The MINT cookbook

Stefano Ghirlanda

Version 0.1 of August 20, 2013

Contents

1

Is this for me?

This short chapter provides a bird's eye view of the principal features of MINT to help you decide whether it fits your needs. MINT is a library for neural network simulations written in the C programming language (specifically, ANSI C). Using it will require writing C programs (possibly very short ones) and simple text files to describe your networks.

1.1 MINT philosophy

MINT takes a "natural scientist" approach to neural network simulations. These are viewed as a means to gain insight into the operation of nervous systems, not as mathematical tools for statistics, curve fitting, and so on. This perspective has affected the design of MINT in a number of ways. If you are an engineer looking for sophisticated supervised learning or network optimization algorithms, MINT may not be the best tool for you. If you are interested in how biological nervous systems process inputs and generate behavior, MINT is potentially for you.

In addition to a good number of built-in capabilities (which you can discover reading this manual), MINT provides you with a uniform representation of neural networks (through the data structures introduced in section 1.4 below) that enables you to focus on the specifics of your particular neural network without worrying about such things as memory management, data input and output, and other administration of general properties of neural networks (e.g., that neural networks are made of nodes wired together). What MINT can do, moreover, can be extended easily. With a few lines of code you can add a new neuron model to MINT and use it as if it were one of those already in the library. Ditto for weight models and learning algorithms. In many cases you will be able to write a small C program once and for all, and do a lot of work simply by changing a text file that describes your network.

1.2 What MINT does, and what it does not

MINT is good at description neural network compactly in small text files. You can also add your own node and weight models while using all of the other MINT functions, and specify in detail what operations are to be performed to update a network. MINT is less good for simulations of a very small number of very detailed neurons (e.g., detailed 3D models of neurons). These are possible, but you would end up doing most of the coding yourself with little benefit of the MINT infrastructure. More specifically here are lists of currently available and unavailable features:[1]

Available: general network topology, i.e., any number of node groups and weight matrices connected in any way (e.g., many weight matrices between two node groups), synchronous and asynchronous dynamics, arbitrary node and weight models (e.g., spiking or non-spiking neurons), "global" influences on neurons and weights (e.g., extracellular neurotransmitter concentrations), use images as input (requires the FreeImage library, see REF), use usb camera as input (on Linux, see REF).

Unavailable: explicit positioning of neurons in 2D or 3D space, event-driven simulations and time delays in synaptic transmission.

The currently unavailable features may be implemented in the future.

1.3 Space and speed

From a technical point of view, MINT has a small footprint. It compiles to under 100 KB on x86-64 (PC) and ARM processors and wastes little memory. For example, the memory overhead of a weight matrix is only 6 integer variables and 3 pointers.[2] Some optimization also make MINT a good choice if you need speed. REF Multi-threaded execution and sparse matrix storage are also available. REF

1.4 The very basics

MINT describes neural networks in terms of five fundamental data structures. The first three should be intuitive to anyone interested in neural network simulations:

- *Node groups*: Groups of simulated neurons
- *Weight matrices*: Simulated synaptic connections between or within nodes
- *Networks*: Collections of node groups and weight matrices

[1] If you think that something should be added to either list, please contact us. REF

[2] By memory overhead we mean variables that MINT uses to keep track of things, rather than to hold information that is strictly part of the neural network, such as weight values.

The last two data structures are specific to MINT and are used to describe aspects of neural network operation:

- *Operations* ("ops" for short): Anything that can be done to a node group, a weight matrix, or a network
- *Spread schemes*: Representations of how the network is updated (how neural activation "spreads" through the network)

Examples of operations are initialization of weight matrices (e.g., setting weights at random), updates of node state based on received input, changes in weight matrices ("learning"), and so on. Using these five data structures many kinds of neural networks can be set up relatively simply and efficiently. The rest of this manual introduces you to the data structures in some detail and walks you through many examples of how MINT can be used in practice.

1.5 Your first MINT network

The easiest way to specify a network architecture in MINT is to write a specially formatted text file. We use the file suffix `.arc` (for "architecture file"), but you can use anything you like. A simple network with 10 input units connected to one output unit with sigmoid nonlinearity is described as follows:

```
1 network 2 1
2 nodes 10 0
3 nodes 1 0 sigmoid 0.1 1
4 weights 1 10 0 from 0 to 1 random -1 1
```

Line 1 says that the network has 2 groups of nodes ("layers") and 1 weight matrix. Line 2 says that the first group of nodes contains 10 nodes, with 0 internal state variables each (see WHERE for an explanation of state variables). Line 3 says that the second group of nodes is made of 1 node, also with 0 state variables. It also identifies this node as having a sigmoid transfer function; the following two numbers are parameters for the sigmoid function (see SIGMOID PARAMETERS). Lastly, line 4 says that there is one weight matrix with 1 row and 10 columns; each weight has 0 internal states. The "from" and "to" parts specify that the output of node group 0 is sent as input to node group 1.[3] The "random" part initializes weights from a uniform distribution between -1 and 1.

Assuming the file is called `network.arc`, the following program creates a network reading with the specified architecture:

[3] Rows and columns could be determined by knowing the "from" and "to" parameters, but repeating them here keeps the library simpler to maintain and easier to use in a number of ways.

```
1  #include "mint.h"
2  #include <stdio.h>
3
4  int main( void ) {
5    FILE *f;
6    struct mint_network *net;
7    f = fopen( "network.arc", "r" );
8    net = mint_network_load( f );        /* load net from file */
9    fclose( f );
10   mint_network_save( net, stdout );    /* display the network */
11   mint_network_del( net );             /* free memory */
12   return 0;
13 }
```

There are only five lines here that are specific to MINT:

- Line 1 reads the `mint.h` header file.
- Line 6 declares a pointer to a variable of type **`struct`** `mint_network *`, which holds all the information pertaining to the network.
- Line 8 loads the network from file. Loading performs error checks that ensure you end up with a valid network (e.g., that weight matrices have dimensions that match the node groups they connect).
- Line 10 displays the network to the C standard output (e.g., terminal window).
- Line 11 deletes the network (frees up the memory the network is using).

You can find this program and the architecture file above in `example/minimal` in the MINT source distribution.

The `mint_network_save` function can be used to save to a regular file, rather than standard output, resulting in an exact copy of the network, complete of weight values and node activation values that were not explicitly given in the architecture file above. Such a file can be read by `mint_network_load` to create an exact copy of the network.

2

How to get MINT, and what you get

2.1 How

—EMPTY UNTIL FIRST PUBLIC RELEASE—

2.2 What

The MINT distribution contains the following files and folders:

- `doc/`: Documentation, consisting of this manual and a set of HTML pages that document the MINT source code.
- `src/`: The C source for MINT.
- `examples/`: Numerous examples of various features.
- `SConstruct`: This is a file used by the installation system.
- `README`: A file pointing you to this document.

2.3 Installation

3

MINT data structures

MINT is organized around five variable types:

1. `mint_nodes` represent groups of neurons.
2. `mint_weights` represent synaptic connections between groups of neurons.
3. A **`struct`** `mint_network` represents a neural networks.
4. A `mint_op` represent an operation that can be performed on the previous three types.
5. A `mint_spread` holds information on how a network is updated.

You will usually deal with nodes, weights, and networks. Occasionally you will want to work with ops, very rarely you will have to use spreads in your C code.

Neural network computations are implemented as operations on nodes, weights, and networks. Once a network is set up, there are three kinds of operations that are most typical.

Node input calculations

A typical operation is, for instance, to calculate the output of neurons based on their current input (and possibly internal state variables). This is called a "node update"

An explanation of operations is provided WHERE.

The building blocks of a MINT network are:

Node groups: They model groups of neurons. They are variables of type `mint_nodes` and look to the user as two-dimensional arrays of **`float`**s. The first index refers to a *state variable* of the nodes, the second is the node index. Two state variables are automatically created for each node group: input and output. Thus, if that `n` is a node group, `n[0][i]` is the input received by the *i*-th node and `n[1][i]` is its output. Other state variables can be created as needed, see typically to endow the nodes with specific properties (section 4.5.

Weight matrices: They model synapses between neurons and other ways neurons interact (see section 4.10, section 4.11). They are variables of type `mint_weights` and look to the user as three-dimensional arrays of **`float`**s.

The first index refers to a state variable of the weights, the second the row index and the third the column index in the weight matrix. One state variable is automatically created and represents the weight value. Thus, if `w` is a weight matrix, `w[0][r][c]` is the value of the weight in row r and column c of the matrix. Other state variables can be created as needed.

Update functions determine how nodes change their activity in response to input, and how weights change their values (how they *learn*). Each node group and each weight matrix is given an update function, unless we do not need to update them (e.g., fixed weights).

Networks are composed of node groups connected by weight matrices, linked together by a *spread scheme*, which is a description of the operations that are performed when the network is updated. They are variables of type **`struct`** `mint_network`.

Typically, you will create one or more MINT networks by specifying the properties of a number of node groups (their size, number of states, activation function, etc.) and how they are connected via weight matrices. You may also specify how the network is updated (synchronouse or asynchronous dynamics, feed-forward spreading of activation, etc.).

4

Cookbook

A MINT node is characterized by an input y, an output z and optionally a state vector $\mathbf{s}$, with any number of components. Node inputs may be set by the user (e.g. to

simulate the reception of a stimulus) or arise from the weighted outputs of other nodes (via weight matrices). Node output is determined from node input and node state by an arbitrary function:

$$z = f(y, \mathbf{s}, \mathbf{g}) \tag{4.1}$$

This operation is a called a *node update*, and is typically performed as part of the *spreading scheme*. The spreading scheme is the sequence of matrix-vector multiplications and node update operations whereby network state is updated (see section 4.18). In addition to affecting node output, an update may have other effects such as modifying node state.

4.16.2 Weight model

A weight is characterized by a value w and a state vector $\mathbf{s}$, with 0 or more components. A weight can be updated in much the same way as a node, though weight update is usually called "learning." In MINT, an arbitrary function can be supplied to update weights based on current value, current state and the current input, output and state variables of the nodes joined by a weight.

4.17 Node update functions

4.17.1 Using update functions provided with MINT

See `update.h` and `nlib.h`.

4.17.2 Adding update functions

Adding an update function to MINT requires two steps:

1. writing the update function itself;
2. letting MINT know about the function.

These operations do not require modifications to the MINT source. The prototype of a node update function is

```
1 void function( mint_nodes n, int min, int max );
```

where

- `n` is the node object to update;
- `min-max` is the range of nodes that the function should update;

The basic structure of an update rule is very simple:[1]

[1] The **for** loop assumes that the same operation is performed to update all nodes. Different operations, however, may be performed if desired.

```
void function( mint_nodes n, int min, int max ) {
  int i;
  /* retrieve update function parameters from node object */
  for( i=min; i<max; i++ ) {
    /* update node i */
  }
}
```

For instance, imagine we want to update nodes as follows: if the input is above a threshold, the output should be set to 1, otherwise it should be set to 0. We store the threshold value as parameter 0 of the rule (we will see below how to tell MINT about the number of parameters taken by a rule). The implementation of the rule would be:

```
void threshold_node( mint_nodes n, int min, int max, float *p ) {
  int i;
  float threshold;
  threshold = mint_update_get_param( mint_nodes_get_update(n), 0 );
  for( i=min; i<max; i++ ) {
    if( n[0][i] > threshold )
      n[1][i] = 1.;
    else
      n[1][i] = 0.;
  }
}
```

Note that the function specifies an update mechanism (the threshold mechanism), but does not determine the threshold value itself, which is stored as a parameter in the node object. Note also that the function need not check that `min` and `max` are sensible values for this node object, MINT checks this (in debug mode, see section 4.19).

To be used with MINT, the function must be registered with a call like:

```
void mint_update_nadd( void (*function)(mint_nodes, int, int, float *),
                       int nstates, int nparam, float *param );
```

Thus for the above function:

```
  float p[1] = { 1. };
  mint_update_nadd( threshold_node, 0, 1, p );
```

The values provided for the parameter(s) become the default values for the function, but can be changed as needed (see `update.h`).

4.18 Spreading schemes

A spreading scheme defines a sequence of matrix-vector multiplications and node updates.

4.19 Debugging

Compiling MINT the `DEBUG` flag turns on a number of checks, ensuring for instance that indices supplied as function parameters are in the proper range, and that object copies operate on objects with the same “geometry” (e.g. node groups with the same size and number of states). If a check fails, the program displays a hopefully informative message and calls `abort()`. Section 2 explains how to compile a debugging version of MINT.

4.20 Replacing the stock matrix-vector multiplication

5

Space and speed

The major efficiency consideration is not having data structures for single nodes and weights, but rather for node groups and weight matrices directly, and caring for the memory layout of data. The benefits are seen mostly in the function that performs matrix-vector multiplication, which looks sort of:

```
1 void mint_weights_mult( mint_weights w, mint_nodes from, mint_nodes to ) {
2   int i, j, r, c;
3   c = mint_weights_cols( w );
4   r = mint_weights_rows( w );
5   for( i=0; i<r; i++ ) for( j=0; j<c; j++ )
6     to[0][i] += w[0][i][j] * from[1][j];
7 }
```

Due to the memory layout of nodes and weights, all memory accesses are sequential in the above **for** loop.

Efficiency considerations have also influenced the design of node and weight update functions. The approach that first comes to mind to update a node group is something like (pseudo-code):

```
1 for( i=0; i<n; i++ )
2   update_the_node( node_group_object, i );
```

where the function call updates node i of the node group. The drawback is that there are n function calls. In MINT, an update function takes a *range* as argument, telling it which nodes to update. Thus the loop is inside the update function, saving $n-1$ function calls. In other words, in MINT we have (pseudo-code):

```
1   update_node_range( node_group_object, imin, imax );
```

where `update_node_range()` does both the looping and the node updating. Note that it is not possible to inline the function in the simple-minded approach,

because if the function can be configured by the user the compiler cannot know which function to inline. The same approach is taken for weight updates.

BENCHMARK RESULTS