The current state of scarlet and looking toward the future

Fred Moolekamp

2021-07-02

1 Introduction

From its inception scarlet was designed to function as a multi-band deblender for the Rubin Observatory Legacy Survey of Space-Time (LSST), a mutli-purpose deblender for joint survey processing of data from multiple instruments, and an all-purpose deblender for more focused analysis of complicated blended regions like galaxy mergers and low surface brighness galaxies (LSB's). But as the codebase has matured, internal testing and user feedback have caused scarlet to have a much wider range of models available (Melchior et al. 2021 in prep) which has resulted in a much more complicated framework. While scarlet is fully capable of meeting the deblending needs of the LSST, a more simplified version of the objective function will allow us to take advantage of the primitives and initialization schemes implemented in scarlet while avoiding the unnecessary bloat that comes with having a more general framework, as well as a significantly reduced memory footprint.

2 The evolution of scarlet models

2.1 The original scarlet model

The key insight that lead to the creation of scarlet is that a large percentage of galaxies can be approximated by a collection of components, where each component is composed of a 2 dimensional morphology (shape) matrix with roughly a constant spectrum (amplitude) over the entire morphology. So in other words the model of a single source k is

$$M_k = A_k \cdot S_k \tag{1}$$

where M_k is the model for the k^{th} component, A_k is its amplitude and S_k is its shape. So the model for the entire blend of K components is

$$M = \sum_{k=1}^{K} M_k. \tag{2}$$

Given our data D and weights W the log likelihood $\mathscr{L}$ without convolution is

$$\mathscr{L} = \frac{1}{2} W \cdot (D - M)^2 \tag{3}$$

and with convolutions is

$$\mathscr{L}_C = \frac{1}{2} W \cdot (D - PM)^2 \tag{4}$$

where P is a tensor that performs a convolution on the entire model M.

In addition to its model, each component can be constrained, for example components are typically centered on peak detections in an image and required to be non-negative (as most astrophysical sources are photo emmitters), monotonically decrease from the center (melchior et al 2018), and implement a normalization to remove degeneracies in the combined matrix product. The ability to set these constraints was an integral feature from day one and other constraints, such as L1 and L2 sparsity, and 180° symmetry, have been attempted and implemented in the past, with the expectation that others will arise in the future. So we employed proximal operators, which can be thought of a optimal projections of the model onto the space where the distance from the updated model to the constraint space is minimized.

In this sense the original scarlet model was a constrained matrix factorization problem of the form

$$\mathscr{L} = \frac{1}{2} W \cdot (D - P \bigotimes AS)^2 \tag{5}$$

where we have taken some algebraic liberties in writing down this equation as follows: $\cdot$ is an elementwise Haddamard product, P is a tensor representing the PSF in each band, A is a $b \times k$ matrix with k sources and b bands storing the SEDs, S is a $k \times N$ matrix with N pixels, and $\bigotimes$ is the tensor product between P and the matrix AS. Our original solution to solve this was to

calculate the entire gradient in one step using a matrix inversion, followed by the application of proximal operators.

2.2 The move to autograd

To improve performance we decided to switch from direct convolutions to convolutions in Fourier space. Although the FFT is still a linear transformation, it is more efficient (in terms of both memory and CPU cycles) to calculate the convolution in both the forward (model building) and backward (gradient udpate) directions by applying it as a function $\boldsymbol{P}(\boldsymbol{M})$ (note the difference from Eq. 5, where $\boldsymbol{P}$ is a tensor that contains the PSF in each band while $\boldsymbol{P}(\boldsymbol{M})$ is a function of the model $\boldsymbol{M}$, potentially using FFTs to convolve the model), giving us the likelihood

$$\mathscr{L}_C = \frac{1}{2}\boldsymbol{W} \cdot (\boldsymbol{D} - \boldsymbol{P}(\boldsymbol{M}))^2. \tag{6}$$

The easiest way to implement this was to use an existing package that performs automatic differentiation. We initially attempted to use the pytorch python package but due to conflicts with the version of the science pipelines at the time we were forced to switch to autograd. While it is not trivial to switch between packages, we have since tested both jax and pytorch on toy versions of scarlet and found the autograd version is still the fastest implementation for our needs (see `https://github.com/fred3m/notebooks/blob/master/direct_conv.ipynb`), the main reason being that updating slices of numpy arrays in pytorch and JAX is unsupported and very slow respectively, as compared to the custom functions we were able to write in scarlet to accomplish slicing using autograd. However, autograd is no longer actively developed, as it was deprecated in favor of JAX, so Peter Melchiors group is likely to switch to either a JAX or pytorch version of scarlet in the near future.

Regardless of the package used, the basic concept of automatic differentiation is the same. One can think of automatic differentiation as an implementation in code of the chain rule from differential calculus. Each time a function updates a variable (which includes updating an array of variables), the function to calculate the gradient of the applied function is stored. Once the likelihood has been calculated, the gradient functions are called in reverse order to update each variable in the model. While we never have to calculate any of the gradients ourselves (autograd does tha for use), if we were to write out the full gradient of a single $\boldsymbol{A}_i$

parameter it would be

$$\frac{\partial \mathscr{L}_C}{\partial A_i} = \frac{\partial \mathscr{L}_C}{\partial P(M)} \frac{\partial P(M)}{\partial M} \frac{\partial M}{\partial M_i} \frac{\partial M_i}{\partial A_i}. \tag{7}$$

Looking at each term individually we see that these are all simple gradients to calculate for ourselves and that we don't really *need* to use autograd. For example, to calculate the gradient update of the SED for the i^{th} component we get

$$\frac{\partial \mathscr{L}_C}{\partial P(M)} = -W \cdot (D - P(M)) \tag{8}$$

$$\frac{\partial M}{\partial M_i} = 1 \tag{9}$$

$$\partial M_i \partial A_i = S^T \tag{10}$$

$$\tag{11}$$

In direct space $\frac{\partial P(M)}{\partial M}$ is just the input gradients flipped and transposed and even in kspace there is a simple analytic form (because this is a linear transformation). So in reality for the scarlet models that we use in the science pipelines we really don't need autograd anymore and in fact removing comes with multiple benefits. As mentioned above, autograd is deprecated and using a deprecated package before the LSST has even begun is ill-advised (for the same reason that we switched to python 3). There is also a significant memory overhead that comes with using autograd and a minor reduction in performance. The memory overhead comes from the fact that autograd was developed primarily for backpropagating gradients for neural networks, which typically involve apply a matrix transformation to an input vector, which is different from the way that our optimizer works. As a result, the type of gradient that autograd calculates (a vector-jacobian-product) requires the output model from the forward direction as an input *after each operation*. In other words, the operation $M = M_1 + M_2$ increases the memory by the size of M, giving us a total increase in memery of an order of magnitude or more! I tested this out using a toy model and did show a substantial increase in memory without even applying the convolutions. See `https://jax.readthedocs.io/en/latest/notebooks/autodiff_cookbook.html` for more details on why JAX chose the gradient method that they use and why it is inefficient.

While the LSST has commited to using CPUs (at least at the beginning of the survey), joint processing will be using GPUs or TPUs, which are not supported by autograd. So Peter Melchior and his group would already like to move away from autograd and convert scarlet to pytorch or JAX. But for the LSST I believe that due to memory and computational issues to calculate gradients that we can already calulate on our own is good reason to just calculate all of the gradients ourselves without using any automatic differention package. This frees the other scarlet developers to move to a GPU/TPU compatible package and allows us to have a faster, more memory efficient codebase.

2.3 The current scarlet model framework

It quickly became obvious that in order to blend more complicated scenes like galaxies with dust lanes, merging galaxies, and crowded fields, we would also need to allow our models to become more complicated. We were also using a simple proximal gradient method as our optimizer, which uses only first order derivatives to calculate the gradient updates. So in 2019 and 2020 we made several significant changes to the scarlet architecture to allow us to use more complicated models and an improved nADAM optimizer that approximates the Hessian for pseudo 2nd order gradients. This section briefly describees the current version of scarlet as a result of those changes.

For the optimizer we created a `Parameter` class that is an extension of a numpy array (technically the autograd ArrayBox wrapper for a numpy array). In addition to having the data of the array, the `Parameter` also stores the gradient and square of the gradient to give nAdam all of the information that it needs to calcualte gradients for each parameter in the model. For the simple models discussed in Section 2.1, each SED is a `Parameter` and each morphology is a `Parameter`.

An object made up of a collection parameters is called a `Model`, a hierarchical structure that may have other models as children as well as its own parameters. In addition to children and parameters, models also have a `get_model` method that describes how to reconstruct the model using its children and parameters. All of the other objects that make up the full model of a blend are derived from `Model`.

A `Component` is a single component in a blend, which can be a source or a component of a source like its bulge or disk. Sources with a similar model to Eq. 1 are `FactorizedComponent` models, which are models with two child models: `Spectrum` and `Morphology`, which are

themselves subclassed for different types of models. The `PSF` is also a model, which makes it possible to fit the PSF during optimization (although this is not currently used in practice) as well as the `Renderer` class that renders a scarlet model in the same frame as an observation (in a single or in multiple bands, at potentially different resolutions). There is no `Source` class, instead sources inherit directly from `Component` or one of its subclasses. In LSST we use `SingleExtendedSource`, which inherits from `FactorizedComponent` for faint sources with only a single component and `MultiExtendedSource`, which inherits from `CombinedComponent`, which is a component with two `SingleExtendedSource` models as children. Finally the `Blend` class is also a `CombinedComponent` that contains all of the sources in the blend as models and performs the optimization.

3 Simplifying scarlet models and the objective function

Because autograd is embedded in the `Parameter` class, which almost every other data structure in in scarlet is based off of, moving away from autograd requires updates to almost every module in the package. But for Rubin there is good reason to overhaul scarlet and use our own optimization scheme. There are two types of overhead that come with the current version of scarlet, one is the complexity of the overall data structures, most of which are unused and unnecessary for a general ground based deblender. So testing out new algorithms and ideas is much more complicated because of all of the overhead needed to make new models and optimization algorithms fit the scarlet data structures. The other overhead is of course the increase in memery and CPU cycles that comes with using autograd.

Part of the reason that scarlet models became so complex is because the developers decided that it would be useful to allow for arbitrary models that are not factorized, so that they don't have to be the outer product of a vector of flux amplitudes and a shape matrix (in other words they don't obey Eq. 1). Another major complication is the ability to (in general) use multiple observations, potentially from multiple different instruments. This requires several steps to go from the model generated by scarlet to the log-likelihood that is optimized, whereas this is just a single line $logL = 0.5 * weights * (data - model) * *2$ when dealing with a single set of multiband observations. The last complication was the desire to have the scarlet optimizer use the proxmin package algorithms for deblending. This has limited our options in terms of optimizers that we can use as well as created a very complicated method to fit a blend that is very difficult for anyone outside of the scarlet developers to parse.

3.1 Simplified scarlet for Rubin and other single instrument surveys

The main scarlet repo now contains a single *lite* branch that implements a more simple autograd-free model. It should be noted that the "lite" only refers to the complexity of the data structures that create the model and update the parameters, while the actual internal algorithm is the same as the current version of scarlet, while obtaining similar results up to numerical accuracy. This section will outline the changes made that this code faster, more memory efficient, and easier to parse for non-scarlet developers.

The `Parameter` class has been replaced with the `LiteParameter` class. In addition to being dependent on autograd, the scarlet `Parameter` class is desigend specifically for the Proximal ADAM optimizer, storing the cumulative gradient, gradient squared, and gradient maximum, which are all passed to the optimizer in *proxmin*. The new `LiteParameter` is instead inherited by the `BeckTeboulleParameter` and `AdaproxParameter` to implement the appropriate variables for the Beck-Teboulle accelerated proximal gradient method and the current scarlet Proximal Adam method respectively. This makes it easier to find/fix bugs in the optimizer and greatly simplifies the `fit` method in the `LiteBlend` class, which will be discussed shortly.

Each component in the blend is an instance of the `LiteComponent` class, which is very similar to the `FactorizedComponent` class in vanilla scarlet, except that it also implements gradient methods for the SED and morphology parameters and allows for proximal operators that can act on the combined model as opposed to the SED and morphology parameters individually. This, for example, now allows us to implement a sparsity constraint that could not previoiusly be implemented. The issue with sparsity in the main scarlet is that because the intensity and shape of each component is split into two different parameters, there is no consistent way to use sparsity as a fraction of the background RMS, which is what one needs to consistently apply sparsity to both bright and faint objects uniformly. Because `LiteComponent` contains an `update` method that applies the proxes for both parameters, it can also implement this additional combined sparsity, which prevents the models from becoming as extended and fractal like, which may fix some of the issues that we are seeing with the rho statistics. The `LiteComponent.update` method is called from the `LiteParameter` class when it is updated, as different optimizers have different ways that they implement proximal operators.

Because a `Source` class in scarlet doesn't have any functionality outside of initialization and, in the case of the `MultiExtendedSource` class, making sure that sources with multiple components are organized in a way that makes it easy to attribute each component to the appropriate

source, we have created a new `LiteSource` class that simply acts as a container of components. It isn't called at all by the optimizer and is only used to group a set of components that we have reason to believe are attributed to the same sources, making it easy to swap components later on if it is deemed necessary.

Finally is the `LiteBlend` class, which acts as the `Blend`, `Observation`, and `Renderer` classes from main scarlet. This is possible because the three classes all actually contain very little logic when there is only a single instrument with all of the images resampled onto the same pixel grid. It builds the model by inserting each component as a slice into the full model, calculates the gradient from the log-likelihood, and passes that on to the individual compoennts. Returning to an older verison of scarlet, the `LiteBlend.fit` method iterates over all of the components (not sources) in the blend, calling the `update` method of each source to calculate the gradients, update their parameters, and apply the proximal operators.

Preliminary testing shows that when both the lite branch and main scarlet branch are allowed to run for the same number of iterations, there is an ~40% speedup in runtime using the lite branch while obtaining a result with a similar residual. Howeer, the improved sparsity allows the lite branch to converge faster with less extended features. There is hope that if scarlet is the cause of the extended stellar sizes noticed by Dan Taranu on DC2 data, this may help with that problem, but more testing is needed to confirm this.

4 Conclusion

We have outlined the evolution of scarlet over time and given an outline of the simplified version of scarlet data structures that will be more compututationally efficient and maintainable for the LSST, and other surveys using a single instrument with pixels resampled onto the same grid. The new branch shows similar results to the current implementation, and will be easier to extend to different models, such as multi-scale models, which will improve the speed of tesing and implementing new features and is much more readable for users outside of scarlet. Once more testing on RC2 and DC2 data has completed, the lite branch will be reviewed, implemented in the main sarlet repo, and utilized by Rubin the *meas_extensions_scarlet* package.

A References

B Acronyms