Modular construction of Bayesian inference algorithms

Adam Ścibior[*]
Department of Engineering
University of Cambridge
United Kingdom
ams240@cam.ac.uk

Zoubin Ghahramani
Department of Engineering
University of Cambridge
United Kingdom
zoubin@eng.cam.ac.uk

Abstract

We propose a set of abstractions to modularize implementation of Bayesian inference algorithms. We provide a proof-of-concept implementation as a Haskell library and demonstrate on several examples how it simplifies implementation of Monte Carlo algorithms. Our technique is based on a method for modular construction of interpreters using monad transformers and is applicable generically to probabilistic programming.

1 Introduction

Implementation of approximate Bayesian inference algorithms is a difficult task. Probabilistic programming systems hide this difficulty from the user, but someone still needs to implement backends for those systems. And since there will never be an algorithm that solves all inference problems, this implementation will forever remain an ongoing effort.

In this paper we propose a set of abstractions that modularizes the implementation of inference algorithms. By devising a collection of small building blocks that algorithms can be constructed from, we reduce the effort required to implement new inference methods based on existing ones, both in terms of the amount of code that needs to be written and in terms of how difficult it is to comprehend and test such code.

Our framework is directly applicable to probabilistic programming in full generality. It is based on a technique for building modular interpreters using monadic denotational semantics [Liang et al., 1995], so it can be employed in a compiler for any probabilistic programming language. We provide a proof-of-concept implementation as a Haskell library[†] where probabilistic programs can be written using arbitrary Haskell code. In the future we plan to apply those techniques to implementation of compilers for domain-specific probabilistic programming languages including imperative ones.

The rest of this paper offers a high-level explanation of the abstractions we propose, describes selected building blocks, shows how to put them together to obtain interesting inference algorithms, and concludes with a discussion of usability of our framework. We do not assume that the reader is familiar with either Haskell or monads.

2 Probability monads

Monads are a popular tool in functional programming [Wadler, 1992] and semantics of programming languages [Moggi, 1989] and have a reputation for being scary and incomprehensible. Here we do

[*]also with MPI Tübingen
[†]https://github.com/adscib/monad-bayes

not explain monads in detail, but rather describe the necessary minimum required to understand this paper.

Our discussion is based on the concept of probabilistic programming, where probabilistic models are identified with programs. We assume there exists a set of primitive distributions, such as Normal and Bernoulli, that can be used to construct bigger models, and that for every primitive distribution we can sample from it and compute its density.

We introduce a notion of a probability monad, which is a data type that can be used to interpret arbitrary probabilistic programs, by which we mean constructing a concrete data structure from the program code. In order to do that it needs to be able to represent distributions on any data types available in the program and to perform the following operations:

- create Dirac distributions that place all the probability mass on one value
- construct a joint distribution on (X, Y) from a marginal distribution of X and a conditional Y given X
- perform marginalization over a subset of random variables
- draw random variables from primitive distributions
- accumulate a likelihood score given by explicit density

In theory there exists a monad of probability distributions [Giry, 1981], but in practice it is not implementable. In this work we use two basic probability monads: one called `Enumerator` that performs inference by enumerating the entire probability space [Erwig and Kollmansberger, 2006], and one called `Sampler` performing random importance sampling from the prior.

We now discuss how more practical inference algorithms can be constructed by extending the simple probability monads above. Although we focus our discussion on sampling methods, other approaches such as Variational Inference, especially in its black-box form [Ranganath et al., 2014], could be implemented in a similar fashion.

3 Building inference algorithms

In our framework inference algorithms are transformations between abstract representations of probabilistic models. The workflow is to first interpret the program using a concrete probability monad and then apply a sequence of transformations to it until we have a representation that is convenient for answering questions about the posterior, such as a collection of samples. For this reason we call transformations in this sequence inference transformations, even though individually they may not constitute inference algorithms.

The initial representation is obtained by constructing a suitable monad stack and using it to interpret the model. A monad stack consists of a bottom monad (in our case `Enumerator` or `Sampler`) and then zero or more monad transformers layered on top of it. A monad transformer is simply a constructor that takes one monad and creates another. Every monad stack is itself a monad, obtained by starting at the bottom at then applying each monad transformer in turn. In our application every monad stack is a probability monad, so it can be used to interpret probabilistic models. From this point on we refer to monad stacks we use as inference stacks and to monad transformers as layers.

There are three distinct but related aspects of modularity displayed by our approach:

1. inference stack consists of independent layers that can be arranged in arbitrary combinations
2. inference transformations abstract over most of the stack, so they can be implemented once and applied to any stack where a particular layer (or combination of layers) is present
3. since output of an inference transformation is itself an interpretation of some probabilistic model in a certain stack, inference transformations can be chained together in arbitrary ways as long as the output stack of each of them matches the input stack of the next

The last of these was previously exploited by Ścibior et al. [2015] and Zinkov and Shan [2016], although each of these papers only allows transformations within a single data type.

We now present some concrete layers and associated inference transformations and show how they can be used to produce interesting inference algorithms. We emphasize this is not an exhaustive account of what can be achieved within our framework, but rather a small and interesting example that demonstrates the utility of our approach. We focus on implementing algorithms already existing in the literature, although different combinations of the building blocks we describe may result in genuinely new algorithms.

3.1 Population

The first layer we introduce is called `Population` by similarity with Population Monte Carlo [Cappé et al., 2004] and converts a distribution over a single value into a distributions over collections of weighted values.

We show three inference transformations associated with `Population`. The first is increasing the population size `n` times, which we call `spawn n >>`. This simply draws `n` independent collections and merges them together. In order to keep the sum of the weights constant, we divide all weights by `n`. The second is `resample`, which picks `n` new samples independently at random with probabilities proportional to the weights in the old population and set the new weights uniform. Again we adjust the weights at the end to make sure the sum of all weights stays the same.

Finally there exists a transformation that removes the `Population` layer called `collapse`. It first computes the sum of all weights and records it in the stack below as artificial likelihood. It then selects one sample at random, with probabilities proportional to weights, and retains just this sample discarding the population. If the output stack consists of `Sampler` only, this operation produces a properly weighted sample in the sense of described by Naesseth et al. [2014, Definition 1].

3.2 Sequential

The second layer can be applied to models that interleave drawing random variables with recording observations, such as state-space models, and can be used to perform inference sequentially targeting a partial posterior before extending the model with further random variables and data.

The principal inference transformation associated with `Sequential` is `advance`, which runs the model to the next observation. Interleaving `advance` with other inference transformations lets us perform inference sequentially. Once this is done, another transformation called `flatten` can be used to remove the `Sequential` layer, running through all the remaining observations if there are any left.

The combination of `Population` and `Sequential` is sufficient to implement the Sequential Monte Carlo algorithm. This is achieved simply by interleaving `advance` and `resample` transformations. Specifically, let `n` be the number of particles and `k` be the number of observations in the model. The code snippet below is the actual implementation of SMC from our library. In all the examples shown in this paper the functions named `hoistX` can be regarded as having no effect other than satisfying the type system.

```
smc :: Int -> Int -> Sequential (Population Sampler) a
                  -> Population Sampler a
smc k n = flatten .
  repeat k (advance . hoistS resample) .
  hoistS (spawn n >>)
```

The first part of the snippet above is a type signature, which asserts that `smc` is an inference transformation that takes two integer parameters and converts a stack consisting of `Sequential Population Sampler` from the top to `Population Sampler`. The input stack is obtained automatically by interpreting a standard probabilistic program in a step transparent to the user. The second part is the actual implementation, where `.` is ordinary function composition such that `(f.g)(x)= f(g(x))` and `repeat k` applies a given function `k` times. The code above corresponds to a variant of SMC for generic probabilistic programs proposed by [Wood et al., 2014].

3.3 Trace

The final layer we present is called `Trace` and corresponds to a distribution on the traces of probabilistic programs. A trace is a collection of all random variables sampled during the program's execution together with some auxiliary information related to the program's structure.

A basic inference transformation associated with `Trace` is called `mhStep` and performs a single step of the Lightweight Metropolis-Hastings algorithm proposed by Wingate et al. [2011]. We also have an inference transformation called `marginal` which discards the trace and thus removes the `Trace` layer.

The `mhStep` transformation can easily be used by itself to perform MCMC inference, but it is more interesting to see how it can be combined with different transformations to incorporate MH transitions into other algorithms. Specifically, we show how to construct an implementation of resample-move SMC [Gilks and Berzuini, 2001]. The idea is to incorporate MH transitions for each particle after the resampling step to ameliorate the problem of degeneracy in SMC. We can achieve this by modifying the `smc` implementation given above, specifically by adding `Trace` layer to the inference stack and `mhStep` transformation after `resample`.

```
smcrm :: Int -> Int ->
         Sequential (Trace (Population Sampler)) a ->
         Population Sampler a
smcrm k n = marginal . flatten .
  repeat k (advance . hoistS mhStep .
                      hoistS (hoistA resample)) .
  (hoistS . hoistA) (spawn n >>)
```

We only include a single `mhStep` transition for brevity, but it would be trivial to use multiple transitions at each step.

To finish this section we present an implementation of an algorithm called Particle Independent Metropolis Hastings (PIMH) [Andrieu et al., 2010] which uses SMC as a proposal distribution for MH. It relies on an inference transformation called `mhPrior`, which performs an MH step by proposing new values for all random variables jointly from the prior. With that in place we can implement a single step of PIMH in the following way.

```
pimh :: Int -> Int ->
        Sequential (Population Sampler) a ->
        Sampler a
pimh n k =
  mhPrior . hoistS (collapse . smc n k)
```

4 Discussion

In this section we discuss some issues related to usability of our framework. Since our library is a proof-of-concept system, we do not evaluate performance at this point.

First of all, we emphasize that Haskell is simply a convenient tool to express our ideas and is in no way necessary to implement them - an equivalent library could be built for any general-purpose programming language such as Python. Second, regarding monads, we believe that understanding monads is in no way necessary to build inference algorithms by composing existing building blocks as shown in the examples above. Understanding monads is only required to implement new layers and inference transformations, but we believe that the benefits offerred by our framework make it worthwhile for potential developers to invest time in learning the relevant concepts.

Apart from that, we would like to highlight how our modular architecture helps with testing randomized inference algorithms. Monte Carlo methods are difficult to test since they can produce arbitrarily bad results with non-zero probability, which results in a trade-off between significance and power of randomized tests [Radul, 2016]. However, in our framework we can replace the bottom `Sampler` layer with `Enumerator` and run an inference algorithm on a small discrete models to check if it targets exactly the correct posterior. Thus we can *deterministically check* that an MCMC kernel preserves the correct posterior distribution or that a resampling scheme does not introduce any bias, which we found invaluable in practice.

As a closing remark, we would like to express our hope that the framework presented in our paper, whether in the implementation described here or a different one, will be useful to researchers designing novel inference algorithms. We hope the modular style of implementation we advocate will be particularly useful for quick initial exploration of the design space of inference algorithms.

References

C. Andrieu, A. Doucet, and R. Holenstein. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society*, 72:269–342, 2010. URL `www.stats.ox.ac.uk/~doucet/andrieu_doucet_holenstein_PMCMC.pdf`.

O. Cappé, A. Guillin, J. M. Marin, and C. P. Robert. Population Monte Carlo. *Journal of Computational and Graphical Statistics*, 13:907–929, 2004. URL `http://amstat.tandfonline.com/doi/abs/10.1198/106186004X12803`.

M. Erwig and S. Kollmansberger. Probabilistic functional programming in haskell. *Journal of Functional Programming*, 16:21–34, 2006. URL `http://dl.acm.org/citation.cfm?id=1114013`.

W. Gilks and C. Berzuini. Following a moving target - Monte Carlo inference for dynamic Bayesian models. *Journal of the Royal Statistical Society*, 63:127–146, 2001. URL `www.mathcs.emory.edu/~whalen/Papers/BNs/MonteCarlo-DBNs.pdf`.

M. Giry. A categorical approach to probability theory. In *Categorical aspects of Topology and Analysis*. Springer, 1981. URL `http://link.springer.com/chapter/10.1007%2FBFb0092872`.

S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL*, 1995. URL `http://dl.acm.org/citation.cfm?id=199528`.

E. Moggi. Notions of computation and monads. In *LiCS*, 1989. URL `http://www.sciencedirect.com/science/article/pii/0890540191900524`.

C. Naesseth, F. Lindsten, and T. Schön. Nested sequential Monte Carlo methods. In *AISTATS*, 2014. URL `http://www.jmlr.org/proceedings/papers/v33/ranganath14.html`.

A. Radul. On testing probabilistic programs. `http://alexey.radul.name/ideas/2016/on-testing-probabilistic-programs/`, 2016.

R. Ranganath, S. Gerrish, and D. Blei. Black-box variational inference. In *AISTATS*, 2014. URL `http://www.jmlr.org/proceedings/papers/v33/ranganath14.html`.

A. Ścibior, Z. Ghahramani, and A. Gordon. Practical probabilistic programming with monads. In *Haskell*, 2015. URL `http://dl.acm.org/citation.cfm?id=2804317`.

P. Wadler. The essence of functional programming. In *POPL*, 1992. URL `http://dl.acm.org/citation.cfm?id=143169`.

D. Wingate, A. Stuhlmüller, and N. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *AISTATS*, 2011. URL `https://web.stanford.edu/~ngoodman/papers/lightweight-mcmc-aistats2011.pdf`. The published version contains a serious bug in the algorithm description, which was fixed in Revision 3 available from the authors page.

F. Wood, J.-W. van de Meent, and V. Mansinghka. A new approach to probabilistic programming inference. In *AISTATS*, 2014. URL `http://www.robots.ox.ac.uk/~fwood/assets/pdf/Wood-AISTATS-2014.pdf`.

R. Zinkov and C. Shan. Composing inference algorithms as program transformations. arXiv:1603.01882, 2016.