NeuraNet

P. Baillehache

August 15, 2018

Contents

1 Definitions 2

2 Interface 3

3 Code 8

3.1 pbmath.c 8

3.2 pbmath-inline.c 17

4 Makefile 21

5 Unit tests 21

6 Unit tests output 29

7 Validation 31

7.1 Iris data set 31

7.2 Abalone data set 42

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.

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 NN_NBPARAMBASE float all in [-1,1]
// 'x' is the input value, in [-1,1]
// NNBaseFun(param,x)=
// {tan(param[0]*NN_THETA)*(x+param[1])+param[2]}[-1,1]
// The generic base function returns a value in [-1,1]
#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 int _nbMaxHidVal;
  // Nb max of base functions
  const int _nbMaxBases;
  // Nb max of links
  const int _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)
  VecShort* _links;
  // Hidden values
  VecFloat* _hidVal;
} 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 int nbMaxHidden, const int nbMaxBases, const int 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 VecShort* const hiddenLayers);

// 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
int NNGetNbMaxHidden(const NeuraNet* const that);

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

// Get the nb max of links of the NeuraNet 'that'
#if BUILDMODE != 0
inline
#endif
```

```
int 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 VecShort* 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 int 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 int 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, VecShort* 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);

// ================= 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 int NNGetGAAdnFloatLength(const NeuraNet* const that)
  __attribute__((unused));
static int 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 int NNGetGAAdnIntLength(const NeuraNet* const that)
  __attribute__((unused));
static int 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 (%d==%d)",
      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 (int 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 (%d==%d)",
      GAGetLengthAdnInt(ga), NNGetGAAdnIntLength(that));
    PBErrCatch(NeuraNetErr);
  }
#endif
  // Declare a vector to memorize the bounds
  VecShort2D bounds = VecShortCreateStatic2D();
  // For each gene
  for (int 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 int nbMaxHidden, const int nbMaxBases, const int 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<=%d)",
      nbMaxHidden);
    PBErrCatch(NeuraNetErr);
  }
  if (nbMaxBases <= 0) {
    NeuraNetErr->_type = PBErrTypeInvalidArg;
    sprintf(NeuraNetErr->_msg, "'nbMaxBases' is invalid (0<%d)",
      nbMaxBases);
    PBErrCatch(NeuraNetErr);
  }
  if (nbMaxLinks <= 0) {
    NeuraNetErr->_type = PBErrTypeInvalidArg;
    sprintf(NeuraNetErr->_msg, "'nbMaxLinks' is invalid (0<%d)",
      nbMaxLinks);
    PBErrCatch(NeuraNetErr);
  }
#endif
  // Declare the new NeuraNet
  NeuraNet* that = PBErrMalloc(NeuraNetErr, sizeof(NeuraNet));
  // Set properties
  *(int*)&(that->_nbInputVal) = nbInput;
  *(int*)&(that->_nbOutputVal) = nbOutput;
  *(int*)&(that->_nbMaxHidVal) = nbMaxHidden;
  *(int*)&(that->_nbMaxBases) = nbMaxBases;
  *(int*)&(that->_nbMaxLinks) = nbMaxLinks;
  that->_bases = VecFloatCreate(nbMaxBases * NN_NBPARAMBASE);
  that->_links = VecShortCreate(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 VecShort* 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
  int nbHiddenVal = 0;
  int nbBases = 0;
  int nbLinks = 0;
  int nbHiddenLayer = 0;
  // If there are hidden layers
  if (hiddenLayers != NULL) {
    // Get the number of hidden layers
    nbHiddenLayer = VecGetDim(hiddenLayers);
    // Declare two variables for computation
    int nIn = nbIn;
    int nOut = 0;
    // Calculate the nb of links and hidden values
    for (int 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
  int iLink = 0;
  // Declare variables for computation
  int shiftIn = 0;
  int shiftOut = nbIn;
  int nIn = nbIn;
  int nOut = 0;
  // Loop on hidden layers
  for (int iLayer = 0; iLayer <= nbHiddenLayer; ++iLayer) {
    // Init the links
    if (iLayer < nbHiddenLayer)
      nOut = VecGet(hiddenLayers, iLayer);
    else
      nOut = nbOut;
    for (int iIn = 0; iIn < nIn; ++iIn) {
      for (int iOut = 0; iOut < nOut; ++iOut) {
        int 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;
}

// 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 (%d!=%d)",
      VecGetDim(input), that->_nbInputVal);
    PBErrCatch(NeuraNetErr);
  }
  if (VecGetDim(output) != that->_nbOutputVal) {
    NeuraNetErr->_type = PBErrTypeInvalidArg;
    sprintf(NeuraNetErr->_msg,
      "'output' 's dimension is invalid (%d!=%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
    int startHid = NNGetNbInput(that);
    int startOut = NNGetNbMaxHidden(that) + NNGetNbInput(that);
    // Declare a variable to memorize the previous link
    int prevLink[2] = {-1, -1};
    // Declare a variable to memorize the previous output value
    float prevOut = 1.0;
    // Loop on links
    int iLink = 0;
    while (iLink < NNGetNbMaxLinks(that) &&
      VecGet(that->_links, NN_NBPARAMLINK * iLink) != -1) {
      // Declare a variable for optimization
      int 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) {
          int iVal = prevLink[1] - startHid;
          float nVal = MIN(1.0, MAX(-1.0, VecGet(that->_hidVal, iVal) + prevOut));
          VecSet(that->_hidVal, iVal, nVal);
        } else {
          int 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) {
      int iVal = prevLink[1] - startHid;
      float nVal = MIN(1.0, MAX(-1.0, VecGet(that->_hidVal, iVal) + prevOut));
      VecSet(that->_hidVal, iVal, nVal);
    } else {
      int 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, "%d", that->_nbMaxHidVal);
  JSONAddProp(json, "_nbMaxHidVal", val);
  // Encode the nbMaxBases
  sprintf(val, "%d", that->_nbMaxBases);
  JSONAddProp(json, "_nbMaxBases", val);
  // Encode the nbMaxLinks
  sprintf(val, "%d", 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;
  }
  int nbMaxHidVal = atoi(JSONLabel(JSONValue(prop, 0)));
  // Decode the nbMaxBases
  prop = JSONProperty(json, "_nbMaxBases");
  if (prop == NULL) {
    return false;
  }
  int nbMaxBases = atoi(JSONLabel(JSONValue(prop, 0)));
  // Decode the nbMaxLinks
  prop = JSONProperty(json, "_nbMaxLinks");
  if (prop == NULL) {
    return false;
  }
  int nbMaxLinks = atoi(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: %d\n", that->_nbMaxHidVal);
  fprintf(stream, "nbMaxBases: %d\n", that->_nbMaxBases);
  fprintf(stream, "nbMaxLinks: %d\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, VecShort* 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 (%d!=%d)",
      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
  int maxId = NNGetNbInput(that) + NNGetNbMaxHidden(that) +
    NNGetNbOutput(that);
  // Loop on links
  for (int iLink = 0; iLink < NNGetNbMaxLinks(that) * NN_NBPARAMLINK;
    iLink += NN_NBPARAMLINK) {
    // If this link is active
    if (VecGet(links, iLink) != -1) {
      // Declare two variable to memorize the effective input and output
      int in = VecGet(links, iLink + 1);
      int out = VecGet(links, iLink + 2);
      // If the input is greater than the output
      if (in > out) {
        // Swap the input and output
        int 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
  int nbLink = GSetNbElem(&set);
  // If there are active links
  if (nbLink > 0) {
    // loop on active sorted links
    GSetIterForward iter = GSetIterForwardCreateStatic(&set);
    int iLink = 0;
    do {
      short *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 (int iLink = nbLink; iLink < NNGetNbMaxLinks(that); ++iLink)
    VecSet(that->_links, iLink * NN_NBPARAMLINK, -1);
  // Free the memory
  GSetFlush(&set);
}
```

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]}[-1,1]
// The generic base function returns a value in [-1,1]
#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 MIN(1.0, MAX(-1.0,
  //  tan(param[0] * NN_THETA) * (x + param[1]) + param[2]));
  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
int 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
int 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 max of links of the NeuraNet 'that'
```

```
#if BUILDMODE != 0
inline
#endif
int 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 VecShort* 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 int 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<=%d<%d)",
      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 (%d!=%d)",
      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 int 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<=%d<%d)",
      iBase, that->_nbMaxBases * NN_NBPARAMBASE);
    PBErrCatch(NeuraNetErr);
  }
#endif
  VecSet(that->_bases, iBase, base);
}
```

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

# 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)/
```

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->_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;
  VecShort* 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 = VecShortCreate(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 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 (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);
  VecShort* links = VecShortCreate(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);
    }
  VecFree(&links);
  NeuraNetFree(&nn);
  printf("UnitTestNeuraNetGetSet OK\n");
}

void UnitTestNeuraNetSaveLoad() {
  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);
  VecShort* links = VecShortCreate(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);
  NeuraNetFree(&nn);
  printf("UnitTestNeuraNetSaveLoad OK\n");
}

void UnitTestNeuraNetEvalPrint() {
  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};
  VecShort *links = VecShortCreate(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);
        }
      }
    }
  }
  NeuraNetFree(&nn);
  printf("UnitTestNeuraNetEvalPrint 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 = 3;
  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) < 100 && 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();
  UnitTestNeuraNetGetSet();
  UnitTestNeuraNetSaveLoad();
  UnitTestNeuraNetEvalPrint();
#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
UnitTestNeuraNetGetSet OK
UnitTestNeuraNetSaveLoad 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>
UnitTestNeuraNetEvalPrint OK
1 -0.589407
2 -0.549744
4 -0.358829
5 -0.310591
11 -0.310447
13 -0.294536
17 -0.289736
18 -0.272652
19 -0.267634
21 -0.258228
25 -0.257314
26 -0.254666
30 -0.254224
32 -0.253986
36 -0.231163
37 -0.194898
41 -0.188746
44 -0.187099
82 -0.186968
84 -0.186710
90 -0.176221
93 -0.170388
94 -0.170171
95 -0.168599
97 -0.168410
99 -0.168334
100 -0.165676
108 -0.164237
112 -0.163052
138 -0.162231
140 -0.162206
256 -0.156286
427 -0.155642
429 -0.154719
445 -0.153960
462 -0.153806
488 -0.152242
499 -0.152129
502 -0.152017
666 -0.151985
1063 -0.151534
1828 -0.151175
2982 -0.150823
2995 -0.150756
3157 -0.150019
```

```
3262 -0.149638
3279 -0.149605
3286 -0.149587
4378 -0.149536
4409 -0.149486
4544 -0.149472
5115 -0.149416
5225 -0.148970
5231 -0.148939
5240 -0.148746
5326 -0.148682
5448 -0.148668
7855 -0.148618
8307 -0.148275
8748 -0.135853
8755 -0.116550
8764 -0.078999
8770 -0.071358
9107 -0.060294
9140 -0.057081
9147 -0.039839
9151 -0.037915
9153 -0.029160
9220 -0.028635
9240 -0.028070
9256 -0.019758
9723 -0.017609
9813 -0.014870
10196 -0.014759
10384 -0.014370
best after 30000 epochs: -0.014370
nbInput: 3
nbOutput: 3
nbHidden: 3
nbMaxBases: 3
nbMaxLinks: 7
bases: <-0.490,0.939,0.919,0.496,0.915,-0.867,0.303,0.226,-0.135>
links: <1,0,5,2,0,5,2,1,6,1,1,7,1,3,3,0,3,7,0,5,6>
hidden values: <0.036,0.000,0.508>
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":"3",
  "_nbMaxLinks":"7",
  "_bases":{
    "_dim":"9",
    "_val":["-0.490231","0.939080","0.918935","0.495953","0.914827","-0.867214","0.302704","0.226264","-0.134999"]
  },
  "_links":{
    "_dim":"21",
    "_val":["1","0","5","2","0","5","2","1","6","1","1","7","1","3","3","0","3","7","0","5","6"]
  }
}
```

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(%02d) 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;
}
```

learning:

```
Learning...
Will stop when curEpoch >= 2000 or bestVal >= 0.999000
Will save the best NeuraNet in ./bestnn.txt at each improvement
Improvement at epoch 00000: 0.666667(011) (in 00:00:00:00s)
Improvement at epoch 00004: 0.693333(025) (in 00:00:00:00s)
Improvement at epoch 00008: 0.960000(063) (in 00:00:00:00s)
Improvement at epoch 00010: 0.973333(084) (in 00:00:00:00s)

Learning complete (in 0:0:0:6s)
```

validation (with confidence vector):

```
#74 pred2 real2 <-0.576,0.694,0.999> OK
#73 pred2 real2 <-0.659,0.841,2.353> OK
#72 pred2 real2 <-0.604,0.753,1.541> OK
#71 pred2 real2 <-0.548,0.724,1.270> OK
#70 pred2 real2 <-0.604,0.841,2.353> OK
#69 pred2 real2 <-0.743,0.899,2.895> OK
#68 pred2 real2 <-0.798,0.841,2.353> OK
```

```
#67 pred2 real2 <-0.576,0.724,1.270> OK
#66 pred2 real2 <-0.576,0.841,2.353> OK
#65 pred2 real2 <-0.715,0.870,2.624> OK
#64 pred2 real2 <-0.659,0.782,1.812> OK
#63 pred2 real2 <-0.493,0.694,0.999> OK
#62 pred2 real2 <-0.687,0.694,0.999> OK
#61 pred2 real2 <-0.715,0.870,2.624> OK
#60 pred2 real2 <-0.854,0.841,2.353> OK
#59 pred1 real2 <-0.715,0.577,-0.084> NG
#58 pred1 real2 <-0.576,0.606,0.187> NG
#57 pred2 real2 <-0.715,0.811,2.083> OK
#56 pred2 real2 <-0.937,0.753,1.541> OK
#55 pred2 real2 <-0.854,0.724,1.270> OK
#54 pred1 real2 <-0.770,0.636,0.458> NG
#53 pred2 real2 <-0.715,0.782,1.812> OK
#52 pred2 real2 <-0.520,0.694,0.999> OK
#51 pred2 real2 <-0.493,0.694,0.999> OK
#50 pred2 real2 <-0.826,0.694,0.999> OK
#49 pred1 real1 <-0.298,0.548,-0.355> OK
#48 pred1 real1 <0.007,0.489,-0.896> OK
#47 pred1 real1 <-0.354,0.548,-0.355> OK
#46 pred1 real1 <-0.326,0.548,-0.355> OK
#45 pred1 real1 <-0.326,0.518,-0.625> OK
#44 pred1 real1 <-0.326,0.548,-0.355> OK
#43 pred1 real1 <-0.076,0.460,-1.167> OK
#42 pred1 real1 <-0.270,0.518,-0.625> OK
#41 pred1 real1 <-0.437,0.577,-0.084> OK
#40 pred1 real1 <-0.381,0.518,-0.625> OK
#39 pred1 real1 <-0.270,0.548,-0.355> OK
#38 pred1 real1 <-0.298,0.548,-0.355> OK
#37 pred1 real1 <-0.381,0.548,-0.355> OK
#36 pred1 real1 <-0.465,0.606,0.187> OK
#35 pred1 real1 <-0.409,0.636,0.458> OK
#34 pred1 real1 <-0.409,0.606,0.187> OK
#33 pred1 real1 <-0.576,0.636,0.458> OK
#32 pred1 real1 <-0.243,0.518,-0.625> OK
#31 pred1 real1 <-0.187,0.460,-1.167> OK
#30 pred1 real1 <-0.215,0.489,-0.896> OK
#29 pred1 real1 <-0.132,0.460,-1.167> OK
#28 pred1 real1 <-0.409,0.606,0.187> OK
#27 pred2 real1 <-0.548,0.665,0.729> NG
#26 pred1 real1 <-0.493,0.577,-0.084> OK
#25 pred1 real1 <-0.381,0.577,-0.084> OK
#24 pred0 real0 <0.452,0.226,-3.334> OK
#23 pred0 real0 <0.424,0.226,-3.334> OK
#22 pred0 real0 <0.452,0.226,-3.334> OK
#21 pred0 real0 <0.396,0.226,-3.334> OK
#20 pred0 real0 <0.452,0.255,-3.063> OK
#19 pred0 real0 <0.313,0.284,-2.792> OK
#18 pred0 real0 <0.396,0.343,-2.250> OK
#17 pred0 real0 <0.480,0.226,-3.334> OK
#16 pred0 real0 <0.480,0.255,-3.063> OK
#15 pred0 real0 <0.480,0.255,-3.063> OK
#14 pred0 real0 <0.424,0.226,-3.334> OK
#13 pred0 real0 <0.480,0.226,-3.334> OK
#12 pred0 real0 <0.452,0.196,-3.604> OK
#11 pred0 real0 <0.480,0.226,-3.334> OK
#10 pred0 real0 <0.507,0.226,-3.334> OK
#9 pred0 real0 <0.424,0.226,-3.334> OK
#8 pred0 real0 <0.452,0.226,-3.334> OK
#7 pred0 real0 <0.424,0.196,-3.604> OK
#6 pred0 real0 <0.424,0.284,-2.792> OK
```

```
#5 pred0 real0 <0.396,0.226,-3.334> OK
#4 pred0 real0 <0.396,0.226,-3.334> OK
#3 pred0 real0 <0.452,0.226,-3.334> OK
#2 pred0 real0 <0.424,0.226,-3.334> OK
#1 pred0 real0 <0.396,0.284,-2.792> OK
#0 pred0 real0 <0.396,0.226,-3.334> OK
Value: 0.946667
```

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 500
// 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(%02d) 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;
}
```

learning:

```
Learning...
Will stop when curEpoch >= 500 or bestVal >= -0.010000
Will save the best NeuraNet in ./bestnn.txt at each improvement
Improvement at epoch 00000: -1.971349(404) (in 00:00:00:00s)
Improvement at epoch 00004: -1.877367(237) (in 00:00:00:01s)
Improvement at epoch 00006: -1.817499(180) (in 00:00:00:01s)
Improvement at epoch 00007: -1.812141(377) (in 00:00:00:02s)
Improvement at epoch 00008: -1.800631(140) (in 00:00:00:02s)
Improvement at epoch 00010: -1.800577(229) (in 00:00:00:03s)
Improvement at epoch 00016: -1.761132(245) (in 00:00:00:06s)
Improvement at epoch 00017: -1.759505(097) (in 00:00:00:07s)
Improvement at epoch 00018: -1.759182(274) (in 00:00:00:08s)
Improvement at epoch 00019: -1.758860(493) (in 00:00:00:09s)
Improvement at epoch 00020: -1.756860(111) (in 00:00:00:10s)
Improvement at epoch 00026: -1.756834(335) (in 00:00:00:13s)
Improvement at epoch 00027: -1.756826(420) (in 00:00:00:14s)
Improvement at epoch 00078: -1.756822(255) (in 00:00:00:39s)
Improvement at epoch 00082: -1.756687(022) (in 00:00:00:41s)
Improvement at epoch 00083: -1.756120(338) (in 00:00:00:42s)
Improvement at epoch 00084: -1.755827(205) (in 00:00:00:43s)
Improvement at epoch 00085: -1.755067(237) (in 00:00:00:44s)
Improvement at epoch 00086: -1.753497(142) (in 00:00:00:45s)
Improvement at epoch 00087: -1.752741(060) (in 00:00:00:47s)
Improvement at epoch 00088: -1.752453(117) (in 00:00:00:48s)
Improvement at epoch 00089: -1.752113(348) (in 00:00:00:50s)
Improvement at epoch 00090: -1.743662(462) (in 00:00:00:51s)
Improvement at epoch 00091: -1.735771(142) (in 00:00:00:53s)
Improvement at epoch 00092: -1.734802(109) (in 00:00:00:55s)
Improvement at epoch 00093: -1.731742(308) (in 00:00:00:57s)
Improvement at epoch 00094: -1.731346(387) (in 00:00:00:58s)
Improvement at epoch 00095: -1.729803(364) (in 00:00:01:00s)
Improvement at epoch 00096: -1.729184(259) (in 00:00:01:03s)
Improvement at epoch 00097: -1.728240(267) (in 00:00:01:05s)
Improvement at epoch 00098: -1.727536(269) (in 00:00:01:07s)
Improvement at epoch 00099: -1.723748(066) (in 00:00:01:10s)
Improvement at epoch 00100: -1.722144(229) (in 00:00:01:12s)
Improvement at epoch 00101: -1.719900(131) (in 00:00:01:15s)
Improvement at epoch 00102: -1.717726(311) (in 00:00:01:17s)
Improvement at epoch 00103: -1.716916(317) (in 00:00:01:20s)
Improvement at epoch 00104: -1.709054(170) (in 00:00:01:23s)
Improvement at epoch 00105: -1.708586(238) (in 00:00:01:26s)
Improvement at epoch 00113: -1.707733(270) (in 00:00:01:39s)
Improvement at epoch 00115: -1.706367(400) (in 00:00:01:45s)
Improvement at epoch 00117: -1.705880(237) (in 00:00:01:50s)
Improvement at epoch 00125: -1.705022(071) (in 00:00:02:04s)
Improvement at epoch 00126: -1.704847(045) (in 00:00:02:07s)
Improvement at epoch 00153: -1.704478(464) (in 00:00:02:49s)
Improvement at epoch 00154: -1.704448(103) (in 00:00:02:51s)
Improvement at epoch 00155: -1.704295(199) (in 00:00:02:54s)
Improvement at epoch 00156: -1.704084(269) (in 00:00:02:56s)
Improvement at epoch 00157: -1.700553(444) (in 00:00:02:59s)
Improvement at epoch 00158: -1.698707(413) (in 00:00:03:02s)
Improvement at epoch 00159: -1.698400(405) (in 00:00:03:04s)
Improvement at epoch 00160: -1.691976(075) (in 00:00:03:07s)
Improvement at epoch 00161: -1.691067(142) (in 00:00:03:10s)
Improvement at epoch 00162: -1.687528(206) (in 00:00:03:13s)
Improvement at epoch 00163: -1.687509(084) (in 00:00:03:16s)
Improvement at epoch 00164: -1.686139(432) (in 00:00:03:19s)
Improvement at epoch 00165: -1.684801(028) (in 00:00:03:21s)
Improvement at epoch 00166: -1.683678(362) (in 00:00:03:25s)
Improvement at epoch 00167: -1.681378(356) (in 00:00:03:28s)
Improvement at epoch 00168: -1.680992(257) (in 00:00:03:31s)
Improvement at epoch 00169: -1.676389(091) (in 00:00:03:34s)
Improvement at epoch 00170: -1.668861(340) (in 00:00:03:38s)
Improvement at epoch 00179: -1.668811(165) (in 00:00:03:55s)
Improvement at epoch 00181: -1.667126(359) (in 00:00:04:01s)
Improvement at epoch 00182: -1.665800(272) (in 00:00:04:05s)
Improvement at epoch 00183: -1.665520(032) (in 00:00:04:08s)
Improvement at epoch 00185: -1.664837(410) (in 00:00:04:15s)
Improvement at epoch 00186: -1.664262(203) (in 00:00:04:19s)
Improvement at epoch 00187: -1.663884(305) (in 00:00:04:22s)
Improvement at epoch 00189: -1.663746(359) (in 00:00:04:30s)
Improvement at epoch 00190: -1.663742(064) (in 00:00:04:33s)
Improvement at epoch 00198: -1.663497(268) (in 00:00:04:49s)
Improvement at epoch 00200: -1.663305(384) (in 00:00:04:56s)
Improvement at epoch 00202: -1.662940(092) (in 00:00:05:04s)
Improvement at epoch 00203: -1.662912(491) (in 00:00:05:08s)
Improvement at epoch 00204: -1.662710(296) (in 00:00:05:11s)
Improvement at epoch 00205: -1.662634(192) (in 00:00:05:15s)
Improvement at epoch 00206: -1.662469(447) (in 00:00:05:19s)
Improvement at epoch 00207: -1.661258(314) (in 00:00:05:23s)
Improvement at epoch 00208: -1.661028(496) (in 00:00:05:27s)
Improvement at epoch 00209: -1.660934(039) (in 00:00:05:31s)
Improvement at epoch 00210: -1.660422(194) (in 00:00:05:35s)
Improvement at epoch 00212: -1.659953(022) (in 00:00:05:43s)
Improvement at epoch 00213: -1.659894(339) (in 00:00:05:47s)
Improvement at epoch 00214: -1.658742(440) (in 00:00:05:52s)
Improvement at epoch 00215: -1.658733(088) (in 00:00:05:56s)
Improvement at epoch 00216: -1.658281(191) (in 00:00:06:00s)
Improvement at epoch 00218: -1.652524(119) (in 00:00:06:09s)
Improvement at epoch 00219: -1.651065(226) (in 00:00:06:13s)
Improvement at epoch 00220: -1.649726(227) (in 00:00:06:17s)
Improvement at epoch 00221: -1.643909(115) (in 00:00:06:21s)
Improvement at epoch 00222: -1.639920(132) (in 00:00:06:26s)
Improvement at epoch 00223: -1.636265(177) (in 00:00:06:30s)
Improvement at epoch 00224: -1.618779(109) (in 00:00:06:35s)
Improvement at epoch 00225: -1.616821(405) (in 00:00:06:39s)
Improvement at epoch 00226: -1.600839(325) (in 00:00:06:43s)
Improvement at epoch 00227: -1.600819(257) (in 00:00:06:48s)
Improvement at epoch 00228: -1.600041(028) (in 00:00:06:52s)
Improvement at epoch 00230: -1.599378(271) (in 00:00:07:01s)
Improvement at epoch 00231: -1.597452(439) (in 00:00:07:06s)
Improvement at epoch 00232: -1.597425(342) (in 00:00:07:11s)
Improvement at epoch 00233: -1.594755(096) (in 00:00:07:16s)
Improvement at epoch 00234: -1.594710(216) (in 00:00:07:21s)
Improvement at epoch 00235: -1.594682(241) (in 00:00:07:25s)
Improvement at epoch 00236: -1.594534(278) (in 00:00:07:30s)
Improvement at epoch 00237: -1.593884(363) (in 00:00:07:35s)
Improvement at epoch 00238: -1.593222(166) (in 00:00:07:40s)
Improvement at epoch 00239: -1.591308(134) (in 00:00:07:45s)
Improvement at epoch 00240: -1.590485(269) (in 00:00:07:50s)
Improvement at epoch 00241: -1.590482(488) (in 00:00:07:55s)
Improvement at epoch 00242: -1.588748(349) (in 00:00:08:00s)
Improvement at epoch 00243: -1.587817(063) (in 00:00:08:05s)
Improvement at epoch 00244: -1.587770(226) (in 00:00:08:10s)
Improvement at epoch 00245: -1.587415(462) (in 00:00:08:15s)
Improvement at epoch 00246: -1.587022(077) (in 00:00:08:20s)
Improvement at epoch 00247: -1.586930(104) (in 00:00:08:25s)
Improvement at epoch 00248: -1.586923(064) (in 00:00:08:30s)
Improvement at epoch 00249: -1.586626(191) (in 00:00:08:35s)
Improvement at epoch 00250: -1.586538(102) (in 00:00:08:41s)
Improvement at epoch 00251: -1.586526(060) (in 00:00:08:46s)
Improvement at epoch 00252: -1.586397(030) (in 00:00:08:51s)
Improvement at epoch 00254: -1.586397(453) (in 00:00:09:01s)
Improvement at epoch 00265: -1.586179(273) (in 00:00:09:38s)
Improvement at epoch 00266: -1.585962(315) (in 00:00:09:43s)
Improvement at epoch 00267: -1.585901(073) (in 00:00:09:48s)
Improvement at epoch 00276: -1.585826(219) (in 00:00:10:15s)
Improvement at epoch 00277: -1.585126(081) (in 00:00:10:20s)
Improvement at epoch 00278: -1.585076(023) (in 00:00:10:25s)
Improvement at epoch 00279: -1.584669(171) (in 00:00:10:30s)
Improvement at epoch 00280: -1.584627(085) (in 00:00:10:35s)
Improvement at epoch 00281: -1.584079(309) (in 00:00:10:40s)
Improvement at epoch 00282: -1.583808(417) (in 00:00:10:45s)
Improvement at epoch 00283: -1.583793(044) (in 00:00:10:50s)
Improvement at epoch 00284: -1.583514(417) (in 00:00:10:55s)
Improvement at epoch 00285: -1.583501(475) (in 00:00:11:00s)
Improvement at epoch 00286: -1.583496(293) (in 00:00:11:05s)
Improvement at epoch 00316: -1.583364(309) (in 00:00:12:50s)
Improvement at epoch 00317: -1.581249(047) (in 00:00:12:55s)
Improvement at epoch 00318: -1.562425(100) (in 00:00:13:00s)
Improvement at epoch 00319: -1.560161(319) (in 00:00:13:05s)
Improvement at epoch 00320: -1.558235(238) (in 00:00:13:10s)
Improvement at epoch 00322: -1.557950(026) (in 00:00:13:20s)
Improvement at epoch 00323: -1.556871(211) (in 00:00:13:24s)
Improvement at epoch 00325: -1.556182(174) (in 00:00:13:34s)
Improvement at epoch 00326: -1.554334(406) (in 00:00:13:39s)
Improvement at epoch 00327: -1.551958(043) (in 00:00:13:44s)
Improvement at epoch 00328: -1.551453(138) (in 00:00:13:49s)
Improvement at epoch 00329: -1.551144(385) (in 00:00:13:54s)
Improvement at epoch 00330: -1.551142(249) (in 00:00:13:59s)
Improvement at epoch 00331: -1.550991(134) (in 00:00:14:04s)
Improvement at epoch 00332: -1.550933(218) (in 00:00:14:09s)
Improvement at epoch 00333: -1.545999(126) (in 00:00:14:14s)
Improvement at epoch 00335: -1.543652(057) (in 00:00:14:24s)
Improvement at epoch 00346: -1.542042(224) (in 00:00:14:59s)
Improvement at epoch 00348: -1.539996(248) (in 00:00:15:09s)
Improvement at epoch 00399: -1.539788(044) (in 00:00:17:32s)
Improvement at epoch 00400: -1.536717(290) (in 00:00:17:37s)
Improvement at epoch 00402: -1.536052(367) (in 00:00:17:47s)
Improvement at epoch 00403: -1.535166(270) (in 00:00:17:52s)
Improvement at epoch 00404: -1.534532(215) (in 00:00:17:57s)
Improvement at epoch 00405: -1.533918(323) (in 00:00:18:02s)
Improvement at epoch 00406: -1.533287(348) (in 00:00:18:07s)
Improvement at epoch 00407: -1.532331(069) (in 00:00:18:12s)
Improvement at epoch 00408: -1.531760(492) (in 00:00:18:17s)
Improvement at epoch 00409: -1.531728(183) (in 00:00:18:23s)
Improvement at epoch 00410: -1.530546(083) (in 00:00:18:28s)
Improvement at epoch 00411: -1.530483(060) (in 00:00:18:33s)
Improvement at epoch 00412: -1.530084(258) (in 00:00:18:38s)
Improvement at epoch 00422: -1.529846(418) (in 00:00:19:13s)
Improvement at epoch 00423: -1.529732(317) (in 00:00:19:18s)
Improvement at epoch 00424: -1.528475(101) (in 00:00:19:24s)
Improvement at epoch 00426: -1.527172(030) (in 00:00:19:36s)
Improvement at epoch 00427: -1.526661(073) (in 00:00:19:42s)
Improvement at epoch 00428: -1.526264(343) (in 00:00:19:49s)

Learning complete (in 0:0:25:22s)
```

validation:

```
age_err count cumul_perc
 0   303 0.257434
 1   480 0.665251
 2   198 0.833475
 3    96 0.915038
 4    39 0.948173
 5    25 0.969414
 6    14 0.981308
 7    13 0.992353
 8     6 0.997451
 9     3 1.000000
10     0 1.000000
11     0 1.000000
12     0 1.000000
13     0 1.000000
14     0 1.000000
15     0 1.000000
16     0 1.000000
17     0 1.000000
18     0 1.000000
19     0 1.000000
20     0 1.000000
21     0 1.000000
22     0 1.000000
23     0 1.000000
24     0 1.000000
25     0 1.000000
26     0 1.000000
27     0 1.000000
28     0 1.000000
Value: -1.466668
```