NeuraNet

P. Baillehache

August 15, 2018

Contents

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 = 7;
  int nbLink = 7;
  NeuraNet* nn = NeuraNetCreate(nbIn, nbOut, nbHid, nbBase, nbLink);
  GenAlg* ga = GenAlgCreate(GENALG_NBENTITIES, GENALG_NBELITES,
    NNGetGAAdnFloatLength(nn), NNGetGAAdnIntLength(nn));
  NNSetGABoundsBases(nn, ga);
  NNSetGABoundsLinks(nn, ga);
  // Must be declared as a GenAlg applied to a NeuraNet or links will
  // get corrupted
  GASetTypeNeuraNet(ga, nbIn, nbHid, nbOut);
  GAInit(ga);
  float best = -1000000.0;
  float ev = 0.0;
```

```
  do {
    for (int iEnt = GAGetNbAdns(ga); iEnt--;) {
      if (GAAdnIsNew(GAAdn(ga, iEnt))) {
        NNSetBases(nn, GAAdnAdnF(GAAdn(ga, iEnt)));
        NNSetLinks(nn, GAAdnAdnI(GAAdn(ga, iEnt)));
        float value = evaluate(nn);
        GASetAdnValue(ga, GAAdn(ga, iEnt), value);
      }
    }
    GAStep(ga);
    NNSetBases(nn, GABestAdnF(ga));
    NNSetLinks(nn, GABestAdnI(ga));
    ev = evaluate(nn);
    if (ev > best + PBMATH_EPSILON) {
      best = ev;
      printf("%lu %f\n", GAGetCurEpoch(ga), best);
      fflush(stdout);
    }
  } while (GAGetCurEpoch(ga) < 30000 && fabs(ev) > 0.001);
  //} while (GAGetCurEpoch(ga) < 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 -0.674822
3 -0.565716
4 -0.466534
5 -0.415402
6 -0.377561
9 -0.354827
11 -0.316937
14 -0.316736
16 -0.243900
18 -0.233233
19 -0.212076
24 -0.210802
25 -0.193109
29 -0.185219
98 -0.167244
335 -0.166981
341 -0.165547
394 -0.163847
402 -0.163752
499 -0.163570
501 -0.146793
524 -0.144805
528 -0.139099
542 -0.111845
1133 -0.077426
1138 -0.058636
1274 -0.051739
1783 -0.043255
2487 -0.043237
3468 -0.043177
4061 -0.040447
5096 -0.035921
5421 -0.030604
5554 -0.027241
5722 -0.026392
5726 -0.023733
14808 -0.023673
16303 -0.018844
best after 30000 epochs: -0.018844
nbInput: 3
nbOutput: 3
nbHidden: 3
nbMaxBases: 7
nbMaxLinks: 7
bases: <-0.750,0.804,-0.831,0.834,0.241,-0.640,0.242,0.177,-0.838,0.962,0.042,-0.712,0.285,0.148,-0.060,0.496,-0.657
```

```
links: <2,0,5,5,0,6,6,0,6,0,1,3,3,1,5,4,1,6,5,1,7>
hidden values: <-1.000,0.000,1.000>
UnitTestNeuraNetGA OK
UnitTestNeuraNet OK
UnitTestAll OK
```

neuranet.txt:


```
{
  "_nbInputVal":"10",
  "_nbOutputVal":"20",
  "_nbMaxHidVal":"30",
  "_nbMaxBases":"4",
  "_nbMaxLinks":"5",
  "_bases":{
    "_dim":"12",
    "_val":["0.000000","0.010000","0.020000","0.030000","0.040000","0.050000","0.060000","0.070000","0.080000","0.09
  },
  "_links":{
    "_dim":"15",
    "_val":["1","1","12","2","2","35","2","15","20","3","15","20","-1","0","0"]
  }
}
```


bestnn.txt:


```
{
  "_nbInputVal":"3",
  "_nbOutputVal":"3",
  "_nbMaxHidVal":"3",
  "_nbMaxBases":"7",
  "_nbMaxLinks":"7",
  "_bases":{
    "_dim":"21",
    "_val":["-0.749753","0.803968","-0.831047","0.833707","0.241039","-0.640398","0.242330","0.176948","-0.838218","
  },
  "_links":{
    "_dim":"21",
    "_val":["2","0","5","5","0","6","6","0","6","0","1","3","3","1","5","4","1","6","5","1","7"]
  }
}
```


7 Validation

7.1 Iris data set

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

main.c:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
```

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

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

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

// Categories of data sets

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

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

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

typedef struct DataSet {
  // Category of the data set
```

```
  DataSetCat _cat;
  // Number of sample
  int _nbSample;
  // Samples
  Iris* _samples;
} DataSet;

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

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

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

```
    that->_nbSample = 75;
    that->_samples =
      PBErrMalloc(NeuraNetErr, sizeof(Iris) * that->_nbSample);
    for (int iCat = 0; iCat < 3; ++iCat) {
      for (int iSample = 0; iSample < 25; ++iSample) {
        ret = fscanf(f, "%s\n", buffer);
        if (ret == EOF) {
          printf("Couldn't read the dataset\n");
          fclose(f);
          return false;
        }
      }
      for (int iSample = 0; iSample < 25; ++iSample) {
        ret = fscanf(f, "%f,%f,%f,%f,%s",
          that->_samples[25 * iCat + iSample]._props,
          that->_samples[25 * iCat + iSample]._props + 1,
          that->_samples[25 * iCat + iSample]._props + 2,
          that->_samples[25 * iCat + iSample]._props + 3,
          buffer);
        if (ret == EOF) {
          printf("Couldn't read the dataset\n");
          fclose(f);
          return false;
        }
        that->_samples[25 * iCat + iSample]._cat = (IrisCat)iCat;
      }
    }
  } else if (cat == dataall) {
    that->_nbSample = 150;
    that->_samples =
      PBErrMalloc(NeuraNetErr, sizeof(Iris) * that->_nbSample);
    for (int iCat = 0; iCat < 3; ++iCat) {
      for (int iSample = 0; iSample < 50; ++iSample) {
        ret = fscanf(f, "%f,%f,%f,%f,%s",
          that->_samples[50 * iCat + iSample]._props,
          that->_samples[50 * iCat + iSample]._props + 1,
          that->_samples[50 * iCat + iSample]._props + 2,
          that->_samples[50 * iCat + iSample]._props + 3,
          buffer);
        if (ret == EOF) {
          printf("Couldn't read the dataset\n");
          fclose(f);
          return false;
        }
        that->_samples[50 * iCat + iSample]._cat = (IrisCat)iCat;
      }
    }
  } else {
    printf("Invalid dataset\n");
    fclose(f);
    return false;
  }
  fclose(f);

  // Return success code
  return true;
}

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

```
  free((*that)->_samples);
  free(*that);
  *that = NULL;
}

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

  // Evaluate

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

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

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

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

  // Return the NeuraNet
  return nn;
}
```

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

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

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

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

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

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

```
  clock_t clockEnd = clock();
  double timeUsed =
    ((double)(clockEnd - clockStart)) / (CLOCKS_PER_SEC * 0.001) ;
  // If the clock has been reset meanwhile
  if (timeUsed < 0.0)
    timeUsed = 0.0;
  printf("Prediction: %s (in %fms)\n", irisCatNames[pred], timeUsed);

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

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

```
  return 0;
}
```

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 5000
// Save NeuraNet in compact format
#define COMPACT true

// Categories of data sets

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

// Structure for the data set

typedef struct Abalone {
  float _props[10];
  float _age;
} Abalone;

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

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

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

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

```
        &age);
      (void)dummy;
      if (ret == EOF) {
        printf("Couldn't read the dataset\n");
        fclose(f);
        return false;
      }
    }
    that->_nbSample = 1177;
    that->_samples =
      PBErrMalloc(NeuraNetErr, sizeof(Abalone) * that->_nbSample);
    for (int iSample = 0; iSample < that->_nbSample; ++iSample) {
      ret = fscanf(f, "%c %f %f %f %f %f %f %f %d\n",
        &sex,
        that->_samples[iSample]._props + 3,
        that->_samples[iSample]._props + 4,
        that->_samples[iSample]._props + 5,
        that->_samples[iSample]._props + 6,
        that->_samples[iSample]._props + 7,
        that->_samples[iSample]._props + 8,
        that->_samples[iSample]._props + 9,
        &age);
      if (ret == EOF) {
        printf("Couldn't read the dataset\n");
        fclose(f);
        return false;
      }
      that->_samples[iSample]._age = (float)age;
      if (sex == 'M') {
        that->_samples[iSample]._props[0] = 1.0;
        that->_samples[iSample]._props[1] = -1.0;
        that->_samples[iSample]._props[2] = -1.0;
      } else if (sex == 'F') {
        that->_samples[iSample]._props[0] = -1.0;
        that->_samples[iSample]._props[1] = 1.0;
        that->_samples[iSample]._props[2] = -1.0;
      } else if (sex == 'I') {
        that->_samples[iSample]._props[0] = -1.0;
        that->_samples[iSample]._props[1] = -1.0;
        that->_samples[iSample]._props[2] = 1.0;
      }
    }
  } else if (cat == dataall) {
    that->_nbSample = 4177;
    that->_samples =
      PBErrMalloc(NeuraNetErr, sizeof(Abalone) * that->_nbSample);
    for (int iSample = 0; iSample < that->_nbSample; ++iSample) {
      ret = fscanf(f, "%c %f %f %f %f %f %f %f %d\n",
        &sex,
        that->_samples[iSample]._props + 3,
        that->_samples[iSample]._props + 4,
        that->_samples[iSample]._props + 5,
        that->_samples[iSample]._props + 6,
        that->_samples[iSample]._props + 7,
        that->_samples[iSample]._props + 8,
        that->_samples[iSample]._props + 9,
        &age);
      if (ret == EOF) {
        printf("Couldn't read the dataset\n");
        fclose(f);
        return false;
      }
```

```
      that->_samples[iSample]._age = (float)age;
      if (sex == 'M') {
        that->_samples[iSample]._props[0] = 1.0;
        that->_samples[iSample]._props[1] = -1.0;
        that->_samples[iSample]._props[2] = -1.0;
      } else if (sex == 'F') {
        that->_samples[iSample]._props[0] = -1.0;
        that->_samples[iSample]._props[1] = 1.0;
        that->_samples[iSample]._props[2] = -1.0;
      } else if (sex == 'I') {
        that->_samples[iSample]._props[0] = -1.0;
        that->_samples[iSample]._props[1] = -1.0;
        that->_samples[iSample]._props[2] = 1.0;
      }
    }
  } else {
    printf("Invalid dataset\n");
    fclose(f);
    return false;
  }
  fclose(f);

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

  // Return success code
  return true;
}

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

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

  // Evaluate
```

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

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

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

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

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

  // Return the NeuraNet
  return nn;
}

// Learn based on the SataSetCat 'cat'
void Learn(DataSetCat cat) {
  // Init the random generator
  srandom(time(NULL));
  // Declare variables to measure time
  struct timespec start, stop;
  // Start measuring time
  clock_gettime(CLOCK_REALTIME, &start);
  // Load the DataSet
  DataSet* dataset = PBErrMalloc(NeuraNetErr, sizeof(DataSet));
  bool ret = DataSetLoad(dataset, cat);
```

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

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

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

// Check the NeuraNet 'that' on the DataSetCat 'cat'
```

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

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

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

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

learning:

```
Learning...
Will stop when curEpoch >= 5000 or bestVal >= -0.010000
Will save the best NeuraNet in ./bestnn.txt at each improvement
Improvement at epoch 00000: -2.360953(365) (in 00:00:00:00s)
Improvement at epoch 00001: -2.165735(126) (in 00:00:00:00s)
Improvement at epoch 00002: -1.997369(121) (in 00:00:00:01s)
Improvement at epoch 00004: -1.767354(154) (in 00:00:00:01s)
Improvement at epoch 00056: -1.764500(127) (in 00:00:00:16s)
Improvement at epoch 00081: -1.748635(311) (in 00:00:00:26s)
Improvement at epoch 00083: -1.744395(077) (in 00:00:00:29s)
Improvement at epoch 00084: -1.743536(136) (in 00:00:00:30s)
Improvement at epoch 00085: -1.741244(101) (in 00:00:00:31s)
Improvement at epoch 00086: -1.740951(126) (in 00:00:00:32s)
Improvement at epoch 00087: -1.739302(419) (in 00:00:00:33s)
Improvement at epoch 00097: -1.739290(191) (in 00:00:00:39s)
Improvement at epoch 00099: -1.738023(463) (in 00:00:00:41s)
Improvement at epoch 00100: -1.734099(181) (in 00:00:00:42s)
Improvement at epoch 00123: -1.726702(045) (in 00:00:00:55s)
Improvement at epoch 00183: -1.723554(371) (in 00:00:01:22s)
Improvement at epoch 00190: -1.723239(302) (in 00:00:01:27s)
Improvement at epoch 00270: -1.718333(185) (in 00:00:02:14s)
Improvement at epoch 00284: -1.716988(111) (in 00:00:02:24s)
Improvement at epoch 00332: -1.716562(193) (in 00:00:02:52s)
Improvement at epoch 00337: -1.708923(218) (in 00:00:02:55s)
Improvement at epoch 00339: -1.706948(173) (in 00:00:02:58s)
```

```
Improvement at epoch 00340: -1.699306(273) (in 00:00:02:59s)
Improvement at epoch 00341: -1.688199(085) (in 00:00:03:01s)
Improvement at epoch 00342: -1.686926(026) (in 00:00:03:03s)
Improvement at epoch 00343: -1.685817(414) (in 00:00:03:05s)
Improvement at epoch 00344: -1.679162(189) (in 00:00:03:06s)
Improvement at epoch 00345: -1.677131(278) (in 00:00:03:08s)
Improvement at epoch 00346: -1.674080(173) (in 00:00:03:10s)
Improvement at epoch 00347: -1.672594(040) (in 00:00:03:12s)
Improvement at epoch 00377: -1.671741(350) (in 00:00:03:36s)
Improvement at epoch 00388: -1.660580(144) (in 00:00:03:44s)
Improvement at epoch 00389: -1.650942(171) (in 00:00:03:45s)
Improvement at epoch 00390: -1.644811(120) (in 00:00:03:47s)
Improvement at epoch 00396: -1.640958(098) (in 00:00:03:52s)
Improvement at epoch 00402: -1.638675(299) (in 00:00:03:56s)
Improvement at epoch 00404: -1.627440(227) (in 00:00:03:59s)
Improvement at epoch 00425: -1.625763(398) (in 00:00:04:18s)
Improvement at epoch 00431: -1.623107(097) (in 00:00:04:23s)
Improvement at epoch 00432: -1.620175(269) (in 00:00:04:25s)
Improvement at epoch 00460: -1.619694(046) (in 00:00:04:49s)
Improvement at epoch 00467: -1.615901(379) (in 00:00:04:56s)
Improvement at epoch 00475: -1.614937(494) (in 00:00:05:02s)
Improvement at epoch 00476: -1.606234(439) (in 00:00:05:03s)
Improvement at epoch 00483: -1.604426(118) (in 00:00:05:10s)
Improvement at epoch 00501: -1.602751(393) (in 00:00:05:24s)
Improvement at epoch 00502: -1.602674(056) (in 00:00:05:26s)
Improvement at epoch 00504: -1.601908(374) (in 00:00:05:29s)
Improvement at epoch 00505: -1.601626(150) (in 00:00:05:31s)
Improvement at epoch 00533: -1.601312(287) (in 00:00:05:54s)
Improvement at epoch 00685: -1.601289(023) (in 00:00:08:03s)
Improvement at epoch 00687: -1.601208(028) (in 00:00:08:07s)
Improvement at epoch 00753: -1.601174(120) (in 00:00:09:04s)
Improvement at epoch 00760: -1.600145(229) (in 00:00:09:11s)
Improvement at epoch 00762: -1.599829(203) (in 00:00:09:15s)
Improvement at epoch 00763: -1.599769(466) (in 00:00:09:17s)
Improvement at epoch 00765: -1.599381(266) (in 00:00:09:20s)
Improvement at epoch 00912: -1.599367(217) (in 00:00:11:40s)
Improvement at epoch 00925: -1.599273(257) (in 00:00:11:53s)
Improvement at epoch 01138: -1.599229(383) (in 00:00:15:32s)
Improvement at epoch 01139: -1.599179(096) (in 00:00:15:35s)
Improvement at epoch 01321: -1.599020(259) (in 00:00:19:10s)
Improvement at epoch 01903: -1.598551(214) (in 00:00:30:55s)
Improvement at epoch 01931: -1.597927(264) (in 00:00:31:30s)
Improvement at epoch 02082: -1.597865(043) (in 00:00:34:13s)
Improvement at epoch 02103: -1.597598(067) (in 00:00:34:44s)
Improvement at epoch 02153: -1.597088(201) (in 00:00:35:56s)
Improvement at epoch 02155: -1.596866(196) (in 00:00:36:03s)
Improvement at epoch 02258: -1.596793(434) (in 00:00:38:52s)
Improvement at epoch 02374: -1.596650(401) (in 00:00:41:21s)
Improvement at epoch 02510: -1.596607(249) (in 00:00:44:20s)
Improvement at epoch 02838: -1.596448(394) (in 00:00:52:00s)
Improvement at epoch 03236: -1.596236(331) (in 00:01:01:14s)
Improvement at epoch 03237: -1.595707(183) (in 00:01:01:18s)
Improvement at epoch 03255: -1.595627(250) (in 00:01:01:48s)
Improvement at epoch 03311: -1.595546(107) (in 00:01:03:30s)
Improvement at epoch 03313: -1.595409(451) (in 00:01:03:38s)
Improvement at epoch 03314: -1.595387(402) (in 00:01:03:42s)
Improvement at epoch 03340: -1.594862(083) (in 00:01:04:37s)
Improvement at epoch 03353: -1.594612(144) (in 00:01:05:01s)
Improvement at epoch 03500: -1.594578(180) (in 00:01:09:13s)
Improvement at epoch 03612: -1.593322(381) (in 00:01:12:19s)
Improvement at epoch 03714: -1.593137(301) (in 00:01:14:50s)
Improvement at epoch 03715: -1.593000(247) (in 00:01:14:53s)
```

```
Improvement at epoch 03785: -1.592284(211) (in 00:01:16:18s)
Improvement at epoch 03786: -1.592068(417) (in 00:01:16:21s)
Improvement at epoch 04469: -1.591921(385) (in 00:01:30:56s)

Learning complete (in 0:1:42:23s)
```

validation:

```
age_err count cumul_perc
 0   291 0.247239
 1   462 0.639762
 2   228 0.833475
 3    94 0.913339
 4    35 0.943076
 5    24 0.963466
 6    18 0.978760
 7    13 0.989805
 8     5 0.994053
 9     4 0.997451
10     3 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: -1.516142
```

7.3 Arrhythmia data set

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

main.c:

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

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

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

// Categories of data sets

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

const char* catNames[NB_OUTPUT] = {
  "Normal",
  "Ischemic changes (Coronary Artery Disease)",
  "Old Anterior Myocardial Infarction",
  "Old Inferior Myocardial Infarction",
  "Sinus tachycardy",
  "Sinus bradycardy",
  "Ventricular Premature Contraction (PVC)",
  "Supraventricular Premature Contraction",
  "Left bundle branch block",
  "Right bundle branch block",
  "1. degree AtrioVentricular block",
  "2. degree AV block",
  "3. degree AV block",
  "Left ventricule hypertrophy",
  "Atrial Fibrillation or Flutter",
  "Others"
};



typedef struct Arrhytmia {
```

```
  float _props[NB_INPUT];
  int _cat;
} Arrhytmia;

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

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

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

  // Load the data according to 'cat'
  FILE* f = fopen("./arrhythmia.data", "r");
  if (f == NULL) {
    printf("Couldn't open the data set file\n");
    return false;
  }
  int ret = 0;
  if (cat == datalearn) {
    that->_nbSample = 300;
    that->_samples =
      PBErrMalloc(NeuraNetErr, sizeof(Arrhytmia) * that->_nbSample);
    for (int iSample = 0; iSample < that->_nbSample; ++iSample) {
      for (int iProp = 0; iProp < NB_INPUT; ++iProp) {
        ret = fscanf(f, "%f,",
          that->_samples[iSample]._props + iProp);
        if (ret == EOF) {
          printf("Couldn't read the dataset\n");
          fclose(f);
          return false;
        }
      }
      ret = fscanf(f, "%d", &(that->_samples[iSample]._cat));
      if (ret == EOF) {
        printf("Couldn't read the dataset\n");
        fclose(f);
        return false;
      }
    }
  } else if (cat == datatest) {
    char buffer[1000];
    for (int iSample = 0; iSample < 300; ++iSample) {
```

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

  // Return success code
  return true;
}

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

```
  // Free the memory
  free((*that)->_samples);
  free(*that);
  *that = NULL;
}

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

  // Evaluate

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

  }
  int nbCat = 0;
  for (int iCat = 0; iCat < NB_OUTPUT; ++iCat) {
    if (countCat[iCat] > 0) {
      ++nbCat;
      float perc = 0.0;
      if (dataset->_cat != datalearn) {
        perc = (float)(countOk[iCat]) / (float)(countCat[iCat]);
        printf("%43s (%3d): %f\n", catNames[iCat], countCat[iCat], perc);
        val += countOk[iCat];
      } else {
        perc = (float)(countOk[iCat] - countNg[iCat]) /
          (float)(countCat[iCat]);
        val += perc;
      }
    }
  }
  if (dataset->_cat != datalearn)
    val /= (float)(dataset->_nbSample);
  else
    val /= (float)nbCat;

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

```
}

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

  // Return the NeuraNet
  return nn;
}

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

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

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

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

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

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

```
    sscanf(inputs[iInp], "%f", &v);
    VecSet(input, iInp, v);
  }
  // Predict
  NNEval(that, input, output);
  int pred = VecGetIMaxVal(output);
  // End measuring time
  clock_t clockEnd = clock();
  double timeUsed =
    ((double)(clockEnd - clockStart)) / (CLOCKS_PER_SEC * 0.001) ;
  // If the clock has been reset meanwhile
  if (timeUsed < 0.0)
    timeUsed = 0.0;
  printf("Prediction: %s (in %fms)\n", catNames[pred], timeUsed);

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

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

```
      return 0;
    }
    fclose(fd);
    Predict(nn, argc - 2, argv + 2);
    NeuraNetFree(&nn);
  }
  // Return success code
  return 0;
}
```

learning:

```
Learning...
Will stop when curEpoch >= 25000 or bestVal >= 0.999000
Will save the best NeuraNet in ./bestnn.txt at each improvement
Improvement at epoch 00000: -0.644689(057) (in 00:00:00:00s)
Improvement at epoch 00001: -0.557136(142) (in 00:00:00:01s)
Improvement at epoch 00002: -0.461538(067) (in 00:00:00:01s)
Improvement at epoch 00003: -0.355556(449) (in 00:00:00:01s)
Improvement at epoch 00005: -0.333578(055) (in 00:00:00:01s)
Improvement at epoch 00016: -0.201709(108) (in 00:00:00:04s)
Improvement at epoch 00018: -0.185226(411) (in 00:00:00:05s)
Improvement at epoch 00019: -0.179731(057) (in 00:00:00:05s)
Improvement at epoch 00020: -0.161416(371) (in 00:00:00:05s)
Improvement at epoch 00021: -0.148596(150) (in 00:00:00:06s)
Improvement at epoch 00022: -0.128449(328) (in 00:00:00:06s)
Improvement at epoch 00023: -0.111355(363) (in 00:00:00:07s)
Improvement at epoch 00024: -0.102808(081) (in 00:00:00:07s)
Improvement at epoch 00026: -0.098535(091) (in 00:00:00:08s)
Improvement at epoch 00033: -0.094872(439) (in 00:00:00:10s)
Improvement at epoch 00034: -0.081441(185) (in 00:00:00:11s)
Improvement at epoch 00036: -0.065526(277) (in 00:00:00:12s)
Improvement at epoch 00038: -0.049611(434) (in 00:00:00:13s)
Improvement at epoch 00039: -0.044306(286) (in 00:00:00:13s)
Improvement at epoch 00040: -0.033695(276) (in 00:00:00:14s)
Improvement at epoch 00041: -0.028390(061) (in 00:00:00:14s)
Improvement at epoch 00049: -0.021317(373) (in 00:00:00:17s)
Improvement at epoch 00051: -0.018602(061) (in 00:00:00:18s)
Improvement at epoch 00052: -0.017697(150) (in 00:00:00:19s)
Improvement at epoch 00053: -0.013297(364) (in 00:00:00:20s)
Improvement at epoch 00054: -0.008540(126) (in 00:00:00:20s)
Improvement at epoch 00055: 0.000278(499) (in 00:00:00:21s)
Improvement at epoch 00056: 0.013438(490) (in 00:00:00:21s)
Improvement at epoch 00057: 0.026108(162) (in 00:00:00:22s)
Improvement at epoch 00058: 0.046027(117) (in 00:00:00:23s)
Improvement at epoch 00059: 0.059601(363) (in 00:00:00:23s)
Improvement at epoch 00060: 0.064191(243) (in 00:00:00:24s)
Improvement at epoch 00061: 0.071433(162) (in 00:00:00:25s)
Improvement at epoch 00062: 0.076928(022) (in 00:00:00:26s)
Improvement at epoch 00069: 0.100316(386) (in 00:00:00:29s)
Improvement at epoch 00070: 0.128210(043) (in 00:00:00:30s)
Improvement at epoch 00098: 0.129115(287) (in 00:00:00:44s)
Improvement at epoch 00099: 0.134420(379) (in 00:00:00:45s)
Improvement at epoch 00107: 0.135030(078) (in 00:00:00:53s)
Improvement at epoch 00108: 0.140525(085) (in 00:00:00:54s)
Improvement at epoch 00109: 0.200016(253) (in 00:00:00:55s)
Improvement at epoch 00110: 0.205511(020) (in 00:00:00:55s)
Improvement at epoch 00111: 0.213258(125) (in 00:00:00:56s)
Improvement at epoch 00112: 0.232765(054) (in 00:00:00:57s)
Improvement at epoch 00126: 0.233670(131) (in 00:00:01:05s)
```

```
Improvement at epoch 00137: 0.237628(160) (in 00:00:01:13s)
Improvement at epoch 00138: 0.243122(027) (in 00:00:01:14s)
Improvement at epoch 00139: 0.250448(042) (in 00:00:01:15s)
Improvement at epoch 00153: 0.254933(359) (in 00:00:01:23s)
Improvement at epoch 00154: 0.262048(041) (in 00:00:01:24s)
Improvement at epoch 00155: 0.264385(469) (in 00:00:01:25s)
Improvement at epoch 00156: 0.271500(075) (in 00:00:01:26s)
Improvement at epoch 00191: 0.276994(429) (in 00:00:01:49s)
Improvement at epoch 00193: 0.277899(069) (in 00:00:01:51s)
Improvement at epoch 00194: 0.285541(200) (in 00:00:01:52s)
Improvement at epoch 00195: 0.286446(021) (in 00:00:01:53s)
Improvement at epoch 00223: 0.287351(258) (in 00:00:02:15s)
Improvement at epoch 00232: 0.292551(363) (in 00:00:02:22s)
Improvement at epoch 00233: 0.312992(216) (in 00:00:02:23s)
Improvement at epoch 00235: 0.313897(020) (in 00:00:02:25s)
Improvement at epoch 00263: 0.322530(444) (in 00:00:02:44s)
Improvement at epoch 00264: 0.327960(373) (in 00:00:02:45s)
Improvement at epoch 00265: 0.333455(051) (in 00:00:02:46s)
Improvement at epoch 00266: 0.334360(120) (in 00:00:02:47s)
Improvement at epoch 00267: 0.335265(025) (in 00:00:02:48s)
Improvement at epoch 00268: 0.346275(477) (in 00:00:02:49s)
Improvement at epoch 00269: 0.348085(080) (in 00:00:02:50s)
Improvement at epoch 00270: 0.365179(382) (in 00:00:02:51s)
Improvement at epoch 00300: 0.367959(043) (in 00:00:03:17s)
Improvement at epoch 00302: 0.372821(032) (in 00:00:03:19s)
Improvement at epoch 00304: 0.377411(085) (in 00:00:03:21s)
Improvement at epoch 00305: 0.378316(053) (in 00:00:03:22s)
Improvement at epoch 00355: 0.381558(098) (in 00:00:03:58s)
Improvement at epoch 00356: 0.386863(412) (in 00:00:03:59s)
Improvement at epoch 00357: 0.398652(354) (in 00:00:04:00s)
Improvement at epoch 00358: 0.403957(274) (in 00:00:04:01s)
Improvement at epoch 00359: 0.414946(029) (in 00:00:04:02s)
Improvement at epoch 00367: 0.418336(281) (in 00:00:04:08s)
Improvement at epoch 00381: 0.437872(340) (in 00:00:04:18s)
Improvement at epoch 00382: 0.439682(033) (in 00:00:04:20s)
Improvement at epoch 00414: 0.440587(192) (in 00:00:04:41s)
Improvement at epoch 00502: 0.442650(158) (in 00:00:05:49s)
Improvement at epoch 00547: 0.445705(325) (in 00:00:06:23s)
Improvement at epoch 00548: 0.447515(058) (in 00:00:06:24s)
Improvement at epoch 00549: 0.449325(297) (in 00:00:06:25s)
Improvement at epoch 00550: 0.450230(046) (in 00:00:06:27s)
Improvement at epoch 00558: 0.450882(090) (in 00:00:06:33s)
Improvement at epoch 00559: 0.452945(086) (in 00:00:06:34s)
Improvement at epoch 00581: 0.456187(436) (in 00:00:06:50s)
Improvement at epoch 00583: 0.457092(154) (in 00:00:06:52s)
Improvement at epoch 00749: 0.465366(271) (in 00:00:09:02s)
Improvement at epoch 00758: 0.468986(035) (in 00:00:09:10s)
Improvement at epoch 00759: 0.469891(028) (in 00:00:09:11s)
Improvement at epoch 00760: 0.475385(250) (in 00:00:09:12s)
Improvement at epoch 00761: 0.489343(246) (in 00:00:09:13s)
Improvement at epoch 00762: 0.494837(041) (in 00:00:09:14s)
Improvement at epoch 00763: 0.500142(406) (in 00:00:09:16s)
Improvement at epoch 00837: 0.501047(275) (in 00:00:10:20s)
Improvement at epoch 00930: 0.501952(205) (in 00:00:11:35s)
Improvement at epoch 00932: 0.509594(407) (in 00:00:11:38s)
Improvement at epoch 00933: 0.510499(023) (in 00:00:11:39s)
Improvement at epoch 00941: 0.511469(272) (in 00:00:11:45s)
Improvement at epoch 00942: 0.515089(093) (in 00:00:11:46s)
Improvement at epoch 00943: 0.525173(067) (in 00:00:11:48s)
Improvement at epoch 00944: 0.526078(371) (in 00:00:11:49s)
Improvement at epoch 01029: 0.526983(396) (in 00:00:13:04s)
Improvement at epoch 01031: 0.530667(280) (in 00:00:13:07s)
```

```
Improvement at epoch 01033: 0.531572(078) (in 00:00:13:10s)
Improvement at epoch 01056: 0.532477(284) (in 00:00:13:30s)
Improvement at epoch 01225: 0.533382(158) (in 00:00:16:05s)
Improvement at epoch 01226: 0.533635(087) (in 00:00:16:06s)
Improvement at epoch 01227: 0.534540(030) (in 00:00:16:07s)
Improvement at epoch 01284: 0.539845(396) (in 00:00:16:56s)
Improvement at epoch 01389: 0.540750(028) (in 00:00:18:32s)
Improvement at epoch 01419: 0.541655(213) (in 00:00:19:01s)
Improvement at epoch 01434: 0.542898(358) (in 00:00:19:19s)
Improvement at epoch 01435: 0.543803(076) (in 00:00:19:21s)
Improvement at epoch 01592: 0.546855(235) (in 00:00:21:56s)
Improvement at epoch 01593: 0.546980(292) (in 00:00:21:57s)
Improvement at epoch 01594: 0.552285(175) (in 00:00:21:58s)
Improvement at epoch 01611: 0.555970(449) (in 00:00:22:17s)
Improvement at epoch 01612: 0.557780(041) (in 00:00:22:19s)
Improvement at epoch 01651: 0.558685(244) (in 00:00:22:54s)
Improvement at epoch 01666: 0.563274(072) (in 00:00:23:08s)
Improvement at epoch 01696: 0.563569(028) (in 00:00:23:37s)
Improvement at epoch 01763: 0.564474(344) (in 00:00:24:40s)
Improvement at epoch 01778: 0.564538(179) (in 00:00:24:54s)
Improvement at epoch 01779: 0.569063(036) (in 00:00:24:55s)
Improvement at epoch 01812: 0.569968(076) (in 00:00:25:29s)
Improvement at epoch 01821: 0.571778(105) (in 00:00:25:38s)
Improvement at epoch 01956: 0.574220(287) (in 00:00:27:36s)
Improvement at epoch 01965: 0.580620(077) (in 00:00:27:44s)
Improvement at epoch 01967: 0.593440(240) (in 00:00:27:47s)
Improvement at epoch 01968: 0.599650(072) (in 00:00:27:48s)
Improvement at epoch 02005: 0.600555(233) (in 00:00:28:24s)
Improvement at epoch 02045: 0.602365(073) (in 00:00:28:59s)
Improvement at epoch 02084: 0.603270(025) (in 00:00:29:34s)
Improvement at epoch 02120: 0.605080(269) (in 00:00:30:12s)
Improvement at epoch 02122: 0.605985(050) (in 00:00:30:15s)
Improvement at epoch 02131: 0.606890(301) (in 00:00:30:24s)
Improvement at epoch 02366: 0.607795(026) (in 00:00:35:30s)
Improvement at epoch 02397: 0.611415(454) (in 00:00:36:21s)
Improvement at epoch 02479: 0.615815(396) (in 00:00:38:34s)
Improvement at epoch 02561: 0.616720(493) (in 00:00:40:46s)
Improvement at epoch 02668: 0.617625(388) (in 00:00:43:33s)
Improvement at epoch 02701: 0.618530(485) (in 00:00:44:30s)
Improvement at epoch 03337: 0.619435(171) (in 00:00:58:33s)
Improvement at epoch 03785: 0.620340(446) (in 00:01:06:31s)
Improvement at epoch 03828: 0.627982(108) (in 00:01:07:19s)
Improvement at epoch 03829: 0.628887(105) (in 00:01:07:20s)
Improvement at epoch 03855: 0.629792(454) (in 00:01:07:48s)
Improvement at epoch 03910: 0.630697(152) (in 00:01:08:49s)
Improvement at epoch 04216: 0.635286(030) (in 00:01:14:23s)
Improvement at epoch 04217: 0.636191(092) (in 00:01:14:25s)
Improvement at epoch 04218: 0.637096(338) (in 00:01:14:26s)
Improvement at epoch 04267: 0.639876(222) (in 00:01:15:18s)
Improvement at epoch 04268: 0.642591(033) (in 00:01:15:19s)
Improvement at epoch 05435: 0.646275(197) (in 00:01:47:45s)
Improvement at epoch 05436: 0.647180(031) (in 00:01:47:48s)
Improvement at epoch 05446: 0.648085(212) (in 00:01:48:06s)
Improvement at epoch 05578: 0.651327(303) (in 00:01:51:38s)
Improvement at epoch 05579: 0.656632(020) (in 00:01:51:41s)
Improvement at epoch 06618: 0.662127(279) (in 00:02:20:47s)
Improvement at epoch 06737: 0.663032(317) (in 00:02:24:08s)
Improvement at epoch 09444: 0.663937(362) (in 00:03:27:27s)
Improvement at epoch 09446: 0.664842(240) (in 00:03:27:30s)
Improvement at epoch 09725: 0.665747(380) (in 00:03:32:50s)
```

validation:

```
                                     Normal ( 75): 0.480000
 Ischemic changes (Coronary Artery Disease) ( 16): 0.187500
         Old Anterior Myocardial Infarction (  6): 0.333333
         Old Inferior Myocardial Infarction (  1): 0.000000
                          Sinus tachycardy (  6): 0.166667
                          Sinus bradycardy (  7): 0.285714
    Ventricular Premature Contraction (PVC) (  2): 0.000000
     Supraventricular Premature Contraction (  1): 0.000000
                   Left bundle branch block (  4): 0.750000
                  Right bundle branch block ( 21): 0.619048
                Left ventricule hypertrophy (  1): 0.000000
             Atrial Fibrillation or Flutter (  2): 1.000000
                                     Others ( 10): 0.100000
Value: 0.414474
```