Nemo user manual
v0.5.1

Andreas Fidjeland

July 27, 2010

1 Overview

nemo is a library for discrete-time simulation of spiking neural networks. It runs simulations on CUDA-enabled GPUs (device generation ≥ 1.2). The library provides a C++ and a C interface.

2 Installation

2.1 General dependencies

- CUDA toolkit ≥ 2.1. The SDK is not required.
- Boost

Note that the CUDA compiler driver `nvcc` is not compatible with all versions of `gcc`, especially newer versions of `gcc`.

For Boost, mostly header-only libraries are required (`shared_ptr`, `tuples`, `random`, `date_time`), so a full install/build is generally not required. Note, however, that some versions of Visual C++ requires `date_time` to be compiled as a proper library. The `date_time` library is only required for simulation timing, and can be disabled in the cmake configuration stage via the `INCLUDE_TIMING_API` variable.

2.2 Unix

The source package can be built using cmake. The basic procedure is

```
cd <nemo-directory>
mkdir build
cd build
cmake ..
```

```
make
sudo make install
```

By default this installs headers to `/usr/include/nemo` and the library to `/usr/lib`.

2.3 Windows

A precompiled library is available (NSIS installer). Both header and library files are stored in `c:\Program Files\nemo-<version>`. To use, make sure this directory is on the include path and on the library path and then include either `nemo.hpp` or `nemo.h`.

The source package can be built by using `cmake-gui` to generate an MSVC project file, and then building from within MSVC:

1. start `cmake-gui`
2. enter the source and target directories (these should be different)
3. configure
4. generate
5. open the resulting project file and build from within visual studio

Builds via cygwin or msys/mingw have not been tested.

3 Usage

3.1 General

Both the C++ and C interfaces follow the same general usage pattern:

1. Create a network object (`nemo::Network`) and add neurons and synapses
2. Create a configuration object (`nemo::Configuration`) and configure as appropriate
3. Create a simulation object (`nemo::Simulation`) from a network and a configuration and run the simulation

For the C++ API include the header file `nemo.hpp`. Errors are reported via exceptions, all of which are subclasses of `std::exception`.

For the C API, include the header file `nemo.h`. In the C API the network, configuration, and simulation objects are controlled via opaque pointers. The objects are generated with methods `nemo_new_x` (`x` = `network`, `configuration`, or `simulation`), and should be explicitly destroyed with the corresponding methods `nemo_delete_x`. The class methods are accessed with functions with names prefixed with `nemo_`, all of which take the relevant opaque pointer as the first parameter. The C API communicates errors via error codes. Most functions return a value of type `nemo_status_t`, which will be `NEMO_OK` if everything went fine and some other value otherwise.

The header files `nemo.hpp` and `nemo.h` document the API functions in more detail. The files in the example directory in the distribution shows how to use the library.

3.2 Matlab

The Matlab interface closely mirrors the underlying C++ interface. The three objects nemoNetwork, nemoConfiguration and nemoSimulation corresponds to the relevant C++ objects, and provides the same methods, with only small variations in usage. The class methods are documented using Matlabs help system, i.e. via calls such as `help nemoNetwork` and `help addNeuron`.

Errors in the simulation backend result in Matlab errors, (i.e. as when `error` is called in a script).

The Matlab path must contain the directory with the m-files defining the three classes and the MEX library that interfaces with libnemo. On Windows this directory defaults to `C:\Program Files\nemo-<version>\Matlab`, and on Linux to `/usr/share/nemo/matlab`. Use `addpath` from within Matlab to set the path.

Additionally, the nemo library and the CUDA runtime library needs to be on the system path (which is different from the Matlab path). If this is not the case Matlab will issue a rather unhelpful message about the MEX-file being invalid.

3.3 Simulation model

The simulation is discrete-time with a fixed one millisecond step size. Neurons are based on the Izhikevich neuron model [1]. Within each step the following actions take place in a fixed order:

1. Compute accumulated current for incoming spikes;
2. Update the neuron state;
3. Determine if any neurons fired. The user can specify neurons which should be forced to fire at this point;
4. Update the state of the fired neurons
5. Accumulate STDP statistics, if STDP is enabled

The state update step uses the Euler method with a step size of 0.25ms.

The weights are stored in a fixed-point format internally for performance reasons. The specific fixed-point format in use depends on the range of weights in the input network. The current accumulation uses saturating arithmetic to avoid overflow. Neuron parameters are stored in single-precision floating point.

The maximum delay is currently limited to 64ms.

3.4 STDP model

`nemo` supports spike-timing dependant plasticity, i.e. synapses can change during simulation depending on the temporal relationship between the firing of the pre- and post-synaptic neurons. To make use of STDP the user must first enable STDP globally by specifying an STDP function, and then enable plasticity for each synapse as appropriate. A single STDP function is applied to the whole network.

Synapses can be either potentiated or depressed. With STDP enabled, the simulation accumulates a weight change which is the sum of potentiation and depression for each synapse. Potentiation always moves the synaptic weight away from zero, which for excitatory synapses is more positive, and for inhibitory synapses is more negative. Depression always moves the synapses weight towards zero. The accumulation of potentiation and depression statistics takes place every cycle, but the modification of the weight only takes place when explicitly requested.

Generally a synapse is potentiated if a spike arrives shortly before the postsynaptic neuron fires. Conversely, if a spike arrives shortly after the postsynaptic firing the synapse is depressed. Also, the effect of either potentiation or depression generally weakens as the time difference, *dt*, between spike arrival and firing increases. Beyond certain values of *dt* before or after the firing, STDP has no effect. These limits for *dt* specify the size of the STDP window.

The user can specify the following aspects of the STDP function:

- the size of the STDP window;
- what values of *dt* cause potentiation and which cause depression;
- the strength of either potentiation or depression for each value of *dt*, i.e. the shape of the STDP function;
- maximum weight of plastic excitatory synapses; and
- minimum weight of plastic inhibitory synapses.

Since the simulation is discrete-time, the STDP function can be specified by providing values of the underlying function sampled at integer values of *dt*. For any value of *dt* a positive value of the function denotes potentiation, while a negative value denotes depression. The STDP function is described using two vectors: one for spike arrivals *before* the postsynaptic firing (pre-post pair), and one for spike arrivals *after* the postsynaptic firing (post-pre pair). The total length of these two vectors is the size of the STDP window. The typical scheme is to have positive values for pre-post pairs and negative values for post-pre pairs, but other schemes can be used.

When accumulating statistics a pairwise nearest-neighbour protocol is used. For each postsynaptic firing potentiation and depression statistics are updated based on the nearest pre-post spike pair (if any inside STDP window) and the nearest post-pre spike pair (if any inside the STDP window).

Excitatory synapses are never potentiated beyond the user-specified maximum weight, and are never depressed below zero. Likewise, inhibitory synapses are never potentiated beyond the user-specified minimum weight, and are never depressed above zero. Synapses can thus be deactivated, but never change from excitatory to inhibitory or vice versa.

References

[1] E. M. Izhikevich. Simple model of spiking neurons. *IEEE Trans. Neural Networks*, 14:1569–1572, 2003.