The MINT cookbook

Stefano Ghirlanda

Version 0.1 of June 30, 2014

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, C89). Using it requires 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 read about in 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 general properties of neural networks (e.g., that neural networks are made of nodes wired together). MINT, moreover, is extensible. 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

With MINT you can describe arbitrary neural networks compactly in small text files. You can also add your own node and weight models and using them seamlessly with the other MINT functions. You can 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 relatively

little benefit from the MINT infrastructure. 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), interface with motors and sensors on a Raspberry Pi computer (to build network controlled robots, see REF).

Unavailable: explicit positioning of neurons in 2D or 3D space, event-driven simulations and time delays in synaptic transmission, networked parallel programming (MPI is planned), modeling of neural morphology

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 integers and 4 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 nodes
- *Networks*: Collections of node groups and weight matrices

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), calculation of node output 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] 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.

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 feedforward
3 nodes input size 10
4 nodes output size 1 sigmoid 0.1 1
5 weights input-output uniform -.1 1
```

Line 1 starts

the network and line 2 specifies it is a feedforward network.[3] Line 3 adds a first group of nodes called "`input`" which contains 10 nodes. Line 4 adds a second group of nodes, called "`output`" with just 1 node and identifies this node as having a `sigmoid` transfer function. Two parameters to configure the sigmoid function follow, see REF SIGMOID PARAMETERS. Lastly, line 5 says that there is one weight matrix connecting input to output, which is initialized upon network creation with uniformly distributed numbers between -1 and 1. Assuming the file is called `starthere.arc`, the following program creates a network with the specified architecture:

```
1  #include "mint.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main( void ) {
6    FILE *f;
7    struct mint_network *net;
8    f = fopen( "starthere.arc", "r" );
9    net = mint_network_load( f );        /* load net from file */
10   fclose( f );
11   mint_network_save( net, stdout );    /* display the network */
12   mint_network_del( net );             /* free memory */
13   return EXIT_SUCCESS;
14 }
```

There are only 5 lines in this file that are specific to MINT:

- Line 1 reads the `mint.h` header file.
- Line 7 declares a pointer to a variable of type **`struct`** `mint_network`, which holds all the information pertaining to the network.
- Line 9 loads the network from file. Loading performs many error checks that ensure you end up with a valid network. If a network can't be loaded, `mint_network_load` terminates the program with an error message.
- Line 11 displays the network to the C standard output (e.g., terminal window). You can also use this function to save a network to a file for later analysis or loading into another program.

[3] MINT actually checks that this is true! The `feedforward` specification influences how the network is updated, see REF.

- Line 12 deletes the network (frees up the memory the network is using).

You can find this program and the architecture file above under `example/starthere` in the MINT source distribution. If you run the program you will get something like:[4]

```
1  network
2  feedforward
3  nodes input
4  size 10 states 0
5  0 0 0 0 0 0 0 0 0 0
6  0 0 0 0 0 0 0 0 0 0
7  nodes output
8  size 1 sigmoid 0.1 1 states 0
9  0
10 0
11 weights input-output
12 random -1 1 1 states 0 cols 10 rows 1
13 -0.155077 0.559366 0.669266 -0.974252 -0.192495 \
14 -0.238614 0.357579 -0.710728 0.0285505 -0.562432
15 spread
16 input input-output output
```

This includes everything we had in the initial file plus additional information necessary to recreate an exact copy of the network:

- Lines 3–6 detail the `input` node group, adding the information that these nodes have 0 internal state variables (the default, see REF to learn about state variables) and also listing the value of node inputs (line 5) and node outputs (line 6). These are all zeros because we have never actually used the network.
- Lines 7–10 detail the `output` node group in the same way.
- Lines 11–14 detail the `input-output` weight matrix, stating explicitly its dimensions (`rows` and `cols`) and the fact that the weights have no internal state variables (see REF). Lines 9–10 list the values that the weights received from the random initialization. If this file is loaded again, these values will be used and the initialization will *not* be performed. This ensures that you can load an exact copy of the network.[5]
- Lines 15–16 detail how the network is updated every time that the function `mint_network_operate` is used (it was not in our sample program. This information consists of the keyword `spread` followed by a list of weight matrix and node group names. In this case, we see that the `input` node group is updated first, then the `input-output` matrix is used to propagate `input` activity to `output` nodes, and finally `output` nodes are updated. This information has been generated by the `feedforward` specification. Each spread step may involve several operations depending on how the network has been setup. In this example, for instance, `output` nodes are updated using the sigmoid function specified on line 8. Setting up network spread is a key component of neural network simulation in MINT, and is detailed REF.

[4] The \ character on line 13 indicates a line break we inserted here, and will not be part of the output.

[5] It is also possible to re-initialize the weights at random, see REF.

1.6 Where to go from here

The next chapter

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.
- A few files used to install MINT, see below.
- `README`: A brief file pointing you to this document.

2.3 Installation

3

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. These operations are programmed through functions with a specific interface (see REF) and they can all be specified in network architecture files. For example, we saw in 1.5 that the `sigmoid` operation instructs mint to calculate node output as a sigmoid function of node input.

3.1 The neural population model

MINT defines a type `mint_nodes` to manage population of model neurons. A variable of type `mint_nodes` is referred to as a "node group." A node group is typically created from a description contained in a text file, such as:

```
1 nodes n1 size 100
```

where n1 is the name of the group. This line creates a node group with 100 neurons. Creating the node group from file is done like this:

```
1 mint_nodes n;
2 file = fopen( ‘‘nodes.arc’’, ‘‘r’’ );
3 n = mint_nodes_load( file );
4 fclose( file );
```

The variable n can now be accessed as a two-dimensional array of **`float`**s with the following semantics:[1]

- `n[0][i]` is the input received by node `i` (`0 ≤ i < size`)
- `n[1][i]` is the output of node `i`.

[1] As it is usual in C, there is no automatic check that array indices make sense, although all MINT functions perform such checks.

It is also possible to endow nodes with additional state variables. The purpose of such variables is to enable programming of neuron models whose dynamics depends on more than input and output, e.g., spiking neuron models in which a 0/1 output is determined based on a continuous membrane potential. In such a case, node output would be set to be 0 or 1 according to the value of a state variable modeling membrane potential (see REF for an example). States are added like this:

```
1 nodes n1 size 100 states 1
```

which would add one state variable, accessed in C code as `n[2][i]`. That is, the *j*-th state variable of node *i* is `n[j+1][i]`.

Many other directives can be added to a node group declaration. The following line, for example, builds a node group in which each node receives normally distributed noise in input (in addition to the input received from other nodes in the network) and computes output from input using a logistic function:

```
1 nodes n1 size 100 noise 0 0 0.1 logistic
```

The three parameters following `noise` mean that the noise is added to variable 0 (node input) with a mean of 0 and standard deviation of 0.1. The `noise` and `operations` are applied in the order given. Writing `logistic noise 0 0 0.1` would result in the logistic function being applied before the noise is added to node input. The noise would therefore have no effect. It is possible to add noise to node output by changing the noise target variable: `logistic noise 1 0 0.1`. As a further example, the following line declares spiking nodes with spontaneous activity of 5 spikes/s:[2]

```
1 nodes n1 size 100 spikes 5
```

In summary, MINT defines a number of common operations on nodes that can be used by declaring them in a configuration file (see REF for a complete list). Additionally, MINT provides a framework for adding node operations in a way that integrates seamlessly with the rest of the library (see REF).

Lastly, everything that can be done by writing and loading configuration files can also be done directly in C code, but in a more tedious way.

3.2 The synaptic connection model

MINT defines a type `mint_weights` to manage synaptic connections between node groups. A variable of type `mint_weights` is referred to as a "weight matrix." It can be created in two ways from an architecture file. The first is by providing explicitly its dimensions:

```
1 weights w rows 50 cols 10
```

This would create a weight matrix named `w`, with appropriate dimensions to the output of a node group of 10 nodes (the number of columns) to a node group of 50 nodes (the number of rows). Creating the weight matrix is similar to creating a node group:

[2] "Per second" implies that each node update is taken to correspond to 1 ms of real time. A more precise statement is that `spikes 5` results in a node being turned on (firing) with probability 5/1000 for every update.

```
mint_weights w;
file = fopen( ‘‘weights.arc’’, ‘‘r’’ );
w = mint_weights_load( file );
fclose( file );
```

A weight matrix created like this, however, would have to be added manually to a neural network and connected to node groups to several function calls. This is generally unnecessary as you can define weight matrices directly as part of a neural network (this is, after all, where they do their job), like this:

```
network
nodes n1 size 10
nodes n2 size 50
weights w n1-n2
```

The `network` keyword introduces a neural network (see REF). Then two node groups are defined, and lastly a weight matrix is defined connecting node groups `n1` and `n2`. The first node group is the *pre-synaptic one*, i.e., the one sending output, and the second is the *post-synaptic* one, i.e., the one receiving input. In other words, the synaptic connections modeled by `w` would be between the axons of `n1` nodes and the dendrites of `n2` nodes.[3] This way of defining weight matrices lets MINT take care of bookkeeping such as what are the appropriate dimensions for matrix `w` and which nodes it connects. The whole network can be then loaded as detailed below (and as already seen in section 1.5).

Once a weight matrix is created (let's suppose for simplicity that it is stored in the variable `w`), it can be accessed as a three-dimensional array of **`float`**s in which `w[0][i][j]` is the value of the synaptic connection *from* node *j* in the pre-synaptic node group *to* node *i* in the post-synaptic one. The array is three-dimensional rather than two-dimensional to enable the use of additional state variables of synaptic connections, as we have seen for node groups. State variables can be used to model the internal dynamics of synapses. Suppose, for instance, that you want to take into account that neurotransmitters can deplete and need to be replenished. A simple way of doing so is do have a state variable that represents the amount of available neurotransmitter, and set up a weight

although Node groups can be used as two-dimensional arrays of floats with special semantics. Let's see an example. Nodes The building blocks of a MINT network are:

Node groups: They model groups of neurons. They are variables of type `mint_nodes` and appear 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

[3] Can you have connections between dendrites or other departures from the "from axon to dendrite" paradigm in MINT? Yes. MINT does not enforce any particular neurobiological interpretation of connections and thus you are free to model any kind of neurobiological interaction between neurons through MINT weight matrices. See REF for tips on how to do this.

`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

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*. 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 `nop.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, float *p );
```

where

- n is the node object to update;
- min-max is the range of nodes that the function should update;
- p is a vector of user-configurable parameters that may modify the operation of the update function

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

```
1 void function( mint_nodes n, int min, int max, float *p ) {
2   int i;
3   /* retrieve update function parameters from node object */
4   for( i=min; i<max; i++ ) {
5     /* update node i */
6   }
7 }
```

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

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 = p[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 to

```
void mint_op_add( const char *name, int type, void *f,
                  int nparam, float *param );
```

Thus for the above function:

```
  float p[1] = { 1. };
  mint_op_add( "threshodl", mint_op_nodes_update, threshold_node,
               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 considerations in MINT 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. A bare-bones such function would be:

```
1  void matrix_vector_multiply( 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** loops. Indeed, the actual code for matrix-vector multiplication in MINT is slightly longer to take advantage of this memory layout:[1]

```
1    cols = mint_weights_cols(w);
2    if( mint_weights_is_sparse( w ) ) {
3      for( i=rmin; i<rmax; i++ ) {
4        colind = mint_weights_colind( w, i );
5        jmax = mint_weights_rowlen( w, i );
6        for( j=0; j<jmax; j++ )
7          to[ target ][ i ] += w[0][i][j] * from[ 1 ] [ colind[j] ];
8      }
9    } else {
10     for( i=rmin; i<rmax; i++ ) {
11       for( j=0; j<cols; j++ ) {
12         to[target][i] += w[0][i][j] * from[1][j];
13       }
14     }
15   }
```

[1] The **float** *p argument comes from the general form of MINT ops, and is not used here (see REF).

In lines 7, 8, and 10 we set up pointers to the first weight value, the first target variable of `to` nodes,[2] and the first output value of `from` nodes. Then in the for loops we simply increment these pointers because we know that all memory locations are contiguous. The C compiler, on the other hand, cannot do this because it does not know that `w[0][1]` points to a location that is next to `w[0][0][cols]`. Thus the compiler has to generate additional instructions to look-up memory addresses.

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.[3] The same approach is taken for weight updates.

5.1 Benchmarks

[2] By default, the target is 0, i.e., node input as in the previous listing, but can be changed if desired, see REF.

[3] The update function cannot be inlined because it can be configured by the user, hence the compiler cannot know which function to inline.