

Compositional approaches to scalable Bayesian computation

Darren Wilkinson

@darrenjw

Durham University, UK

darrenjw.github.io

Laplace's Demon Webinar

May 2022

Overview

Background

Bayesian computation

Category Theory

Probability monads

Conclusion

Background

Background

- For non-trivial problems, Bayesian computation typically relies on computationally intensive methods, such as MCMC, SMC or ABC
- These often require a custom implementation in some programming language
- All of the languages commonly used are very old, dating back to the dawn of the computing age, and are quite unsuitable for scalable and efficient Bayesian computation
- Interpreted dynamic languages such as R, Python and Matlab are far too slow, among many other things
- Languages such as C, C++, Java and Fortran are much faster, but are also very poorly suited to the development of efficient, well-tested, compositional, scalable code, able to take advantage of modern computing hardware

Alternative languages and approaches

- All of the languages on the previous slide are fundamentally **imperative** programming languages, mimicking closely the way computer processors actually operate
- There have been huge advances in computing science in the decades since these languages were created, and many new, different and better programming languages have been created
- Although **functional** programming (FP) languages are not new, there has been a large resurgence of interest in functional languages in the last decade or two, as people have begun to appreciate the advantages of the functional approach, especially in the context of developing large, scalable software systems, and the ability to take advantage of modern computing hardware
- There has also been a swing away from **dynamically typed** programming languages back to **statically typed** languages

Functional programming

- FP languages emphasise the use of **immutable** data, **pure**, **referentially transparent functions**, and **higher order functions**
- FP languages more naturally support composition of models, data and computation than imperative languages, leading to more scalable and testable software
- Statically typed FP languages (such as Haskell and Scala) correspond closely to the **simply-typed lambda calculus** which is one of the canonical examples of a **Cartesian closed category** (CCC)
- This connection between typed FP languages and CCCs enables the borrowing of ideas from category theory into FP
- Category theory concepts such as **functors**, **monads** and **comonads** are useful for simplifying code that would otherwise be somewhat cumbersome to express in pure FP languages

Concurrency, parallelism, distribution, state

- Modern computing platforms feature processors with many cores, and possibly many such processors — parallel programming is required to properly exploit these
- Most of the notorious difficulties associated with parallel programming revolve around **shared mutable state**
- In pure FP, state is not mutable, so there is no mutable state, shared or otherwise
- Consequently, most of the difficulties typically associated with parallel, distributed, and concurrent programming simply don't exist in FP — parallelism in FP is so easy and natural that it is sometimes completely automatic
- This natural scalability of FP languages is one reason for their recent resurgence

Compositionality

- Not all issues relating to scalability of models and algorithms relate to parallelism
- A good way to build a large model is to construct it from smaller models
- A good way to develop a complex computation is to construct it from simpler computations
- This (recursive) decomposition-composition approach is at the heart of the so-called “divide and conquer” approach to problem solution, and is very natural in FP (eg. FFT and BP for PGMs)
- It also makes code much easier to **test** for correct behaviour
- Category theory is in many ways the mathematical study of (associative) composition, and this leads to useful insights

Bayesian computation

Bayesian computation

- **map-reduce** operations on **functorial** data collections can trivially parallelise (and distribute):
 - Likelihood evaluations for big data
 - ABC algorithms
 - SMC re-weighting and re-sampling
- Gibbs sampling algorithms can be implemented as **cobind** operations on an appropriately coloured (parallel) **comonadic** conditional independence graph
- **Probabilistic programming languages** (PPLs) can be implemented as embedded domain specific languages (DSLs) trivially using **for/do** syntax for **monadic composition** in conjunction with **probability monads**
- **Automatic differentiation** (AD) is natural and convenient in functional languages, facilitating gradient-based algorithms such as MALA, HMC and PDMP

Parallel immutable collections

- Using `map` to apply a `pure` function to all of the elements in a collection can clearly be done in parallel
- So if the collection contains n elements, then the computation time can be reduced from $O(n)$ to $O(1)$ (on infinite parallel hardware)
 - `Vector(3,5,7) map (_ * 2) = Vector(6,10,14)`
- We can carry out `reductions` as `folds` over collections:
`Vector(6,10,14) reduce (_ + _) = 30`
- In general, sequential folds can not be parallelised, but...

Monoids and parallel “map–reduce”

- A **monoid** is a very important concept in FP
- For now we will think of a monoid as a **set** of elements with a **binary relation** \star which is **closed** and **associative**, and having an **identity** element wrt the binary relation
- You can think of it as a **semi-group** with an identity or a **group** without an inverse
- **folds**, **scans** and **reduce** operations can be computed in parallel using **tree reduction**, reducing time from $O(n)$ to $O(\log n)$ (on infinite parallel hardware)
- “**map–reduce**” is just the pattern of processing large amounts of data in an immutable collection by first **mapping** the data (in parallel) into a monoid and then **tree-reducing** the result (in parallel), sometimes called **foldMap**

Distributed parallel collections with Apache Spark

- **Apache Spark** (`spark.apache.org`) is a Scala library for distributed Big Data processing on (large) clusters of machines
- The basic datatype provided by Spark is an **RDD** — a resilient distributed dataset
- An RDD is just a **lazy, distributed**, parallel monadic collection, supporting methods such as `map`, `flatMap`, `reduce`, etc., which can be used in exactly the same way as any other Scala collection
- Code looks exactly the same whether the RDD is a small dataset on a laptop or terabytes in size, distributed over a large Spark cluster
- It is a powerful framework for the development of scalable algorithms for Bayesian computation

- **JAX** (jax.readthedocs.io) is a Python library embedding a DSL for automatic differentiation and JIT-compiling (array) functions to run very fast on (multiple) CPU or GPU
- It is especially good at speeding up likelihood evaluations and (MCMC-based) sampling algorithms for complex models
- It is not unheard of for MCMC algorithms to run 100 times faster than regular Python code, even on a single CPU (multiple cores and GPUs will speed things up further)
- The JAX eDSL is **pure functional array language**
- Despite targeting a completely different kind of scalability to Spark, and being embedded in a very different language, the fundamental computational model is very similar: **express algorithms in terms of lazy transformations of immutable data structures using pure functions**

Functional algorithms

- By expressing algorithms in a functional style (eg. lazy transformations of immutable data structures with pure functions), we allow many code optimisations to be automatically applied
- Pure functional algorithms are relatively easy to analyse, optimise, compile, parallelise, distribute, differentiate, push to GPU, etc.
- These optimisations can typically be done automatically by the library, compiler, framework, etc., without significant user intervention
- It is very difficult (often impossible) to analyse and reason about imperative code in a similar way

JAX ecosystem

- (Unnormalised) log-posteriors expressed in JAX can be sampled using a variety of different algorithms (including HMC and NUTS, via auto-diff) using **BlackJAX** (blackjax-devs.github.io/blackjax/)
- The original Pyro uses PyTorch as a back-end, but the popular **NumPyro** (<https://github.com/pyro-ppl/numpyro>) fork uses JAX for a back-end, leading to significant performance improvements
- The PyMC4 project intended to use TensorFlow for a back-end, but this project was abandoned, and **PyMC3** (docs.pymc.io) is now in the process of switching from Theano to JAX as a back-end (currently experimental, via NumPyro)

- Although the conceptual computational model of JAX has a number of good features, the embedding of such a language in a dynamic, interpreted, imperative language such as Python has a number of limitations and drawbacks
- A similar issue arises with Spark – although it is possible to develop Spark applications in Python using PySpark, in practice most non-trivial applications are developed in Scala, for good reason
- This motivates the development of a JAX-like array processing DSL in a strongly typed functional programming language
- **DEX** (github.com/google-research/dex-lang) is a new experimental (Haskell-like) language in this space with a number of interesting and desirable features

MCMC for a fully Bayesian logistic regression model

github.com/darrenjw/logreg

- (WIP) repo containing code examples for RW MH and MALA MCMC algorithms for a Bayesian analysis of a logistic regression model
- Code in R, rjags, rstan, Python (with numpy), Python (with JAX), BlackJAX, Scala, Spark, etc.
- Running on a single core on my laptop, a JAX version of an MCMC scheme runs over 100 times faster than the same algorithm in regular Python with numpy
- *tl;dr* if you have a small-to-medium sized problem, and the only thing you care about is speed, somehow express your log-posterior in JAX and then sample it with BlackJAX

Category Theory

Category theory

- A category \mathcal{C} consists of a collection of **objects**, $\text{ob}(\mathcal{C})$, and **morphisms**, $\text{hom}(\mathcal{C})$. Each morphism is an ordered pair of objects (an arrow between objects). For $x, y \in \text{ob}(\mathcal{C})$, the set of morphisms from x to y is denoted $\text{hom}_{\mathcal{C}}(x, y)$.
 $f \in \text{hom}_{\mathcal{C}}(x, y)$ is often written $f : x \longrightarrow y$.
- Morphisms are closed under **composition**, so that if $f : x \longrightarrow y$ and $g : y \longrightarrow z$, then there must also exist a morphism $h : x \longrightarrow z$ written $h = g \circ f$.
- Composition is associative, so that $f \circ (g \circ h) = (f \circ g) \circ h$ for all composable $f, g, h \in \text{hom}(\mathcal{C})$.
- For every $x \in \text{ob}(\mathcal{C})$ there exists an **identity** morphism $\text{id}_x : x \longrightarrow x$, with the property that for any $f : x \longrightarrow y$ we have $f = f \circ \text{id}_x = \text{id}_y \circ f$.

Functors

- A **functor** is a mapping from one category to another which preserves some structure
- A functor F from \mathcal{C} to \mathcal{D} , written $F : \mathcal{C} \longrightarrow \mathcal{D}$ is a pair of functions (both denoted F):
 - $F : \text{ob}(\mathcal{C}) \longrightarrow \text{ob}(\mathcal{D})$
 - $F : \text{hom}(\mathcal{C}) \longrightarrow \text{hom}(\mathcal{D})$, where $\forall f \in \text{hom}(\mathcal{C})$, we have $F(f : x \longrightarrow y) : F(x) \longrightarrow F(y)$
 - In other words, if $f \in \text{hom}_{\mathcal{C}}(x, y)$, then $F(f) \in \text{hom}_{\mathcal{D}}(F(x), F(y))$
- The functor must satisfy the **functor laws**:
 - $F(\text{id}_x) = \text{id}_{F(x)}, \forall x \in \text{ob}(\mathcal{C})$
 - $F(f \circ g) = F(f) \circ F(g)$ for all composable $f, g \in \text{hom}(\mathcal{C})$
- A functor $F : \mathcal{C} \longrightarrow \mathcal{C}$ is called an **endofunctor** — in the context of functional programming, the word functor usually refers to an endofunctor $F : \mathbf{Set} \longrightarrow \mathbf{Set}$

Laziness, composition, laws and optimisations

- Laziness allows some optimisations to be performed that would be difficult to automate otherwise
- Consider a dataset `rdd: RDD[T]`, functions `f: T => U`, `g: U => V`, and a binary operation `op: (V,V) => V` for monoidal type `V`
- We can map the two functions and then reduce with:
 - `rdd map f map g reduce op`
 - to get a value of type `V`, all computed in parallel
- However, re-writing this as:
 - `rdd map (g compose f) reduce op`
 - would eliminate an intermediate collection, but is equivalent due to the 2nd functor law (`map fusion`)
- Category theory **laws** often correspond to **optimisations** that can be applied to code without affecting results — Spark can do these optimisations **automatically** due to lazy evaluation

Natural transformations

- Often there are multiple functors between pairs of categories, and sometimes it is useful to be able to transform one to another
- Suppose we have two functors $F, G : \mathcal{C} \longrightarrow \mathcal{D}$
- A **natural transformation** $\alpha : F \Rightarrow G$ is a family of morphisms in \mathcal{D} , where $\forall x \in \mathcal{C}$, the **component** $\alpha_x : F(x) \longrightarrow G(x)$ is a morphism in \mathcal{D}
- To be considered **natural**, this family of morphisms must satisfy the **naturality law**:
 - $\alpha_y \circ F(f) = G(f) \circ \alpha_x, \quad \forall f : x \longrightarrow y \in \text{hom}(\mathcal{C})$
- **Naturality** is one of the most fundamental concepts in category theory
- In the context of FP, a natural transformation could (say) map an **Option** to a **List** (with at most one element)

Monads

- A **monad** on a category \mathcal{C} is an endofunctor $T : \mathcal{C} \longrightarrow \mathcal{C}$ together with two natural transformations $\eta : \text{Id}_{\mathcal{C}} \longrightarrow T$ (**unit**) and $\mu : T^2 \longrightarrow T$ (**multiplication**) fulfilling the **monad laws**:
 - **Associativity**: $\mu \circ T\mu = \mu \circ \mu_T$, as transformations $T^3 \longrightarrow T$
 - **Identity**: $\mu \circ T\eta = \mu \circ \eta_T = 1_T$, as transformations $T \longrightarrow T$
- The associativity law says that the two ways of **flattening** $T(T(T(x)))$ to $T(x)$ are the same
- The identity law says that the two ways of **lifting** $T(x)$ to $T(T(x))$ and then flattening back to $T(x)$ both get back to the original $T(x)$
- In FP, we often use **M** (for monad) rather than T (for triple), and say that there are three monad laws — the identity law is considered to be two separate laws

Kleisli category

- Kleisli categories formalise monadic composition
- For any monad T over a category \mathcal{C} , the **Kleisli category** of \mathcal{C} , written \mathcal{C}_T is a category with the same objects as \mathcal{C} , but with morphisms given by:
 - $\text{hom}_{\mathcal{C}_T}(x, y) = \text{hom}_{\mathcal{C}}(x, T(y)), \forall x, y \in \text{ob}(\mathcal{C})$
- The identity morphisms in \mathcal{C}_T are given by $\text{id}_x = \eta(x), \forall x$, and morphisms $f : x \longrightarrow T(y)$ and $g : y \longrightarrow T(z)$ in \mathcal{C} can compose to form $g \circ_T f : x \longrightarrow T(z)$ via
 - $g \circ_T f = \mu_z \circ T(g) \circ f$leading to composition of morphisms in \mathcal{C}_T .
- In FP, the morphisms in \mathcal{C}_T are often referred to as **Kleisli arrows**, or **Kleislis**, or sometimes just **arrows** (although **Arrow** usually refers to a generalisation of Kleisli arrows, sometimes known as **Hughes arrows**)

Comonads

- The comonad is the categorical dual of the monad, obtained by “reversing the arrows” for the definition of a monad
- A **comonad** on a category \mathcal{C} is an endofunctor $W : \mathcal{C} \longrightarrow \mathcal{C}$ together with two natural transformations $\varepsilon : W \longrightarrow \text{Id}_{\mathcal{C}}$ (**counit**) and $\delta : W \longrightarrow W^2$ (**comultiplication**) fulfilling the **comonad laws**:
 - **Associativity**: $\delta_W \circ \delta = W\delta \circ \delta$, as transformations $W \longrightarrow W^3$
 - **Identity**: $\varepsilon_W \circ \delta = W\varepsilon \circ \delta = 1_W$, as transformations $W \longrightarrow W$
- The associativity law says that the two ways of **duplicating** a $W(x)$ duplicated to a $W(W(x))$ to a $W(W(W(x)))$ are the same
- The identity law says that the two ways of **extracting** a $W(x)$ from a $W(x)$ duplicated to a $W(W(x))$ are the same

Comonads for statistical computation

- Monads are good for sequencing operations that can be carried out on data points independently
- For computations requiring knowledge of some kind of local neighbourhood structure, **Comonads** are a better fit
- `coflatMap` will take a function representing a **local computation** producing one value for the new structure, and then extend this to generate all values associated with the comonad
- Useful for defining linear filters, Gibbs samplers, convolutional neural networks, etc.
- Concepts from category theory are often useful for providing abstractions which make convenient computations that might otherwise seem cumbersome within a pure functional programming language

Probability monads

Composing random variables with the probability monad

- The **probability monad** provides a foundation for describing random variables in a pure functional way (cf. **Giry monad**)
- We can build up joint distributions from marginal and conditional distributions using **monadic composition**
- For example, consider an exponential mixture of Poissons (marginally negative binomial): we can think of an exponential distribution parametrised by a rate as a function `Exponential: Double => Rand[Double]` and a Poisson parametrised by its mean as a function `Poisson: Double => Rand[Int]`
- Those two functions don't directly compose, but do in the Kleisli category of the `Rand` monad, so `Exponential(3) flatMap {Poisson(_)}` will return a `Rand[Int]` which we can draw samples from if required

Monads for probabilistic programming

- For larger probability models we can use **for-comprehensions** to simplify the model building process, eg.

```
for { mu <- Gaussian(10,1)
      tau <- Gamma(1,1)
      sig = 1.0/sqrt(tau)
      obs <- Gaussian(mu,sig) }
yield ((mu,tau,obs))
```

- We can use a regular probability monad for building forward models this way, and even for building models with simple Bayesian inference procedures allowing conditioning
- For sophisticated probabilistic sampling algorithms (eg. SMC, MCMC, pMCMC, HMC, ...) and hybrid compositions, it is better to build models like this using a **free monad** which can be **interpreted** in different ways

Composing probabilistic programs

- Describing probabilistic programs as **monadic values** in a functional programming language with syntax for monadic composition leads immediately to an **embedded PPL DSL** “for free”
- This in turn enables a fully **compositional** approach to the (scalable) development of (hierarchical) probabilistic models
- Model components can be easily “re-used” in order to build big models from small
- eg. a regression model component can be re-used to create a hierarchical random effects model over related regressions
- In some well-known PPLs, this would require manual copy-and-pasting of code, wrapping with a “for loop”, and manual hacking in of an extra array index into all array references — not at all compositional!

Representation independence using the free monad

- However you implement your probability monad, the semantics of your probabilistic program are (essentially) the same
- It would be nice to be able to define and compose probabilistic programs independently of concerns about implementation, and then to **interpret** the program with a particular implementation later
- Building a probability monad on top of the **free monad** allows this — implementation of **pure** and **flatMap** is “suspended” in a way that allows subsequent interpretation with concrete implementations later
- This allows **layering** of multiple inference algorithms, and different interpretation of different parts of the model, enabling sophisticated **composition** of different (hybrid) inference algorithms

Bayesian hierarchical models, graphical models and probabilistic programs

- Bayesian hierarchical models are one of the most powerful tools we have for understanding large and complex data sets — their hierarchical nature makes them naturally compositional
- Bayesian graphical models are also quite compositional, and can often be used to represent Bayesian hierarchical models
- Bayesian hierarchical and graphical models can be modelled as probabilistic programs, so probabilistic programming can be used as a unifying framework for describing (Bayesian) statistical models
- We want our probabilistic programming languages to be compositional (functional and monadic?), in order to naturally scale with model size and complexity

Compositionality of inference algorithms








- As well as building **models** in scalable, compositional way, we would also like our **inference algorithms** to be compositional, ideally reflecting the compositional structure of our models
- Some algorithms, such as component-wise samplers and message-passing algorithms, naturally reflect the compositional structure of the underlying model
- Other algorithms, such as Langevin and HMC samplers, deliberately don't decompose with the model structure, but do have other structure that can be exploited, such as decomposing over observations
- Understanding and exploiting the **compositional structure** of models and algorithms will be crucial for developing scalable inferential methods

Conclusion

Summary

- All of the programming languages commonly used for Bayesian computation are poorly suited to the task
- Functional programming languages borrowing ideas from **Cartesian closed categories** provide an excellent foundation for scalable, compositional modelling and computation, including probabilistic and differentiable (array) programming
 - Concepts from category theory, such as **functors**, **monads** and **comonads** provide a useful framework for organising computation
- This isn't about one particular programming language or language syntax — it is about a class of languages (statically typed compiled functional programming languages with type inference and higher-kinded types) and algorithmic approaches (eg. lazy transformation of immutable data structures with pure functions)

References

-  Elliott, C. (2017) [Generic functional parallel algorithms: Scan and FFT](#). *ICFP 2017*, 1:7.
-  Fong, B., Spivak, D., Tuyéras, R. (2019) [Backprop as Functor: A compositional perspective on supervised learning](#), *arXiv*, 1711.10455.
-  Law, J., Wilkinson, D. J. (2019) [Functional probabilistic programming for scalable Bayesian modelling](#), *arXiv*, 1908.02062.
-  van der Meulen, F., Schauer, M. (2020) [Automatic backward filtering forward guiding for Markov processes and graphical models](#), *arXiv*, 2010.03509.
-  Paszke, A. et al (2021) [Getting to the point. Index sets and parallelism-preserving autodiff for pointful array programming](#), *arXiv*, 2104.05372.
-  Särkkä, S., García-Fernández, A. F. (2019) [Temporal parallelization of Bayesian smoothers](#), *arXiv*, 1905.13002.
-  Ścibior, A., Kammar, O., Gharamani, Z. (2018) [Functional programming for modular Bayesian inference](#), *Proc. ACM Prog. Lang.*, 2(ICFP): 83.

Further information: github.com/darrenjw/talks

darrenjw.github.io

@darrenjw