Implementing EM algorithms

Alexandre Bouchard-Côté

April 12, 2007

Although simple conceptually, EM can be frustrating at implementation time. Some of these difficulties can be addressed using simple tricks such as modular debugging and verification of invariants. Others are inherent to the non-convex nature of the objective function and make the application of EM sometimes resembles a black art.

This document identify some of the main problems one might encounter with EM in applications.

We will often assume implicitly that the r.v. being marginalized are discrete, but many of the recommendations apply to the continuous hidden nodes setting as well.

1 Writing a correct EM implementation

The first thing to make sure is that your EM software is correct. The approach that we propose is to write, test and debug the M and E steps separately, and to test them together only once you are confident that each piece does what it should.

Instead of performing the tests described below manually, it is better to create assertions in your code that automatically check that the properties of interest hold not only during your isolated test, but also at every iteration of the full EM algorithm.

It is generally a good idea to start by writing the M step, but before doing that, it is very useful to be able to generate artificial data using known parameters.

1.1 Generating data for debugging

Start by picking a set of parameters that will induce a simple model and that will be easily identifiable. For instance, in a mixture model, start with only 2 mixtures components, and let the parameters for the mixtures components be far apart. Also, try first with short sequences in a HMM model, to avoid underflow problems initially. If you can set some of the parameters so that your model becomes something simpler, try that first. For example in assignment 4, question 3, by setting the number of states in the backbone to be one, we get a sequence of iid mixture models as a special case.

All these assumptions should be relaxed once the algorithm works on the toy data, to explore where the limitations of EM are in your model of interest. But starting with a manageable model helps very much in the initial stages and avoids many frustration: unsupervised learning problems are already hard enough, you don't want to be doubting your implementation on top of these difficulties.

For completeness, we describe how to generate samples from a directed graphical model[1]:

[1] For undirected graphical models, sampling is harder. The first thing to do is to check if the model can be reduced to a directed graphical model.

If this is not the case, one alternative is to use MCMC methods to perform *approximate* sampling. In particular, Gibbs sampling is a very popular alternative because it is often easy to implement and converges quickly to the correct sampling distribution for many models. This will be the topic of some of the later lectures in the class.

1. Pick a set of parameters Θ that fully specify the initial distribution $p_r(\cdot)$ of the root r and the conditional probabilities $p_{i|pa(i)}(\cdot|\cdot)$ of each node i given the value of its parents $pa(i)$ (these might be densities or probability mass function).

2. Pick a topological ordering of the nodes in the graph,

3. Sample the root X_r from $p_r(\cdot)$,

4. Following the topological ordering, sample node i using $p_i(\cdot|x_1, \ldots, x_k)$, where $x_1, \ldots, x_k$ are the values sampled for the k parent nodes of node i.

The distributions of the local conditional probabilities or densities p are problem specific, and very often sampling procedures are built-in (for instance, with matlab, see binornd, chi2rnd, exprnd, gamrnd, normrnd, poissrnd, trnd, unifrnd, weibrnd).[2]

1.2 Testing the M step in isolation

Write your M step modularly, a component that takes expected sufficient statistics and return parameter estimates.

Generate iid hidden states and observations using your parameters as discussed in the previous section. Estimate the average value of the sufficient statistics from this data and plug-in these sufficient statistics estimates into you M component.

Are the estimated parameters close to the true parameters? Does the estimate get better as you compute the expected sufficient statistics on more iid samples? Your code should answer "yes" to each of these questions.

1.3 Testing the E step in isolation

Now, write your E step component. It should take observations and parameters as an input, and output expected sufficient statistics. Many implementation errors come from the somewhat delicate message-passing algorithms required for large graphical models (Sum-Product or Junction-Tree).

Checking the correctness of these message-passing algorithms can be a little bit more difficult.[3] You have three main options:

1. For very small graphical model, naively sum over the exponential number of assignments for the marginalized nodes. This should be very easy to write and less error-prone than the fancy message-passing algorithms.

2. For larger graphical model, the elimination algorithm will be a more viable testing option than the brute-force method, but it is also a bit more complex to implement. It is still slightly simpler than most message-passing algorithms though.

[2] For less standard distribution, see 21 in the text, section 1.1. Another important basic sampling technique, when the inverse of the cdf can be computed or approximated analytically is the *inverse cdf technique* (many articles on that available from google).

[3] Notice that message-passing can be seen thru the lenses of Dynamic Programming. In particular, many can be implemented as recursive memoized functions. Although less efficient than directly iterating thru multidimensional arrays, these recursive functions often yield more intuitive code structure and highlight invariants better.

In general, you should never worry about (constant-factor) efficiency in the first phase of implementation, as a slow and correct algorithm is preferable to a fast and broken one. In this class, the datasets are small enough to dispel any worries on almost any constant-factor efficiency.

3. Gibbs sampling[4] is a third interesting correctness-verification alternative. Its inconvenient is that it has its own implementation difficulties (fixing the burn-in length, number of samples that should be used, etc.). On the other hand, it has the interesting property of being immune to underflow problems.

Having two different algorithms give approximately the same answer is a very comforting sanity check, especially for the Gibbs alternative, which takes a very different approach to the marginalization problem.

Make sure however that if any part of your code is shared among the two algorithms being compared, this code is also checked using another test case.

Another strategy that can be used to detect problems with message-passing algorithms (but that does not necessarily help to establish correctness) is to identify invariants is the recursions. For instance, in assignment 4, question 2-3, the following property holds for all $s, t \in \{1, \ldots, T\}$:

$$\sum_q L_t(q) M_t(q) N_t(q) = \sum_q L_s(q) M_s(q) N_s(q).$$

Here, we are using the notation M, N for the forward and backward recursions and L for the observation likelihood marginalized over the intermediate mixture indicators, as defined in the solutions. Such invariants can be derived easily by expanding out the meaning as conditional probabilities of the recursions.

1.4 Sanity checks for EM

Once the E and M steps seem to be working properly, there should not be too much debugging left, at least in theory. As a sanity check, initialize EM with the parameters you used in the previous sections, and run EM on the observations generated using these parameters. Your algorithm should converge in the vicinity of the true parameter, and the bias should decrease as more samples are used for estimation.

Another important sanity check is to verify that the expected complete log-likelihood increases at each EM iteration on the training data. Note that checking that this property holds is cheap since the computation of the expected completed log-likelihood can reuse the result of the E computation.

Next, see how much you can perturb the initial values and still get a converging algorithm. This will give you an idea on a lower bound on how accurate your initial values for parameters will need to be for your real data.

A common practice in applications is to initialize the EM algorithm for the model of interest with parameters obtained using an EM algorithm ran on a simpler model. For instance, HMM models for word alignment can only be made to work well when they are initialized with reasonable parameters obtained from a simpler, bag-of-word model.

Finally, apply EM on your real data. Keep in mind that the data that you are analyzing was *not* generated by your model. Your model is almost always broken—if it is not, it is probably intractable!—, but your hope is that it can still be useful, by exhibiting the salient features found in the data. This is where creativity, insight and intuition comes in, but also trial-and-error and some black art.

2 Some pitfalls

2.1 A few words on underflows

Arithmetic underflow is a condition that can occur when the result of a floating point operation would be smaller in magnitude than the smallest quantity representable. Even when the result of some computation can be represented, it can be the case that one of the intermediate stage of the computation underflows, for instance when the outcome of the computation is a ratio of small numbers.

[4] A MCMC approximate sampling technique that will be discussed in more details later in the class. See chapter 21 in the text.

This can often occur in message passing algorithms, even for graphical model of moderate size. A common strategy is to store the recursions of the message passing algorithm in log space. When two such quantities must be multiplied, their log space representations are simply added. When two log space quantities, say a, b, must be added, the operation is rearranged as follows:

$$\log(\exp(a) + \exp(b)) = \log(1 + \exp(b - a)),$$

and most programming environment provide numerically-robust implementation for the function $\log(1 + x)$ (e.g. Math.log1p(double x) in Java).

Another option is to use rescaling of the recursion quantities, as discussed in the text (12.7). This is slightly more involved to implement but more efficient, as log operations are much more expensive than multiplications and additions.

Note that in the assignments, the length of the chain and the initialization are picked so that these problems do not arise with double precision arithmetics.

In assignment 4, some people experienced a related problem, when computing the probability of the Poisson observations:

$$\frac{e^{-\lambda}\lambda^{y}}{y!}.$$

One way to solve this overflow problem is to take the log of this expression, so that the denominator and numerator do not overflow, and take the exponential only as a final step, after the division (subtraction in log space) brings back the quantity to a more reasonable magnitude. Note that there are efficient and numerically robust algorithms for computing $\log(n!) = \log\Gamma(n + 1)$. Another option was to call matlab's built-in function (poisspdf).

2.2 Confusing nested iteration with M step using IPF

One of the algorithm that can be used for the M step in undirected graphical models is the Iterative Proportional Fitting (IPF) algorithm, covered in the previous lectures.

The is a source of confusion due to the somewhat ambiguous definition of the IPF update equation in the text (Equation 9.61).

One might interpret the notation as defining IPF as follows: apply the updates to each clique, and then, once this is done, replace in one operation all the potentials with the new value. It turns out that this algorithm does not have the properties mentioned in the text (invariance of Z, non-divergence, etc.). The right interpretation is that the updates of the clique parameters should be done "in place", i.e. at iteration $t+1$, when the second clique say is being updated, use the parameter obtained at iteration $t+1$ for the first clique when computing the marginal, not the one obtained at iteration t.