NeuralOps

Artificial Neural Networks in C++
The Manual

First Edition

By

Magnus Erik Hvass Pedersen

May 2008

Copyright © 2008, all rights reserved by the author.
Printing & distribution for personal and academic use allowed.
Commercial use requires written consent from the author.
Please see page 4 for license details.

Contents

Contents 2
1. Introduction 4
1.1 Manual Outline 4
1.2 License 4
1.3 Updates 5
2. Installation 6
2.1 Mirror Development Path 6
2.2 New Development Path 6
2.3 Direct Compiling 7
2.4 Requirements 7
3. Feedforward Networks 8
3.1 Dataflow Model 8
3.2 Node Processing 8
3.3 General Approximator 10
4. Tutorial 12
4.1 Error Handling 12
4.2 Header Files 12
4.3 Optimization Settings 14
4.4 ANN Settings 14
4.5 Dataset Loading 15
4.6 ANN Object Initialization 15
4.7 Boundary Vectors 16
4.8 PRNG Seeding 16

4.9 Optimization 17
4.10 Printing Results.. 17
4.11 Classification Error.. 18
4.12 Free Memory 19
Bibliography 20

1. Introduction

NeuralOps is a source-code library implementing a basic form of Artificial Neural Network (ANN) in the C++ programming language; namely a fully connected, sigmoidal, feedforward network. NeuralOps is intended primarily as an example on how to use the SwarmOps source-code library from (1) for Numerical Optimization, Meta-Optimization (or Meta-Optimisation, Meta-Evolution, Super-Optimization, Parameter Calibration, etc.), and Meta-Meta-Optimization of a real-world problem.

1.1 Manual Outline

This manual briefly describes how to use NeuralOps with SwarmOps. The manual is divided into the following chapters:

- Chapter 1 is this introduction.
- Chapter 2 is an installation manual.
- Chapter 3 is a brief description of the type of ANN implemented here.
- Chapter 4 contains a tutorial for using NeuralOps with SwarmOps.

1.2 License

The NeuralOps source-code is published under the GNU Lesser General Public License (2), which essentially means that you may distribute commercial programs that link with the NeuralOps library, as well as make alterations to the NeuralOps library itself. There are certain terms to be met though, but for those details please see the license included in the source-code distribution.

This manual may be downloaded, printed, and used for any personal purpose, be it commercial or non-commercial, provided the author(s) are not held responsible for

your actions, or any damage caused by your use of the manual. If you want to distribute the manual commercially, for example in a printed book, or on a web-page that requires payment, then you must obtain a license from the author(s).

1.3 Updates

To obtain updates to the NeuralOps source-code library or to get newer revisions of this manual, go to the library's webpage at: http://www.Hvass-Labs.org/

2. Installation

This chapter describes the various ways of installing and using the NeuralOps source-code library. The chapter assumes you have already downloaded the latest source-code archive through the internet website: http://www.Hvass-Labs.org/
After you have got the source-code to compile see the tutorial in chapter 4 on how to include and use NeuralOps in your own program.

2.1 Mirror Development Path

The easiest way to install and use NeuralOps for Microsoft Visual C++ is to mirror the directory path of the development computer. All the C++ libraries from Hvass Laboratories are located in the following path on that computer:

C:\Users\Magnus\Documents\Development\Libraries\HvassLabs-CPP

To install the NeuralOps source-code simply un-zip the archive to this path, and you should be able to open and use the MS Visual C++ projects as is. You should do the same for SwarmOps, although to another directory path:

C:\Users\Magnus\Documents\Development\Libraries\HvassLabs-C

2.2 New Development Path

If you do not wish to mirror the directory path of the development computer, but would still like to reuse the MS Visual C++ projects included with the NeuralOps library, then you will have to manually alter all paths in the compilation projects. The compiler should inform you whenever you have an incorrect path that needs to be changed.

2.3 Direct Compiling

Another easy way of using NeuralOps in your own source-code is to add the proper include-path, and then simply include the NeuralOps header-files you require. The current version of NeuralOps does not have any source-files and therefore does not require for you to make a separate NeuralOps link-library.

2.4 Requirements

This section describes the various requirements for NeuralOps to compile and work.

SwarmOps

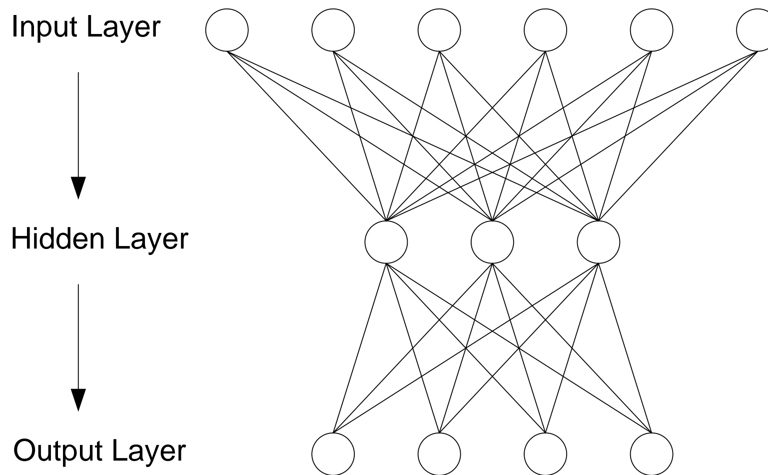
Although it is pretty straight-forward to implement gradient-based optimization for NeuralOps, it is not included in the NeuralOps library itself, as it is recommended you use the optimization methods implemented in the SwarmOps library (1). Be advised however, that SwarmOps has additional requirements, in particular a Pseudo-Random Number Generator (PRNG) which by default is the RandomOps library (3), but may be changed if you wish to use another PRNG library. Please see the SwarmOps manual for details on this.

3. Feedforward Networks

This chapter briefly describes the kind of Artificial Neural Network (ANN) implemented in NeuralOps; namely a fully connected, sigmoidal, feedforward network. Several researchers contributed to the development of this kind of ANN, see (4) for a detailed historical account and a comprehensive textbook on the subject. A good introduction to ANNs may be found in (5).

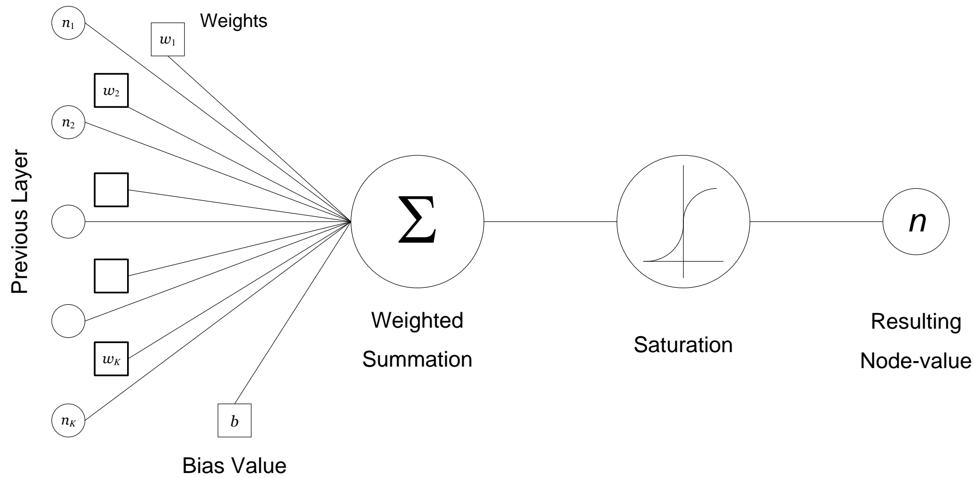
3.1 Dataflow Model

An ANN is a dataflow model where the data flows from an input layer through one or more hidden layers to an output layer, like so:



3.2 Node Processing

The processing that occurs at each node (also called neuron in the literature) of the ANN consists of computing and saturating (or compressing, squashing) a weighted sum of the previous layer's nodes, which can be depicted as follows:



To be more precise, we may denote the value of the j'th node in the i'th layer as n_{ij}, and that node's bias value as b_{ij}. The weight that connects node n_{ij} to the k'th node in the previous layer is denoted w_{ijk}. The weighted sum of the previous layer's nodes and the bias value may be denoted s_{ij} and computed as follows:

$$s_{ij} = b_{ij} + \sum_{k=1}^{N_{i-1}} w_{ijk} \cdot n_{i-1,k}$$

Where N_{i-1} is the number of nodes in the previous layer. This sum must then be saturated using the Sigmoid function $\sigma: \mathbb{R} \to (0,1)$ defined as follows:

$$\sigma(s) = \frac{1}{1 + e^{-s}}$$

And finally we may compute the actual node-value as:

$$n_{ij} = \sigma(s_{ij})$$

Computing the node-values for the entire ANN thus consists of iterating over the layer indices i and the node indices j, and computing the node values n_{ij} for all such combinations. NeuralOps gives you the choice of whether or not to use saturation for the output layer, thus allowing for arbitrary ranges of output values.

3.3 General Approximator

It has been shown in the literature that this kind of ANN is a general approximator of input-output mappings under certain conditions, see e.g. (6). The ANN weights and bias values must therefore be trained (that is, optimized) so as to make the ANN mimic the input-output mapping of some dataset, thus enabling the ANN to produce sensible output from previously unseen input.

Dataset

Two kinds of datasets are supported in NeuralOps:

1. Datasets from the Proben1 library (7).
2. The Facial Direction classification problem from (5).

It should be noted that NeuralOps currently does not support the splitting of datasets into training-, test-, and validation-sets, because NeuralOps was created to compare optimization performance and not ANN generalization performance.

Sum of Squared Error (SSE)

To compute the fitness value for use in optimization, that relates the quality of an ANN approximation between the input-output mappings of a dataset, we first need to compute the approximation error of a single input-output mapping. Let $\vec{o} = [o_1, \cdots, o_{N_M}]$ denote the actual output of the ANN and let $\vec{t}_o$ denote the desired output from the dataset (which corresponds to an input denoted $\vec{t}_i$). The Sum of Squared Error (SSE) between desired and actual ANN output is then defined as:

$$\mathrm{SSE}(\vec{o}, \vec{t}_o) = \sum_{j=1}^{N_M} \left(o_j - t_{oj}\right)^2$$

Where N_M is the number of nodes in the ANN output layer.

Mean Squared Error (MSE)

Iterating over all the input-output pairs $(\vec{t}_i, \vec{t}_o)$ from the dataset T and computing the ANN output $\vec{o}$ from the dataset input $\vec{t}_i$, and then summing and normalizing the SSE measures for these, gives the Mean Squared Error (MSE) measure:

$$\mathrm{MSE}(T) = \frac{1}{N_M \cdot |T|} \cdot \sum_{(\vec{t}_i, \vec{t}_o) \in T} \mathrm{SSE}(\vec{o}, \vec{t}_o)$$

This is the fitness measure to be minimized to make the ANN mimic the input-output mapping of a given dataset. Note that the normalization factor includes both the size of the dataset $|T|$ and the number of output-nodes N_M, which makes for a more uniform MSE measure across different datasets, than the MSE measure that is typically used in the literature and which only normalizes using $|T|$.

Classification Error (CLS)

Once the ANN weights and bias values have been optimized with regard to the MSE measure so as to mimic an input-output mapping of a dataset, the ANN can also be used for making classifications. The kind of classification supported in NeuralOps is known as 1-of-m classification, because the index of the output-node with the highest value is taken to be the classification. Mathematical results suggest that an ANN trained using the MSE measure will also work with regard to this kind of classification (8). The Classification Error (CLS) is hence taken to be the total number of incorrect classifications (that is, where the 1-of-m indices mismatch), divided by the total number of input-output pairs in the dataset. It is important to relate the classification error and not the correctness, because it is easier for a human to compare and assess the error measures.

4. Tutorial

This chapter is a tutorial for getting started with NeuralOps. It describes how to perform basic optimization of ANN weights using optimization methods from the SwarmOps library (1), and is loosely based on the source-file ANN.cpp supplied with NeuralOps. It is assumed you have already installed the NeuralOps and SwarmOps source-code libraries, as described in chapter 2. Tutorials for doing Meta-Optimization and Meta-Meta-Optimization are not included in this manual, and the reader is instead referred directly to the source-code files MetaANN.cpp and Meta2ANN.cpp on how to do Meta- and Meta-Meta-Optimization, respectively.

4.1 Error Handling

Before giving the actual tutorial on how to use NeuralOps, it should be noted that the error handling in NeuralOps is done primarily through use of the assert() function. The advantage of assertions is that they do not compile into the release-build of the library, but only exist in the debug-version where they trigger exceptions if the assertions are not met. Since the use of NeuralOps can normally be tested quite thoroughly in debug-mode during development, this approach is considered the best in terms of runtime efficiency and ease of development. But it means you should always test your new program in debug-mode before executing it in release-mode.

4.2 Header Files

There are a number of header files you must include to use NeuralOps.

NeuralOps Header Files

The NeuralOps header-files that must be included are:

```
#include <NeuralOps/Network/FeedForward.h>
#include <NeuralOps/Dataset/DatasetProben.h>
#include <NeuralOps/Scoring/CLS.h>
#include <NeuralOps/Optimize/SwarmOpsWrapper.h>
#include <NeuralOps/Optimize/Context.h>
```

In order of appearance these are: FeedForward.h implements the actual ANN whose weights and bias values must be optimized, DatasetProben.h supports datasets from the Proben1 library (7), CLS.h computes the CLS classification score, and SwarmOpsWrapper.h with Context.h enable support for NeuralOps with SwarmOps.

SwarmOps Header Files

The SwarmOps header-files that must be included are:

```
#include <SwarmOps/Optimize.h>
#include <SwarmOps/Methods/Methods.h>
#include <SwarmOps/Tools/Vector.h>
```

In order of appearance these are: Optimize.h supplies a function for making optimization using the SwarmOps framework easy, Methods.h holds a list of the optimization methods that are available in SwarmOps, and Vector.h provides functions for printing the results of optimization.

RandomOps Header File

Assuming you are using RandomOps as the PRNG for the SwarmOps library, the RandomOps header-file that must be included is:

```
#include <RandomOps/Random.h>
```

System Header-Files

Various header-files must be included to provide system-calls:

```
#include <limits>
#include <cstdio>
#include <cstdlib>
```

4.3 Optimization Settings

By optimization settings are meant the choice of optimization method, the number of optimization runs, and the number of iterations per run. These may be defined as a number of constants, as follows:

```
const size_t kMethodId = SO_kMethodGD;
const size_t kNumRuns = 10;
const size_t kDimFactor = 20;
```

These determine the following things, respectively: The optimization method (here taken to be Gradient Descent (GD)), the number of optimization runs to perform (10), and the dimensionality-factor (20) which will be multiplied with the total number of weights and bias values of the given ANN, once the dataset has been loaded (see below).

4.4 ANN Settings

The feedforward ANN in this tutorial will have a single hidden layer; meaning it has 3 layers in total. The hidden layer has 4 nodes, which works well for many dataset. The output layer should have linear nodes and not use sigmoidal compression. These settings for the ANN are defined as a number of constants:

```
const size_t kNumLayers = 3;
const size_t kNumHiddenNodes = 4;
const bool kSigmoidOutput = false;
```

Next define the initialization and search-space boundaries for the ANN weights. Note that the SwarmOps datatype SO_TElm will be used here as the datatype for the nodes of the ANN:

```
typedef SO_TElm T;
const T kLowerInit = -0.05;
const T kUpperInit = 0.05;
const T kLowerBound = -7.0;
const T kUpperBound = 7.0;
```

4.5 Dataset Loading

The dataset object must be created and its contents loaded from file. Here we load the dataset located in the file cancer1.dt, as follows:

```
NeuralOps::DatasetProben<T> dataset;
dataset.Load("cancer1.dt");
```

It is important that the file containing the dataset is located in the execution path of your program.

4.6 ANN Object Initialization

Now that the dataset has been loaded we also know how many input- and output-nodes the ANN must have. We can then create the ANN object, as well as the optimization context to be used by the SwarmOps framework:

```
size_t kNumNodes[kNumLayers] =
        {dataset.GetInputDim(),
        kNumHiddenNodes,
        dataset.GetOutputDim()};
NeuralOps::FeedForward<T> ann(kNumNodes,
        kNumLayers,
        kSigmoidOutput);
NeuralOps::Context<T> context(ann, dataset);
```

4.7 Boundary Vectors

The SwarmOps framework must also be supplied with proper initialization and search-space boundaries. These arrays must first be allocated:

```
SO_TElm *lowerInit = new double[kNumWeights];
SO_TElm *upperInit = new double[kNumWeights];
SO_TElm *lowerBound = new double[kNumWeights];
SO_TElm *upperBound = new double[kNumWeights];
```

And then initialize these with the constants previously defined:

```
for (size_t i=0; i<kNumWeights; i++)
{
        lowerInit[i] = kLowerInit;
        upperInit[i] = kUpperInit;
        lowerBound[i] = kLowerBound;
        upperBound[i] = kUpperBound;
}
```

4.8 PRNG Seeding

The PRNG must be seeded before stochastic optimization can begin. By default SwarmOps uses the PRNG from the RandomOps library, which is seeded like this:

```
RO_RandSeedClock(9385839);
```

4.9 Optimization

The number of optimization iterations to perform depends on the number of ANN weights (which in turn depends on the given dataset), and after having determined these the actual optimization can be performed, by calling a SwarmOps function with the appropriate parameters:

```
const size_t kNumWeights = ann.NumWeights();
const size_t kNumIterations = kNumWeights*kDimFactor;
SO_Results res = SO_Optimize(
        kMethodId, kNumRuns, kNumIterations, 0,
        SO_ANNFitness, SO_ANNGradient, (void*) &context,
        kNumWeights,
        lowerInit, upperInit, lowerBound, upperBound,
        0);
```

The last zero can instead be a char-pointer (that is, a textual string) to a filename the fitness-trace will be dumped to. Please see the SwarmOps manual (1) for further details on how to customize its use.

4.10 Printing Results

After performing optimization of the ANN weights, the results can be printed:

```
printf("Average MSE: %g (%g)\n",
        res.stat.fitnessAvg,
        res.stat.fitnessStdDev);
printf("Best MSE: %g\n", res.best.fitness);
```

We can also print the best found ANN weights:

```
SO_PrintVector(res.best.x, kNumWeights);
printf("\n");
```

4.11 Classification Error

Since the cancer1.dt dataset is actually a classification problem, we can compute the CLS measure from the weights obtained through optimization of the MSE measure. First we define an array to hold the CLS values for all the optimization runs, as well as a variable for keeping track of the best CLS value:

```
SO_TFitness cls[kNumRuns];
SO_TFitness bestCLS = SO_kFitnessMax;
```

Where the CLS values are computed by a NeuralOps function:

```
for (size_t i=0; i<kNumRuns; i++)
{
	cls[i] = NeuralOps::CLS(ann, dataset, res.results[i]);
	if (cls[i]<bestCLS) { bestCLS = cls[i]; }
}
```

Then the average CLS and its standard deviation can be printed along with the best-found CLS value:

```
printf("Average CLS: %g (%g)\n",
	SO_Average(cls, kNumRuns),
	SO_StdDeviation(cls, kNumRuns));
printf("Best CLS: %g\n", bestCLS);
```

Note the convenient use of SwarmOps functions for computing the average and standard deviation.

4.12 Free Memory

Finally the memory that has been allocated must also be freed. First the memory for the initialization and boundary vectors is freed:

```
delete [] lowerInit;
delete [] upperInit;
delete [] lowerBound;
delete [] upperBound;
```

And then the memory for the optimization results must be freed by calling the appropriate SwarmOps function:

```
SO_FreeResults(&res);
```

Bibliography

1. **Pedersen, M.E.H.** *SwarmOps - Black-Box Optimization in ANSI C, URL http://www.Hvass-Labs.org/.* s.l. : Hvass Laboratories, 2008.

2. **Free Software Foundation.** *GNU Lesser General Public License. URL http://www.gnu.org/copyleft/lesser.html.*

3. **Pedersen, M.E.H.** *RandomOps - Pseudo-Random Number Generator Source-Code Library for ANSI C, URL http://www.Hvass-Labs.org/.* s.l. : Hvass Laboratories, 2008.

4. **Haykin, S.** *Neural Networks: A Comprehensive Foundation.* s.l. : Prentice Hall, 1999.

5. **Mitchell, T.** *Machine Learning.* s.l. : McGraw-Hill, 1997.

6. *Approximation by superpositions of a sigmoidal function.* **Cybenko, G.** s.l. : Mathematics of Control, Signals, and Systems, 1989, Vol. 2, pp. 303-314.

7. **Prechelt, L.** *Proben1 - A set of neural network benchmark problems and benchmarking rules. Technical Report 21/94.* s.l. : Faculty of Informatics, University of Karlsruhe, Germany, 1994.

8. *Neural network classifiers estimate bayesian a-posteriori probabilities.* **Richard, M.D. and Lippmann, R.P.** s.l. : Neural Computation, 1991, Vol. 3, pp. 461-483.