This Handbook is partially out of date!

Contents

Simulator Handbook

Introduction

With this neural net simulator, you edit your own network and training algorithm using a built-in “batch”-language. In this language, you specify every detail of your algorithm. You may extend the language by adding and compiling to the simulator your own functions written in C. The language is pre-structured to work itself down from *iterations* to *relaxations (over time)* to *area (layer) visits* to *neuron visits*. A variety of networks with connectionist neurons can be programmed with ease, in particular if neuronal updates are local. Examples are: Hopfield networks, Boltzmann- and Helmholtz machines, backpropagation-trained multi-layer perceptrons. Non-local interactions can also be implemented (like winner-finding within one layer of a Kohonen network) but severely non-local operations (like matrix inversion for standard ICA) will make you program your whole algorithm in a C-procedure.

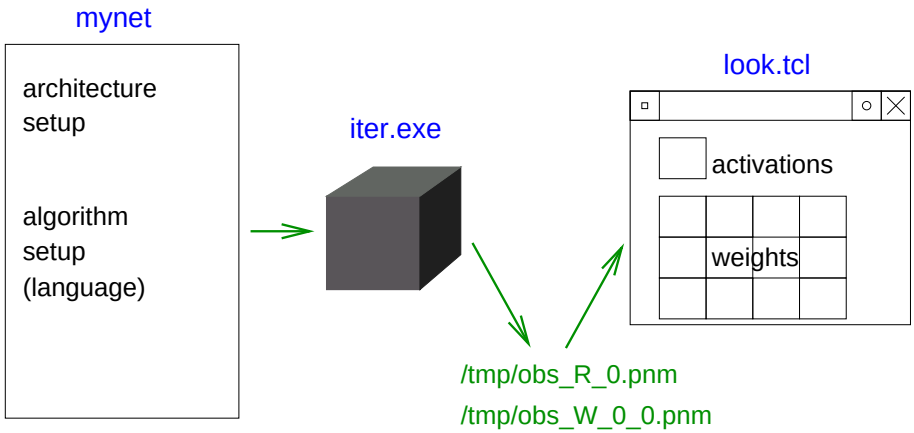

Figure 1: Information flow: language file → code → image files → display tool.

The activations as well as the neuronal weights (synaptic strengths) shall be written periodically into the `/tmp/coco/` directory of your computer as image files. Then a separate, non-interactive GUI “`look.tcl`” will collect and display these results.

Installation

The simulator has been tested on a couple of UNIX/Linux systems and on Windows with cygwin. It requires something equivalent to *(i)* `g++` for compilation, *(ii)* `lex` and `yacc` to generate the parser of the language file and *(iii)* `Tcl/Tk` for the result viewer `look.tcl`. Get the file `coco06.tar`. Then type `tar xvf coco06.tar` which creates the subdirectory `coco06/` with the simulator in it. Type `./make` in `coco06/` which creates the executable `cococo` (and also `prae.exe` as a pre-processor).

Directories and files

This structure is currently undergoing frequent changes. Don't take serious!

coco06/	main directory
cococo	the executable
look.tcl	"GUI" (image viewer)
prae.exe	called by `cococo`, praeprocesses the language file
make	only for convenience here
d/	data used for training
src/kernel/	C source files
Makefile	type `make` to compile the program
series.h	typedef struct's: `SIMULATION SERIES SWEEP STAY COMMAND`
coco.h	typedef struct's: `STATE AREA AGENT PARAMS DATA`
coco.c	`main()`, calls `do_simulation`
relax.c .h	`do_simulation` does nested loops: series - sweep - stay - command
vehicle.c .h	memory allocation, initialization, function assignments
local.c .h	functions for local computations on neurons
utils.c .h	e.g. for matrix memory allocation
src/parser/	parser directory for the language
Makefile	called by `src/Makefile`, this makes `prae.exe`
prae.c	praeprocesses the language file
r.lex	definitions of lexical elements
r.yacc.c, .h	grammar definition (also allocates `SIMULATION AGENT PARAMS`)
v/	network description files (language)
mylanguagefile	edit your own file!
docu/	e.g. handbook.ps
/tmp/coco/	activations and weights as `pnm`-image files

Program call

In general, start the program in `coco04/` with the following call:

```
./cococo -file v/mylanguagefile
```

This will start the program with it reading the language file `v/mylanguagefile`. The program will allocate memory for the network you defined, it will run the activation & training algorithm over and over again, as you have described in your language file. Make sure a directory for the activations & weights exists (e.g. `/tmp/coco/`).

Options

`-analyze 1`	The weights will **not** be exported/overwritten into files.
`-seed n`	Initializes the random seed to number `n`.
`-set string value`	Fill-in the macro which appears as `$string` in the language file with `value`. This option invites you to write a shell script which calls the simulator with different parameters over and over again.

"GUI" (image file viewer)

After starting the program, start from a separate shell the file viewer like:

```
./look.tcl a w 0 1
```

where "`a`" means display activation files, "`w`" means display weight and threshold files, and numbers "`n`" specify the areas from which to collect these files. In the window, click the left mouse button or Return to reload the files, right mouse button to quit.

The program `cococo` writes the activations to be observed and all weights and thresholds into the directory `/tmp/coco/` (this directory is actually set in the language file, and is a variable in the file look.tcl). The file names are composed of *(i)* the beginning "`obs_`", *(ii)* the activation or weight name (a letter), or "`Th`" for thresholds, followed via underscore "`_`" by the area number(s) involved and *(iii)* the ending "`.pnm`". For example, weights W from area `0` to area `1` are called "`obs_W_1_0.pnm`". This image file can also be used to import the weights, but a file "`obs_W_1_0.ext`" with higher resolution for the weight values will actually be used. The image viewer will display below the image files their truncated names as well as the minimum and maximum values.

If you want to change the sizes for display, you can change the zoom-factor in the tcl-script file `look.tcl`. Search for the expression "zoom" within `look.tcl` and change the number behind (must be an integer).

If you want the files to be reloaded and displayed continually, switch to auto-update by editing `look.tcl`: change the initialization of the variable, `set AUTOREFRESH 0` , to: `set AUTOREFRESH 1`.

The language

A language file has three main parts, describing *(i)* global variables, *(ii)* area sizes and *(iii)* the algorithm.

Global variables

Here you define first three specific global variables and then introduce an arbitrary number of own standardized pointers to memory space.

`global {`

`iter`	is the time at which the simulation starts, usually 0. Setting might be ignored here!
`areas`	is the number of areas. For example, "`2`" will create areas `0` and `1`.
`mult`	is the multiplicity of activation values. Use "`1`" here, since more aren't supported by any functions yet.
`ptr1 " "`	here some standardized pointers to memory space are introduced, to be used in the algorithm by their name (e.g. `ptr1`). Pointers have a void `data` pointer entry and a string which can be set here. They can be useful here to set directories or files for data import or export.

`}`

Area sizes

The part introduced by "`all`" defines the areas' default sizes (areas are rectangular):

`all {`

`d_a`	is the vertically displayed size, indexed by the slow counting index.
`d_b`	is the horizontally displayed size, indexed by the fast counting index.

`}`

Individual properties of specific areas, for example for area `0` may be specified, e.g.:

`area 0 {`

`d_b`	Only those properties that differ from the defaults need to be specified, but don't leave this list empty (may cause syntax problems)!

`}`

Algorithm

Structures contain all information about the algorithm in 4 nested levels. From the outside to the inside, we find first "`series`", "`sweeps`" and "`stays`" which are concerned with repetitions, time-, area- and neuron selection. The inner level, "`commands`", does essentials like taking data, computing neuronal activations or doing the learning.

1. "series" repeat over and over again, e.g. for each time picking up a new data point. The syntax to repeat ilen times is:

   ```
   series ( ilen ) {
   }
   ```

2. “sweeps” relaxate in time, i.e. with the one data point, update the neurons possibly a couple of times. The syntax to iterate (relaxate) from time begin to time end while visiting the areas (“stays”) in the area_order given. is:

   `sw ( begin ; end ; area_order ) {`
   `}`

3. “stays” visit a given area. The syntax to visit neurons on area no. area in the neuron_order given is:

   `area ( neuron_order )`

4. “commands” compute a “target” value on a neuron, using a selected “function”, “source area(s)”, “source value(s)” and optional parameters. The syntax is:

   `{ target ; function ; source area(s) ; source value(s) ; parameters }`

Note that since `stays` are nested within `sweeps`, first all areas within a `sweep` are visited at relaxation time 0, and only then all areas at time 1 and so on.

Find out more about these structures in the header file `series.h`. Starting from the back, you will find that each outer structure has (a) pointer(s) to its “nextmost” inner structure as well as an `int`eger telling how many of those inner structures are contained. From outside to the inside we have the structures: `SIMULATION`, `SERIES`, `SWEEP`, `STAY`, `COMMAND`. The outermost, `SIMULATION` structure is a cover to allow for several `SERIES`, but it is not reflected in the language.

Find out more about the use of these structures in the program file `relax.c`. There, from the outermost to the innermost (in `relax.c` from back to front), each structure is worked over to work over the structures contained within. The functions are: `do_simulation`, `do_series`, `do_sweep`, `do_stay`, `do_command`.

The area_order argument of a `sweep` is usually “`order`” so that `stays` are selected in order. The arguments “random” or “propto” choose the `stays` randomly. We need this only for a Glauber dynamics in an attractor network with several layers (see `relax.c` if you want to use these).

The neuron_order argument of a `stay` is usually “`o`” to select the neurons in order, “`r`” to select them randomly or “`s`” to shuffle, i.e. to select them randomly in a way that each comes once. Here, a command is done locally on one neuron.

The neuron_order parameter “t” (for total) of a `stay`, however, determines that a command is only done once on a whole area. This is appropriate for some functions, e.g. a non-local function to find the maximum-active neuron. Its use therefore implies the use of an appropriate function in the respective `command`.

The area_order parameter “alltime” of a `sweep` even determines that a command is done only once on an area for a whole `sweep`, i.e. at all time steps of that relaxation and, of course, for all neurons in the respective areas. Appropriate functions then have to be used in all commands under that `sweep`. This is convenient for functions that assign training data to the neural activations or for functions to export the activations for display. The relaxation duration parameters begin and end are internally

passed on as parameters, and the `neuron_order` parameters of the `stays` are ignored here.

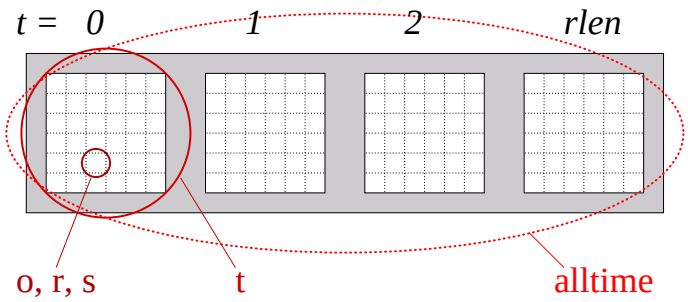

Figure 2: Function Scopes. Each of the 4 larger squares shows the neurons of one square area at one relaxation time step, in the way it is also displayed in the GUI look.tcl. 4 relaxation steps are performed, from $t = 0$ to $t = rlen$. The circles denote one invocation of the corresponding command function: if the `stay`'s neuron_order argument is "`o`", "`r`" or "`s`", then the function is executed for every neuron. If it is "`t`", the function will be executed only once for an area. If the `sweep`'s area_order argument is `alltime`, then the command function is invoked only once during a whole relaxation.

Commands Only the targets are updated within a command. Every `command` has a neural activation as target which is denoted by a capital letter from `A` to `Z`. Additionally, after a "**,**" (comma), a standardized pointer, or several separated by a "+" symbol may be used, e.g. for weights. The pointers allow read and write access by the command function and they may have been introduced already in the `global` section of the language file in which case their string argument is already initialized.

The function is what computes the target. Its string name is similar to the C-procedure name. Function keys in the C-program denote whether a function is used for 's'=single ("o", "r" or "s") neuron update, for 't'=total area update or invoked in a 'a'=alltime manner. For sloppy use, 'v'=versatile (total or alltime) or 'n'=nevermind (single or total or alltime) are also supported.

Source areas are denoted by the area numbers from which to take the source values which are activations (letters from `A` to `Z`). Several source areas can be separated by a "," or by a "+", and the same structure of "," and "+" must then be reflected in the source values so that they are taken from the correct areas. If the source area field is left empty, then the area is taken from the `stay`. If the source value field is empty, then it is the same as target.

Some functions use optional parameters which are all of data type "double". They are arranged in a matrix named "quantums" in the C-code, such that, for example, `quantums[0][0]` and `quantums[1][0]` have to be separated by a "," whereas `quantums[0][0]` and `quantums[0][1]` have to be separated by a "+" in the command.

Controls

The language support the control structure `if (argument)` which can be placed before a `series`, a `sweep` or a `stay`, but only one at each position, i.e. no directly nested controls! If takes either a single integer `argument` (that is usually used as a Macro, see below), in which case it is TRUE if the argument is larger than zero. If it has two integer arguments, then one is usually a Macro integer, the other is the keyword "iter" denoting the iter'th iteration of the whole series. These two integers are compared via the symbols "=", "<", ">" or "%". The modulo sign "%" leads to TRUE if the left integer modulo the right integer is zero, i.e. if the left is a multiple of the right argument. See the fi le `parser/r.yacc.c` for details.

Macros

It is possible to assign a string a value and to obtain that value further down in the language file from the string. The syntax is:

```
set string value
```
(the "`set`" must be at the beginning of a line!)

The value can then be obtained via

```
$string
```

The string may also be defined as an option to the program call (see above). Then, later assignments in the language file will be ignored.

Comments

Use them frequently in the language file. They have the form

```
/* This is a comment. */
```

In order to comment-out larger regions (including `/* */`-comments) use

```
[
out-commented code
]
```

Each of the comments cannot be nested. Beware of "`[`", "`]`" signs within comments!

Emacs and color

It is helpful to mark pieces of the language file with color, but the editor `emacs` doesn't automatically save the color information. Only if you switch to "enriched" mode, then `emacs` writes files with additional information (similar to `html`) which, however, the program cannot parse. Thus, we will maintain two files, the actual language file and another with "enriched" information which should be given the ending `.enr`.

In order to ease the work of: saving – switching mode – saving – switching mode back, you should copy and paste the emacs-lisp code in `tools/.emacs` into your `.emacs` file (in your home directory). Then, typing `ALT_x save-enriched` (meaning `ALT` and `x` simultaneously), makes emacs write the two files as desired. The one with ending `.enr` has the color information and should be loaded into `emacs`, the one without the ending is plain and should be given as file argument to the program `cococo`.

Help

There is no built-in help, but most functions are well documented in the C-code.

Find a function

Use UNIX functions to find a C-function in the `src/` directory. Type:

`grep` *function_name* `*.c`

When you know the file name type:

`less` *file_name.c*

Then use the search functionality of `less` by typing:

/ search_word

Debugging

- Parser level: The language file is reproduced on program output so that you can check whether it has been correctly parsed.
- Language level: Usually, a language file isn't right at the first try. Think about the order of calls, the times within relaxations that a variable is available, boundaries, function parameters, exceptions (`stay` has order "`t`" for a "total"-function, etc.). Read the comments and the code of the functions which are used.
- Program level: If you have a segmentation error in an unknown function, the following will create additional debugging output which will lead you to it. Edit the file `coco.h` and change in the line

  ```
  #define REPORT_ERR 0
  ```

  the "`0`" to a "`1`". Don't forget to re-compile.
- Function level: Debug the C-code of an identified function using `fprintf(stderr, ...)`.

Maintenance

Frequently, functions are added or changed, as well as structural parts which may affect the language. So if you install the newest version, your older language files may have to be updated. If you want additions or modifications to the C-code to be incorporated into the simulator, then email these together with a demo language file to the maintainer at <Cornelius.Weber@sunderland.ac.uk>.

Introduce a new function

If you write a new function for use in the language, you should first consider which class of functions it should belong to: `local-`, `feed-`, `total-`, `weight-`, `observe-` or `data-` functions. The choice depends on the variables, parameters or data this function must have access to and on its usage. Generally, choose the simplest possible class!

As an example, let us assume we want to introduce a function which returns the sum of two (activation) values on one node (neuron). This is, of course, a prototypical example of a `local-`function (if these two values are not to be taken at different relaxation time steps). The following steps have to be done:

0. Before you implement the function, test-write the whole language file which includes it in order to see, whether the concept works. Our example function may be used like:
   ```
   {T; local_sum; , ; R, S; }
   ```
   On all neurons of the area (given in the `stay`), the value $R + S$ shall be written to T.

1. Write the commented procedure, here in the file `local.c`:

   ```
   /****************************** local_sum **********************/
   /* Returns the sum of two arguments. Does not use parameters.    */

   double local_sum (double *par, double val1, double val2) {

       return (val1 + val2);
   }
   ```

2. Publish the function prototype in the header file `local.h`:

   ```
   double local_sum     (double *par, double val1, double val2);
   ```

3. Add the function to the function table in `vehicle.c`:

```
if  (!strcmp (cmd->func_name, "local_sum"))
    cmd->localfunc = local_sum;
```

4. Recompile, done.

An example

View the example language file `v/HopfieldContinuous(.enr)` with `emacs`. Widen the editor to accommodate ~ 140 characters in one line. Make sure the directory `/tmp/coco/` exists.

Run `cococo -file v/HopfieldContinuous`

and then (from another terminal) `look.tcl a w 0`.

About

This simulator was developed to bridge the gap between pure C-code that becomes messy over time and simulators (such as SNNS) which restrict the user too much. It is meant as a scheme to organize new C-code that piles up over further development. New versions are not compatible with older ones, sorry.

The need for it arose from testing new neural network learning algorithms with different architectures, in order to explain cortico-cortical connections (hence the name). See the publication: Emergence of modularity within one sheet of intrinsically active stochastic neurons. C. Weber and K. Obermayer. Proc. ICONIP, 732-737 (2000).

Correspondence: Dr. Cornelius Weber, Room 0.318, Frankfurt Institute for Advanced Studies, Johann Wolfgang Goethe University, Max-von-Laue Str. 1, 60438 Frankfurt am Main, Germany. Tel: +49 69 798 47536. Fax: +49 69 798 47611.
WWW: http://fias.uni-frankfurt.de/∼cweber/
Email: c.weber@fias.uni-frankfurt.de