NeuraNet

P. Baillehache

September 5, 2018

Contents

1 Definitions **3**

2 Interface **3**

3 Code **9**
3.1 pbmath.c 9
3.2 pbmath-inline.c 26

4 Makefile **31**

5 Unit tests **32**

6 Unit tests output **40**

7 Validation **43**
7.1 Iris data set 43
7.2 Abalone data set 53
7.3 Arrhythmia data set 63
7.4 Wisconsin Diagnostic Breast Cancer 72
7.5 MNIST 83
7.6 ORHD 94

8 nn2cloud **106**

Introduction

NeuraNet is a C library providing structures and functions to implement a neural network.

The neural network implemented in NeuraNet consists of a layer of input values, a layer of output values, a layer of hidden values, a set of generic base functions and a set of links. Each base function has 3 parameters (detailed below) and each links has 3 parameters: the base function index and the indices of input and output values. A NeuraNet is defined by the parameters' values of its generic base functions and links, and the number of input, output and hidden values.

The evaluation of the NeuraNet consists of taking each link, ordered on index of values, and apply the generic base function on the first value and store the result in the second value. If several links has the same second value index, the sum value of all these links is used. However if several links have same input and output values, the outputs of these links are multiplied instead of added (before being eventually added to other links having same output value but different input value).

The generic base functions is a linear function. However by using several links with same input and output values it is possible to simulate any polynomial function. Also, there is no concept of layer inside hidden values, but the input value index is constrained to be lower than the output one. So, the links can be arranged to form layers of subset of hidden values, while still allowing any other type of arrangement inside hidden values. Also, a link can be inactivated by setting its base function index to -1. Finally, the parameters of the base function and the hidden values are constrained to [-1.0,1.0].

NeuraNet provides functions to easily use the library GenAlg to search the values of base functions and links' parameters. An example is given in the unit tests (see below). It also provides functions to save and load the neural network (in JSON format).

NeuraNet has been validated on the Iris data set, the Abalone data set, the Arrhythmia data set, the Wisconsin Diagnostic Breast Cancer data set, the MNIST data set, the ORHD data set.

A utility tool allows to generate a file to be used as input of the CloudGraph tool to visualize the network of the NeuraNet.

It uses the `PBErr` library.

1 Definitions

The generic base function is defined as follow:

$$B(x) = [tan(1.57079 * b_0)(x + b_1) + b_2] \tag{1}$$

where $\{b_0, b_1, b_2\} \in [-1.0, 1.0]^3$ are the parameters of the base function and $x \in \mathbb{R}$ and $B(x) \in \mathbb{R}$.

2 Interface

```
// ============ NEURANET.H ================

#ifndef NEURANET_H
#define NEURANET_H

// ================= Include =================

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include "pberr.h"
#include "pbmath.h"
#include "gset.h"

// ----- NeuraNetBaseFun

// ================= Define ==================

#define NN_THETA 1.57079

// ================ Functions declaration ====================

// Generic base function for the NeuraNet
// 'param' is an array of 3 float all in [-1,1]
// 'x' is the input value
// NNBaseFun(param,x)=
// tan(param[0]*NN_THETA)*(x+param[1])+param[2]
#if BUILDMODE != 0
inline
#endif
float NNBaseFun(const float* const param, const float x);

// ----- NeuraNet

// ================= Define ==================

#define NN_NBPARAMBASE 3
#define NN_NBPARAMLINK 3

// ================= Data structure ==================

typedef struct NeuraNet {
  // Nb of input values
  const int _nbInputVal;
```

```
  // Nb of output values
  const int _nbOutputVal;
  // Nb max of hidden values
  const long _nbMaxHidVal;
  // Nb max of base functions
  const long _nbMaxBases;
  // Nb max of links
  const long _nbMaxLinks;
  // VecFloat describing the base functions
  // NN_NBPARAMBASE values per base function
  VecFloat* _bases;
  // VecShort describing the links
  // NN_NBPARAMLINK values per link (base id, input id, output id)
  // if (base id equals -1 the link is inactive)
  VecLong* _links;
  // Hidden values
  VecFloat* _hidVal;
  // Nb bases used for convolution
  const long _nbBasesConv;
  // Nb bases per cell used for convolution
  const long _nbBasesCellConv;
} NeuraNet;

// ================ Functions declaration ====================

// Create a new NeuraNet with 'nbInput' input values, 'nbOutput'
// output values, 'nbMaxHidden' hidden values, 'nbMaxBases' base
// functions, 'nbMaxLinks' links
NeuraNet* NeuraNetCreate(const int nbInput, const int nbOutput,
  const long nbMaxHidden, const long nbMaxBases, const long nbMaxLinks);

// Free the memory used by the NeuraNet 'that'
void NeuraNetFree(NeuraNet** that);

// Create a new NeuraNet with 'nbInput' input values, 'nbOutput'
// output values and a set of hidden layers described by
// 'hiddenLayers' as follow:
// The dimension of 'hiddenLayers' is the number of hidden layers
// and each component of 'hiddenLayers' is the number of hidden value
// in the corresponding hidden layer
// For example, <3,4> means 2 hidden layers, the first one with 3
// hidden values and the second one with 4 hidden values
// If 'hiddenValues' is null it means there is no hidden layers
// Then, links are automatically added between each input values
// toward each hidden values in the first hidden layer, then from each
// hidden values of the first hidden layer to each hidden value of the
// 2nd hidden layer and so on until each values of the output
NeuraNet* NeuraNetCreateFullyConnected(const int nbIn, const int nbOut,
  const VecLong* const hiddenLayers);

// Create a NeuraNet using convolution
// The input's dimension is equal to the dimension of 'dimIn', for
// example if dimIn==<2,3> the input is a 2D array of width 2 and
// height 3, input values are expected ordered by lines
// The NeuraNet has 'nbOutput' outputs
// The dimension of each convolution cells is 'dimCell'
// The maximum number of convolution (in depth) is 'depthConv'
// Each convolution layer has 'thickConv' convolutions in parallel
// The outputs are fully connected to the last layer of convolution cells
// For example, if the input is a 2D array of 4 cols and 3 rows, 2
// outputs, 2x2 convolution cell, convolution depth of 2, and
// convolution thickness of 2:
```

```
// index of values from input layer to ouput layer
// 00,01,02,03,
// 04,05,06,07,
// 08,09,10,11
//
// 12,13,14,  18,19,20,
// 15,16,17,  21,22,23,
//
// 24,25  26,27
//
// 28,29
//
// nbInput: 12
// nbOutput: 2
// nbHidden: 16
// nbMaxBases: 24
// nbMaxLinks: 72
// links:
//    0,0,12, 4,0,18, 1,1,12, 0,1,13, 5,1,18, 4,1,19, 1,2,13, 0,2,14,
//    5,2,19, 4,2,20, 1,3,14, 5,3,20, 2,4,12, 0,4,15, 6,4,18, 4,4,21,
//    3,5,12, 2,5,13, 1,5,15, 0,5,16, 7,5,18, 6,5,19, 5,5,21, 4,5,22,
//    3,6,13, 2,6,14, 1,6,16, 0,6,17, 7,6,19, 6,6,20, 5,6,22, 4,6,23,
//    3,7,14, 1,7,17, 7,7,20, 5,7,23, 2,8,15, 6,8,21, 3,9,15, 2,9,16,
//    7,9,21, 6,9,22, 3,10,16, 2,10,17, 7,10,22, 6,10,23, 3,11,17,
//    7,11,23, 8,12,24, 9,13,24, 8,13,25, 9,14,25, 10,15,24, 11,16,24,
//    10,16,25, 11,17,25, 12,18,26, 13,19,26, 12,19,27, 13,20,27,
//    14,21,26, 15,22,26, 14,22,27, 15,23,27, 16,24,28, 17,24,29,
//    18,25,28, 19,25,29, 20,26,28, 21,26,29, 22,27,28, 23,27,29
NeuraNet* NeuraNetCreateConvolution(const VecShort* const dimIn,
  const int nbOutput, const VecShort* const dimCell,
  const int depthConv, const int thickConv);

// Get the nb of input values of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
int NNGetNbInput(const NeuraNet* const that);

// Get the nb of output values of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
int NNGetNbOutput(const NeuraNet* const that);

// Get the nb max of hidden values of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
long NNGetNbMaxHidden(const NeuraNet* const that);

// Get the nb max of base functions of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
long NNGetNbMaxBases(const NeuraNet* const that);

// Get the nb of base functions for convolution of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
long NNGetNbBasesConv(const NeuraNet* const that);
```

```
// Get the nb of base functions per cell for convolution of
// the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
long NNGetNbBasesCellConv(const NeuraNet* const that);

// Get the nb max of links of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
long NNGetNbMaxLinks(const NeuraNet* const that);

// Get the parameters of the base functions of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
const VecFloat* NNBases(const NeuraNet* const that);

// Get the links description of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
const VecLong* NNLinks(const NeuraNet* const that);

// Get the hidden values of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
const VecFloat* NNHiddenValues(const NeuraNet* const that);

// Get the 'iVal'-th hidden value of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
float NNGetHiddenValue(const NeuraNet* const that, const long iVal);

// Set the parameters of the base functions of the NeuraNet 'that' to
// a copy of 'bases'
// 'bases' must be of dimension that->nbMaxBases * NN_NBPARAMBASE
//  each base is defined as param[3] in [-1,1]
// tan(param[0]*NN_THETA)*(x+param[1])+param[2]
#if BUILDMODE != 0
inline
#endif
void NNSetBases(NeuraNet* const that, const VecFloat* const bases);

// Set the 'iBase'-th parameter of the base functions of the NeuraNet
// 'that' to 'base'
#if BUILDMODE != 0
inline
#endif
void NNBasesSet(NeuraNet* const that, const long iBase, const float base);

// Set the links description of the NeuraNet 'that' to a copy of 'links'
// Links with a base function equals to -1 are ignored
// If the input id is higher than the output id they are swap
// The links description in the NeuraNet are ordered in increasing
// value of input id and output id, but 'links' doesn't have to be
// sorted
// Each link is defined by (base index, input index, output index)
// If base index equals -1 it means the link is inactive
```

```
void NNSetLinks(NeuraNet* const that, VecLong* const links);

// Calculate the output values for the input values 'input' for the
// NeuraNet 'that' and memorize the result in 'output'
// input values in [-1,1] and output values in [-1,1]
// All values of 'output' are set to 0.0 before evaluating
// Links which refer to values out of bounds of 'input' or 'output'
// are ignored
void NNEval(const NeuraNet* const that, const VecFloat* const input, VecFloat* const output);

// Function which return the JSON encoding of 'that'
JSONNode* NNEncodeAsJSON(const NeuraNet* const that);

// Function which decode from JSON encoding 'json' to 'that'
bool NNDecodeAsJSON(NeuraNet** that, const JSONNode* const json);

// Save the NeuraNet 'that' to the stream 'stream'
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true if the NeuraNet could be saved, false else
bool NNSave(const NeuraNet* const that, FILE* const stream, const bool compact);

// Load the NeuraNet 'that' from the stream 'stream'
// If 'that' is not null the memory is first freed
// Return true if the NeuraNet could be loaded, false else
bool NNLoad(NeuraNet** that, FILE* const stream);

// Print the NeuraNet 'that' to the stream 'stream'
void NNPrintln(const NeuraNet* const that, FILE* const stream);

// Get the number of active links in the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
long NNGetNbActiveLinks(const NeuraNet* const that);

// Save the links of the NeuraNet 'that' into the file at 'url' in a
// format readable by CloudGraph
// Return true if we could save, else false
bool NNSaveLinkAsCloudGraph(const NeuraNet* const that, const char* url);

// Get the Simpson's diversity index of the hidden values of the
// NeuraNet 'that'
// Return value in [0.0, 1.0], 0.0 means no diversity, 1.0 means max
// diversity
float NNGetHiddenValSimpsonDiv(const NeuraNet* const that);

// Prune the NeuraNet 'that' by removing the useless links (those with
// no influence on outputs
void NNPrune(const NeuraNet* const that);

// ================= Interface with library GenAlg ==================
// To use the following functions the user must include the header
// 'genalg.h' before the header 'neuranet.h'

#ifdef GENALG_H

// Get the length of the adn of float values to be used in the GenAlg
// library for the NeuraNet 'that'
static long NNGetGAAdnFloatLength(const NeuraNet* const that)
  __attribute__((unused));
static long NNGetGAAdnFloatLength(const NeuraNet* const that) {
```

```
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
#endif
  return NNGetNbMaxBases(that) * NN_NBPARAMBASE;
}

// Get the length of the adn of int values to be used in the GenAlg
// library for the NeuraNet 'that'
static long NNGetGAAdnIntLength(const NeuraNet* const that)
  __attribute__((unused));
static long NNGetGAAdnIntLength(const NeuraNet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
#endif
  return NNGetNbMaxLinks(that) * NN_NBPARAMLINK;
}

// Set the bounds of the GenAlg 'ga' to be used for bases parameters of
// the NeuraNet 'that'
static void NNSetGABoundsBases(const NeuraNet* const that, GenAlg* const ga)
  __attribute__((unused));
static void NNSetGABoundsBases(const NeuraNet* const that, GenAlg* const ga) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
  if (ga == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'ga' is null");
    PBErrCatch(NeuraNetErr);
  }
  if (GAGetLengthAdnFloat(ga) != NNGetGAAdnFloatLength(that)) {
    NeuraNetErr->_type = PBErrTypeInvalidArg;
    sprintf(NeuraNetErr->_msg, "'ga' 's float genes dimension doesn't\
 matches 'that' 's max nb of bases (%ld==%ld)",
      GAGetLengthAdnFloat(ga), NNGetGAAdnFloatLength(that));
    PBErrCatch(NeuraNetErr);
  }
#endif
  // Declare a vector to memorize the bounds
  VecFloat2D bounds = VecFloatCreateStatic2D();
  // Init the bounds
  VecSet(&bounds, 0, -1.0); VecSet(&bounds, 1, 1.0);
  // For each gene
  for (long iGene = NNGetGAAdnFloatLength(that); iGene--;)
    // Set the bounds
    GASetBoundsAdnFloat(ga, iGene, &bounds);
}

// Set the bounds of the GenAlg 'ga' to be used for links description of
// the NeuraNet 'that'
static void NNSetGABoundsLinks(const NeuraNet* const that, GenAlg* const ga)
```

```
  __attribute__((unused));
static void NNSetGABoundsLinks(const NeuraNet* const that, GenAlg* const ga) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
  if (ga == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'ga' is null");
    PBErrCatch(NeuraNetErr);
  }
  if (GAGetLengthAdnInt(ga) != NNGetGAAdnIntLength(that)) {
    NeuraNetErr->_type = PBErrTypeInvalidArg;
    sprintf(NeuraNetErr->_msg, "'ga' 's int genes dimension doesn't\
 matches 'that' 's max nb of links (%ld==%ld)",
      GAGetLengthAdnInt(ga), NNGetGAAdnIntLength(that));
    PBErrCatch(NeuraNetErr);
  }
#endif
  // Declare a vector to memorize the bounds
  VecLong2D bounds = VecLongCreateStatic2D();
  // For each gene
  for (long iGene = 0; iGene < NNGetGAAdnIntLength(that);
    iGene += NN_NBPARAMLINK) {
    // Set the bounds for base id
    VecSet(&bounds, 0, -1);
    VecSet(&bounds, 1, NNGetNbMaxBases(that) - 1);
    GASetBoundsAdnInt(ga, iGene, &bounds);
    // Set the bounds for input value
    VecSet(&bounds, 0, 0);
    VecSet(&bounds, 1, NNGetNbInput(that) + NNGetNbMaxHidden(that) - 1);
    GASetBoundsAdnInt(ga, iGene + 1, &bounds);
    // Set the bounds for input value
    VecSet(&bounds, 0, NNGetNbInput(that));
    VecSet(&bounds, 1, NNGetNbInput(that) + NNGetNbMaxHidden(that) +
      NNGetNbOutput(that) - 1);
    GASetBoundsAdnInt(ga, iGene + 2, &bounds);
  }
}

#endif

// ================ Inliner ====================

#if BUILDMODE != 0
#include "neuranet-inline.c"
#endif


#endif
```

3 Code

3.1 pbmath.c

```
// ============ NEURANET.C ================
```

```
// ================= Include =================

#include "neuranet.h"
#if BUILDMODE == 0
#include "neuranet-inline.c"
#endif

// ----- NeuraNet

// ================ Functions implementation ====================

// Create a new NeuraNet with 'nbInput' input values, 'nbOutput'
// output values, 'nbMaxHidden' hidden values, 'nbMaxBases' base
// functions, 'nbMaxLinks' links
NeuraNet* NeuraNetCreate(const int nbInput, const int nbOutput,
  const long nbMaxHidden, const long nbMaxBases, const long nbMaxLinks) {
#if BUILDMODE == 0
  if (nbInput <= 0) {
    NeuraNetErr->_type = PBErrTypeInvalidArg;
    sprintf(NeuraNetErr->_msg, "'nbInput' is invalid (0<%d)", nbInput);
    PBErrCatch(NeuraNetErr);
  }
  if (nbOutput <= 0) {
    NeuraNetErr->_type = PBErrTypeInvalidArg;
    sprintf(NeuraNetErr->_msg, "'nbOutput' is invalid (0<%d)", nbOutput);
    PBErrCatch(NeuraNetErr);
  }
  if (nbMaxHidden < 0) {
    NeuraNetErr->_type = PBErrTypeInvalidArg;
    sprintf(NeuraNetErr->_msg, "'nbMaxHidden' is invalid (0<=%ld)",
      nbMaxHidden);
    PBErrCatch(NeuraNetErr);
  }
  if (nbMaxBases <= 0) {
    NeuraNetErr->_type = PBErrTypeInvalidArg;
    sprintf(NeuraNetErr->_msg, "'nbMaxBases' is invalid (0<%ld)",
      nbMaxBases);
    PBErrCatch(NeuraNetErr);
  }
  if (nbMaxLinks <= 0) {
    NeuraNetErr->_type = PBErrTypeInvalidArg;
    sprintf(NeuraNetErr->_msg, "'nbMaxLinks' is invalid (0<%ld)",
      nbMaxLinks);
    PBErrCatch(NeuraNetErr);
  }
#endif
  // Declare the new NeuraNet
  NeuraNet* that = PBErrMalloc(NeuraNetErr, sizeof(NeuraNet));
  // Set properties
  *(int*)&(that->_nbInputVal) = nbInput;
  *(int*)&(that->_nbOutputVal) = nbOutput;
  *(long*)&(that->_nbMaxHidVal) = nbMaxHidden;
  *(long*)&(that->_nbMaxBases) = nbMaxBases;
  *(long*)&(that->_nbMaxLinks) = nbMaxLinks;
  *(long*)&(that->_nbBasesConv) = 0;
  *(long*)&(that->_nbBasesCellConv) = 0;
  that->_bases = VecFloatCreate(nbMaxBases * NN_NBPARAMBASE);
  that->_links = VecLongCreate(nbMaxLinks * NN_NBPARAMLINK);
  if (nbMaxHidden > 0)
    that->_hidVal = VecFloatCreate(nbMaxHidden);
  else
```

```
    that->_hidVal = NULL;
  // Return the new NeuraNet
  return that;
}

// Free the memory used by the NeuraNet 'that'
void NeuraNetFree(NeuraNet** that) {
  // Check argument
  if (that == NULL || *that == NULL)
    // Nothing to do
    return;
  // Free memory
  VecFree(&((*that)->_bases));
  VecFree(&((*that)->_links));
  VecFree(&((*that)->_hidVal));
  free(*that);
  *that = NULL;
}

// Create a new NeuraNet with 'nbIn' innput values, 'nbOut'
// output values and a set of hidden layers described by
// 'hiddenLayers' as follow:
// The dimension of 'hiddenLayers' is the number of hidden layers
// and each component of 'hiddenLayers' is the number of hidden value
// in the corresponding hidden layer
// For example, <3,4> means 2 hidden layers, the first one with 3
// hidden values and the second one with 4 hidden values
// If 'hiddenValues' is null it means there is no hidden layers
// Then, links are automatically added between each input values
// toward each hidden values in the first hidden layer, then from each
// hidden values of the first hidden layer to each hidden value of the
// 2nd hidden layer and so on until each values of the output
NeuraNet* NeuraNetCreateFullyConnected(const int nbIn, const int nbOut,
  const VecLong* const hiddenLayers) {
#if BUILDMODE == 0
  if (nbIn <= 0) {
    NeuraNetErr->_type = PBErrTypeInvalidArg;
    sprintf(NeuraNetErr->_msg, "'nbInput' is invalid (0<%d)", nbIn);
    PBErrCatch(NeuraNetErr);
  }
  if (nbOut <= 0) {
    NeuraNetErr->_type = PBErrTypeInvalidArg;
    sprintf(NeuraNetErr->_msg, "'nbOutput' is invalid (0<%d)", nbOut);
    PBErrCatch(NeuraNetErr);
  }
#endif
  // Declare variable to memorize the number of links, bases
  // and hidden values
  long nbHiddenVal = 0;
  long nbBases = 0;
  long nbLinks = 0;
  long nbHiddenLayer = 0;
  // If there are hidden layers
  if (hiddenLayers != NULL) {
    // Get the number of hidden layers
    nbHiddenLayer = VecGetDim(hiddenLayers);
    // Declare two variables for computation
    long nIn = nbIn;
    long nOut = 0;
    // Calculate the nb of links and hidden values
    for (long iLayer = 0; iLayer < nbHiddenLayer; ++iLayer) {
      nOut = VecGet(hiddenLayers, iLayer);
```

```
      nbHiddenVal += nOut;
      nbLinks += nIn * nOut;
      nIn = nOut;
    }
    nbLinks += nIn * nbOut;
  // Else, there is no hidden layers
  } else {
    // Set the number of links
    nbLinks = nbIn * nbOut;
  }
  // There is one base function per link
  nbBases = nbLinks;
  // Create the NeuraNet
  NeuraNet* nn =
    NeuraNetCreate(nbIn, nbOut, nbHiddenVal, nbBases, nbLinks);
  // Declare a variable to memorize the index of the link
  long iLink = 0;
  // Declare variables for computation
  long shiftIn = 0;
  long shiftOut = nbIn;
  long nIn = nbIn;
  long nOut = 0;
  // Loop on hidden layers
  for (long iLayer = 0; iLayer <= nbHiddenLayer; ++iLayer) {
    // Init the links
    if (iLayer < nbHiddenLayer)
      nOut = VecGet(hiddenLayers, iLayer);
    else
      nOut = nbOut;
    for (long iIn = 0; iIn < nIn; ++iIn) {
      for (long iOut = 0; iOut < nOut; ++iOut) {
        long jLink = NN_NBPARAMLINK * iLink;
        VecSet(nn->_links, jLink, iLink);
        VecSet(nn->_links, jLink + 1, iIn + shiftIn);
        VecSet(nn->_links, jLink + 2, iOut + shiftOut);
        ++iLink;
      }
    }
    shiftIn = shiftOut;
    shiftOut += nOut;
    nIn = nOut;
  }
  // Return the new NeuraNet
  return nn;
}

// Create a NeuraNet using convolution
// The input's dimension is equal to the dimension of 'dimIn', for
// example if dimIn==<2,3> the input is a 2D array of width 2 and
// height 3, input values are expected ordered by lines
// The NeuraNet has 'nbOutput' outputs
// The dimension of each convolution cells is 'dimCell'
// The maximum number of convolution (in depth) is 'depthConv'
// Each convolution layer has 'thickConv' convolutions in parallel
// The outputs are fully connected to the last layer of convolution cells
// For example, if the input is a 2D array of 4 cols and 3 rows, 2
// outputs, 2x2 convolution cell, convolution depth of 2, and
// convolution thickness of 2:
// index of values from input layer to ouput layer
// 00,01,02,03,
// 04,05,06,07,
// 08,09,10,11
```

```
//
// 12,13,14,  18,19,20,
// 15,16,17,  21,22,23,
//
// 24,25  26,27
//
// 28,29
//
// nbInput: 12
// nbOutput: 2
// nbHidden: 16
// nbMaxBases: 24
// nbMaxLinks: 72
// links:
//    0,0,12, 4,0,18, 1,1,12, 0,1,13, 5,1,18, 4,1,19, 1,2,13, 0,2,14,
//    5,2,19, 4,2,20, 1,3,14, 5,3,20, 2,4,12, 0,4,15, 6,4,18, 4,4,21,
//    3,5,12, 2,5,13, 1,5,15, 0,5,16, 7,5,18, 6,5,19, 5,5,21, 4,5,22,
//    3,6,13, 2,6,14, 1,6,16, 0,6,17, 7,6,19, 6,6,20, 5,6,22, 4,6,23,
//    3,7,14, 1,7,17, 7,7,20, 5,7,23, 2,8,15, 6,8,21, 3,9,15, 2,9,16,
//    7,9,21, 6,9,22, 3,10,16, 2,10,17, 7,10,22, 6,10,23, 3,11,17,
//    7,11,23, 8,12,24, 9,13,24, 8,13,25, 9,14,25, 10,15,24, 11,16,24,
//    10,16,25, 11,17,25, 12,18,26, 13,19,26, 12,19,27, 13,20,27,
//    14,21,26, 15,22,26, 14,22,27, 15,23,27, 16,24,28, 17,24,29,
//    18,25,28, 19,25,29, 20,26,28, 21,26,29, 22,27,28, 23,27,29
NeuraNet* NeuraNetCreateConvolution(const VecShort* const dimIn,
  const int nbOutput, const VecShort* const dimCell,
  const int depthConv, const int thickConv) {
#if BUILDMODE == 0
  if (dimIn == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'dimIn' is null");
    PBErrCatch(NeuraNetErr);
  }
  for (long iDim = VecGetDim(dimIn); iDim--;)
    if (VecGet(dimIn, iDim) <= 0) {
      NeuraNetErr->_type = PBErrTypeInvalidArg;
      sprintf(NeuraNetErr->_msg, "'dimIn' %ldth dim is invalid (%d>0)",
        iDim, VecGet(dimIn, iDim));
      PBErrCatch(NeuraNetErr);
    }
  if (nbOutput <= 0) {
    NeuraNetErr->_type = PBErrTypeInvalidArg;
    sprintf(NeuraNetErr->_msg, "'nbOutput' is invalid (0<%d)", nbOutput);
    PBErrCatch(NeuraNetErr);
  }
  if (dimCell == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'dimCell' is null");
    PBErrCatch(NeuraNetErr);
  }
  if (VecGetDim(dimCell) != VecGetDim(dimIn)) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'dimCell' 's dim is invalid (%ld==%ld)",
      VecGetDim(dimCell), VecGetDim(dimIn));
    PBErrCatch(NeuraNetErr);
  }
  for (long iDim = VecGetDim(dimCell); iDim--;)
    if (VecGet(dimCell, iDim) <= 0) {
      NeuraNetErr->_type = PBErrTypeInvalidArg;
      sprintf(NeuraNetErr->_msg, "'dimCell' %ldth dim is invalid (%d>0)",
        iDim, VecGet(dimCell, iDim));
      PBErrCatch(NeuraNetErr);
```

```
    }
  if (depthConv < 0) {
    NeuraNetErr->_type = PBErrTypeInvalidArg;
    sprintf(NeuraNetErr->_msg, "'depthConv' is invalid (0<=%d)",
      depthConv);
    PBErrCatch(NeuraNetErr);
  }
#endif
  // Declare a variable to memorize the nb of input, hidden values,
  // bases and links
  long nbIn = 0;
  long nbHiddenVal = 0;
  long nbBases = 0;
  long nbLinks = 0;
  // Calculate the number of inputs
  nbIn = 1;
  for (long iDim = VecGetDim(dimIn); iDim--;)
    nbIn *= VecGet(dimIn, iDim);
  // Calculate the number of bases, links and hidden values
  // Declare a variable to memorize the number of links per cell
  long nbLinkPerCell = 1;
  for (long iDim = VecGetDim(dimCell); iDim--;)
    nbLinkPerCell *= VecGet(dimCell, iDim);
  // Declare a variable to memorize the position of the convolution
  // cell in the current convolution layer
  VecShort* pos = VecShortCreate(VecGetDim(dimIn));
  // Declare a variable to memorize the position in the convolution cell
  VecShort* posCell = VecShortCreate(VecGetDim(dimIn));
  // Declare variables to memorize the dimension and size of the input
  // layer at current convolution level
  VecShort* curDimIn = VecClone(dimIn);
  long sizeLayerIn = 1;
  for (long iDim = VecGetDim(curDimIn); iDim--;)
    sizeLayerIn *= VecGet(curDimIn, iDim);
  // Declare variables to memorize the dimension and size of the
  // output layer at current convolution level
  VecShort* curDimOut = VecClone(curDimIn);
  long sizeLayerOut = 1;
  for (long iDim = VecGetDim(curDimOut); iDim--;) {
    VecSetAdd(curDimOut, iDim, -1 * VecGet(dimCell, iDim) + 1);
    sizeLayerOut *= VecGet(curDimOut, iDim);
  }
  // Loop on convolution levels
  for (long iConv = 0; iConv < depthConv; ++iConv) {
    // Update the number of bases
    nbBases += nbLinkPerCell;
    // Update the number of hidden values
    nbHiddenVal += sizeLayerOut;
    // Update the number of links
    nbLinks += sizeLayerOut * nbLinkPerCell;
    // If we are not a the last convolution level
    if (iConv < depthConv - 1) {
      // Update input and output dimensions at next convolution level
      VecCopy(curDimIn, curDimOut);
      sizeLayerIn = sizeLayerOut;
      sizeLayerOut = 1;
      for (long iDim = VecGetDim(curDimOut); iDim--;) {
        VecSetAdd(curDimOut, iDim, -1 * VecGet(dimCell, iDim) + 1);
        sizeLayerOut *= VecGet(curDimOut, iDim);
      }
    }
  }
```

```
// Multiply by the number of convolution in parallel
nbHiddenVal *= thickConv;
nbBases *= thickConv;
nbLinks *= thickConv;
long nbBasesConv = nbBases;
// Add the links and bases for the fully connected layer toward output
nbBases += sizeLayerOut * thickConv * nbOutput;
nbLinks += sizeLayerOut * thickConv * nbOutput;
// Create the NeuraNet
NeuraNet* nn =
  NeuraNetCreate(nbIn, nbOutput, nbHiddenVal, nbBases, nbLinks);
*(long*)&(nn->_nbBasesConv) = nbBasesConv;
*(long*)&(nn->_nbBasesCellConv) = nbLinkPerCell;
// Declare variables to create the links
VecLong* links = VecLongCreate(nbLinks * NN_NBPARAMLINK);
// Declare a variable to memorize the index of the currenlty
// created link
long iLink = 0;
// Reset the dimension and size of the input layer at current
// convolution level
VecCopy(curDimIn, dimIn);
sizeLayerIn = 1;
for (long iDim = VecGetDim(curDimIn); iDim--;) {
  sizeLayerIn *= VecGet(curDimIn, iDim);
}
// Reset the dimension and size of the output layer at current
// convolution level
VecCopy(curDimOut, dimIn);
sizeLayerOut = 1;
for (long iDim = VecGetDim(curDimOut); iDim--;) {
  VecSetAdd(curDimOut, iDim, -1 * VecGet(dimCell, iDim) + 1);
  sizeLayerOut *= VecGet(curDimOut, iDim);
}
// Declare variables to memorize the index of the beginning of the
// input and output layer and base functions at current convolution
// level
long* iStartBase = PBErrMalloc(NeuraNetErr, sizeof(long) * thickConv);
long* iStartLayerIn = PBErrMalloc(NeuraNetErr,
  sizeof(long) * thickConv);
long* iStartLayerOut = PBErrMalloc(NeuraNetErr,
  sizeof(long) * thickConv);
for (long iThick = 0; iThick < thickConv; ++iThick) {
  iStartLayerIn[iThick] = 0;
  iStartLayerOut[iThick] = sizeLayerIn + iThick * sizeLayerOut;
  iStartBase[iThick] = iThick * nbLinkPerCell;
}
// Loop on convolution levels
for (long iConv = 0; iConv < depthConv; ++iConv) {
  // Reset the position of the convolution cell in the input layer
  VecSetNull(pos);
  // Loop on position of the convolution cell at the current
  // convolution levels
  do {
    do {
      // Loop on convolution in parallel
      for (long iThick = 0; iThick < thickConv; ++iThick) {
        // Declare a variable to memorize the index of the input of the
        // current link
        long iInput = 0;
        for (long iDim = VecGetDim(curDimIn); iDim--;) {
          iInput *= VecGet(curDimIn, iDim);
          iInput += VecGet(posCell, iDim) + VecGet(pos, iDim);
```

```
        }
        iInput += iStartLayerIn[iThick];
        // Declare a variable to memorize the index of the output of
        // the current link
        long iOutput = 0;
        for (long iDim = VecGetDim(curDimOut); iDim--;) {
          iOutput *= VecGet(curDimOut, iDim);
          iOutput += VecGet(pos, iDim);
        }
        iOutput += iStartLayerOut[iThick];
        // Declare a variable to memorize the index of the base of the
        // current link
        long iBase = 0;
        for (long iDim = VecGetDim(posCell); iDim--;) {
          iBase *= VecGet(dimCell, iDim);
          iBase += VecGet(posCell, iDim);
        }
        iBase += iStartBase[iThick];
        // Set the current link's parameters
        VecSet(links, iLink * NN_NBPARAMLINK, iBase);
        VecSet(links, iLink * NN_NBPARAMLINK + 1, iInput);
        VecSet(links, iLink * NN_NBPARAMLINK + 2, iOutput);
        // Increment the index of the current link
        ++iLink;
      }
    } while (VecPStep(posCell, dimCell));
  } while (VecPStep(pos, curDimOut));
  // If we are not at the last convolution level
  if (iConv < depthConv - 1) {
    // Update input and output dimensions at next convolution level
    VecCopy(curDimIn, curDimOut);
    sizeLayerIn = sizeLayerOut;
    sizeLayerOut = 1;
    for (long iDim = VecGetDim(curDimOut); iDim--;) {
      VecSetAdd(curDimOut, iDim, -1 * VecGet(dimCell, iDim) + 1);
      sizeLayerOut *= VecGet(curDimOut, iDim);
    }
  }
  // Update the start index of input and output layers and bases
  // for each convolution in parallel
  for (long iThick = 0; iThick < thickConv; ++iThick) {
    iStartLayerIn[iThick] = iStartLayerOut[iThick];
    iStartLayerOut[iThick] = iStartLayerIn[0] +
      thickConv * sizeLayerIn + iThick * sizeLayerOut;
    iStartBase[iThick] =
      ((iConv + 1) * thickConv + iThick) * nbLinkPerCell;
  }
}
// Set the links of the last fully connected layer between last
// convolution and NeuraNet output
// Declare a variable to remember the index of the base
long iBase = iStartBase[0];
// Loop on the last output of convolution layer
for (long iLayerOut = 0; iLayerOut < sizeLayerOut; ++iLayerOut) {
  // Loop on parallel convolution
  for (long iThick = 0; iThick < thickConv; ++iThick) {
    // Loop on output of the NeuraNet
    for (long iOut = 0; iOut < nbOutput; ++iOut) {
      // Declare a variable to memorize the index of the input of
      // the link
      long iInput = iStartLayerIn[0] +
        iLayerOut * thickConv + iThick;
```

```
        // Declare a variable to memorize the index of the output of
        // the link
        long iOutput = iOut + nbIn + nbHiddenVal;
        // Set the link's parameters
        VecSet(links, iLink * NN_NBPARAMLINK, iBase);
        VecSet(links, iLink * NN_NBPARAMLINK + 1, iInput);
        VecSet(links, iLink * NN_NBPARAMLINK + 2, iOutput);
        // Increment the link index
        ++iLink;
        // Increment the base function
        ++iBase;
      }
    }
  }
  // Set up the links
  NNSetLinks(nn, links);
  // Free memory
  VecFree(&links);
  VecFree(&pos);
  VecFree(&posCell);
  VecFree(&curDimIn);
  VecFree(&curDimOut);
  free(iStartBase);
  free(iStartLayerIn);
  free(iStartLayerOut);
  // Return the new NeuraNet
  return nn;
}

// Calculate the output values for the input values 'input' for the
// NeuraNet 'that' and memorize the result in 'output'
// input values in [-1,1] and output values in [-1,1]
// All values of 'output' are set to 0.0 before evaluating
// Links which refer to values out of bounds of 'input' or 'output'
// are ignored
void NNEval(const NeuraNet* const that, const VecFloat* const input, VecFloat* const output) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
  if (input == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'input' is null");
    PBErrCatch(NeuraNetErr);
  }
  if (output == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'output' is null");
    PBErrCatch(NeuraNetErr);
  }
  if (VecGetDim(input) != that->_nbInputVal) {
    NeuraNetErr->_type = PBErrTypeInvalidArg;
    sprintf(NeuraNetErr->_msg,
      "'input' 's dimension is invalid (%ld!=%d)",
      VecGetDim(input), that->_nbInputVal);
    PBErrCatch(NeuraNetErr);
  }
  if (VecGetDim(output) != that->_nbOutputVal) {
    NeuraNetErr->_type = PBErrTypeInvalidArg;
    sprintf(NeuraNetErr->_msg,
```

```
      "'output' 's dimension is invalid (%ld!=%d)",
      VecGetDim(output), that->_nbOutputVal);
    PBErrCatch(NeuraNetErr);
  }
#endif
  // Reset the hidden values and output
  if (NNGetNbMaxHidden(that) > 0)
    VecSetNull(that->_hidVal);
  VecSetNull(output);
  // If there are links in the network
  if (VecGet(that->_links, 0) != -1) {
    // Declare two variables to memorize the starting index of hidden
    // values and output values in the link definition
    long startHid = NNGetNbInput(that);
    long startOut = NNGetNbMaxHidden(that) + NNGetNbInput(that);
    // Declare a variable to memorize the previous link
    long prevLink[2] = {-1, -1};
    // Declare a variable to memorize the previous output value
    float prevOut = 1.0;
    // Loop on links
    long iLink = 0;
    while (iLink < NNGetNbMaxLinks(that) &&
      VecGet(that->_links, NN_NBPARAMLINK * iLink) != -1) {
      // Declare a variable for optimization
      long jLink = NN_NBPARAMLINK * iLink;
      // If this link has different input or output than previous link
      // and we are not on the first link
      if (iLink != 0 &&
        (VecGet(that->_links, jLink + 1) != prevLink[0] ||
        VecGet(that->_links, jLink + 2) != prevLink[1])) {
        // Add the previous output value to the output of the previous
        // link
        if (prevLink[1] < startOut) {
          long iVal = prevLink[1] - startHid;
          float nVal = MIN(1.0, MAX(-1.0, VecGet(that->_hidVal, iVal) + prevOut));
          VecSet(that->_hidVal, iVal, nVal);
        } else {
          long iVal = prevLink[1] - startOut;
          float nVal = VecGet(output, iVal) + prevOut;
          VecSet(output, iVal, nVal);
        }
        // Reset the previous output
        prevOut = 1.0;
      }
      // Update the previous link
      prevLink[0] = VecGet(that->_links, jLink + 1);
      prevLink[1] = VecGet(that->_links, jLink + 2);
      // Multiply the previous output by the evaluation of the current
      // link with the base function of the link and the normalised
      // input value
      float* param = that->_bases->_val +
        VecGet(that->_links, jLink) * NN_NBPARAMBASE;
      float x = 0.0;
      if (prevLink[0] < startHid)
        x = VecGet(input, prevLink[0]);
      else
        x = NNGetHiddenValue(that, prevLink[0] - startHid);
      prevOut *= NNBaseFun(param, x);
      // Move to the next link
      ++iLink;
    }
    // Update the output of the last link
```

```
    if (prevLink[1] < startOut) {
      long iVal = prevLink[1] - startHid;
      float nVal =
        MIN(1.0, MAX(-1.0, VecGet(that->_hidVal, iVal) + prevOut));
      VecSet(that->_hidVal, iVal, nVal);
    } else {
      long iVal = prevLink[1] - startOut;
      float nVal = VecGet(output, iVal) + prevOut;
      VecSet(output, iVal, nVal);
    }
  }
}

// Function which return the JSON encoding of 'that'
JSONNode* NNEncodeAsJSON(const NeuraNet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
#endif
  // Create the JSON structure
  JSONNode* json = JSONCreate();
  // Declare a buffer to convert value into string
  char val[100];
  // Encode the nbInputVal
  sprintf(val, "%d", that->_nbInputVal);
  JSONAddProp(json, "_nbInputVal", val);
  // Encode the nbOutputVal
  sprintf(val, "%d", that->_nbOutputVal);
  JSONAddProp(json, "_nbOutputVal", val);
  // Encode the nbMaxHidVal
  sprintf(val, "%ld", that->_nbMaxHidVal);
  JSONAddProp(json, "_nbMaxHidVal", val);
  // Encode the nbMaxBases
  sprintf(val, "%ld", that->_nbMaxBases);
  JSONAddProp(json, "_nbMaxBases", val);
  // Encode the nbMaxLinks
  sprintf(val, "%ld", that->_nbMaxLinks);
  JSONAddProp(json, "_nbMaxLinks", val);
  // Encode the bases
  JSONAddProp(json, "_bases", VecEncodeAsJSON(that->_bases));
  // Encode the links
  JSONAddProp(json, "_links", VecEncodeAsJSON(that->_links));
  // Return the created JSON
  return json;
}

// Function which decode from JSON encoding 'json' to 'that'
bool NNDecodeAsJSON(NeuraNet** that, const JSONNode* const json) {
#if BUILDMODE == 0
  if (that == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'that' is null");
    PBErrCatch(PBMathErr);
  }
  if (json == NULL) {
    PBMathErr->_type = PBErrTypeNullPointer;
    sprintf(PBMathErr->_msg, "'json' is null");
    PBErrCatch(PBMathErr);
  }
```

```
#endif
  // If 'that' is already allocated
  if (*that != NULL)
    // Free memory
    NeuraNetFree(that);
  // Decode the nbInputVal
  JSONNode* prop = JSONProperty(json, "_nbInputVal");
  if (prop == NULL) {
    return false;
  }
  int nbInputVal = atoi(JSONLabel(JSONValue(prop, 0)));
  // Decode the nbOutputVal
  prop = JSONProperty(json, "_nbOutputVal");
  if (prop == NULL) {
    return false;
  }
  int nbOutputVal = atoi(JSONLabel(JSONValue(prop, 0)));
  // Decode the nbMaxHidVal
  prop = JSONProperty(json, "_nbMaxHidVal");
  if (prop == NULL) {
    return false;
  }
  long nbMaxHidVal = atol(JSONLabel(JSONValue(prop, 0)));
  // Decode the nbMaxBases
  prop = JSONProperty(json, "_nbMaxBases");
  if (prop == NULL) {
    return false;
  }
  long nbMaxBases = atol(JSONLabel(JSONValue(prop, 0)));
  // Decode the nbMaxLinks
  prop = JSONProperty(json, "_nbMaxLinks");
  if (prop == NULL) {
    return false;
  }
  long nbMaxLinks = atol(JSONLabel(JSONValue(prop, 0)));
  // Allocate memory
  *that = NeuraNetCreate(nbInputVal, nbOutputVal, nbMaxHidVal,
    nbMaxBases, nbMaxLinks);
  // Decode the bases
  prop = JSONProperty(json, "_bases");
  if (prop == NULL) {
    return false;
  }
  if (!VecDecodeAsJSON(&((*that)->_bases), prop)) {
    return false;
  }
  // Decode the links
  prop = JSONProperty(json, "_links");
  if (prop == NULL) {
    return false;
  }
  if (!VecDecodeAsJSON(&((*that)->_links), prop)) {
    return false;
  }
  // Return the success code
  return true;
}

// Save the NeuraNet 'that' to the stream 'stream'
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true if the NeuraNet could be saved, false else
```

```
bool NNSave(const NeuraNet* const that, FILE* const stream, const bool compact) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
  if (stream == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'stream' is null");
    PBErrCatch(NeuraNetErr);
  }
#endif
  // Get the JSON encoding
  JSONNode* json = NNEncodeAsJSON(that);
  // Save the JSON
  if (!JSONSave(json, stream, compact)) {
    return false;
  }
  // Free memory
  JSONFree(&json);
  // Return success code
  return true;
}

// Load the NeuraNet 'that' from the stream 'stream'
// If 'that' is not null the memory is first freed
// Return true if the NeuraNet could be loaded, false else
bool NNLoad(NeuraNet** that, FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
  if (stream == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'stream' is null");
    PBErrCatch(NeuraNetErr);
  }
#endif
  // Declare a json to load the encoded data
  JSONNode* json = JSONCreate();
  // Load the whole encoded data
  if (!JSONLoad(json, stream)) {
    return false;
  }
  // Decode the data from the JSON
  if (!NNDecodeAsJSON(that, json)) {
    return false;
  }
  // Free the memory used by the JSON
  JSONFree(&json);
  // Return the success code
  return true;
}

// Print the NeuraNet 'that' to the stream 'stream'
void NNPrintln(const NeuraNet* const that, FILE* const stream) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
```

```
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
  if (stream == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'stream' is null");
    PBErrCatch(NeuraNetErr);
  }
#endif
  fprintf(stream, "nbInput: %d\n", that->_nbInputVal);
  fprintf(stream, "nbOutput: %d\n", that->_nbOutputVal);
  fprintf(stream, "nbHidden: %ld\n", that->_nbMaxHidVal);
  fprintf(stream, "nbMaxBases: %ld\n", that->_nbMaxBases);
  fprintf(stream, "nbMaxLinks: %ld\n", that->_nbMaxLinks);
  fprintf(stream, "bases: ");
  VecPrint(that->_bases, stream);
  fprintf(stream, "\n");
  fprintf(stream, "links: ");
  VecPrint(that->_links, stream);
  fprintf(stream, "\n");
  fprintf(stream, "hidden values: ");
  VecPrint(that->_hidVal, stream);
  fprintf(stream, "\n");
}

// Set the links description of the NeuraNet 'that' to a copy of 'links'
// Links with a base function equals to -1 are ignored
// If the input id is higher than the output id they are swap
// The links description in the NeuraNet are ordered in increasing
// value of input id and output id, but 'links' doesn't have to be
// sorted
// Each link is defined by (base index, input index, output index)
// If base index equals -1 it means the link is inactive
void NNSetLinks(NeuraNet* const that, VecLong* const links) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
  if (links == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'links' is null");
    PBErrCatch(NeuraNetErr);
  }
  if (VecGetDim(links) != that->_nbMaxLinks * NN_NBPARAMLINK) {
    NeuraNetErr->_type = PBErrTypeInvalidArg;
    sprintf(NeuraNetErr->_msg,
      "'links' 's dimension is invalid (%ld!=%ld)",
      VecGetDim(links), that->_nbMaxLinks);
    PBErrCatch(NeuraNetErr);
  }
#endif
  // Declare a GSet to sort the links
  GSet set = GSetCreateStatic();
  // Declare a variable to memorize the maximum id
  long maxId = NNGetNbInput(that) + NNGetNbMaxHidden(that) +
    NNGetNbOutput(that);
  // Loop on links
  for (long iLink = 0; iLink < NNGetNbMaxLinks(that) * NN_NBPARAMLINK;
    iLink += NN_NBPARAMLINK) {
    // If this link is active
```

```
    if (VecGet(links, iLink) != -1) {
      // Declare two variables to memorize the effective input and output
      long in = VecGet(links, iLink + 1);
      long out = VecGet(links, iLink + 2);
      // If the input is greater than the output
      if (in > out) {
        // Swap the input and output
        long tmp = in;
        in = out;
        out = tmp;
      }
      // Add the link to the set, sorting on input and ouput
      float sortVal = (float)(in * maxId + out);
      GSetAddSort(&set, links->_val + iLink, sortVal);
    }
  }
  // Declare a variable to memorize the number of active links
  long nbLink = GSetNbElem(&set);
  // If there are active links
  if (nbLink > 0) {
    // loop on active sorted links
    GSetIterForward iter = GSetIterForwardCreateStatic(&set);
    long iLink = 0;
    do {
      long *link = GSetIterGet(&iter);
      VecSet(that->_links, iLink * NN_NBPARAMLINK, link[0]);
      if (link[1] <= link[2]) {
        VecSet(that->_links, iLink * NN_NBPARAMLINK + 1, link[1]);
        VecSet(that->_links, iLink * NN_NBPARAMLINK + 2, link[2]);
      } else {
        VecSet(that->_links, iLink * NN_NBPARAMLINK + 1, link[2]);
        VecSet(that->_links, iLink * NN_NBPARAMLINK + 2, link[1]);
      }
      ++iLink;
    } while (GSetIterStep(&iter));
  }
  // Reset the inactive links
  for (long iLink = nbLink; iLink < NNGetNbMaxLinks(that); ++iLink)
    VecSet(that->_links, iLink * NN_NBPARAMLINK, -1);
  // Free the memory
  GSetFlush(&set);
}

// Save the links of the NeuraNet 'that' into the file at 'url' in a
// format readable by CloudGraph
// Return true if we could save, else false
bool NNSaveLinkAsCloudGraph(const NeuraNet* const that,
  const char* url) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
  if (url == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'url' is null");
    PBErrCatch(NeuraNetErr);
  }
#endif
  // Declare a flag to memorize the returned value
  bool ret = true;
```

```
// Open the file
FILE* cloud = fopen(url, "w");
// If we could open the file
if (cloud != NULL) {
  // Save the categories
  if (!PBErrPrintf(NeuraNetErr, cloud, "%s", "3\n")) {
    fclose(cloud);
    return false;
  }
  if (!PBErrPrintf(NeuraNetErr, cloud, "%s", "0 255 0 0 Input\n")) {
    fclose(cloud);
    return false;
  }
  if (!PBErrPrintf(NeuraNetErr, cloud, "%s", "1 0 255 0 Hidden\n")) {
    fclose(cloud);
    return false;
  }
  if (!PBErrPrintf(NeuraNetErr, cloud, "%s", "2 0 0 255 Output\n")) {
    fclose(cloud);
    return false;
  }
  // Save the nb of nodes
  if (!PBErrPrintf(NeuraNetErr, cloud, "%ld\n",
    NNGetNbInput(that) + NNGetNbMaxHidden(that) +
    NNGetNbOutput(that))) {
    fclose(cloud);
    return false;
  }
  // Save the input nodes
  for (long iNode = 0; iNode < NNGetNbInput(that); ++iNode) {
    if (!PBErrPrintf(NeuraNetErr, cloud, "%ld 0 ", iNode)) {
      fclose(cloud);
      return false;
    }
    if (!PBErrPrintf(NeuraNetErr, cloud, "%ld\n", iNode)) {
      fclose(cloud);
      return false;
    }
  }
  // Save the hidden nodes
  for (long iNode = 0; iNode < NNGetNbMaxHidden(that); ++iNode) {
    long jNode = iNode + NNGetNbInput(that);
    if (!PBErrPrintf(NeuraNetErr, cloud, "%ld 1 ", jNode)) {
      fclose(cloud);
      return false;
    }
    if (!PBErrPrintf(NeuraNetErr, cloud, "%ld\n", jNode)) {
      fclose(cloud);
      return false;
    }
  }
  // Save the output nodes
  for (long iNode = 0; iNode < NNGetNbOutput(that); ++iNode) {
    long jNode = iNode + NNGetNbInput(that) + NNGetNbMaxHidden(that);
    if (!PBErrPrintf(NeuraNetErr, cloud, "%ld 2 ", jNode)) {
      fclose(cloud);
      return false;
    }
    if (!PBErrPrintf(NeuraNetErr, cloud, "%ld\n", jNode)) {
      fclose(cloud);
      return false;
    }
```

```
    }
    // Save the links
    PBErrPrintf(NeuraNetErr, cloud, "%ld\n", NNGetNbActiveLinks(that));
    for (long iLink = 0; iLink < NNGetNbMaxLinks(that); ++iLink) {
      // If this link is active
      if (VecGet(NNLinks(that), iLink * NN_NBPARAMLINK) != -1) {
        if (!PBErrPrintf(NeuraNetErr, cloud, "%ld ",
          VecGet(NNLinks(that), iLink * NN_NBPARAMLINK + 1))) {
          fclose(cloud);
          return false;
        }
        if (!PBErrPrintf(NeuraNetErr, cloud, "%ld\n",
          VecGet(NNLinks(that), iLink * NN_NBPARAMLINK + 2))) {
          fclose(cloud);
          return false;
        }
      }
    }
    // Close the file
    fclose(cloud);
  // Else, we coudln't open the file
  } else {
    // Set the flag
    ret = false;
  }
  // Return the flag
  return ret;
}

// Get the Simpson's diversity index of the hidden values of the
// NeuraNet 'that'
// Return value in [0.0, 1.0], 0.0 means no diversity, 1.0 means max
// diversity
float NNGetHiddenValSimpsonDiv(const NeuraNet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
#endif
  // Declare a variable to emmorize the result
  float div = 0.0;
  // Declare a variable to check if a link output to a hidden value
  long iMaxHidden = NNGetNbInput(that) + NNGetNbMaxHidden(that);
  // Declare two variables for calculation
  VecFloat* nb = VecFloatCreate(iMaxHidden);
  float tot = 0.0;
  // Loop on links
  for (int iLink = NNGetNbMaxLinks(that); iLink--;) {
    // If this link is active and its ouput is a hidden value
    if (VecGet(NNLinks(that), iLink * NN_NBPARAMLINK) != -1 &&
      VecGet(NNLinks(that), iLink * NN_NBPARAMLINK + 2) < iMaxHidden) {
      // Calculate the diversity
      ++tot;
      VecSetAdd(nb, VecGet(NNLinks(that),
        iLink * NN_NBPARAMLINK + 1), 1.0);
    }
  }
  // Calculate the diversity
  if (tot >= 2.0) {
    for (int iLink = iMaxHidden; iLink--;) {
```

```
      div += VecGet(nb, iLink) * (VecGet(nb, iLink) - 1.0);
    }
    div = 1.0 - div / (tot * (tot - 1.0));
    div *= tot / (float)NNGetNbMaxLinks(that);
  }
  // Free memory
  VecFree(&nb);
  // Return the diversity
  return MIN(1.0, MAX(-1.0, div));
}

// Prune the NeuraNet 'that' by removing the useless links (those with
// no influence on outputs
void NNPrune(const NeuraNet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
#endif
  // Declare a variable to memorise the start index of output
  long startOut = NNGetNbInput(that) + NNGetNbMaxHidden(that);
  // Loop on links
  for (int iLink = NNGetNbMaxLinks(that); iLink--;) {
    // If the output of this link is not an output and it's active
    if (VecGet(NNLinks(that), iLink * NN_NBPARAMLINK) != -1 &&
      VecGet(NNLinks(that), iLink * NN_NBPARAMLINK + 2) < startOut) {
      // Search in following links one that use the output of this link
      // as an input
      bool flag = false;
      for (int jLink = iLink + 1;
        !flag && jLink < NNGetNbMaxLinks(that); ++jLink) {
        if (VecGet(NNLinks(that), iLink * NN_NBPARAMLINK + 2) ==
          VecGet(NNLinks(that), jLink * NN_NBPARAMLINK + 1)) {
          flag = true;
        }
      }
      // If the output of this link is never used
      if (!flag)
        // Disactivate it
        VecSet(that->_links, iLink * NN_NBPARAMLINK, -1);
    }
  }
}
```

3.2 pbmath-inline.c

```
// ============ NEURANET-INLINE.C ================

// ----- NeuraNetBaseFun

// ================ Functions implementation ====================

// Generic base function for the NeuraNet
// 'param' is an array of 3 float all in [-1,1]
// 'x' is the input value
// NNBaseFun(param,x)=
// tan(param[0]*NN_THETA)*(x+param[1])+param[2]
```

```
#if BUILDMODE != 0
inline
#endif
float NNBaseFun(const float* const param, const float x) {
#if BUILDMODE == 0
  if (param == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'param' is null");
    PBErrCatch(NeuraNetErr);
  }
#endif
  return tan(param[0] * NN_THETA) * (x + param[1]) + param[2];
}

// ----- NeuraNet

// ================ Functions implementation ====================

// Get the nb of input values of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
int NNGetNbInput(const NeuraNet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
#endif
  return that->_nbInputVal;
}

// Get the nb of output values of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
int NNGetNbOutput(const NeuraNet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
#endif
  return that->_nbOutputVal;
}

// Get the nb max of hidden values of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
long NNGetNbMaxHidden(const NeuraNet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
#endif
  return that->_nbMaxHidVal;
}
```

```
// Get the nb max of base functions of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
long NNGetNbMaxBases(const NeuraNet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
#endif
  return that->_nbMaxBases;
}

// Get the nb of base functions for convolution of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
long NNGetNbBasesConv(const NeuraNet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
#endif
  return that->_nbBasesConv;
}

// Get the nb of base functions per cell for convolution of
// the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
long NNGetNbBasesCellConv(const NeuraNet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
#endif
  return that->_nbBasesCellConv;
}

// Get the nb max of links of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
long NNGetNbMaxLinks(const NeuraNet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
#endif
  return that->_nbMaxLinks;
}
```

```
// Get the parameters of the base functions of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
const VecFloat* NNBases(const NeuraNet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
#endif
  return that->_bases;
}

// Get the links description of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
const VecLong* NNLinks(const NeuraNet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
#endif
  return that->_links;
}

// Get the hidden values of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
const VecFloat* NNHiddenValues(const NeuraNet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
#endif
  return that->_hidVal;
}

// Get the 'iVal'-th hidden value of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
float NNGetHiddenValue(const NeuraNet* const that, const long iVal) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
  if (iVal < 0 || iVal >= that->_nbMaxHidVal) {
    NeuraNetErr->_type = PBErrTypeInvalidArg;
    sprintf(NeuraNetErr->_msg, "'iVal' is invalid (0<=%ld<%ld)",
      iVal, that->_nbMaxHidVal);
    PBErrCatch(NeuraNetErr);
  }
```

```
#endif
  return VecGet(that->_hidVal, iVal);
}

// Set the parameters of the base functions of the NeuraNet 'that' to
// a copy of 'bases'
// 'bases' must be of dimension that->nbMaxBases * NN_NBPARAMBASE
//  each base is defined as param[3] in [-1,1]
// tan(param[0]*NN_THETA)*(x+param[1])+param[2]
#if BUILDMODE != 0
inline
#endif
void NNSetBases(NeuraNet* const that, const VecFloat* const bases) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
  if (bases == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'bases' is null");
    PBErrCatch(NeuraNetErr);
  }
  if (VecGetDim(bases) != that->_nbMaxBases * NN_NBPARAMBASE) {
    NeuraNetErr->_type = PBErrTypeInvalidArg;
    sprintf(NeuraNetErr->_msg,
      "'bases' 's dimension is invalid (%ld!=%ld)",
      VecGetDim(bases), that->_nbMaxBases * NN_NBPARAMBASE);
    PBErrCatch(NeuraNetErr);
  }
#endif
  VecCopy(that->_bases, bases);
}
// Set the 'iBase'-th parameter of the base functions of the NeuraNet
// 'that' to 'base'
#if BUILDMODE != 0
inline
#endif
void NNBasesSet(NeuraNet* const that, const long iBase,
  const float base) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
  if (iBase < 0 || iBase >= that->_nbMaxBases * NN_NBPARAMBASE) {
    NeuraNetErr->_type = PBErrTypeInvalidArg;
    sprintf(NeuraNetErr->_msg,
      "'iBase' is invalid (0<=%ld<%ld)",
      iBase, that->_nbMaxBases * NN_NBPARAMBASE);
    PBErrCatch(NeuraNetErr);
  }
#endif
  VecSet(that->_bases, iBase, base);
}

// Get the number of active links in the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
```

```
long NNGetNbActiveLinks(const NeuraNet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    NeuraNetErr->_type = PBErrTypeNullPointer;
    sprintf(NeuraNetErr->_msg, "'that' is null");
    PBErrCatch(NeuraNetErr);
  }
#endif
  // Declare a variable to memorize the result
  long nb = 0;
  // Loop on links
  for (long iLink = NNGetNbMaxLinks(that); iLink--;) {
    // If this link is active
    if (VecGet(NNLinks(that), iLink * NN_NBPARAMLINK) != -1)
      // Increment the number of active links
      ++nb;
  }
  // Return the result
  return nb;
}
```

4 Makefile

```
# Build mode
# 0: development (max safety, no optimisation)
# 1: release (min safety, optimisation)
# 2: fast and furious (no safety, optimisation)
BUILD_MODE?=1

all: main nn2cloud

# Makefile definitions
MAKEFILE_INC=../PBMake/Makefile.inc
include $(MAKEFILE_INC)

# Rules to make the executable
repo=neuranet
$($(repo)_EXENAME): \
$($(repo)_EXENAME).o \
$($(repo)_EXE_DEP) \
$($(repo)_DEP)
$(COMPILER) `echo "$($(repo)_EXE_DEP) $($(repo)_EXENAME).o" | tr ' ' '\n' | sort -u` $(LINK_ARG) $($(repo)_LINK_ARG)

$($(repo)_EXENAME).o: \
$($(repo)_DIR)/$($(repo)_EXENAME).c \
$($(repo)_INC_H_EXE) \
$($(repo)_EXE_DEP)
$(COMPILER) $(BUILD_ARG) $($(repo)_BUILD_ARG) `echo "$($(repo)_INC_DIR)" | tr ' ' '\n' | sort -u` -c $($(repo)_DIR)/$

nn2cloud: \
nn2cloud.o \
$($(repo)_EXENAME).o \
$($(repo)_EXE_DEP) \
$($(repo)_DEP)
$(COMPILER) nn2cloud.o `echo "$($(repo)_EXE_DEP)" | tr ' ' '\n' | sort -u` $(LINK_ARG) $($(repo)_LINK_ARG) -o nn2clou

nn2cloud.o: \
```

```
nn2cloud.c \
$($(repo)_INC_H_EXE) \
$($(repo)_EXE_DEP)
$(COMPILER) $(BUILD_ARG) $($(repo)_BUILD_ARG) `echo "$($(repo)_INC_DIR)" | tr ' ' '\n' | sort -u` -c $($(repo)_DIR)/n

cloud: cloud.txt
../CloudGraph/cloudGraph -file ./cloud.txt -tga cloud.tga -familyLabel -circle -curved 0.5
```

5 Unit tests

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "pberr.h"
#include "genalg.h"
#include "neuranet.h"

#define RANDOMSEED 4

void UnitTestNNBaseFun() {
  srandom(RANDOMSEED);
  float param[4];
  float x = 0.0;
  float check[100] = {
    -4.664967,-3.920526,-3.176085,-2.431644,-1.687203,-0.942763,
    -0.198322,0.546119,1.290560,2.035000,-0.153181,-0.403978,
    -0.654776,-0.905573,-1.156371,-1.407168,-1.657966,-1.908763,
    -2.159561,-2.410358,0.586943,0.301165,0.015387,-0.270391,
    -0.556169,-0.841946,-1.127724,-1.413502,-1.699280,-1.985057,
    2.760699,2.805863,2.851027,2.896191,2.941355,2.986519,
    3.031683,3.076847,3.122011,3.167175,0.774302,0.903425,
    1.032548,1.161672,1.290795,1.419918,1.549042,1.678165,
    1.807288,1.936412,2.321817,2.100005,1.878192,1.656379,
    1.434567,1.212754,0.990941,0.769129,0.547316,0.325503,
    -1.349660,-1.452492,-1.555323,-1.658154,-1.760985,-1.863817,
    -1.966648,-2.069479,-2.172311,-2.275142,2.030713,1.867117,
    1.703522,1.539926,1.376330,1.212735,1.049139,0.885544,0.721949,
    0.558353,-1.439830,-1.174441,-0.909051,-0.643662,-0.378272,
    -0.112883,0.152507,0.417896,0.683286,0.948675,0.819425,0.765620,
    0.711816,0.658011,0.604206,0.550401,0.496596,0.442791,0.388987,
    0.335182
    };
  for (int iTest = 0; iTest < 10; ++iTest) {
    param[0] = 2.0 * (rnd() - 0.5);
    param[1] = 2.0 * rnd();
    param[2] = 2.0 * (rnd() - 0.5) * PBMATH_PI;
    param[3] = 2.0 * (rnd() - 0.5);
    for (int ix = 0; ix < 10; ++ix) {
      x = -1.0 + 2.0 * 0.1 * (float)ix;
      float y = NNBaseFun(param, x);
      if (ISEQUALF(y, check[iTest * 10 + ix]) == false) {
        NeuraNetErr->_type = PBErrTypeUnitTestFailed;
        sprintf(NeuraNetErr->_msg, "NNBaseFun failed");
        PBErrCatch(NeuraNetErr);
      }
```

```
    }
  }
  printf("UnitTestNNBaseFun OK\n");
}

void UnitTestNeuraNetCreateFree() {
  int nbIn = 1;
  int nbOut = 2;
  int nbHid = 3;
  int nbBase = 4;
  int nbLink = 5;
  NeuraNet* nn = NeuraNetCreate(nbIn, nbOut, nbHid, nbBase, nbLink);
  if (nn == NULL ||
    nn->_nbInputVal != nbIn ||
    nn->_nbOutputVal != nbOut ||
    nn->_nbMaxHidVal != nbHid ||
    nn->_nbMaxBases != nbBase ||
    nn->_nbBasesConv != 0 ||
    nn->_nbMaxLinks != nbLink ||
    nn->_bases == NULL ||
    nn->_links == NULL ||
    nn->_hidVal == NULL) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NeuraNetFree failed");
    PBErrCatch(NeuraNetErr);
  }
  NeuraNetFree(&nn);
  if (nn != NULL) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NeuraNetFree failed");
    PBErrCatch(NeuraNetErr);
  }
  printf("UnitTestNeuraNetCreateFree OK\n");
}

void UnitTestNeuraNetCreateFullyConnected() {
  int nbIn = 2;
  int nbOut = 3;
  VecLong* hiddenLayers = NULL;
  NeuraNet* nn = NeuraNetCreateFullyConnected(nbIn, nbOut, hiddenLayers);
  if (nn == NULL ||
    nn->_nbInputVal != nbIn ||
    nn->_nbOutputVal != nbOut ||
    nn->_nbMaxHidVal != 0 ||
    nn->_nbMaxBases != 6 ||
    nn->_nbMaxLinks != 6 ||
    nn->_bases == NULL ||
    nn->_links == NULL ||
    nn->_hidVal != NULL) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NeuraNetCreateFullyConnected failed");
    PBErrCatch(NeuraNetErr);
  }
  int checka[18] = {
    0,0,2, 1,0,3, 2,0,4,
    3,1,2, 4,1,3, 5,1,4
  };
  for (int i = 18; i--;)
    if (VecGet(nn->_links, i) != checka[i]) {
      NeuraNetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(NeuraNetErr->_msg, "NeuraNetCreateFullyConnected failed");
      PBErrCatch(NeuraNetErr);
```

```
    }
  NeuraNetFree(&nn);
  nbIn = 5;
  nbOut = 2;
  hiddenLayers = VecLongCreate(2);
  VecSet(hiddenLayers, 0, 4);
  VecSet(hiddenLayers, 1, 3);
  nn = NeuraNetCreateFullyConnected(nbIn, nbOut, hiddenLayers);
  if (nn == NULL ||
    nn->_nbInputVal != nbIn ||
    nn->_nbOutputVal != nbOut ||
    nn->_nbMaxHidVal != 7 ||
    nn->_nbMaxBases != 38 ||
    nn->_nbMaxLinks != 38 ||
    nn->_bases == NULL ||
    nn->_links == NULL ||
    nn->_hidVal == NULL) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NeuraNetCreateFullyConnected failed");
    PBErrCatch(NeuraNetErr);
  }
  int checkb[114] = {
    0,0,5, 1,0,6, 2,0,7, 3,0,8,
    4,1,5, 5,1,6, 6,1,7, 7,1,8,
    8,2,5, 9,2,6, 10,2,7, 11,2,8,
    12,3,5, 13,3,6, 14,3,7, 15,3,8,
    16,4,5, 17,4,6, 18,4,7, 19,4,8,
    20,5,9, 21,5,10, 22,5,11,
    23,6,9, 24,6,10, 25,6,11,
    26,7,9, 27,7,10, 28,7,11,
    29,8,9, 30,8,10, 31,8,11,
    32,9,12, 33,9,13,
    34,10,12, 35,10,13,
    36,11,12, 37,11,13
  };
  for (int i = 114; i--;)
    if (VecGet(nn->_links, i) != checkb[i]) {
      NeuraNetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(NeuraNetErr->_msg, "NeuraNetCreateFullyConnected failed");
      PBErrCatch(NeuraNetErr);
    }
  NeuraNetFree(&nn);
  VecFree(&hiddenLayers);
  printf("UnitTestNeuraNetCreateFullyConnected OK\n");
}

void UnitTestNeuraNetCreateConvolution() {
  int nbOut = 2;
  int thickConv = 2;
  int depthConv = 2;
  VecShort* dimIn = VecShortCreate(2);
  VecSet(dimIn, 0, 4);
  VecSet(dimIn, 1, 3);
  VecShort* dimCell = VecShortCreate(2);
  VecSet(dimCell, 0, 2);
  VecSet(dimCell, 1, 2);
  NeuraNet* nn = NeuraNetCreateConvolution(dimIn, nbOut, dimCell,
    depthConv, thickConv);
  NNPrintln(nn, stdout);
  if (nn == NULL ||
    nn->_nbInputVal != 12 ||
    nn->_nbOutputVal != 2 ||
```

```
    nn->_nbMaxHidVal != 16 ||
    nn->_nbMaxBases != 24 ||
    nn->_nbMaxLinks != 72 ||
    nn->_bases == NULL ||
    nn->_links == NULL ||
    nn->_hidVal == NULL) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NeuraNetCreateConvolution failed");
    PBErrCatch(NeuraNetErr);
  }
  int check[216] = {
    0,0,12, 4,0,18, 1,1,12, 0,1,13, 5,1,18, 4,1,19, 1,2,13, 0,2,14,
    5,2,19, 4,2,20, 1,3,14, 5,3,20, 2,4,12, 0,4,15, 6,4,18, 4,4,21,
    3,5,12, 2,5,13, 1,5,15, 0,5,16, 7,5,18, 6,5,19, 5,5,21, 4,5,22,
    3,6,13, 2,6,14, 1,6,16, 0,6,17, 7,6,19, 6,6,20, 5,6,22, 4,6,23,
    3,7,14, 1,7,17, 7,7,20, 5,7,23, 2,8,15, 6,8,21, 3,9,15, 2,9,16,
    7,9,21, 6,9,22, 3,10,16, 2,10,17, 7,10,22, 6,10,23, 3,11,17,
    7,11,23, 8,12,24, 9,13,24, 8,13,25, 9,14,25, 10,15,24, 11,16,24,
    10,16,25, 11,17,25, 12,18,26, 13,19,26, 12,19,27, 13,20,27,
    14,21,26, 15,22,26, 14,22,27, 15,23,27, 16,24,28, 17,24,29,
    18,25,28, 19,25,29, 20,26,28, 21,26,29, 22,27,28, 23,27,29
    };
  for (int iCheck = 216; iCheck--;) {
    if (VecGet(nn->_links, iCheck) != check[iCheck]) {
      NeuraNetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(NeuraNetErr->_msg, "NeuraNetCreateConvolution failed");
      PBErrCatch(NeuraNetErr);
    }
  }
  NNSaveLinkAsCloudGraph(nn, "./cloudConv.txt");
  NeuraNetFree(&nn);
  VecFree(&dimIn);
  VecFree(&dimCell);
  printf("UnitTestNeuraNetCreateConvolution OK\n");
}

void UnitTestNeuraNetGetSet() {
  int nbIn = 10;
  int nbOut = 20;
  int nbHid = 30;
  int nbBase = 4;
  int nbLink = 5;
  NeuraNet* nn = NeuraNetCreate(nbIn, nbOut, nbHid, nbBase, nbLink);
  if (NNGetNbInput(nn) != nbIn) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNGetNbInput failed");
    PBErrCatch(NeuraNetErr);
  }
  if (NNGetNbMaxBases(nn) != nbBase) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNGetNbMaxBases failed");
    PBErrCatch(NeuraNetErr);
  }
  if (NNGetNbBasesConv(nn) != 0) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNGetNbBasesConv failed");
    PBErrCatch(NeuraNetErr);
  }
  if (NNGetNbBasesCellConv(nn) != 0) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNGetNbBasesCellConv failed");
    PBErrCatch(NeuraNetErr);
```

```
}
if (NNGetNbMaxHidden(nn) != nbHid) {
  NeuraNetErr->_type = PBErrTypeUnitTestFailed;
  sprintf(NeuraNetErr->_msg, "NNGetNbMaxHidden failed");
  PBErrCatch(NeuraNetErr);
}
if (NNGetNbMaxLinks(nn) != nbLink) {
  NeuraNetErr->_type = PBErrTypeUnitTestFailed;
  sprintf(NeuraNetErr->_msg, "NNGetNbMaxLinks failed");
  PBErrCatch(NeuraNetErr);
}
if (NNGetNbOutput(nn) != nbOut) {
  NeuraNetErr->_type = PBErrTypeUnitTestFailed;
  sprintf(NeuraNetErr->_msg, "NNGetNbOutput failed");
  PBErrCatch(NeuraNetErr);
}
if (NNBases(nn) != nn->_bases) {
  NeuraNetErr->_type = PBErrTypeUnitTestFailed;
  sprintf(NeuraNetErr->_msg, "NNBases failed");
  PBErrCatch(NeuraNetErr);
}
if (NNLinks(nn) != nn->_links) {
  NeuraNetErr->_type = PBErrTypeUnitTestFailed;
  sprintf(NeuraNetErr->_msg, "NNLinks failed");
  PBErrCatch(NeuraNetErr);
}
if (NNHiddenValues(nn) != nn->_hidVal) {
  NeuraNetErr->_type = PBErrTypeUnitTestFailed;
  sprintf(NeuraNetErr->_msg, "NNHiddenValues failed");
  PBErrCatch(NeuraNetErr);
}
VecFloat* bases = VecFloatCreate(nbBase * NN_NBPARAMBASE);
for (int i = nbBase * NN_NBPARAMBASE; i--;)
  VecSet(bases, i, 0.01 * (float)i);
NNSetBases(nn, bases);
for (int i = nbBase * NN_NBPARAMBASE; i--;)
  if (ISEQUALF(VecGet(NNBases(nn), i), 0.01 * (float)i) == false) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNSetBases failed");
    PBErrCatch(NeuraNetErr);
  }
VecFree(&bases);
VecLong* links = VecLongCreate(15);
short data[15] = {2,2,35, 1,1,12, -1,0,0, 2,15,20, 3,20,15};
for (int i = 15; i--;)
  VecSet(links, i, data[i]);
NNSetLinks(nn, links);
short check[15] = {1,1,12,2,2,35,2,15,20,3,15,20,-1,0,0};
for (int i = 15; i--;)
  if (VecGet(NNLinks(nn), i) != check[i]) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNSetLinks failed");
    PBErrCatch(NeuraNetErr);
  }
if (NNGetNbActiveLinks(nn) != 4) {
  NeuraNetErr->_type = PBErrTypeUnitTestFailed;
  sprintf(NeuraNetErr->_msg, "NNGetNbActiveLinks failed");
  PBErrCatch(NeuraNetErr);
}
VecFree(&links);
NeuraNetFree(&nn);
printf("UnitTestNeuraNetGetSet OK\n");
```

```
}

void UnitTestNeuraNetSaveLoadPrune() {
  int nbIn = 10;
  int nbOut = 20;
  int nbHid = 30;
  int nbBase = 4;
  int nbLink = 5;
  NeuraNet* nn = NeuraNetCreate(nbIn, nbOut, nbHid, nbBase, nbLink);
  VecFloat* bases = VecFloatCreate(nbBase * NN_NBPARAMBASE);
  for (int i = nbBase * NN_NBPARAMBASE; i--;)
    VecSet(bases, i, 0.01 * (float)i);
  NNSetBases(nn, bases);
  VecFree(&bases);
  VecLong* links = VecLongCreate(15);
  short data[15] = {2,2,35, 1,1,12, -1,0,0, 2,15,20, 3,20,15};
  for (int i = 15; i--;)
    VecSet(links, i, data[i]);
  NNSetLinks(nn, links);
  VecFree(&links);
  FILE* fd = fopen("./neuranet.txt", "w");
  if (NNSave(nn, fd, false) == false) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNSave failed");
    PBErrCatch(NeuraNetErr);
  }
  fclose(fd);
  fd = fopen("./neuranet.txt", "r");
  NeuraNet* loaded = NeuraNetCreate(1, 1, 1, 1, 1);
  if (NNLoad(&loaded, fd) == false) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNLoad failed");
    PBErrCatch(NeuraNetErr);
  }
  if (NNGetNbInput(loaded) != nbIn ||
    NNGetNbMaxBases(loaded) != nbBase ||
    NNGetNbMaxHidden(loaded) != nbHid ||
    NNGetNbMaxLinks(loaded) != nbLink ||
    NNGetNbOutput(loaded) != nbOut) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNLoad failed");
    PBErrCatch(NeuraNetErr);
  }
  for (int i = nbBase * NN_NBPARAMBASE; i--;)
    if (ISEQUALF(VecGet(NNBases(loaded), i), 0.01 * (float)i) == false) {
      NeuraNetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(NeuraNetErr->_msg, "NNLoad failed");
      PBErrCatch(NeuraNetErr);
    }
  short check[15] = {1,1,12,2,2,35,2,15,20,3,15,20,-1,0,0};
  for (int i = 15; i--;)
    if (VecGet(NNLinks(loaded), i) != check[i]) {
      NeuraNetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(NeuraNetErr->_msg, "NNLoad failed");
      PBErrCatch(NeuraNetErr);
    }
  fclose(fd);
  NeuraNetFree(&loaded);
  NNPrune(nn);
  short checkprune[15] = {-1,1,12,-1,2,35,-1,15,20,-1,15,20,-1,0,0};
  for (int i = 15; i--;)
    if (VecGet(NNLinks(nn), i) != checkprune[i]) {
```

```
      NeuraNetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(NeuraNetErr->_msg, "NNPrune failed");
      PBErrCatch(NeuraNetErr);
    }
  NeuraNetFree(&nn);
  printf("UnitTestNeuraNetSaveLoadPrune OK\n");
}

void UnitTestNeuraNetEvalPrintHiddenLinkSimpsonDiv() {
  int nbIn = 3;
  int nbOut = 3;
  int nbHid = 3;
  int nbBase = 3;
  int nbLink = 7;
  NeuraNet* nn = NeuraNetCreate(nbIn, nbOut, nbHid, nbBase, nbLink);
  // hidden[0] = tan(0.5*NN_THETA)*tan(-0.5*NN_THETA)*input[0]^2
  // hidden[1] = tan(0.5*NN_THETA)*input[1]
  // hidden[2] = 0
  // output[0] = tan(0.5*NN_THETA)*hidden[0]+tan(0.5*NN_THETA)*hidden[1]
  // output[1] = tan(0.5*NN_THETA)*hidden[1]
  // output[2] = 0
  NNBasesSet(nn, 0, 0.5);
  NNBasesSet(nn, 3, -0.5);
  NNBasesSet(nn, 8, -0.5);
  short data[21] = {0,0,3, 1,0,3, 0,1,4, 0,3,6, 0,4,6, 0,4,7, -1,0,0};
  VecLong *links = VecLongCreate(21);
  for (int i = 21; i--;)
    VecSet(links, i, data[i]);
  NNSetLinks(nn, links);
  VecFree(&links);
  VecFloat3D input = VecFloatCreateStatic3D();
  VecFloat3D output = VecFloatCreateStatic3D();
  VecFloat3D check = VecFloatCreateStatic3D();
  VecFloat3D checkhidden = VecFloatCreateStatic3D();
  NNPrintln(nn, stdout);
  for (int i = -10; i <= 10; ++i) {
    for (int j = -10; j <= 10; ++j) {
      for (int k = -10; k <= 10; ++k) {
        VecSet(&input, 0, 0.1 * (float)i);
        VecSet(&input, 1, 0.1 * (float)j);
        VecSet(&input, 2, 0.1 * (float)k);
        NNEval(nn, (VecFloat*)&input, (VecFloat*)&output);
        VecSet(&checkhidden, 0, tan(0.5 * NN_THETA) * tan(-0.5 * NN_THETA) * fsquare(VecGet(&input, 0)));
        VecSet(&checkhidden, 1, tan(0.5 * NN_THETA) * VecGet(&input, 1));
        VecSet(&check, 0,
          tan(0.5 * NN_THETA) * (VecGet(&checkhidden, 0) + VecGet(&checkhidden, 1)));
        VecSet(&check, 1, tan(0.5 * NN_THETA) * VecGet(&checkhidden, 1));
        if (VecIsEqual(&output, &check) == false ||
          VecIsEqual(NNHiddenValues(nn), &checkhidden) == false) {
          NeuraNetErr->_type = PBErrTypeUnitTestFailed;
          sprintf(NeuraNetErr->_msg, "NNEval failed");
          PBErrCatch(NeuraNetErr);
        }
      }
    }
  }
  char* cloudUrl = "./cloud.txt";
  if (NNSaveLinkAsCloudGraph(nn, cloudUrl) == false) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNSaveLinkAsCloudGraph failed");
    PBErrCatch(NeuraNetErr);
  }
```

```
  if (ISEQUALF(NNGetHiddenValSimpsonDiv(nn), 0.285714) == false) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNGetHiddenValSimpsonDiv failed");
    PBErrCatch(NeuraNetErr);
  }
  NeuraNetFree(&nn);
  printf("UnitTestNeuraNetEvalPrintHiddenLinkSimpsonDiv OK\n");
}

#ifdef GENALG_H
float evaluate(const NeuraNet* const nn) {
  VecFloat3D input = VecFloatCreateStatic3D();
  VecFloat3D output = VecFloatCreateStatic3D();
  VecFloat3D check = VecFloatCreateStatic3D();
  float val = 0.0;
  int nb = 0;
  for (int i = -5; i <= 5; ++i) {
    for (int j = -5; j <= 5; ++j) {
      for (int k = -5; k <= 5; ++k) {
        VecSet(&input, 0, 0.2 * (float)i);
        VecSet(&input, 1, 0.2 * (float)j);
        VecSet(&input, 2, 0.2 * (float)k);
        NNEval(nn, (VecFloat*)&input, (VecFloat*)&output);
        VecSet(&check, 0,
          0.5 * (VecGet(&input, 1) - fsquare(VecGet(&input, 0))));
        VecSet(&check, 1, VecGet(&input, 1));
        val += VecDist(&output, &check);
        ++nb;
      }
    }
  }
  return -1.0 * val / (float)nb;
}

void UnitTestNeuraNetGA() {
  //srandom(RANDOMSEED);
  srandom(time(NULL));
  int nbIn = 3;
  int nbOut = 3;
  int nbHid = 3;
  int nbBase = 7;
  int nbLink = 7;
  NeuraNet* nn = NeuraNetCreate(nbIn, nbOut, nbHid, nbBase, nbLink);
  GenAlg* ga = GenAlgCreate(GENALG_NBENTITIES, GENALG_NBELITES,
    NNGetGAAdnFloatLength(nn), NNGetGAAdnIntLength(nn));
  NNSetGABoundsBases(nn, ga);
  NNSetGABoundsLinks(nn, ga);
  // Must be declared as a GenAlg applied to a NeuraNet or links will
  // get corrupted
  GASetTypeNeuraNet(ga, nbIn, nbHid, nbOut);
  GAInit(ga);
  float best = -1000000.0;
  float ev = 0.0;
  do {
    for (int iEnt = GAGetNbAdns(ga); iEnt--;) {
      if (GAAdnIsNew(GAAdn(ga, iEnt))) {
        NNSetBases(nn, GAAdnAdnF(GAAdn(ga, iEnt)));
        NNSetLinks(nn, GAAdnAdnI(GAAdn(ga, iEnt)));
        float value = evaluate(nn);
        GASetAdnValue(ga, GAAdn(ga, iEnt), value);
      }
    }
```

```
    GAStep(ga);
    NNSetBases(nn, GABestAdnF(ga));
    NNSetLinks(nn, GABestAdnI(ga));
    ev = evaluate(nn);
    if (ev > best + PBMATH_EPSILON) {
      best = ev;
      printf("%lu %f\n", GAGetCurEpoch(ga), best);
      fflush(stdout);
    }
  } while (GAGetCurEpoch(ga) < 30000 && fabs(ev) > 0.001);
  //} while (GAGetCurEpoch(ga) < 10 && fabs(ev) > 0.001);
  printf("best after %lu epochs: %f \n", GAGetCurEpoch(ga), best);
  NNPrintln(nn, stdout);
  FILE* fd = fopen("./bestnn.txt", "w");
  NNSave(nn, fd, false);
  fclose(fd);
  NeuraNetFree(&nn);
  GenAlgFree(&ga);
  printf("UnitTestNeuraNetGA OK\n");
}
#endif

void UnitTestNeuraNet() {
  UnitTestNeuraNetCreateFree();
  UnitTestNeuraNetCreateFullyConnected();
  UnitTestNeuraNetCreateConvolution();
  UnitTestNeuraNetGetSet();
  UnitTestNeuraNetSaveLoadPrune();
  UnitTestNeuraNetEvalPrintHiddenLinkSimpsonDiv();
#ifdef GENALG_H
  UnitTestNeuraNetGA();
#endif

  printf("UnitTestNeuraNet OK\n");
}

void UnitTestAll() {
  UnitTestNNBaseFun();
  UnitTestNeuraNet();
  printf("UnitTestAll OK\n");
}

int main() {
  UnitTestAll();
  // Return success code
  return 0;
}
```

6 Unit tests output

```
UnitTestNNBaseFun OK
UnitTestNeuraNetCreateFree OK
UnitTestNeuraNetCreateFullyConnected OK
nbInput: 12
nbOutput: 2
nbHidden: 16
nbMaxBases: 24
nbMaxLinks: 72
```

```
bases: <0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0
links: <0,0,12,4,0,18,1,1,12,0,1,13,5,1,18,4,1,19,1,2,13,0,2,14,5,2,19,4,2,20,1,3,14,5,3,20,2,4,12,0,4,15,6,4,18,4,4
hidden values: <0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000,0.000>
UnitTestNeuraNetCreateConvolution OK
UnitTestNeuraNetGetSet OK
UnitTestNeuraNetSaveLoadPrune OK
nbInput: 3
nbOutput: 3
nbHidden: 3
nbMaxBases: 3
nbMaxLinks: 7
bases: <0.500,0.000,0.000,-0.500,0.000,0.000,0.000,0.000,-0.500>
links: <0,0,3,1,0,3,0,1,4,0,3,6,0,4,6,0,4,7,-1,0,0>
hidden values: <0.000,0.000,0.000>
UnitTestNeuraNetEvalPrintHiddenLinkSimpsonDiv OK
1 -2.439442
2 -0.610493
3 -0.523390
4 -0.427185
5 -0.385936
6 -0.384753
10 -0.384385
11 -0.348133
12 -0.261669
14 -0.238000
15 -0.182214
19 -0.177692
87 -0.169309
119 -0.166232
122 -0.166038
126 -0.165657
127 -0.165309
268 -0.163007
573 -0.162364
586 -0.160925
727 -0.153403
2276 -0.150986
2323 -0.150562
2333 -0.150498
2350 -0.150420
2358 -0.150385
2373 -0.150349
2376 -0.150311
2495 -0.149283
2500 -0.149251
2509 -0.149174
2514 -0.148692
2515 -0.142850
2609 -0.127797
2646 -0.127437
2662 -0.126345
2688 -0.113117
3251 -0.111485
3268 -0.108593
3275 -0.099425
3296 -0.099081
3307 -0.098814
5154 -0.045894
5200 -0.042697
5211 -0.039684
5230 -0.038729
7269 -0.023437
```

```
7277 -0.017808
8809 -0.012034
8859 -0.011546
16981 -0.009548
best after 30000 epochs: -0.009548
nbInput: 3
nbOutput: 3
nbHidden: 3
nbMaxBases: 7
nbMaxLinks: 7
bases: <0.496,-0.306,0.522,-0.169,-0.716,-0.685,0.263,-0.801,0.129,-0.809,0.491,-0.054,0.228,-0.771,0.330,0.022,-0.79
links: <2,0,5,3,0,5,6,1,4,6,1,6,0,1,7,5,4,7,4,5,6>
hidden values: <0.000,0.347,-1.000>
UnitTestNeuraNetGA OK
UnitTestNeuraNet OK
UnitTestAll OK
```

neuranet.txt:


```
{
  "_nbInputVal":"10",
  "_nbOutputVal":"20",
  "_nbMaxHidVal":"30",
  "_nbMaxBases":"4",
  "_nbMaxLinks":"5",
  "_bases":{
    "_dim":"12",
    "_val":["0.000000","0.010000","0.020000","0.030000","0.040000","0.050000","0.060000","0.070000","0.080000","0.090
  },
  "_links":{
    "_dim":"15",
    "_val":["1","1","12","2","2","35","2","15","20","3","15","20","-1","0","0"]
  }
}
```

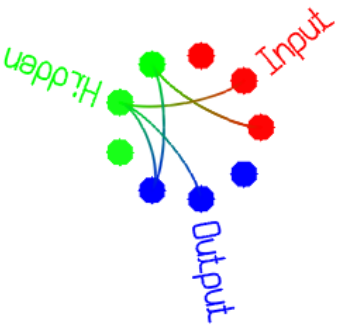

bestnn.txt:


```
{
  "_nbInputVal":"3",
  "_nbOutputVal":"3",
  "_nbMaxHidVal":"3",
  "_nbMaxBases":"7",
  "_nbMaxLinks":"7",
  "_bases":{
    "_dim":"21",
    "_val":["0.495885","-0.306034","0.521754","-0.168978","-0.715908","-0.685417","0.262754","-0.801479","0.128937","
  },
  "_links":{
    "_dim":"21",
    "_val":["2","0","5","3","0","5","6","1","4","6","1","6","0","1","7","5","4","7","4","5","6"]
  }
}
```


cloud.txt:

```
3
0 255 0 0 Input
1 0 255 0 Hidden
2 0 0 255 Output
```

```
9
0 0 0
1 0 1
2 0 2
3 1 3
4 1 4
5 1 5
6 2 6
7 2 7
8 2 8
6
0 3
0 3
1 4
3 6
4 6
4 7
```



7 Validation

7.1 Iris data set

Source: https://archive.ics.uci.edu/ml/datasets/iris

main.c:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "pberr.h"
#include "genalg.h"
#include "neuranet.h"
```

```
// https://archive.ics.uci.edu/ml/datasets/iris

// Nb of step between each save of the GenAlg
// Saving it allows to restart a stop learning process but is
// very time consuming if there are many input/hidden/output
// If 0 never save
#define SAVE_GA_EVERY 0
// Nb input and output of the NeuraNet
#define NB_INPUT 4
#define NB_OUTPUT 3
// Nb max of hidden values, links and base functions
#define NB_MAXHIDDEN 20
#define NB_MAXLINK 20
#define NB_MAXBASE NB_MAXLINK
// Size of the gene pool and elite pool
#define ADN_SIZE_POOL 100
#define ADN_SIZE_ELITE 20
// Initial best value during learning, must be lower than any
// possible value returned by Evaluate()
#define INIT_BEST_VAL 0.0
// Value of the NeuraNet above which the learning process stops
#define STOP_LEARNING_AT_VAL 0.999
// Number of epoch above which the learning process stops
#define STOP_LEARNING_AT_EPOCH 2000
// Save NeuraNet in compact format
#define COMPACT true

// Categories of data sets

typedef enum DataSetCat {
  unknownDataSet,
  datalearn,
  datatest,
  dataall
} DataSetCat;
#define NB_DATASET 4
const char* dataSetNames[NB_DATASET] = {
  "unknownDataSet", "datalearn", "datatest", "dataall"
  };

// Structure for the data set
typedef enum IrisCat {
  setosa, versicolor, virginica
} IrisCat;
const char* irisCatNames[3] = {
  "setosa", "versicolor", "virginica"
  };

typedef struct Iris {
  float _props[4];
  IrisCat _cat;
} Iris;

typedef struct DataSet {
  // Category of the data set
  DataSetCat _cat;
  // Number of sample
  int _nbSample;
  // Samples
  Iris* _samples;
} DataSet;
```

```
// Get the DataSetCat from its 'name'
DataSetCat GetCategoryFromName(const char* const name) {
  // Declare a variable to memorize the DataSetCat
  DataSetCat cat = unknownDataSet;
  // Search the dataset
  for (int iSet = NB_DATASET; iSet--;)
    if (strcmp(name, dataSetNames[iSet]) == 0)
      cat = iSet;
  // Return the category
  return cat;
}

// Load the data set of category 'cat' in the DataSet 'that'
// Return true on success, else false
bool DataSetLoad(DataSet* const that, const DataSetCat cat) {
  // Set the category
  that->_cat = cat;

  // Load the data according to 'cat'
  FILE* f = fopen("./bezdekIris.data", "r");
  if (f == NULL) {
    printf("Couldn't open the data set file\n");
    return false;
  }
  char buffer[500];
  int ret = 0;
  if (cat == datalearn) {
    that->_nbSample = 75;
    that->_samples =
      PBErrMalloc(NeuraNetErr, sizeof(Iris) * that->_nbSample);
    for (int iCat = 0; iCat < 3; ++iCat) {
      for (int iSample = 0; iSample < 25; ++iSample) {
        ret = fscanf(f, "%f,%f,%f,%f,%s",
          that->_samples[25 * iCat + iSample]._props,
          that->_samples[25 * iCat + iSample]._props + 1,
          that->_samples[25 * iCat + iSample]._props + 2,
          that->_samples[25 * iCat + iSample]._props + 3,
          buffer);
        if (ret == EOF) {
          printf("Couldn't read the dataset\n");
          fclose(f);
          return false;
        }
        that->_samples[25 * iCat + iSample]._cat = (IrisCat)iCat;
      }
      for (int iSample = 0; iSample < 25; ++iSample) {
        ret = fscanf(f, "%s\n", buffer);
        if (ret == EOF) {
          printf("Couldn't read the dataset\n");
          fclose(f);
          return false;
        }
      }
    }
  } else if (cat == datatest) {
    that->_nbSample = 75;
    that->_samples =
      PBErrMalloc(NeuraNetErr, sizeof(Iris) * that->_nbSample);
    for (int iCat = 0; iCat < 3; ++iCat) {
      for (int iSample = 0; iSample < 25; ++iSample) {
        ret = fscanf(f, "%s\n", buffer);
```

```
        if (ret == EOF) {
          printf("Couldn't read the dataset\n");
          fclose(f);
          return false;
        }
      }
      for (int iSample = 0; iSample < 25; ++iSample) {
        ret = fscanf(f, "%f,%f,%f,%f,%s",
          that->_samples[25 * iCat + iSample]._props,
          that->_samples[25 * iCat + iSample]._props + 1,
          that->_samples[25 * iCat + iSample]._props + 2,
          that->_samples[25 * iCat + iSample]._props + 3,
          buffer);
        if (ret == EOF) {
          printf("Couldn't read the dataset\n");
          fclose(f);
          return false;
        }
        that->_samples[25 * iCat + iSample]._cat = (IrisCat)iCat;
      }
    }
  } else if (cat == dataall) {
    that->_nbSample = 150;
    that->_samples =
      PBErrMalloc(NeuraNetErr, sizeof(Iris) * that->_nbSample);
    for (int iCat = 0; iCat < 3; ++iCat) {
      for (int iSample = 0; iSample < 50; ++iSample) {
        ret = fscanf(f, "%f,%f,%f,%f,%s",
          that->_samples[50 * iCat + iSample]._props,
          that->_samples[50 * iCat + iSample]._props + 1,
          that->_samples[50 * iCat + iSample]._props + 2,
          that->_samples[50 * iCat + iSample]._props + 3,
          buffer);
        if (ret == EOF) {
          printf("Couldn't read the dataset\n");
          fclose(f);
          return false;
        }
        that->_samples[50 * iCat + iSample]._cat = (IrisCat)iCat;
      }
    }
  } else {
    printf("Invalid dataset\n");
    fclose(f);
    return false;
  }
  fclose(f);

  // Return success code
  return true;
}

// Free memory for the DataSet 'that'
void DataSetFree(DataSet** that) {
  if (*that == NULL) return;
  // Free the memory
  free((*that)->_samples);
  free(*that);
  *that = NULL;
}

// Evalutation function for the NeuraNet 'that' on the DataSet 'dataset'
```

```
// Return the value of the NeuraNet, the bigger the better
float Evaluate(const NeuraNet* const that,
  const DataSet* const dataset) {
  // Declare 2 vectors to memorize the input and output values
  VecFloat* input = VecFloatCreate(NNGetNbInput(that));
  VecFloat* output = VecFloatCreate(NNGetNbOutput(that));
  // Declare a variable to memorize the value
  float val = 0.0;

  // Evaluate

  for (int iSample = dataset->_nbSample; iSample--;) {
    for (int iInp = 0; iInp < NNGetNbInput(that); ++iInp) {
      VecSet(input, iInp,
        dataset->_samples[iSample]._props[iInp]);
    }
    NNEval(that, input, output);
    int pred = VecGetIMaxVal(output);
    if (dataset->_cat == datatest) {
      printf("#%d pred%d real%d ", iSample, pred,
        dataset->_samples[iSample]._cat);
      VecPrint(output, stdout);
    }
    if ((IrisCat)pred == dataset->_samples[iSample]._cat) {
      if (dataset->_cat == datatest)
        printf(" OK\n");
      val += 1.0;
    } else {
      if (dataset->_cat == datatest)
        printf(" NG\n");
    }

  }
  val /= (float)(dataset->_nbSample);

  // Free memory
  VecFree(&input);
  VecFree(&output);
  // Return the result of the evaluation
  return val;
}

// Create the NeuraNet
NeuraNet* createNN(void) {
  // Create the NeuraNet
  int nbIn = NB_INPUT;
  int nbOut = NB_OUTPUT;
  int nbMaxHid = NB_MAXHIDDEN;
  int nbMaxLink = NB_MAXLINK;
  int nbMaxBase = NB_MAXBASE;
  NeuraNet* nn =
    NeuraNetCreate(nbIn, nbOut, nbMaxHid, nbMaxBase, nbMaxLink);

  // Return the NeuraNet
  return nn;
}

// Learn based on the SataSetCat 'cat'
void Learn(DataSetCat cat) {
  // Init the random generator
  srandom(time(NULL));
  // Declare variables to measure time
```

```
struct timespec start, stop;
// Start measuring time
clock_gettime(CLOCK_REALTIME, &start);
// Load the DataSet
DataSet* dataset = PBErrMalloc(NeuraNetErr, sizeof(DataSet));
bool ret = DataSetLoad(dataset, cat);
if (!ret) {
  printf("Couldn't load the data\n");
  return;
}
// Create the NeuraNet
NeuraNet* nn = createNN();
// Declare a variable to memorize the best value
float bestVal = INIT_BEST_VAL;
// Declare a variable to memorize the limit in term of epoch
unsigned long int limitEpoch = STOP_LEARNING_AT_EPOCH;
// Create the GenAlg used for learning
// If previous weights are available in "./bestga.txt" reload them
GenAlg* ga = NULL;
FILE* fd = fopen("./bestga.txt", "r");
if (fd) {
  printf("Reloading previous GenAlg...\n");
  if (!GALoad(&ga, fd)) {
    printf("Failed to reload the GenAlg.\n");
    NeuraNetFree(&nn);
    DataSetFree(&dataset);
    return;
  } else {
    printf("Previous GenAlg reloaded.\n");
    if (GABestAdnF(ga) != NULL)
      NNSetBases(nn, GABestAdnF(ga));
    if (GABestAdnI(ga) != NULL)
      NNSetLinks(nn, GABestAdnI(ga));
    bestVal = Evaluate(nn, dataset);
    printf("Starting with best at %f.\n", bestVal);
    limitEpoch += GAGetCurEpoch(ga);
  }
  fclose(fd);
} else {
  ga = GenAlgCreate(ADN_SIZE_POOL, ADN_SIZE_ELITE,
    NNGetGAAdnFloatLength(nn), NNGetGAAdnIntLength(nn));
  NNSetGABoundsBases(nn, ga);
  NNSetGABoundsLinks(nn, ga);
  // Must be declared as a GenAlg applied to a NeuraNet or links will
  // get corrupted
  GASetTypeNeuraNet(ga, NB_INPUT, NB_MAXHIDDEN, NB_OUTPUT);
  GAInit(ga);
}
// If there is a NeuraNet available, reload it into the GenAlg
fd = fopen("./bestnn.txt", "r");
if (fd) {
  printf("Reloading previous NeuraNet...\n");
  if (!NNLoad(&nn, fd)) {
    printf("Failed to reload the NeuraNet.\n");
    NeuraNetFree(&nn);
    DataSetFree(&dataset);
    return;
  } else {
    printf("Previous NeuraNet reloaded.\n");
    bestVal = Evaluate(nn, dataset);
    printf("Starting with best at %f.\n", bestVal);
    GenAlgAdn* adn = GAAdn(ga, 0);
```

```
    VecCopy(adn->_adnF, nn->_bases);
    VecCopy(adn->_adnI, nn->_links);
  }
  fclose(fd);
}
// Start learning process
printf("Learning...\n");
printf("Will stop when curEpoch >= %lu or bestVal >= %f\n",
  limitEpoch, STOP_LEARNING_AT_VAL);
printf("Will save the best NeuraNet in ./bestnn.txt at each improvement\n");
fflush(stdout);
// Declare a variable to memorize the best value in the current epoch
float curBest = 0.0;
float curWorst = 0.0;
// Declare a variable to manage the save of GenAlg
int delaySave = 0;
// Learning loop
while (bestVal < STOP_LEARNING_AT_VAL &&
  GAGetCurEpoch(ga) < limitEpoch) {
  curWorst = curBest;
  curBest = INIT_BEST_VAL;
  int curBestI = 0;
  unsigned long int ageBest = 0;
  // For each adn in the GenAlg
  //for (int iEnt = GAGetNbAdns(ga); iEnt--;) {
  for (int iEnt = 0; iEnt < GAGetNbAdns(ga); ++iEnt) {
    // Get the adn
    GenAlgAdn* adn = GAAdn(ga, iEnt);
    // Set the links and base functions of the NeuraNet according
    // to this adn
    if (GABestAdnF(ga) != NULL)
      NNSetBases(nn, GAAdnAdnF(adn));
    if (GABestAdnI(ga) != NULL)
      NNSetLinks(nn, GAAdnAdnI(adn));
    // Evaluate the NeuraNet
    float value = Evaluate(nn, dataset);
    // Update the value of this adn
    GASetAdnValue(ga, adn, value);
    // Update the best value in the current epoch
    if (value > curBest) {
      curBest = value;
      curBestI = iEnt;
      ageBest = GAAdnGetAge(adn);
    }
    if (value < curWorst)
      curWorst = value;
  }
  // Measure time
  clock_gettime(CLOCK_REALTIME, &stop);
  float elapsed = stop.tv_sec - start.tv_sec;
  int day = (int)floor(elapsed / 86400);
  elapsed -= (float)(day * 86400);
  int hour = (int)floor(elapsed / 3600);
  elapsed -= (float)(hour * 3600);
  int min = (int)floor(elapsed / 60);
  elapsed -= (float)(min * 60);
  int sec = (int)floor(elapsed);
  // If there has been improvement during this epoch
  if (curBest > bestVal) {
    bestVal = curBest;
    // Display info about the improvment
    printf("Improvement at epoch %05lu: %f(%03d) (in %02d:%02d:%02d:%02ds)       \n",
```

```
      GAGetCurEpoch(ga), bestVal, curBestI, day, hour, min, sec);
    fflush(stdout);
    // Set the links and base functions of the NeuraNet according
    // to the best adn
    GenAlgAdn* bestAdn = GAAdn(ga, curBestI);
    if (GAAdnAdnF(bestAdn) != NULL)
      NNSetBases(nn, GAAdnAdnF(bestAdn));
    if (GAAdnAdnI(bestAdn) != NULL)
      NNSetLinks(nn, GAAdnAdnI(bestAdn));
    // Save the best NeuraNet
    fd = fopen("./bestnn.txt", "w");
    if (!NNSave(nn, fd, COMPACT)) {
      printf("Couldn't save the NeuraNet\n");
      NeuraNetFree(&nn);
      GenAlgFree(&ga);
      DataSetFree(&dataset);
      return;
    }
    fclose(fd);
  } else {
    fprintf(stderr,
      "Epoch %05lu: v%f a%03lu(%03d) kt%03lu ",
      GAGetCurEpoch(ga), curBest, ageBest, curBestI,
      GAGetNbKTEvent(ga));
    fprintf(stderr, "(in %02d:%02d:%02d:%02ds)  \r",
      day, hour, min, sec);
    fflush(stderr);
  }
  ++delaySave;
  if (SAVE_GA_EVERY != 0 && delaySave >= SAVE_GA_EVERY) {
    delaySave = 0;
    // Save the adns of the GenAlg, use a temporary file to avoid
    // loosing the previous one if something goes wrong during
    // writing, then replace the previous file with the temporary one
    fd = fopen("./bestga.tmp", "w");
    if (!GASave(ga, fd, COMPACT)) {
      printf("Couldn't save the GenAlg\n");
      NeuraNetFree(&nn);
      GenAlgFree(&ga);
      DataSetFree(&dataset);
      return;
    }
    fclose(fd);
    int ret = system("mv ./bestga.tmp ./bestga.txt");
    (void)ret;
  }
  // Step the GenAlg
  GAStep(ga);
}
// Measure time
clock_gettime(CLOCK_REALTIME, &stop);
float elapsed = stop.tv_sec - start.tv_sec;
int day = (int)floor(elapsed / 86400);
elapsed -= (float)(day * 86400);
int hour = (int)floor(elapsed / 3600);
elapsed -= (float)(hour * 3600);
int min = (int)floor(elapsed / 60);
elapsed -= (float)(min * 60);
int sec = (int)floor(elapsed);
printf("\nLearning complete (in %d:%d:%d:%ds)\n",
  day, hour, min, sec);
// Free memory
```

```
  NeuraNetFree(&nn);
  GenAlgFree(&ga);
  DataSetFree(&dataset);
}

// Check the NeuraNet 'that' on the DataSetCat 'cat'
void Validate(const NeuraNet* const that, const DataSetCat cat) {
  // Load the DataSet
  DataSet* dataset = PBErrMalloc(NeuraNetErr, sizeof(DataSet));
  bool ret = DataSetLoad(dataset, cat);
  if (!ret) {
    printf("Couldn't load the data\n");
    return;
  }
  // Evaluate the NeuraNet
  float value = Evaluate(that, dataset);
  // Display the result
  printf("Value: %.6f\n", value);
  // Free memory
  DataSetFree(&dataset);
}

// Predict using the NeuraNet 'that' on 'inputs' (given as an array of
// 'nbInp' char*)
void Predict(const NeuraNet* const that, const int nbInp,
  char** const inputs) {
  // Start measuring time
  clock_t clockStart = clock();
  // Check the number of inputs
  if (nbInp != NNGetNbInput(that)) {
    printf("Wrong number of inputs, there should %d, there was %d\n",
      NNGetNbInput(that), nbInp);
    return;
  }
  // Declare 2 vectors to memorize the input and output values
  VecFloat* input = VecFloatCreate(NNGetNbInput(that));
  VecFloat* output = VecFloatCreate(NNGetNbOutput(that));
  // Set the input
  for (int iInp = 0; iInp < nbInp; ++iInp) {
    float v = 0.0;
    sscanf(inputs[iInp], "%f", &v);
    VecSet(input, iInp, v);
  }
  // Predict
  NNEval(that, input, output);
  int pred = -1;
  if (VecGet(output, 0) > VecGet(output, 1) &&
    VecGet(output, 0) > VecGet(output, 2))
    pred = 0;
  else if (VecGet(output, 1) > VecGet(output, 0) &&
    VecGet(output, 1) > VecGet(output, 2))
    pred = 1;
  else if (VecGet(output, 2) > VecGet(output, 1) &&
    VecGet(output, 2) > VecGet(output, 0))
    pred = 2;
  // End measuring time
  clock_t clockEnd = clock();
  double timeUsed =
    ((double)(clockEnd - clockStart)) / (CLOCKS_PER_SEC * 0.001) ;
  // If the clock has been reset meanwhile
  if (timeUsed < 0.0)
    timeUsed = 0.0;
```

```
  printf("Prediction: %s (in %fms)\n", irisCatNames[pred], timeUsed);

  // Free memory
  VecFree(&input);
  VecFree(&output);
}

int main(int argc, char** argv) {
  // Declare a variable to memorize the mode (learning/checking)
  int mode = -1;
  // Declare a variable to memorize the dataset used
  DataSetCat cat = unknownDataSet;
  // Decode mode from arguments
  if (argc >= 3) {
    if (strcmp(argv[1], "-learn") == 0) {
      mode = 0;
      cat = GetCategoryFromName(argv[2]);
    } else if (strcmp(argv[1], "-check") == 0) {
      mode = 1;
      cat = GetCategoryFromName(argv[2]);
    } else if (strcmp(argv[1], "-predict") == 0) {
      mode = 2;
    }
  }
  // If the mode is invalid print some help
  if (mode == -1) {
    printf("Select a mode from:\n");
    printf("-learn <dataset name>\n");
    printf("-check <dataset name>\n");
    printf("-predict <input values>\n");
    return 0;
  }
  if (mode == 0) {
    Learn(cat);
  } else if (mode == 1) {
    NeuraNet* nn = NULL;
    FILE* fd = fopen("./bestnn.txt", "r");
    if (!NNLoad(&nn, fd)) {
      printf("Couldn't load the best NeuraNet\n");
      return 0;
    }
    fclose(fd);
    Validate(nn, cat);
    NeuraNetFree(&nn);
  } else if (mode == 2) {
    NeuraNet* nn = NULL;
    FILE* fd = fopen("./bestnn.txt", "r");
    if (!NNLoad(&nn, fd)) {
      printf("Couldn't load the best NeuraNet\n");
      return 0;
    }
    fclose(fd);
    Predict(nn, argc - 2, argv + 2);
    NeuraNetFree(&nn);
  }
  // Return success code
  return 0;
}
```

7.2 Abalone data set

Source: http://www.cs.toronto.edu/ delve/data/abalone/desc.html

main.c:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "pberr.h"
#include "genalg.h"
#include "neuranet.h"

// http://www.cs.toronto.edu/~delve/data/abalone/desc.html

// Nb of step between each save of the GenAlg
// Saving it allows to restart a stop learning process but is
// very time consuming if there are many input/hidden/output
// If 0 never save
#define SAVE_GA_EVERY 0
// Nb input and output of the NeuraNet
#define NB_INPUT 10
#define NB_OUTPUT 1
// Nb max of hidden values, links and base functions
#define NB_MAXHIDDEN 50
#define NB_MAXLINK 100
#define NB_MAXBASE NB_MAXLINK
// Size of the gene pool and elite pool
#define ADN_SIZE_POOL 500
#define ADN_SIZE_ELITE 20
// Initial best value during learning, must be lower than any
// possible value returned by Evaluate()
#define INIT_BEST_VAL -10000.0
// Value of the NeuraNet above which the learning process stops
#define STOP_LEARNING_AT_VAL -0.01
// Number of epoch above which the learning process stops
#define STOP_LEARNING_AT_EPOCH 5000
// Save NeuraNet in compact format
#define COMPACT true

// Categories of data sets

typedef enum DataSetCat {
  unknownDataSet,
  datalearn,
  datatest,
  dataall
} DataSetCat;
#define NB_DATASET 4
const char* dataSetNames[NB_DATASET] = {
  "unknownDataSet", "datalearn", "datatest", "dataall"
  };

// Structure for the data set

typedef struct Abalone {
```

```
  float _props[10];
  float _age;
} Abalone;

typedef struct DataSet {
  // Category of the data set
  DataSetCat _cat;
  // Number of sample
  int _nbSample;
  // Samples
  Abalone* _samples;
  float _weights[29];
} DataSet;

// Get the DataSetCat from its 'name'
DataSetCat GetCategoryFromName(const char* const name) {
  // Declare a variable to memorize the DataSetCat
  DataSetCat cat = unknownDataSet;
  // Search the dataset
  for (int iSet = NB_DATASET; iSet--;)
    if (strcmp(name, dataSetNames[iSet]) == 0)
      cat = iSet;
  // Return the category
  return cat;
}

// Load the data set of category 'cat' in the DataSet 'that'
// Return true on success, else false
bool DataSetLoad(DataSet* const that, const DataSetCat cat) {
  // Set the category
  that->_cat = cat;

  // Load the data according to 'cat'
  FILE* f = fopen("./Prototask.data", "r");
  if (f == NULL) {
    printf("Couldn't open the data set file\n");
    return false;
  }
  char sex;
  int age;
  int ret = 0;
  if (cat == datalearn) {
    that->_nbSample = 3000;
    that->_samples =
      PBErrMalloc(NeuraNetErr, sizeof(Abalone) * that->_nbSample);
    for (int iSample = 0; iSample < that->_nbSample; ++iSample) {
      ret = fscanf(f, "%c %f %f %f %f %f %f %f %d\n",
        &sex,
        that->_samples[iSample]._props + 3,
        that->_samples[iSample]._props + 4,
        that->_samples[iSample]._props + 5,
        that->_samples[iSample]._props + 6,
        that->_samples[iSample]._props + 7,
        that->_samples[iSample]._props + 8,
        that->_samples[iSample]._props + 9,
        &age);
      if (ret == EOF) {
        printf("Couldn't read the dataset\n");
        fclose(f);
        return false;
      }
      that->_samples[iSample]._age = (float)age;
```

```
    if (sex == 'M') {
      that->_samples[iSample]._props[0] = 1.0;
      that->_samples[iSample]._props[1] = -1.0;
      that->_samples[iSample]._props[2] = -1.0;
    } else if (sex == 'F') {
      that->_samples[iSample]._props[0] = -1.0;
      that->_samples[iSample]._props[1] = 1.0;
      that->_samples[iSample]._props[2] = -1.0;
    } else if (sex == 'I') {
      that->_samples[iSample]._props[0] = -1.0;
      that->_samples[iSample]._props[1] = -1.0;
      that->_samples[iSample]._props[2] = 1.0;
    }
  }
} else if (cat == datatest) {
  for (int iSample = 0; iSample < 3000; ++iSample) {
    float dummy;
    ret = fscanf(f, "%c %f %f %f %f %f %f %f %d\n",
      &sex,
      &dummy,
      &dummy,
      &dummy,
      &dummy,
      &dummy,
      &dummy,
      &dummy,
      &age);
    (void)dummy;
    if (ret == EOF) {
      printf("Couldn't read the dataset\n");
      fclose(f);
      return false;
    }
  }
  that->_nbSample = 1177;
  that->_samples =
    PBErrMalloc(NeuraNetErr, sizeof(Abalone) * that->_nbSample);
  for (int iSample = 0; iSample < that->_nbSample; ++iSample) {
    ret = fscanf(f, "%c %f %f %f %f %f %f %f %d\n",
      &sex,
      that->_samples[iSample]._props + 3,
      that->_samples[iSample]._props + 4,
      that->_samples[iSample]._props + 5,
      that->_samples[iSample]._props + 6,
      that->_samples[iSample]._props + 7,
      that->_samples[iSample]._props + 8,
      that->_samples[iSample]._props + 9,
      &age);
    if (ret == EOF) {
      printf("Couldn't read the dataset\n");
      fclose(f);
      return false;
    }
    that->_samples[iSample]._age = (float)age;
    if (sex == 'M') {
      that->_samples[iSample]._props[0] = 1.0;
      that->_samples[iSample]._props[1] = -1.0;
      that->_samples[iSample]._props[2] = -1.0;
    } else if (sex == 'F') {
      that->_samples[iSample]._props[0] = -1.0;
      that->_samples[iSample]._props[1] = 1.0;
      that->_samples[iSample]._props[2] = -1.0;
```

```
    } else if (sex == 'I') {
      that->_samples[iSample]._props[0] = -1.0;
      that->_samples[iSample]._props[1] = -1.0;
      that->_samples[iSample]._props[2] = 1.0;
    }
  }
} else if (cat == dataall) {
  that->_nbSample = 4177;
  that->_samples =
    PBErrMalloc(NeuraNetErr, sizeof(Abalone) * that->_nbSample);
  for (int iSample = 0; iSample < that->_nbSample; ++iSample) {
    ret = fscanf(f, "%c %f %f %f %f %f %f %f %d\n",
      &sex,
      that->_samples[iSample]._props + 3,
      that->_samples[iSample]._props + 4,
      that->_samples[iSample]._props + 5,
      that->_samples[iSample]._props + 6,
      that->_samples[iSample]._props + 7,
      that->_samples[iSample]._props + 8,
      that->_samples[iSample]._props + 9,
      &age);
    if (ret == EOF) {
      printf("Couldn't read the dataset\n");
      fclose(f);
      return false;
    }
    that->_samples[iSample]._age = (float)age;
    if (sex == 'M') {
      that->_samples[iSample]._props[0] = 1.0;
      that->_samples[iSample]._props[1] = -1.0;
      that->_samples[iSample]._props[2] = -1.0;
    } else if (sex == 'F') {
      that->_samples[iSample]._props[0] = -1.0;
      that->_samples[iSample]._props[1] = 1.0;
      that->_samples[iSample]._props[2] = -1.0;
    } else if (sex == 'I') {
      that->_samples[iSample]._props[0] = -1.0;
      that->_samples[iSample]._props[1] = -1.0;
      that->_samples[iSample]._props[2] = 1.0;
    }
  }
} else {
  printf("Invalid dataset\n");
  fclose(f);
  return false;
}
fclose(f);

for (int iCat = 29; iCat--;)
  that->_weights[iCat] = 0.0;
for (int iSample = that->_nbSample; iSample--;) {
  int cat = (int)round(that->_samples[iSample]._age) - 1;
  if (cat < 0 || cat >= 29) {
    printf("Invalid age #%d %f\n", iSample,
      that->_samples[iSample]._age);
    return false;
  }
  that->_weights[cat] += 1.0;
}
for (int iCat = 29; iCat--;)
  that->_weights[iCat] =
    ((float)(that->_nbSample) - that->_weights[iCat]) /
```

```
      (float)(that->_nbSample);

  // Return success code
  return true;
}

// Free memory for the DataSet 'that'
void DataSetFree(DataSet** that) {
  if (*that == NULL) return;
  // Free the memory
  free((*that)->_samples);
  free(*that);
  *that = NULL;
}

// Evalutation function for the NeuraNet 'that' on the DataSet 'dataset'
// Return the value of the NeuraNet, the bigger the better
float Evaluate(const NeuraNet* const that,
  const DataSet* const dataset) {
  // Declare 2 vectors to memorize the input and output values
  VecFloat* input = VecFloatCreate(NNGetNbInput(that));
  VecFloat* output = VecFloatCreate(NNGetNbOutput(that));
  // Declare a variable to memorize the value
  float val = 0.0;

  // Evaluate

  int count[29] = {0};
  for (int iSample = dataset->_nbSample; iSample--;) {
    for (int iInp = 0; iInp < NNGetNbInput(that); ++iInp) {
      VecSet(input, iInp,
        dataset->_samples[iSample]._props[iInp]);
    }
    NNEval(that, input, output);

    float pred = VecGet(output, 0);
    float age = dataset->_samples[iSample]._age + 0.5;
    float v = fabs(pred - age);
    val -= v;
    if (dataset->_cat != datalearn) {
      int iErr = (int)round(v);
      ++(count[iErr]);
    }

  }
  val /= (float)(dataset->_nbSample);
  if (dataset->_cat != datalearn) {
    float perc = 0.0;
    printf("age_err count cumul_perc\n");
    for (int iErr = 0; iErr < 29; ++ iErr) {
      perc += (float)(count[iErr]) / (float)(dataset->_nbSample);
      printf("%2d %4d %f\n", iErr, count[iErr], perc);
    }
  }

  // Free memory
  VecFree(&input);
  VecFree(&output);
  // Return the result of the evaluation
  return val;
}
```

```
// Create the NeuraNet
NeuraNet* createNN(void) {
  // Create the NeuraNet
  int nbIn = NB_INPUT;
  int nbOut = NB_OUTPUT;
  int nbMaxHid = NB_MAXHIDDEN;
  int nbMaxLink = NB_MAXLINK;
  int nbMaxBase = NB_MAXBASE;
  NeuraNet* nn =
    NeuraNetCreate(nbIn, nbOut, nbMaxHid, nbMaxBase, nbMaxLink);

  // Return the NeuraNet
  return nn;
}

// Learn based on the SataSetCat 'cat'
void Learn(DataSetCat cat) {
  // Init the random generator
  srandom(time(NULL));
  // Declare variables to measure time
  struct timespec start, stop;
  // Start measuring time
  clock_gettime(CLOCK_REALTIME, &start);
  // Load the DataSet
  DataSet* dataset = PBErrMalloc(NeuraNetErr, sizeof(DataSet));
  bool ret = DataSetLoad(dataset, cat);
  if (!ret) {
    printf("Couldn't load the data\n");
    return;
  }
  // Create the NeuraNet
  NeuraNet* nn = createNN();
  // Declare a variable to memorize the best value
  float bestVal = INIT_BEST_VAL;
  // Declare a variable to memorize the limit in term of epoch
  unsigned long int limitEpoch = STOP_LEARNING_AT_EPOCH;
  // Create the GenAlg used for learning
  // If previous weights are available in "./bestga.txt" reload them
  GenAlg* ga = NULL;
  FILE* fd = fopen("./bestga.txt", "r");
  if (fd) {
    printf("Reloading previous GenAlg...\n");
    if (!GALoad(&ga, fd)) {
      printf("Failed to reload the GenAlg.\n");
      NeuraNetFree(&nn);
      DataSetFree(&dataset);
      return;
    } else {
      printf("Previous GenAlg reloaded.\n");
      if (GABestAdnF(ga) != NULL)
        NNSetBases(nn, GABestAdnF(ga));
      if (GABestAdnI(ga) != NULL)
        NNSetLinks(nn, GABestAdnI(ga));
      bestVal = Evaluate(nn, dataset);
      printf("Starting with best at %f.\n", bestVal);
      limitEpoch += GAGetCurEpoch(ga);
    }
    fclose(fd);
  } else {
    ga = GenAlgCreate(ADN_SIZE_POOL, ADN_SIZE_ELITE,
      NNGetGAAdnFloatLength(nn), NNGetGAAdnIntLength(nn));
    NNSetGABoundsBases(nn, ga);
```

```
  NNSetGABoundsLinks(nn, ga);
  // Must be declared as a GenAlg applied to a NeuraNet or links will
  // get corrupted
  GASetTypeNeuraNet(ga, NB_INPUT, NB_MAXHIDDEN, NB_OUTPUT);
  GAInit(ga);
}
// If there is a NeuraNet available, reload it into the GenAlg
fd = fopen("./bestnn.txt", "r");
if (fd) {
  printf("Reloading previous NeuraNet...\n");
  if (!NNLoad(&nn, fd)) {
    printf("Failed to reload the NeuraNet.\n");
    NeuraNetFree(&nn);
    DataSetFree(&dataset);
    return;
  } else {
    printf("Previous NeuraNet reloaded.\n");
    bestVal = Evaluate(nn, dataset);
    printf("Starting with best at %f.\n", bestVal);
    GenAlgAdn* adn = GAAdn(ga, 0);
    VecCopy(adn->_adnF, nn->_bases);
    VecCopy(adn->_adnI, nn->_links);
  }
  fclose(fd);
}
// Start learning process
printf("Learning...\n");
printf("Will stop when curEpoch >= %lu or bestVal >= %f\n",
  limitEpoch, STOP_LEARNING_AT_VAL);
printf("Will save the best NeuraNet in ./bestnn.txt at each improvement\n");
fflush(stdout);
// Declare a variable to memorize the best value in the current epoch
float curBest = 0.0;
float curWorst = 0.0;
// Declare a variable to manage the save of GenAlg
int delaySave = 0;
// Learning loop
while (bestVal < STOP_LEARNING_AT_VAL &&
  GAGetCurEpoch(ga) < limitEpoch) {
  curWorst = curBest;
  curBest = INIT_BEST_VAL;
  int curBestI = 0;
  unsigned long int ageBest = 0;
  // For each adn in the GenAlg
  //for (int iEnt = GAGetNbAdns(ga); iEnt--;) {
  for (int iEnt = 0; iEnt < GAGetNbAdns(ga); ++iEnt) {
    // Get the adn
    GenAlgAdn* adn = GAAdn(ga, iEnt);
    // Set the links and base functions of the NeuraNet according
    // to this adn
    if (GABestAdnF(ga) != NULL)
      NNSetBases(nn, GAAdnAdnF(adn));
    if (GABestAdnI(ga) != NULL)
      NNSetLinks(nn, GAAdnAdnI(adn));
    // Evaluate the NeuraNet
    float value = Evaluate(nn, dataset);
    // Update the value of this adn
    GASetAdnValue(ga, adn, value);
    // Update the best value in the current epoch
    if (value > curBest) {
      curBest = value;
      curBestI = iEnt;
```

```
    ageBest = GAAdnGetAge(adn);
  }
  if (value < curWorst)
    curWorst = value;
}
// Measure time
clock_gettime(CLOCK_REALTIME, &stop);
float elapsed = stop.tv_sec - start.tv_sec;
int day = (int)floor(elapsed / 86400);
elapsed -= (float)(day * 86400);
int hour = (int)floor(elapsed / 3600);
elapsed -= (float)(hour * 3600);
int min = (int)floor(elapsed / 60);
elapsed -= (float)(min * 60);
int sec = (int)floor(elapsed);
// If there has been improvement during this epoch
if (curBest > bestVal) {
  bestVal = curBest;
  // Display info about the improvment
  printf("Improvement at epoch %05lu: %f(%03d) (in %02d:%02d:%02d:%02ds)       \n",
    GAGetCurEpoch(ga), bestVal, curBestI, day, hour, min, sec);
  fflush(stdout);
  // Set the links and base functions of the NeuraNet according
  // to the best adn
  GenAlgAdn* bestAdn = GAAdn(ga, curBestI);
  if (GAAdnAdnF(bestAdn) != NULL)
    NNSetBases(nn, GAAdnAdnF(bestAdn));
  if (GAAdnAdnI(bestAdn) != NULL)
    NNSetLinks(nn, GAAdnAdnI(bestAdn));
  // Save the best NeuraNet
  fd = fopen("./bestnn.txt", "w");
  if (!NNSave(nn, fd, COMPACT)) {
    printf("Couldn't save the NeuraNet\n");
    NeuraNetFree(&nn);
    GenAlgFree(&ga);
    DataSetFree(&dataset);
    return;
  }
  fclose(fd);
} else {
  fprintf(stderr,
    "Epoch %05lu: v%f a%03lu(%03d) kt%03lu ",
    GAGetCurEpoch(ga), curBest, ageBest, curBestI,
    GAGetNbKTEvent(ga));
  fprintf(stderr, "(in %02d:%02d:%02d:%02ds)  \r",
    day, hour, min, sec);
  fflush(stderr);
}
++delaySave;
if (SAVE_GA_EVERY != 0 && delaySave >= SAVE_GA_EVERY) {
  delaySave = 0;
  // Save the adns of the GenAlg, use a temporary file to avoid
  // loosing the previous one if something goes wrong during
  // writing, then replace the previous file with the temporary one
  fd = fopen("./bestga.tmp", "w");
  if (!GASave(ga, fd, COMPACT)) {
    printf("Couldn't save the GenAlg\n");
    NeuraNetFree(&nn);
    GenAlgFree(&ga);
    DataSetFree(&dataset);
    return;
  }
```

```
      fclose(fd);
      int ret = system("mv ./bestga.tmp ./bestga.txt");
      (void)ret;
    }
    // Step the GenAlg
    GAStep(ga);
  }
  // Measure time
  clock_gettime(CLOCK_REALTIME, &stop);
  float elapsed = stop.tv_sec - start.tv_sec;
  int day = (int)floor(elapsed / 86400);
  elapsed -= (float)(day * 86400);
  int hour = (int)floor(elapsed / 3600);
  elapsed -= (float)(hour * 3600);
  int min = (int)floor(elapsed / 60);
  elapsed -= (float)(min * 60);
  int sec = (int)floor(elapsed);
  printf("\nLearning complete (in %d:%d:%d:%ds)\n",
    day, hour, min, sec);
  // Free memory
  NeuraNetFree(&nn);
  GenAlgFree(&ga);
  DataSetFree(&dataset);
}

// Check the NeuraNet 'that' on the DataSetCat 'cat'
void Validate(const NeuraNet* const that, const DataSetCat cat) {
  // Load the DataSet
  DataSet* dataset = PBErrMalloc(NeuraNetErr, sizeof(DataSet));
  bool ret = DataSetLoad(dataset, cat);
  if (!ret) {
    printf("Couldn't load the data\n");
    return;
  }
  // Evaluate the NeuraNet
  float value = Evaluate(that, dataset);
  // Display the result
  printf("Value: %.6f\n", value);
  // Free memory
  DataSetFree(&dataset);
}

// Predict using the NeuraNet 'that' on 'inputs' (given as an array of
// 'nbInp' char*)
void Predict(const NeuraNet* const that, const int nbInp,
  char** const inputs) {
  // Check the number of inputs
  if (nbInp != NNGetNbInput(that)) {
    printf("Wrong number of inputs, there should %d, there was %d\n",
      NNGetNbInput(that), nbInp);
    return;
  }
  // Declare 2 vectors to memorize the input and output values
  VecFloat* input = VecFloatCreate(NNGetNbInput(that));
  VecFloat* output = VecFloatCreate(NNGetNbOutput(that));
  // Set the input
  for (int iInp = 0; iInp < nbInp; ++iInp) {
    float v = 0.0;
    sscanf(inputs[iInp], "%f", &v);
    VecSet(input, iInp, v);
  }
  // Predict
```

```
  NNEval(that, input, output);
  printf("Prediction: %f rings\n", VecGet(output, 0));
  // Free memory
  VecFree(&input);
  VecFree(&output);
}

int main(int argc, char** argv) {
  // Declare a variable to memorize the mode (learning/checking)
  int mode = -1;
  // Declare a variable to memorize the dataset used
  DataSetCat cat = unknownDataSet;
  // Decode mode from arguments
  if (argc >= 3) {
    if (strcmp(argv[1], "-learn") == 0) {
      mode = 0;
      cat = GetCategoryFromName(argv[2]);
    } else if (strcmp(argv[1], "-check") == 0) {
      mode = 1;
      cat = GetCategoryFromName(argv[2]);
    } else if (strcmp(argv[1], "-predict") == 0) {
      mode = 2;
    }
  }
  // If the mode is invalid print some help
  if (mode == -1) {
    printf("Select a mode from:\n");
    printf("-learn <dataset name>\n");
    printf("-check <dataset name>\n");
    printf("-predict <input values>\n");
    return 0;
  }
  if (mode == 0) {
    Learn(cat);
  } else if (mode == 1) {
    NeuraNet* nn = NULL;
    FILE* fd = fopen("./bestnn.txt", "r");
    if (!NNLoad(&nn, fd)) {
      printf("Couldn't load the best NeuraNet\n");
      return 0;
    }
    fclose(fd);
    Validate(nn, cat);
    NeuraNetFree(&nn);
  } else if (mode == 2) {
    NeuraNet* nn = NULL;
    FILE* fd = fopen("./bestnn.txt", "r");
    if (!NNLoad(&nn, fd)) {
      printf("Couldn't load the best NeuraNet\n");
      return 0;
    }
    fclose(fd);
    Predict(nn, argc - 2, argv + 2);
    NeuraNetFree(&nn);
  }
  // Return success code
  return 0;
}
```

7.3 Arrhythmia data set

Source: https://archive.ics.uci.edu/ml/datasets/arrhythmia

main.c:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "pberr.h"
#include "genalg.h"
#include "neuranet.h"

// https://archive.ics.uci.edu/ml/datasets/arrhythmia

// Nb of step between each save of the GenAlg
// Saving it allows to restart a stop learning process but is
// very time consuming if there are many input/hidden/output
// If 0 never save
#define SAVE_GA_EVERY 100
// Nb input and output of the NeuraNet
#define NB_INPUT 279
#define NB_OUTPUT 16
// Nb max of hidden values, links and base functions
#define NB_MAXHIDDEN 50
#define NB_MAXLINK 1000
#define NB_MAXBASE NB_MAXLINK
// Size of the gene pool and elite pool
#define ADN_SIZE_POOL 500
#define ADN_SIZE_ELITE 20
// Initial best value during learning, must be lower than any
// possible value returned by Evaluate()
#define INIT_BEST_VAL -100000.0
// Value of the NeuraNet above which the learning process stops
#define STOP_LEARNING_AT_VAL 0.999
// Number of epoch above which the learning process stops
#define STOP_LEARNING_AT_EPOCH 25000
// Save NeuraNet in compact format
#define COMPACT true

// Categories of data sets

typedef enum DataSetCat {
  unknownDataSet,
  datalearn,
  datatest,
  dataall
} DataSetCat;
#define NB_DATASET 4
const char* dataSetNames[NB_DATASET] = {
  "unknownDataSet", "datalearn", "datatest", "dataall"
  };

const char* catNames[NB_OUTPUT] = {
  "Normal",
  "Ischemic changes (Coronary Artery Disease)",
```

```
  "Old Anterior Myocardial Infarction",
  "Old Inferior Myocardial Infarction",
  "Sinus tachycardy",
  "Sinus bradycardy",
  "Ventricular Premature Contraction (PVC)",
  "Supraventricular Premature Contraction",
  "Left bundle branch block",
  "Right bundle branch block",
  "1. degree AtrioVentricular block",
  "2. degree AV block",
  "3. degree AV block",
  "Left ventricule hypertrophy",
  "Atrial Fibrillation or Flutter",
  "Others"
};



typedef struct Arrhytmia {
  float _props[NB_INPUT];
  int _cat;
} Arrhytmia;

typedef struct DataSet {
  // Category of the data set
  DataSetCat _cat;
  // Number of sample
  int _nbSample;
  // Samples
  Arrhytmia* _samples;
} DataSet;

// Get the DataSetCat from its 'name'
DataSetCat GetCategoryFromName(const char* const name) {
  // Declare a variable to memorize the DataSetCat
  DataSetCat cat = unknownDataSet;
  // Search the dataset
  for (int iSet = NB_DATASET; iSet--;)
    if (strcmp(name, dataSetNames[iSet]) == 0)
      cat = iSet;
  // Return the category
  return cat;
}

// Load the data set of category 'cat' in the DataSet 'that'
// Return true on success, else false
bool DataSetLoad(DataSet* const that, const DataSetCat cat) {
  // Set the category
  that->_cat = cat;

  // Load the data according to 'cat'
  FILE* f = fopen("./arrhythmia.data", "r");
  if (f == NULL) {
    printf("Couldn't open the data set file\n");
    return false;
  }
  int ret = 0;
  if (cat == datalearn) {
    that->_nbSample = 300;
    that->_samples =
      PBErrMalloc(NeuraNetErr, sizeof(Arrhytmia) * that->_nbSample);
    for (int iSample = 0; iSample < that->_nbSample; ++iSample) {
```

```
    for (int iProp = 0; iProp < NB_INPUT; ++iProp) {
      ret = fscanf(f, "%f,",
        that->_samples[iSample]._props + iProp);
      if (ret == EOF) {
        printf("Couldn't read the dataset\n");
        fclose(f);
        return false;
      }
    }
    ret = fscanf(f, "%d", &(that->_samples[iSample]._cat));
    if (ret == EOF) {
      printf("Couldn't read the dataset\n");
      fclose(f);
      return false;
    }
  }
} else if (cat == datatest) {
  char buffer[1000];
  for (int iSample = 0; iSample < 300; ++iSample) {
    ret = fscanf(f, "%s", buffer);
    if (ret == EOF) {
      printf("Couldn't read the dataset\n");
      fclose(f);
      return false;
    }
  }
  that->_nbSample = 152;
  that->_samples =
    PBErrMalloc(NeuraNetErr, sizeof(Arrhytmia) * that->_nbSample);
  for (int iSample = 0; iSample < that->_nbSample; ++iSample) {
    for (int iProp = 0; iProp < NB_INPUT; ++iProp) {
      ret = fscanf(f, "%f,",
        that->_samples[iSample]._props + iProp);
      if (ret == EOF) {
        printf("Couldn't read the dataset\n");
        fclose(f);
        return false;
      }
    }
    ret = fscanf(f, "%d", &(that->_samples[iSample]._cat));
    if (ret == EOF) {
      printf("Couldn't read the dataset\n");
      fclose(f);
      return false;
    }
  }
} else if (cat == dataall) {
  that->_nbSample = 452;
  that->_samples =
    PBErrMalloc(NeuraNetErr, sizeof(Arrhytmia) * that->_nbSample);
  for (int iSample = 0; iSample < that->_nbSample; ++iSample) {
    for (int iProp = 0; iProp < NB_INPUT; ++iProp) {
      ret = fscanf(f, "%f,",
        that->_samples[iSample]._props + iProp);
      if (ret == EOF) {
        printf("Couldn't read the dataset\n");
        fclose(f);
        return false;
      }
    }
    ret = fscanf(f, "%d", &(that->_samples[iSample]._cat));
    if (ret == EOF) {
```

```
        printf("Couldn't read the dataset\n");
        fclose(f);
        return false;
      }
    }
  } else {
    printf("Invalid dataset\n");
    fclose(f);
    return false;
  }
  fclose(f);

  // Return success code
  return true;
}

// Free memory for the DataSet 'that'
void DataSetFree(DataSet** that) {
  if (*that == NULL) return;
  // Free the memory
  free((*that)->_samples);
  free(*that);
  *that = NULL;
}

// Evalutation function for the NeuraNet 'that' on the DataSet 'dataset'
// Return the value of the NeuraNet, the bigger the better
float Evaluate(const NeuraNet* const that,
  const DataSet* const dataset) {
  // Declare 2 vectors to memorize the input and output values
  VecFloat* input = VecFloatCreate(NNGetNbInput(that));
  VecFloat* output = VecFloatCreate(NNGetNbOutput(that));
  // Declare a variable to memorize the value
  float val = 0.0;

  // Evaluate

  int countCat[NB_OUTPUT] = {0};
  int countOk[NB_OUTPUT] = {0};
  int countNg[NB_OUTPUT] = {0};
  for (int iSample = dataset->_nbSample; iSample--;) {
    for (int iInp = 0; iInp < NNGetNbInput(that); ++iInp) {
      VecSet(input, iInp,
        dataset->_samples[iSample]._props[iInp]);
    }
    NNEval(that, input, output);
    int pred = VecGetIMaxVal(output) + 1;
    ++(countCat[dataset->_samples[iSample]._cat - 1]);
    if (pred == dataset->_samples[iSample]._cat) {
      ++(countOk[dataset->_samples[iSample]._cat - 1]);
    } else if (dataset->_cat == datalearn) {
      ++(countNg[dataset->_samples[iSample]._cat - 1]);
    }

  }
  int nbCat = 0;
  for (int iCat = 0; iCat < NB_OUTPUT; ++iCat) {
    if (countCat[iCat] > 0) {
      ++nbCat;
      float perc = 0.0;
      if (dataset->_cat != datalearn) {
        perc = (float)(countOk[iCat]) / (float)(countCat[iCat]);
```

```
        printf("%43s (%3d): %f\n", catNames[iCat], countCat[iCat], perc);
        val += countOk[iCat];
      } else {
        perc = (float)(countOk[iCat] - countNg[iCat]) /
          (float)(countCat[iCat]);
        val += perc;
      }
    }
  }
  if (dataset->_cat != datalearn)
    val /= (float)(dataset->_nbSample);
  else
    val /= (float)nbCat;

  // Free memory
  VecFree(&input);
  VecFree(&output);
  // Return the result of the evaluation
  return val;
}

// Create the NeuraNet
NeuraNet* createNN(void) {
  // Create the NeuraNet
  int nbIn = NB_INPUT;
  int nbOut = NB_OUTPUT;
  int nbMaxHid = NB_MAXHIDDEN;
  int nbMaxLink = NB_MAXLINK;
  int nbMaxBase = NB_MAXBASE;
  NeuraNet* nn =
    NeuraNetCreate(nbIn, nbOut, nbMaxHid, nbMaxBase, nbMaxLink);

  // Return the NeuraNet
  return nn;
}

// Learn based on the SataSetCat 'cat'
void Learn(DataSetCat cat) {
  // Init the random generator
  srandom(time(NULL));
  // Declare variables to measure time
  struct timespec start, stop;
  // Start measuring time
  clock_gettime(CLOCK_REALTIME, &start);
  // Load the DataSet
  DataSet* dataset = PBErrMalloc(NeuraNetErr, sizeof(DataSet));
  bool ret = DataSetLoad(dataset, cat);
  if (!ret) {
    printf("Couldn't load the data\n");
    return;
  }
  // Create the NeuraNet
  NeuraNet* nn = createNN();
  // Declare a variable to memorize the best value
  float bestVal = INIT_BEST_VAL;
  // Declare a variable to memorize the limit in term of epoch
  unsigned long int limitEpoch = STOP_LEARNING_AT_EPOCH;
  // Create the GenAlg used for learning
  // If previous weights are available in "./bestga.txt" reload them
  GenAlg* ga = NULL;
  FILE* fd = fopen("./bestga.txt", "r");
  if (fd) {
```

```
  printf("Reloading previous GenAlg...\n");
  if (!GALoad(&ga, fd)) {
    printf("Failed to reload the GenAlg.\n");
    NeuraNetFree(&nn);
    DataSetFree(&dataset);
    return;
  } else {
    printf("Previous GenAlg reloaded.\n");
    if (GABestAdnF(ga) != NULL)
      NNSetBases(nn, GABestAdnF(ga));
    if (GABestAdnI(ga) != NULL)
      NNSetLinks(nn, GABestAdnI(ga));
    bestVal = Evaluate(nn, dataset);
    printf("Starting with best at %f.\n", bestVal);
    limitEpoch += GAGetCurEpoch(ga);
  }
  fclose(fd);
} else {
  ga = GenAlgCreate(ADN_SIZE_POOL, ADN_SIZE_ELITE,
    NNGetGAAdnFloatLength(nn), NNGetGAAdnIntLength(nn));
  NNSetGABoundsBases(nn, ga);
  NNSetGABoundsLinks(nn, ga);
  // Must be declared as a GenAlg applied to a NeuraNet or links will
  // get corrupted
  GASetTypeNeuraNet(ga, NB_INPUT, NB_MAXHIDDEN, NB_OUTPUT);
  GAInit(ga);
}
// If there is a NeuraNet available, reload it into the GenAlg
fd = fopen("./bestnn.txt", "r");
if (fd) {
  printf("Reloading previous NeuraNet...\n");
  if (!NNLoad(&nn, fd)) {
    printf("Failed to reload the NeuraNet.\n");
    NeuraNetFree(&nn);
    DataSetFree(&dataset);
    return;
  } else {
    printf("Previous NeuraNet reloaded.\n");
    bestVal = Evaluate(nn, dataset);
    printf("Starting with best at %f.\n", bestVal);
    GenAlgAdn* adn = GAAdn(ga, 0);
    VecCopy(adn->_adnF, nn->_bases);
    VecCopy(adn->_adnI, nn->_links);
  }
  fclose(fd);
}
// Start learning process
printf("Learning...\n");
printf("Will stop when curEpoch >= %lu or bestVal >= %f\n",
  limitEpoch, STOP_LEARNING_AT_VAL);
printf("Will save the best NeuraNet in ./bestnn.txt at each improvement\n");
fflush(stdout);
// Declare a variable to memorize the best value in the current epoch
float curBest = 0.0;
float curWorst = 0.0;
// Declare a variable to manage the save of GenAlg
int delaySave = 0;
// Learning loop
while (bestVal < STOP_LEARNING_AT_VAL &&
  GAGetCurEpoch(ga) < limitEpoch) {
  curWorst = curBest;
  curBest = INIT_BEST_VAL;
```

```
int curBestI = 0;
unsigned long int ageBest = 0;
// For each adn in the GenAlg
//for (int iEnt = GAGetNbAdns(ga); iEnt--;) {
for (int iEnt = 0; iEnt < GAGetNbAdns(ga); ++iEnt) {
  // Get the adn
  GenAlgAdn* adn = GAAdn(ga, iEnt);
  // Set the links and base functions of the NeuraNet according
  // to this adn
  if (GABestAdnF(ga) != NULL)
    NNSetBases(nn, GAAdnAdnF(adn));
  if (GABestAdnI(ga) != NULL)
    NNSetLinks(nn, GAAdnAdnI(adn));
  // Evaluate the NeuraNet
  float value = Evaluate(nn, dataset);
  // Update the value of this adn
  GASetAdnValue(ga, adn, value);
  // Update the best value in the current epoch
  if (value > curBest) {
    curBest = value;
    curBestI = iEnt;
    ageBest = GAAdnGetAge(adn);
  }
  if (value < curWorst)
    curWorst = value;
}
// Measure time
clock_gettime(CLOCK_REALTIME, &stop);
float elapsed = stop.tv_sec - start.tv_sec;
int day = (int)floor(elapsed / 86400);
elapsed -= (float)(day * 86400);
int hour = (int)floor(elapsed / 3600);
elapsed -= (float)(hour * 3600);
int min = (int)floor(elapsed / 60);
elapsed -= (float)(min * 60);
int sec = (int)floor(elapsed);
// If there has been improvement during this epoch
if (curBest > bestVal) {
  bestVal = curBest;
  // Display info about the improvment
  printf("Improvement at epoch %05lu: %f(%03d) (in %02d:%02d:%02d:%02ds)       \n",
    GAGetCurEpoch(ga), bestVal, curBestI, day, hour, min, sec);
  fflush(stdout);
  // Set the links and base functions of the NeuraNet according
  // to the best adn
  GenAlgAdn* bestAdn = GAAdn(ga, curBestI);
  if (GAAdnAdnF(bestAdn) != NULL)
    NNSetBases(nn, GAAdnAdnF(bestAdn));
  if (GAAdnAdnI(bestAdn) != NULL)
    NNSetLinks(nn, GAAdnAdnI(bestAdn));
  // Save the best NeuraNet
  fd = fopen("./bestnn.txt", "w");
  if (!NNSave(nn, fd, COMPACT)) {
    printf("Couldn't save the NeuraNet\n");
    NeuraNetFree(&nn);
    GenAlgFree(&ga);
    DataSetFree(&dataset);
    return;
  }
  fclose(fd);
} else {
  fprintf(stderr,
```

```
        "Epoch %05lu: v%f a%03lu(%03d) kt%03lu ",
        GAGetCurEpoch(ga), curBest, ageBest, curBestI,
        GAGetNbKTEvent(ga));
      fprintf(stderr, "(in %02d:%02d:%02d:%02ds)  \r",
        day, hour, min, sec);
      fflush(stderr);
    }
    ++delaySave;
    if (SAVE_GA_EVERY != 0 && delaySave >= SAVE_GA_EVERY) {
      delaySave = 0;
      // Save the adns of the GenAlg, use a temporary file to avoid
      // loosing the previous one if something goes wrong during
      // writing, then replace the previous file with the temporary one
      fd = fopen("./bestga.tmp", "w");
      if (!GASave(ga, fd, COMPACT)) {
        printf("Couldn't save the GenAlg\n");
        NeuraNetFree(&nn);
        GenAlgFree(&ga);
        DataSetFree(&dataset);
        return;
      }
      fclose(fd);
      int ret = system("mv ./bestga.tmp ./bestga.txt");
      (void)ret;
    }
    // Step the GenAlg
    GAStep(ga);
  }
  // Measure time
  clock_gettime(CLOCK_REALTIME, &stop);
  float elapsed = stop.tv_sec - start.tv_sec;
  int day = (int)floor(elapsed / 86400);
  elapsed -= (float)(day * 86400);
  int hour = (int)floor(elapsed / 3600);
  elapsed -= (float)(hour * 3600);
  int min = (int)floor(elapsed / 60);
  elapsed -= (float)(min * 60);
  int sec = (int)floor(elapsed);
  printf("\nLearning complete (in %d:%d:%d:%ds)\n",
    day, hour, min, sec);
  // Free memory
  NeuraNetFree(&nn);
  GenAlgFree(&ga);
  DataSetFree(&dataset);
}

// Check the NeuraNet 'that' on the DataSetCat 'cat'
void Validate(const NeuraNet* const that, const DataSetCat cat) {
  // Load the DataSet
  DataSet* dataset = PBErrMalloc(NeuraNetErr, sizeof(DataSet));
  bool ret = DataSetLoad(dataset, cat);
  if (!ret) {
    printf("Couldn't load the data\n");
    return;
  }
  // Evaluate the NeuraNet
  float value = Evaluate(that, dataset);
  // Display the result
  printf("Value: %.6f\n", value);
  // Free memory
  DataSetFree(&dataset);
}
```

```
// Predict using the NeuraNet 'that' on 'inputs' (given as an array of
// 'nbInp' char*)
void Predict(const NeuraNet* const that, const int nbInp,
  char** const inputs) {
  // Start measuring time
  clock_t clockStart = clock();
  // Check the number of inputs
  if (nbInp != NNGetNbInput(that)) {
    printf("Wrong number of inputs, there should %d, there was %d\n",
      NNGetNbInput(that), nbInp);
    return;
  }
  // Declare 2 vectors to memorize the input and output values
  VecFloat* input = VecFloatCreate(NNGetNbInput(that));
  VecFloat* output = VecFloatCreate(NNGetNbOutput(that));
  // Set the input
  for (int iInp = 0; iInp < nbInp; ++iInp) {
    float v = 0.0;
    sscanf(inputs[iInp], "%f", &v);
    VecSet(input, iInp, v);
  }
  // Predict
  NNEval(that, input, output);
  int pred = VecGetIMaxVal(output);
  // End measuring time
  clock_t clockEnd = clock();
  double timeUsed =
    ((double)(clockEnd - clockStart)) / (CLOCKS_PER_SEC * 0.001) ;
  // If the clock has been reset meanwhile
  if (timeUsed < 0.0)
    timeUsed = 0.0;
  printf("Prediction: %s (in %fms)\n", catNames[pred], timeUsed);

  // Free memory
  VecFree(&input);
  VecFree(&output);
}

int main(int argc, char** argv) {
  // Declare a variable to memorize the mode (learning/checking)
  int mode = -1;
  // Declare a variable to memorize the dataset used
  DataSetCat cat = unknownDataSet;
  // Decode mode from arguments
  if (argc >= 3) {
    if (strcmp(argv[1], "-learn") == 0) {
      mode = 0;
      cat = GetCategoryFromName(argv[2]);
    } else if (strcmp(argv[1], "-check") == 0) {
      mode = 1;
      cat = GetCategoryFromName(argv[2]);
    } else if (strcmp(argv[1], "-predict") == 0) {
      mode = 2;
    }
  }
  // If the mode is invalid print some help
  if (mode == -1) {
    printf("Select a mode from:\n");
    printf("-learn <dataset name>\n");
    printf("-check <dataset name>\n");
    printf("-predict <input values>\n");
```

```
    return 0;
  }
  if (mode == 0) {
    Learn(cat);
  } else if (mode == 1) {
    NeuraNet* nn = NULL;
    FILE* fd = fopen("./bestnn.txt", "r");
    if (!NNLoad(&nn, fd)) {
      printf("Couldn't load the best NeuraNet\n");
      return 0;
    }
    fclose(fd);
    Validate(nn, cat);
    NeuraNetFree(&nn);
  } else if (mode == 2) {
    NeuraNet* nn = NULL;
    FILE* fd = fopen("./bestnn.txt", "r");
    if (!NNLoad(&nn, fd)) {
      printf("Couldn't load the best NeuraNet\n");
      return 0;
    }
    fclose(fd);
    Predict(nn, argc - 2, argv + 2);
    NeuraNetFree(&nn);
  }
  // Return success code
  return 0;
}
```

7.4 Wisconsin Diagnostic Breast Cancer

Source: https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+Diagnostic

main.c:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "pberr.h"
#include "genalg.h"
#include "neuranet.h"

// https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29

// Nb of step between each save of the GenAlg
// Saving it allows to restart a stop learning process but is
// very time consuming if there are many input/hidden/output
// If 0 never save
#define SAVE_GA_EVERY 100
// Nb input and output of the NeuraNet
#define NB_INPUT 30
#define NB_OUTPUT 2
// Nb max of hidden values, links and base functions
```

```
#define NB_MAXHIDDEN 0
#define NB_MAXLINK 300
#define NB_MAXBASE NB_MAXLINK
// Size of the gene pool and elite pool
#define ADN_SIZE_POOL 100
#define ADN_SIZE_ELITE 20
// Initial best value during learning, must be lower than any
// possible value returned by Evaluate()
#define INIT_BEST_VAL -1000.0
// Value of the NeuraNet above which the learning process stops
#define STOP_LEARNING_AT_VAL 0.999
// Number of epoch above which the learning process stops
#define STOP_LEARNING_AT_EPOCH 5000
// Save NeuraNet in compact format
#define COMPACT true

// Categories of data sets

typedef enum DataSetCat {
  unknownDataSet,
  datalearn,
  datatest,
  dataall
} DataSetCat;
#define NB_DATASET 4
const char* dataSetNames[NB_DATASET] = {
  "unknownDataSet", "datalearn", "datatest", "dataall"
  };

const char* catNames[NB_OUTPUT] = {
  "Malignant",
  "Benign"
};

// Structure for the data set

typedef struct Sample {
  float _props[NB_INPUT];
  int _cat;
  int _id;
} Sample;

typedef struct DataSet {
  // Category of the data set
  DataSetCat _cat;
  // Number of sample
  int _nbSample;
  // Samples
  Sample* _samples;
} DataSet;

// Get the DataSetCat from its 'name'
DataSetCat GetCategoryFromName(const char* const name) {
  // Declare a variable to memorize the DataSetCat
  DataSetCat cat = unknownDataSet;
  // Search the dataset
  for (int iSet = NB_DATASET; iSet--;)
    if (strcmp(name, dataSetNames[iSet]) == 0)
      cat = iSet;
  // Return the category
  return cat;
}
```

```
// Load the data set of category 'cat' in the DataSet 'that'
// Return true on success, else false
bool DataSetLoad(DataSet* const that, const DataSetCat cat) {
  // Set the category
  that->_cat = cat;

  // Load the data according to 'cat'
  FILE* f = fopen("./wdbc.data", "r");
  if (f == NULL) {
    printf("Couldn't open the data set file\n");
    return false;
  }
  int ret = 0;
  if (cat == datalearn) {
    that->_nbSample = 400;
    that->_samples =
      PBErrMalloc(NeuraNetErr, sizeof(Sample) * that->_nbSample);
    for (int iSample = 0; iSample < that->_nbSample; ++iSample) {
      ret = fscanf(f, "%d,", &(that->_samples[iSample]._id));
      if (ret == EOF) {
        printf("Couldn't read the dataset\n");
        fclose(f);
        return false;
      }
      char cat;
      ret = fscanf(f, "%c,", &cat);
      if (ret == EOF) {
        printf("Couldn't read the dataset\n");
        fclose(f);
        return false;
      }
      if (cat == 'M')
        that->_samples[iSample]._cat = 0;
      else if (cat == 'B')
        that->_samples[iSample]._cat = 1;
      else {
        printf("Couldn't read the dataset\n");
        fclose(f);
        return false;
      }
      for (int iProp = 0; iProp < NB_INPUT; ++iProp) {
        ret = fscanf(f, "%f,",
          that->_samples[iSample]._props + iProp);
        if (ret == EOF) {
          printf("Couldn't read the dataset\n");
          fclose(f);
          return false;
        }
      }
    }
  } else if (cat == datatest) {
    char buffer[1000];
    for (int iSample = 0; iSample < 400; ++iSample) {
      ret = fscanf(f, "%s", buffer);
      if (ret == EOF) {
        printf("Couldn't read the dataset\n");
        fclose(f);
        return false;
      }
    }
    that->_nbSample = 169;
```

```
  that->_samples =
    PBErrMalloc(NeuraNetErr, sizeof(Sample) * that->_nbSample);
  for (int iSample = 0; iSample < that->_nbSample; ++iSample) {
    ret = fscanf(f, "%d,", &(that->_samples[iSample]._id));
    if (ret == EOF) {
      printf("Couldn't read the dataset\n");
      fclose(f);
      return false;
    }
    char cat;
    ret = fscanf(f, "%c,", &cat);
    if (ret == EOF) {
      printf("Couldn't read the dataset\n");
      fclose(f);
      return false;
    }
    if (cat == 'M')
      that->_samples[iSample]._cat = 0;
    else if (cat == 'B')
      that->_samples[iSample]._cat = 1;
    else {
      printf("Couldn't read the dataset\n");
      fclose(f);
      return false;
    }
    for (int iProp = 0; iProp < NB_INPUT; ++iProp) {
      ret = fscanf(f, "%f,",
        that->_samples[iSample]._props + iProp);
      if (ret == EOF) {
        printf("Couldn't read the dataset\n");
        fclose(f);
        return false;
      }
    }
  }
} else if (cat == dataall) {
  that->_nbSample = 569;
  that->_samples =
    PBErrMalloc(NeuraNetErr, sizeof(Sample) * that->_nbSample);
  for (int iSample = 0; iSample < that->_nbSample; ++iSample) {
    ret = fscanf(f, "%d,", &(that->_samples[iSample]._id));
    if (ret == EOF) {
      printf("Couldn't read the dataset\n");
      fclose(f);
      return false;
    }
    char cat;
    ret = fscanf(f, "%c,", &cat);
    if (ret == EOF) {
      printf("Couldn't read the dataset\n");
      fclose(f);
      return false;
    }
    if (cat == 'M')
      that->_samples[iSample]._cat = 0;
    else if (cat == 'B')
      that->_samples[iSample]._cat = 1;
    else {
      printf("Couldn't read the dataset\n");
      fclose(f);
      return false;
    }
```

```
      for (int iProp = 0; iProp < NB_INPUT; ++iProp) {
        ret = fscanf(f, "%f,",
          that->_samples[iSample]._props + iProp);
        if (ret == EOF) {
          printf("Couldn't read the dataset\n");
          fclose(f);
          return false;
        }
      }
    }
  } else {
    printf("Invalid dataset\n");
    fclose(f);
    return false;
  }
  fclose(f);

  // Return success code
  return true;
}

// Free memory for the DataSet 'that'
void DataSetFree(DataSet** that) {
  if (*that == NULL) return;
  // Free the memory

  free(*that);
  *that = NULL;
}

// Evalutation function for the NeuraNet 'that' on the DataSet 'dataset'
// Return the value of the NeuraNet, the bigger the better
float Evaluate(const NeuraNet* const that,
  const DataSet* const dataset) {
  // Declare 2 vectors to memorize the input and output values
  VecFloat* input = VecFloatCreate(NNGetNbInput(that));
  VecFloat* output = VecFloatCreate(NNGetNbOutput(that));
  // Declare a variable to memorize the value
  float val = 0.0;

  // Evaluate

  int countCat[NB_OUTPUT] = {0};
  int countOk[NB_OUTPUT] = {0};
  int countNg[NB_OUTPUT] = {0};
  for (int iSample = dataset->_nbSample; iSample--;) {
    for (int iInp = 0; iInp < NNGetNbInput(that); ++iInp) {
      VecSet(input, iInp,
        dataset->_samples[iSample]._props[iInp]);
    }
    NNEval(that, input, output);
    int pred = VecGetIMaxVal(output);
    ++(countCat[dataset->_samples[iSample]._cat]);
    if (pred == dataset->_samples[iSample]._cat) {
      ++(countOk[dataset->_samples[iSample]._cat]);
    } else if (dataset->_cat == datalearn) {
      ++(countNg[dataset->_samples[iSample]._cat]);
    }
    if (dataset->_cat != datalearn) {
      printf("%010d %10s ", dataset->_samples[iSample]._id,
        catNames[pred]);
      if (pred == dataset->_samples[iSample]._cat)
```

```
        printf("OK");
      else
        printf("NG");
      printf("\n");
    }
  }
  int nbCat = 0;
  for (int iCat = 0; iCat < NB_OUTPUT; ++iCat) {
    if (countCat[iCat] > 0) {
      ++nbCat;
      float perc = 0.0;
      if (dataset->_cat != datalearn) {
        perc = (float)(countOk[iCat]) / (float)(countCat[iCat]);
        printf("%10s (%3d): %f\n", catNames[iCat], countCat[iCat], perc);
        val += countOk[iCat];
      } else {
        perc = (float)(countOk[iCat] - countNg[iCat]) /
          (float)(countCat[iCat]);
        val += perc;
      }
    }
  }
  if (dataset->_cat != datalearn)
    val /= (float)(dataset->_nbSample);
  else
    val /= (float)nbCat;

  // Free memory
  VecFree(&input);
  VecFree(&output);
  // Return the result of the evaluation
  return val;
}

// Create the NeuraNet
NeuraNet* createNN(void) {
  // Create the NeuraNet
  int nbIn = NB_INPUT;
  int nbOut = NB_OUTPUT;
  int nbMaxHid = NB_MAXHIDDEN;
  int nbMaxLink = NB_MAXLINK;
  int nbMaxBase = NB_MAXBASE;
  NeuraNet* nn =
    NeuraNetCreate(nbIn, nbOut, nbMaxHid, nbMaxBase, nbMaxLink);

  // Set the links

  //VecShort* links = VecShortCreate(nbMaxLink * NN_NBPARAMLINK);
  //for (int iLink = VecGetDim(links); iLink--;) {
  //  VecSet(links, iLink * NN_NBPARAMLINK, base index
  //    -1 means inactive );
  //  VecSet(links, iLink * NN_NBPARAMLINK + 1, input index );
  //  VecSet(links, iLink * NN_NBPARAMLINK + 2, output index );
  //}
  //NNSetLinks(nn, links);
  //VecFree(&links);

  // Set the bases

  //VecFloat* bases = VecFloatCreate(nbMaxBase * NN_NBPARAMBASE);
  //for (int iBase = VecGetDim(bases); iBases--;) {
  //  tan(param[0]*NN_THETA)*(x+param[1])+param[2]
```

```
  //  param[] in [-1,1]
  //  VecSet(bases, iBase * NN_NBPARAMBASE,  );
  //  VecSet(bases, iBase * NN_NBPARAMBASE + 1,  );
  //  VecSet(bases, iBase * NN_NBPARAMBASE + 2,  );
  //}
  //NNSetBases(nn, bases);
  //VecFree(&bases);

  // Return the NeuraNet
  return nn;
}

// Learn based on the SataSetCat 'cat'
void Learn(DataSetCat cat) {
  // Init the random generator
  srandom(time(NULL));
  // Declare variables to measure time
  struct timespec start, stop;
  // Start measuring time
  clock_gettime(CLOCK_REALTIME, &start);
  // Load the DataSet
  DataSet* dataset = PBErrMalloc(NeuraNetErr, sizeof(DataSet));
  bool ret = DataSetLoad(dataset, cat);
  if (!ret) {
    printf("Couldn't load the data\n");
    return;
  }
  // Create the NeuraNet
  NeuraNet* nn = createNN();
  // Declare a variable to memorize the best value
  float bestVal = INIT_BEST_VAL;
  // Declare a variable to memorize the limit in term of epoch
  unsigned long int limitEpoch = STOP_LEARNING_AT_EPOCH;
  // Create the GenAlg used for learning
  // If previous weights are available in "./bestga.txt" reload them
  GenAlg* ga = NULL;
  FILE* fd = fopen("./bestga.txt", "r");
  if (fd) {
    printf("Reloading previous GenAlg...\n");
    if (!GALoad(&ga, fd)) {
      printf("Failed to reload the GenAlg.\n");
      NeuraNetFree(&nn);
      DataSetFree(&dataset);
      return;
    } else {
      printf("Previous GenAlg reloaded.\n");
      if (GABestAdnF(ga) != NULL)
        NNSetBases(nn, GABestAdnF(ga));
      if (GABestAdnI(ga) != NULL)
        NNSetLinks(nn, GABestAdnI(ga));
      bestVal = Evaluate(nn, dataset);
      printf("Starting with best at %f.\n", bestVal);
      limitEpoch += GAGetCurEpoch(ga);
    }
    fclose(fd);
  } else {
    ga = GenAlgCreate(ADN_SIZE_POOL, ADN_SIZE_ELITE,
      NNGetGAAdnFloatLength(nn), NNGetGAAdnIntLength(nn));
    NNSetGABoundsBases(nn, ga);
    NNSetGABoundsLinks(nn, ga);
    // Must be declared as a GenAlg applied to a NeuraNet or links will
    // get corrupted
```

```
  GASetTypeNeuraNet(ga, NB_INPUT, NB_MAXHIDDEN, NB_OUTPUT);
  GAInit(ga);
}
// If there is a NeuraNet available, reload it into the GenAlg
fd = fopen("./bestnn.txt", "r");
if (fd) {
  printf("Reloading previous NeuraNet...\n");
  if (!NNLoad(&nn, fd)) {
    printf("Failed to reload the NeuraNet.\n");
    NeuraNetFree(&nn);
    DataSetFree(&dataset);
    return;
  } else {
    printf("Previous NeuraNet reloaded.\n");
    bestVal = Evaluate(nn, dataset);
    printf("Starting with best at %f.\n", bestVal);
    GenAlgAdn* adn = GAAdn(ga, 0);
    VecCopy(adn->_adnF, nn->_bases);
    VecCopy(adn->_adnI, nn->_links);
  }
  fclose(fd);
}
// Start learning process
printf("Learning...\n");
printf("Will stop when curEpoch >= %lu or bestVal >= %f\n",
  limitEpoch, STOP_LEARNING_AT_VAL);
printf("Will save the best NeuraNet in ./bestnn.txt at each improvement\n");
fflush(stdout);
// Declare a variable to memorize the best value in the current epoch
float curBest = 0.0;
float curWorst = 0.0;
// Declare a variable to manage the save of GenAlg
int delaySave = 0;
// Learning loop
while (bestVal < STOP_LEARNING_AT_VAL &&
  GAGetCurEpoch(ga) < limitEpoch) {
  curWorst = curBest;
  curBest = INIT_BEST_VAL;
  int curBestI = 0;
  unsigned long int ageBest = 0;
  // For each adn in the GenAlg
  //for (int iEnt = GAGetNbAdns(ga); iEnt--;) {
  for (int iEnt = 0; iEnt < GAGetNbAdns(ga); ++iEnt) {
    // Get the adn
    GenAlgAdn* adn = GAAdn(ga, iEnt);
    // Set the links and base functions of the NeuraNet according
    // to this adn
    if (GABestAdnF(ga) != NULL)
      NNSetBases(nn, GAAdnAdnF(adn));
    if (GABestAdnI(ga) != NULL)
      NNSetLinks(nn, GAAdnAdnI(adn));
    // Evaluate the NeuraNet
    float value = Evaluate(nn, dataset);
    // Update the value of this adn
    GASetAdnValue(ga, adn, value);
    // Update the best value in the current epoch
    if (value > curBest) {
      curBest = value;
      curBestI = iEnt;
      ageBest = GAAdnGetAge(adn);
    }
    if (value < curWorst)
```

```
    curWorst = value;
}
// Measure time
clock_gettime(CLOCK_REALTIME, &stop);
float elapsed = stop.tv_sec - start.tv_sec;
int day = (int)floor(elapsed / 86400);
elapsed -= (float)(day * 86400);
int hour = (int)floor(elapsed / 3600);
elapsed -= (float)(hour * 3600);
int min = (int)floor(elapsed / 60);
elapsed -= (float)(min * 60);
int sec = (int)floor(elapsed);
// If there has been improvement during this epoch
if (curBest > bestVal) {
  bestVal = curBest;
  // Display info about the improvment
  printf("Improvement at epoch %05lu: %f(%03d) (in %02d:%02d:%02d:%02ds)       \n",
    GAGetCurEpoch(ga), bestVal, curBestI, day, hour, min, sec);
  fflush(stdout);
  // Set the links and base functions of the NeuraNet according
  // to the best adn
  GenAlgAdn* bestAdn = GAAdn(ga, curBestI);
  if (GAAdnAdnF(bestAdn) != NULL)
    NNSetBases(nn, GAAdnAdnF(bestAdn));
  if (GAAdnAdnI(bestAdn) != NULL)
    NNSetLinks(nn, GAAdnAdnI(bestAdn));
  // Save the best NeuraNet
  fd = fopen("./bestnn.txt", "w");
  if (!NNSave(nn, fd, COMPACT)) {
    printf("Couldn't save the NeuraNet\n");
    NeuraNetFree(&nn);
    GenAlgFree(&ga);
    DataSetFree(&dataset);
    return;
  }
  fclose(fd);
} else {
  fprintf(stderr,
    "Epoch %05lu: v%f a%03lu(%03d) kt%03lu ",
    GAGetCurEpoch(ga), curBest, ageBest, curBestI,
    GAGetNbKTEvent(ga));
  fprintf(stderr, "(in %02d:%02d:%02d:%02ds)  \r",
    day, hour, min, sec);
  fflush(stderr);
}
++delaySave;
if (SAVE_GA_EVERY != 0 && delaySave >= SAVE_GA_EVERY) {
  delaySave = 0;
  // Save the adns of the GenAlg, use a temporary file to avoid
  // loosing the previous one if something goes wrong during
  // writing, then replace the previous file with the temporary one
  fd = fopen("./bestga.tmp", "w");
  if (!GASave(ga, fd, COMPACT)) {
    printf("Couldn't save the GenAlg\n");
    NeuraNetFree(&nn);
    GenAlgFree(&ga);
    DataSetFree(&dataset);
    return;
  }
  fclose(fd);
  int ret = system("mv ./bestga.tmp ./bestga.txt");
  (void)ret;
```

```
    }
    // Step the GenAlg
    GAStep(ga);
  }
  // Measure time
  clock_gettime(CLOCK_REALTIME, &stop);
  float elapsed = stop.tv_sec - start.tv_sec;
  int day = (int)floor(elapsed / 86400);
  elapsed -= (float)(day * 86400);
  int hour = (int)floor(elapsed / 3600);
  elapsed -= (float)(hour * 3600);
  int min = (int)floor(elapsed / 60);
  elapsed -= (float)(min * 60);
  int sec = (int)floor(elapsed);
  printf("\nLearning complete (in %d:%d:%d:%ds)\n",
    day, hour, min, sec);
  // Free memory
  NeuraNetFree(&nn);
  GenAlgFree(&ga);
  DataSetFree(&dataset);
}

// Check the NeuraNet 'that' on the DataSetCat 'cat'
void Check(const NeuraNet* const that, const DataSetCat cat) {
  // Load the DataSet
  DataSet* dataset = PBErrMalloc(NeuraNetErr, sizeof(DataSet));
  bool ret = DataSetLoad(dataset, cat);
  if (!ret) {
    printf("Couldn't load the data\n");
    return;
  }
  // Evaluate the NeuraNet
  float value = Evaluate(that, dataset);
  // Display the result
  printf("Value: %.6f\n", value);
  // Free memory
  DataSetFree(&dataset);
}

// Predict using the NeuraNet 'that' on 'inputs' (given as an array of
// 'nbInp' char*)
void Predict(const NeuraNet* const that, const int nbInp,
  char** const inputs) {
  // Start measuring time
  clock_t clockStart = clock();
  // Check the number of inputs
  if (nbInp != NNGetNbInput(that)) {
    printf("Wrong number of inputs, there should %d, there was %d\n",
      NNGetNbInput(that), nbInp);
    return;
  }
  // Declare 2 vectors to memorize the input and output values
  VecFloat* input = VecFloatCreate(NNGetNbInput(that));
  VecFloat* output = VecFloatCreate(NNGetNbOutput(that));
  // Set the input
  for (int iInp = 0; iInp < nbInp; ++iInp) {
    float v = 0.0;
    sscanf(inputs[iInp], "%f", &v);
    VecSet(input, iInp, v);
  }
  // Predict
  NNEval(that, input, output);
```

```
  // End measuring time
  clock_t clockEnd = clock();
  double timeUsed =
    ((double)(clockEnd - clockStart)) / (CLOCKS_PER_SEC * 0.001) ;
  // If the clock has been reset meanwhile
  if (timeUsed < 0.0)
    timeUsed = 0.0;
  //if (VecGet(output, 0) == ...)
  //  printf("...(in %fms)", timeUsed);

  // Free memory
  VecFree(&input);
  VecFree(&output);
}

int main(int argc, char** argv) {
  // Declare a variable to memorize the mode (learning/checking)
  int mode = -1;
  // Declare a variable to memorize the dataset used
  DataSetCat cat = unknownDataSet;
  // Decode mode from arguments
  if (argc >= 3) {
    if (strcmp(argv[1], "-learn") == 0) {
      mode = 0;
      cat = GetCategoryFromName(argv[2]);
    } else if (strcmp(argv[1], "-check") == 0) {
      mode = 1;
      cat = GetCategoryFromName(argv[2]);
    } else if (strcmp(argv[1], "-predict") == 0) {
      mode = 2;
    }
  }
  // If the mode is invalid print some help
  if (mode == -1) {
    printf("Select a mode from:\n");
    printf("-learn <dataset name>\n");
    printf("-check <dataset name>\n");
    printf("-predict <input values>\n");
    return 0;
  }
  if (mode == 0) {
    Learn(cat);
  } else if (mode == 1) {
    NeuraNet* nn = NULL;
    FILE* fd = fopen("./bestnn.txt", "r");
    if (!NNLoad(&nn, fd)) {
      printf("Couldn't load the best NeuraNet\n");
      return 0;
    }
    fclose(fd);
    Check(nn, cat);
    NeuraNetFree(&nn);
  } else if (mode == 2) {
    NeuraNet* nn = NULL;
    FILE* fd = fopen("./bestnn.txt", "r");
    if (!NNLoad(&nn, fd)) {
      printf("Couldn't load the best NeuraNet\n");
      return 0;
    }
    fclose(fd);
    Predict(nn, argc - 2, argv + 2);
```

```
    NeuraNetFree(&nn);
  }
  // Return success code
  return 0;
}
```

7.5 MNIST

Source: http://yann.lecun.com/exdb/mnist/

main.c:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "pberr.h"
#include "genalg.h"
#include "neuranet.h"

// http://yann.lecun.com/exdb/mnist/

// Nb of step between each save of the GenAlg
// Saving it allows to restart a stop learning process but is
// very time consuming if there are many input/hidden/output
// If 0 never save
#define SAVE_GA_EVERY 100
// Nb input and output of the NeuraNet
#define MNIST_IMGSIZE 28
#define NB_INPUT MNIST_IMGSIZE * MNIST_IMGSIZE
#define NB_OUTPUT 10
// Nb max of hidden values, links and base functions
#define NB_MAXHIDDEN 100
#define NB_MAXLINK 300
#define NB_MAXBASE NB_MAXLINK
// Size of the gene pool and elite pool
#define ADN_SIZE_POOL 100
#define ADN_SIZE_ELITE 20
// Initial best value during learning, must be lower than any
// possible value returned by Evaluate()
#define INIT_BEST_VAL -1000.0
// Value of the NeuraNet above which the learning process stops
#define STOP_LEARNING_AT_VAL 0.999
// Number of epoch above which the learning process stops
#define STOP_LEARNING_AT_EPOCH 100000
// Save NeuraNet in compact format
#define COMPACT true
// Use convolution if true
#define CONVOLUTION false

// Categories of data sets

typedef enum DataSetCat {
  unknownDataSet,
```

```
  datalearn,
  datatest,
  dataall
} DataSetCat;
#define NB_DATASET 4
const char* dataSetNames[NB_DATASET] = {
  "unknownDataSet", "datalearn", "datatest", "dataall"
  };

const char* catNames[NB_OUTPUT] = {
  "0",
  "1",
  "2",
  "3",
  "4",
  "5",
  "6",
  "7",
  "8",
  "9",
};

// Structure for the data set

typedef struct MNISTImg {
  unsigned char _cat;
  unsigned char _pixels[MNIST_IMGSIZE * MNIST_IMGSIZE];
} MNISTImg;

void MNISTImgPrintln(MNISTImg* img) {
  for (int i = 0; i < MNIST_IMGSIZE; ++i) {
    for (int j = 0; j < MNIST_IMGSIZE; ++j) {
      if (img->_pixels[i * MNIST_IMGSIZE + j] > 127)
        printf("#");
      else
        printf(" ");
    }
    printf("\n");
  }
}

typedef struct MNIST {
  int _nbImg;
  MNISTImg* _imgs;
} MNIST;

void MNISTFree(MNIST** that) {
  free((*that)->_imgs);
  free(*that);
  *that = NULL;
}

typedef struct DataSet {
  // Category of the data set
  DataSetCat _cat;
  // Number of sample
  unsigned int _nbMNISTImg;
  // MNISTImgs
  MNISTImg* _samples;
} DataSet;

// Get the DataSetCat from its 'name'
```

```
DataSetCat GetCategoryFromName(const char* const name) {
  // Declare a variable to memorize the DataSetCat
  DataSetCat cat = unknownDataSet;
  // Search the dataset
  for (int iSet = NB_DATASET; iSet--;)
    if (strcmp(name, dataSetNames[iSet]) == 0)
      cat = iSet;
  // Return the category
  return cat;
}

// Load the data set of category 'cat' in the DataSet 'that'
// Return true on success, else false

MNIST* MNISTLoad(char* fnLbl, char* fnImg) {
  FILE* fLbl = fopen(fnLbl, "rb");
  if (!fLbl) {
    printf("Couldn't open %s\n", fnLbl);
    return NULL;
  }
  FILE* fImg = fopen(fnImg, "rb");
  if (!fImg) {
    printf("Couldn't open %s\n", fnImg);
    fclose(fLbl);
    return NULL;
  }
  MNIST* mnist = PBErrMalloc(&thePBErr, sizeof(MNIST));
  int buff;
  int ret;
  // Magic number
  for (int i = 4; i--;)
    ret = fread((char*)(&buff) + i, 1, 1, fLbl);
  if (buff != 2049) {
    printf("Magic number for %s is invalid (%d==2049)\n", fnLbl, buff);
    fclose(fLbl);
    fclose(fImg);
    return NULL;
  }
  for (int i = 4; i--;)
    ret = fread((char*)(&buff) + i, 1, 1, fImg);
  if (buff != 2051) {
    printf("Magic number for %s is invalid (%d==2051)\n", fnLbl, buff);
    fclose(fLbl);
    fclose(fImg);
    return NULL;
  }
  // Number of items
  for (int i = 4; i--;)
    ret = fread((char*)(&(mnist->_nbImg)) + i, 1, 1, fLbl);
  for (int i = 4; i--;)
    ret = fread((char*)(&buff) + i, 1, 1, fImg);
  if (buff != mnist->_nbImg) {
    printf("Nb of items doesn't match (%d==%d)\n", buff, mnist->_nbImg);
    fclose(fLbl);
    fclose(fImg);
    return NULL;
  }
  // Number of rows and columns
  for (int i = 4; i--;)
    ret = fread((char*)(&buff) + i, 1, 1, fImg);
  if (buff != 28) {
    printf("Unexpected image size (rows) (%d==%d)\n",
```

```
      buff, 28);
    fclose(fLbl);
    fclose(fImg);
    return NULL;
  }
  for (int i = 4; i--;)
    ret = fread((char*)(&buff) + i, 1, 1, fImg);
  if (buff != 28) {
    printf("Unexpected image size (columns) (%d==%d)\n",
      buff, 28);
    fclose(fLbl);
    fclose(fImg);
    return NULL;
  }
  // Images
  printf("Loading %d images...\n", mnist->_nbImg);
  mnist->_imgs =
    PBErrMalloc(&thePBErr, sizeof(MNISTImg) * mnist->_nbImg);
  for (int iImg = 0; iImg < mnist->_nbImg; ++iImg) {
    MNISTImg* img = mnist->_imgs + iImg;
    // Label
    ret = fread(&(img->_cat), 1, 1, fLbl);
    // Pixels
    for (int iPixel = 0; iPixel < MNIST_IMGSIZE * MNIST_IMGSIZE;
      ++iPixel) {
      ret = fread(img->_pixels + iPixel, 1, 1, fImg);
    }
  }
  printf("Loaded MNIST successfully.\n");
  fflush(stdout);
  fclose(fImg);
  fclose(fLbl);
  (void)ret;
  return mnist;
}

bool DataSetLoad(DataSet* const that, const DataSetCat cat) {
  // Set the category
  that->_cat = cat;
  // Load the data according to 'cat'
  MNIST* mnist =
    MNISTLoad("train-labels.idx1-ubyte", "train-images.idx3-ubyte");
  if (!mnist) {
    printf("Couldn't load the MNIST data\n");
    return false;
  }
  if (cat == datalearn) {
    that->_nbMNISTImg = 50000;
    that->_samples =
      PBErrMalloc(NeuraNetErr, sizeof(MNISTImg) * that->_nbMNISTImg);
    memcpy(that->_samples, mnist->_imgs,
      sizeof(MNISTImg) * that->_nbMNISTImg);
  } else if (cat == datatest) {
    that->_nbMNISTImg = 10000;
    that->_samples =
      PBErrMalloc(NeuraNetErr, sizeof(MNISTImg) * that->_nbMNISTImg);
    memcpy(that->_samples, mnist->_imgs + 50000,
      sizeof(MNISTImg) * that->_nbMNISTImg);
  } else if (cat == dataall) {
    that->_nbMNISTImg = 60000;
    that->_samples =
      PBErrMalloc(NeuraNetErr, sizeof(MNISTImg) * that->_nbMNISTImg);
```

```
    memcpy(that->_samples, mnist->_imgs,
      sizeof(MNISTImg) * that->_nbMNISTImg);
  } else {
    printf("Invalid dataset\n");
    MNISTFree(&mnist);
    return false;
  }
  MNISTFree(&mnist);
  printf("Created dataset with %u samples\n", that->_nbMNISTImg);
  fflush(stdout);
  // Return success code
  return true;
}

// Free memory for the DataSet 'that'
void DataSetFree(DataSet** that) {
  if (*that == NULL) return;
  // Free the memory

  free(*that);
  *that = NULL;
}

// Evalutation function for the NeuraNet 'that' on the DataSet 'dataset'
// Return the value of the NeuraNet, the bigger the better
float Evaluate(const NeuraNet* const that,
  const DataSet* const dataset) {
  // Declare 2 vectors to memorize the input and output values
  VecFloat* input = VecFloatCreate(NNGetNbInput(that));
  VecFloat* output = VecFloatCreate(NNGetNbOutput(that));
  // Declare a variable to memorize the value
  float val = 0.0;

  // Evaluate

  int countCat[NB_OUTPUT] = {0};
  int countOk[NB_OUTPUT] = {0};
  int countNg[NB_OUTPUT] = {0};
  for (unsigned int iMNISTImg = dataset->_nbMNISTImg; iMNISTImg--;) {
    // batching
    //if (dataset->_cat != datalearn || rnd() < 0.1) {
    if (dataset->_cat != datalearn || rnd() < 1.0) {
      for (int iInp = 0; iInp < NNGetNbInput(that); ++iInp) {
        VecSet(input, iInp,
          dataset->_samples[iMNISTImg]._pixels[iInp]);
      }
      NNEval(that, input, output);
      int pred = VecGetIMaxVal(output);
      ++(countCat[dataset->_samples[iMNISTImg]._cat]);
      if (pred == dataset->_samples[iMNISTImg]._cat) {
        ++(countOk[dataset->_samples[iMNISTImg]._cat]);
      } else if (dataset->_cat == datalearn) {
        ++(countNg[dataset->_samples[iMNISTImg]._cat]);
      }
    }
  }
  int nbCat = 0;
  for (int iCat = 0; iCat < NB_OUTPUT; ++iCat) {
    if (countCat[iCat] > 0) {
      ++nbCat;
      float perc = 0.0;
      if (dataset->_cat != datalearn) {
```

```
        perc = (float)(countOk[iCat]) / (float)(countCat[iCat]);
        printf("%10s (%4d): %f\n",
          catNames[iCat], countCat[iCat], perc);
        val += countOk[iCat];
      } else {
        perc = (float)(countOk[iCat] - countNg[iCat]) /
          (float)(countCat[iCat]);
        val += perc;
      }
    }
  }
  if (dataset->_cat != datalearn)
    val /= (float)(dataset->_nbMNISTImg);
  else
    val /= (float)nbCat;

  // Free memory
  VecFree(&input);
  VecFree(&output);
  // Return the result of the evaluation
  return val;
}

// Create the NeuraNet
NeuraNet* createNN(void) {
#if CONVOLUTION
  // Create the NeuraNet
  int nbOut = NB_OUTPUT;
  int thickConv = 5;
  int depthConv = 2;
  VecShort* dimIn = VecShortCreate(2);
  VecSet(dimIn, 0, MNIST_IMGSIZE);
  VecSet(dimIn, 1, MNIST_IMGSIZE);
  VecShort* dimCell = VecShortCreate(2);
  VecSet(dimCell, 0, 5);
  VecSet(dimCell, 1, 5);
  printf("Creating convoluted NeuraNet...\n");
  fflush(stdout);
  NeuraNet* nn = NeuraNetCreateConvolution(dimIn, nbOut, dimCell,
    depthConv, thickConv);
  printf("Created convoluted NeuraNet\n");
  printf("%ld links, %ld bases\n", NNGetNbMaxLinks(nn),
    NNGetNbMaxBases(nn));
  fflush(stdout);
  // Return the NeuraNet
  return nn;
#else
  // Create the NeuraNet
  int nbIn = NB_INPUT;
  int nbOut = NB_OUTPUT;
  int nbMaxHid = NB_MAXHIDDEN;
  int nbMaxLink = NB_MAXLINK;
  int nbMaxBase = NB_MAXBASE;
  NeuraNet* nn =
    NeuraNetCreate(nbIn, nbOut, nbMaxHid, nbMaxBase, nbMaxLink);
  // Return the NeuraNet
  return nn;
#endif
}

// Learn based on the SataSetCat 'cat'
void Learn(DataSetCat cat) {
```

```
  // Init the random generator
  srandom(time(NULL));
  // Declare variables to measure time
  struct timespec start, stop;
  // Start measuring time
  clock_gettime(CLOCK_REALTIME, &start);
  // Load the DataSet
  DataSet* dataset = PBErrMalloc(NeuraNetErr, sizeof(DataSet));
  bool ret = DataSetLoad(dataset, cat);
  if (!ret) {
    printf("Couldn't load the data\n");
    return;
  }
  // Create the NeuraNet
  NeuraNet* nn = createNN();
  // Declare a variable to memorize the best value
  float bestVal = INIT_BEST_VAL;
  // Declare a variable to memorize the limit in term of epoch
  unsigned long int limitEpoch = STOP_LEARNING_AT_EPOCH;
  // Create the GenAlg used for learning
  // If previous weights are available in "./bestga.txt" reload them
  GenAlg* ga = NULL;
  FILE* fd = fopen("./bestga.txt", "r");
  if (fd) {
    printf("Reloading previous GenAlg...\n");
    if (!GALoad(&ga, fd)) {
      printf("Failed to reload the GenAlg.\n");
      NeuraNetFree(&nn);
      DataSetFree(&dataset);
      return;
    } else {
      printf("Previous GenAlg reloaded.\n");
      if (GABestAdnF(ga) != NULL)
        NNSetBases(nn, GABestAdnF(ga));
#if CONVOLUTION == false
      if (GABestAdnI(ga) != NULL)
        NNSetLinks(nn, GABestAdnI(ga));
#endif
      bestVal = Evaluate(nn, dataset);
      printf("Starting with best at %f.\n", bestVal);
      limitEpoch += GAGetCurEpoch(ga);
    }
    fclose(fd);
  } else {
    printf("Creating new GenAlg...\n");
    fflush(stdout);
#if CONVOLUTION
    ga = GenAlgCreate(ADN_SIZE_POOL, ADN_SIZE_ELITE,
      NNGetGAAdnFloatLength(nn), 0);
    NNSetGABoundsBases(nn, ga);
#else
    ga = GenAlgCreate(ADN_SIZE_POOL, ADN_SIZE_ELITE,
      NNGetGAAdnFloatLength(nn), NNGetGAAdnIntLength(nn));
    NNSetGABoundsBases(nn, ga);
    NNSetGABoundsLinks(nn, ga);
#endif
    // Must be declared as a GenAlg applied to a NeuraNet with
    // convolution
#if CONVOLUTION
    GASetTypeNeuraNetConv(ga, NB_INPUT, NNGetNbMaxHidden(nn), NB_OUTPUT,
      NNGetNbBasesConv(nn), NNGetNbBasesCellConv(nn));
#else
```

```
    GASetTypeNeuraNet(ga, NB_INPUT, NB_MAXHIDDEN, NB_OUTPUT);
#endif
    GAInit(ga);
  }
  // If there is a NeuraNet available, reload it into the GenAlg
  fd = fopen("./bestnn.txt", "r");
  if (fd) {
    printf("Reloading previous NeuraNet...\n");
    if (!NNLoad(&nn, fd)) {
      printf("Failed to reload the NeuraNet.\n");
      NeuraNetFree(&nn);
      DataSetFree(&dataset);
      return;
    } else {
      printf("Previous NeuraNet reloaded.\n");
      bestVal = Evaluate(nn, dataset);
      printf("Starting with best at %f.\n", bestVal);
      GenAlgAdn* adn = GAAdn(ga, 0);
      VecCopy(adn->_adnF, nn->_bases);
      VecCopy(adn->_adnI, nn->_links);
    }
    fclose(fd);
  }
  // Start learning process
  printf("Learning...\n");
  printf("Will stop when curEpoch >= %lu or bestVal >= %f\n",
    limitEpoch, STOP_LEARNING_AT_VAL);
  printf("Will save the best NeuraNet in ./bestnn.txt at each improvement\n");
  fflush(stdout);
  // Declare a variable to memorize the best value in the current epoch
  float curBest = 0.0;
  float curWorst = 0.0;
  // Declare a variable to manage the save of GenAlg
  int delaySave = 0;
  // Learning loop
  while (bestVal < STOP_LEARNING_AT_VAL &&
    GAGetCurEpoch(ga) < limitEpoch) {
    curWorst = curBest;
    curBest = INIT_BEST_VAL;
    int curBestI = 0;
    unsigned long int ageBest = 0;
    // For each adn in the GenAlg
    //for (int iEnt = GAGetNbAdns(ga); iEnt--;) {
    for (int iEnt = 0; iEnt < GAGetNbAdns(ga); ++iEnt) {
      // Get the adn
      GenAlgAdn* adn = GAAdn(ga, iEnt);
      // Set the links and base functions of the NeuraNet according
      // to this adn
      if (GABestAdnF(ga) != NULL)
        NNSetBases(nn, GAAdnAdnF(adn));
#if CONVOLUTION == false
      if (GABestAdnI(ga) != NULL)
        NNSetLinks(nn, GAAdnAdnI(adn));
#endif
      // Evaluate the NeuraNet
      float value = Evaluate(nn, dataset);
      // Update the value of this adn
      GASetAdnValue(ga, adn, value);
      // Update the best value in the current epoch
      if (value > curBest) {
        curBest = value;
        curBestI = iEnt;
```

```
        ageBest = GAAdnGetAge(adn);
      }
      if (value < curWorst)
        curWorst = value;
    }
    // Measure time
    clock_gettime(CLOCK_REALTIME, &stop);
    float elapsed = stop.tv_sec - start.tv_sec;
    int day = (int)floor(elapsed / 86400);
    elapsed -= (float)(day * 86400);
    int hour = (int)floor(elapsed / 3600);
    elapsed -= (float)(hour * 3600);
    int min = (int)floor(elapsed / 60);
    elapsed -= (float)(min * 60);
    int sec = (int)floor(elapsed);
    // If there has been improvement during this epoch
    if (curBest > bestVal) {
      bestVal = curBest;
      // Display info about the improvment
      printf("Improvement at epoch %05lu: %f(%03d) (in %02d:%02d:%02d:%02ds)       \n",
        GAGetCurEpoch(ga), bestVal, curBestI, day, hour, min, sec);
      fflush(stdout);
      // Set the links and base functions of the NeuraNet according
      // to the best adn
      GenAlgAdn* bestAdn = GAAdn(ga, curBestI);
      if (GAAdnAdnF(bestAdn) != NULL)
        NNSetBases(nn, GAAdnAdnF(bestAdn));
#if CONVOLUTION == false
      if (GAAdnAdnI(bestAdn) != NULL)
        NNSetLinks(nn, GAAdnAdnI(bestAdn));
#endif
      // Save the best NeuraNet
      fd = fopen("./bestnn.txt", "w");
      if (!NNSave(nn, fd, COMPACT)) {
        printf("Couldn't save the NeuraNet\n");
        NeuraNetFree(&nn);
        GenAlgFree(&ga);
        DataSetFree(&dataset);
        return;
      }
      fclose(fd);
    } else {
      fprintf(stderr,
        "Epoch %05lu: v%f a%03lu(%03d) kt%03lu ",
        GAGetCurEpoch(ga), curBest, ageBest, curBestI,
        GAGetNbKTEvent(ga));
      fprintf(stderr, "(in %02d:%02d:%02d:%02ds)  \r",
        day, hour, min, sec);
      fflush(stderr);
    }
    ++delaySave;
    if (SAVE_GA_EVERY != 0 && delaySave >= SAVE_GA_EVERY) {
      delaySave = 0;
      // Save the adns of the GenAlg, use a temporary file to avoid
      // loosing the previous one if something goes wrong during
      // writing, then replace the previous file with the temporary one
      fd = fopen("./bestga.tmp", "w");
      if (!GASave(ga, fd, COMPACT)) {
        printf("Couldn't save the GenAlg\n");
        NeuraNetFree(&nn);
        GenAlgFree(&ga);
        DataSetFree(&dataset);
```

```
        return;
      }
      fclose(fd);
      int ret = system("mv ./bestga.tmp ./bestga.txt");
      (void)ret;
    }
    // Step the GenAlg
    GAStep(ga);
  }
  // Measure time
  clock_gettime(CLOCK_REALTIME, &stop);
  float elapsed = stop.tv_sec - start.tv_sec;
  int day = (int)floor(elapsed / 86400);
  elapsed -= (float)(day * 86400);
  int hour = (int)floor(elapsed / 3600);
  elapsed -= (float)(hour * 3600);
  int min = (int)floor(elapsed / 60);
  elapsed -= (float)(min * 60);
  int sec = (int)floor(elapsed);
  printf("\nLearning complete (in %d:%d:%d:%ds)\n",
    day, hour, min, sec);
  fflush(stdout);
  // Free memory
  NeuraNetFree(&nn);
  GenAlgFree(&ga);
  DataSetFree(&dataset);
}

// Check the NeuraNet 'that' on the DataSetCat 'cat'
void Check(const NeuraNet* const that, const DataSetCat cat) {
  // Load the DataSet
  DataSet* dataset = PBErrMalloc(NeuraNetErr, sizeof(DataSet));
  bool ret = DataSetLoad(dataset, cat);
  if (!ret) {
    printf("Couldn't load the data\n");
    return;
  }
  // Evaluate the NeuraNet
  float value = Evaluate(that, dataset);
  // Display the result
  printf("Value: %.6f\n", value);
  // Free memory
  DataSetFree(&dataset);
}

// Predict using the NeuraNet 'that' on 'inputs' (given as an array of
// 'nbInp' char*)
void Predict(const NeuraNet* const that, const int nbInp,
  char** const inputs) {
  // Start measuring time
  clock_t clockStart = clock();
  // Check the number of inputs
  if (nbInp != NNGetNbInput(that)) {
    printf("Wrong number of inputs, there should %d, there was %d\n",
      NNGetNbInput(that), nbInp);
    return;
  }
  // Declare 2 vectors to memorize the input and output values
  VecFloat* input = VecFloatCreate(NNGetNbInput(that));
  VecFloat* output = VecFloatCreate(NNGetNbOutput(that));
  // Set the input
  for (int iInp = 0; iInp < nbInp; ++iInp) {
```

```
    float v = 0.0;
    sscanf(inputs[iInp], "%f", &v);
    VecSet(input, iInp, v);
  }
  // Predict
  NNEval(that, input, output);

  // End measuring time
  clock_t clockEnd = clock();
  double timeUsed =
    ((double)(clockEnd - clockStart)) / (CLOCKS_PER_SEC * 0.001) ;
  // If the clock has been reset meanwhile
  if (timeUsed < 0.0)
    timeUsed = 0.0;
  //if (VecGet(output, 0) == ...)
  //  printf("...(in %fms)", timeUsed);

  // Free memory
  VecFree(&input);
  VecFree(&output);
}

int main(int argc, char** argv) {
  // Declare a variable to memorize the mode (learning/checking)
  int mode = -1;
  // Declare a variable to memorize the dataset used
  DataSetCat cat = unknownDataSet;
  // Decode mode from arguments
  if (argc >= 3) {
    if (strcmp(argv[1], "-learn") == 0) {
      mode = 0;
      cat = GetCategoryFromName(argv[2]);
    } else if (strcmp(argv[1], "-check") == 0) {
      mode = 1;
      cat = GetCategoryFromName(argv[2]);
    } else if (strcmp(argv[1], "-predict") == 0) {
      mode = 2;
    }
  }
  // If the mode is invalid print some help
  if (mode == -1) {
    printf("Select a mode from:\n");
    printf("-learn <dataset name>\n");
    printf("-check <dataset name>\n");
    printf("-predict <input values>\n");
    return 0;
  }
  if (mode == 0) {
    Learn(cat);
  } else if (mode == 1) {
    NeuraNet* nn = NULL;
    FILE* fd = fopen("./bestnn.txt", "r");
    if (!NNLoad(&nn, fd)) {
      printf("Couldn't load the best NeuraNet\n");
      return 0;
    }
    fclose(fd);
    Check(nn, cat);
    NeuraNetFree(&nn);
  } else if (mode == 2) {
    NeuraNet* nn = NULL;
    FILE* fd = fopen("./bestnn.txt", "r");
```

```
    if (!NNLoad(&nn, fd)) {
      printf("Couldn't load the best NeuraNet\n");
      return 0;
    }
    fclose(fd);
    Predict(nn, argc - 2, argv + 2);
    NeuraNetFree(&nn);
  }
  // Return success code
  return 0;
}
```

7.6 ORHD

Source: https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

main.c:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "pberr.h"
#include "genalg.h"
#include "neuranet.h"

// https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

// Nb of step between each save of the GenAlg
// Saving it allows to restart a stop learning process but is
// very time consuming if there are many input/hidden/output
// If 0 never save
#define SAVE_GA_EVERY 100
// Nb input and output of the NeuraNet
#define NB_INPUT 64
#define NB_OUTPUT 10
// Nb max of hidden values, links and base functions
#define NB_MAXHIDDEN 100
#define NB_MAXLINK 300
#define NB_MAXBASE NB_MAXLINK
// Size of the gene pool and elite pool
#define ADN_SIZE_POOL 100
#define ADN_SIZE_ELITE 20
// Initial best value during learning, must be lower than any
// possible value returned by Evaluate()
#define INIT_BEST_VAL -1000.0
// Value of the NeuraNet above which the learning process stops
#define STOP_LEARNING_AT_VAL 0.999
// Number of epoch above which the learning process stops
#define STOP_LEARNING_AT_EPOCH 100000
// Save NeuraNet in compact format
#define COMPACT true
// Use convolution if true
#define CONVOLUTION false
```

```
// Categories of data sets

typedef enum DataSetCat {
  unknownDataSet,
  datalearn,
  datatest,
  dataall
} DataSetCat;
#define NB_DATASET 4
const char* dataSetNames[NB_DATASET] = {
  "unknownDataSet", "datalearn", "datatest", "dataall"
  };

const char* catNames[NB_OUTPUT] = {
  "0",
  "1",
  "2",
  "3",
  "4",
  "5",
  "6",
  "7",
  "8",
  "9",
};

// Structure for the data set

typedef struct ORHD {
  float _props[NB_INPUT];
  int _cat;
} ORHD;

typedef struct DataSet {
  // Category of the data set
  DataSetCat _cat;
  // Number of sample
  unsigned int _nbSample;
  // Samples
  ORHD* _samples;
} DataSet;

// Get the DataSetCat from its 'name'
DataSetCat GetCategoryFromName(const char* const name) {
  // Declare a variable to memorize the DataSetCat
  DataSetCat cat = unknownDataSet;
  // Search the dataset
  for (int iSet = NB_DATASET; iSet--;)
    if (strcmp(name, dataSetNames[iSet]) == 0)
      cat = iSet;
  // Return the category
  return cat;
}

// Load the data set of category 'cat' in the DataSet 'that'
// Return true on success, else false
// Load the data set of category 'cat' in the DataSet 'that'
// Return true on success, else false

bool DataSetLoad(DataSet* const that, const DataSetCat cat) {
  // Set the category
```

```
that->_cat = cat;

// Load the data according to 'cat'
int ret = 0;
if (cat == datalearn) {
  FILE* f = fopen("./optdigits.tra", "r");
  if (f == NULL) {
    printf("Couldn't open the data set file\n");
    return false;
  }
  that->_nbSample = 3823;
  that->_samples =
    PBErrMalloc(NeuraNetErr, sizeof(ORHD) * that->_nbSample);
  for (unsigned int iSample = 0; iSample < that->_nbSample; ++iSample) {
    ret = fscanf(f, "%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%
      that->_samples[iSample]._props,
      that->_samples[iSample]._props + 1,
      that->_samples[iSample]._props + 2,
      that->_samples[iSample]._props + 3,
      that->_samples[iSample]._props + 4,
      that->_samples[iSample]._props + 5,
      that->_samples[iSample]._props + 6,
      that->_samples[iSample]._props + 7,
      that->_samples[iSample]._props + 8,
      that->_samples[iSample]._props + 9,
      that->_samples[iSample]._props + 10,
      that->_samples[iSample]._props + 11,
      that->_samples[iSample]._props + 12,
      that->_samples[iSample]._props + 13,
      that->_samples[iSample]._props + 14,
      that->_samples[iSample]._props + 15,
      that->_samples[iSample]._props + 16,
      that->_samples[iSample]._props + 17,
      that->_samples[iSample]._props + 18,
      that->_samples[iSample]._props + 19,
      that->_samples[iSample]._props + 20,
      that->_samples[iSample]._props + 21,
      that->_samples[iSample]._props + 22,
      that->_samples[iSample]._props + 23,
      that->_samples[iSample]._props + 24,
      that->_samples[iSample]._props + 25,
      that->_samples[iSample]._props + 26,
      that->_samples[iSample]._props + 27,
      that->_samples[iSample]._props + 28,
      that->_samples[iSample]._props + 29,
      that->_samples[iSample]._props + 30,
      that->_samples[iSample]._props + 31,
      that->_samples[iSample]._props + 32,
      that->_samples[iSample]._props + 33,
      that->_samples[iSample]._props + 34,
      that->_samples[iSample]._props + 35,
      that->_samples[iSample]._props + 36,
      that->_samples[iSample]._props + 37,
      that->_samples[iSample]._props + 38,
      that->_samples[iSample]._props + 39,
      that->_samples[iSample]._props + 40,
      that->_samples[iSample]._props + 41,
      that->_samples[iSample]._props + 42,
      that->_samples[iSample]._props + 43,
      that->_samples[iSample]._props + 44,
      that->_samples[iSample]._props + 45,
      that->_samples[iSample]._props + 46,
```

```
        that->_samples[iSample]._props + 47,
        that->_samples[iSample]._props + 48,
        that->_samples[iSample]._props + 49,
        that->_samples[iSample]._props + 50,
        that->_samples[iSample]._props + 51,
        that->_samples[iSample]._props + 52,
        that->_samples[iSample]._props + 53,
        that->_samples[iSample]._props + 54,
        that->_samples[iSample]._props + 55,
        that->_samples[iSample]._props + 56,
        that->_samples[iSample]._props + 57,
        that->_samples[iSample]._props + 58,
        that->_samples[iSample]._props + 59,
        that->_samples[iSample]._props + 60,
        that->_samples[iSample]._props + 61,
        that->_samples[iSample]._props + 62,
        that->_samples[iSample]._props + 63,
        &(that->_samples[iSample]._cat));
      /*for (int iProp = 64; iProp--;) {
        that->_samples[iSample]._props[iProp] /= 8.0;
        that->_samples[iSample]._props[iProp] -= 1.0;
      }*/
      if (ret == EOF) {
        printf("Couldn't read the dataset\n");
        fclose(f);
        return false;
      }
    }
    fclose(f);
  } else if (cat == datatest) {
    FILE* f = fopen("./optdigits.tes", "r");
    if (f == NULL) {
      printf("Couldn't open the data set file\n");
      return false;
    }
    that->_nbSample = 1797;
    that->_samples =
      PBErrMalloc(NeuraNetErr, sizeof(ORHD) * that->_nbSample);
    for (unsigned int iSample = 0; iSample < that->_nbSample; ++iSample) {
      ret = fscanf(f, "%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%
        that->_samples[iSample]._props,
        that->_samples[iSample]._props + 1,
        that->_samples[iSample]._props + 2,
        that->_samples[iSample]._props + 3,
        that->_samples[iSample]._props + 4,
        that->_samples[iSample]._props + 5,
        that->_samples[iSample]._props + 6,
        that->_samples[iSample]._props + 7,
        that->_samples[iSample]._props + 8,
        that->_samples[iSample]._props + 9,
        that->_samples[iSample]._props + 10,
        that->_samples[iSample]._props + 11,
        that->_samples[iSample]._props + 12,
        that->_samples[iSample]._props + 13,
        that->_samples[iSample]._props + 14,
        that->_samples[iSample]._props + 15,
        that->_samples[iSample]._props + 16,
        that->_samples[iSample]._props + 17,
        that->_samples[iSample]._props + 18,
        that->_samples[iSample]._props + 19,
        that->_samples[iSample]._props + 20,
        that->_samples[iSample]._props + 21,
```

```
        that->_samples[iSample]._props + 22,
        that->_samples[iSample]._props + 23,
        that->_samples[iSample]._props + 24,
        that->_samples[iSample]._props + 25,
        that->_samples[iSample]._props + 26,
        that->_samples[iSample]._props + 27,
        that->_samples[iSample]._props + 28,
        that->_samples[iSample]._props + 29,
        that->_samples[iSample]._props + 30,
        that->_samples[iSample]._props + 31,
        that->_samples[iSample]._props + 32,
        that->_samples[iSample]._props + 33,
        that->_samples[iSample]._props + 34,
        that->_samples[iSample]._props + 35,
        that->_samples[iSample]._props + 36,
        that->_samples[iSample]._props + 37,
        that->_samples[iSample]._props + 38,
        that->_samples[iSample]._props + 39,
        that->_samples[iSample]._props + 40,
        that->_samples[iSample]._props + 41,
        that->_samples[iSample]._props + 42,
        that->_samples[iSample]._props + 43,
        that->_samples[iSample]._props + 44,
        that->_samples[iSample]._props + 45,
        that->_samples[iSample]._props + 46,
        that->_samples[iSample]._props + 47,
        that->_samples[iSample]._props + 48,
        that->_samples[iSample]._props + 49,
        that->_samples[iSample]._props + 50,
        that->_samples[iSample]._props + 51,
        that->_samples[iSample]._props + 52,
        that->_samples[iSample]._props + 53,
        that->_samples[iSample]._props + 54,
        that->_samples[iSample]._props + 55,
        that->_samples[iSample]._props + 56,
        that->_samples[iSample]._props + 57,
        that->_samples[iSample]._props + 58,
        that->_samples[iSample]._props + 59,
        that->_samples[iSample]._props + 60,
        that->_samples[iSample]._props + 61,
        that->_samples[iSample]._props + 62,
        that->_samples[iSample]._props + 63,
        &(that->_samples[iSample]._cat));
      /*for (int iProp = 64; iProp--;) {
        that->_samples[iSample]._props[iProp] /= 8.0;
        that->_samples[iSample]._props[iProp] -= 1.0;
      }*/
      if (ret == EOF) {
        printf("Couldn't read the dataset\n");
        fclose(f);
        return false;
      }
    }
    fclose(f);
  } else {
    printf("Invalid dataset\n");
    return false;
  }

  // Return success code
  return true;
}
```

```
// Free memory for the DataSet 'that'
void DataSetFree(DataSet** that) {
  if (*that == NULL) return;
  // Free the memory

  free(*that);
  *that = NULL;
}

// Evalutation function for the NeuraNet 'that' on the DataSet 'dataset'
// Return the value of the NeuraNet, the bigger the better
float Evaluate(const NeuraNet* const that,
  const DataSet* const dataset) {
  // Declare 2 vectors to memorize the input and output values
  VecFloat* input = VecFloatCreate(NNGetNbInput(that));
  VecFloat* output = VecFloatCreate(NNGetNbOutput(that));
  // Declare a variable to memorize the value
  float val = 0.0;

  // Evaluate

  int countCat[NB_OUTPUT] = {0};
  int countOk[NB_OUTPUT] = {0};
  int countNg[NB_OUTPUT] = {0};
  NNPrune(that);
  for (unsigned int iSample = dataset->_nbSample; iSample--;) {
    // batching
    //if (dataset->_cat != datalearn || rnd() < 0.5) {
    if (dataset->_cat != datalearn || rnd() < 1.0) {
      for (int iInp = 0; iInp < NNGetNbInput(that); ++iInp) {
        VecSet(input, iInp,
          dataset->_samples[iSample]._props[iInp]);
      }
      NNEval(that, input, output);
      int pred = VecGetIMaxVal(output);
      ++(countCat[dataset->_samples[iSample]._cat]);
      if (pred == dataset->_samples[iSample]._cat) {
        ++(countOk[dataset->_samples[iSample]._cat]);
      } else if (dataset->_cat == datalearn) {
        ++(countNg[dataset->_samples[iSample]._cat]);
      }
    }
  }
  int nbCat = 0;
  for (int iCat = 0; iCat < NB_OUTPUT; ++iCat) {
    if (countCat[iCat] > 0) {
      ++nbCat;
      float perc = 0.0;
      if (dataset->_cat != datalearn) {
        perc = (float)(countOk[iCat]) / (float)(countCat[iCat]);
        printf("%10s (%4d): %f\n",
          catNames[iCat], countCat[iCat], perc);
        val += countOk[iCat];
      } else {
        perc = (float)(countOk[iCat] - countNg[iCat]) /
          (float)(countCat[iCat]);
        val += perc;
      }
    }
  }
  if (dataset->_cat != datalearn)
```

```
    val /= (float)(dataset->_nbSample);
  else
    val /= (float)nbCat;

  // Free memory
  VecFree(&input);
  VecFree(&output);
  // Return the result of the evaluation
  return val;
}

// Create the NeuraNet
NeuraNet* createNN(void) {
#if CONVOLUTION
  // Create the NeuraNet
  int nbOut = NB_OUTPUT;
  int thickConv = 10;
  int depthConv = 1;
  VecShort* dimIn = VecShortCreate(2);
  VecSet(dimIn, 0, 8);
  VecSet(dimIn, 1, 8);
  VecShort* dimCell = VecShortCreate(2);
  VecSet(dimCell, 0, 8);
  VecSet(dimCell, 1, 8);
  printf("Creating convoluted NeuraNet...\n");
  fflush(stdout);
  NeuraNet* nn = NeuraNetCreateConvolution(dimIn, nbOut, dimCell,
    depthConv, thickConv);
  printf("Created convoluted NeuraNet\n");
  printf("%ld links, %ld bases, %ld hiddens\n", NNGetNbMaxLinks(nn),
    NNGetNbMaxBases(nn), NNGetNbMaxHidden(nn));
  fflush(stdout);
  // Return the NeuraNet
  return nn;
#else
  // Create the NeuraNet
  int nbIn = NB_INPUT;
  int nbOut = NB_OUTPUT;
  int nbMaxHid = NB_MAXHIDDEN;
  int nbMaxLink = NB_MAXLINK;
  int nbMaxBase = NB_MAXBASE;
  NeuraNet* nn =
    NeuraNetCreate(nbIn, nbOut, nbMaxHid, nbMaxBase, nbMaxLink);
  // Return the NeuraNet
  return nn;
#endif
}

// Learn based on the SataSetCat 'cat'
void Learn(DataSetCat cat) {
  // Init the random generator
  srandom(time(NULL));
  // Declare variables to measure time
  struct timespec start, stop;
  // Start measuring time
  clock_gettime(CLOCK_REALTIME, &start);
  // Load the DataSet
  DataSet* dataset = PBErrMalloc(NeuraNetErr, sizeof(DataSet));
  bool ret = DataSetLoad(dataset, cat);
  if (!ret) {
    printf("Couldn't load the data\n");
    return;
```

```
  }
  // Create the NeuraNet
  NeuraNet* nn = createNN();
  // Declare a variable to memorize the best value
  float bestVal = INIT_BEST_VAL;
  // Declare a variable to memorize the limit in term of epoch
  unsigned long int limitEpoch = STOP_LEARNING_AT_EPOCH;
  // Create the GenAlg used for learning
  // If previous weights are available in "./bestga.txt" reload them
  GenAlg* ga = NULL;
  FILE* fd = fopen("./bestga.txt", "r");
  if (fd) {
    printf("Reloading previous GenAlg...\n");
    if (!GALoad(&ga, fd)) {
      printf("Failed to reload the GenAlg.\n");
      NeuraNetFree(&nn);
      DataSetFree(&dataset);
      return;
    } else {
      printf("Previous GenAlg reloaded.\n");
      if (GABestAdnF(ga) != NULL)
        NNSetBases(nn, GABestAdnF(ga));
#if CONVOLUTION == false
      if (GABestAdnI(ga) != NULL)
        NNSetLinks(nn, GABestAdnI(ga));
#endif
      bestVal = Evaluate(nn, dataset);
      printf("Starting with best at %f.\n", bestVal);
      limitEpoch += GAGetCurEpoch(ga);
    }
    fclose(fd);
  } else {
    printf("Creating new GenAlg...\n");
    fflush(stdout);
#if CONVOLUTION
    ga = GenAlgCreate(ADN_SIZE_POOL, ADN_SIZE_ELITE,
      NNGetGAAdnFloatLength(nn), 0);
    NNSetGABoundsBases(nn, ga);
#else
    ga = GenAlgCreate(ADN_SIZE_POOL, ADN_SIZE_ELITE,
      NNGetGAAdnFloatLength(nn), NNGetGAAdnIntLength(nn));
    NNSetGABoundsBases(nn, ga);
    NNSetGABoundsLinks(nn, ga);
#endif
    // Must be declared as a GenAlg applied to a NeuraNet with
    // convolution
#if CONVOLUTION
    GASetTypeNeuraNetConv(ga, NB_INPUT, NNGetNbMaxHidden(nn), NB_OUTPUT,
      NNGetNbBasesConv(nn), NNGetNbBasesCellConv(nn));
#else
    GASetTypeNeuraNet(ga, NB_INPUT, NB_MAXHIDDEN, NB_OUTPUT);
#endif
    GAInit(ga);
  }
  // If there is a NeuraNet available, reload it into the GenAlg
  fd = fopen("./bestnn.txt", "r");
  if (fd) {
    printf("Reloading previous NeuraNet...\n");
    if (!NNLoad(&nn, fd)) {
      printf("Failed to reload the NeuraNet.\n");
      NeuraNetFree(&nn);
      DataSetFree(&dataset);
```

```
      return;
    } else {
      printf("Previous NeuraNet reloaded.\n");
      bestVal = Evaluate(nn, dataset);
      printf("Starting with best at %f.\n", bestVal);
      GenAlgAdn* adn = GAAdn(ga, 0);
      VecCopy(adn->_adnF, nn->_bases);
      VecCopy(adn->_adnI, nn->_links);
    }
    fclose(fd);
  }
  // Start learning process
  printf("Learning...\n");
  printf("Will stop when curEpoch >= %lu or bestVal >= %f\n",
    limitEpoch, STOP_LEARNING_AT_VAL);
  printf("Will save the best NeuraNet in ./bestnn.txt at each improvement\n");
  fflush(stdout);
  // Declare a variable to memorize the best value in the current epoch
  float curBest = 0.0;
  float curWorst = 0.0;
  // Declare a variable to manage the save of GenAlg
  int delaySave = 0;
  // Learning loop
  while (bestVal < STOP_LEARNING_AT_VAL &&
    GAGetCurEpoch(ga) < limitEpoch) {
    curWorst = curBest;
    curBest = INIT_BEST_VAL;
    int curBestI = 0;
    unsigned long int ageBest = 0;
    // For each adn in the GenAlg
    //for (int iEnt = GAGetNbAdns(ga); iEnt--;) {
    for (int iEnt = 0; iEnt < GAGetNbAdns(ga); ++iEnt) {
      // Get the adn
      GenAlgAdn* adn = GAAdn(ga, iEnt);
      // Set the links and base functions of the NeuraNet according
      // to this adn
      if (GABestAdnF(ga) != NULL)
        NNSetBases(nn, GAAdnAdnF(adn));
#if CONVOLUTION == false
      if (GABestAdnI(ga) != NULL)
        NNSetLinks(nn, GAAdnAdnI(adn));
#endif
      // Evaluate the NeuraNet
      float value = Evaluate(nn, dataset);
      value += NNGetHiddenValSimpsonDiv(nn) * 0.01;
      // Update the value of this adn
      GASetAdnValue(ga, adn, value);
      // Update the best value in the current epoch
      if (value > curBest) {
        curBest = value;
        curBestI = iEnt;
        ageBest = GAAdnGetAge(adn);
      }
      if (value < curWorst)
        curWorst = value;
    }
    // Measure time
    clock_gettime(CLOCK_REALTIME, &stop);
    float elapsed = stop.tv_sec - start.tv_sec;
    int day = (int)floor(elapsed / 86400);
    elapsed -= (float)(day * 86400);
    int hour = (int)floor(elapsed / 3600);
```

```
    elapsed -= (float)(hour * 3600);
    int min = (int)floor(elapsed / 60);
    elapsed -= (float)(min * 60);
    int sec = (int)floor(elapsed);
    // If there has been improvement during this epoch
    if (curBest > bestVal) {
      bestVal = curBest;
      // Display info about the improvment
      printf("Improvement at epoch %05lu: %f(%03d) (in %02d:%02d:%02d:%02ds)       \n",
        GAGetCurEpoch(ga), bestVal, curBestI, day, hour, min, sec);
      fflush(stdout);
      // Set the links and base functions of the NeuraNet according
      // to the best adn
      GenAlgAdn* bestAdn = GAAdn(ga, curBestI);
      if (GAAdnAdnF(bestAdn) != NULL)
        NNSetBases(nn, GAAdnAdnF(bestAdn));
#if CONVOLUTION == false
      if (GAAdnAdnI(bestAdn) != NULL)
        NNSetLinks(nn, GAAdnAdnI(bestAdn));
#endif
      // Save the best NeuraNet
      fd = fopen("./bestnn.txt", "w");
      if (!NNSave(nn, fd, COMPACT)) {
        printf("Couldn't save the NeuraNet\n");
        NeuraNetFree(&nn);
        GenAlgFree(&ga);
        DataSetFree(&dataset);
        return;
      }
      fclose(fd);
    } else {
      fprintf(stderr,
        "Epoch %05lu: v%f a%03lu(%03d) kt%03lu ",
        GAGetCurEpoch(ga), curBest, ageBest, curBestI,
        GAGetNbKTEvent(ga));
      fprintf(stderr, "(in %02d:%02d:%02d:%02ds)  \r",
        day, hour, min, sec);
      fflush(stderr);
    }
    ++delaySave;
    if (SAVE_GA_EVERY != 0 && delaySave >= SAVE_GA_EVERY) {
      delaySave = 0;
      // Save the adns of the GenAlg, use a temporary file to avoid
      // loosing the previous one if something goes wrong during
      // writing, then replace the previous file with the temporary one
      fd = fopen("./bestga.tmp", "w");
      if (!GASave(ga, fd, COMPACT)) {
        printf("Couldn't save the GenAlg\n");
        NeuraNetFree(&nn);
        GenAlgFree(&ga);
        DataSetFree(&dataset);
        return;
      }
      fclose(fd);
      int ret = system("mv ./bestga.tmp ./bestga.txt");
      (void)ret;
    }
    // Step the GenAlg
    GAStep(ga);
  }
  // Measure time
  clock_gettime(CLOCK_REALTIME, &stop);
```

```
  float elapsed = stop.tv_sec - start.tv_sec;
  int day = (int)floor(elapsed / 86400);
  elapsed -= (float)(day * 86400);
  int hour = (int)floor(elapsed / 3600);
  elapsed -= (float)(hour * 3600);
  int min = (int)floor(elapsed / 60);
  elapsed -= (float)(min * 60);
  int sec = (int)floor(elapsed);
  printf("\nLearning complete (in %d:%d:%d:%ds)\n",
    day, hour, min, sec);
  fflush(stdout);
  // Free memory
  NeuraNetFree(&nn);
  GenAlgFree(&ga);
  DataSetFree(&dataset);
}

// Check the NeuraNet 'that' on the DataSetCat 'cat'
void Check(const NeuraNet* const that, const DataSetCat cat) {
  // Load the DataSet
  DataSet* dataset = PBErrMalloc(NeuraNetErr, sizeof(DataSet));
  bool ret = DataSetLoad(dataset, cat);
  if (!ret) {
    printf("Couldn't load the data\n");
    return;
  }
  // Evaluate the NeuraNet
  float value = Evaluate(that, dataset);
  // Display the result
  printf("Value: %.6f\n", value);
  // Free memory
  DataSetFree(&dataset);
}

// Predict using the NeuraNet 'that' on 'inputs' (given as an array of
// 'nbInp' char*)
void Predict(const NeuraNet* const that, const int nbInp,
  char** const inputs) {
  // Start measuring time
  clock_t clockStart = clock();
  // Check the number of inputs
  if (nbInp != NNGetNbInput(that)) {
    printf("Wrong number of inputs, there should %d, there was %d\n",
      NNGetNbInput(that), nbInp);
    return;
  }
  // Declare 2 vectors to memorize the input and output values
  VecFloat* input = VecFloatCreate(NNGetNbInput(that));
  VecFloat* output = VecFloatCreate(NNGetNbOutput(that));
  // Set the input
  for (int iInp = 0; iInp < nbInp; ++iInp) {
    float v = 0.0;
    sscanf(inputs[iInp], "%f", &v);
    VecSet(input, iInp, v);
  }
  // Predict
  NNEval(that, input, output);

  // End measuring time
  clock_t clockEnd = clock();
  double timeUsed =
    ((double)(clockEnd - clockStart)) / (CLOCKS_PER_SEC * 0.001) ;
```

```
  // If the clock has been reset meanwhile
  if (timeUsed < 0.0)
    timeUsed = 0.0;
  //if (VecGet(output, 0) == ...)
  //  printf("...(in %fms)", timeUsed);

  // Free memory
  VecFree(&input);
  VecFree(&output);
}

int main(int argc, char** argv) {
  // Declare a variable to memorize the mode (learning/checking)
  int mode = -1;
  // Declare a variable to memorize the dataset used
  DataSetCat cat = unknownDataSet;
  // Decode mode from arguments
  if (argc >= 3) {
    if (strcmp(argv[1], "-learn") == 0) {
      mode = 0;
      cat = GetCategoryFromName(argv[2]);
    } else if (strcmp(argv[1], "-check") == 0) {
      mode = 1;
      cat = GetCategoryFromName(argv[2]);
    } else if (strcmp(argv[1], "-predict") == 0) {
      mode = 2;
    }
  }
  // If the mode is invalid print some help
  if (mode == -1) {
    printf("Select a mode from:\n");
    printf("-learn <dataset name>\n");
    printf("-check <dataset name>\n");
    printf("-predict <input values>\n");
    return 0;
  }
  if (mode == 0) {
    Learn(cat);
  } else if (mode == 1) {
    NeuraNet* nn = NULL;
    FILE* fd = fopen("./bestnn.txt", "r");
    if (!NNLoad(&nn, fd)) {
      printf("Couldn't load the best NeuraNet\n");
      return 0;
    }
    fclose(fd);
    Check(nn, cat);
    NeuraNetFree(&nn);
  } else if (mode == 2) {
    NeuraNet* nn = NULL;
    FILE* fd = fopen("./bestnn.txt", "r");
    if (!NNLoad(&nn, fd)) {
      printf("Couldn't load the best NeuraNet\n");
      return 0;
    }
    fclose(fd);
    Predict(nn, argc - 2, argv + 2);
    NeuraNetFree(&nn);
  }
  // Return success code
  return 0;
}
```

8 nn2cloud

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "pberr.h"
#include "neuranet.h"

int main(int argc, char **argv) {
  char* NNUrl = NULL;
  // Decode arguments
  bool flagPrune = false;
  for (int iArg = 0; iArg < argc; ++iArg) {
    if (strcmp(argv[iArg] , "-nn") == 0 && iArg + 1 < argc) {
      NNUrl = argv[iArg + 1];
      ++iArg;
    } else if (strcmp(argv[iArg] , "-prune") == 0) {
      flagPrune = true;
    } else if (strcmp(argv[iArg] , "-help") == 0) {
      printf("arguments : -nn <NeuraNet file url>\n");
      // Stop here
      return 0;
    }
  }
  if (NNUrl == NULL)
    return 1;
  FILE* fd = fopen(NNUrl, "r");
  NeuraNet* nn = NULL;
  if (NNLoad(&nn, fd) == false) {
    fprintf(stderr, "Failed to open the NeuraNet %s\n", NNUrl);
  }
  fclose(fd);
  if (flagPrune)
    NNPrune(nn);
  char* cloudUrl = "./cloud.txt";
  if (NNSaveLinkAsCloudGraph(nn, cloudUrl) == false) {
    fprintf(stderr, "Failed to save the CloudGraph %s\n", cloudUrl);
  }
  NeuraNetFree(&nn);
  // Return success code
  return 0;
}
```