Scaling up CELESTE

November 4, 2015

1 Introduction

This document contains some first thoughts about how to scale up the CELESTE model.

1.1 Model

The sky contains a large number (~400 million) of Celestial objects, each of which is characterized in Celeste by 32 parameters (this may change over time). In the long-run there may be trans-object parameters, such as optical information about the telescope. These parameters describe the light we expect an object to emit. Turning this light into image pixels via the telescope is the source of randomness in the model. The goal is to take the pixel images and infer the object parameters.

The image data consists of ~1 million "field" files for each of five bands. Each field is 1361 x 2048, or ~2.7 million pixels. In total, there are 1.3e13 pixels. Obviously, most celestial objects do not contribute to most pixels. We have been breaking fields up into superpixels, or "tiles" and building a many-to-many map between tiles and objects. Fields overlap so an object can appear in multiple fields. Given the parameters, each pixel is assumed to be independent.

For detailed background on the model, see Regier, Jeffrey, et al. "Celeste: Variational inference for a generative model of astronomical images." arXiv preprint arXiv:1506.01351 (2015). For some nice summary statistics, see `http://www.sdss.org/dr12/scope/`.

1.2 Optimization

CELESTE builds a catalog by fitting the celestial parameters to the images using a particular loss function called the "evidence lower bound" or "ELBO". (This special loss function allows us to measure both fit to the data and uncertainty.) The goal is to optimize this function. The steps are:

- Initialize the parameters using the existing catalog
- Repeat until convergence:

1. Using the parameters, evaluate the loss function and its derivatives on the images
2. Using the derivatives, take an objective-increasing step in object parameter space

Step (1) is the computationally expensive one. Currently, it takes on the order of seconds to evaluate an object parameter's derivative on all the pixels that it affects. We use a second-order method, and calculating a Hessian takes about a minute (though we hope to hard-code the Hessian and reduce this time significantly in the future). On a single object, the algorithm converges in 20-30 steps using second-order methods.

Since the pixels are independent given the parameters, step (2) can be executed completely in parallel.

One can imagine the celestial objects and images as a many-to-many bipartite graph. Objects occur in several images, and an image may contain several objects. Given this, one might think of three optimization strategies, in increasing order of complexity:

1. Optimize each object separately and in parallel, keeping all the objects fixed at their initial values. Repeat, hopefully not too many times.
2. Put the images and parameters on separate, distributed processes and add "shuffle" steps 1.5 and 2.5, and perform updates synchronously.
   (a) 1.5. Communicate the parameters to tile processes
   (b) 2.5. Communicate the derivatives to the parameter processes
3. Like (2) but with asynchronous updating.

Our current approach is (1). We haven't yet evaluated whether it will be good enough.

1.3 Code

CELESTE is written in julia. The code can be found at `https://github.com/jeff-regier/Celeste.jl`. Below is an introduction to the codebase that I wrote for Kiran.

The file with most of the interesting computation happens is src/ElboDeriv.jl. You may also want to look at src/CelesteTypes.jl for the definition of data structures used there. The objective function that we are maximizing is called the "elbo" (for "evidence lower bound"), and the bulk of our computation is in evaluating this objective and its derivative. You can look at bin/benchmark_elbo.jl for an example of generating a small sample image and evaluating the elbo and its derivatives once – this is done with the function ElboDeriv.elbo.

The key data structures are ModelParams (usually "mp" in our code) and arrays of ImageTile objects. The ImageTile objects contain the pixels in a photograph of a small section of the sky and related information about what band (color) image the tile was taken from, its location, and background noise estimates. ImageTile objects are kept around in a TiledBlob, which has five elements, one for each band of an picture of the sky. Each band contains a TiledImage, which is a 2d array of ImageTiles that partition that band's raw pixels into smaller tiles.

A single celestial object (a star or a galaxy) is called a "source", as in "a source of light". The type ModelParams contains information about sources. Each image contains S sources, which for now we take to be fixed. For your purposes, there are three important fields in the ModelParams object: - vp: These are the mutable model parameters that we are optimizing over. It is an array of length S, where each element has the parameters germane to one source. You can think of this as our parameter vector. - patches: This is an immutable array of length S containing about each source that is initialized once and never changed. For example, it has information about the optics at that source location and an estimated size of the source. - tile_sources: This is an array that matches the dimensions of the TiledImage. It contains the many-to-many mapping between sources and ImageTiles. Each tile may be affected by multiple sources and each source may appear in multiple tiles, and this mapping is contained in mp.tile_sources.

One more data structure is worth mentioning. The return value of an evaluation of ElboDeriv.elbo is a SensitiveFloat. This contains both the value of the objective and the derivatives of the objective with repsect to a range of parameters.

The steps of Celeste are something like:

- Initialize the TiledImage objects and ModelParams (using an existing catalog).
- Until converged:
  - Using the current parameters, run over each of the ImageTiles to get the function value and derivatives in SensitiveFloats
  - Add up the SensitiveFloats to get an overall function value and gradient
  - Using the gradient, take a step, updating the value of each source in ModelParams Done

Since the function value and derivatives is simply the sum over ImageTiles, that is the step can be done completely in parallel. The resulting SensitiveFloats must be added to compute the gradient step to update the ModelParams.

With this in mind, you can take a look at src/ElboDeriv.jl. You may want to read from the bottom to the top, which is the direction of high- to low-level functions. Specifically, if you take a look at src/ElboDeriv.jl:elbo_likelihood!, you will find a naive attempt to parallelize the elbo evaluation over the tiles using @parallel. (It's currently disabled by default because it doesn't work very well.) This is, I think, the first place to start looking at parallizing Celeste.

In the long run, we will run over not only a single TiledBlob, but over many of them. The basic paradigm, however, will remain the same – we will have a large number of sources, a many-to-many mapping with ImageTiles, and a reduce over SensitiveFloats at each step.