The nnlib2 C++ Library and nnlib2Rcpp R package for Artificial Neural Networks.
Vasilis N. Nikolaidis
Revision: 0.1 - under development

1 Introduction

1.1 nnlib2

The nnlib2 library is a small collection of C++ classes for implementation of Neural Networks (NN). It contains classes that model typical NN parts and components (processing nodes, connections, layers, groups of connections etc) as well as a class for composing these parts in complete NNs. Users of this library can combine predefined parts and components with ones they define by sub-classing the current ones. In all cases,considerable predefined functionality in available to aid the process, s.a methods for building models, presenting data and retrieving output, encoding and mapping data, serialization to and from files etc. The produced models can be included in and employed by any C++ application. The nnlib2 code can be found on GitHub at this link.

For more information on nnlib2, go to Section 2, below.

1.2 nnlib2Rcpp

The R package nnlib2Rcpp interfaces R with nnlib2. The package contains the entire nnlib2, as well as R functions that employ ready-to-use models implemented with that library. Furthermore, the package (via module 'NN') allows the instantiation of individual NN parts and components (processing nodes, connections, layers, groups of connections etc) , combine parts in custom NNs, and use, control and monitor them in R. Thus, new types of NN parts can be developed using R-related tools (Rtools and RStudio), and then employed in R; the same components can be transferred and used in a pure C++ application (inside a nnlib2 'nn' class object), if needed.

Stable version of the package (along with source and reference manual) can be found on CRAN at this link.

Development version can be found on GitHub at this link.

For more information on nnlib2Rcpp, go to Section 3, below.

2 The nnlib2 library

The nnlib2 library is a collection of C++ classes and templates that provide simple predefined base components useful for implementing and using NNs. The nnlib2 library may interest NN students and experimenters who prefer implementing new NN components and models using a small collection of base classes and templates whose purpose is clear, have simple interfaces, follow familiar NN concepts, and allow significant control. A small collection of ready-to-use NN components and models are also implemented and included in nnlib2.

The nnlib2 library requires a standard C++ compiler (has been tested with various GNU and Microsoft Visual Studio versions), and produces lightweight, standalone NNs that can be invoced within any C++ application. Being written a compiled language, the produced NNs are relatively fast and can be used in real applications and problems of small data size and NN complexity.

2.1 The nnlib2 class structure

The `nnlib2` library consists of several C++ class and class-template definitions, most of which match typical NN parts, sub-components and components (processing nodes, connections, layers, groups of connections, complete NNs etc). Sub-components include processing elements (PEs a.k.a. nodes) and connections; these are grouped in components (layers of PEs, sets of connections, and complete NNs) to provide typical NN functionality that can be inherited, overridden and/or extended when implementing a specific new NN model behavior.

Two important virtual methods are provided by all component and sub-component classes: `encode()`— invoked when the NN is trained (training stage), and `recall()`— applied when retrieving data from the model (mapping stage), and thus should contain the core instructions for data processing.

Some of the classes in `namespace nnlib2` are briefly outlined below. A brief (and somewhat simplified) outline of the most significant classes in `nnlib2` is also shown in the class-diagram of Figure 1.

Class '`pe`' for processing elements (PEs). Provides typical PE (node) functionality and placeholders for internal input, activation, and threshold functions. All objects of this class maintain typical PE internal values s.a. input, bias, output etc. and inherit functionality for collecting inputs, applying the internal PE functions, state serialization etc. If left unmodified such objects will be referred to as generic PEs.

Class '`connection`'. Provides typical connection functionality for communicating data between two PEs. Objects of this class maintain source and destination PE information as well as functions and values (including weights) needed to modify the transferred value. If used without modifications, objects of this class will be referred to as generic connections.

Class '`component`' for a component of the NN topology, such as layers, sets of connections, control components, etc. Provides a common interface and functionality shared by all components (for processing, streaming, etc.). Component-type objects may be registered (added) to the NN's `topology` structure (discussed later) creating complex topologies. Sub-classes that inherit '`component`', include the following:

Class '`layer`', a '`component`' for a layer of PEs. It maintains a layer's '`pe`' objects, and provides functionality to initialize, interface with, trigger processing and in general, manipulate the layer's PEs. A template where generic or model-specific '`pe`' types can be used (as well as a 2-d variation) is provided; it can be sub-classed to define new types of '`layer`' classes with specific behavior.

Class '`connection_set`', a '`component`' for a set of connections between any two '`layer`'s (can be the same layer), and a template where generic or model-specific '`connection`' types can be used and '`connection`' objects are maintained. It includes functionality to create connections between two PEs, initialize, serialize, trigger processing and in general, manipulate a set of connections; it too can be sub-classed to define new types of specialized '`connection_set`' objects.

Class '`nn`' for a neural network. It contains the `topology`, implemented as a double-linked list-based structure that maintains the '`component`' objects (of any type) which constitute the NN. By default the order of components in the `topology` corresponds to the order of processing performed when a NN executes a typical feed-forward operation (or feed-backward if in reverse order), but this can be modified by overriding the '`nn`' `encode()`— and `recall()`— methods. Alternatively, for simple NN topologies, the developer may choose to not use the `topology` structure, define the '`component`' objects as member variables and provide instructions for manipulating them. However, components that are dynamically created and registered to the `topology` are handled automatically by the default '`nn`' predefined methods: encode/recall revocations, display, serialization, deletion etc. may be performed with little or no extra code (subject to the specifics of the particular NN model implemented). Registering the components in the `topology` structure also allows implementation of dynamic and/or multilayer NN models, with complex topologies and "deep(er)"-learning NN configurations. Finally, '`nn`' class objects are also derived from class '`component`', allowing embedment of NNs inside the topology of other NNs.

In addition to the above classes, `nnlib2` includes a collection of several secondary classes. For example, '`aux_control`' components are classes of objects that can be used to provide auxiliary, user-defined functionality (e.g. control, display output, user-break, handle other components or sub-components [create, delete, communicate, call methods], perform data operations [filter, mutate, which_max, feature expansion] etc.). Being themselves '`component`' objects, they can be added to NN's topology, thus be activated during the NN's data processing sequence via their respective `encode()`— and `recall()`— methods. Other secondary classes include helper objects for communicating data, sharing run-time error information etc.

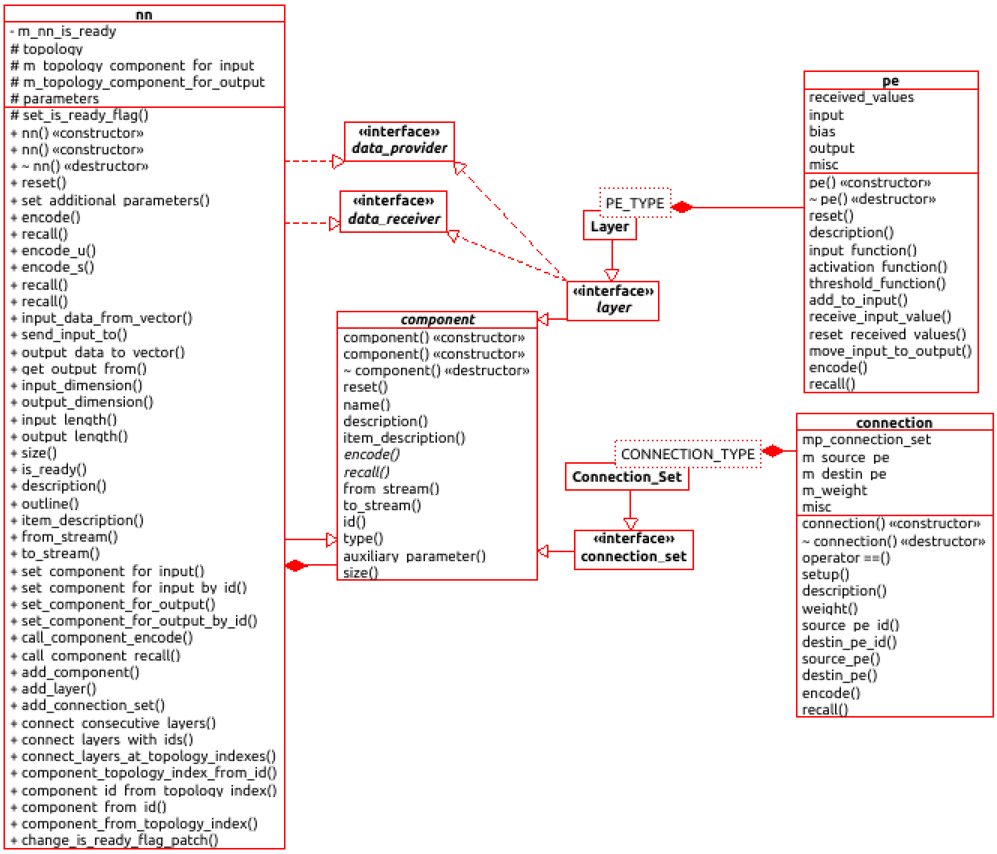

Figure 1: Significant nnlib2 classes.

Some ready-to-use NN model implementations are also included in `nnlib2`, such as versions of Learning Vector Quantization ('`lvq_nn`' class, supervised, subclass of '`nn`') and Self-Organizing Map ('`som_nn`' class, unsupervised, subclass of '`lvq_nn`'), Back-Propagation multilayer perceptron (in '`bp_nn`' class, supervised, subclass of '`nn`'), Autoencoder (in '`bpu_autoencoder_nn`' class, unsupervised, a subclass of '`bp_nn`'), and MAM (in '`mam_nn`', supervised, subclass of '`nn`').

2.2 Defining a new NN model in `nnlib2`

The implementation of a Matrix Associative Memory (MAM) NN using `nnlib2` classes will be presented in this section as an example. MAM is described in (?) and is chosen for its simplicity; it is a supervised model trained by Hebbian learning that stores input-output vector pairs (x, z) by computing their tensor product and storing it in a d x c matrix M (where d and c is the dimensionality (length) of x and z vectors respectively). Encoding is a non-iterative process, where

$$M = x^T z \tag{1}$$

is computed. Recall is also done in a single step where, given input x and matrix M,

$$xM = xx^T z \tag{2}$$

is computed, which simplifies to

$$(x_1^2 + x_2^2 + \ldots + x_d^2)z \tag{3}$$

i.e vector z multiplied by a number. Ideally, if this number equals to 1, a perfect recall of z is performed. Multiple vector pairs can be stored in a single matrix M_s which is the sum of the matrices M_i resulting for each vector pair i. MAM is the basis for several other associate memory models, but even the simple version described here has interesting properties: MAMs are bidirectional (x can be recalled from z as well as z from x) and allow deletion of stored vector pairs. However, this MAM has limited and variable storage capacity which may be enhanced by proper encoding and normalization of the input and output data. Another solution to the storage issue is to employ a system of multiple MAMs, adding new MAMs when the current ability to store a vector pair is exhausted.

While the MAM model is probably best implemented as a series of vector and matrix operations, it can also be realized as a simple NN. Only two layers of PEs, for x and z vectors respectively, are needed. These layers are fully connected (each PE in the first layer is linked to all PEs in the second) with connections whose weights collectively form matrix M_s. PEs apply simple functions: composition of inputs (the PE *input function*) is summation of the incoming values, and the result is typically copied to the output (no *activation function* is used, while a *threshold function* is optional in MAM NN PEs). During the MAM NN feed-forward encode process, the x feature values are input to the corresponding layer PEs, similarly z is input to the other layer, and connection weights (which are initially 0) are adjusted by

$$w_{ij} = w_{ij} + (x_i z_j) \tag{4}$$

where w_{ij} is the weight of the connection between the i-th PE in the first layer (having input equal to x_i [and same output, being the sum of the single input value]) to the j-th PE in the second layer (having input z_j). To retrieve vector z given vector x, x is presented as input to the corresponding layer and a single feed-forward recall step is performed towards the other layer: PEs in the first layer transfer their inputs unmodified (again being the sum of the single input value) and pass them to the connections which multiply their weights and input the results to the second layer PEs which sum them (as mentioned earlier) and output values that collectively form the NN's output vector (ideally resembling z).

To implement this simple NN (or in fact any NN model) using `nnlib2`, the user has to define the model-specific processing behavior. This can be done by modifying '`component`' classes (the NN's '`layer`' and '`connection_set`' classes), sub-component classes ('`pe`' and '`connection`'), or both. In the first case, the new components should have their `encode()`— and `recall()`— functions overridden (if the default behavior does not suffice); if all required processing is defined in these functions, such components could even contain only generic, unmodified '`pe`' and/or 'connection' objects. In the second case, custom 'pe' and 'connection' classes are created and placed in (possibly unmodified) 'layer' and 'connection_set'-based objects. This second approach is used below, where MAM-specific 'pe' and 'connection' classes containing the functionality specific to a simple MAM NN (called sMAM to distinguish it from the one already included in nnlib2), as follows:

(a) PEs. The generic (unmodified) 'pe' class suffices for sMAM and no new 'pe'-based type is needed. Its default encode()— and recall()— functions invoke three methods, with the result of each (a single value) passed to the next, namely input_function()—, activation_function()— and threshold_function()—. This last threshold_function()— produces the final PE output value. The default input_function()— is summation, while the default activation_function()— and threshold_function()— are both set to perform the identity function. Therefore generic, unmodified, 'pe' objects output the sum of their inputs. Modifying these methods and/or 'pe' encode()— and recall()— customizes PE behavior. Any 'pe' object collects and temporarily stores the individual values received as input (via its receive_input_value()— method) and uses input_function()— to process them and produce a single final input value; alternatively, this final input value can be accessed and computed directly (bypassing the collection of individual values and invocation of input_function()—, as briefly discussed later). While this fits perfectly the sMAM example, other NN models may require one or more PE types with modified behavior; to illustrate how this is done, a 'pe'-type ('sMAM_pe') where a threshold_function()— applies sin()— to its incoming data will be defined and used in the sMAM example below:

```
class sMAM_pe : public pe
 {
 DATA threshold_function (DATA value) { return sin(value); }
 };
```

Note that DATA is defined in header nnlib2.h, usually as as double.

(b) Connections. Unlike MAM PEs, connections do need to be modified to provide the MAM-specific processing described earlier. Thus a 'connection'-based class ('sMAM_connection') is defined and its encode()— and recall()— functions modified. Here the 'connection' methods source_pe()— and destin_pe()— (which return the source and destination PEs linked by the connection) are also useful:

```
class sMAM_connection : public connection
 {
 public:
 void encode()
  { weight() = weight() + source_pe().output * destin_pe().input; }
 void recall()
  { destin_pe().receive_input_value( weight() * source_pe().output ); }
 };
```

Function encode()— effectively performs (4), while decode()— sends the input multiplied by weight to the output layer PEs (via their receive_input_value()—) to be summed during their decode()— step (as described earlier).

(c) Define the sMAM component types from component templates. 'Layer' and 'Connection_Set' (note the upper-case letters) are templates for the corresponding base classes, and can be defined to contain objects of the classes created above:

```
typedef Layer <sMAM_pe> sMAM_layer;
typedef Connection_Set <sMAM_connection> sMAM_connection_set;
```

If the defined layers were to contain generic 'pe' objects instead of 'sMAM_pe's, the above sMAM_layer definition would be:

```
typedef Layer <pe> sMAM_layer;
```

While these suffice for sMAM, in other more complex NN models multiple 'layer' and/or 'connection_set'-type classes may need to be defined. Also (as mentioned earlier), in a NN implementation the processing details could be defined at component objects (such as 'layer' and 'connection_set') instead of modifying sub-components ('connection' and/or 'pe'). This is sometimes dictated by the model's algorithm, and unavoidable, or could be useful for certain optimizations. To implement this approach in the sMAM example, components would sub-classed from templates 'Layer' and 'Connection_Set' containing only generic 'pe' and 'connection' objects. Their encode()— and recall()— functions would need to be modified to provide the needed behavior. For example, a MAM-specific 'connection_set' class could have its recall()— function modified to perform for each connection c in the set:

```
destin_pe(c).input = destin_pe(c).input + c.weight() * source_pe(c).output;
```

To allow data processing be defined at component level, the internal variables that sub-components maintain (including weight for 'connection' or input and output for 'pe's are accessible from components. Here it was also chosen to bypass the destination 'pe' input_function()— and directly modify its final input value.

(d) finally, create the class for the actual MAM NN objects, based on 'nn'. Here the specific components will be created and the topology defined. In the sMAM case, only a constructor is needed; once the components are (dynamically) created and registered to the topology, the default 'nn' functions manipulating them suffice:

```
class sMAM_nn : public nn
{
public:

sMAM_nn(int input_length, int output_length)
 :nn("MAM Neural Network")
 {
 topology.append(new sMAM_layer("Input layer", input_length, my_error_flag()));
 topology.append(new sMAM_connection_set);
 topology.append(new sMAM_layer("Output layer", output_length, my_error_flag()));
 connect_consequent_layers();
 set_ready();
 }
};
```

A common local flag (my_error_flag()—) is shared by the NN and its components to communicate run-time errors between them; the 'nn' method connect_consequent_layers()— is called to detect sequences of layers and setup their internal connection sets (fully connecting them with 0 weights - other options, including random or pre-computed weights are available); finally, set_ready()— sets a flag indicating that the 'nn' is ready to encode or decode.

Once defined sMAM_nn objects can be created and used in the C++ project. To create one that maps input vectors of length 3 to output vectors of length 2:

```
sMAM_nn theMAM(3,2);
```

Two functions provided by parent 'nn' class, namely `encode_u()`— and `encode_s()`— can be used for unsupervised and supervised training respectively, presenting data to the NN and triggering data encoding for the entire NN topology. The first, `encode_u()`— by default presents a single data vector to the NN and initiates its encoding, while the second `encode_s()`— is similar but presents a pair of vectors (input and desired output). Data recall functions are also provided by 'nn' class. To encode an input-output vector pair (in corresponding vectors `input` and `output` of length 3 and 2 respectively):

```
theMAM.encode_s( input, 3, output, 2 );
```

and similarly, to get the sMAM output for given `input`:

```
theMAM.recall( input, 3, output_buffer, 2 );
```

where `output_buffer` is a buffer (of length 2) to receive the NN's output.

The core code (without some comments etc) needed to create a similar simple MAM NN is shown below. It can be found in `nnlib2` file `nn_mam.h`.

```
#include "nn.h"
namespace nnlib2 {

// define what MAM connections:
class mam_connection: public connection
{
public:
void encode() { weight() = weight() + source_pe().output * destin_pe().input; }
void recall() { destin_pe().receive_input_value ( weight()*source_pe().output ); }
};

// define components (layers and connections_sets)
// MAM layers are generic (no need to be defined) while
// MAM connection sets simply consist of MAM connections:
typedef Connection_Set<mam_connection> mam_connection_set;

// define MAM NN:
class mam_nn : public NN_PARENT_CLASS
{
  public:

mam_nn()
:nn ("MAM Neural Network") {}

bool setup(int input_length,int output_length)
{
reset();
add_layer( new Layer < pe > ( "Input layer" , input_length ) );
add_connection_set( new mam_connection_set );
add_layer( new Layer < pe > ( "Output layer", output_length ) );;
connect_consecutive_layers();
return no_error();
```

```
}
};
}
```

3 The nnlib2Rcpp R package

The `nnlib2Rcpp` R package provides wrapper R functions for some of the predefined NN models in `nnlib2`, as well as a class for building and employing custom NNs created from components (nodes, layers, connections and connection-sets) defined using `nnlib2`. Package `nnlib2Rcpp` provides a collection of ready-to-use versions of some time-proven neural network models that can directly be used in moderately sized problems, and also some tools to aid the implementation of new neural networks, which may be used for adding models to the collection, experimentation with custom models, or educational purposes. Below is a brief discussion of how the predefined NN collection is used, and how it can be expanded by implementing new models and components.

3.1 Using predefined neural network models in `nnlib2Rcpp`

The package contains several ready-to-use neural network (NN) types. These currently include versions of an auto-encoding NN (`Autoencoder`, for PCA-like dimensionality reduction or expansion), Back-Propagation (`BP`, for input-output mappings), unsupervised Learning Vector Quantization (`LVQu`, for clustering), supervised LVQ (`LVQs`, for supervised classification), and Matrix Associative Memory (`MAM`, for storing vector pairs). BP and LVQ variations are based on their description found in **?**; more information on LVQ-type networks can be found in **?**, for the auto-encoder implementation in **?**, while MAMs are described in **?**. All implemented models accept and process numerical data, usually in the form of vectors or matrices. Details and information for each function can be found in the package reference manual . Two brief examples follow below.

3.2 An unsupervised example

Functions are provided for the predefined NN models that employ an unsupervised training approach (are not trained using a second dataset of desired output); such are the Auto-encoder and Unsupervised-LVQ. For example, placing the `iris` data set (from R package `datasets`) in 3 clusters using Unsupervised LVQ can be done by invoking the related `LVQu` function, in a manner similar to:

```
LVQu( iris_s, 3, number_of_training_epochs = 100, neighborhood_size = 1 )
```

This will return a vector of cluster id numbers (0, 1 or 2) indicating the cluster assigned to each `iris` case. Before doing so, however, some data pre-processing must be done. LVQs require numerical data only, and this data needs to be in [0,1] range, so the complete example is:

```
# Create data to use in examples below (scale iris features to 0..1 range):
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 )
```

```
# Apply unsupervised LVQ
ids <- LVQu( iris_s, 3, 100, 1 )
```

which results in:

```
> ids
  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 [39] 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 [77] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2 1 2 2 1 2 1 2 2 1 2 1
[115] 1 2 1 2 2 1 2 1 2 1 2 2 1 1 2 2 2 2 2 1 1 2 2 2 1 2 2 2 1 2 2 2 1 2 2 1
```

3.3 A supervised example

Models that use supervised training (such as BP, Supervised-LVQ and MAM) are implemented as modules. This allows them to be placed in R variables and structures (such as vectors) and thus maintain state, be trained, used, saved to and restored from files, later retrained or applied to new data etc. For example, `"LVQs"` module is a Supervised Learning LVQ. To train it with `iris_s` data, a proper known classification id, i.e. the desired output indicating the correct species for each `iris_s` case, is required. These ids should be integers from 0 to n-1, where n the number of classes (here n=3 species):

```
iris_desired_class_ids <- as.integer( iris$Species ) - 1
```

A supervised learning LVQ (`LVQs`) object is created and stored in variable `lvq` as follows:

```
lvq <- new( "LVQs" )
```

To encode input-output pairs in the `lvq` object (here for 100 training epochs):

```
lvq$encode( iris_s, iris_desired_class_ids, 100 )
```

Once trained, the model can be used to recall the class id for given data by using its `recall` method. To reclassify the original `iris_s` data and plot results (Figure 2) :

```
lvq_recalled_class_ids <- lvq$recall( iris_s )
plot( iris_s, pch = lvq_recalled_class_ids + 1)
```

In addition to `encode` and `recall`, all supervised NN modules provide methods to display their internal state (`print`) and store it (`save`) or retrieve it (`load`) using files.

3.4 Defining new neural network components and models in `nnlib2Rcpp` (using `nnlib2` classes)

In the next three sections, the process of defining new neural network parts and entire models in C++ is outlined. While R programmers may choose to skip these sections (and only use predefined items) they provide insight on the underlying classes, which is also needed for developing new components that the `"NN"` R module (discussed later) may support. Implementing new components does currently (as of version 0.1.4 described herein) require the package source code, package `Rcpp`, RTools and some familiarity with the C++ language, but since significant NN functionality is provided by the included library of C++ classes (`nnlib2`), this last requirement may be minimal.

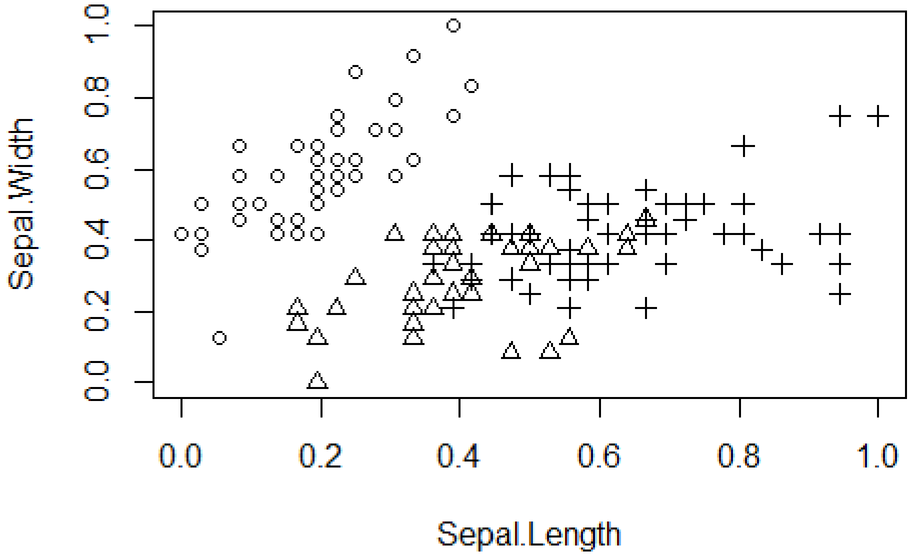

Figure 2: Iris classes recalled by a Supervised LVQ NN.

All predefined neural network models in nnlib2Rcpp are implemented using a collection of C++ base classes and templates for creating NNs. This class library, called nnlib2, is included in the package source and interfaced to R via Rcpp (?). It contains classes that correspond to the basic NN building elements described in classic practical related literature, such as ?, ?, ? and elsewhere. In particular, the defined base classes (and their methods and hierarchy) were initially inspired by the patterns used in ? to present (in writing) a large number of dissimilar NN systems using a scheme common for all models, which breaks each described model into the same components and sub-components, presents how they are organized in a network "topology" and then focuses on the processing performed in each during encoding or recalling (outputting) data. The nnlib2 base classes can be used for implementing different NN parts and models with code that follows a common pattern, allowing functionality reuse and helping code readability. Being written in a compiled language, the produced models are also relatively fast (at least for limited data sizes and NN complexities) and, since nnlib2 is actually a separate standalone target-independent C++ library, they can be included in various other types of C++ projects.

A brief (and somewhat simplified) outline of the most significant classes in nnlib2 is shown in the class-diagram of Figure 1, and is as follows: All NNs are based on class "nn" which maintains one or multiple "component"-class objects in its topology (an internal ordered list of NN components). Such components may include "layer" objects, "connection_set" objects (sets of connections between two layers), entire other "nn" objects, special-purpose components etc. The class "layer" defines components that are layers of processing nodes and provides predefined layer-related functionality; objects of this class internally maintain the nodes (here called 'processing elements') which are objects based on class "pe". Similarly, a "connection_set" is a set of connections between two "layer" objects, and maintains objects inherited from class "connection" which connect particular nodes in the layers. To simplify the creation of layers and sets of connections containing custom "pe" and "connection" types respectively, class templates "Layer" (or "Layer2D" for 2-d layers) and "Connection_Set" can be used (note that template names use capital initial letters). New "layer" and "connection_set" types or sub-classes can also be defined based on these templates.

All aforementioned classes have an 'encode' method, invoked when the NN is trained (training stage), and a 'recall' method invoked when data is retrieved (mapping stage). These two methods contain the core instructions for a single step of data processing (for example a single encoding step in an iterative encoding process). Calling the encode method of a "nn" object (a neural network) probably -and by default- triggers a sequence of encode invocations in its components which, in turn, invoke the encode method of their sub-components ("pe" objects in a "layer", "connection" objects in a "connection_set"). In any case, new NN parts and models can be defined by sub-classing these base classes and overloading the methods (especially constructors and 'encode'/'recall') to modify their functionality. Some examples follow below.

3.5 Defining a new layer of nodes

Assume a layer is needed with a new (rather useless) type of nodes that output the sum of their inputs plus 10 when recalling data. First, the new node class needs to be defined, based on"pe". The "pe" base class provides a method for receiving multiple incoming values (receive_input_value) and an overridable method for initial processing of these values (input_function) whose result is stored in its internal variable 'input'. Subsequent internal processing will produce a final value, and place it in variable 'output'. By default, unmodified (generic) "pe" objects simply output the sum of all incoming values. The new type of nodes called "JustAdd10_pe" could be implemented as shown below, first invoking the base-class recall and then adding 10 (note: this example may be already implemented in the package "additional_parts.h" file on the development version on GitHub i.e. at this link):

```
class JustAdd10_pe : public pe
 {
 public:
 void recall() { pe::recall(); output = output + 10; }
 };
```

Below, the "Layer" template is used, to create (in variable r) a layer containing 25 such nodes labeled "test layer":

```
Layer< JustAdd10_pe > r ( "test layer", 25 );
```

The same template can be used to define new "layer" component types, or sub-classes with customized layer functionality. For example to define a type (named "JustAdd10_layer") :

```
typedef Layer < JustAdd10_pe > JustAdd10_layer;
```

3.6 Defining a new set of connections

To illustrate the definition of a new type of connections (and of an entire NN later), the code for the (particularly simple) MAM NN (as included in file "nn_mam.h" of the package source) is analyzed below. The MAM connections encode data by multiplying the values of the nodes they connect, adding the result to their (initially zeroed) weights. During recall, the connection multiplies the incoming value to its weight and sends the result to the connected (destination) node. With base-class "connection" methods source_pe and destin_pe providing access to the connected source and destination nodes, a class for MAM-specific connections may be defined as:

```
class mam_connection: public connection
{
public:
void encode() { weight() = weight() + source_pe().output * destin_pe().input; }
void recall() { destin_pe().receive_input_value ( weight() * source_pe().output ); }
};
```

Below, the "Connection_Set" template is used to create (in variable q) an empty set of such connections labeled "test connections" (the actual connections will be added later):

```
Connection_Set < mam_connection > q ( "test connections" );
```

The same template can be used to define new "connection_set" component types or sub-classes with customized functionality. For example to define a type (named "mam_connection_set"):

```
typedef Connection_Set < mam_connection >  mam_connection_set;
```

3.7 Defining a new neural net with bare Rcpp bindings

The obvious (but not particurlarly versatile) way to add a new NN model to R is to sub-class the "nn" class in C++, modify the new class to the desired model-specific functionality and expose it to R via Rcpp. This approach was taken for the predefined models, and results in the best run-time performing NN implementations. Since the NN class is the one actually presented to the user, its implementation may vary. For example, the code for the MAM NN (taken from file mam_nn.h in the package source with only headers and comments removed) is shown below. MAMs have a 2-layer, fully connected topology (each node in one layer has connections with all nodes in the other); the simplest version of MAM (implemented here) uses generic nodes and connections as defined in the previously:

```
class mam_nn : public nn
{
public:
mam_nn():nn("MAM Neural Network") {}

bool setup(int input_length,int output_length)
 {
 reset();
 add_layer( new Layer < pe > ( "Input layer" , input_length ) );
 add_connection_set( new mam_connection_set );
 add_layer( new Layer < pe > ( "Output layer", output_length ) );
 connect_consecutive_layers();
 return no_error();
 }
};
```

The `setup` method above adds the three components to the NN topology, i.e. two layers containing generic `"pe"` and a set of MAM-specific connections between them; layers must be setup before creating connections between their nodes, so `"nn"` method `connect_consecutive_layers` is called last, which (by default) populates the set with all possible connections between the nodes of the two layers, fully connecting them. For this very simple NN model no other code is required, except the "glue code" exposing this class (including the default `encode` and `recall` methods inherited from parent `"nn"` class) as a module to R using typical `Rcpp` methodology (see **?**).

4 Defining new models with the 'NN' R module

For a more versatile, R-based creation and control of the NNs, the `"NN"` module which `nnlib2Rcpp` package includes (as of version 0.1.4) can be used. This module maintains a neural network object with empty topology and provides an interface to build and manipulate it from R. Any of the component types (such as layers and connection sets) created for the predefined NNs in the package, or any other components defined by the user (as discussed later), can be added to its topology. The module can be used to create NN models with mixed component types, recursive or reverse-direction connections, unusual encoding or recalling sequences and, generally, aids experimentation by allowing significant control of the models from R. Once a `"NN"` module object is created, it can be manipulated by methods such as:

- `add_layer`: to create (and append to the topology) a new layer containing a given number of nodes; the type of this layer (and thus also of the nodes it contains) is defined by '`name`' parameter, with names available for several predefined layer types while additional names can be supported for user-defined components.
- `add_connection_set`: to create (and append to the topology) a new empty set of connections. It does not connect any layers (as they may not be setup yet) nor contains any connections. The type of this set (and thus also of the connections it will contain) is defined

by 'name' parameter, with names available for several predefined types of such sets; again additional names can be supported for user-defined ones.

- create_connections_in_sets: to fill connection sets with connections, fully connecting adjacent layers by adding all possible connections between their nodes.
- connect_layers_at: to insert a new empty set of connections (whose type is specified by 'name' parameter) between two layers and prepare it to connect them (no actual connections between layer nodes are created).
- fully_connect_layers_at: as above, but also fills the set with all possible connections between the nodes of the two layers.
- add_single_connection: to add a single connection between two nodes.
- remove_single_connection: to remove a single connection between two nodes.
- input_at: to input a data vector to a component in the topology.
- encode_at: to trigger the encoding operation of a component in the topology.
- recall_at: to trigger the recall (mapping, data retrieval) operation of a component in the topology.
- encode_all: to trigger the encoding operation of all components in the topology, in forward (first-to-last) or backward (last-to-first) order.
- recall_all: to trigger the recall (mapping, data retrieval) operation of all components in the topology, in forward or backward order.
- get_output_from: to get the current output of a component.
- get_input_at: to get the current input at a component.
- get_weights_at: to get the current weights of the connections in a connection_set component.
- print: to print the internal NN state, including the state of each component in topology.
- outline: to print a summary description of all components in topology.

For example, the "NN" module can be used to create a NN similar in functionality to the simple MAM described earlier; To define a NN that accepts input-output pairs of 3 element vectors, first a "NN" object is created (in variable m) and then two generic layers of 3 nodes are added to its topology:

```
m <- new( "NN" )       # create NN object in variable m
m$add_layer( "generic" , 3 )   # add a layer of 4 generic nodes
m$add_layer( "generic" , 3 )   # add another layer of 3 generic nodes
m$fully_connect_layers_at(1, 2, "MAM", 0, 0)  # add set of MAM connections btwn them
```

As specified by its parameters, the last step where fully_connect_layers_at is called effectively inserts a "mam_connection_set" component between positions 1 and 2 of the topology and connects all corresponding layer nodes by adding connections having weights initialized to 0. The final topology contains three components, a layer in topology position 1, a set of 9 MAM connections in 2, and another layer in position 3. This can be verified by using the outline method, which results in:

```
> m$outline()
------Network outline (BEGIN)--------
Neural Network (Ready - No Error)
Current NN topology:
@1 (c=0) component (id=67) is Layer : generic of size 3
@2 (c=1) component (id=69) is Connection Set : MAM (Fully Connected) 67-->68 of size 9
@3 (c=2) component (id=68) is Layer : generic of size 3
--------Network outline (END)--------
```

(note that component ids are assigned at run-time, and may differ). This NN stores input-output vector pairs. Two such vector pairs are encoded below (this simple MAM is not very powerful in mapping data, so ideal examples were selected):

```
m$input_at(1, c( 1, 0, 0 ) ) # input first vector at layer in position 1
m$input_at(3, c( 0, 0, 1 ) ) # input second vector at layer in position 3
m$encode_all( TRUE ) # encode, adjusting weights (fwd-direction)

m$input_at(1, c( 0, 0, 1 ) ) # input first vector at layer in position 1
m$input_at(3, c( 1, 0, 0 ) ) # input second vector at layer in position 3
m$encode_all( TRUE ) # encode, adjusting weights (fwd-direction)
```

To recall the second vector given the first:

```
m$input_at(1, c( 1, 0, 0 ) ) # input first vector at layer at position 1
m$recall_all( TRUE ) # recall (fwd-direction)
m$get_output_from( 3 )       # get second vector from layer at position 3
```

which returns:

```
[1] 0 0 1
```

and similarly,

```
m$input_at(1, c( 0, 0, 1 ) ) # input first vector at layer at position 1
m$recall_all( TRUE ) # recall (fwd-direction)
m$get_output_from( 3 )       # get second vector from layer at position 3
```

which returns:

```
[1] 1 0 0
```

In the next example, a back-propagation-based auto-encoding NN is created, with a the network topology composed mostly of predefined back-propagation (BP) components:

```
a <- new( "NN" )      # create a NN object in variable a
a$add_layer( "generic", 4 )   # 1. a layer of 4 generic nodes
a$add_connection_set( "BP" )  # 2. a set of BP connections
a$add_layer( "BP-hidden", 3 ) # 3. a layer of 3 BP pes
a$add_connection_set( "BP" )  # 4. another set of BP connections
a$add_layer( "BP-hidden", 2 ) # 5. another layer of 2 BP pes
a$add_connection_set( "BP" )  # 6. another set of BP connections
a$add_layer( "BP-hidden", 3 ) # 7. another layer of 3 BP pes
a$add_connection_set( "BP" )  # 8. another set of BP connections
a$add_layer( "BP-output", 4 ) # 9. a layer of 4 BP output pes
a$create_connections_in_sets ( 0, 1 ) # Populate sets with actual connections
```

This defines a network of 5 layers (sized 4, 3, 2, 3, and 4 nodes respectively) and sets of BP connections between them. The data encoding example shown below (for the scaled `iris_s` data defined earlier) presents each data vector to the first layer and performs a recall. It then presents the same vector to the last layer as the correct (desired) value, and performs encoding in all the components (from last to first), where discrepancies between recalled and desired output values are used to adjust connection weights, a functionality provided by the BP components used. The process is repeated for 1000 epochs:

```
for(e in 1:1000)        # for 1000 epochs
for(r in 1:nrow(iris_s))        # for each data case
 {
 a$input_at( 1, iris_s[ r , ] ) # present data at 1st layer
 a$recall_all( TRUE )   # recall (in fwd direction) entire topology
 a$input_at( 9, iris_s[ r , ] ) # present data at last layer
 a$encode_all ( FALSE ) # encode, adjusting weights (bwd-direction)
 }
```

Once the data is encoded (or auto-encoded since input and desired output are the same), new composite variables for the data can be collected at an intermediate layer. Below, the layer of 2 nodes (in position 5 of the topology) is used, so a set of 2 variables will be collected:

```
result <- NULL
for(r in 1:nrow(iris_s))        # for each data case
 {
 a$input_at( 1, iris_s[ r , ] ) # present data at 1st layer
 a$recall_all( TRUE )   # recall (in fwd direction) entire topology
 z <- a$get_output_from( 5 )    # collect output from layer at position 5
 result <- rbind( result, z )
 }
plot( result, pch = unclass( iris$Species ) )
```

The plot of the resulting output is shown in Figure 3, with corresponding `iris` species used for symbols.

4.1 Using additional NN components

In addition to predefined ones, new neural network components can be created and used by `"NN"` objects. Currently, the definition of new components must be done in C++, requires the `nnlib2Rcpp` package source code (which includes the `nnlib2` base classes) and the ability to compile it. In particular:

1. Any new component type definition can be added to a single header file called "`additional_parts.h`" (which is included in the package source). All new components to be employed by the `"NN"` R module must be defined in this file (or be accessible from `generate_custom_layer()` and `generate_custom_connection_set()` functions in the file).

2. The new `"pe"`, `"layer"`, `"connection"` or `"connection_set"` definitions must (at least loosely) comply to the `nnlib2` base class hierarchy and structure, and follow the related guidelines outlined earlier (minimal examples can be found in the "`additional_parts.h`" file itself).

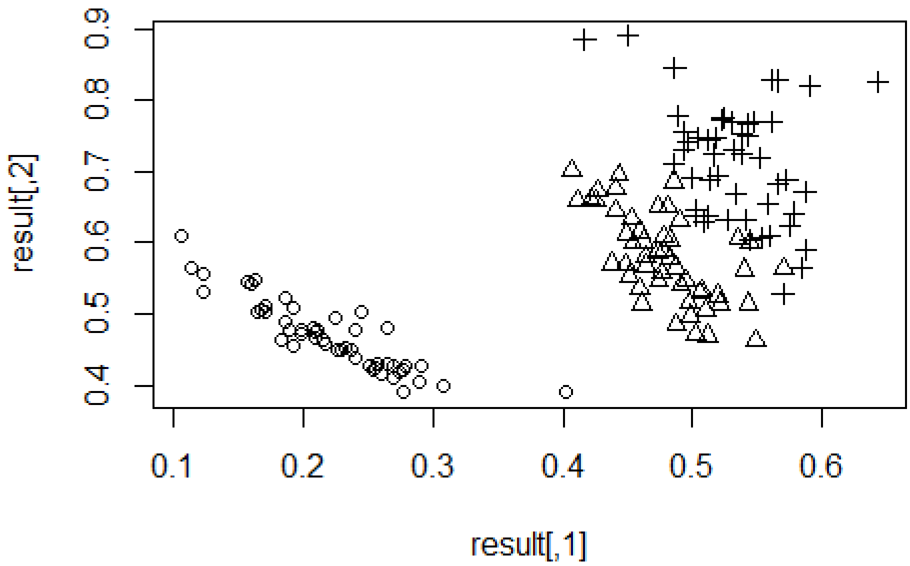

Figure 3: Results from a custom auto-encoding NN on Iris data.

3. A name must be reserved for the new "layer" and "connection_set" types or classes, 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 generated 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.

In an earlier example, a custom layer component type (called "JustAdd10_layer") was defined; it contains "JustAdd10_pe" nodes, which 'recall' the sum of their inputs plus 10. Should the definitions be placed in the "additional_parts.h" header file, the new layer type can be used in "NN" objects. The only other modification required, is to register a name for such layers, which can be done by adding the following line of code to function generate_custom_layer (also in "additional_parts.h"):

```
if(name == "JustAdd10")   return new JustAdd10_layer (name,size);
```

(note: this example may be already implemented in the package "additional_parts.h" file on the development version on GitHub i.e. at this link.

With these two steps completed and the modified package compiled, specifying the name "JustAdd10" when creating a layer in "NN" objects will result in a layer of "JustAdd10_pe" nodes:

```
x <- new( "NN" )       # create NN object in variable x
x$add_layer( "JustAdd10" , 3 )  # add layer of 3 'JustAdd10_pe' nodes
x$add_connection_set( "pass-through" ) # add set of pass-through connections
x$add_layer( "JustAdd10" , 1 )  # add layer of 1 'JustAdd10_pe' nodes
x$create_connections_in_sets ( 1, 1 )  # Create the connections
```

The network, with 3 nodes at its first layer, a set of 3 connections (than pass data unmodified) and a single node in the last layer, will effectively output sum(i + 10) + 10 for any 3 element input vector i. For input c(0, 10, 20) output is expected to be 70. To verify this:

```
x$input_at( 1, c( 0, 10, 20 ) )      # present data at 1st layer
x$recall_all( TRUE )         # recall (in fwd direction) entire topology
```

If output at 1st layer is checked, the initial values are increased by 10:

```
> x$get_output_from(1)
[1] 10 20 30
```

while at the last layer (3rd topology component), summation is done on these incoming values, and the result also increased by 10, producing the final output:

```
> x$get_output_from(3)
[1] 70
```

5 Summary

We introduced nnlib2Rcpp, an R package that targets small neural network applications, experimentation with such models, and related educational applications. Furthermore, we presented the underlying classes for defining new neural network components and models which can be used to extend the current list of supported models; Finally, we presented a module that can contain and utilize such neural network components, and can be used to define arbitrary network topologies and processing sequences.