Documentation of C Neural Network Library

Paul O'Brien (paul.obrien@aero.org)

March 27, 2009

Contents

1 Introduction **1**

2 Changes **1**

3 What is a Neural Network? **1**

3.1 Measurement Error Covariance and the Penalty Function . . . 2

3.2 Gradients and Derivatives . . . 2

3.3 Hessian and Error Estimates . . . 3

4 The Library and its Functions **3**

4.1 Representation of a Neural Network . . . 4

4.2 Representation of a Training Data Set . . . 4

4.3 Network and Data Set File IO Functions . . . 5

4.3.1 nnlib_save_net . . . 5

4.3.2 nnlib_load_net . . . 5

4.3.3 nnlib_save_training_set . . . 6

4.3.4 nnlib_load_training_set . . . 6

4.4 Network Training and Evaluation . . . 7

4.4.1 nnlib_fit . . . 7

4.4.2 nnlib_eval . . . 8

5 Matlab Wrappers **9**

6 Sample Program **9**

7 Parallelization **9**

7.1 Parallelization with OpenMP . . . 9

7.2 Parallelization with MPI . . . 9

8 Neural Network Tribal Knowledge **10**

1 Introduction

This document provides the equations for fitting and evaluating a feed-forward perceptron neural network.

2 Changes

- 3/27/2009 Fixed interpretation of `inflag==0` in `nnlib_load_net`
- 10/31/2008 Support for OpenMP added (`make nntrain.ompx`).
- 10/29/2008 MPI features abstracted and tested on two different clusters.
- 10/23/2008 Modified for use with MPI (`make nntrain.mpix`).
- 10/16/2008 Fixed error (X for Z) in nntrain.c (no effect on nnlib.dll)

- 10/2/2008 Added verification information.
- 9/29/2008 Created manual
- 9/24/2008 Added "epsabs" parameter to function calls
- 9/23/2008 Corrected position of $\bar{y}_k$ in definition of h_k, "unit mean" to "zero mean", and missing $\bar{x}_j$ in gradients and Hessians.
- 9/22/2008 Rewritten with regularization penalty, and for multivariate case only.

3 What is a Neural Network?

A feed-forward neural network deterministically maps a vector space with N_x dimensions to one with N_y dimensions $\vec{y} \approx \vec{h}(\vec{x})$.

$$
\begin{aligned}
h_k(\vec{x}_t) &= \bar{y}_k + s_k^{(y)} \left(v_{0k} + \sum_{i=1}^{N_h} v_{ik} g(u_i(\vec{x}_t; \vec{\theta})) \right), & (1) \\
g(u) &= \frac{1}{1+e^{-u}}, & (2) \\
u_i(\vec{x}_t) &= w_{0i} + \sum_{j=1}^{N_x} w_{ji} \frac{x_{tj} - \bar{x}_j}{s_j^{(x)}}, & (3) \\
\vec{\theta} &= (\underline{\underline{w}}, \underline{\underline{v}}, \vec{w}_0, \vec{v}_0). & (4)
\end{aligned}
$$

The subscript t denotes the sample number (up to N_t), for example in a time series, and N_h denotes the number of "hidden nodes", which controls the complexity of the network. The sample means $\bar{x}_j$ and $\bar{y}_k$, and standard deviations, $s_k^{(y)}$ and $s_j^{(x)}$ are taken from the training data. They are used to ensure that the neural network weights operate on variables with zero mean and unit variance; this helps with regularization.

The sizes of the various vectors and matrices is:

$$
\begin{aligned}
\underline{\underline{x}} &\sim N_t \times N_x, & (5) \\
\underline{\underline{y}} &\sim N_t \times N_y, & (6) \\
\vec{\bar{x}} &\sim 1 \times N_x, & (7) \\
\vec{\bar{y}} &\sim 1 \times N_y, & (8) \\
\vec{s}^{(x)} &\sim 1 \times N_x, & (9) \\
\vec{s}^{(y)} &\sim 1 \times N_y, & (10) \\
\vec{v}_0 &\sim 1 \times N_y, & (11) \\
\underline{\underline{v}} &\sim N_h \times N_y, & (12) \\
\vec{w}_0 &\sim 1 \times N_h, & (13) \\
\underline{\underline{w}} &\sim N_x \times N_h, & (14) \\
\vec{\theta} &\sim 1 \times N_\theta = (N_x + N_y + 1) N_h + N_y. & (15)
\end{aligned}
$$

3.1 Measurement Error Covariance and the Penalty Function

In order to compute $\vec{\theta}$, we must define an optimality criterion. We assume the measurement errors on $\vec{y}_t$ have nonzero covariance given by a (possibly) sample-dependent $\underline{\underline{\Sigma}}_t$. The penalty function consists of one term due to the misfit between the neural network and the observations and another term due to the size of the weights. This latter term is a regularization term that (1) limits the over-fitting impact of extra hidden nodes by forcing unused or insignificant weights to zero, and (2) ensures that the Hessian of the penalty function with respect to the weights is not singular, which, in turn, ensures a finite error covariance matrix for the weights and for any output vector produced by the network.

$$
\ell = \frac{1}{2} \sum_{t=1}^{N_t} \sum_{k=1}^{N_y} \sum_{m=1}^{N_y} s_{tkm}^{-1} e_{tk} e_{tm} + \frac{1}{2} \sum_n \theta_n^2, \qquad (16)
$$

$$s^{-1}_{tkm} = \left(\underline{\underline{\Sigma}}_t^{-1}\right)_{km}, \tag{17}$$

$$e_{tk} = h_k(\vec{x}_t;\vec{\theta}) - y_{tk}. \tag{18}$$

In this formulation as $N_t \to \infty$, the penalty for having non-zero weights vanishes in comparison to the penalty for not exactly matching the training data.

The optimization ("training") algorithm requires an initial estimate for $\vec{\theta}$, which is chosen to consist of random numbers between $[-0.5, 0.5]$. It is a good idea to try the optimization from multiple initial guesses to account for the possibility of a non-global solution.

3.2 Gradients and Derivatives

In order to assist the optimization, it is helpful to have the gradient and some derivatives.

$$\frac{\partial \ell}{\partial \theta_{n'}} = \sum_{t=1}^{N_t}\sum_{k=1}^{N_y}\sum_{m=1}^{N_y} s^{-1}_{tkm} e_{tk} \frac{\partial e_{tm}}{\partial \theta_{n'}} + \theta_{n'} = \sum_{t=1}^{N_t}\sum_{k=1}^{N_y}\sum_{m=1}^{N_y} s^{-1}_{tkm} e_{tk} \frac{\partial h_{tm}}{\partial \theta_{n'}} + \theta_{n'}, \tag{19}$$

$$\frac{\partial e_{tk}}{\partial \theta_{n'}} = \frac{\partial h_{tk}}{\partial \theta_{n'}}, \tag{20}$$

$$\frac{\partial h_{tk}}{\partial v_{0k'}} = s_k^{(y)} \delta_{kk'}, \tag{21}$$

$$\frac{\partial h_{tk}}{\partial v_{i'k'}} = s_k^{(y)} g(u_{ti'}) \delta_{kk'}, \tag{22}$$

$$\frac{\partial h_{tk}}{\partial w_{0i'}} = s_k^{(y)} v_{i'k} g'(u_{ti'}), \tag{23}$$

$$\frac{\partial h_{tk}}{\partial w_{j'i'}} = s_k^{(y)} v_{i'k} g'(u_{ti'}) \frac{x_{tj'} - \bar{x}_{j'}}{s_{j'}^{(x)}} = \frac{\partial h_{tk}}{\partial w_{0i'}} \frac{x_{tj'} - \bar{x}_{j'}}{s_{j'}^{(x)}}, \tag{24}$$

$$g'(u) = \frac{dg}{du} = \frac{e^{-u}}{(1+e^{-u})^2} = g^2(u) e^{-u}. \tag{25}$$

3.3 Hessian and Error Estimates

It can also be useful to know the Hessian, especially when computing the error estimates on $\vec{\theta}$ or $\vec{y}$:

$$\frac{\partial^2 \ell}{\partial \theta_{n'} \partial \theta_{n''}} = \sum_{t=1}^{N_t}\sum_{k=1}^{N_y}\sum_{m=1}^{N_y} s^{-1}_{tkm} \left(e_{tk} \frac{\partial^2 e_{tm}}{\partial \theta_{n'} \partial \theta_{n''}} + \frac{\partial e_{tk}}{\partial \theta_{n'}} \frac{\partial e_{tm}}{\partial \theta_{n''}} \right) + \delta_{n'n''} \tag{26}$$

$$= \sum_{t=1}^{N_t}\sum_{k=1}^{N_y}\sum_{m=1}^{N_y} s^{-1}_{tkm} \left(e_{tk} \frac{\partial^2 h_{tm}}{\partial \theta_{n'} \partial \theta_{n''}} + \frac{\partial h_{tk}}{\partial \theta_{n'}} \frac{\partial h_{tm}}{\partial \theta_{n''}} \right) + \delta_{n'n''}, \tag{27}$$

$$\frac{\partial^2 e_{tk}}{\partial \theta_{n'} \partial \theta_{n''}} = \frac{\partial^2 h_{tk}}{\partial \theta_{n'} \partial \theta_{n''}}, \tag{28}$$

$$\frac{\partial^2 h_{tk}}{\partial v_{0k'} \partial \theta_{n''}} = 0 \tag{29}$$

$$\frac{\partial^2 h_{tk}}{\partial v_{i'k'} \partial v_{i''k''}} = 0, \tag{30}$$

$$\frac{\partial^2 h_{tk}}{\partial v_{i'k'} \partial w_{0i''}} = s_k^{(y)} g'(u_{ti'}) \delta_{i'i''} \delta_{kk'}, \tag{31}$$

$$\frac{\partial^2 h_{tk}}{\partial v_{i'k'} \partial w_{j''i''}} = s_k^{(y)} g'(u_{ti'}) \delta_{i'i''} \delta_{kk'} \frac{x_{tj''} - \bar{x}_{j''}}{s_{j''}^{(x)}} = \frac{\partial^2 h_{tk}}{\partial v_{i'k'} \partial w_{0i''}} \frac{x_{tj''} - \bar{x}_{j''}}{s_{j''}^{(x)}}, \tag{32}$$

$$\frac{\partial^2 h_{tk}}{\partial w_{0i'} \partial w_{0i''}} = s_k^{(y)} v_{i'k} g''(u_{ti'}) \delta_{i'i''}, \tag{33}$$

$$\frac{\partial^2 h_{tk}}{\partial w_{0i'} \partial w_{j''i''}} = s_k^{(y)} v_{i'k} g''(u_{ti'}) \delta_{i'i''} \frac{x_{tj''} - \bar{x}_{j''}}{s_{j''}^{(x)}} = \frac{\partial^2 h_{tk}}{\partial w_{0i'} \partial w_{0i''}} \frac{x_{tj''} - \bar{x}_{j''}}{s_{j''}^{(x)}}, \tag{34}$$

$$\frac{\partial^2 h_{tk}}{\partial w_{j'i'}\partial w_{j''i''}} = s_k^{(y)} v_{i'k} g''(u_{ti'}) \frac{x_{tj'} - \bar{x}_{j'}}{s_{j'}^{(x)}} \delta_{i'i''} \frac{x_{tj''} - \bar{x}_{j''}}{s_{j''}^{(x)}} = \frac{\partial^2 h_{tk}}{\partial w_{0i'}\partial w_{j''i''}} \frac{x_{tj''} - \bar{x}_{j''}}{s_{j''}^{(x)}}, \tag{35}$$

$$g''(u) = \frac{d^2 g}{du^2} = e^{-u}(e^{-u} - 1) g^3(u) = (e^{-u} - 1) g(u) g'(u). \tag{36}$$

The error covariance matrix for $\vec{\theta}$ is given by:

$$\operatorname{cov}\vec{\theta} = \left(\frac{\partial^2 \ell}{\partial \theta_{n'} \partial \theta_{n''}}\right)^{-1}. \tag{37}$$

Note that this covariance matrix depends on ℓ evaluated for the best-fit solution $\vec{\theta}$ over the training data set. Once the network is trained, θ and $\operatorname{cov}\vec{\theta}$ are fixed.

The error covariance matrix for $\vec{y}$ estimated at point $\vec{x}$ by the neural network is:

$$\operatorname{cov}\vec{y} = \left(\frac{\partial \vec{h}}{\partial \vec{\theta}}\right)_{\vec{x}}^T \left(\operatorname{cov}\vec{\theta}\right) \left(\frac{\partial \vec{h}}{\partial \vec{\theta}}\right)_{\vec{x}} \tag{38}$$

As usual, the square root of the diagonal elements of $\operatorname{cov}\vec{y}$ give the standard error $(1-\sigma)$ of the outputs. Note that $\operatorname{cov}\vec{y}$ depends on the input point $\vec{x}$ and requires the weights $\vec{\theta}$ and their covariance $\operatorname{cov}\vec{\theta}$.

4 The Library and its Functions

The neural network library is available in several C files with a Makefile. It depends on the GNU Scientific Library (GSL). The library is provided as a dynamic link library (DLL) `.so` or `.dll`. The function prototypes are specified in `nnlib.h`. This manual is meant to supersede any comments given therein as to the structure of data or meaning of flags.

The `nnlib` C routines follow the GSL storage convention for matrices, and extends it to tensors. Matrices are ordered as such: for an $N_t \times N_x$ matrix $\underline{\underline{X}}$, X_{ti} = `X[(t-1)*Nx+i-1]`. That is, rows are bunched together. For an $N_t \times N_x \times N_y$ matrix $\underline{\underline{A}}$, A_{tij} = `A[(t-1)*Nx*Ny + (i-1)*Ny + j-1]`. That is, the matrices are bunched together. Macros to manipulate these indices are provided `subscripts.h`.

Note: Matlab and GSL store matrices with the opposite majority. For example, an $N \times M$ matrix in GSL is an $M \times N$ matrix in Matlab. Similarly, Matlab and `nnlib` store tensors with the opposite majority. For example, an $N \times M \times L$ tensor in `nnlib` is an $L \times M \times N$ tensor in Matlab. The Matlab `nnlib.m` wrapper handles these index permutations.

4.1 Representation of a Neural Network

The neural network is represented in the computer as a set of scalars, vectors, and matrices:

- `cov_flag` scalar, bitmap of flags describing network
  - `cov_flag & 3` $= 0$ no covariance or Hessian of $\vec{\theta}$
  - `cov_flag & 3` $= 1$ covariance of $\vec{\theta}$ stored in `theta_cov`
  - `cov_flag & 3` $= 2$ Hessian of ℓ stored in `theta_cov`
- `Nx` scalar, N_x, dimension of $\vec{x}$.
- `Nh` scalar, N_h, number of hidden nodes.
- `Ny` scalar, N_y, dimension of $\vec{y}$.
- `theta` vector (N_θ), network weights, as in (4).
- `xbar` vector (N_x), $\vec{\bar{x}}$, estimated mean of $\vec{x}$.
- `ybar` vector (N_y), $\vec{\bar{y}}$, estimated mean of $\vec{y}$.
- `sx` vector (N_x), $\vec{s}^{(x)}$, estimated standard deviation of $\vec{x}$.
- `sy` vector (N_y) $\vec{s}^{(y)}$, estimated standard deviation of $\vec{y}$.
- `theta_cov` matrix $(N_\theta \times N_\theta)$, error covariance of network weights, or Hessian of ℓ.

4.2 Representation of a Training Data Set

A training data set is represented in the computer as a set of scalars, matrices, and tensors:

- `Nt` scalar, N_t, number of training samples.
- `Nx` scalar, N_x, dimension of $\vec{x}$.
- `Ny` scalar, N_y, dimension of $\vec{y}$.
- `X` matrix $(N_t \times N_x)$, $\vec{\vec{x}}$ at each training sample.
- `Y` matrix $(N_t \times N_y)$, $\vec{\vec{y}}$ at each training sample.
- `s` error estimate $\underline{\underline{\Sigma}}_t$ on training samples $\vec{y}$. Meaning depends on `sflag`.
- `sflag` scalar, bitmap of flags describing error covariance of $\vec{y}$ training data.

  `sflag & 3` $= 0$ `s` is scalar, same error variance for all y and all samples

  `sflag & 3` $= 1$ `s` is vector N_y, same error variance for all samples

  `sflag & 3` $= 2$ `s` is matrix $N_t \times N_y$, sample-dependent error variance

  `sflag & 3` $= 3$ `s` is tensor $N_t \times N_y \times N_y$, sample-dependent error covariance. For each sample, there is a different $N_y \times N_y$ error covariance matrix for $\vec{y}$, i.e., a full sample-dependent $\underline{\underline{\Sigma}}_t$.

4.3 Network and Data Set File IO Functions

4.3.1 nnlib_save_net

The function `nnlib_save_net` saves a neural network to a binary file. It has only inputs.

```
void nnlib_save_net(const char *filename, const unsigned long int Nx,
                    const unsigned long int Nh, const unsigned long int Ny, const double *theta,
                    const double *xbar,const double *ybar, const double *sx, const double *sy,
                    const unsigned long int cov_flag, const double *theta_cov);
```

- `filename` null-terminated C string giving the NN file name.
- `Nx` scalar, N_x, dimension of $\vec{x}$.
- `Nh` scalar, N_h, number of hidden nodes.
- `Ny` scalar, N_y, dimension of $\vec{y}$.
- `theta` vector (N_θ), network weights, as in (4).
- `xbar` vector (N_x), $\vec{\bar{x}}$, estimated mean of $\vec{x}$.
- `ybar` vector (N_y), $\vec{\bar{y}}$, estimated mean of $\vec{y}$.
- `sx` vector (N_x), $\vec{s}^{(x)}$, estimated standard deviation of $\vec{x}$.
- `sy` vector (N_y) $\vec{s}^{(y)}$, estimated standard deviation of $\vec{y}$.
- `cov_flag` scalar, bitmap of flags describing network

  `cov_flag & 3` $= 0$ don't save `theta_cov`.

  otherwise, save `theta_cov`.
- `theta_cov` matrix $(N_\theta \times N_\theta)$, error covariance of network weights, or Hessian of ℓ. If `NULL`, nothing is written to the file for `theta_cov`, and `cov_flag` is set to zero in the file.

4.3.2 nnlib_load_net

The function `nnlib_load_net` loads a neural network from a binary file. Because the size of the arrays needed by the network aren't known until one starts to read the file, one can call `nnlib_load_net` with `cov_flag` = 99999 to have it only set the scalar size variables (`Nx`, `Nh`, and `Ny`). Using those values, one can allocate the necessary arrays and call the function again to actually load the network.

```
void nnlib_load_net(const char *filename, unsigned long int *Nx,
                    unsigned long int *Nh, unsigned long int *Ny,
                    double *theta,
                    double *xbar,double *ybar, double *sx, double *sy,
                    unsigned long int *cov_flag, double *theta_cov);
```

- `filename` (input) null-terminated C string giving the NN file name.
- `Nx` (output) scalar, N_x, dimension of $\vec{x}$.
- `Nh` (output) scalar, N_h, number of hidden nodes.
- `Ny` (output) scalar, N_y, dimension of $\vec{y}$.
- `theta` (output) vector (N_θ), network weights, as in (4).
- `xbar` (output) vector (N_x), $\bar{\vec{x}}$, estimated mean of $\vec{x}$.
- `ybar` (output) vector (N_y), $\bar{\vec{y}}$, estimated mean of $\vec{y}$.
- `sx` (output) vector (N_x), $\vec{s}^{(x)}$, estimated standard deviation of $\vec{x}$.
- `sy` (output) vector (N_y) $\vec{s}^{(y)}$, estimated standard deviation of $\vec{y}$.
- `cov_flag` (input/output) scalar, bitmap of flags describing network

  `cov_flag` = 99999 don't load anything, just set `Nx`, `Nh`, `Ny`.

  `cov_flag & 3` = 0 no covariance or Hessian of $\vec{\theta}$ available

  `cov_flag & 3` = 1 covariance of $\vec{\theta}$ stored in `theta_cov`

  `cov_flag & 3` = 2 Hessian of ℓ stored in `theta_cov`
- `theta_cov` (output) matrix ($N_\theta \times N_\theta$), error covariance of network weights, or Hessian of ℓ.

Note: All arrays must be allocated before a call with `cov_flag` $\neq$ 99999. Otherwise a memory fault will occur. The routine will print an error to `stderr` and return when an unexpected `NULL` is encountered.

4.3.3 nnlib_save_training_set

The function `nnlib_save_training_set` saves a training data set to a binary file. It has only inputs.

```
void nnlib_save_training_set(const char *filename, const unsigned long int Nt,
                             const unsigned long int Nx, const double *X,
                             const unsigned long int Ny, const double *Y,
                             const double *s, const unsigned long int sflag);
```

- `filename` (input) null-terminated C string giving the data set file name.
- `Nt` scalar, N_t, number of training samples.
- `Nx` scalar, N_x, dimension of $\vec{x}$.
- `Ny` scalar, N_y, dimension of $\vec{y}$.
- `X` matrix ($N_t \times N_x$), $\vec{\vec{x}}$ at each training sample.
- `Y` matrix ($N_t \times N_y$), $\vec{\vec{y}}$ at each training sample.
- `s` error estimate $\underline{\underline{\Sigma}}_t$ on training samples $\vec{y}$. Meaning depends on `sflag`.

- `sflag` scalar, bitmap of flags describing error covariance of $\vec{y}$ training data.

  `sflag & 3` $= 0$ `s` is scalar, same error variance for all y and all samples

  `sflag & 3` $= 1$ `s` is vector N_y, same error variance for all samples

  `sflag & 3` $= 2$ `s` is matrix $N_t \times N_y$, sample-dependent error variance

  `sflag & 3` $= 3$ `s` is tensor $N_t \times N_y \times N_y$, sample-dependent error covariance. For each sample, there is a different $N_y \times N_y$ error covariance matrix for $\vec{y}$, i.e., a full sample-dependent $\underline{\underline{\Sigma}}_t$.

4.3.4 nnlib_load_training_set

The function `nnlib_load_training_set` loads a training data set from a binary file. Because the size of the arrays needed by the data set aren't known until one starts to read the file, one can call `nnlib_load_training_set` with `sflag` $= 99999$ to have it only set the scalar size variables (`Nt`, `Nx`, and `Ny`). Using those values, one can allocate the necessary arrays and call the function again to actually load the data set.

```
void nnlib_load_training_set(const char *filename, unsigned long int *Nt,
                             unsigned long int *Nx, double *X,
                             unsigned long int *Ny, double *Y,
                             double *s, unsigned long int *sflag);
```

- `filename` (input) null-terminated C string giving the data set file name.
- `Nt` (output) scalar, N_t, number of training samples.
- `Nx` (output) scalar, N_x, dimension of $\vec{x}$.
- `Ny` (output) scalar, N_y, dimension of $\vec{y}$.
- `X` (output) matrix ($N_t \times N_x$), $\vec{\vec{x}}$ at each training sample.
- `Y` (output) matrix ($N_t \times N_y$), $\vec{\vec{y}}$ at each training sample.
- `s` (output) error estimate $\underline{\underline{\Sigma}}_t$ on training samples $\vec{y}$. Meaning depends on `sflag`.
- `sflag` (output) scalar, bitmap of flags describing error covariance of $\vec{y}$ training data.

  `sflag & 3` $= 0$ `s` is scalar, same error variance for all y and all samples

  `sflag & 3` $= 1$ `s` is vector N_y, same error variance for all samples

  `sflag & 3` $= 2$ `s` is matrix $N_t \times N_y$, sample-dependent error variance

  `sflag & 3` $= 3$ `s` is tensor $N_t \times N_y \times N_y$, sample-dependent error covariance. For each sample, there is a different $N_y \times N_y$ error covariance matrix for $\vec{y}$, i.e., a full sample-dependent $\underline{\underline{\Sigma}}_t$.

Note: All arrays must be allocated before a call with `sflag` $\neq 99999$. Otherwise a memory fault will occur. The routine will print an error to `stderr` and return when an unexpected `NULL` is encountered.

4.4 Network Training and Evaluation

4.4.1 nnlib_fit

The function `nnlib_fit` will fit a neural network, i.e., determine $\vec{\theta}$ that best reproduces $\underline{\underline{Y}}$. Upon request, it will also compute $\mathrm{cov}\,\vec{\theta}$. The function returns ℓ at the end of the optimization. In a sense, the function takes a training data set and returns a neural network.

```
double nnlib_fit(const unsigned long int Nt,
                 const unsigned long int Nx, const double *X,
                 const unsigned long int Nh,
                 const unsigned long int Ny, const double *Y,
                 const double *s, const unsigned long int flag,
                 const unsigned long int MaxIter,
                 double epsabs,
                 double *theta,
                 double *xbar, double *ybar, double *sx, double *sy,
                 double *theta_cov);
```

- `Nt` (input) scalar, N_t, number of training samples.
- `Nx` (input) scalar, N_x, dimension of $\vec{x}$.
- `Nh` (input) scalar, N_h, number of hidden nodes.
- `Ny` (input) scalar, N_y, dimension of $\vec{y}$.
- `X` (input) matrix ($N_t \times N_x$), $\vec{\vec{x}}$ at each training sample.
- `Y` (input) matrix ($N_t \times N_y$), $\vec{\vec{y}}$ at each training sample.
- `s` (input) error estimate $\underline{\underline{\Sigma}}_t$ on training samples $\vec{y}$. Meaning depends on `flag`.
- `flag` (input) scalar, bitmap of various flags

  `flag & 3` $= 0$ `s` is scalar, same error variance for all y and all samples

  `flag & 3` $= 1$ `s` is vector N_y, same error variance for all samples

  `flag & 3` $= 2$ `s` is matrix $N_t \times N_y$, sample-dependent error variance

  `flag & 3` $= 3$ `s` is tensor $N_t \times N_y \times N_y$, sample-dependent error covariance. For each sample, there is a different $N_y \times N_y$ error covariance matrix for $\vec{y}$, i.e., a full sample-dependent $\underline{\underline{\Sigma}}_t$.

  `flag & 8` $= 0$ quiet, no output `stdout`.

  `flag & 8` $= 8$ verbose, report progress to `stdout`.

  `flag & 16` $= 0$ initialize $\vec{\theta}$ to random values.

  `flag & 16` $= 16$ start at $\vec{\theta}$ supplied in `theta`.

  `flag & 96` $= 0$ optimize with Broyden-Fletcher-Goldfarb-Shanno, BFGS (recommended).

  `flag & 96` $= 32$ optimize with Conjugate Fletcher-Reeves, Conjugate FR.

  `flag & 96` $= 64$ optimize with Conjugate Polak-Ribiere, Conjugate PR.

  `flag & 96` $= 96$ optimize with Nelder-Mead Simplex.

  `flag & 132` $= 0$ don't populate `theta_cov`.

  `flag & 132` $= 4$ populate `theta_cov` with $\operatorname{cov} \vec{\theta}$.

  `flag & 132` $= 128$ populate `theta_cov` with the Hessian of ℓ.

  `flag & 256` $= 256$ use input values of `xbar`, `ybar`, `sx`, `sy`.

- `MaxIter` (input) maximum number of iterations for optimizer.
- `epsabs` (input) stopping criterion for optimizer:

  Nelder-Mead stops when simplex size is less than `epsabs`.

  All other optimizers stop when gradient length is less than `epsabs`.

- `theta` (input/output) vector (N_θ), network weights, as in (4).
- `xbar` (output) vector (N_x), $\vec{\bar{x}}$, estimated mean of $\vec{x}$.
- `ybar` (output) vector (N_y), $\vec{\bar{y}}$, estimated mean of $\vec{y}$.
- `sx` (output) vector (N_x), $\vec{s}^{(x)}$, estimated standard deviation of $\vec{x}$.
- `sy` (output) vector (N_y) $\vec{s}^{(y)}$, estimated standard deviation of $\vec{y}$.
- `theta_cov` (output) matrix ($N_\theta \times N_\theta$), error covariance of network weights, or Hessian of ℓ, depends on `flag`.

4.4.2 nnlib_eval

The function `nnlib_eval` evaluates a trained neural network and, if requested, returns error estimates for the estimate $\vec{y}$. Note that the function is built to evaluate the network at multiple points ($N_t \geq 1$).

```
void nnlib_eval(const unsigned long int Nt,
                const unsigned long int Nx, const double *X,
                const unsigned long int Nh, const double *theta,
                const double *xbar, const double *ybar,
                const double *sx, const double *sy,
                const unsigned long int Ny, double *Y,
                const unsigned long int dY_flag, const double *theta_cov, double *dY);
```

- `Nt` (input) scalar, N_t, number of training samples.
- `Nx` (input) scalar, N_x, dimension of $\vec{x}$.
- `Nh` (input) scalar, N_h, number of hidden nodes.
- `Ny` (input) scalar, N_y, dimension of $\vec{y}$.
- `X` (input) matrix ($N_t \times N_x$), $\vec{\vec{x}}$ at each training sample.
- `dY_flag` (input) scalar, bitmap of various flags

  `dY_flag & 3` $= 0$ don't compute errors on $\vec{y}$.

  `dY_flag & 3` $= 1$ return "standard errors" for $\vec{y}$, `dY` is $N_t \times N_y$.

  `dY_flag & 3` $= 2$ return covariance matrix for $\vec{y}$, `dY` is $N_t \times N_y \times N_y$.
- `theta` (input) vector (N_θ), network weights, as in (4).
- `xbar` (input) vector (N_x), $\bar{\vec{x}}$, estimated mean of $\vec{x}$.
- `ybar` (input) vector (N_y), $\bar{\vec{y}}$, estimated mean of $\vec{y}$.
- `sx` (input) vector (N_x), $\vec{s}^{(x)}$, estimated standard deviation of $\vec{x}$.
- `sy` (input) vector (N_y) $\vec{s}^{(y)}$, estimated standard deviation of $\vec{y}$.
- `theta_cov` (input) matrix ($N_\theta \times N_\theta$), $\operatorname{cov} \vec{\theta}$.
- `Y` (output) matrix ($N_t \times N_y$), $\vec{\vec{y}}$ at each training sample.
- `dY` (output) error estimate for $\vec{y}$, format depends on `dY_flag`. covariance of network weights, or Hessian of ℓ, depends on `flag`.

5 Matlab Wrappers

A Matlab wrapper `nnlib.m` is provided which can interface all of the functions in `nnlib.h`. In Matlab, the matrix and tensor variables have their appropriate linear-algebraic shape, but their indices are transposed/permuted as needed by the wrapper for calls to the DLL.

6 Sample Program

The library comes with a sample program `nntrain.c` which will train a neural network on a training set. The Makefile can compile this sample program with `make nntrain.exe` or `make nntrain.x` on Unix systems.

7 Parallelization

The library now supports parallel processing. This has no effect on the shared library (dll or so). The remainder of this section is for users of `nntrain.c` and C programmers using the library code.

7.1 Parallelization with OpenMP

The library now supports parallel processing with OpenMP: `make nntrain.ompx` will create the OpenMP-enable executable. All that is needed is gcc version 4.2 or higher (note: many systems still use gcc 3 by default. Higher level versions are often available as gcc4 gcc42 etc.). This functionality is enabled when the compiler "macro" `USEOMP` is set, e.g., with `-DUSEOMP` on the compiler command line. The resulting executable `nntrain.ompx` can be run on the command line like any other unix command. I have not yet attempted to create a dos executable with OpenMP enabled. This will require an upgrade to experimental MinGW gcc 4.

The code is parallelized by having each thread work on part of the training set. Only the function `nn_fit_eval` had to be modified.

The OpenMP implementation should work well on multi-core systems (e.g., a fancy modern desktop). It has been tested on a 16-core Solaris system.

7.2 Parallelization with MPI

The library now supports parallel processing with MPI: `make nntrain.mpix` will create the MPI-enable executable, if an MPI wrapper mpicc (which wraps around gcc) is available. This functionality is enabled when the compiler "macro" `USEMPI` is set, e.g., with `-DUSEMPI` on the compiler command line. The resulting executable `nntrain.mpix` can be run on the command line with `mpirun nntrain.mpix ...`

The code is parallelized by breaking up the training set into pieces, and distributing these pieces to the "slave" processes. Each evaluation of the neural network on the training set is accomplished by sending the network coefficients to each slave and retrieving the penalty function (ℓ), its gradient and hessian (as needed). Because ℓ is a linear sum over the training data (index variable t above), the results from the training subsets can simply be added up. (The regularization term is evaluated only in the master process).

A handful of routines `nn_mpi...` have been created, and `nn_fit_eval` has been modified to break up the evaluation of the network (and calculation of gradient and hessian) over the training data into independent chunks. The example routine `nntrain.c` now includes the calls necessary to make all of this work. Aside from breaking up the calculation, it was necessary to slightly modify how the network is initialized; namely, the calculation of `xbar`, `ybar`, `sx`, and `sy` had to be performed on the whole training set before it was broken into pieces. This introduced a new flag (256) to indicate that `xbar-sy` were computed before the call to `nnlib_fit`.

The new MPI routines are briefly described below:

- `int nn_mpi_init(int *argc,char **argv[])` called as `mpi_rank = nn_mpi_init(&argc,&argv)`, initializes MPI, forks slave processes, removes MPI-specific args from `argv`, and returns the process's rank (id number) after the fork. Process 0 is the master, all others are slaves. This function should be the first entry in `main`.

- `void nn_mpi_slave()` runs a slave process (this is the only function a slave process should call after `nn_mpi_init` before it exits).

- `set_type nn_mpi_dispatch(set_type set, net_type net)` called as `subset = nn_mpi_dispatch(set,net)`, sets the net's `xbar`, `ybar`, `sx`, and `sy`, breaks up set into subsets and distributes the initial network parameters (size, `xbar`, etc.) to the slaves. Returns the subset that the master is to work on. This is the subset that should be passed to `nnlib_fit`, but the original set will still need to be freed with `free_set`. Subsequent calls to `nn_mpi_dispatch` will prepare the slaves to work on the new subsets and net.

- `void nn_mpi_cleanup()` signals the slaves to clean up and terminate. Calls MPI's finalize routine for the master.

When `USEMPI` is not defined, these routines do little or nothing.

8 Neural Network Tribal Knowledge

The neural networks described herein are "feed-forward perceptron networks." "Feed-forward" means that there is no memory between separate evaluations of the network, and "perceptron" just means that the network is made up of mathematical functions that resemble the step-like response of a biological neuron to multiple inputs. This kind of network is in many ways the simplest, and it exhibits essentially no "intelligent" behavior. It is a simple, parametric mathematical fit to some training data.

Over-fitting is a common concern with neural networks. As one increases the number of inputs (N_x) or hidden nodes (N_h), one will tend to increase the apparent fit to the training data. However, as with a high-order polynomial fit, this can lead to unintended features between the training points. The regularization term ($\vec{\theta}^T\vec{\theta}/2$) helps a bit with this problem, but ultimately it is better to reserve some portion of the training data, say 10-20%, for out-of-sample validation.

A robust strategy for training a neural network is to (1) partition the data into 5 distinct, randomly selected partitions. For each partition in turn, hold the partition as a validation set and train on the other 4 partitions. Perform this training 5 times, each time from a distinct random starting point, to ensure improve the odds of finding a global minimum. Thus, one trains the network 25 times. Perform this 25-fold training process for various values of N_h, i.e. for various numbers of free parameters in the network. Select the N_h for which the worst performance on any validation partition is minimum. Retrain the network on the whole data set, with the selected value of N_h 5 times, and pick the network with the best in-sample performance.

The optimization schemes all have their various strengths and weaknesses. Experience shows that the Nelder-Mead simplex tends to find the best global minimum but is comparatively slow to get there. Thus, a good approach is to cycle between NM-Simplex and BFGS, the fastest minimizer. The `nntrain.exe` example does this when the `-nm0` switch is provided on the command line.

At present, I am still working on the problem of a Hessian for $\vec{\theta}$ that has some negative eigenvalues. It should not. Negative eigenvalues imply a local maximum in ℓ along some direction. Either the minimization routine needs to keep working (likely) or the negative eigenvalues result from round-off errors. If it's simply round-off errors, then I'll zero-out the negative eigenvectors in the calculation of $\mathrm{cov}\,\vec{\theta}$. For now, I'm seeing how far I can push the minimizer to get a numerically non-singular Hessian. I can definitely get rid of the singularity by reducing the number of hidden nodes.

If there are known analytical approximations between the inputs and outputs, it is a good idea to apply these as a preprocessing/postprocessing step. While neural networks are capable of describing more-or-less arbitrary nonlinear structures, they can do this much more economically if the outputs are approximately linear functions of the inputs, and if each output is well-approximated by one of the inputs. For example, it is helpful to use dipole L as an input for a neural network that is fitting L^*.

When using periodic functions (like local time, latitude, or longitude) as inputs or outputs, it is often helpful to decompose one or more of these into sine and cosine, thereby eliminating the roll-over discontinuity. For outputs, of course, one must post-process the results to address sine or cosine greater than 1.

A note on verification: I have verified the C implementation of the calculation of ℓ and its gradient and Hessian with respect to $\vec{\theta}$ by performing the same calculations with the Matlab symbolic math toolbox. The answers agree to within numerical precision ($\sim 10^{-15}$).