map. For each pattern, the winner neuron is computed from its activation map. As unit activations represent Euclidean distances, the winner is the unit with minimal activation. The map shows then, for how many patterns each neuron was the winner.

- `model$componentMaps`: For each input dimension there is one component map, showing where in the map this input component leads to high activation.

- `model$labeledMap`: A version of the map where for each unit the target class number is determined, to which the majority of patterns where the neuron won belong to. So, performance of the (unsupervised) SOM learning can be controlled using a problem that could also be trained with supervised methods.

- `model$spanningTree`: It is the same as `model$map`, except that the numbers do not represent the amount of patterns where the neuron won, but the number identifies the last pattern that led to minimal activation in the neuron. In contrast to the other results of SOM training, the spanning tree is available directly from the **SNNS** kernel. As the other results are probably more informative, the spanning tree is only interesting if the other functions require high computation times, or if the original **SNNS** implementation is needed.

The resulting SOM can be visualized using the `plotActMap` function which displays a heat map, or by any other R standard method, e.g., `persp`. If some units win much more often than most of the others, a logarithmic scale may be appropriate (plots are shown in Figure 4):

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

The component maps can be visualized in the same way as the other maps (Figure 5 shows the plots):

```
R> for(i in 1:ncol(irisValues)) plotActMap(model$componentMaps[[i]],
+    col = rev(topo.colors(12)))
```

6.4. An ART2 network

ART networks (Grossberg 1988) are unsupervised learning methods for clustering. They offer a solution to a central problem in neural networks, the stability/plasticity dilemma, which

(a) SOM of the iris example

(b) SOM, logarithmic scale

(c) SOM, logarithmic scale

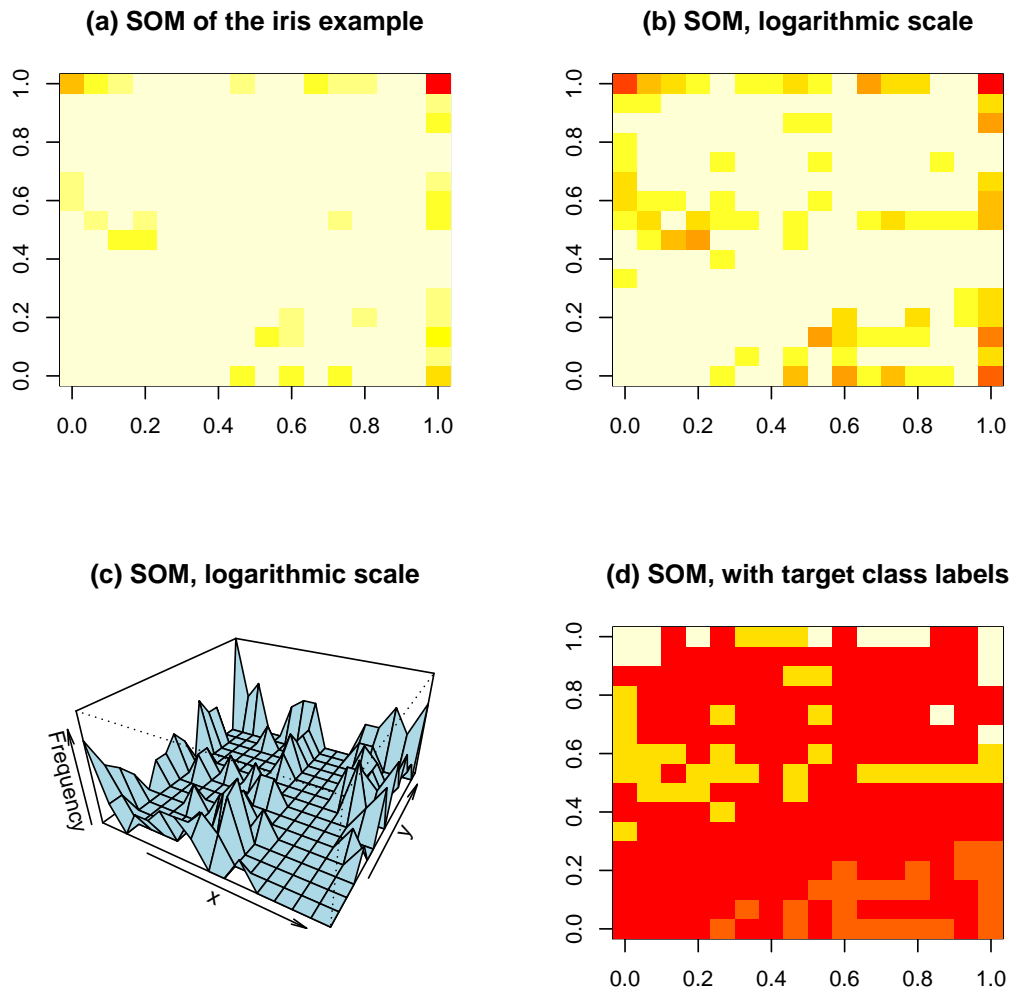**(d) SOM, with target class labels**

Figure 4: A SOM trained with the `iris` dataset. (a) A heat map that shows for each unit the amount of patterns where the unit won, from no patterns (white) to many patterns (red). (b) Same as (a), but on a logarithmic scale. (c) Same as (b), but as a perspective plot instead of a heat map. (d) Labeled map, showing for each unit the class to which the majority of patterns belong, for which the unit won.

means, that in general it is difficult in neural networks to learn new things without altering/deleting things already present in the net. In ART networks, plasticity is implemented in the way that new input patterns may generate new prototypes/cluster centers, if they are not represented yet by another prototype of the net. Stability is present, as a prototype is not altered by all new patterns, but only by new patterns that are similar to the prototype. The ART1 networks (Carpenter and Grossberg 1987b) only allow binary input patterns, ART2 (Carpenter and Grossberg 1987a) was developed for real-valued inputs. In the **SNNS** example `art2_tetra`, which we reimplement here, the inputs are noisy coordinates of the corners of a tetrahedron.
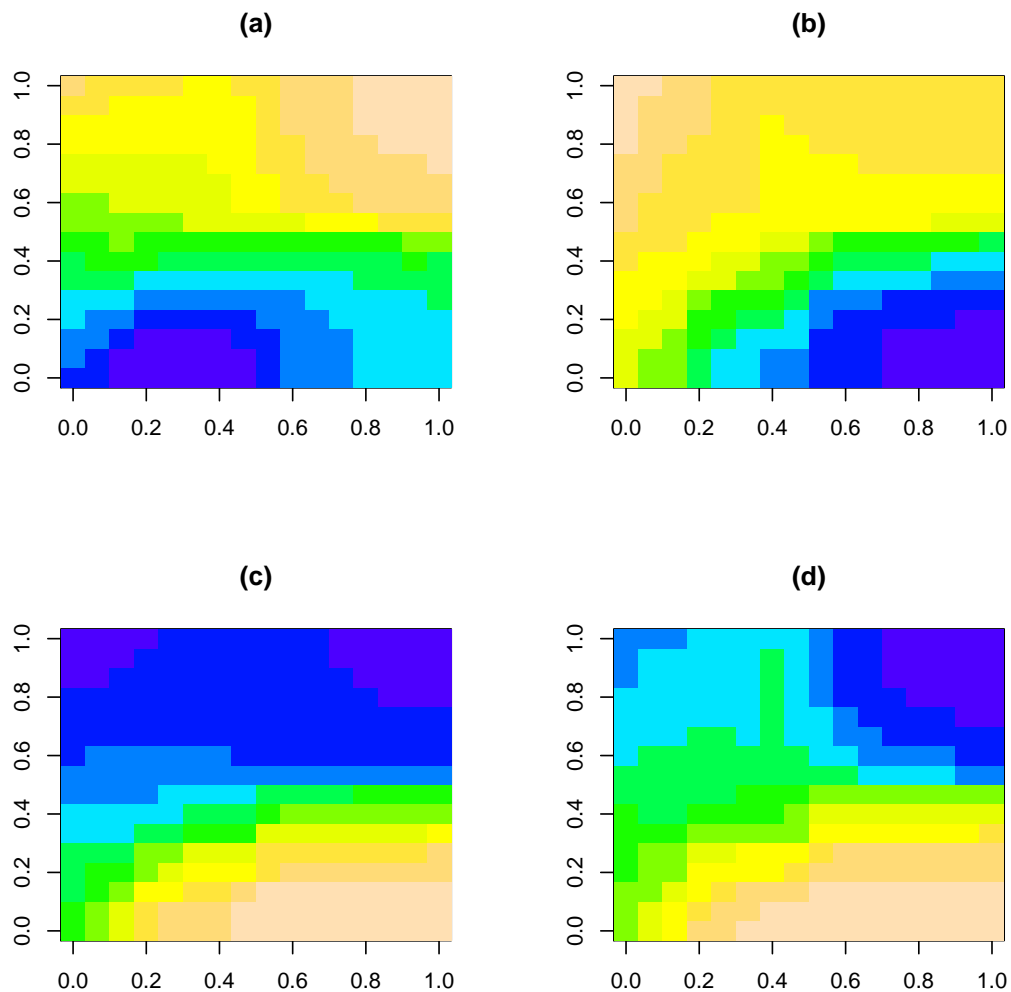
Figure 5: (a)–(d) Component maps for the SOM trained with the `iris` data. As the `iris` dataset has four inputs, four component maps are present that show for each input, where in the map it leads to high activation.

The architecture parameter `f2Units` defines the amount of hidden units present in the f2-layer of the network, and so gives the maximal amount of clusters that can be saved in the net (for details, see Zell *et al.* (1998) and Herrmann (1992)).

The model can be built in the following way:

```
R> patterns <- snnsData$art2_tetra_med.pat
R> model <- art2(patterns, f2Units = 5,
+    learnFuncParams = c(0.99, 20, 20, 0.1, 0),
+    updateFuncParams = c(0.99, 20, 20, 0.1, 0))
```

For visualization of this example, it is convenient to use the R package **scatterplot3d** (Ligges

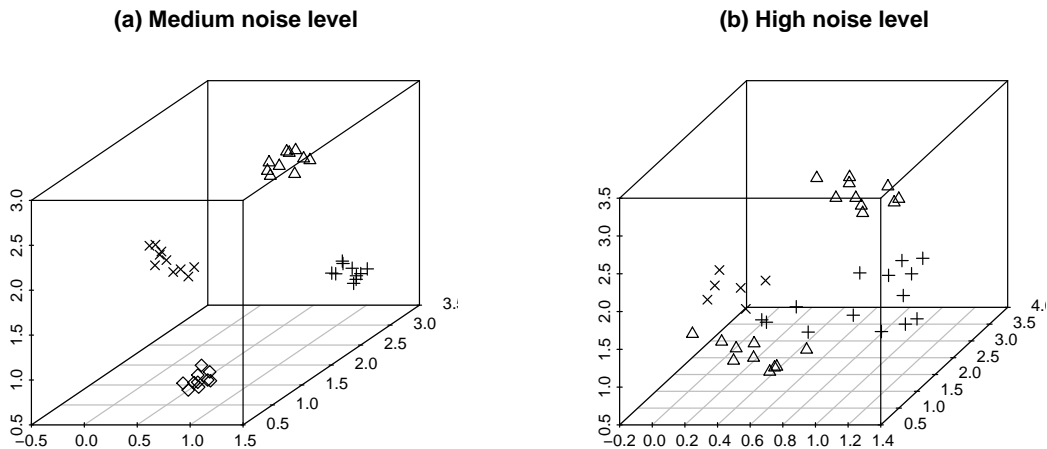**(a) Medium noise level**   **(b) High noise level**

Figure 6: An ART2 network trained with noisy input data which represent the corners of a three-dimensional tetrahedron. (a) Data with medium noise level, the method correctly assumes four different clusters and clusters the points correctly. (b) Data with high noise level. The method generates three clusters. Note that the points of two of the corners are in one cluster though they are not spatially near. As ART2 uses normalized vectors, it only is able to take into account the direction of the vectors, which yields this unintuitive result (Herrmann 1992).

and Maechler 2003), for three-dimensional scatter plots:

```
R> library("scatterplot3d")
R> scatterplot3d(patterns, pch = encodeClassLabels(model$fitted.values))
```

Figure 6 shows this scatter plot, and the result of the same processing, but using as input data snnsData$art2_tetra_high.pat, which has more noise added.

6.5. Combined use of RSNNS and independent SNNS software

While **RSNNS** includes the major part of **SNNS** functionality, there are some bits left out, which can be interesting for some specific purposes: for example, the **SNNS** GUI is a very convenient tool for interactive visual development of a neural network topology or for manual analysis of a network weight's role. **RSNNS** offers functionality to interact with **SNNS**. This includes mainly reading/writing native dataset files (.pat, where files containing patterns with variable length are currently not supported) or neural network files (.net), as well as a rudimentary parser for .res files. This way, data and networks can be interchanged with an installation of the original **SNNS**, e.g., to visualize the network architecture in **SNNS**, or to train a net in **SNNS** and use R to analyze the capabilities of the net.

```
R> exportToSnnsNetFile(model, filename)
R> readPatFile(filename)
R> savePatFile(inputs, targets, filename)
R> readResFile(filename)
```

The `.pat` file methods make use of the original **SNNS** methods. Furthermore, `.net` files can be loaded and saved with the normal **SNNS** kernel methods `loadNet` and `saveNet`.

7. Neural network packages in R

In this section, we review the packages available directly in R or from CRAN implementing neural networks, and compare their functionality with algorithms available through **RSNNS**. As all neural network packages we are aware of on CRAN implement either feed-forward networks or Kohonen networks (LVQ, SOM), we discuss the packages grouped accordingly in this section. Furthermore, we give a general discussion about the functionality that has become available through **RSNNS** to R, and on general limitations of both **SNNS** and the wrapper package.

The only package present on CRAN yet that tackles connection between **SNNS** and R is **write.snns** (Castejón Limas *et al.* 2007), which implements a function to export data from R to a **SNNS** pattern file (`.pat`). This functionality is included in **RSNNS**, using directly the original **SNNS** functions for input and output of pattern files.

7.1. Feed-forward neural networks

There are several packages available for R that implement multi-layer perceptrons. Furthermore, implementations of quantile regression neural networks and flexible radial basis function networks exist.

nnet The package **nnet** (Venables and Ripley 2002) is part of the recommended R packages that usually ship directly with R. So, **nnet** can be considered the R standard neural network package. It implements a multi-layer perceptron with one hidden layer. For weight adjustment, it does not use backpropagation nor one of its variants, but a general quasi-Newton optimization procedure, the BFGS algorithm. Ripley (2007) argues that to "use general algorithms from unconstrained optimization [...] seems the most fruitful approach". A similar method, the scaled conjugate gradient (SCG, Møller 1993), is implemented in **SNNS**. SCG combines a conjugate gradient approach with ideas from the Levenberg-Marquardt algorithm. Møller (1993) compares SCG with standard backpropagation, another conjugate gradient algorithm, and the BFGS algorithm. In his comparison, the SCG performs best.

AMORE The package **AMORE** (Castejón Limas *et al.* 2010) implements the "TAO-robust backpropagation learning algorithm" (Pernía Espinoza *et al.* 2005), which is a backpropagation learning algorithm designed to be robust against outliers in the data. Furthermore, adaptive backpropagation and adaptive backpropagation with momentum term, both in online and batch versions, are implemented. The algorithms use an individual learning rate for each unit. Adaptive backpropagation procedures are not implemented in this way in **SNNS**, but using different learning rates for the units or the weights is a common idea for enhancing neural network learning procedures, and e.g., resilient backpropagation, which is implemented in **SNNS**, adapts the learning rate for each weight.

Furthermore, the package aims at giving a general framework for the implementation of neural networks (`http://rwiki.sciviews.org/doku.php?id=packages:cran:amore`), i.e.,

for defining units, their activation functions, connections, etc. The **SNNS** kernel interface implements such structures that are used internally by the algorithms implemented in **SNNS**. This interface can be accessed by the low-level interface functions of **RSNNS**.

neuralnet   The package **neuralnet** (Fritsch *et al.* 2010) implements standard backpropagation and two types of resilient backpropagation (Riedmiller and Braun 1993; Riedmiller 1994). These algorithms are also available in **SNNS**, in implementations of the original authors of the algorithms. Furthermore, the package implements a "modified globally convergent version of the algorithm" presented by Anastasiadis *et al.* (2005), and a method for "the calculation of generalized weights", which are not present in **SNNS**.

monmlp   The package **monmlp** (Cannon 2011a) implements a multi-layer perceptron with partial monotonicity constraints (Zhang and Zhang 1999). The algorithm allows for the definition of monotonic relations between inputs and outputs, which are then respected during training. If no constraints are defined, the algorithms behave as the usual, unconstrained versions. Implemented are standard backpropagation, and learning using a nonlinear least-squares optimizer. Furthermore, a stopping criterion using a bootstrap procedure is implemented. In **SNNS**, monotonic constraints methods are not implemented. Regarding the standard procedures, backpropagation and SCG, which uses a general optimizer, are present in **SNNS**. Stopping criteria are not implemented in **SNNS**, but as this part is controlled by **RSNNS** in the R code, could be considered for future versions of the package.

qrnn   The package **qrnn** (Cannon 2011b) implements a quantile regression neural network, which can be used to produce probability density forecasts, especially in environments with both continuous and discrete input variables. This is not implemented in **SNNS**.

frbf   The package **frbf** (Martins 2009) implements an algorithm for flexible radial basis functions (Falcao *et al.* 2006). The algorithm can be used for classification only. The algorithm constructs in a first phase of unsupervised learning the network topology from the training data, and later uses different kernels for each class. The algorithm is not included in **SNNS**, but standard radial basis functions are. Furthermore, an algorithm that also is only suitable for classification and constructs the network topology on its own is present with the RBF dynamic decay adjustment algorithm (Berthold and Diamond 1995).

7.2. Kohonen networks

Several packages in R implement SOMs. The version that is implemented in **SNNS** uses Euclidean distance and a quadratic neighborhood.

class   The package **class** (Venables and Ripley 2002) is one of the recommended packages in R . It implements a SOM with rectangular or hexagonal grid, and the learning vector quantization algorithms LVQ1, LVQ2.1, LVQ3, and OLVQ. LVQ1 only adapts the winning prototype, whereas LVQ2.1 and LVQ3 also adapt the second best prototype. OLVQ uses a different learning rate for every unit. An implementation of LVQ is present in **SNNS**, where it is called dynamic LVQ, because it starts with an empty network/codebook, and adds successively new units.

som The package **som** (Yan 2010) implements a self-organizing map, focused on its application in gene clustering. It has hexagonal and rectangular topologies implemented, as well as basic visualization functionality.

kohonen The package **kohonen** (Wehrens and Buydens 2007) implements a standard SOM, as well as a supervised SOM with two parallel maps, and a SOM with multiple parallel maps. It is based on the package **class**. It has hexagonal and quadratic neighborhood relationships implemented, as well as toroidal maps. Besides the usual Euclidean distance, for class labels the Tanimoto distance can be used.

When class labels are available, there are various potential possibilities of how to use them during SOM training. An easy possibility is to use them (as in Section 6.3) merely during visualization. Another possibility that can be used with the standard SOM, is to add the class labels as additional feature, possibly with a weighting. However, the method implemented in this package, presented by Melssen *et al.* (2006), uses two different maps. The algorithm is an enhancement of counterpropagation (Hecht-Nielsen 1987), which is implemented in **SNNS**.

Besides the supervised versions of the SOM, the package also offers various visualization possibilities for the maps.

wccsom The package **wccsom** (Wehrens 2011) is another package from the authors of the **kohonen** package. It implements another distance measure, the weighted cross-correlation (WCC). Furthermore, it implements maps that start with few units, and add additional units during training by interpolation. This way, training can be performed faster.

7.3. Possibilities and limitations of RSNNS

So far, neural network methods are scattered along several packages. **RSNNS** addresses this lack of a standard neural network package in R by making the **SNNS** functionality usable from R, and therewith offering a uniform interface to many different standard learning procedures and architectures. For those models or algorithms for which there are already implementations available in R, such as for resilient propagation, standard backpropagation, or DLVQ, **RSNNS** offers alternatives that can be used complementary and for comparison. Regarding SOMs, there exist yet sophisticated packages in R, which offer powerful visualization procedures and flexible network construction, so use of the SOM standard implementation in **RSNNS** will usually not have benefits. But **RSNNS** also makes available a lot of architectures and learning procedures that were not present for R before, to the best of our knowledge, such as partial recurrent neural networks (Jordan and Elman nets), the ART theory, associative memories, Hopfield networks, the cascade correlation networks, or network pruning algorithms.

SNNS is a robust and fast software with standard implementations of a large amount of neural network techniques (see Section 2), validated by many years of employment by a huge group of users. Though **RSNNS** successfully overcomes some of the main problems that the use of **SNNS** in a modern experiment design raises, some others persist. As active development terminated in 1998, newer network types are not present. However, building a comprehensive up-to-date neural network standard package is a difficult task. The packages currently available from CRAN mainly focus on implementation of special types of architectures and/or learning functions. An important question in this context seems, if **RSNNS** could provide a suitable architecture to add new network types. Though **SNNS** in general is well-written

software with a suitable architecture and kernel API, to extend the kernel, knowledge of its internals is necessary, and the networks would need to be added directly in `SnnsCLib`, as the current wrapping mechanism of **RSNNS** is one-way, i.e., **RSNNS** provides no mechanisms that allow to implement learning, initialization, or update functions for use from within the **SNNS** kernel in R.

Regarding other limitations of the wrapping mechanism itself, the low-level interface offers full access to not only the kernel user interface, but also some functions that were not part of the kernel (bignet), and some functions that were not part of the original **SNNS**, but implemented for `SnnsCLib`. Some functions of the kernel interface are currently excluded from wrapping (see the file KnownIssues, that installs with the package), but this does not limit the functionality of **RSNNS**, as those functions usually implement redundant features, or features that can be easily reimplemented from R. So, with the low-level interface, all of the functionality of **SNNS** can be used.

In contrast to the direct mapping of the **SNNS** kernel to R functions of the low-level interface, the high-level interface implements task-oriented functionality that is mainly inspired by the original **SNNS** examples. The high-level functions define the architecture of the net, and the way the learning function is called (e.g., iteratively or not). So, the high-level functions are still pretty flexible, and e.g., the function `mlp` is suitable for a wide range of learning functions. Naturally, there are cases where such flexibility is not necessary, so for example the ART networks are only suitable for their specific type of learning.

8. Conclusions

In this paper, we presented the package **RSNNS** and described its main features. It is essentially a wrapper in R for the well-known **SNNS** software. It includes an API at different levels of trade-off between flexibility/complexity of the neural nets and convenience of use. In addition it provides several tools to visualize and analyze different features of the models. **RSNNS** includes a fork of **SNNS**, called `SnnsCLib`, which is the base for the integration of the **SNNS** functionality into R, an environment where automatization/scriptability and parallelization play an important role. Through the use of **Rcpp**, the wrapping code is kept straightforward and well encapsulated, so that `SnnsCLib` could also be used on its own, in projects that do not use R. Flexibility and full control of the networks is given through the **RSNNS** low-level interface `SnnsR`. The implemented calling mechanism enables object serialization and error handling. The high-level interface `rsnns` allows for seamless integration of many common **SNNS** methods in R programs.

In addition, various scenarios exist where the combined usage of the original **SNNS** and **RSNNS** can be beneficial. **SNNS** can be used as an editor to build a network which afterwards can be trained and analyzed using **RSNNS**. Or, a network trained with **RSNNS** can be saved along with its patterns, and **SNNS** can be used for a detailed analysis of the behavior of the net (or parts of the net or even single units or connections) on certain patterns.

As discussed in Section 7, packages currently available in R are focused on distinct network types or applications, so that **RSNNS** is the first general purpose neural network package for R, and could become the new standard for neural networks in R.

Acknowledgments

References

Anastasiadis AD, Magoulas GD, Vrahatis MN (2005). "New Globally Convergent Training Scheme Based on the Resilient Propagation Algorithm." *Neurocomputing*, **64**(1–4), 253–270.

Bergmeir C, Benítez JM (2012). *Neural Networks in R Using the Stuttgart Neural Network Simulator:* **RSNNS**. R package version 0.4-3, URL http://CRAN.R-Project.org/package=RSNNS.

Berthold MR, Diamond J (1995). "Boosting the Performance of RBF Networks with Dynamic Decay Adjustment." In *Advances in Neural Information Processing Systems*, pp. 521–528. MIT Press.

Bishop CM (2003). *Neural Networks for Pattern Recognition*. Oxford University Press.

Cannon AJ (2011a). **monmlp**: *Monotone Multi-Layer Perceptron Neural Network*. R package version 1.1, URL http://CRAN.R-project.org/package=monmlp.

Cannon AJ (2011b). "Quantile Regression Neural Networks: Implementation in R and Application to Precipitation Downscaling." *Computers & Geosciences*, **37**(9), 1277–1284.

Carpenter GA, Grossberg S (1987a). "ART 2: Self-Organization of Stable Category Recognition Codes for Analog Input Patterns." *Applied Optics*, **26**(23), 4919–4930.

Carpenter GA, Grossberg S (1987b). "A Massively Parallel Architecture for a Self-Organizing Neural Pattern Recognition Machine." *Computer Vision, Graphics and Image Processing*, **37**, 54–115.

Carpenter GA, Grossberg S, Reynolds JH (1991). "ARTMAP: Supervised Real-Time Learning and Classification of Nonstationary Data by a Self-Organizing Neural Network." *Neural Networks*, **4**(5), 565–588.

Castejón Limas M, Ordieres Meré JB, de Cos Juez FJ, Martínez de Pisón Ascacibar FJ (2007). **write.snns**: *Function for Exporting Data to SNNS Pattern Files*. R package version 0.0-4.2, URL http://CRAN.R-project.org/package=write.snns.

Castejón Limas M, Ordieres Meré JB, González Marcos A, Martínez de Pisón Ascacibar FJ, Pernía Espinoza AV, Alba Elías F (2010). **AMORE**: *A MORE Flexible Neural Network Package*. R package version 0.2-12, URL http://CRAN.R-project.org/package=AMORE.

Cun YL, Denker JS, Solla SA (1990). "Optimal Brain Damage." In *Advances in Neural Information Processing Systems*, pp. 598–605. Morgan Kaufmann.

Eddelbuettel D, François R (2011). "**Rcpp**: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18.

Elman JL (1990). "Finding Structure in Time." *Cognitive Science*, **14**(2), 179–211.

Fahlman S, Lebiere C (1990). "The Cascade-Correlation Learning Architecture." In *Advances in Neural Information Processing Systems 2*, pp. 524–532. Morgan Kaufmann.

Fahlman SE (1988). "An Empirical Study of Learning Speed in Back-Propagation Networks." *Technical Report CMU-CS-88-162*, Carnegie Mellon, Computer Science Department.

Fahlman SE (1991). "The Recurrent Cascade-Correlation Architecture." In *Advances in Neural Information Processing Systems 3*, pp. 190–196. Morgan Kaufmann.

Falcao AO, Langlois T, Wichert A (2006). "Flexible Kernels for RBF Networks." *Neurocomputing*, **69**(16-18), 2356–2359.

Fritsch S, Guenther F, Suling M (2010). **neuralnet**: *Training of Neural Networks*. R package version 1.31, URL http://CRAN.R-project.org/package=neuralnet.

Grossberg S (1988). *Adaptive Pattern Classification and Universal Recoding. I.: Parallel Development and Coding of Neural Feature Detectors*, chapter I, pp. 243–258. MIT Press.

Haykin SS (1999). *Neural Networks: A Comprehensive Foundation*. Prentice Hall.

Hecht-Nielsen R (1987). "Counterpropagation Networks." *Applied Optics*, **26**(23), 4979–4984.

Herrmann KU (1992). *ART – Adaptive Resonance Theory – Architekturen, Implementierung und Anwendung*. Diplomarbeit 929, IPVR, University of Stuttgart.

Hopfield JJ (1982). "Neural Networks and Physical Systems with Emergent Collective Computational Abilities." *Proceedings of the National Academy of Sciences of the United States of America*, **79**(8), 2554–2558.

Hudak M (1993). "RCE Classifiers: Theory and Practice." *Cybernetics and Systems*, **23**(5), 483–515.

Jordan MI (1986). "Serial Order: A Parallel, Distributed Processing Approach." *Advances in Connectionist Theory Speech*, **121**(ICS-8604), 471–495.

Jurik M (1994). "BackPercolation." *Technical report*, Jurik Research. URL http://www.jurikres.com/.

Kohonen T (1988). *Self-Organization and Associative Memory*, volume 8. Springer-Verlag.

Lang KJ, Waibel AH, Hinton GE (1990). "A Time-Delay Neural Network Architecture for Isolated Word Recognition." *Neural Networks*, **3**(1), 23–43.

Lange JM, Voigt HM, Wolf D (1994). "Growing Artificial Neural Networks Based on Correlation Measures, Task Decomposition and Local Attention Neurons." In *IEEE International Conference on Neural Networks – Conference Proceedings*, volume 3, pp. 1355–1358.

Ligges U, Maechler M (2003). "**scatterplot3d** – An R Package for Visualizing Multivariate Data." *Journal of Statistical Software*, **8**(11), 1–20. URL http://www.jstatsoft.org/v08/i11/.

Martins F (2009). ***frbf**: Implementation of the 'Flexible Kernels for RBF Network' Algorithm*. R package version 1.0.1, URL http://CRAN.R-project.org/package=frbf.

Melssen W, Wehrens R, Buydens L (2006). "Supervised Kohonen Networks for Classification Problems." *Chemometrics and Intelligent Laboratory Systems*, **83**(2), 99–113.

Møller MF (1993). "A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning." *Neural Networks*, **6**(4), 525–533.

Pernía Espinoza AV, Ordieres Meré JB, Martínez de Pisón Ascacibar FJ, González Marcos A (2005). "TAO-Robust Backpropagation Learning Algorithm." *Neural Networks*, **18**(2), 191–204.

Poggio T, Girosi F (1989). "A Theory of Networks for Approximation and Learning." *Technical Report A.I. Memo No.1140, C.B.I.P. Paper No. 31*, MIT Artifical Intelligence Laboratory.

R Development Core Team (2011). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

Riedmiller M (1994). "Rprop – Description and Implementation Details." *Technical report*, University of Karlsruhe.

Riedmiller M, Braun H (1993). "Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm." In *1993 IEEE International Conference on Neural Networks*, pp. 586–591.

Ripley BD (2007). *Pattern Recognition and Neural Networks*. Cambridge University Press.

Rojas R (1996). *Neural Networks: A Systematic Introduction*. Springer-Verlag.

Rosenblatt F (1958). "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain." *Psychological Review*, **65**(6), 386–408.

Rumelhart DE, Clelland JLM, Group PR (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. MIT Press.

Venables WN, Ripley BD (2002). *Modern Applied Statistics with S*. Springer-Verlag, New York. URL http://www.stats.ox.ac.uk/pub/MASS4.

Wehrens R (2011). ***wccsom**: SOM Networks for Comparing Patterns with Peak Shifts*. R package version 1.2.4, URL http://CRAN.R-project.org/package=wccsom.

Wehrens R, Buydens LMC (2007). "Self- and Super-Organizing Maps in R: The **kohonen** Package." *Journal of Statistical Software*, **21**(5), 1–19. URL http://www.jstatsoft.org/v21/i05/.

Yan J (2010). ***som**: Self-Organizing Map*. R package version 0.3-5, URL http://CRAN.R-project.org/package=som.

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

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

Zhang H, Zhang Z (1999). "Feedforward Networks with Monotone Constraints." In *Proceedings of the International Joint Conference on Neural Networks*, volume 3, pp. 1820–1823.

Affiliation:

Christoph Bergmeir, José M. Benítez
Department of Computer Science and Artificial Intelligence
E.T.S. de Ingenierías Informática y de Telecomunicación
CITIC-UGR, University of Granada
18071 Granada, Spain
E-mail: c.bergmeir@decsai.ugr.es, j.m.benitez@decsai.ugr.es
URL: http://dicits.ugr.es/, http://sci2s.ugr.es/