# NeuraMorph

P. Baillehache

September 3, 2020

## Contents



## Introduction

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

It uses the `PBErr`, `PBMath`, `GSet` library.

# 1 Definitions

# 2 Interface

```
// ============ NEURAMORPH.H ================
```

```
#ifndef NEURAMORPH_H
#define NEURAMORPH_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"
#include "gdataset.h"

// ----- NeuraMorphUnit

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

typedef struct NeuraMorphUnit {

  // Input indices in parent NeuraMorph
  VecLong* iInputs;

  // Output indices in parent NeuraMorph
  VecLong* iOutputs;

  // Lowest and highest values for filtering inputs
  VecFloat* lowFilters;
  VecFloat* highFilters;

  // Lowest and highest values of outputs
  VecFloat* lowOutputs;
  VecFloat* highOutputs;

  // Vector to memorize the output values
  VecFloat* outputs;

  // Transfer function coefficients
  // Seen as (nb output) triangular matrices of size (nb input + 1)
  VecFloat** coeffs;

  // Working variables to avoid reallocation of memory at each Evaluate()
  bool* activeInputs;
  VecFloat* unitInputs;

  // Variable to memorize the value of the unit during training
  float value;

} NeuraMorphUnit;

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

// Create a new NeuraMorphUnit between the input 'iInputs' and the
// outputs 'iOutputs'
NeuraMorphUnit* NeuraMorphUnitCreate(
  const VecLong* iInputs,
  const VecLong* iOutputs);

// Free the memory used by the NeuraMorphUnit 'that'
void NeuraMorphUnitFree(NeuraMorphUnit** that);
```

```
// Get the input indices of the NeuraMorphUnit 'that'
#if BUILDMODE != 0
static inline
#endif
const VecLong* NMUnitIInputs(const NeuraMorphUnit* that);

// Get the output indices of the NeuraMorphUnit 'that'
#if BUILDMODE != 0
static inline
#endif
const VecLong* NMUnitIOutputs(const NeuraMorphUnit* that);

// Get the output values of the NeuraMorphUnit 'that'
#if BUILDMODE != 0
static inline
#endif
const VecFloat* NMUnitOutputs(const NeuraMorphUnit* that);

// Calculate the outputs for the 'inputs' with the NeuraMorphUnit 'that'
// Update 'that->outputs'
void NMUnitEvaluate(
  NeuraMorphUnit* that,
  const VecFloat* inputs);

// Get the number of input values of the NeuraMorphUnit 'that'
#if BUILDMODE != 0
static inline
#endif
long NMUnitGetNbInputs(const NeuraMorphUnit* that);

// Get the number of output values of the NeuraMorphUnit 'that'
#if BUILDMODE != 0
static inline
#endif
long NMUnitGetNbOutputs(const NeuraMorphUnit* that);

// Get the value of the NeuraMorphUnit 'that'
#if BUILDMODE != 0
static inline
#endif
float NMUnitGetValue(const NeuraMorphUnit* that);

// Set the value of the NeuraMorphUnit 'that' to 'val'
#if BUILDMODE != 0
static inline
#endif
void NMUnitSetValue(
  NeuraMorphUnit* that,
            float val);

// Print the NeuraMorphUnit 'that' on the 'stream'
void NMUnitPrint(
  const NeuraMorphUnit* that,
                 FILE* stream);
#define NMUnitPrintln(T, S) \
  NMUnitPrint(T, S);fprintf(S, "\n")

// ----- NeuraMorph

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

typedef struct NeuraMorph {
```

```
  // Number of inputs and outputs
  long nbInput;
  long nbOutput;

  // Inputs and outputs values
  VecFloat* inputs;
  VecFloat* outputs;

  // Internal values
  VecFloat* hiddens;

  // Lowest and highest values for internal values
  VecFloat* lowHiddens;
  VecFloat* highHiddens;

  // GSet of NeuraMorphUnit
  GSet units;

} NeuraMorph;

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

// Create a new NeuraMorph with 'nbInput' inputs and 'nbOutput' outputs
NeuraMorph* NeuraMorphCreate(
  long nbInput,
  long nbOutput);

// Free the memory used by the NeuraMorph 'that'
void NeuraMorphFree(NeuraMorph** that);

// Get the number of input values of the NeuraMorph 'that'
#if BUILDMODE != 0
static inline
#endif
long NMGetNbInput(const NeuraMorph* that);

// Get the number of output values of the NeuraMorph 'that'
#if BUILDMODE != 0
static inline
#endif
long NMGetNbOutput(const NeuraMorph* that);

// Get the input values of the NeuraMorph 'that'
#if BUILDMODE != 0
static inline
#endif
VecFloat* NMInputs(NeuraMorph* that);

// Get the output values of the NeuraMorph 'that'
#if BUILDMODE != 0
static inline
#endif
const VecFloat* NMOutputs(const NeuraMorph* that);

// Get the number of hidden values of the NeuraMorph 'that'
#if BUILDMODE != 0
static inline
#endif
long NMGetNbHidden(const NeuraMorph* that);

// Set the number of hidden values of the NeuraMorph 'that' to 'nb'
```

```
#if BUILDMODE != 0
static inline
#endif
void NMSetNbHidden(
  NeuraMorph* that,
         long nb);

// Add one NeuraMorphUnit with input and output indices 'iInputs'
// and 'iOutputs' to the NeuraMorph 'that'
// Return the created NeuraMorphUnit
NeuraMorphUnit* NMAddUnit(
     NeuraMorph* that,
  const VecLong* iInputs,
  const VecLong* iOutputs);

// Remove the NeuraMorphUnit 'unit' from the NeuraMorph 'that'
// The NeuraMorphUnit is not freed
void NMRemoveUnit(
      NeuraMorph* that,
  NeuraMorphUnit* unit);

// Burry the NeuraMorphUnits in the 'units' set into the
// NeuraMorph 'that'
// 'units' is empty after calling this function
// The NeuraMorphUnits iOutputs must point toward the NeuraMorph
// outputs
// NeuraMorphUnits' iOutputs are redirected toward new hidden values
// 'that->hiddens' is resized as necessary
void NMBurryUnits(
  NeuraMorph* that,
        GSet* units);

// Get a new vector with indices of the outputs in the NeuraMorph 'that'
VecLong* NMGetVecIOutputs(const NeuraMorph* that);

// Evaluate the NeuraMorph 'that' on the 'inputs' values
void NMEvaluate(
  NeuraMorph* that,
    VecFloat* inputs);

// ----- NeuraMorphTrainer

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

typedef struct NeuraMorphTrainer {

  // Trained NeuraMorph
  NeuraMorph* neuraMorph;

  // Training dataset
  GDataSetVecFloat* dataset;

  // Index of the dataset's category used for training
  unsigned int iCatTraining;

  // Depth of the training
  short depth;

  // Threshold used to discard weakest units during training
  // in [0.0,1.0]
  float weakUnitThreshold;
```

```
} NeuraMorphTrainer;

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

// Create a static NeuraMorphTrainer for the NeuraMorph 'neuraMorph' and the
// GDataSet 'dataset'
// Default depth: 2
NeuraMorphTrainer NeuraMorphTrainerCreateStatic(
        NeuraMorph* neuraMorph,
  GDataSetVecFloat* dataset);

// Free the memory used by the static NeuraMorphTrainer 'that'
void NeuraMorphTrainerFreeStatic(NeuraMorphTrainer* that);

// Run the training process for the NeuraMorphTrainer 'that'
void NMTrainerRun(NeuraMorphTrainer* that);

// Get the depth of the NeuraMorphTrainer 'that'
#if BUILDMODE != 0
static inline
#endif
short NMTrainerGetDepth(const NeuraMorphTrainer* that);

// Set the depth of the NeuraMorphTrainer 'that' to 'depth'
#if BUILDMODE != 0
static inline
#endif
void NMTrainerSetDepth(
  NeuraMorphTrainer* that,
               short depth);

// Get the weakness threshold of the NeuraMorphTrainer 'that'
#if BUILDMODE != 0
static inline
#endif
float NMTrainerGetWeakThreshold(const NeuraMorphTrainer* that);

// Set the weakness threshold of the NeuraMorphTrainer 'that'
// to 'iCat'
#if BUILDMODE != 0
static inline
#endif
void NMTrainerSetWeakThreshold(
  NeuraMorphTrainer* that,
               float weakUnitThreshold);

// Get the index of the training category of the NeuraMorphTrainer 'that'
#if BUILDMODE != 0
static inline
#endif
unsigned int NMTrainerGetICatTraining(const NeuraMorphTrainer* that);

// Set the index of the training category of the NeuraMorphTrainer 'that'
// to 'iCat'
#if BUILDMODE != 0
static inline
#endif
void NMTrainerSetICatTraining(
  NeuraMorphTrainer* that,
        unsigned int iCatTraining);

// Get the NeuraMorph of the NeuraMorphTrainer 'that'
```

```
#if BUILDMODE != 0
static inline
#endif
NeuraMorph* NMTrainerNeuraMorph(const NeuraMorphTrainer* that);

// Get the GDataSet of the NeuraMorphTrainer 'that'
#if BUILDMODE != 0
static inline
#endif
GDataSetVecFloat* NMTrainerDataset(const NeuraMorphTrainer* that);

// ================ static inliner ====================

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

#endif
```

# 3 Code

## 3.1 neuramorph.c

```
// ============ NEURAMORPH.C ================

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

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

// ----- NeuraMorphUnit

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

// Return the number of coefficients of a NeuraMorphUnit having 'nbIn' inputs
long NMUnitGetNbCoeff(long nbIn);

// Get the coefficient for the pair of inputs 'iInputA', 'iInputB' in the
// NeuraMorphUnit 'that' for the output 'iOutput'
float NMUnitGetCoeff(
  const NeuraMorphUnit* that,
                  long iInputA,
                  long iInputB,
                  long iOutput);

// Update the low and high of the hiddens of the NeuraMorph 'that' with
// the low and high of its units
void NMUpdateLowHighHiddens(NeuraMorph* that);

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

// Create a new NeuraMorphUnit between the input 'iInputs' and the
// outputs 'iOutputs'
NeuraMorphUnit* NeuraMorphUnitCreate(
  const VecLong* iInputs,
  const VecLong* iOutputs) {
```

```
#if BUILDMODE == 0

  if (iInputs == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'iInputs' is null");
    PBErrCatch(NeuraMorphErr);

  }

  if (iOutputs == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'iOutputs' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  // Allocate memory for the NeuraMorphUnit
  NeuraMorphUnit* that =
    PBErrMalloc(
      NeuraMorphErr,
      sizeof(NeuraMorphUnit));

  // Get the number of inputs (including the constant) and outputs
  long nbIn = VecGetDim(iInputs) + 1;
  long nbOut = VecGetDim(iOutputs);

  // Init properties
  that->iInputs = VecClone(iInputs);
  that->iOutputs = VecClone(iOutputs);
  that->lowFilters = VecFloatCreate(nbIn);
  that->highFilters = VecFloatCreate(nbIn);
  that->lowOutputs = NULL;
  that->highOutputs = NULL;
  that->outputs = VecFloatCreate(nbOut);
  that->coeffs =
    PBErrMalloc(
      NeuraMorphErr,
      sizeof(VecFloat*) * nbOut);
  long nbCoeff = NMUnitGetNbCoeff(nbIn);
  for (
    long iOut = nbOut;
    iOut--;
    that->coeffs[iOut] = VecFloatCreate(nbCoeff));

  // 'nbIn + 1' for the constant
  that->activeInputs =
    PBErrMalloc(
      NeuraMorphErr,
      sizeof(bool) * nbIn);
  that->unitInputs = VecFloatCreate(nbIn);
  that->value = 0.0;

  // Set the input value, filters and active flag for the constant
```

```
  VecSet(
    that->unitInputs,
    0,
    1.0);
  that->activeInputs[0] = true;

  // Return the new NeuraMorphUnit
  return that;

}

// Free the memory used by the NeuraMorphUnit 'that'
void NeuraMorphUnitFree(NeuraMorphUnit** that) {

  // Check the input
  if (that == NULL || *that == NULL) {

    return;

  }

  // Free memory
  long nbOut = VecGetDim((*that)->iOutputs);
  VecFree(&((*that)->iInputs));
  VecFree(&((*that)->iOutputs));
  VecFree(&((*that)->lowFilters));
  VecFree(&((*that)->highFilters));
  if ((*that)->lowOutputs != NULL) {

    VecFree(&((*that)->lowOutputs));

  }

  if ((*that)->highOutputs != NULL) {

    VecFree(&((*that)->highOutputs));

  }

  VecFree(&((*that)->outputs));
  for (
    long iOut = nbOut;
    iOut--;
    VecFree((*that)->coeffs + iOut));
  free((*that)->coeffs);
  free((*that)->activeInputs);
  VecFree(&((*that)->unitInputs));
  free(*that);
  *that = NULL;

}

// Return the number of coefficients of a NeuraMorphUnit having 'nbIn' inputs
long NMUnitGetNbCoeff(long nbIn) {

#if BUILDMODE == 0

  if (nbIn <= 0) {

    NeuraMorphErr->_type = PBErrTypeInvalidArg;
    sprintf(
      NeuraMorphErr->_msg,
```

```
      "'nbIn' is invalid (%ld>0)",
      nbIn);
    PBErrCatch(NeuraMorphErr);

  }

#endif

  // Declare a variable to memorise the result
  long nb = 0;

  // Calculate the number of values in the triangular matrix of size
  // nbIn
  for (
    long i = nbIn;
    i >= 0;
    nb += (i--));

  // Return the result
  return nb;

}

// Calculate the outputs for the 'inputs' with the NeuraMorphUnit 'that'
// Update 'that->outputs'
void NMUnitEvaluate(
  NeuraMorphUnit* that,
  const VecFloat* inputs) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

  if (VecGetDim(inputs) != VecGetDim(that->iInputs)) {

    NeuraMorphErr->_type = PBErrTypeInvalidArg;
    sprintf(
      NeuraMorphErr->_msg,
      "'inputs' has invalid dimension (%ld!=%ld)",
      VecGetDim(inputs),
      VecGetDim(that->iInputs));
    PBErrCatch(NeuraMorphErr);

  }

#endif

  // Reset the outputs
  VecSetNull(that->outputs);

  // Update the active flags and scaled inputs (skip the constant)
  for (
    long iInput = 1;
    iInput < VecGetDim(that->unitInputs);
```

```
  ++iInput) {

  // Get the input value and its low/high filters
  float val =
    VecGet(
      inputs,
      iInput - 1);
  float low =
    VecGet(
      that->lowFilters,
      iInput);
  float high =
    VecGet(
      that->highFilters,
      iInput);

  // If the value is inside the filter
  if (
    low <= val &&
    val <= high &&
    (high - low) > PBMATH_EPSILON) {

    // Set this value as active
    that->activeInputs[iInput] = true;

    // Set the value in the unit inputs
    VecSet(
      that->unitInputs,
      iInput,
      val);

  // Else the value is outside the filter
  } else {

    // Set this value as inactive
    that->activeInputs[iInput] = false;

  }

}

// Loop on the pair of active inputs
for (
  long iInputA = 0;
  iInputA < VecGetDim(that->unitInputs);
  ++iInputA) {

  if (that->activeInputs[iInputA] == true) {

    for (
      long iInputB = 0;
      iInputB <= iInputA;
      ++iInputB) {

      if (that->activeInputs[iInputB] == true) {

        // Loop on the outputs
        for (
          long iOutput = 0;
          iOutput < VecGetDim(that->outputs);
          ++iOutput) {
```

```
          // Calculate the components for this output and pair of inputs
          float comp =
            VecGet(
              that->unitInputs,
              iInputA) *
            VecGet(
              that->unitInputs,
              iInputB) *
            NMUnitGetCoeff(
              that,
              iInputA,
              iInputB,
              iOutput);

          // Add the component to the output
          float cur =
            VecGet(
              that->outputs,
              iOutput);
          VecSet(
            that->outputs,
            iOutput,
            cur + comp);

        }

      }

    }

  }

}

// If the low and high values for outputs don't exist yet
if (that->lowOutputs == NULL) {

  // Create the low and high values by cloning the current output
  that->lowOutputs = VecClone(that->outputs);
  that->highOutputs = VecClone(that->outputs);

// Else, the low and high values for outputs exist
} else {

  // Loop on the outputs
  for (
    long iOutput = 0;
    iOutput < VecGetDim(that->outputs);
    ++iOutput) {

    // Update the low and high values for this output
    float val =
      VecGet(
        that->outputs,
        iOutput);

    float curLow =
      VecGet(
        that->lowOutputs,
        iOutput);
    if (curLow > val) {
```

```
        VecSet(
          that->lowOutputs,
          iOutput,
          val);

      }

      float curHigh =
        VecGet(
          that->highOutputs,
          iOutput);
      if (curHigh < val) {

        VecSet(
          that->highOutputs,
          iOutput,
          val);

      }

    }

  }

}

// Get the coefficient for the pair of inputs 'iInputA', 'iInputB' in the
// NeuraMorphUnit 'that' for the output 'iOutput'
float NMUnitGetCoeff(
  const NeuraMorphUnit* that,
                  long iInputA,
                  long iInputB,
                  long iOutput) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

  if (
    iInputA < 0 ||
    iInputA >= VecGetDim(that->unitInputs)) {

    NeuraMorphErr->_type = PBErrTypeInvalidArg;
    sprintf(
      NeuraMorphErr->_msg,
      "'iInputA' is invalid (0<=%ld<%ld)",
      iInputA,
      VecGetDim(that->unitInputs));
    PBErrCatch(NeuraMorphErr);

  }

  if (
    iInputB < 0 ||
```

```
    iInputB >= VecGetDim(that->unitInputs)) {

    NeuraMorphErr->_type = PBErrTypeInvalidArg;
    sprintf(
      NeuraMorphErr->_msg,
      "'iInputB' is invalid (0<=%ld<%ld)",
      iInputB,
      VecGetDim(that->unitInputs));
    PBErrCatch(NeuraMorphErr);

  }

  if (iInputA < iInputB) {

    NeuraMorphErr->_type = PBErrTypeInvalidArg;
    sprintf(
      NeuraMorphErr->_msg,
      "The pair of indices is invalid (%ld>=%ld)",
      iInputA,
      iInputB);
    PBErrCatch(NeuraMorphErr);

  }

  if (
    iOutput < 0 ||
    iOutput >= VecGetDim(that->outputs)) {

    NeuraMorphErr->_type = PBErrTypeInvalidArg;
    sprintf(
      NeuraMorphErr->_msg,
      "'iInputB' is invalid (0<=%ld<%ld)",
      iInputB,
      VecGetDim(that->outputs));
    PBErrCatch(NeuraMorphErr);

  }

#endif

  // Calculate the index of the coefficient
  long iCoeff = 0;
  for (
    long shift = 0;
    shift < iInputA;
    iCoeff += (shift++) + 1);
  iCoeff += iInputB;

  // Return the coefficient
  float coeff =
    VecGet(
      that->coeffs[iOutput],
      iCoeff);
  return coeff;

}

// Print the NeuraMorphUnit 'that' on the 'stream'
void NMUnitPrint(
  const NeuraMorphUnit* that,
                 FILE* stream) {
```

```
#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

  if (stream == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'stream' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  VecPrint(
    NMUnitIInputs(that),
    stream);
  fprintf(
    stream,
    " -> ");
  VecPrint(
    NMUnitIOutputs(that),
    stream);
  fprintf(
    stream,
    " (%04.6f)",
    NMUnitGetValue(that));

}

// ----- NeuraMorph

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

// Create a new NeuraMorph with 'nbInput' inputs and 'nbOutput' outputs
NeuraMorph* NeuraMorphCreate(
  long nbInput,
  long nbOutput) {

  // Allocate memory for the NeuraMorph
  NeuraMorph* that =
    PBErrMalloc(
      NeuraMorphErr,
      sizeof(NeuraMorph));

  // Init properties
  that->nbInput = nbInput;
  that->nbOutput = nbOutput;
  that->inputs = VecFloatCreate(nbInput);
  that->outputs = VecFloatCreate(nbOutput);
  that->hiddens = NULL;
  that->lowHiddens = NULL;
```

```
  that->highHiddens = NULL;
  that->units = GSetCreateStatic();

  // Return the NeuraMorph
  return that;

}

// Free the memory used by the NeuraMorph 'that'
void NeuraMorphFree(NeuraMorph** that) {

  // Check the input
  if (that == NULL || *that == NULL) {

    return;

  }

  // Free memory
  VecFree(&((*that)->inputs));
  VecFree(&((*that)->outputs));
  if ((*that)->hiddens != NULL) {

    VecFree(&((*that)->hiddens));
    VecFree(&((*that)->lowHiddens));
    VecFree(&((*that)->highHiddens));

  }

  while (GSetNbElem(&((*that)->units)) > 0) {

    NeuraMorphUnit* unit = GSetPop(&((*that)->units));
    NeuraMorphUnitFree(&unit);

  }

  free(*that);
  *that = NULL;

}

// Add one NeuraMorphUnit with input and output indices 'iInputs'
// and 'iOutputs' to the NeuraMorph 'that'
// Return the created NeuraMorphUnit
NeuraMorphUnit* NMAddUnit(
     NeuraMorph* that,
  const VecLong* iInputs,
  const VecLong* iOutputs) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

  if (iInputs == NULL) {
```

```
    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'iInputs' is null");
    PBErrCatch(NeuraMorphErr);

  }

  if (iOutputs == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'iOutputs' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  // Create the NeuraMorphUnit
  NeuraMorphUnit* unit =
    NeuraMorphUnitCreate(
      iInputs,
      iOutputs);

  // Append the new NeuraorphUnit to the set of NeuraMorphUnit
  GSetAppend(
    &(that->units),
    unit);

  // Return the new unit
  return unit;

}

// Remove the NeuraMorphUnit 'unit' from the NeuraMorph 'that'
// The NeuraMorphUnit is not freed
void NMRemoveUnit(
      NeuraMorph* that,
  NeuraMorphUnit* unit) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  // Remove the NeuraorphUnit from the set of NeuraMorphUnit
  GSetRemoveAll(
    &(that->units),
    unit);
```

```
}

// Burry the NeuraMorphUnits in the 'units' set into the
// NeuraMorph 'that'
// 'units' is empty after calling this function
// The NeuraMorphUnits iOutputs must point toward the NeuraMorph
// outputs
// NeuraMorphUnits' iOutputs are redirected toward new hidden values
// 'that->hiddens' is resized as necessary
void NMBurryUnits(
  NeuraMorph* that,
        GSet* units) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  // Declare a variable to memorize the number of hidden values
  // to add
  long nbHiddenValues = 0;

  // While there are units to burry
  while (GSetNbElem(units) > 0) {

    // Get the unit
    NeuraMorphUnit* unit = GSetPop(units);

    // Loop on the iOutputs of the unit
    for (
      long iOutput = 0;
      iOutput < VecGetDim(NMUnitIOutputs(unit));
      ++iOutput) {

      long indice =
        VecGet(
          NMUnitIOutputs(unit),
          iOutput);
      VecSet(
        unit->iOutputs,
        iOutput,
        indice + nbHiddenValues);

    }

    // Append the unit to the set of NeuraMorphUnit
    GSetAppend(
      &(that->units),
      unit);

    // Update the number of new hidden values
    nbHiddenValues += VecGetDim(NMUnitIOutputs(unit));
```

```
  }

  // If there is already hidden values
  if (that->hiddens != NULL) {

    // Add the previous number of hidden values
    nbHiddenValues += VecGetDim(that->hiddens);

    // Free memory
    VecFree(&(that->hiddens));
    VecFree(&(that->lowHiddens));
    VecFree(&(that->highHiddens));

  }

  // If there are hidden values after burrying
  if (nbHiddenValues > 0) {

    // Resize the hiddens value vector
    that->hiddens = VecFloatCreate(nbHiddenValues);
    that->lowHiddens = VecFloatCreate(nbHiddenValues);
    that->highHiddens = VecFloatCreate(nbHiddenValues);

    // Update the low and high of the hiddens with the low and high
    // of the units
    NMUpdateLowHighHiddens(that);

  }

}

// Get a new vector with indices of the outputs in the NeuraMorph 'that'
VecLong* NMGetVecIOutputs(const NeuraMorph* that) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  // Allocate memory for the result
  VecLong* iOutputs = VecLongCreate(NMGetNbOutput(that));

  // Loop on indices
  for (
    long iOutput = 0;
    iOutput < NMGetNbOutput(that);
    ++iOutput) {

    // Set the indice of this output
    VecSet(
      iOutputs,
      iOutput,
      iOutput + NMGetNbHidden(that));
```

```
  }

  // Return the result
  return iOutputs;

}

// Update the low and high of the hiddens of the NeuraMorph 'that' with
// the low and high of its units
void NMUpdateLowHighHiddens(NeuraMorph* that) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  // Loop on the units
  GSetIterForward iter =
    GSetIterForwardCreateStatic(&(that->units));
  do {

    // Get the unit
    NeuraMorphUnit* unit = GSetIterGet(&iter);

    // Loop on the iOutputs of the unit
    for (
      long iOutput = 0;
      iOutput < VecGetDim(NMUnitIOutputs(unit));
      ++iOutput) {

      // Get the indice
      long indice =
        VecGet(
          NMUnitIOutputs(unit),
          iOutput);

      // If the indice points to a hidden value
      if (indice < NMGetNbHidden(that)) {

        // If the low and high exist
        if (
          unit->lowOutputs != NULL &&
          unit->highOutputs != NULL) {

          // Update the low and high
          float low =
            VecGet(
              unit->lowOutputs,
              iOutput);
          float high =
            VecGet(
              unit->highOutputs,
```

```
            iOutput);
          VecSet(
            that->lowHiddens,
            indice,
            low);
          VecSet(
            that->highHiddens,
            indice,
            high);

        }

      }

    }

  } while (GSetIterStep(&iter));

}

// Evaluate the NeuraMorph 'that' on the 'inputs' values
void NMEvaluate(
  NeuraMorph* that,
    VecFloat* inputs) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

  if (inputs == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'inputs' is null");
    PBErrCatch(NeuraMorphErr);

  }

  if (VecGetDim(inputs) != VecGetDim(that->inputs)) {

    NeuraMorphErr->_type = PBErrTypeInvalidArg;
    sprintf(
      NeuraMorphErr->_msg,
      "'inputs' has invalid size (%ld==%ld)",
      VecGetDim(inputs),
      VecGetDim(that->inputs));
    PBErrCatch(NeuraMorphErr);

  }

#endif

  // Copy the inputs into the internal inputs
```

```
VecCopy(
  that->inputs,
  inputs);

// Reset the internal outputs
VecSetNull(that->outputs);

// Loop on the units
GSetIterForward iter = GSetIterForwardCreateStatic(&(that->units));
do {

  // Get the unit
  NeuraMorphUnit* unit = GSetIterGet(&iter);

  // Allocate memory for inputs sent to the unit
  VecFloat* unitInputs = VecFloatCreate(NMUnitGetNbInputs(unit));

  // Loop on the input indices of the unit
  for (
    long iInput = 0;
    iInput < NMUnitGetNbInputs(unit);
    ++iInput) {

    // Get the input indice
    long indiceInput =
      VecGet(
        NMUnitIInputs(unit),
        iInput);

    // Declare a variable to memorize the input value
    float val = 0.0;

    // If this indice points toward an input
    if (indiceInput < NMGetNbInput(that)) {

      // Get the input value of the NeuraMorph for this indice
      val =
        VecGet(
          NMInputs(that),
          indiceInput);

    // Else, the indice points toward a hidden value
    } else {

      // Get the hidden value of the NeuraMorph for this indice
      val =
        VecGet(
          that->hiddens,
          indiceInput - NMGetNbInput(that));

    }

    // Set the input value for the unit for this indice
    VecSet(
      unitInputs,
      iInput,
      val);

  }

  // Evaluate the unit
  NMUnitEvaluate(
```

```
      unit,
      unitInputs);

    // Free the memory used by the unit input
    VecFree(&unitInputs);

    // Loop on the output indices of the unit
    for (
      long iOutput = 0;
      iOutput < NMUnitGetNbOutputs(unit);
      ++iOutput) {

      // Get the output value of the unit for this indice
      float val =
        VecGet(
          NMUnitOutputs(unit),
          iOutput);

      // Get the output indice
      long indiceOutput =
        VecGet(
          NMUnitIOutputs(unit),
          iOutput);

      // If the indice points toward a hidden
      if (indiceOutput < NMGetNbHidden(that)) {

        // Set the hidden value of the NeuraMorph for this indice
        VecSet(
          that->hiddens,
          indiceOutput,
          val);

      // Else, the indice points toward an output
      } else {

        // Set the output value of the NeuraMorph for this indice
        VecSet(
          that->outputs,
          indiceOutput - NMGetNbHidden(that),
          val);

      }

    }

  } while (GSetIterStep(&iter));

}

// ----- NeuraMorphTrainer

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

// Return true if the vector 'v' is a valid indices configuration
// i.e. v[i]<v[j] for all i<j
bool NMTrainerIsValidInputConfig(
  const VecLong* v,
            long iMinInput);

// Train a new NeuraMorphUnit with the interface defined by 'iInputs'
// and 'iOutputs', and add it to the set, sorted on its value
```

```
void NMTrainerTrainUnit(
  NeuraMorphTrainer* that,
              GSet* trainedUnits,
      const VecLong* iInputs,
      const VecLong* iOutputs);

// ================ Functions implemetation ====================

// Create a static NeuraMorphTrainer for the NeuraMorph 'neuraMorph' and the
// GDataSet 'dataset'
// Default depth: 2
NeuraMorphTrainer NeuraMorphTrainerCreateStatic(
        NeuraMorph* neuraMorph,
  GDataSetVecFloat* dataset) {

#if BUILDMODE == 0

  if (neuraMorph == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'neuraMorph' is null");
    PBErrCatch(NeuraMorphErr);

  }

  if (dataset == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'dataset' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  // Declare the new NeuraMorphTrainer
  NeuraMorphTrainer that;

  // Init properties
  that.neuraMorph = neuraMorph;
  that.dataset = dataset;
  that.depth = 2;
  that.iCatTraining = 0;
  that.weakUnitThreshold = 0.9;

  // Return the NeuraMorphTrainer
  return that;

}

// Free the memory used by the static NeuraMorphTrainer 'that'
void NeuraMorphTrainerFreeStatic(NeuraMorphTrainer* that) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
```

```
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  // Nothing to do

}

// Run the training process for the NeuraMorphTrainer 'that'
void NMTrainerRun(NeuraMorphTrainer* that) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  // Declare a variable to memorize the minimum index needed in the
  // inputs of the new unit to ensure we do not train twice the same
  // unit
  long iMinInput = 0;

  // Loop on training depth
  for (
    short iDepth = 1;
    iDepth <= NMTrainerGetDepth(that);
    ++iDepth) {

    printf(
      "Depth %d/%d...\n",
      iDepth,
      NMTrainerGetDepth(that));

    // Get the number of available inputs for the new unit
    long nbAvailInputs =
      NMGetNbInput(NMTrainerNeuraMorph(that)) +
      NMGetNbHidden(NMTrainerNeuraMorph(that));

    printf(
      "Nb available inputs: %ld\n",
      nbAvailInputs);

    // Get the output indices
    VecLong* iOutputs = NMGetVecIOutputs(NMTrainerNeuraMorph(that));

    // Declare a set to memorize the trained units
    GSet trainedUnits = GSetCreateStatic();

    // Loop on the number of inputs for the new unit
```

```
// TODO restrain nbUnitInput to a maximum
for (
  long nbUnitInputs = 1;
  nbUnitInputs <= nbAvailInputs;
  ++nbUnitInputs) {

  printf(
    "Train units with %04ld inputs\n",
    nbUnitInputs);

  // Loop on the possible input configurations for the new units
  VecLong* iInputs = VecLongCreate(nbUnitInputs);
  VecLong* iInputsBound = VecLongCreate(nbUnitInputs);
  VecSetAll(
    iInputsBound,
    nbAvailInputs);
  bool hasStepped = true;
  do {

    bool isValidInputConfig =
      NMTrainerIsValidInputConfig(
        iInputs,
        iMinInput);
    if (isValidInputConfig == true) {

      // Train the unit
      NMTrainerTrainUnit(
        that,
        &trainedUnits,
        iInputs,
        iOutputs);

      }

    // Step to the next input configuration
    hasStepped =
      VecStep(
        iInputs,
        iInputsBound);

  } while (hasStepped);

  // Free memory
  VecFree(&iInputs);
  VecFree(&iInputsBound);

}

// If this is the last depth
if (iDepth == NMTrainerGetDepth(that)) {

  // Add the best of all units to the NeuraMorph
  NeuraMorphUnit* bestUnit = GSetDrop(&trainedUnits);
  GSetAppend(
    &(NMTrainerNeuraMorph(that)->units),
    bestUnit);

  printf("Add the last unit\n");
  NMUnitPrintln(
    bestUnit,
    stdout);
```

```
      // Discard all other units
      while (GSetNbElem(&trainedUnits) > 0) {

        NeuraMorphUnit* unit = GSetPop(&trainedUnits);
        NeuraMorphUnitFree(&unit);

      }

    // Else, this is not the last depth
    } else {

      // Get the value of the weakest and strongest units
      float weakVal = GSetElemGetSortVal(GSetHeadElem(&trainedUnits));
      float strongVal = GSetElemGetSortVal(GSetTailElem(&trainedUnits));

      // Get the threshold to discard the weakest units
      float threshold =
        weakVal + (strongVal - weakVal) *
        NMTrainerGetWeakThreshold(that);

      // Discard the weakest units
      long nbTrainedUnits = GSetNbElem(&trainedUnits);
      while (
        GSetElemGetSortVal(GSetHeadElem(&trainedUnits)) < threshold) {

        NeuraMorphUnit* unit = GSetPop(&trainedUnits);
        NeuraMorphUnitFree(&unit);

      }

      printf(
        "Burry %ld out of %ld units\n",
        GSetNbElem(&trainedUnits),
        nbTrainedUnits);
      GSetIterForward iter = GSetIterForwardCreateStatic(&trainedUnits);
      do {

        NeuraMorphUnit* unit = GSetIterGet(&iter);
        NMUnitPrintln(
          unit,
          stdout);

      } while (GSetIterStep(&iter));

      // Burry the remaining units
      NMBurryUnits(
        NMTrainerNeuraMorph(that),
        &trainedUnits);

    }

    // Update the minimum index of a valid configuration
    iMinInput = nbAvailInputs;

    // Free memory
    VecFree(&iOutputs);

  }

}

// Return true if the vector 'v' is a valid indices configuration
```

```
// i.e. v[i]<v[j] for all i<j and there exists i such as
// v[i]>=iMinInput
bool NMTrainerIsValidInputConfig(
  const VecLong* v,
            long iMinInput) {

#if BUILDMODE == 0

  if (v == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'v' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  bool noveltyCond = false;
  long a =
    VecGet(
      v,
      0);
  if (a >= iMinInput) {

    noveltyCond = true;

  }

  for (
    long i = 1;
    i < VecGetDim(v);
    ++i) {

    long b =
      VecGet(
        v,
        i);
    if (a >= b) {

      return false;

    }

    a = b;

    if (a >= iMinInput) {

      noveltyCond = true;

    }

  }

  return noveltyCond;

}

// Train a new NeuraMorphUnit with the interface defined by 'iInputs'
// and 'iOutputs', and add it to the set, sorted on its value
```

```
void NMTrainerTrainUnit(
  NeuraMorphTrainer* that,
              GSet* trainedUnits,
      const VecLong* iInputs,
      const VecLong* iOutputs) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

  if (trainedUnits == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'trainedUnits' is null");
    PBErrCatch(NeuraMorphErr);

  }

  if (iInputs == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'iInputs' is null");
    PBErrCatch(NeuraMorphErr);

  }

  if (iOutputs == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'iOutputs' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  // Create the unit
  NeuraMorphUnit* unit =
    NeuraMorphUnitCreate(
      iInputs,
      iOutputs);

  // TODO
  NMUnitSetValue(
    unit,
    rand());

  // Add the unit to the set of trained units
```

```
  GSetAddSort(
    trainedUnits,
    unit,
    NMUnitGetValue(unit));

}
```

## 3.2 neuramorph-inline.c

```
// ============ NEURAMORPH-INLINE.C ================

// ----- NeuraMorphUnit

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

// Get the input indices of the NeuraMorphUnit 'that'
#if BUILDMODE != 0
static inline
#endif
const VecLong* NMUnitIInputs(const NeuraMorphUnit* that) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  return that->iInputs;

}

// Get the output indices of the NeuraMorphUnit 'that'
#if BUILDMODE != 0
static inline
#endif
const VecLong* NMUnitIOutputs(const NeuraMorphUnit* that) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  return that->iOutputs;
```

```
}

// Get the output values of the NeuraMorphUnit 'that'
#if BUILDMODE != 0
static inline
#endif
const VecFloat* NMUnitOutputs(const NeuraMorphUnit* that) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  return that->outputs;

}

// Get the number of input values of the NeuraMorphUnit 'that'
#if BUILDMODE != 0
static inline
#endif
long NMUnitGetNbInputs(const NeuraMorphUnit* that) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  return VecGetDim(that->iInputs);

}

// Get the number of output values of the NeuraMorphUnit 'that'
#if BUILDMODE != 0
static inline
#endif
long NMUnitGetNbOutputs(const NeuraMorphUnit* that) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
```

```
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  return VecGetDim(that->iOutputs);

}

// Get the value of the NeuraMorphUnit 'that'
#if BUILDMODE != 0
static inline
#endif
float NMUnitGetValue(const NeuraMorphUnit* that) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  return that->value;

}

// Set the value of the NeuraMorphUnit 'that' to 'val'
#if BUILDMODE != 0
static inline
#endif
void NMUnitSetValue(
  NeuraMorphUnit* that,
            float val) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  that->value = val;

}
```

```
// ----- NeuraMorph

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

// Get the number of input values of the NeuraMorph 'that'
#if BUILDMODE != 0
static inline
#endif
long NMGetNbInput(const NeuraMorph* that) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  return that->nbInput;

}

// Get the number of output values of the NeuraMorph 'that'
#if BUILDMODE != 0
static inline
#endif
long NMGetNbOutput(const NeuraMorph* that) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  return that->nbOutput;

}

// Get the input values of the NeuraMorph 'that'
#if BUILDMODE != 0
static inline
#endif
VecFloat* NMInputs(NeuraMorph* that) {

#if BUILDMODE == 0

  if (that == NULL) {
```

```
    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  return that->inputs;

}

// Get the output values of the NeuraMorph 'that'
#if BUILDMODE != 0
static inline
#endif
const VecFloat* NMOutputs(const NeuraMorph* that) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  return that->outputs;

}

// Get the number of hidden values of the NeuraMorph 'that'
#if BUILDMODE != 0
static inline
#endif
long NMGetNbHidden(const NeuraMorph* that) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  long nb = 0;
  if (that->hiddens != NULL) {
```

```
    nb = VecGetDim(that->hiddens);

  }

  return nb;

}

// Set the number of hidden values of the NeuraMorph 'that' to 'nb'
#if BUILDMODE != 0
static inline
#endif
void NMSetNbHidden(
  NeuraMorph* that,
         long nb) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

  if (nb <= 0) {

    NeuraMorphErr->_type = PBErrTypeInvalidArg;
    sprintf(
      NeuraMorphErr->_msg,
      "'nb' is invalid (%ld>0)",
      nb);
    PBErrCatch(NeuraMorphErr);

  }

#endif

  if (that->hiddens != NULL) {

    VecFree(&(that->hiddens));

  }

  that->hiddens = VecFloatCreate(nb);

}

// ----- NeuraMorphTrainer

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

// Get the depth of the NeuraMorphTrainer 'that'
#if BUILDMODE != 0
static inline
#endif
short NMTrainerGetDepth(const NeuraMorphTrainer* that) {

#if BUILDMODE == 0
```

```
  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  return that->depth;

}

// Set the depth of the NeuraMorphTrainer 'that' to 'depth'
#if BUILDMODE != 0
static inline
#endif
void NMTrainerSetDepth(
  NeuraMorphTrainer* that,
               short depth) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

  if (depth < 1) {

    NeuraMorphErr->_type = PBErrTypeInvalidArg;
    sprintf(
      NeuraMorphErr->_msg,
      "'depth' is invalid (%d>=1)",
      depth);
    PBErrCatch(NeuraMorphErr);

  }

#endif

  that->depth = depth;

}

// Get the NeuraMorph of the NeuraMorphTrainer 'that'
#if BUILDMODE != 0
static inline
#endif
NeuraMorph* NMTrainerNeuraMorph(const NeuraMorphTrainer* that) {

#if BUILDMODE == 0
```

```
  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  return that->neuraMorph;

}

// Get the GDataSet of the NeuraMorphTrainer 'that'
#if BUILDMODE != 0
static inline
#endif
GDataSetVecFloat* NMTrainerDataset(const NeuraMorphTrainer* that) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  return that->dataset;

}

// Get the index of the training category of the NeuraMorphTrainer 'that'
#if BUILDMODE != 0
static inline
#endif
unsigned int NMTrainerGetICatTraining(const NeuraMorphTrainer* that) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  return that->iCatTraining;
```

```
}

// Set the index of the training category of the NeuraMorphTrainer 'that'
// to 'iCat'
#if BUILDMODE != 0
static inline
#endif
void NMTrainerSetICatTraining(
  NeuraMorphTrainer* that,
        unsigned int iCatTraining) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  that->iCatTraining = iCatTraining;

}

// Get the weakness threshold of the NeuraMorphTrainer 'that'
#if BUILDMODE != 0
static inline
#endif
float NMTrainerGetWeakThreshold(const NeuraMorphTrainer* that) {

#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  return that->weakUnitThreshold;

}

// Set the weakness threshold of the NeuraMorphTrainer 'that'
// to 'iCat'
#if BUILDMODE != 0
static inline
#endif
void NMTrainerSetWeakThreshold(
  NeuraMorphTrainer* that,
               float weakUnitThreshold) {
```

```
#if BUILDMODE == 0

  if (that == NULL) {

    NeuraMorphErr->_type = PBErrTypeNullPointer;
    sprintf(
      NeuraMorphErr->_msg,
      "'that' is null");
    PBErrCatch(NeuraMorphErr);

  }

#endif

  that->weakUnitThreshold = weakUnitThreshold;

}
```

# 4 Makefile

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

all: pbmake_wget main

# Automatic installation of the repository PBMake in the parent folder
pbmake_wget:
if [ ! -d ../PBMake ]; then wget https://github.com/BayashiPascal/PBMake/archive/master.zip; unzip master.zip; rm -f

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

# Rules to make the executable
repo=neuramorph
$($(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 "neuramorph.h"

void UnitTestNeuraMorphUnitCreateFree() {

  VecLong* iIn = VecLongCreate(3);
  VecSet(
    iIn,
    0,
    0);
  VecSet(
    iIn,
    1,
    1);
  VecSet(
    iIn,
    2,
    2);
  VecLong* iOut = VecLongCreate(2);
  VecSet(
    iOut,
    0,
    0);
  VecSet(
    iOut,
    1,
    1);
  NeuraMorphUnit* unit =
    NeuraMorphUnitCreate(
      iIn,
      iOut);
  bool isSame =
    ISEQUALF(
      unit->value,
      0.0);
  if (
    VecGetDim(unit->coeffs[0]) != 10 ||
    VecGetDim(unit->outputs) != 2 ||
    VecGetDim(unit->lowFilters) != 4 ||
    VecGetDim(unit->highFilters) != 4 ||
    VecGetDim(unit->unitInputs) != 4 ||
    isSame != true ||
    unit->lowOutputs != NULL ||
    unit->highOutputs != NULL) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
      NeuraMorphErr->_msg,
      "NeuraMorphUnitCreate failed (1)");
    PBErrCatch(NeuraMorphErr);

  }

  isSame =
    VecIsEqual(
      unit->iInputs,
      iIn);
  if (isSame == false) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
```

```
    sprintf(
      NeuraMorphErr->_msg,
      "NeuraMorphUnitCreate failed (2)");
    PBErrCatch(NeuraMorphErr);

  }

  isSame =
    VecIsEqual(
      unit->iOutputs,
      iOut);
  if (isSame == false) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
      NeuraMorphErr->_msg,
      "NeuraMorphUnitCreate failed (3)");
    PBErrCatch(NeuraMorphErr);

  }

  NeuraMorphUnitFree(&unit);
  if (unit != NULL) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
      NeuraMorphErr->_msg,
      "NeuraMorphUnitFree failed");
    PBErrCatch(NeuraMorphErr);

  }

  VecFree(&iIn);
  VecFree(&iOut);
  printf("UnitTestNeuraMorphUnitCreateFree OK\n");

}

void UnitTestNeuraMorphUnitGetSetPrint() {

  VecLong* iIn = VecLongCreate(3);
  VecLong* iOut = VecLongCreate(2);
  NeuraMorphUnit* unit =
    NeuraMorphUnitCreate(
      iIn,
      iOut);

  if (NMUnitIInputs(unit) != unit->iInputs) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
      NeuraMorphErr->_msg,
      "NMUnitIInputs failed");
    PBErrCatch(NeuraMorphErr);

  }

  if (NMUnitIOutputs(unit) != unit->iOutputs) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
      NeuraMorphErr->_msg,
```

```
    "NMUnitIOutputs failed");
  PBErrCatch(NeuraMorphErr);

}

if (NMUnitOutputs(unit) != unit->outputs) {

  NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
  sprintf(
    NeuraMorphErr->_msg,
    "NMUnitOutputs failed");
  PBErrCatch(NeuraMorphErr);

}

if (NMUnitGetNbInputs(unit) != 3) {

  NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
  sprintf(
    NeuraMorphErr->_msg,
    "NMUnitGetNbInputs failed");
  PBErrCatch(NeuraMorphErr);

}

if (NMUnitGetNbOutputs(unit) != 2) {

  NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
  sprintf(
    NeuraMorphErr->_msg,
    "NMUnitGetNbOutputs failed");
  PBErrCatch(NeuraMorphErr);

}

bool isSame =
  ISEQUALF(
    NMUnitGetValue(unit),
    0.0);
if (isSame != true) {

  NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
  sprintf(
    NeuraMorphErr->_msg,
    "NMUnitGetValue failed");
  PBErrCatch(NeuraMorphErr);

}

NMUnitSetValue(
  unit,
  0.5);
isSame =
  ISEQUALF(
    NMUnitGetValue(unit),
    0.5);
if (isSame != true) {

  NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
  sprintf(
    NeuraMorphErr->_msg,
    "NMUnitSetValue failed");
```

```
    PBErrCatch(NeuraMorphErr);

  }

  NMUnitPrintln(
    unit,
    stdout);

  NeuraMorphUnitFree(&unit);
  VecFree(&iIn);
  VecFree(&iOut);
  printf("UnitTestNeuraMorphUnitGetSetPrint OK\n");

}

void UnitTestNeuraMorphUnitEvaluate() {

  VecLong* iIn = VecLongCreate(3);
  VecLong* iOut = VecLongCreate(2);
  NeuraMorphUnit* unit =
    NeuraMorphUnitCreate(
      iIn,
      iOut);

  for (
    long iInput = 3;
    iInput--;) {

    VecSet(
      unit->lowFilters,
      iInput + 1,
      0.0);
    VecSet(
      unit->highFilters,
      iInput + 1,
      2.0);

  }

  // iOutput == 0 -> 1.0+x+y+z+x^2+xy+xz+y^2+yz+z^2
  // iOutput == 1 -> x^2-xy+2xz+3y^2-4yz+5z^2
  float coeffs[2][10] = {

    { 1.0, 1.0,  1.0, 1.0,  1.0,  1.0, 1.0,  1.0,  1.0, 1.0},
    { 0.0, 0.0,  1.0, 0.0, -1.0,  3.0, 0.0,  2.0, -4.0, 5.0}

  };
  for (
    long iOutput = 2;
    iOutput--;) {

    for (
      long iCoeff = 10;
      iCoeff--;) {

      VecSet(
        unit->coeffs[iOutput],
        iCoeff,
        coeffs[iOutput][iCoeff]);

    }
```

```
}

VecFloat* inputs = VecFloatCreate(3);
VecSet(
  inputs,
  0,
  1.0);
VecSet(
  inputs,
  1,
  3.0);
VecSet(
  inputs,
  2,
  1.5);

NMUnitEvaluate(
  unit,
  inputs);

float check[2];
float x = 1.0;
float y = 0.0;
float z = 1.5;
check[0] = 1.0 + x + y + z + x * x + x * y + x * z + y * y + y * z + z * z;
check[1] =
  x * x - x * y + 2.0 * x * z + 3.0 * y * y - 4.0 * y * z + 5.0 * z * z;
VecFloat2D checkHigh = VecFloatCreateStatic2D();
VecSet(
  &checkHigh,
  0,
  check[0]);
VecSet(
  &checkHigh,
  1,
  check[1]);
VecFloat2D checkLow = checkHigh;
for (
  long iOutput = 2;
  iOutput--;) {

  float v =
    VecGet(
      unit->outputs,
      iOutput);
  bool same =
    ISEQUALF(
      v,
      check[iOutput]);
  if (same == false) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
      NeuraMorphErr->_msg,
      "NMUnitEvaluate failed (1)");
    PBErrCatch(NeuraMorphErr);

  }

}

bool sameLow =
```

```
    VecIsEqual(
      &checkLow,
      unit->lowOutputs);
  bool sameHigh =
    VecIsEqual(
      &checkHigh,
      unit->highOutputs);
  if (
    sameLow == false ||
    sameHigh == false) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
      NeuraMorphErr->_msg,
      "NMUnitEvaluate failed (2)");
    PBErrCatch(NeuraMorphErr);

  }

  NeuraMorphUnitFree(&unit);
  VecFree(&iIn);
  VecFree(&iOut);
  VecFree(&inputs);
  printf("UnitTestNeuraMorphUnitEvaluate OK\n");

}

void UnitTestNeuraMorphUnit() {

  UnitTestNeuraMorphUnitCreateFree();
  UnitTestNeuraMorphUnitGetSetPrint();
  UnitTestNeuraMorphUnitEvaluate();
  printf("UnitTestNeuraMorphUnit OK\n");

}

void UnitTestNeuraMorphCreateFree() {

  NeuraMorph* nm =
    NeuraMorphCreate(
      3,
      2);
  if (
    nm->nbInput != 3 ||
    nm->nbOutput != 2 ||
    VecGetDim(nm->inputs) != 3 ||
    VecGetDim(nm->outputs) != 2 ||
    nm->hiddens != NULL ||
    GSetNbElem(&(nm->units)) != 0) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
      NeuraMorphErr->_msg,
      "NeuraMorphCreate failed");
    PBErrCatch(NeuraMorphErr);

  }

  NeuraMorphFree(&nm);
  if (nm != NULL) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
```

```
    sprintf(
      NeuraMorphErr->_msg,
      "NeuraMorphFree failed");
    PBErrCatch(NeuraMorphErr);

  }

  printf("UnitTestNeuraMorphCreateFree OK\n");

}

void UnitTestNeuraMorphGetSet() {

  NeuraMorph* nm =
    NeuraMorphCreate(
      3,
      2);
  if (NMGetNbInput(nm) != 3) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
      NeuraMorphErr->_msg,
      "NMGetNbInput failed");
    PBErrCatch(NeuraMorphErr);

  }

  if (NMGetNbOutput(nm) != 2) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
      NeuraMorphErr->_msg,
      "NMGetNbOutput failed");
    PBErrCatch(NeuraMorphErr);

  }

  if (NMGetNbHidden(nm) != 0) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
      NeuraMorphErr->_msg,
      "NMGetNbHidden failed");
    PBErrCatch(NeuraMorphErr);

  }

  NMSetNbHidden(
    nm,
    5);
  if (NMGetNbHidden(nm) != 5) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
      NeuraMorphErr->_msg,
      "NMSetNbHidden failed");
    PBErrCatch(NeuraMorphErr);

  }

  VecLong* iOuts = NMGetVecIOutputs(nm);
  VecLong2D checkOuts =
```

```
    VecLongCreateStatic2D();
  VecSet(
    &checkOuts,
    0,
    5);
  VecSet(
    &checkOuts,
    1,
    6);
  bool isSame =
    VecIsEqual(
      &checkOuts,
      iOuts);
  if (isSame == false) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
      NeuraMorphErr->_msg,
      "NMGetVecIOutputs failed");
    PBErrCatch(NeuraMorphErr);

  }

  VecFree(&iOuts);

  if (NMInputs(nm) != nm->inputs) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
      NeuraMorphErr->_msg,
      "NMInputs failed");
    PBErrCatch(NeuraMorphErr);

  }

  if (NMOutputs(nm) != nm->outputs) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
      NeuraMorphErr->_msg,
      "NMOutputs failed");
    PBErrCatch(NeuraMorphErr);

  }

  NeuraMorphFree(&nm);

  printf("UnitTestNeuraMorphGetSet OK\n");

}

void UnitTestNeuraMorphAddRemoveUnit() {

  VecLong3D iInputs = VecLongCreateStatic3D();
  VecSet(
    &iInputs,
    0,
    0);
  VecSet(
    &iInputs,
    1,
    1);
```

```
VecSet(
  &iInputs,
  2,
  2);
VecLong2D iOutputs = VecLongCreateStatic2D();
VecSet(
  &iOutputs,
  0,
  0);
VecSet(
  &iOutputs,
  1,
  1);

NeuraMorph* nm =
  NeuraMorphCreate(
    3,
    2);

NeuraMorphUnit* unit =
  NMAddUnit(
    nm,
    (VecLong*)&iInputs,
    (VecLong*)&iOutputs);

bool isSameA =
  VecIsEqual(
    &iInputs,
    unit->iInputs);
bool isSameB =
  VecIsEqual(
    &iOutputs,
    unit->iOutputs);
if (
  GSetNbElem(&(nm->units)) != 1 ||
  GSetHead(&(nm->units)) != unit ||
  isSameA == false ||
  isSameB == false) {

  NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
  sprintf(
    NeuraMorphErr->_msg,
    "NMAddUnit failed");
  PBErrCatch(NeuraMorphErr);

}

NeuraMorphFree(&nm);

nm =
  NeuraMorphCreate(
    3,
    2);

unit =
  NMAddUnit(
    nm,
    (VecLong*)&iInputs,
    (VecLong*)&iOutputs);

NMRemoveUnit(
  nm,
```

```
    unit);

  if (GSetNbElem(&(nm->units)) != 0) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
      NeuraMorphErr->_msg,
      "NMRemoveUnit failed");
    PBErrCatch(NeuraMorphErr);

  }

  NeuraMorphUnitFree(&unit);
  NeuraMorphFree(&nm);

  printf("UnitTestNeuraMorphAddRemoveUnit OK\n");

}

void UnitTestNeuraMorphBurryUnitsEvaluate() {

  VecLong3D iInputs = VecLongCreateStatic3D();
  VecSet(
    &iInputs,
    0,
    0);
  VecSet(
    &iInputs,
    1,
    1);
  VecSet(
    &iInputs,
    2,
    2);
  VecLong2D iOutputs = VecLongCreateStatic2D();
  VecSet(
    &iOutputs,
    0,
    0);
  VecSet(
    &iOutputs,
    1,
    1);

  NeuraMorph* nm =
    NeuraMorphCreate(
      3,
      2);

  NeuraMorphUnit* unitA =
    NeuraMorphUnitCreate(
      (VecLong*)&iInputs,
      (VecLong*)&iOutputs);

  NeuraMorphUnit* unitB =
    NeuraMorphUnitCreate(
      (VecLong*)&iInputs,
      (VecLong*)&iOutputs);

  for (
    long iInput = 3;
    iInput--;) {
```

```
  VecSet(
    unitA->lowFilters,
    iInput + 1,
    0.0);
  VecSet(
    unitA->highFilters,
    iInput + 1,
    2.0);
  VecSet(
    unitB->lowFilters,
    iInput + 1,
    0.0);
  VecSet(
    unitB->highFilters,
    iInput + 1,
    2.0);

}

float coeffsA[2][10] = {

  { 1.0, 1.0,  1.0, 1.0,  1.0,  1.0, 1.0,  1.0,  1.0, 1.0},
  { 0.0, 0.0,  1.0, 0.0, -1.0,  3.0, 0.0,  2.0, -4.0, 5.0}

};
float coeffsB[2][10] = {

  { 0.0, 0.0,  1.0, 0.0, -1.0,  3.0, 0.0,  2.0, -4.0, 5.0},
  { 1.0, 1.0,  1.0, 1.0,  1.0,  1.0, 1.0,  1.0,  1.0, 1.0}

};
for (
  long iOutput = 2;
  iOutput--;) {

  for (
    long iCoeff = 10;
    iCoeff--;) {

    VecSet(
      unitA->coeffs[iOutput],
      iCoeff,
      coeffsA[iOutput][iCoeff]);

    VecSet(
      unitB->coeffs[iOutput],
      iCoeff,
      coeffsB[iOutput][iCoeff]);

  }

}

float x = 1.0;
float y = 0.5;
float z = 1.5;
VecFloat* evalInputs = VecFloatCreate(3);
VecSet(
  evalInputs,
  0,
  x);
```

```
VecSet(
  evalInputs,
  1,
  y);
VecSet(
  evalInputs,
  2,
  z);

NMUnitEvaluate(
  unitA,
  evalInputs);
NMUnitEvaluate(
  unitB,
  evalInputs);

GSet units = GSetCreateStatic();
GSetAppend(
  &units,
  unitA);
GSetAppend(
  &units,
  unitB);

NMBurryUnits(
  nm,
  &units);

if (
  GSetNbElem(&units) != 0 ||
  nm->hiddens == NULL ||
  VecGetDim(nm->hiddens) != 4) {

  NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
  sprintf(
    NeuraMorphErr->_msg,
    "NMBurryUnits failed (1)");
  PBErrCatch(NeuraMorphErr);

}

VecLong2D checkA = VecLongCreateStatic2D();
VecSet(
  &checkA,
  0,
  0);
VecSet(
  &checkA,
  1,
  1);
VecLong2D checkB = VecLongCreateStatic2D();
VecSet(
  &checkB,
  0,
  2);
VecSet(
  &checkB,
  1,
  3);

bool isSameA =
  VecIsEqual(
```

```
    &checkA,
    unitA->iOutputs);
bool isSameB =
  VecIsEqual(
    &checkB,
    unitB->iOutputs);
if (
  isSameA == false ||
  isSameB == false) {

  NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
  sprintf(
    NeuraMorphErr->_msg,
    "NMBurryUnits failed (2)");
  PBErrCatch(NeuraMorphErr);

}

float checkLowAa =
  VecGet(
    unitA->lowOutputs,
    0);
checkLowAa -=
  VecGet(
    nm->lowHiddens,
    0);
bool isSameLowAa =
  ISEQUALF(
    checkLowAa,
    0.0);
float checkLowAb =
  VecGet(
    unitA->lowOutputs,
    1);
checkLowAb -=
  VecGet(
    nm->lowHiddens,
    1);
bool isSameLowAb =
  ISEQUALF(
    checkLowAb,
    0.0);
float checkLowBa =
  VecGet(
    unitB->lowOutputs,
    0);
checkLowBa -=
  VecGet(
    nm->lowHiddens,
    2);
bool isSameLowBa =
  ISEQUALF(
    checkLowBa,
    0.0);
float checkLowBb =
  VecGet(
    unitB->lowOutputs,
    1);
checkLowBb -=
  VecGet(
    nm->lowHiddens,
    3);
```

```
bool isSameLowBb =
  ISEQUALF(
    checkLowBb,
    0.0);
float checkHighAa =
  VecGet(
    unitA->lowOutputs,
    0);
checkHighAa -=
  VecGet(
    nm->lowHiddens,
    0);
bool isSameHighAa =
  ISEQUALF(
    checkHighAa,
    0.0);
float checkHighAb =
  VecGet(
    unitA->lowOutputs,
    1);
checkHighAb -=
  VecGet(
    nm->lowHiddens,
    1);
bool isSameHighAb =
  ISEQUALF(
    checkHighAb,
    0.0);
float checkHighBa =
  VecGet(
    unitB->lowOutputs,
    0);
checkHighBa -=
  VecGet(
    nm->lowHiddens,
    2);
bool isSameHighBa =
  ISEQUALF(
    checkHighBa,
    0.0);
float checkHighBb =
  VecGet(
    unitB->lowOutputs,
    1);
checkHighBb -=
  VecGet(
    nm->lowHiddens,
    3);
bool isSameHighBb =
  ISEQUALF(
    checkHighBb,
    0.0);
if (
  isSameLowAa == false ||
  isSameLowAb == false ||
  isSameLowBa == false ||
  isSameLowBb == false ||
  isSameHighAa == false ||
  isSameHighAb == false ||
  isSameHighBa == false ||
  isSameHighBb == false) {
```

```
  NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
  sprintf(
    NeuraMorphErr->_msg,
    "NMBurryUnits failed (3)");
  PBErrCatch(NeuraMorphErr);

}

VecSet(
  &iInputs,
  0,
  3);
VecSet(
  &iInputs,
  1,
  4);
VecSet(
  &iInputs,
  2,
  5);
VecSet(
  &iOutputs,
  0,
  4);
VecSet(
  &iOutputs,
  1,
  5);
NeuraMorphUnit* unitC =
  NMAddUnit(
    nm,
    (VecLong*)&iInputs,
    (VecLong*)&iOutputs);

for (
  long iInput = 3;
  iInput--;) {

  VecSet(
    unitC->lowFilters,
    iInput + 1,
    0.0);
  VecSet(
    unitC->highFilters,
    iInput + 1,
    20.0);

}

float coeffsC[2][10] = {

  { 1.0, 1.0,  1.0, 1.0,  1.0,  1.0, 1.0,  1.0,  1.0, 1.0},
  { 0.0, 0.0,  1.0, 0.0, -1.0,  3.0, 0.0,  2.0, -4.0, 5.0}

};
for (
  long iOutput = 2;
  iOutput--;) {

  for (
    long iCoeff = 10;
    iCoeff--;) {
```

```
    VecSet(
      unitC->coeffs[iOutput],
      iCoeff,
      coeffsC[iOutput][iCoeff]);

  }

}

NMEvaluate(
  nm,
  evalInputs);

float checkAout[2];
checkAout[0] =
  1.0 + x + y + z + x * x + x * y + x * z + y * y + y * z + z * z -
  VecGet(
    nm->hiddens,
    0);
checkAout[1] =
  x * x - x * y + 2.0 * x * z + 3.0 * y * y - 4.0 * y * z + 5.0 * z * z -
  VecGet(
    nm->hiddens,
    1);
float checkBout[2];
checkBout[0] =
  x * x - x * y + 2.0 * x * z + 3.0 * y * y - 4.0 * y * z + 5.0 * z * z -
  VecGet(
    nm->hiddens,
    2);
checkBout[1] =
  1.0 + x + y + z + x * x + x * y + x * z + y * y + y * z + z * z -
  VecGet(
    nm->hiddens,
    3);

bool isSameAa =
  ISEQUALF(
    checkAout[0],
    0.0);
bool isSameAb =
  ISEQUALF(
    checkAout[1],
    0.0);
bool isSameBa =
  ISEQUALF(
    checkBout[0],
    0.0);
bool isSameBb =
  ISEQUALF(
    checkBout[1],
    0.0);
if (
  isSameAa == false ||
  isSameAb == false ||
  isSameBa == false ||
  isSameBb == false) {

  NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
  sprintf(
    NeuraMorphErr->_msg,
```

```
      "NMEvaluate failed (1)");
    PBErrCatch(NeuraMorphErr);

  }

  x =
    VecGet(
      nm->hiddens,
      0);
  y =
    VecGet(
      nm->hiddens,
      1);
  z =
    VecGet(
      nm->hiddens,
      2);
  float checkCout[2];
  checkCout[0] =
    1.0 + x + y + z + x * x + x * y + x * z + y * y + y * z + z * z -
    VecGet(
      unitC->outputs,
      0);
  checkCout[1] =
    x * x - x * y + 2.0 * x * z + 3.0 * y * y - 4.0 * y * z + 5.0 * z * z -
    VecGet(
      unitC->outputs,
      1);

  bool isSameCa =
    ISEQUALF(
      checkCout[0],
      0.0);
  bool isSameCb =
    ISEQUALF(
      checkCout[1],
      0.0);
  bool isSameCc =
    VecIsEqual(
      unitC->outputs,
      nm->outputs);
  if (
    isSameCa == false ||
    isSameCb == false ||
    isSameCc == false) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
      NeuraMorphErr->_msg,
      "NMEvaluate failed (2)");
    PBErrCatch(NeuraMorphErr);

  }

  VecFree(&evalInputs);
  NeuraMorphFree(&nm);

  printf("UnitTestNeuraMorphBurryUnitsEvaluate OK\n");

}

void UnitTestNeuraMorph() {
```

```
  UnitTestNeuraMorphCreateFree();
  UnitTestNeuraMorphGetSet();
  UnitTestNeuraMorphAddRemoveUnit();
  UnitTestNeuraMorphBurryUnitsEvaluate();
  printf("UnitTestNeuraMorph OK\n");

}

void UnitTestNeuraMorphTrainerCreateFree() {

  GDataSetVecFloat dataset =
    GDataSetVecFloatCreateStaticFromFile("./Datasets/iris.json");
  NeuraMorph* nm =
    NeuraMorphCreate(
      GDSGetNbInputs(&dataset),
      GDSGetNbOutputs(&dataset));
  NeuraMorphTrainer trainer =
    NeuraMorphTrainerCreateStatic(
      nm,
      &dataset);
  bool isSame =
    ISEQUALF(
      trainer.weakUnitThreshold,
      0.9);
  if (
    trainer.neuraMorph != nm ||
    trainer.depth != 2 ||
    isSame != true ||
    trainer.iCatTraining != 0 ||
    trainer.dataset != &dataset) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
      NeuraMorphErr->_msg,
      "NeuraMorphTrainerCreateStatic failed");
    PBErrCatch(NeuraMorphErr);

  }

  NeuraMorphTrainerFreeStatic(&trainer);
  NeuraMorphFree(&nm);
  GDataSetVecFloatFreeStatic(&dataset);

  printf("UnitTestNeuraMorphTrainerCreateFree OK\n");

}

void UnitTestNeuraMorphTrainerGetSet() {

  GDataSetVecFloat dataset =
    GDataSetVecFloatCreateStaticFromFile("./Datasets/iris.json");
  NeuraMorph* nm =
    NeuraMorphCreate(
      GDSGetNbInputs(&dataset),
      GDSGetNbOutputs(&dataset));
  NeuraMorphTrainer trainer =
    NeuraMorphTrainerCreateStatic(
      nm,
      &dataset);
  if (NMTrainerGetDepth(&trainer) != 2) {
```

```
  NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
  sprintf(
    NeuraMorphErr->_msg,
    "NeuraMorphTrainerGetDepth failed");
  PBErrCatch(NeuraMorphErr);

}

if (NMTrainerGetICatTraining(&trainer) != 0) {

  NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
  sprintf(
    NeuraMorphErr->_msg,
    "NeuraMorphTrainerGetICatTraining failed");
  PBErrCatch(NeuraMorphErr);

}

bool isSame =
  ISEQUALF(
    NMTrainerGetWeakThreshold(&trainer),
    0.9);
if (isSame != true) {

  NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
  sprintf(
    NeuraMorphErr->_msg,
    "NeuraMorphTrainerGetWeakThreshold failed");
  PBErrCatch(NeuraMorphErr);

}

NMTrainerSetDepth(
  &trainer,
  3);
if (NMTrainerGetDepth(&trainer) != 3) {

  NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
  sprintf(
    NeuraMorphErr->_msg,
    "NeuraMorphTrainerSetDepth failed");
  PBErrCatch(NeuraMorphErr);

}

NMTrainerSetICatTraining(
  &trainer,
  3);
if (NMTrainerGetICatTraining(&trainer) != 3) {

  NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
  sprintf(
    NeuraMorphErr->_msg,
    "NeuraMorphTrainerSetICatTraining failed");
  PBErrCatch(NeuraMorphErr);

}

NMTrainerSetWeakThreshold(
  &trainer,
  0.5);
isSame =
```

```
    ISEQUALF(
      NMTrainerGetWeakThreshold(&trainer),
      0.5);
  if (isSame != true) {

    NeuraMorphErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
      NeuraMorphErr->_msg,
      "NeuraMorphTrainerSetWeakThreshold failed");
    PBErrCatch(NeuraMorphErr);

  }

  NeuraMorphTrainerFreeStatic(&trainer);
  NeuraMorphFree(&nm);
  GDataSetVecFloatFreeStatic(&dataset);

  printf("UnitTestNeuraMorphTrainerGetSet OK\n");

}

void UnitTestNeuraMorphTrainerRun() {

  GDataSetVecFloat dataset =
    GDataSetVecFloatCreateStaticFromFile("./Datasets/iris.json");
  NeuraMorph* nm =
    NeuraMorphCreate(
      GDSGetNbInputs(&dataset),
      GDSGetNbOutputs(&dataset));
  NeuraMorphTrainer trainer =
    NeuraMorphTrainerCreateStatic(
      nm,
      &dataset);

  NMTrainerRun(&trainer);

  NeuraMorphTrainerFreeStatic(&trainer);
  NeuraMorphFree(&nm);
  GDataSetVecFloatFreeStatic(&dataset);

  printf("UnitTestNeuraMorphTrainerRun OK\n");

}

void UnitTestNeuraMorphTrainer() {

  UnitTestNeuraMorphTrainerCreateFree();
  UnitTestNeuraMorphTrainerGetSet();
  UnitTestNeuraMorphTrainerRun();
  printf("UnitTestNeuraMorphTrainer OK\n");

}

void UnitTestAll() {

  UnitTestNeuraMorphUnit();
  UnitTestNeuraMorph();
  UnitTestNeuraMorphTrainer();
  printf("UnitTestAll OK\n");

}
```

```
int main() {

  UnitTestAll();

  // Return success code
  return 0;

}
```

# 6 Unit tests output

```
UnitTestNeuraMorphUnitCreateFree OK
<0,0,0> -> <0,0> (0.500000)
UnitTestNeuraMorphUnitGetSetPrint OK
UnitTestNeuraMorphUnitEvaluate OK
UnitTestNeuraMorphUnit OK
UnitTestNeuraMorphCreateFree OK
UnitTestNeuraMorphGetSet OK
UnitTestNeuraMorphAddRemoveUnit OK
UnitTestNeuraMorphBurryUnitsEvaluate OK
UnitTestNeuraMorph OK
UnitTestNeuraMorphTrainerCreateFree OK
UnitTestNeuraMorphTrainerGetSet OK
Depth 1/2...
Nb available inputs: 4
Train units with 0001 inputs
Train units with 0002 inputs
Train units with 0003 inputs
Train units with 0004 inputs
Burry 2 out of 15 units
<0,1> -> <0,1,2> (1957747840.000000)
<0,1,2,3> -> <0,1,2> (2044897792.000000)
Depth 2/2...
Nb available inputs: 10
Train units with 0001 inputs
Train units with 0002 inputs
Train units with 0003 inputs
Train units with 0004 inputs
Train units with 0005 inputs
Train units with 0006 inputs
Train units with 0007 inputs
Train units with 0008 inputs
Train units with 0009 inputs
Train units with 0010 inputs
Add the last unit
<4,7,9> -> <6,7,8> (2147469824.000000)
UnitTestNeuraMorphTrainerRun OK
UnitTestNeuraMorphTrainer OK
UnitTestAll OK
```