

Functional languages for data science (at scale)

Darren Wilkinson

@darrenjw

Durham University, UK

darrenjw.github.io

DUDSS meeting

19th January 2023

github.com/darrenjw/talks

Current data science languages

- Currently, the most popular programming languages for data science are R and Python
 - R is a language specifically designed for statistical computing, and is great for exploratory data analysis, visualisation and statistical modelling
 - Python is a general purpose programming language that is very effective for data science when used in conjunction with appropriate libraries, and has good interfaces to a variety of specialised machine learning frameworks
- All aspiring data scientists should be very familiar with at least one of R and Python (ideally both), and nothing in this talk changes that!
- In this talk we will examine the limitations of commonly used languages, and why modern functional programming languages might make a better future data science platform

Background

- For non-trivial problems, statistical computation and machine learning typically relies on computationally intensive methods
- 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 statistical 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

What does ChatGPT think?



Why are functional programming languages good for data science and machine learning?



Functional programming languages, such as Haskell and Lisp, are good for data science and machine learning because they encourage immutability, which can make it easier to reason about and debug code. They also often have strong support for handling large data sets and parallel processing, which can improve performance when working with large amounts of data. Additionally, functional programming languages often have powerful type systems, which can help prevent errors and make it easier to understand and maintain code.



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

Imperative pseudo-code

```
1: function MONTEC( $n$ )
2:    $x \leftarrow 0$ 
3:   for  $i \leftarrow 1$  to  $n$  do
4:     draw  $u \sim U(0, 1)$ 
5:     draw  $v \sim U(0, 1)$ 
6:     if  $u^2 + v^2 < 1$  then
7:        $x \leftarrow x + 1$ 
8:     end if
9:   end for
10:  return  $4x/n$ 
11: end function

1: function METROP( $n, \varepsilon$ )
2:    $x \leftarrow 0$ 
3:   for  $i \leftarrow 1$  to  $n$  do
4:     draw  $z \sim U(-\varepsilon, \varepsilon)$ 
5:      $x' \leftarrow x + z$ 
6:      $A \leftarrow \phi(x')/\phi(x)$ 
7:     draw  $u \sim U(0, 1)$ 
8:     if  $u < A$  then
9:        $x \leftarrow x'$ 
10:    end if
11:  end for
12:  return  $x$ 
13: end function
```

Not obvious that one of these is naturally parallel. . .

Functional programming

- In pure FP, all state is **immutable** — you can assign names to things, but you can't change what the name points to — no “variables” in the usual sense
- Functions are **pure** and **referentially transparent** — they can't have side-effects — they are just like functions in mathematics...
- Functions can be recursive, and **recursion** can be used to iterate over recursive data structures — useful since no conventional “for” or “while” loops in pure FP languages
- Functions are first class objects, and **higher-order functions** (HOFs) are used extensively — functions which return a function or accept a function as argument

Ideal languages for statistical computing and ML

- We should approach the problem of statistical modelling and efficient computation in a modular, composable, functional way
- To do this we need programming languages which are:
 - **Strongly statically typed** (but with type inference)
 - **Compiled** (but possibly to a VM)
 - **Functional** (with support for immutable values, immutable collections, ADTs and higher-order functions)
 - and have support for **typeclasses** and **higher-kinded types**, allowing the adoption of design patterns from **category theory**
- For efficient statistical computing, it can be argued that evaluation should be **strict** rather than **lazy** by default
- **Scala** is a popular language which meets the above constraints

What does ChatGPT think?



Why is Scala good for data engineering and big data processing?



Scala is good for data engineering and big data processing because it is a functional programming language that runs on the JVM (Java Virtual Machine) and can make use of Java's vast ecosystem of libraries and tools. This makes it a powerful and efficient option for working with large data sets and distributed systems.



Scala also has strong support for functional programming, which makes it easy to write code that is concise, expressive, and easy to reason about. It also has a rich set of libraries and frameworks like Apache Spark, Apache Kafka, and Apache Storm, which makes it well-suited for big data processing tasks like data streaming, batch processing, and real-time analytics.

Furthermore, Scala's strong type system and its ability to handle concurrency and parallelism, makes it an ideal choice for data engineering and big data processing tasks.

Monadic collections (in Scala)

- A collection of type `M[T]` can contain (multiple) values of type `T`
- If the collection supports a higher-order function `map(f: T => S): M[S]` then we call the collection a **Functor**
 - eg. `List(1,3,5,7) map (x => x*2) = List(2,6,10,14)`
- If the collection additionally supports a higher-order function `flatMap(f: T => M[S]): M[S]` then we call the collection a **Monad**
 - eg. `List(1,3,5,7) flatMap (x => List(x,x+1))`
`= List(1, 2, 3, 4, 5, 6, 7, 8)`
 - instead of `List(1,3,5,7) map (x => List(x,x+1))`
`= List(List(1,2),List(3,4),List(5,6),List(7,8))`

[Scastie]

Other monadic types: Option

- Some computations can fail, and we can capture that possibility with a type called `Option`
 - in Scala — it is `Optional` in Java 8 and `Maybe` in Haskell
- An `Option[T]` can contain `Some[T]` or `None`
- So if we have `chol: Matrix =>Option[TriMatrix]` we can check to see if we have a result
- But if we also have `triSolve: (TriMatrix,Vector) =>Option[Vector]`, how do we “compose” these?
 - `chol(mat) map (tm =>triSolve(tm,vec))` has type `Option[Option[Vector]]` which isn't quite what we want
 - `chol(mat) flatMap (tm =>triSolve(tm,vec))` has type `Option[Vector]` which we do want
 - `flatMap` allows **composition** of monadic functions

Composing monadic functions

- Given functions $f: S \Rightarrow T$, $g: T \Rightarrow U$, $h: U \Rightarrow V$, we can compose them as $h \text{ compose } g \text{ compose } f$ or $s \Rightarrow h(g(f(s)))$ to get $hgf: S \Rightarrow V$
- Monadic functions $f: S \Rightarrow M[T]$, $g: T \Rightarrow M[U]$, $h: U \Rightarrow M[V]$ don't compose directly, but do using `flatMap`:
 $s \Rightarrow f(s) \text{ flatMap } g \text{ flatMap } h$ has type $S \Rightarrow M[V]$
- Can be written as a **for-comprehension** (**do** in Haskell):
 $s \Rightarrow \text{for } (t \leftarrow f(s); u \leftarrow g(t); v \leftarrow h(u)) \text{ yield } v$
- Just syntactic sugar for the chained `flatMap`s above — really **not** an imperative-style “for loop” at all...

Other monadic types: Future

- A `Future[T]` is used to dispatch a (long-running) computation to another thread to run in parallel with the main thread
- When a `Future` is created, the call returns immediately, and the main thread continues, allowing the `Future` to be “used” before its result (of type `T`) is computed
- `map` can be used to transform the result of a `Future`, and `flatMap` can be used to chain together `Futures` by allowing the output of one `Future` to be used as the input to another
- `Futures` can be transformed using `map` and `flatMap` irrespective of whether or not the `Future` computation has yet completed and actually contains a value
- `Futures` are a powerful method for developing parallel and concurrent programs in a modular, composable way

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 statistical computing and machine learning

- **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**



How will using JAX improve my Python machine learning code?



JAX is a library for machine learning in Python that uses XLA (Accelerated Linear Algebra) to accelerate the performance of numerical computations. Using JAX can improve your Python machine learning code in a few ways:



1. **Speed:** JAX can accelerate the performance of your code by using XLA to optimize and parallelize the computations, which can lead to significant speedups compared to using standard NumPy or other libraries.
2. **Automatic differentiation:** JAX provides automatic differentiation capabilities, which allows you to easily compute gradients of your machine learning models. This can make it easier to implement optimization algorithms like stochastic gradient descent and backpropagation.
3. **Hardware support:** JAX supports a variety of hardware platforms, including CPUs, GPUs, and TPUs, which makes it easy to take advantage of the available hardware acceleration to speed up your computations.
4. **Easy to use:** JAX's API is designed to be similar to that of NumPy, which makes it easy to use and integrate with other libraries and frameworks.
5. **Portability:** JAX functions are pure Python, and can be run in any environment, including on a CPU, GPU, or TPU, without modification.

Overall, JAX is a powerful tool that can help you write more efficient and effective machine learning code in Python.

What frameworks can I use to automate building neural network models with JAX?



There are several frameworks available that can automate building neural network models with JAX:



1. **Flax:** Flax is a neural network library built on top of JAX that provides a high-level, Keras-like API for building and training neural networks. It makes it easy to define complex models, including transformer and attention-based architectures.
2. **Optax:** Optax is a library for neural network optimization built on top of JAX. It provides a high-level, modular API for optimizers and schedules, making it easy to define and implement different optimization strategies.
3. **Haiku:** Haiku is a neural network library built on top of JAX that provides a simple, expressive, and easy-to-use API for building neural networks. It is designed to be minimal and easy to understand, making it a good choice for beginners.
4. **Trax:** Trax is a neural network library built on top of JAX that provides a high-level, Keras-like API for building and training neural networks. It includes support for transformer and attention-based architectures, as well as a variety of pre-trained models.
5. **JaxNet:** JaxNet is a neural network library built on top of JAX that provides a high-level, PyTorch-like API for building and training neural networks. It makes it easy to define complex models, including transformer and attention-based architectures.

All of these frameworks provide a high-level and easy-to-use API for building and training neural networks with JAX, and can help you automate the process of building and training neural networks, making it faster and more convenient.

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

- 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



My apologies, I misunderstood your question. Dex is a pointful array processing language developed by Google, which is used to express array computations in a more natural and readable way.



Dex is a domain-specific programming language for array computations, it aims to express array computations in a more natural and readable way. It is designed to be expressive, readable, and easy to use. It is used to express array computations in a more natural and readable way.

Dex is useful for:

1. Scientific computing: Dex can be used to express complex array computations in a simple and natural way, making it well-suited for scientific computing tasks such as numerical simulations and data analysis.
2. Machine learning: Dex can be used to express array computations in a more natural and readable way, making it well-suited for machine learning tasks such as training neural networks and other machine learning models.
3. High-performance computing: Dex is designed to be highly efficient and can be used to express array computations that are performant on GPUs and other high-performance computing hardware.
4. Data engineering: Dex can be used to express array operations in a more natural and readable way, making it well-suited for data engineering tasks such as data processing and data analysis.

Overall, Dex is a powerful and expressive language that can be used to express array computations in a more natural and readable way, making it well-suited for a wide range of scientific, machine learning, and data engineering tasks.

MCMC for a fully Bayesian logistic regression model

github.com/darrenjw/logreg

- Repo containing code examples for various MCMC algorithms for a Bayesian analysis of a logistic regression model
- Code in R, rjags, rstan, Python (with numpy), Python (with JAX), BlackJAX, numPyro, PyMC, Scala, Spark, Haskell, Dex, C
- 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 relatively simple small-to-medium sized Bayesian model, and the only thing you care about is speed, somehow express your log-posterior in JAX and then sample it with BlackJAX

Conclusions

- We should approach the problem of statistical modelling and computation in a modular, composable, functional way, guided by underpinning principles from category theory
- To implement solutions to problems in statistical modelling and computation in a more scalable way, we need programming languages which are:
 - Strongly statically typed
 - Compiled
 - Functional
 - and support concurrency, parallelism, typeclasses and higher-kinded types
- Scala and Spark provide a nice illustration of the power of this approach, but there are other interesting languages, including: Haskell, Idris, OCaml, Dex, . . .

Summary

- All of the programming languages commonly used for statistical computation and ML 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)

Further information

<https://github.com/darrenjw/talks/>