Tensor Store

Architecture

Start with a basic in-memory database. This could be a custom database, or an off the shelf product such as:

- Redis
- Dragonfly
- KeyDB
- ValKey

All of these support both simple key/value stores and lists.

There doesn't really seem to be any native support for tensors, except as either blobs or slobs. May need to write an extension to cover this.

There will be (at least) four collections:

- `status::` – Keys with this prefix are for communicating system status among processes.
- `metadata::` – This records mostly static information about the model.
- `cpt::` – This is the cache of the most recent CPTs read by the E-step/scoring algorithm.
- `xtabs::` – This is the data cache for the M-step of the EM algorithm.
- `deviance::` – Used for storing convergence information.
- `pvec::` – Used for caching parameter vector values.

There are three types of processes (which might be distributed over several machines): *E-workers* – which are responsible for scoring students, *M-workers* – which are responsible for updating CPTs and a *supervisor* which is responsible for controlling the other processes. In general, the E-workers take data from the `cpt::` collection, and send output to the `xtabs::` collection. The M-workers take data from the `xtabs::` and output to the `cpt::` data.

Status collection.

These keys will be proceeded with `status::` to identify them.

- `status::e-step`, `status::m-step` – this are flags which have the value "Running" or "Done", or "Error" and are controled by the modules.
- `status::signal` – This has possible value "Run", "Stop" and "Halt". This allows an external process to indicate that the processes should either stop when current work is completed, or immediately halt.
- `status::iterations` – This is a cycle counter for the overall cycles.
- `status::deviance` – This is updated every cycle during the convergence check.
- `status::deviance_components` – This is a list of deviance values from each CPT calculation. The total deviance is its sum.
- `status::convergence` – This has three states "Converged", "Not yet converged", and "Did not converged" or "Error".
- `status::subjectrecords` – This is a stream of the student data (subject ID plus a list of tasks/procedures and outcomes) which the E-step workers can draw from.
- `status::components` – This is a stream of CPT identifiers which the M-step workers can draw from.

Metadata

This is for information which is fairly static and doesn't change much. The most important element is `metadata::competencies` which defines the dimensions of the tensor. Its value should be a list (or array) of lists (or arrays).

```
metadata::competencies =
((var1 state1.1 state1.2 state1.3 ...)
 (var2 state2.1 state2.2 state2.3 ...)
 ...)
```

Let K be the length of the lists, i.e., the number of dimensions of the master tensor. The length of each element (minus 1) gives the size of each tensor dimension, so the master tensor has dims $(M_1, M_2, ..., M_K)$.

Some additional metadata fields:

- `metadata::model` – An identifier for the model
- `metadata::version` – A version number (string)
- `metadata::timestamp` – A date time for this particular version.

Representing potentials

A potential is an unnormalized probability distribution. It has a *frame*–an ordered list of variables it is defined over–and tensor, whose dimensions match the list of variables. Potentials are combined by element-wise multiplication, but this requires the two potentials to be over the same frame. To facilitate this, all potentials in the system use `metadata::competencies` as thier frame. Call this the *full frame* of the model.

Often, not all of the dimensions are relevant to a potential. Let $\phi[\cdot]$ be a potential, and let $\mathcal{F}(\phi[\cdot])$ be its frame. This can be represented as a logical vector of length K which is true if the corresponding variable is relevant and false if it is not. In the tensor representation, if $\mathcal{F}(\phi[\cdot])[k] = 0$, then set $M_k = 1$. For a tensor in compact form, $\mathcal{F}(\phi[\cdot])[k] = (M_k > 1)$. A tensor in compact form can be extended to the full frame by replicating the size one dimensions, up to the full size. Both `PyTorch` and `TensorFlow` do this during multiplication: they call it *broadcasting*.

The process to move a potential over the full frame to the compact representation is *marginalization*. This is accomplished by summing over the unneeded dimensions. Where possible, tensor should be passed around in compact form.

Evidence tensors have $K+1$ dimensions: the observable variable is added as the last dimension. In the inference algorithm, a sub-tensor (over the full frame) is chosen based on the value of the observable. When passed back to the M-step, the potential should be reduced to the compact form (over the competency variables), and the value of the observable variable passed along with it (the complete tensor would simply have zeros in the other sub-tensors, so this representation is more compact).

CPT collection

The tensors in this collection are named either `cpt::cm_`*groupid* or `cpt::em_`*taskid*`=`*obsval*. The first are competency tensors and the the latter are evidence tensors, with `obsval` representing the value of the observable.

The competency tensors are always over the full frame. The are identified by a group identifier (e.g., "PGY1", "PGY2", …). Note that these tensors may be themselves represented as products of tensors over smaller frames, but it is the responsibility of the M-workers to build the complete potential.

In the simplest case, there is only one observable with each task, so the *taskid* just represent tasks. If there is more than one observable per task, then these need to be combinined into a superobservable which ranges over the states of the cross product of the other observables. The M-workers again would be responsible for putting them together.

The evidence potentials actually have $K + 1$ dimensions, where the last dimension is the observable variable. However, in the collection they are split into a number of substreams

adding the observable value, so that the correct sub-tensor can be directly pulled from the memory story.

The xtabs

Performing the scoring creates a subject-specific tensor over the full frame. The expected value for the sufficient statistic for the competency tensors is the sum over all of the subjects in each group of these score tensors. As the score tensors are potentially calculated by different workers, they need to be gathered together. This is done through the `xtabs::cm_`*groupid*, which is a list. Each E-worker pushes the score tensor onto the list and the M-worker responsible for that group will eventually sum all of those tensors.

The sufficient statistics for the evidence tensors is a bit different as it involves the observed value. This can be thought of a collection of sub-tensors over the full frame, one for each observable variable. As the addition can be done over the smaller sub-tensors, there is a separate cross-tab collection for each observable variable. Therefore, the key for the evidence tensor cross-tabs is `xtabs::em_`*taskid*`=`*obsval*. Note also, that the evidence tensors often don't use all of the variables in the full frame, so they should be reduced to compact form before being added to the list. The M-worker is then responsible for adding the lists for each observable and then assembling them into the full tensor over both competency and observable variables.

The E-worker needs to add an entry to the list corresponding to each task processed by the subject. The score tensor is marginalized to the frame of the evidence tensor and the entry is added to the list corresponding to the output value.

As the addition can be done in any order, it may be possible to start the addition over the cross-tabs before the scoring of the E-step is complete.

Deviance collections

Tracking the deviance over cycles is useful for diagnosing problems with convergence. Keys starting `deviance` are used for this tracking.

- `deviance::all` – This tag should be a list of overall deviances (from the supervisor).
- `deviance::cm_`*groupid* – These tags should be used for the competency tensor tables.
- `deviance::em_`*taskid* – These tags should be used for evidence tensor tables.

It could be that convergence can be judged on a CPT-by-CPT basis, stoping maximization for models which have seemingly converged.

Parameter vectors.

There is one set of parameter vectors for each CPT, either `pvec::cm_`*groupid*, or `pvec::em_`*taskid*. (This does not need to be separated by observable value). In each case, it a list of parameter vectors. After each M-step, the new values are pushed onto the list, so that the most recent value is the head of the list.

The processes

There is a single supervisor process and any number of E-workers and M-workers who can do their thing in parallel.

The Supervisor process

The supervisor primarily deals with the status database. It has the following phases:

1. *E-step*. The supervisor refills the `status::subjectrecords` stream. The E-workers can then score all of the subjects asynchronously, adding information to the `xtabs::`. This phase ends when all student records have been processed.
2. *M-step*. The supervisor refills the `status::components` stream, and empties the `status::deviance_components` list. The M-workers can then recalculate all of the CPTs using the data from the `xtabs::` collection and putting the results in the `cpt::` variables. They also add a deviance component to the `status::deviance_components` list. The M-workers can work independently and this phase ends when they all complete their work.
3. *Convergence check*. The scheduler calculates the new deviance by summing the `status::deviance_components`. This is then compared to the value in `status::deviance`. If the difference is less than the tolerance, then the procedure halts, marking itself as converged. The newly computed deviance is used to update `status::deviance` and `status::deviance_history`. The value of `status::iterations` is incremented. If it exceeds the maximum, then the procedure halts and is marked as not converged. Otherwise, the cycle starts over with the E-step.

The `status::deviance_components` can be added in any order, so this could be built into the supervisor.

E-workers

These are responsible for scoring each subject, producing the scored tensor. The work is similar to the general scoring algorithm except that (1) there is no need to calculate statistics of the scored tensor for individual level reporting, and (2) they need to pass information back into the cross-tabs collection to feed the M-step.

The record for each subject includes: *groupid*, a list of tuples containing *taskid* and *obsval*. The algorithm works as follows:

1. *Initialize scoring tensor.* Copy the tensor at `cpt::cm_`*groupid* to be the scoring tensor.
2. *Score each task.* These can be done in any order or though grouping a bunch of things, as what is important is the product.

2.1 *Fetch Evidence Tensor.* Fetch the `cpt::em_`*taskid*=*obsval* corresponding to the evidence tuple.

2.2 Multiply (with broadcasting) the evidence tensor with the current scoring tensor.

3. After all the scores are processed, the result tensor is the scored tensor. Push this tensor onto the `xtabs::cm_`*groupid* list.
4. Again, looping through all observations in any order (or in parallel).
   4.1. Marginalize the score tensor onto the frame of `cpt::em_`*taskid*=*obsval*.
   4.2. Push the marginalized tensor onto the `xtabs::em_`*taskid*=*obsval* list.
5. Done, this worker can now process another subject.

There might be some efficiency gains by processing all tasks with the same frame at the same time, as the marginalization needs to be done only once. Also, if one frame is a subset of another, then the marginalization can be done in multiple steps.

M-workers

The worker is assigned either a competency model (`cm_`*groupid*) or an evidence model (`em_`*taskid*). The processing is similar, but different.

1. *Create cross-tab.*

1.`cm` For the competency model just sum the tensors to create the cross-tab.

1.`em` There is one xtab for each value of the observable, these should be summed separately and then joined together in the proper order.

2. *(Optional Partitioning)* The CPT might be the product of several peices over smaller frames, in which case the cross-tab needs to be marginalized onto these smaller frames.

3. *Run optimizer.* Run the optimizer to find parameters which minimize the deviance. (This might be a loop over several subcpts.) *Note bene.* The optimizer does not need to be run to convergence, several cycles might be fine. The deviance from the final cycle is saved.

There might be a schedule here where the optimizer is run for longer in later iterations.

4. *(Optional Reassembly).* If the CPT was broken into pieces, reassemble it. The parameter vector becomes the concatenation of the parameter vectors, and the deviance the sum of the deviances.

5. *Record results.*

5.1 Push the deviance onto `status::deviance_components`, and the deviance list for this CPT.

5.2 Push the parameter vector onto the appropriate list for this CPT.

5.3`cm` Replace the value of `cpt::cm_`*group_id* with the newly calculated tensor.

5.3`em` Split the tensor along the observable values and replace the `cpt::em_`*taskid*=*obsval.* records.

At this point the worker has finished with this CPT and can process another.

Discussion

Saving Tensors

Maximizing in-memory use

As you go summation.

- Add new tables to the tail of the list.
- Summer replaces `list` with `(cons (+ (car list) (cdar list)) (cddr list)`