CxSOM

Hervé Frezza-Buet
Herve.Frezza-Buet@centralesupelec.fr

Contents

1 Introduction

The `CxSOM` software suite enables to model consensus based multi-SOMs, explored in the BISCUIT team of teh Loria lab.

The current documentation is at work, and mainly technical stuff are reported here. See the README at the package root for an introduction, as well as examples.

2 The model

The computation enabled by `CxSOM` consists of successively updating data, keeping trace of the data values in history files.

2.1 Data instances (DI)

The elementary piece of data handled in `CxSOM` is called a *data instance* (*DI*). Each of the computed DIs ends as a record in a file, once it has been computed.

DIs are denoted by a triplet $[T,\ X,\ t]$, that reads as "the instance of variable X hosted by the timeline T at the time instant t of that specific timeline". Only elementary $[T,\ X,\ t]$ DIs matter, the concepts of *variable*, *timeline* and *time instant* only serve for describing the computation.

Each DIs is a value that is stored in a file once definitively computed. A value has a type, that can be:

- `Scalar`: A floating point value, usually in $[0, 1]$.
- `Pos1D`: A value in $[0, 1]$ corresponding to a position in a 1D map.
- `Pos2D`: A value in $[0, 1]^2$ corresponding to a position in a 2D map.
- `Array=`n: A value in $\mathbb{R}^n$, usually $[0,\ 1]^n$, $n \in \mathbb{N}^\star$.
- `Map1D<`$\mathcal{X}$`>=`n: A 1D map of type $\mathcal{X}$, i.e. a value in $\mathcal{X}^n$, $n \in \mathbb{N}^\star$.
- `Map2D<`$\mathcal{X}$`>=`n: A 2D squared map of type $\mathcal{X}$, i.e. a value in $(\mathcal{X}^n)^n$, $n \in \mathbb{N}^\star$.

where type $\mathcal{X} \in \{$`Scalar`, `Pos1D`, `Pos2D`, `Array=`$k\}$.

In `CxSOM`, the type is associated to a variable, i.e. all the $[T,\ X,\ \bullet]$ are DIs with the same type.

Every DI is also doted with a status variable $\langle[T,\ X,\ t]\rangle \in \{$`busy`, `ready`$\}$:

- `ready`: The computation of the value of the DI is definitively done. The DI value will not change anymore.
- `busy`: The definitive value of the DI is still to be determined.

More generally, in the following, $\langle x\rangle$ reads as "the status of x".

2.2 Updates

An *update* of some DI is the (re)computation by the `CxSOM` computer of the value of that DI. There are at most two updates for a DI, one for the first update (initialization) that is optional, and one, mandatory, for other updates.

2.2.1 An update is a function taking arguments and computing a result

An update u for a DI $[T,\ X,\ t]$ is made of:

- The decription of some computation, i.e. the function called to realize an update.
- The result res_u, i.e. $\text{res}_u \stackrel{\text{def}}{=} [T,\ X,\ t]$ itself, that is computed by the function of the update.
- Other DIs, thats serve as arguments to the function (their value is read when the function is called). Among arguments, a distinction is made between *in-arguments* and *out-arguments*:

– The in-arguments in_u of the update u such as $\text{res}_u = [T,\ X,\ t]$ is the set of DIs used as arguments of u of the form $[T,\ \bullet,\ t]$. All the DIs handled in the simulation of the form $[T,\ \bullet,\ t]$ define a *time step* $\mathcal{S}_T^t$ of the simulation, so the in-arguments are the DIs used for the computation of an update which belong to the same timestep as the result.

– The out-arguments out_u of the update u are the other DIs used as arguments for the computation of the update.

2.2.2 Datation and relaxation

Before it has a definitive value (i.e. before being in the `ready` status), the value of a DI may change several times. Indeed, all DIs in a timestep $\mathcal{S}_T^t = [T,\ \bullet,\ t]$ are updated until all of them get `ready`, and during that process, the values of the DIs may be recomputed several times. In order to handle the dependancies of the DIs within a timestep, we need to timestamp the values of the DIs, in order to determine those of them that have to be updated. Timestamp play the role of file dates in makefile, when the date of a target is compared to the date of its dependancies. Here, in order to know if an update u needs to be performed, we have to consider two things:

- Do we have $\forall [T,\ X,\ t] \in \text{out}_u,\ \langle [T,\ X,\ t] \rangle =$ `ready`? If not, the update cannot be done, it is considered as *impossible*.
- Have the in-arguments been updated since the last computation of the result? If the answer is yes, the result needs to be recomputed.

To compute the second conditon, every DI comes with a *datation* denoted by $\text{d}_{[T,\ X,\ t]} \in \mathbb{N}$.

An update then stores, for each of its in-arguments, the datation it had at the last computation of the result. When the simuator considers an update, it compares the current datation of the in-arguments to the ones stored by the update. If some are newer (or if the result has never been computed so far) the result is recomputed, and

$$\text{d}_{\text{res}_u} = 1 + \max_{[T,\ X,\ t] \in \text{in}_u} \text{d}_{[T,\ X,\ t]} \tag{1}$$

Considering an update for computation can thus lead to the following status of the update u, denoted by $\langle u \rangle$:

- $\langle u \rangle =$ `impossible`: $\langle \text{res}_u \rangle =$ `busy`, computation is not feasible yet, since some out-arguments are `busy`.
- $\langle u \rangle =$ `uptodate`: $\langle \text{res}_u \rangle =$ `busy`, nothing changed in the in-arguments input dates from last update, or the new value was not a significant modification. The datation d_{res_u} has not been modified.
- $\langle u \rangle =$ `updated`: $\langle \text{res}_u \rangle =$ `busy`, the computation has modified the value of res_u significantly.
- $\langle u \rangle =$ `done`: $\langle \text{res}_u \rangle =$ `ready`, the computation has modified the value res_u definitively.
- $\langle u \rangle =$ `none`: Update status is not determined yet (used for initialization only).

The datation mechanism enables to control the update of all the DIs in a given timestep, i.e. all the DIs like $[T,\ \bullet,\ t]$ for the time instant t of the timeline T. When no more significant writes of the results can be done, the whole timestep is stable. So the relevance of the datation mechanism is to enable a *relaxation* of the DIs inside a timestep until stabilization of all of them is reached.

2.2.3 Update computation cycle

When a specific update is defined in `CxSOM` (as average, learning, etc.), the update is inherited from a base `cxsom::update::Base` class. In the subclass, the specificity of the computation has to be implemented, while the base class handles the details of the full update cycle, as datation and status. Algorithm 1 shows the cycle, the `on_*` calls are the method that subclass need to override in order to implement a specific update computation.

2.3 Timesteps and tasks

2.3.1 A computational structure for timesteps

In the simulator, the computation of the updates are put in a pool task and computed in parallel. However, since relaxation is related to the computation of updates belonging to the same timestep, this task management needs an intermediate level, which is a computational internal structure representing the timesteps.

Let us recall the notation $\mathcal{S}_T^t \stackrel{\text{def}}{=} \{[T',\ X',\ t'] \in \text{simulation} \mid T' = T \wedge t' = t\}$ for the timestep at instant t in the timeline T. For each $[T,\ X,\ t] \in \mathcal{S}_T^t$, the internal structure used in the simulator gathers at least an *usual update*, and an optional *initialization update*. The initialization update, if present, is used for the first feasible setting of the value of $[T,\ X,\ t]$ instead of the ususal update. Otherwise, the usual update is used throughout the successive setting of the value of $[T,\ X,\ t]$ during the relaxation.

In order to avoid further confusions, let us denote by $\overline{u}$ an update handled by a time step $\mathcal{S}_T^t$: it can be either a single update u (so u is the usual update) or a pair (u, u') if an initialization update is defined (then u' is the initialization update). In the case for which $\overline{u} = (u, u')$ is a pair, $\text{res}_u = \text{res}_{u'}$ obviously stands, so $\text{res}_u = \text{res}_{u'}$. In the case of a pair as well, the simulator uses u' until it gets a first computation of $\text{res}_{\overline{u}}$, and then it will switch to the use of u for next computations of $\text{res}_{\overline{u}}$.

As at least a call to the usual update u is required to be done for $\overline{u}$, we have particularized the status returned by initialization updates in algorithm 1, so that the evaluation of an initialization update can never lead to the definitive computation of res_u during the relaxation of the timestep.

2.3.2 Timestep status

The timestep structures $\mathcal{S}_T^t$ at the simulator level have a status

$$\langle \mathcal{S}_T^t \rangle \in \{\texttt{blocked},\ \texttt{relaxing},\ \texttt{checking},\ \texttt{done}\}.$$

The status $\langle \mathcal{S}_T^t \rangle$ depends on the status of the updates of the timestep, observed when the updates are realized. The meaning of the $\langle \mathcal{S}_T^t \rangle$ is:

- $\langle \mathcal{S}_T^t \rangle = \texttt{blocked}$: The timestep is blocked due to impossible updates.
- $\langle \mathcal{S}_T^t \rangle = \texttt{relaxing}$: The timestep is under unstable computation.
- $\langle \mathcal{S}_T^t \rangle = \texttt{checking}$: Every update seem stable, we are checking this.
- $\langle \mathcal{S}_T^t \rangle = \texttt{done}$: The timestep is done, all updates $\overline{u}$ have lead to $\text{res}_{\overline{u}} = \texttt{ready}$ and have quit the simulator.

Algorithm 1 `One update cycle for update` u

Require: a boolean attribute `out_ok`, telling if all out-args have been read successfully previously.
Require: a boolean attribute `is_init`, telling if the update is an initialization.

1: **if** $\langle \text{res}_u \rangle = \texttt{ready}$ **then**
2: **return** `done`
3: **end if**
4: on_computation_start ()
5: **if** $\neg$`out_ok` **then**
6: // out-arguments need to be read.
7: **for all** $[T,\ X,\ t] \in \text{out}_u$ **do**
8: **if** $\langle [T,\ X,\ t] \rangle = \texttt{ready}$ **then**
9: on_read_out_arg $([T,\ X,\ t])$
10: **else**
11: `out_ok` $\leftarrow$ false
12: on_read_out_arg_aborted ()
13: **return** `impossible`
14: **end if**
15: **end for**
16: `out_ok` $\leftarrow$ true // All in-args have been successfully read.
17: **end if**
18: **for all** $[T,\ X,\ t] \in \text{in}_u$ **do**
19: // Datation issues are handled in this loop, but not detailed here.
20: on_read_in_arg $([T,\ X,\ t])$
21: **end for**
22: **if** none of the in-arguments have been updated since last cycle **then**
23: **if** `is_init` **then**
24: **return** `updated` // updates used as initialization are never considered as up-to-date.
25: **else**
26: **return** `uptodate`
27: **end if**
28: **end if**
29: // From here, we know that the result has to be recomputed.
30: $x \leftarrow$ `none` // This is the status we plan to return, it is set next.
31: $\alpha \leftarrow$ on_write_result (res_u) // α is a Boolean telling if the value change is significant.
32: **if** we have just written a new significant value in res_u **then**
33: $x \leftarrow$ `updated`
34: **else**
35: **if** `is_init` **then**
36: **return** $x \leftarrow$ `updated` // Even not significant changes are considered as an actual update for initialization updates.
37: **else**
38: **return** $x \leftarrow$ `uptodate`
39: **end if**
40: **end if**
41: // Before returning, we have to check if in-arg may change in the future
42: **if** all in-arguments are `ready` **then**
43: **if** `is_init` **then**
44: $x \leftarrow$ `updated` // As u an init, we return `updated` since further change can be consider by the usual update comming next for updating res_u.
45: **else**
46: $x \leftarrow$ `done` // We have definitively computed the result...
47: $\langle \text{res}_u \rangle \leftarrow$ `ready` // The result DI value is locked.
48: **end if**
49: **end if**
50: **return** x

2.3.3 Timestep update queues

The timestep structures $\mathcal{S}_T^t$ at the simulator level are doted each with 4 update queues, where the updates $\overline{u}$ are stored. Each the $\overline{u} \in \mathcal{S}_T^t$ belongs to one of the 4 queues of $\mathcal{S}_T^t$.

When a timestep is asked by the simulator to provide computation, it extracts updates from some of the queues. When the update is performed by the simulator, it is given back to the timestep, with the return status resulting from algorithm 1. According to this status, the update is stored in the appropriate queue, and some supplementary transfers from one queue to another queue may happen. This will be detailed further, as the *timestep state machine*.

For now, let us present the 4 queues used by a timestep $\mathcal{S}_T^t$.

$$
\begin{array}{lcl}
[\mathcal{S}_T^t]_{\text{unstable}} & : & \{\overline{u} \in \mathcal{S}_T^t \mid \langle\overline{u}\rangle \text{ needs to be known.}\} \\
[\mathcal{S}_T^t]_{\text{impossible}} & : & \{\overline{u} \in \mathcal{S}_T^t \mid \langle\overline{u}\rangle \text{ has been detected as } \texttt{impossible}\} \\
[\mathcal{S}_T^t]_{\text{stable}} & : & \{\overline{u} \in \mathcal{S}_T^t \mid \overline{u} \text{ have been seen stable for the first time are here.}\} \\
[\mathcal{S}_T^t]_{\text{confirmed}} & : & \{\overline{u} \in \mathcal{S}_T^t \mid \overline{u} \text{ for which stability is confirmed.}\}
\end{array}
$$

2.3.4 Providing jobs

A timestep is periodically asked by the simulator to provide tasks, i.e. to offer updates that will be evaluated by the simulator. These updates are extracted from the timestep and inserted in a pool task, waiting for execution. The timestep keeps trace of their existency while they are outside, waiting for being executed.

So when a timestep $\mathcal{S}_T^t$ is asked for new updates to be done, it provides (and extracts) :

- all the $\overline{u} \in [\mathcal{S}_T^t]_{\text{unstable}} \cup [\mathcal{S}_T^t]_{\text{stable}}$ if $\langle\overline{u}\rangle \in \{\texttt{relaxing}, \texttt{checking}\}$
- nothing otherwise.