The Essential Tools of Scientific Machine Learning (Scientific ML)

Christopher Rackauckas[1]

[1]Affiliation not available

April 17, 2023

the WINNOWER

MATHEMATICS

π

The Essential Tools of Scientific Machine Learning (Scientific ML)

CHRISTOPHER RACKAUCKAS

CORRESPONDENCE:
me@chrisrackauckas.com

DATE RECEIVED:
August 20, 2019

DOI:
10.15200/winn.156631.13064

ARCHIVED:
August 20, 2019

CITATION:
Christopher Rackauckas, The Essential Tools of Scientific Machine Learning (Scientific ML), *The Winnower* 6:e156631.13064 , 2019 , DOI: 10.15200/winn.156631.13064

Scientific machine learning is a burgeoning discipline which blends scientific computing and machine learning. Traditionally, scientific computing focuses on large-scale mechanistic models, usually differential equations, that are derived from scientific laws that simplified and explained phenomena. On the other hand, machine learning focuses on developing non-mechanistic data-driven models which require minimal knowledge and prior assumptions. The two sides have their pros and cons: differential equation models are great at extrapolating, the terms are explainable, and they can be fit with small data and few parameters. Machine learning models on the other hand require "big data" and lots of parameters but are not biased by the scientists ability to correctly identify valid laws and assumptions.

However, the recent trend has been to merge the two disciplines, allowing explainable models that are data-driven, require less data than traditional machine learning, and utilize the knowledge encapsulated in centuries of scientific literature. The promise is to fuse a priori domain knowledge which doesn't fit into a "dataset", allow this knowledge to specify a general structure that prevents overfitting, reduces the number of parameters, and promotes extrapolatability, while still utilizing machine learning techniques to learn specific unknown terms in the model. This has started to be used for outcomes like automated hypothesis generation and accelerated scientific simulation.

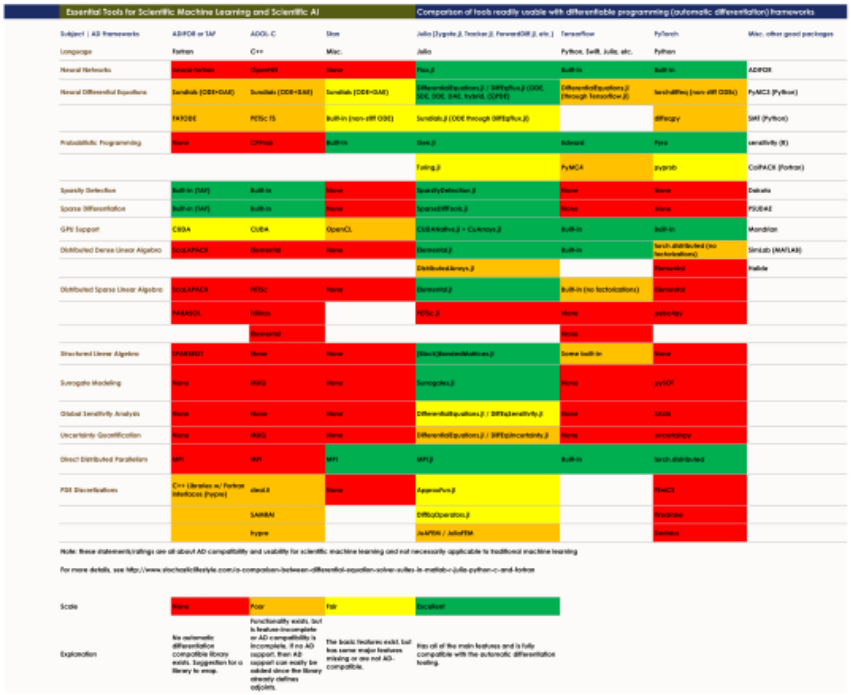
The purpose of this blog post is to introduce the reader to the tools of scientific machine learning, identify how they come together, and showcase the existing open source tools which can help one get started. We will be focusing on differentiable programming frameworks in the major languages for scientific machine learning: C++, Fortran, Julia, MATLAB, Python, and R.

We will be comparing two important aspects: efficiency and composability. Efficiency will be taken in the context of scientific machine learning: by now most tools are well-optimized for the giant neural networks found in traditional machine learning, but, as will be discussed here, that does not necessarily make them efficient when deployed inside of differential equation solvers or when mixed with probabilistic programming tools. Additionally, composability is a key aspect of scientific machine learning since our toolkit is not ML in isolation. Our goal is not to do machine learning as seen in a machine learning conference (classification, NLP, etc.), and it's not to do traditional machine learning as applied to scientific data. Instead, we are putting ML models and techniques into the heart of scientific simulation tools to accelerate and enhance them. Our neural networks need to fully integrate with tools that simulate satellites and robotics simulators. They need to integrate with the packages that we use in our scientific work for verifying numerical accuracy, tracking units, estimating uncertainty, and much more. We need our neural networks to play nicely with existing packages for delay

differential equations or reconstruction of dynamical systems. Otherwise we need to write the entire toolchain from scratch! While writing a neural network framework may be a good undergraduate project with modern tools, writing a neural network framework plus adaptive stiff differential equation solvers plus a robotics simulation platform plus automatic differentiation plus probabilistic programming tooling plus ... is infeasible. Instead of trying to reinvent 60 years of scientific computing for every new scientific ML for every new project, we need some way to make use of existing tools within the domain contexts!

QUICK SUMMARY TABLE

The following is a quick summary table of the tools to check out in each of the languages, with a color indicator for the efficiency and composability of these components.



Note that these indicators are for the use case of scientific ML, which is described in detail below, and not indicative of the support these AD systems give in traditional machine learning tasks.

OVERVIEW OF SCIENTIFIC MACHINE LEARNING AND SCIENTIFIC AI

First let's give a quick overview of scientific machine learning. There are three overview resources that I would point to. For a broad overview of the topic in a position paper, the technical report from the workshop on "Basic Research Needs for Scientific Machine Lea\nrning" is a good overview of what people care about in the field. The ICERM Scientific Machine Learning Workshop page has quite a few videos on the topic. And lastly, I myself have a video giving an overview of different applications of scientific ML and scientific AI:

From these resources you can see that there are some common questions, for example:

- How can you accelerate the solution of partial differential equations using deep learning?
- How can you automatically identify dynamical systems and scientific laws?
- How can you train neural networks so that their resulting model generates an explainable hypothesi\ns?

- How can you utilize existing data to utilize scientific simulators to allow for training networks with minimal or sparse da\nta?
- If you put a neural network as a controller to some system, how can you ensure its behavior is "safe"?

Some of these questions are starting to get answers, others are still just questions. **The key idea behind scientific machine learning is that applications of machine learning need to be approached from the context of the scientific domains with their existing tools and knowledge**. While training a neural network to identify fraudulent credit card transactions is the end of the story (you have a good predictor, yay!), whereas in scientific ML, making a prediction is just one step among many in the scientific process.

For example, in systems biology and quantitative systems pharmacology, the ordinary differential equation models encode the known structure of the chemical reaction networks. To a biologist or pharmacologist, the Oregonator system:

$$\begin{aligned}\frac{dx}{dt} &= s(y - xy + x - qx^2)\\ \frac{dy}{dt} &= (-y - xy + z)/s\\ \frac{dz}{dt} &= w(x - z)\end{aligned}$$

is saying that protein z is upregulated by x and has linear decay. There are many possible ways to predict a time series, but this is the one generated from first principles of the chemical reaction network, and will do well in extrapolating to areas where you don't have data because it has a basis in what we already know about how these biological systems work. While a neural network can be trained to predict the time series on the data that it is trained on, it may not have the ability to predict bifurcations where the dynamics occurs completely differently but we don't have data: but prediction of bifurcations is the bread and butter of mathematical biology.

Also, a trained neural network doesn't necessarily spit out hypotheses. When doing an investigation of pharmacological or climate models, one develops sets of differential equations and showcases how a given term is necessary for specific behavior such as oscillations to occur. Sometimes, dynamical outcomes can predict the existence of some unknown chemical factor since other feedbacks may be required for a differential equation to exhibit some global geometric result. So okay, you trained a neural network so it matches the timeseries... does it say that we're missing some important chemical species? Does it say that a critical transition will occur if the climate enters a new regime (where we have no \ndata)?

For a long time scientific computing has kept machine learning at an arm's length because its lack of interpretability and structure mean that, despite it's tremendous predicitve success, it is almost useless for answering these kinds of central scientific questions. While it could be used to great effect in some scenarios, predicting more answers like the data you've already seen is not scientifically interesting in many cases. Science requires understanding and extrapolation beyond the familiar. Also, the idea of using a system starting from a random configuration that goes to a local optimum is a little disconcerting. However, a confluence of events is quickly leading to a change of heart. The key has not been "let's throw a neural network on this since it's a universal approximator" like some early methods showcased (indeed, this lead to new but very inefficient ways of doing something that people had well-established tools for, so you can understand why the hype died down). Rather, the key has been to re-envision how to utilize a universal approximator into the context of tools and theories that were already being used in scientific computing. My favorite example is this paper which showcases how to solve high dimensional partial differential equations using backwards SDEs which are parameterized by neural networks. While there was some work before under the name latent differential equations, the 2018 NeurIPS best paper on neural ordinary differential equations really sparked a surge in thinking about directly learning differential equations, or pieces of differential equations. While the neural network itself may not be interpretable, if it learns a differential equation and differential equations have

an interpretation, then it would seem that we have a machine-learning generated scientific hypothesis.

So, this is where the trend has taken us. Neural networks are being used in the context of ordinary and partial differential equations, and these are being mixed with probabilistic programming approaches to quantify uncertainties and are being slammed into toolchains which require differentiable programming to generate the gradients, and...

As you can see, tooling is getting complicated. And tooling is what needs to be discussed.

THE TOOLS OF SCIENTIFIC MACHINE LEARNING

So, what exactly are the computational tools which are utilized in this burgeoning field of scientific machine learning? Since the field is still being developed it's hard to pinpoint exactly what people are focusing on, but there are a few trends. Two of the big pieces are a neural network framework, and libraries for numerical differential equations. The use of a neural network framework is obvious: the whole point is training neural networks in many contexts. By far the most common context is a differential equation, hence the tooling for discretizing and solving differential equations is necessary. This involves tools such as solvers for ordinary and stochastic differential equations, tools for discretizing PDEs with finite difference, finite volume, finite element, and pseudospectral discretizations. The existence of PDEs just begs for sparse linear algebra tooling. Construction of sparse Jacobians for the Newton methods within implicit schemes calls for color differentiation mixed with sparse or compressed factorization methods.

And all of this needs to play nicely with the automatic differentiation tools used for the differentiable programming and probabilistic programming frameworks of the training process. That is the key issue: just because some old FORTRAN code solves a PDE well isn't the end of the story anymore: if you cannot find a way to compute the derivatives of it then gradient descent won't work!

That's where our tooling list comes in. Partial differential equations are computationally difficult to solve. Neural networks are computationally difficult to train. Partial differential equations with neural networks is very difficult to do anything with. So to get anywhere, we need to have the most advanced methods for solving PDEs be compatible with our neural network frameworks. We need to make sure both methodologies are utilizing state-of-the-art techniques with efficient implementations, and that they can be correctly and efficiently composed. The types of tools which are necessary for large-scale scientific ML are:

1. Tooling for solving neural differential equations
2. Differentiable programming (automatic differentiation) tools
3. Probabilistic programming tools to learn uncertainty from data
4. Helper tools for sparsity detection and sparse differentiation
5. Structured linear algebra tools
6. Number types for mixed precision arithmetic
7. Methods for discretizing partial differential equations
8. Tools for generating and utilizing GPU kernels
9. Uncertainty quantification and Global sensitivity analysis
10. Surrogate modeling techniques

EXAMPLE CHALLENGE PROBLEM: NATURAL LANGUAGE PROCESSING + PDE CONSTRUCTION

To motivate the tooling that is needed, let's set the focus by picking an example challenge problem. A recent funding call asked for:

The Defense Advanced Research Projects Agency (DARPA) Defense Sciences Office (DSO) is requesting information on state-of-the-art approaches to generate multi-physics modeling and simulation codes directly from a description of the physical phenomena. Of interest are modeling and simulating increasingly complex systems involving multiple physics that require high fidelity simulations

but have limited test data (e.g., combustion, hypersonics, nuclear stockpile).

One way to approach this with scientific ML would be to do the following:

1. Build an Natural Language Processing (NLP) stack that interprets text into PDEs
2. Autodiscretize and solve the PDE
3. Write a loss function which checks the PDE solution against data
4. Add regularization based on the global sensitivity and uncertainty of the solution

If you've been following recent advancements in the field of automated model building, you'll see that such ideas are not that farfetched anymore. In fact, this is somewhat akin torecent differentiable rendering systems which are being tested for automatically learning an environment simultaneously to training a control circuit. Here, we are just training an NLP method to understand some scientific text simultaneously to its predictive ability through its PDE simulations.

However, just like in the case of differentiable rendering, we will need to make each of the "layers" differentiable: we will need to compute derivatives of the global sensitivity, uncertainty, the PDE's solution, and the PDE's generation. **This means that while there may be great libraries available for each of these tasks, to arrive at our overall goal all of the components will need to be compatible with our chosen automatic differentiation (/differentiable programming) framework, and that is the most difficult part.**

Let's dig in.

COMPARISON OF SCIENTIFIC MACHINE LEARNING PACKAGES AND TOOLS

THE AUTOMATIC DIFFERENTIATION (DIFFERENTIABLE PROGRAMMING) FRAMEWORKS

The dividing factor for scientific machine learning frameworks is not the language. Rather, it's the differentiable programming or automatic differentiation framework which is utilized. For those who aren't familiar, automatic differentiation is an umbrella term for a variety of techniques for efficien\ntly computing accurate derivatives of more or less general programs. It is employed by all major neural network frameworks since a single reverse-mode AD backpass (also known as a backpropagation) can compute a full gradient, whereas numerical differentiation would take many forward passes and symbolic differentiation is simply untenable due to expression explosion.

Thus, the key to making a scientific ML stack work is by making every component compatible with the AD-system. This is because, if there is just one part of your loss function that isn't AD-compatible, then the whole network won't train. When this happens, you the user either have to derive and define adjoints for each of the missing functions, or you need either:

- Beg the developers of the framework to add the package as a dependency and define the adjoints
- Define the adjoints yourself
- Rewrite the package utilizing the tools of the AD framework

For this reason, neural network frameworks likeTensorflow have painstakingly made sure that their frameworks cover all of the standard ML tasks. However, since we are not only doing machine learning but also are incorporating scientific computing, there are a lot of methods and packages that we will want to use which are simply not AD-compatible or have never been setup to be compatible with the AD framework. This is by far the most difficult problem for the practice of scientific machine learning.

The term **differentiable programming** has been adopted to describe AD systems which attempt to support an entire programming language. Each of the differentiable programming systems have had varying degrees of success with the language coverage. Early AD systems like ADIFOR (Fortran), TAF (Fortran), and ADOL-C (C++) achieved high coverage of their base languages, but did not attempt a large set of third-party packages. A lot of these frameworks are very efficient and can do source-to-source differentiation, which builds a new source program to compile which has very little overhead. This is very helpful for scientific machine learning since many nonlinear functions, like ODE definitions,

can only be described in a highly scalar fashion, meaning that it is efficient for this use case. (ADOL-C also has a tracing mode, which is much slower than its source-to-source)

The next generation of AD systems were domain-specific. Stan is a probabilistic programming language for Bayesian estimation which included a robust AD system. Tensorflow is a well-known AD system for building computational graphs for neural networks and machine learning. By restricting to a specific domain, these systems were able to achieve very good results in their respective areas, and overtime have grown their support to other domains.

The third generation of AD systems attempted to improve upon Tensorflow and bring machine learning AD to a language level. Examples of this are PyTorch and Flux.jl's Tracker.jl, which use tracing to generate a local computational graph to backpropagate through. These systems will work on any code for which adjoints have been defined for all of their operations. PyTorch simulates differentiable programming by having an internal module which has pretty good coverage of numpy, meaning that lots of numpy code can be ported over by changing the underlying "numpy module" that is used. However, this does mean that you cannot take arbitrary packages off of pip and expect PyTorch to work on it. This is especially an issue since a lot of Python packages are written with C++ extensions or using Cython, meaning that PyTorch will not be directly compatible with them without a lot of work. On the other hand, Flux.jl / Tracker.jl ties into Julia's multiple dispatch system which allows it to directly work on any pure-Julia package with sufficiently generic code. The reason is because there is simple to explain through an example. In Julia, there is only one `*` function, which stands for matrix multiplication, that everyone extends. `*` between different types calls a different methods (different implementations), but is still the same function. For example, Array*Array is standard dense multiplication defined in Julia's Base, while Elemental.Matrix*Elemental.Matrix would use the the MPI-compatible Elemental.jl distributed linear algebra library. Thus one definition of the adjoint for `*` would then apply to all libraries which implement new types and new matrix multiplications, as long as they conform to the system. Since the differentiation rules for the Julia AD systems are defined in common libraries, like DiffRules.jl (with the next generation being ChainRules.jl), this means that all of the Julia AD packages, like ForwardDiff.jl and ReverseDiff.jl, all have this same property. Thus while Python packages might choose which AD system to support, packages in Julia choose to support AD and then allow the various AD systems to compete.

However, tracing-based AD systems have a very high overhead since their computational graph changes every time the values change (meaning they have to compile a new backpass each time, or worse, are interpreted), and they have to generate such a computational graph (meaning tons of allocations!). This is not a problem for traditional machine learning since the cost of a single matrix multiplication would dwarf the overhead. However, if highly scalar code shows up in the pipeline, such as the definition of a nonlinear ODE for a chemical reaction network for a pharmacometric system, this overhead starts to dominate the run time.

To handle this problem, differentiable programming frameworks have gone back to the history books and the newest versions utilize source-to-source transformations instead of tracing. In this category there is Zygote.jl and Swift for Tensorflow. While Zygote is fairly new, the property of Julia AD support mentioned before means that many packages already have a good degree of Zygote support simply by utilizing the multiple dispatch mechanism. Thus Zygote has been already used to showcase scientific machine learning applications like quantum machine learning and neural stochastic differential equations. While theoretically Swift for Tensorflow could do the same, I am not aware of anything like a Swift finite element method PDE discretization library, and thus while it's a turning into a great AD system, it doesn't currently have enough scientific ML support to warrant further consideration at the moment. It’s possible that Swift for TensorFlow will be successful enough as an AD system that people will start to build scientific libraries in Swift, but we won’t know for at least another five years.

Those are the differentiable programming frameworks that we will explore in more detail. There are other good AD systems, but they tend to lack the support for both scientific computing and machine

learning required to complete scientific ML tasks. For example, there are some good MATLAB AD tools that you can find (and some support automatic sparsity detection). However, these tools do not have the kind of neural network and GPU support necessary for example to use a convolutional neural network alongside a PDE. On the other hand, there are some AD systems like JAX which are great for traditional machine learning but I do not think anyone has built any PDE solver libraries on (again, this is not a post about traditional ML). I'll also throw a shoutout to packages like CasADi and DAEPACK which are good AD systems with differential equations, but do not integrate with larger package ecosystems. Tapenade is another great AD system which comes to mind (Fortran).

NEURAL NETWORK SUPPORT

The first thing to discuss is neural network support. It's not surprising that the AD systems which were built for traditional machine learning, like Tensorflow, Flux.jl (Julia), and PyTorch, have very complete support in this area. Convolutional layers, LSTMs, batch normalization, etc. are all readily available with all of the goodies like GPU support. If you really wanted to pick a winner here, PyTorch probably has the most complete support right now, but the other two are not far behind and developments occur daily.

The non-ML AD frameworks have a surprising lack of neural network support. Before digging into this, I would've guessed that someone would've made a neural network framework for ADIFOR or ADOL-C. However, my Google skills could not find any. That said, it's not very difficult to roll your own dense layers (W*x + b). The limitation will come here when you want to start using convolutional neural networks or transformers, where you'd need to start rolling some significant CUDA code to get there while the other frameworks will give you it in one call. I would like to see the upstart frameworks like neural-fortran and OpenNN get some integration with these big AD systems. Another surprising tale was that Stan hasn't received much neural network support yet. I say yet because there are some early results trying to integrate Stan with PyTorch. It would be really beneficial for Stan for Scientific ML if that effort was completed and pulled into the standard Stan build.

Summary: TensorFlow, Flux.jl, and PyTorch lead the pack quite a bit here.

NEURAL DIFFERENTIAL EQUATIONS

Scientific computing has a lot of differential equations. Ordinary differential equations are a major topic of their own, with many scientific laws described in their language. Partial differential equations with a time component are solved by discretizing down to a set of ODEs to be solved. If there are constraints on the ODE, such as conservation of energy, then one has a differential-algebraic equation, and PDEs with some boundary conditions discretize down to DAEs as well. In other cases when there is randomness that is modeled, like in biological or financial models, stochastic differential equations (SDEs) are used. If the reactions are not instantaneous, then delay differential equations are used.

If you do scientific computing, you know this. These models are pervasive, and thus support for all of these kinds of models is essential for doing scientific ML. However, differential equations only recently got on the minds of those in ML. The method of neural differential equations is very interesting though, because there are two very different use cases of it. The best paper at NeurIPS 2018 showcased how neural ODEs could be used for standard ML classification problems, while here we want to do things like automated discovery of dynamical equations. This may seem like a detail, but it has a profound effect on the tools and methods which are necessary. From our studies (and folks in David Duvenaud's lab) with neural differential equations for ML, it seems that the learned functions tend to be non-stiff and "fairly reversible" (this is mentioned at the end of this video). Thus the tools built for neural ODEs in traditional ML, like torchdiffeq, work for this domain. However, it has beenpointed out multiple times that the adjoint function they use is generally unstable. Additionally, torchdiffeq only has non-stiff ODE solvers, which makes them unsuitable for learning the dynamics of many equationsdue to a lack of stability. Thus, while torchdiffeq is a fantastic package for its domain, its domain is not scientific ML.

A comprehensive overview of software for differential equations can be found here. If you're using

Julia, there already exists DiffEqFlux.jl which supports neural ODEs, neural SDEs, neural DDEs, neural PDEs, etc. the whole gambit of neural differential equations using the highly efficient DifferentialEquations.jl. It can also be used from R and Python, and its sensitivity analysis could be used to plug into systems in these other languages. If you're using C++ or Fortran and only for ODEs/DAEs, then Sundials is a good choice. While it's not directly integrated with ADOL-C, TAF, or ADIFOR, this library comes with adjoint sensitivity analysis functions that would make it easy to define the right gradients to hook into the system. In fact, Stan does exactly this to give ODE support (but not DAEs). A lesser known choice is FATODE which also defines adjoints, so it could also plug into AD systems but only supports ODEs.

Summary: Julia has expansive neural differential equation support with DiffEqFlux.jl, Stan has some ODE support, while the other systems leave you to your own devices. However, DifferentialEquations.jl, Sundials, and FATODE could be used to give existing AD systems support for ODEs and DAEs.

PROBABILISTIC PROGRAMMING

Probabilistic programming is the fancy differentiable programming word for Bayesian estimation of parameters. Instead of optimizing and getting point estimates for the best parameters, you get a posterior distribution. This lets you generate a fit with some uncertainty, which is necessary for the uncertainty quantification in the discussed challenge problem.

Probabilistic programming comes in two major flavors: Monte Carlo based methods and variational inference methods. Monte Carlo methods use a stochastic algorithm to approximate a distribution asymptotically, while variational inference methods write out the prior in some basis and push through the basis elements to get the posterior written in said basis. MCMC is generally used for problems with less parameters and variational inference for those with more, but there are times when it's worthwhile to switch it up. Stan is the undeniable leader in probabilistic programming with MCMC, with it being the first big program to make use of Hamiltonian Monte Carlo (HMC) and the No U-Turn Sampling (NUTS) method for very robust automatic tuning of hyperparameters. This make Stan seem to "just work" in a way MCMC usually doesn't. However, other ecosystems are now catching up, with Turing.jl in Julia and PyMC3 being two very good systems. While PyMC3 is built on Theano and thus not compatible with these AD systems, the experimental PyMC4 is a very promising system for TensorFlow. The new Gen system in Julia takes an interesting approach by making it more flexible and less automatic which can be helpful in the most difficult cases. These systems all seem to be in these higher level languages, so I could not find very many in Fortran or C++ (just CPProb).

For variational inference, Pyro for TensorFlow seems to be at the head of the pack for Bayesian neural networks, with Edward being another good choice. Gen in Julia is a recent addition with variational inference as well.

Summary: TensorFlow, PyTorch, and Julia have some good options for probabilistic programming. It will be interesting to see how this space keeps evolving.

AUTOMATED SPARSITY DETECTION AND SPARSE DIFFERENTIATION

Partial differential equations essentially always give an ODE discretization where the Jacobian is sparse, or it gives a nonlinear rootfinding problem where the Jacobian is sparse. Either way, using this sparsity is something that is required in order to do anything efficient in this domain. Thus, it's no surprise that the AD systems which were built to handle scientific computing, such as TAF and ADOL-C, come with automated sparsity detection and sparse differentiation as standard features. Both of these make use of ColPACK for the matrix coloring portion. A recent addition to the Julia ecosystem is SparsityDetection.jl and SparseDiffTools.jl which give similar automated sparsity detection, matrix colorization, and integration with AD. My Google-foo could not find such support in Tensorflow, PyTorch, Stan, JAX, TAPENADE, or ADIFOR (this one was surprising). Since traditional neural networks are never sparse, and newer sparse ADs like graphical neural networks have a sparsity

pattern that you know, it makes sense that machine learning libraries never cared about this. But it's something we do miss when doing scientific ML.

Summary: TAF, ADOL-C, and the Julia AD tools are the ones which showcase sparsity support.

GPU SUPPORT

While the scientific computing ecosystems like Fortran, C++, and Julia picked up a few wins with differential equations and sparsity, the machine learning ecosystems like TensorFlow, PyTorch, and Julia are where a lot of nice tooling for GPUs exist. High level functions let users define efficient kernels directly in Julia and Python, and these will then be automatically compatible with AD. In Fortran or C++, you're back to the same old story of writing a CUDA kernel yourself, and then having to define adjoint functions for the AD to work. This is still better than Stan which seems to just have OpenCL support. While I get the appeal of not being vendor-locked to NVIDIA, sources routinely show CUDA libraries like cudnn are much more efficient than their OpenCL counterparts, making OpenCL not the best choice for scientific ML which is already heavy on compute.

Summary: TensorFlow, PyTorch, and Julia have good tooling that will work with AD. C++ and Fortran you can of course use directly with CUDA, but it's BYOA (bring your own adjoint).

DISTRIBUTED DENSE, STRUCTURED, AND SPARSE LINEAR ALGEBRA

Once again, we have a similar language divide. In Fortran and C++ we have tools likeScaLAPACK, PARASOL, PETSc, Trilinos, and Elemental. While these are not necessarily baked into the AD systems, defining the adjoints for matrix multiplication and linear solving are pretty straightforward, so that's not really a drawback here. Fortran, with its scientific computing focus, has the library SPARSEKIT which has some nice structured matrix support as well.

On the other hand, TensorFlow and PyTorch have very good distributed linear algebra support, except they leave off factorizations here and there. torch.distributed and TensorFlow sparse both seem to leave off LU and QR factorizations from the list of things to support. Once again, this is fine for traditional ML which would almost never use these, but when writing a PDE solver these will come up quite often. Julia stands in an interesting position here since, due to the way its AD overrides work, the Elemental.jl library automatically works for distributed and sparse linear algebra. This is currently a much better solution than the pure-Julia DistributedArrays.jl, though this space may be get a lot of development in the near future.

The piece that is quite unique to Julia here is BandedMatices.jl and BlockBandedMatrices.jl, which are specialized structured matrix libraries for these matrix types. It just so happens that many discretizations of systems of PDEs give you block banded matrices, and these libraries give a pretty significant speedup over using just a sparse matrix. Thus, this is a really nice library to have around (and it uses `*` and `\`... so it's AD compatible).

Summary: Julia has a good number of tools for this, almost by accident. I'm not sure this interaction between AD and Elemental.jl / BlockBandedMatrices.jl was intended, but it makes for a very good PDE linear algebra ecosystem for Julia. TensorFlow has the right building blocks but needs sparse factorizations. PyTorch just needs more. C++ and Fortran give you a ton of tools but leave you to write your own adjoints, which isn't that difficult in this case.

SURROGATES, GLOBAL SENSITIVITY ANALYSIS, AND UNCERTAINTY QUANTIFICATION

Libraries for surrogates, global sensitivity analysis, and uncertainty quantification are commonly utilized for analyzing whether a PDE fit is good. It's only natural that these will be used to analyze whether a neural PDE's fit is good. Our challenge problem made note that I would like to use the global sensitivity or uncertainty of the result as a loss value that quantifies how trustworthy the result is.

However, there seems to be a lack of AD-compatible complete GSA and UQ systems. Julia has a

newly created Surrogates.jl for surrogate modeling and optimization, which rivals the pySOT surrogate optimization package, but that's the only complete package in this area that I could find which is AD compatible. That isn't to say there aren't other good packages, it's just that they don't integrate with any of the AD systems to be readily used in the loss functions. SMT for Python and MUQ for C++ are some great examples, with Dakota and PSUDAE are interfaced through their binaries. Even R has a great sensitivity package, and SimLab in MATLAB is pretty good as well. But you cannot expect to just run AD through them, and some of the functionality may not be easy to define adjoints for. Here, the Julia libraries of DiffEqSensitivity and DiffEqUncertainty are compatible with AD, but do not have all of the functionality you see from the other libraries. So, there's no silver bullet and this field could use some work.

Summary: The only AD-compatible packages in this field are in Julia, but they still need a little more work to be as strong as some of the C++ offerings like Dakota or MUQ. This means there's no silver bullet for GSA or UQ for scientific ML right now.

DISTRIBUTED PARALLELISM

At this point, all of the ecosystems have pretty good distributed parallelism support. TensorFlow in particular seems built for deploying to clusters with GPUs and TPUs. torch.distributed has a very good list of what's supported with its various backends. Julia can make use of MPI.jl, of which some of the AD systems are compatible (as long as the AD components are serializable). C++ and Fortran of course have MPI, and I am not aware of whether the source-to-source AD systems can handle MPI so please let me know! Stan also has a way of making use of MPI.

Summary: If you know MPI (common among people doing scientific computing), then any of these are fine.

AUTOMATED PDE DISCRETIZATIONS

At some point, if we are automatically generating PDEs, then we will need tooling for automatically discretizing PDEs. If you aren't familiar with the topic, the ways to solve PDEs is to essentially transform them into systems of linear equations, nonlinear equations, or o\nrdinary differential equations (or DAEs, or SDEs, etc.). There are a lot of great libraries for discretizing PDEs. An abundance of such libraries exist in C++, with deal.ii, SAMRAI, and hypre being well-known examples (and each of the C++ libraries also generally document how to use them from Fortran). There is a smaller but strong contingent in Python as well, with FEniCS being one of the top FEM packages, with Firedrake also in the running. Additionally, Daedalus is a strong package for pseudospectral discretizations. But again, none of these hook into ADOL-C, TensorFlow, or PyTorch, so they are great libraries if not used in an auto-PDE construction toolchain, but do have this fundamental limitation. There are some startup libraries brewing in Julia as well. ApproxFun.jl is an enhanced version of Chebfun for Julia which is really good for generating pseudospectral PDE discretizations (with adaptive mesh sizes), though its documentation will need to be more complete for most people to recognize all that it can do. DiffEqOperators.jl is a fledgling finite difference method library which ties into the neural network software to make its convolutions utilize cudnn play nicely with Flux.jl. And there are efforts for FEM in pure-Julia, with JuliaFEM and JuAFEM.jl picking up a lot of steam, with at least the latter having the ability to have neural networks hook into its local assembly routines as it's just user-written Julia code. It'll be interesting to see how these tools evolve in terms of their neural network friendliness.

Summary: There are some good PDE discretization libraries for Fortran, C++, and Python users, but neural networks / AD don't generally play well with them. Julia has some libraries growing which are AD-friendly, but not feature complete yet.

CONCLUSION

There are many great differentiable programming frameworks, but expansive feature sets for

traditional machine learning do not necessarily mean expansive feature sets for scientific machine learning. Here we took a deep dive into scientific machine learning and the tools which are available for solving its problems. By looking at the available packages, we see that Julia has the most mature scientific machine learning package ecosystem, though it (and all others) have a ton of work to do. While Fortran and C++ still have a lot of very great scientific computing tooling, the widely used AD systems with automated sparsity support, like ADOL-C and TAF, do not come with neural network libraries, ODE solver libraries, UQ libraries, etc. all baked in. In addition, the big Python frameworks have some great neural network support, but are missing many of the major features used in the study of (partial) differential equations, making them not as ideal for this specific purpose. At the same time, there do not seem to be competitive AD ecosystems in other popular scientific computing languages for MATLAB and R (again, there are great AD packages, just missing the things like convolutional neural networks and partial differential equation mixtures!). Thus, the easiest way forward would likely be to further develop the analysis features (global sensitivity analysis and uncertainty quantification), where one can rely on full-language AD systems like Zygote to then incorporate it into neural partial differential equation pipelines. The next most viable path would be to wrap a bunch of adjoints into ADOL-C for neural networks and analysis, which given the strong scientific computing contingent in C++, and this will likely happen in the near feature.

While this is somewhat of a fringe topic right now, I hope to see a lot of development in scientific ML and scientific AI in the near feature, and plan to track the developments with this blog. What I think is made clear here, however, is that simply focusing on traditional ML workflows will not be sufficient for coverage of this domain, and thus to see progress in scientific ML tools we will need to see features in AD systems which may not have been covered by their original agendas.

CITATION

To cite this post, please use the following from The Winnower: