Anatomy of a Probabilistic Programming Framework

In this blog post, we'll break down what probabilistic programming frameworks are made up of, and how the various pieces are organized and structured. We'll take a look at some open source frameworks as examples.

13 minute read

Photo credit: coolbackgrounds.io (https://coolbackgrounds.io/)

Recently, the PyMC4 developers submitted an abstract (https://openreview.net/forum?id=rkgzj5Za8H) to the *Program Transformations for Machine Learning* NeurIPS workshop (https://program-transformations.github.io/). I realized that despite knowing a thing or two about Bayesian modelling, I don't understand how probabilistic programming frameworks are structured, and therefore couldn't appreciate the sophisticated design work going into PyMC4. So I trawled through papers, documentation and source code[1] of various open-source probabilistic programming frameworks, and this is what I've managed to take away from it.

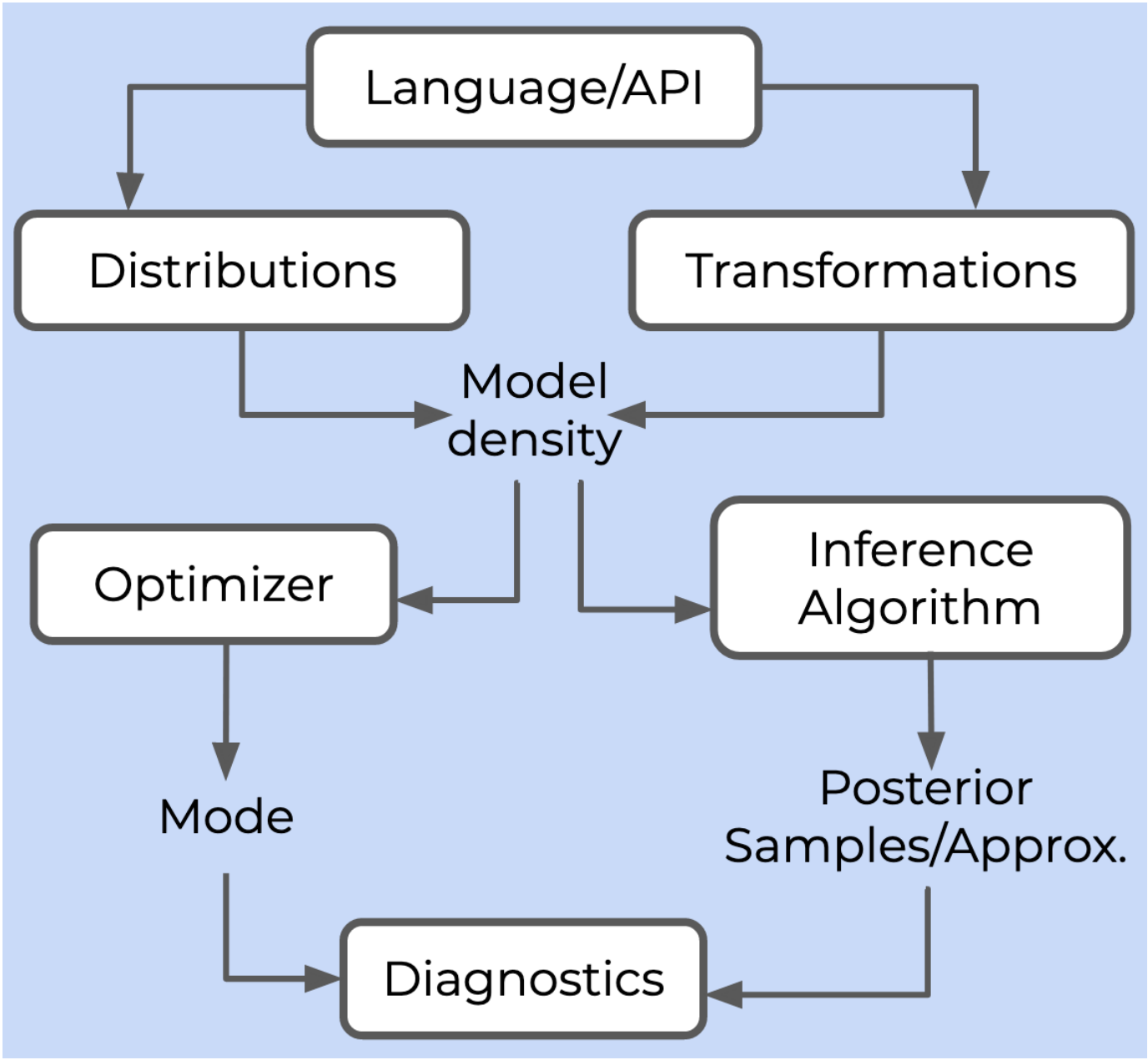
I assume you know a fair bit about probabilistic programming and Bayesian modelling, and are familiar with the big players in the probabilistic programming world. If you're unsure, you can read up here (https://eigenfoo.xyz/bayesian-inference-reading/).

Dissecting Probabilistic Programming Frameworks

A probabilistic programming framework needs to provide six things:

1. A language or API for users to specify a model
2. A library of probability distributions and transformations to build the posterior density
3. At least one inference algorithm, which either draws samples from the posterior (in the case of Markov Chain Monte Carlo, MCMC) or computes some approximation of it (in the case of variational inference, VI)
4. At least one optimizer, which can compute the mode of the posterior density
5. An autodifferentiation library to compute gradients required by the inference algorithm and optimizer
6. A suite of diagnostics to monitor and analyze the quality of inference

These six pieces come together like so:



Let's break this down one by one.

Specifying the model: language/API

This is what users will use to specify their models. Most frameworks will let users write in some existing programming language and call the framework's functions and classes, but ~~some others~~ — why don't I just say it — Stan rolls their own domain-specific language.

The main question here is what language you think is best for users to specify models in: any sufficiently popular host language (such as Python) will reduce the learning curve for users and make the framework easier to develop and maintain, but a creating your own language allows you to introduce helpful abstractions for your framework's particular use case (as Stan does (https://mc-stan.org/docs/2_20/reference-manual/blocks-chapter.html), for example).

At this point I should point out the non-universal, Python bias in this post: there are plenty of interesting non-Python probabilistic programming frameworks out there (e.g. Greta (https://greta-stats.org/) in R, Turing (https://turing.ml/dev/) and Gen (https://www.gen.dev/) in Julia, Figaro (https://github.com/p2t2/figaro) and Rainier (https://github.com/stripe/rainier) in Scala), as well as universal probabilistic programming systems[2] (e.g. Venture (http://probcomp.csail.mit.edu/software/venture/) from MIT, Angelican (https://probprog.github.io/anglican/index.html) from Oxford)[3]. I just don't know anything about any of them.

Building the model density: distributions and transformations

These are what the user's model calls, in order to compile/build the model itself (whether that means a posterior log probability, in the case of MCMC, or some loss function to minimize, in the case of VI). By *distributions*, I mean the probability distributions that the random variables in your model can assume (e.g. Normal or Poisson), and by *transformations* I mean deterministic mathematical operations you can perform on these random variables, while still keeping track of the derivative of these transformations[4] (e.g. exponentials, logarithms, sines or cosines).

This is a good time to point out that the interactions between the language/API and the distributions and transformations libraries is a major design problem. Here's a (by no means exhaustive) list of necessary considerations:

1. In order to build the model density, the framework must keep track of every distribution and transformation, while also computing the derivatives of any such transformations. This results in a Jekyll-and-Hyde problem where every transformation requires a forward and backwards definition. Should this tracking happen eagerly, or should it be deferred until the user specifies what the model will be used for?
2. Theoretically, a model's specification should be the same whether it is to be used for evaluation, inference or debugging. However, in practice, the program execution (and computational graph) are different for these three purposes. How should the framework manage this?
3. The framework must also keep track of the shapes of random variables, which is frighteningly non-trivial! Check out this blog post (https://ericmjl.github.io/blog/2019/5/29/reasoning-about-shapes-and-probability-distributions/) or the original Tensorflow Distributions paper (https://arxiv.org/abs/1711.10604) (specifically section 3.3 on shape semantics) for more details.

For a more comprehensive treatment, I can't recommend Junpeng Lao's PyData Córdoba 2019 talk (https://docs.google.com/presentation/d/1xgNRJDwkWjTHOYMj5aGefwWiV8x-Tz55GfkBksZsN3g/edit?usp=sharing) highly enough — he explains in depth the main challenges in implementing a probabilistic programming API and highlights how various frameworks manage these difficulties.

Computing the posterior: inference algorithm

Having specified and built the model, the framework must now actually perform inference: given a model and some data, obtain the posterior (either by sampling from it, in the case of MCMC, or by approximating it, in the case of VI).

Most probabilistic programming frameworks out there implement both MCMC and VI algorithms, although strength of support and quality of documentation can lean heavily one way or another. For example, Stan invests heavily into its MCMC, whereas Pyro has the most extensive support for its stochastic VI.

Computing the mode: optimizer

Sometimes, instead of performing full-blown inference, it's useful to find the mode of the model density. These modes can be used as point estimates of parameters, or as the basis of approximations to a Bayesian posterior. Or perhaps you're doing VI, and you need some way to perform SGD on a loss function. In either case, a probabilistic programming framework calls for an optimizer.

If you don't need to do VI, then a simple and sensible thing to do is to use some BFGS-based optimization algorithm (https://en.wikipedia.org/wiki/Broyden%E2%80%93Fletcher%E2%80%93Goldfarb%E2%80%93Shanno_algorithm) (e.g. some quasi-Newton method like L-BFGS (https://en.wikipedia.org/wiki/Limited-memory_BFGS)) and call it a day. However, frameworks that focus on VI need to implement optimizers commonly seen in deep learning (http://docs.pyro.ai/en/stable/optimization.html#module-pyro.optim.optim), such as Adam or RMSProp.

Computing gradients: autodifferentiation

Both the inference algorithm and the optimizer require gradients (at least, if you're not using ancient inference algorithms and optimizers!), and so you'll need some way to compute these gradients.

The easiest thing to do would be to rely on a deep learning framework like TensorFlow or PyTorch. I've learned not to get too excited about this though: while deep learning frameworks' heavy optimization of parallelized routines lets you e.g. obtain thousands of MCMC chains in a reasonable amount of time (https://colindcarroll.com/2019/08/18/very-parallel-mcmc-sampling/), it's not obvious that this is useful at all (although there's definitely some work going on in this area).

Monitoring inference: diagnostics

Finally, once the inference algorithm has worked its magic, you'll want a way to verify the validity and efficiency of that inference. This involves some off-the-shelf statistical diagnostics (https://arviz-devs.github.io/arviz/api.html#stats) (e.g. BFMI, information criteria, effective sample size, etc.), but mainly lots and lots of visualization (https://arviz-devs.github.io/arviz/api.html#plots).

A Zoo of Probabilistic Programming Frameworks

Having outlined the basic internals of probabilistic programming frameworks, I think it's helpful to go through several of the popular frameworks as examples. I've tried to link to the relevant source code in the frameworks where possible.

Stan

It's very easy to describe how Stan is structured: literally everything is implemented from scratch in C++.

1. Stan has a compiler for a small domain-specific language for specifying Bayesian models (https://github.com/stan-dev/stan/tree/develop/src/stan/lang)
2. Stan has libraries of probability distributions (https://github.com/stan-dev/math/tree/develop/stan/math/prim) and transforms (https://github.com/stan-dev/math/tree/develop/stan/math/prim/fun)
3. Stan implements dynamic HMC (https://github.com/stan-dev/stan/tree/develop/src/stan/mcmc/hmc) and variational inference (https://github.com/stan-dev/stan/tree/develop/src/stan/variational)
4. Stan also rolls their own autodifferentiation library (https://github.com/stan-dev/math/tree/develop/stan/math)[5]
5. Stan implements an L-BFGS based optimizer (https://github.com/stan-dev/stan/tree/develop/src/stan/optimization) (but also implements a less efficient Newton optimizer (https://mc-stan.org/docs/2_20/reference-manual/optimization-algorithms-chapter.html))
6. Finally, Stan has a suite of diagnostics (https://github.com/stan-dev/stan/tree/develop/src/stan/analyze/mcmc)

Note that contrary to popular belief, Stan *does not* implement NUTS:

> *Stan implements a dynamic Hamiltonian Monte Carlo method with multinomial sampling of dynamic length trajectories, generalized termination criterion, and improved adaptation of the Euclidean metric.*
>
> *— Dan Simpson (@dan_p_simpson (https://twitter.com/dan_p_simpson)) September 5, 2018 (https://twitter.com/dan_p_simpson/status/1037332473175265280)*

And in case you're looking for a snazzy buzzword to drop:

> *Adaptive HMC. @betanalpha (https://twitter.com/betanalpha) is reluctant to give it a more specific name because, to paraphrase, that's just marketing bullshit that leads to us celebrating tiny implementation details rather than actual meaningful contributions to comp stats. This is a wide-ranging subtweet.*
>
> *— Dan Simpson (@dan_p_simpson (https://twitter.com/dan_p_simpson)) August 27, 2018 (https://twitter.com/dan_p_simpson/status/1034098649406554113)*

TensorFlow Probability (a.k.a. TFP)

1. TFP users write Python (albeit through an extremely verbose API (https://colcarroll.github.io/ppl-api/))
2. TFP implements their own distributions (https://github.com/tensorflow/probability/tree/master/tensorflow_probability/python/distributions) and transforms (https://github.com/tensorflow/probability/tree/master/tensorflow_probability/python/bijectors) (which TensorFlow, for some reason, calls "bijectors"). You can find more details in their arXiv paper (https://arxiv.org/abs/1711.10604)
3. TFP implements a ton of MCMC (https://github.com/tensorflow/probability/tree/master/tensorflow_probability/python/mcmc) algorithms and a handful of VI algorithms (https://github.com/tensorflow/probability/tree/master/tensorflow_probability/python/vi) in TensorFlow
4. TFP implements several optimizers (https://github.com/tensorflow/probability/tree/master/tensorflow_probability/python/optimizer), including Nelder-Mead, BFGS and L-BFGS (again, in TensorFlow)
5. TFP relies on TensorFlow to compute gradients (er, duh)
6. TFP implements a handful of metrics (https://github.com/tensorflow/probability/blob/master/tensorflow_probability/python/mcmc/diagnostic.py) (e.g. effective sample size and potential scale reduction), but seems to lack a comprehensive suite of diagnostics and visualizations: even Edward2 (https://github.com/tensorflow/probability/tree/master/tensorflow_probability/python/experimental/edward2) (an experimental interface to TFP for flexible modelling, inference and criticism) suggests that you build your metrics manually or use boilerplate in `tf.metrics` (https://github.com/tensorflow/probability/blob/master/tensorflow_probability/python/experimental/edward2/Upgrading_From_Edward_To_Edward2.md#model--inference-criticism)

PyMC3

1. PyMC3 users write Python code, using a context manager pattern (i.e. `with pm.Model as model`)
2. PyMC3 implements its own distributions (https://github.com/pymc-devs/pymc3/tree/master/pymc3/distributions) and transforms (https://github.com/pymc-devs/pymc3/blob/master/pymc3/distributions/transforms.py)
3. PyMC3 implements NUTS (https://github.com/pymc-devs/pymc3/blob/master/pymc3/step_methods/hmc/nuts.py), (as well as a range of other MCMC step methods (https://github.com/pymc-devs/pymc3/tree/master/pymc3/step_methods)) and several variational inference algorithms (https://github.com/pymc-devs/pymc3/tree/master/pymc3/variational), although NUTS is the default and recommended inference algorithm
4. PyMC3 (specifically, the `find_MAP` function) relies on `scipy.optimize` (https://github.com/pymc-devs/pymc3/blob/master/pymc3/tuning/starting.py), which in turn implements a BFGS-based optimizer
5. PyMC3 relies on Theano (https://github.com/pymc-devs/pymc3/blob/master/pymc3/theanof.py) to compute gradients
6. PyMC3 delegates posterior visualization and diagnostics (https://github.com/pymc-devs/pymc3/blob/master/pymc3/plots/__init__.py) to its cousin project ArviZ (https://arviz-devs.github.io/arviz/)

Some remarks:

- PyMC3's context manager pattern is an interceptor for sampling statements: essentially an accidental implementation of effect handlers (https://arxiv.org/abs/1811.06150).
- PyMC3's distributions are simpler than those of TFP or PyTorch: they simply need to have a `random` and a `logp` method, whereas TFP/PyTorch implement a whole bunch of other methods to handle shapes, parameterizations, etc. In retrospect, we realize that this is one of PyMC3's design flaws (https://docs.pymc.io/developer_guide.html#what-we-got-wrong).

PyMC4

PyMC4 is still under active development (at least, at the time of writing), but it's safe to call out the overall architecture.

1. PyMC4 users will write Python, although now with a generator pattern (e.g. `x = yield Normal(0, 1, "x")`), instead of a context manager
2. PyMC4 will rely on TensorFlow distributions (a.k.a. `tfd`) (https://github.com/pymc-devs/pymc4/tree/master/pymc4/distributions/) for both distributions and transforms
3. PyMC4 will also rely on TensorFlow for MCMC (https://github.com/pymc-devs/pymc4/tree/master/pymc4/inference/) (although the specifics of the exact MCMC algorithm are still fairly fluid at the time of writing)
4. As far as I can tell, the optimizer is still TBD
5. Because PyMC4 relies on TFP, which relies on TensorFlow, TensorFlow manages all gradient computations automatically
6. Like its predecessor, PyMC4 will delegate diagnostics and visualization to ArviZ

Some remarks:

- With the generator pattern for model specification, PyMC4 embraces the notion of a probabilistic program as one that defers its computation. For more color on this, see this Twitter thread (https://twitter.com/avibryant/status/1150827954319982592) I had with Avi Bryant (https://about.me/avibryant).

Pyro

1. Pyro users write Python
2. Pyro relies on PyTorch distributions (https://github.com/pyro-ppl/pyro/blob/dev/pyro/distributions/__init__.py) (implementing its own where necessary (https://github.com/pyro-ppl/pyro/tree/dev/pyro/distributions)), and also relies on PyTorch distributions for its transforms (https://github.com/pyro-ppl/pyro/tree/dev/pyro/distributions/transforms)
3. Pyro implements many inference algorithms (http://docs.pyro.ai/en/stable/inference.html) in PyTorch (including HMC and NUTS (https://github.com/pyro-ppl/pyro/tree/dev/pyro/infer/mcmc)), but support for stochastic VI (https://github.com/pyro-ppl/pyro/blob/dev/pyro/infer/svi.py) is the most extensive
4. Pyro implements its own optimizer (https://github.com/pyro-ppl/pyro/blob/master/pyro/optim/optim.py) in PyTorch
5. Pyro relies on PyTorch to compute gradients (again, duh)
6. As far as I can tell, Pyro doesn't provide any diagnostic or visualization functionality

Some remarks:

- Pyro includes the Poutine submodule, which is a library of composable effect handlers (https://arxiv.org/abs/1811.06150). While this might sound like recondite abstractions, they allow you to implement your own custom inference algorithms and otherwise manipulate Pyro probabilistic programs. In fact, all of Pyro's inference algorithms use these effect handlers.

1. In case you're testifying under oath and need more reliable sources than a blog post, I've kept a Zotero collection (https://www.zotero.org/eigenfoo/items/collectionKey/AE8882GQ) for this project. ↩

2. Universal probabilistic programming is an interesting field of inquiry, but has mainly remained in the realm of academic research. For a (much) more comprehensive treatment, check out Tom Rainforth's PhD thesis (http://www.robots.ox.ac.uk/~twgr/assets/pdf/rainforth2017thesis.pdf). ↩

3. Since publishing this blog post, I have been informed that I am more ignorant than I know: I have forgotten Soss.jl (https://github.com/cscherrer/Soss.jl) in Julia and ZhuSuan (https://github.com/thu-ml/zhusuan) in Python. ↩

4. It turns out that such transformations must be local diffeomorphisms (https://en.wikipedia.org/wiki/Local_diffeomorphism), and the derivative information requires computing the log determinant of the Jacobian of the transformation, commonly abbreviated to `log_det_jac` or something similar. ↩

5. As an aside, I'll say that it's mind boggling how Stan does this. To quote a (nameless) PyMC core developer:

   > *I think that maintaining your own autodifferentiation library is the path of a crazy person.*

   ↩

Tags: open source | probabilistic programming | pymc

Updated: September 30, 2019

Want to hear more from me?

Subscribe to my newsletter! My thoughts on what I'm reading and learning, delivered once a month.
More information here (https://buttondown.email/eigenfoo/archive/hello-and-welcome/). Newsletter archive here (https://buttondown.email/eigenfoo/archive/).

Subscribe