NeuraNet

P. Baillehache

July 22, 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 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:

$$B(x) = [tan(1.57079 * b_0)(x + b_1) + b_2] \cap [-1.0, 1.0] \quad (1)$$

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);

// 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, const 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;
}

// 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 = 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 = 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, const 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]));
}

// ----- 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] = {
    -1.000000, -1.000000, -1.000000, -1.000000, -1.000000, -0.942763,
    -0.198322, 0.546119, 1.000000, 1.000000,
    -0.153181, -0.403978, -0.654776, -0.905573, -1.000000, -1.000000,
    -1.000000, -1.000000, -1.000000, -1.000000,
    0.586943, 0.301165, 0.015387, -0.270391, -0.556169, -0.841946,
    -1.000000, -1.000000, -1.000000, -1.000000,
    1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000,
    1.000000, 1.000000, 1.000000, 1.000000,
    0.774302, 0.903425, 1.000000, 1.000000, 1.000000, 1.000000,
    1.000000, 1.000000, 1.000000, 1.000000,
    1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000,
    0.990941, 0.769129, 0.547316, 0.325503,
    -1.000000, -1.000000, -1.000000, -1.000000, -1.000000, -1.000000,
    -1.000000, -1.000000, -1.000000, -1.000000,
    1.000000, 1.000000, 1.000000, 1.000000, 1.000000, 1.000000,
    1.000000, 0.885544, 0.721949, 0.558353,
    -1.000000, -1.000000, -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 UnitTestNeuraNetGetSet() {
  int nbIn = 10;
  int nbOut = 20;
  int nbHid = 30;
  int nbBase = 4;
  int nbLink = 5;
  NeuraNet* nn = NeuraNetCreate(nbIn, nbOut, nbHid, nbBase, nbLink);
  if (NNGetNbInput(nn) != nbIn) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNGetNbInput failed");
    PBErrCatch(NeuraNetErr);
  }
  if (NNGetNbMaxBases(nn) != nbBase) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNGetNbMaxBases failed");
    PBErrCatch(NeuraNetErr);
  }
  if (NNGetNbMaxHidden(nn) != nbHid) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNGetNbMaxHidden failed");
    PBErrCatch(NeuraNetErr);
  }
  if (NNGetNbMaxLinks(nn) != nbLink) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNGetNbMaxLinks failed");
    PBErrCatch(NeuraNetErr);
  }
  if (NNGetNbOutput(nn) != nbOut) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNGetNbOutput failed");
    PBErrCatch(NeuraNetErr);
  }
  if (NNBases(nn) != nn->_bases) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNBases failed");
    PBErrCatch(NeuraNetErr);
  }
  if (NNLinks(nn) != nn->_links) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNLinks failed");
    PBErrCatch(NeuraNetErr);
  }
  if (NNHiddenValues(nn) != nn->_hidVal) {
```

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

void UnitTestNeuraNetSaveLoad() {
  int nbIn = 10;
  int nbOut = 20;
  int nbHid = 30;
  int nbBase = 4;
  int nbLink = 5;
  NeuraNet* nn = NeuraNetCreate(nbIn, nbOut, nbHid, nbBase, nbLink);
  VecFloat* bases = VecFloatCreate(nbBase * NN_NBPARAMBASE);
  for (int i = nbBase * NN_NBPARAMBASE; i--;)
    VecSet(bases, i, 0.01 * (float)i);
  NNSetBases(nn, bases);
  VecFree(&bases);
  VecShort* links = VecShortCreate(15);
  short data[15] = {2,2,35, 1,1,12, -1,0,0, 2,15,20, 3,20,15};
  for (int i = 15; i--;)
    VecSet(links, i, data[i]);
  NNSetLinks(nn, links);
  VecFree(&links);
  FILE* fd = fopen("./neuranet.txt", "w");
  if (NNSave(nn, fd, false) == false) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNSave failed");
    PBErrCatch(NeuraNetErr);
  }
  fclose(fd);
  fd = fopen("./neuranet.txt", "r");
  NeuraNet* loaded = NeuraNetCreate(1, 1, 1, 1, 1);
  if (NNLoad(&loaded, fd) == false) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNLoad failed");
```

```
    PBErrCatch(NeuraNetErr);
  }
  if (NNGetNbInput(loaded) != nbIn ||
    NNGetNbMaxBases(loaded) != nbBase ||
    NNGetNbMaxHidden(loaded) != nbHid ||
    NNGetNbMaxLinks(loaded) != nbLink ||
    NNGetNbOutput(loaded) != nbOut) {
    NeuraNetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(NeuraNetErr->_msg, "NNLoad failed");
    PBErrCatch(NeuraNetErr);
  }
  for (int i = nbBase * NN_NBPARAMBASE; i--;)
    if (ISEQUALF(VecGet(NNBases(loaded), i), 0.01 * (float)i) == false) {
      NeuraNetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(NeuraNetErr->_msg, "NNLoad failed");
      PBErrCatch(NeuraNetErr);
    }
  short check[15] = {1,1,12,2,2,35,2,15,20,3,15,20,-1,0,0};
  for (int i = 15; i--;)
    if (VecGet(NNLinks(loaded), i) != check[i]) {
      NeuraNetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(NeuraNetErr->_msg, "NNLoad failed");
      PBErrCatch(NeuraNetErr);
    }
  fclose(fd);
  NeuraNetFree(&loaded);
  NeuraNetFree(&nn);
  printf("UnitTestNeuraNetSaveLoad OK\n");
}

void UnitTestNeuraNetEvalPrint() {
  int nbIn = 3;
  int nbOut = 3;
  int nbHid = 3;
  int nbBase = 3;
  int nbLink = 7;
  NeuraNet* nn = NeuraNetCreate(nbIn, nbOut, nbHid, nbBase, nbLink);
  // hidden[0] = tan(0.5*NN_THETA)*tan(-0.5*NN_THETA)*input[0]^2
  // hidden[1] = tan(0.5*NN_THETA)*input[1]
  // hidden[2] = 0
  // output[0] = tan(0.5*NN_THETA)*hidden[0]+tan(0.5*NN_THETA)*hidden[1]
  // output[1] = tan(0.5*NN_THETA)*hidden[1]
  // output[2] = 0
  NNBasesSet(nn, 0, 0.5);
  NNBasesSet(nn, 3, -0.5);
  NNBasesSet(nn, 8, -0.5);
  short data[21] = {0,0,3, 1,0,3, 0,1,4, 0,3,6, 0,4,6, 0,4,7, -1,0,0};
  VecShort *links = VecShortCreate(21);
  for (int i = 21; i--;)
    VecSet(links, i, data[i]);
  NNSetLinks(nn, links);
  VecFree(&links);
  VecFloat3D input = VecFloatCreateStatic3D();
  VecFloat3D output = VecFloatCreateStatic3D();
  VecFloat3D check = VecFloatCreateStatic3D();
  VecFloat3D checkhidden = VecFloatCreateStatic3D();
  NNPrintln(nn, stdout);
  for (int i = -10; i <= 10; ++i) {
    for (int j = -10; j <= 10; ++j) {
      for (int k = -10; k <= 10; ++k) {
        VecSet(&input, 0, 0.1 * (float)i);
        VecSet(&input, 1, 0.1 * (float)j);
```

```
        VecSet(&input, 2, 0.1 * (float)k);
        NNEval(nn, (VecFloat*)&input, (VecFloat*)&output);
        VecSet(&checkhidden, 0, tan(0.5 * NN_THETA) * tan(-0.5 * NN_THETA) * fsquare(VecGet(&input, 0)));
        VecSet(&checkhidden, 1, tan(0.5 * NN_THETA) * VecGet(&input, 1));
        VecSet(&check, 0,
          tan(0.5 * NN_THETA) * (VecGet(&checkhidden, 0) + VecGet(&checkhidden, 1)));
        VecSet(&check, 1, tan(0.5 * NN_THETA) * VecGet(&checkhidden, 1));
        if (VecIsEqual(&output, &check) == false ||
          VecIsEqual(NNHiddenValues(nn), &checkhidden) == false) {
          NeuraNetErr->_type = PBErrTypeUnitTestFailed;
          sprintf(NeuraNetErr->_msg, "NNEval failed");
          PBErrCatch(NeuraNetErr);
        }
      }
    }
  }
  NeuraNetFree(&nn);
  printf("UnitTestNeuraNetEvalPrint OK\n");
}

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

void UnitTestNeuraNetGA() {
  srandom(RANDOMSEED);
  //srandom(time(NULL));
  int nbIn = 3;
  int nbOut = 3;
  int nbHid = 3;
  int nbBase = 3;
  int nbLink = 7;
  NeuraNet* nn = NeuraNetCreate(nbIn, nbOut, nbHid, nbBase, nbLink);
  GenAlg* ga = GenAlgCreate(GENALG_NBENTITIES, GENALG_NBELITES,
    NNGetGAAdnFloatLength(nn), NNGetGAAdnIntLength(nn));
  NNSetGABoundsBases(nn, ga);
  NNSetGABoundsLinks(nn, ga);
  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);
    }
  } 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();
  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
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 -0.603145
3 -0.354291
15 -0.350419
16 -0.346406
20 -0.338872
23 -0.334632
25 -0.319703
29 -0.292745
34 -0.280032
55 -0.279363
67 -0.275094
69 -0.274985
127 -0.264797
133 -0.264651
170 -0.259245
194 -0.258304
196 -0.254653
275 -0.220771
357 -0.220467
450 -0.214935
477 -0.209358
478 -0.199559
491 -0.187458
492 -0.171047
536 -0.171014
547 -0.170631
549 -0.150539
554 -0.149827
559 -0.149203
623 -0.148514
691 -0.148501
697 -0.148394
768 -0.148256
1062 -0.148235
1091 -0.148115
1189 -0.148061
1266 -0.147983
2836 -0.147962
2913 -0.147951
3007 -0.147846
3071 -0.147821
3317 -0.147805
3380 -0.144487
3481 -0.128739
3511 -0.068447
3563 -0.053965
3767 -0.031360
3865 -0.028133
3966 -0.018840
4010 -0.018435
4031 -0.017098
4205 -0.009860
```

```
4438 -0.009831
4486 -0.009140
4688 -0.008967
4784 -0.007260
4916 -0.007220
5237 -0.007124
5375 -0.006518
5980 -0.006453
6127 -0.006436
6967 -0.006364
7602 -0.006288
7844 -0.006189
8046 -0.006022
8324 -0.004848
8898 -0.004312
9116 -0.004168
10073 -0.004080
11296 -0.003809
15172 -0.003552
19742 -0.003535
21913 -0.003516
22454 -0.003447
27877 -0.003150
29153 -0.002812
best after 30000 epochs: -0.002812
nbInput: 3
nbOutput: 3
nbHidden: 3
nbMaxBases: 3
nbMaxLinks: 7
bases: <0.502,0.305,-0.307,-0.494,0.551,0.540,0.295,0.094,-0.048>
links: <1,0,4,2,0,4,2,1,6,0,1,7,1,3,5,0,4,5,0,5,6>
hidden values: <0.000,-0.490,-0.494>
UnitTestNeuraNetGA OK
UnitTestNeuraNet OK
UnitTestAll OK
```

neuranet.txt:


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


bestnn.txt:


```
{
  "_nbInputVal":"3",
  "_nbOutputVal":"3",
```

```
  "_nbMaxHidVal":"3",
  "_nbMaxBases":"3",
  "_nbMaxLinks":"7",
  "_bases":{
    "_dim":"9",
    "_val":["0.501901","0.305286","-0.306601","-0.493893","0.551495","0.539556","0.294944","0.094123","-0.048077"]
  },
  "_links":{
    "_dim":"21",
    "_val":["1","0","4","2","0","4","2","1","6","0","1","7","1","3","5","0","4","5","0","5","6"]
  }
}
```


7 Validation

7.1 Iris data set

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

main.c:

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

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

// Nb 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 1000
// Save NeuraNet in compact format
#define COMPACT true

// Categories of data sets

typedef enum DataSetCat {
```

```
  unknownDataSet,
  datalearn,
  datatest
} DataSetCat;
#define NB_DATASET 3
const char* dataSetNames[NB_DATASET] = {
  "unknownDataSet", "datalearn", "datatest"
  };

// 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;
  }
  that->_nbSample = 75;
  that->_samples =
    PBErrMalloc(NeuraNetErr, sizeof(Iris) * that->_nbSample);
  char buffer[500];
  int ret = 0;
  if (cat == datalearn) {
    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) {
    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 {
    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 = -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;
    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));
  // 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) {
    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);
    GAInit(ga);
  }
  // Start learning process
  printf("Learning...\n");
  printf("Will stop when curEpoch >= %lu or bestVal >= %f\n",
    limitEpoch, STOP_LEARNING_AT_VAL);
  fflush(stdout);
  // Declare a variable to memorize the best value in the current epoch
```

```
float curBest = bestVal;
while (fabs(bestVal) < STOP_LEARNING_AT_VAL &&
  GAGetCurEpoch(ga) < limitEpoch) {
  // For each adn in the GenAlg
  for (int 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;
    // Display infos about the current epoch
    //printf("ep%lu ent%3d(age%6lu) val%.4f bestEpo%.4f bestAll%.4f         \r",
    //  GAGetCurEpoch(ga), iEnt, GAAdnGetAge(adn), value, curBest,
    //  bestVal);
    fflush(stdout);
  }
  // Step the GenAlg
  GAStep(ga);
  // If there has been improvement during this epoch
  if (curBest > bestVal) {
    bestVal = curBest;
    // Display info about the improvment
    printf("\nImprovement at epoch %lu: %f\n",
      GAGetCurEpoch(ga), bestVal);
    fflush(stdout);
    // Set the links and base functions of the NeuraNet according
    // to the best adn
    if (GABestAdnF(ga) != NULL)
      NNSetBases(nn, GABestAdnF(ga));
    if (GABestAdnI(ga) != NULL)
      NNSetLinks(nn, GABestAdnI(ga));
    // 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;
    }
    printf("Saved the best NeuraNet in ./bestnn.txt\n");
    fclose(fd);
  }
  // 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;
  }
  printf("\nLearning complete\n");
  // 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);
  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;
  printf("Prediction: %s\n", irisCatNames[pred]);
```

```
  // 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 >= 1000 or bestVal >= 0.999000

Improvement at epoch 1: 0.666667
Saved the best NeuraNet in ./bestnn.txt

Improvement at epoch 14: 0.706667
Saved the best NeuraNet in ./bestnn.txt

Improvement at epoch 47: 0.960000
Saved the best NeuraNet in ./bestnn.txt

Improvement at epoch 125: 0.973333
Saved the best NeuraNet in ./bestnn.txt

Learning complete
```

validation:

```
#74 pred2 real2 <-1.000,0.000,0.010> OK
#73 pred2 real2 <-1.000,0.000,0.010> OK
#72 pred2 real2 <-1.000,0.000,0.010> OK
#71 pred2 real2 <-1.000,0.000,0.010> OK
#70 pred2 real2 <-1.000,0.000,0.010> OK
#69 pred2 real2 <-1.000,0.000,0.010> OK
#68 pred2 real2 <-1.000,0.000,0.010> OK
#67 pred2 real2 <-1.000,0.000,0.010> OK
#66 pred2 real2 <-1.000,0.000,0.010> OK
#65 pred2 real2 <-1.000,0.000,0.010> OK
#64 pred2 real2 <-1.000,0.000,0.010> OK
#63 pred2 real2 <-1.000,0.000,0.010> OK
#62 pred2 real2 <-1.000,0.000,0.010> OK
#61 pred2 real2 <-1.000,0.000,0.010> OK
#60 pred2 real2 <-1.000,0.000,0.010> OK
#59 pred1 real2 <-1.000,0.000,-0.354> NG
#58 pred1 real2 <-1.000,0.000,-0.191> NG
#57 pred2 real2 <-1.000,0.000,0.010> OK
#56 pred2 real2 <-1.000,0.000,0.010> OK
#55 pred2 real2 <-1.000,0.000,0.010> OK
#54 pred1 real2 <-1.000,0.000,-0.029> NG
#53 pred2 real2 <-1.000,0.000,0.010> OK
#52 pred2 real2 <-1.000,0.000,0.010> OK
#51 pred2 real2 <-1.000,0.000,0.010> OK
#50 pred2 real2 <-1.000,0.000,0.010> OK
#49 pred1 real1 <-1.000,0.000,-0.517> OK
#48 pred1 real1 <-1.000,0.000,-0.843> OK
#47 pred1 real1 <-1.000,0.000,-0.517> OK
#46 pred1 real1 <-1.000,0.000,-0.517> OK
#45 pred1 real1 <-1.000,0.000,-0.680> OK
#44 pred1 real1 <-1.000,0.000,-0.517> OK
#43 pred1 real1 <-0.756,0.000,-1.006> OK
#42 pred1 real1 <-1.000,0.000,-0.680> OK
#41 pred1 real1 <-1.000,0.000,-0.354> OK
#40 pred1 real1 <-1.000,0.000,-0.680> OK
#39 pred1 real1 <-1.000,0.000,-0.517> OK
#38 pred1 real1 <-1.000,0.000,-0.517> OK
#37 pred1 real1 <-1.000,0.000,-0.517> OK
#36 pred1 real1 <-1.000,0.000,-0.191> OK
#35 pred1 real1 <-1.000,0.000,-0.029> OK
#34 pred1 real1 <-1.000,0.000,-0.191> OK
#33 pred1 real1 <-1.000,0.000,-0.029> OK
#32 pred1 real1 <-1.000,0.000,-0.680> OK
```

```
#31 pred1 real1 <-0.756,0.000,-1.006> OK
#30 pred1 real1 <-1.000,0.000,-0.843> OK
#29 pred1 real1 <-0.756,0.000,-1.006> OK
#28 pred1 real1 <-1.000,0.000,-0.191> OK
#27 pred2 real1 <-1.000,0.000,0.010> NG
#26 pred1 real1 <-1.000,0.000,-0.354> OK
#25 pred1 real1 <-1.000,0.000,-0.354> OK
#24 pred0 real0 <1.000,0.000,-1.990> OK
#23 pred0 real0 <1.000,0.000,-1.990> OK
#22 pred0 real0 <1.000,0.000,-1.990> OK
#21 pred0 real0 <1.000,0.000,-1.990> OK
#20 pred0 real0 <1.000,0.000,-1.990> OK
#19 pred0 real0 <1.000,0.000,-1.983> OK
#18 pred0 real0 <1.000,0.000,-1.657> OK
#17 pred0 real0 <1.000,0.000,-1.990> OK
#16 pred0 real0 <1.000,0.000,-1.990> OK
#15 pred0 real0 <1.000,0.000,-1.990> OK
#14 pred0 real0 <1.000,0.000,-1.990> OK
#13 pred0 real0 <1.000,0.000,-1.990> OK
#12 pred0 real0 <1.000,0.000,-1.990> OK
#11 pred0 real0 <1.000,0.000,-1.990> OK
#10 pred0 real0 <1.000,0.000,-1.990> OK
#9 pred0 real0 <1.000,0.000,-1.990> OK
#8 pred0 real0 <1.000,0.000,-1.990> OK
#7 pred0 real0 <1.000,0.000,-1.990> OK
#6 pred0 real0 <1.000,0.000,-1.983> OK
#5 pred0 real0 <1.000,0.000,-1.990> OK
#4 pred0 real0 <1.000,0.000,-1.990> OK
#3 pred0 real0 <1.000,0.000,-1.990> OK
#2 pred0 real0 <1.000,0.000,-1.990> OK
#1 pred0 real0 <1.000,0.000,-1.983> OK
#0 pred0 real0 <1.000,0.000,-1.990> OK
Value: 0.946667
```