Interlingual Automatic Differentiation: Software 2.0 between PyTorch and Julia

Daniel O'Malley[1], **Javier E. Santos**[1,2], **Nicholas Lubbers**[3]

[1] EES-16, Los Alamos National Laboratory, Los Alamos, NM 87545 USA
[2] Center for Nonlinear Studies, Los Alamos National Laboratory, Los Alamos, NM 87545 USA
[3] CCS-3, Los Alamos National Laboratory, Los Alamos, NM 87545 USA

Abstract

Julia is a state-of-the-art tool for scientific computing with good support for automatic differentiation. PyTorch is a leading framework for machine learning. We describe how to perform automatic differentiation across the language boundary and connect these two ecosystems. By using the automatic differentiation ecosystems in each language, the time complexity is improved compared to using a standard foreign function interface, which would require naive, black-box techniques like finite difference derivatives. In bridging the language gap, we allow PyTorch users to exploit the excellent and growing scientific computing ecosystem that is written in Julia. Simultaneously, we make the work of Julia developers available to the PyTorch community, which is larger than the Julia community. We illustrate the strengths of this approach by considering an application to flow in porous media. In this illustration, we utilize the rich PyTorch ecosystem, including PyTorch Lightning, and combine it with DPFEHM, a differentiable subsurface physics simulator written in Julia.

Introduction

Machine learning has recently emerged as an approach to computing that is transforming many fields, including science. Machine learning frameworks enable the development of software, where the derivatives of the software can be efficiently computed. Software for which derivatives can be computed is often called Software 2.0. The two leading frameworks are PyTorch (Paszke et al. 2019) and TensorFlow (Abadi et al. 2016), both of which are predominantly used in Python (Van Rossum and Drake 2009). The machine learning process typically involves a model with a fixed structure and trainable parameters that is implemented in one of these frameworks. The derivatives are then used to train the model using gradient descent or a variation thereof.

Scientific computing (Strang 2007) has been a stalwart tool used in the scientific community for decades. During this time, scientific simulators have been implemented in a wide variety of programming languages often including C (Ritchie et al. 1978), C++ (Stroustrup 2013), FORTRAN (Backus 1978), and more recently in Julia (Bezanson et al. 2017). A commonly-used feature in programming languages for scientific computing is the ability to call functions written in another language through a foreign function interface. This enables, e.g., a simulator written in C to use a nonlinear solver written in FORTRAN. More broadly, these foreign function interfaces enable the scientific computing ecosystem to be written in a variety of languages rather than just one.

Combining scientific simulations and machine learning models is an increasingly common paradigm for computing (Bedrunka et al. 2021; Wu et al. 2022; Pachalieva et al. 2022). However, uniting the machine learning and scientific computing paradigms does not currently come with the freedom in programming languages that the scientific computing community has historically enjoyed. This is because one cannot simply use a foreign function interface to call a C function from PyTorch (for example) and expect the automatic differentiation to work.

Machine learning frameworks generally use reverse-mode automatic differentiation to compute the gradient of a loss function, which has many parameters. Doing this involves both a forward pass to compute the value of the loss function and backward pass to compute the gradient of the loss function. Traditionally foreign function interfaces enable the forward pass, but not the backward pass. To bring these forward function interfaces into the Software 2.0 era, the backward pass must also be enabled.

In this paper, we show how to build a Software 2.0 foreign function interface. In particular, we enable Julia functions to be integrated into a PyTorch workflow, with the automatic differentiation working seamlessly. The remainder of this paper is organized as follows. First, we describe the interface between PyTorch and Julia. Next, we present two illustrative examples – one simple and one complex. Finally, we describe our conclusions.

Differentiable language interoperability

Basic, non-differentiable interoperability between Python and Julia can be readily accomplished with the PyJulia package, which allows a Python program to import Julia modules and to execute arbitrary code strings in an active Julia interpreter. Fortunately, this interface is already robust to the passing of arrays, which are native to Julia and represented in Python using numpy. Interoperability between PyTorch and numpy is provided by PyTorch.

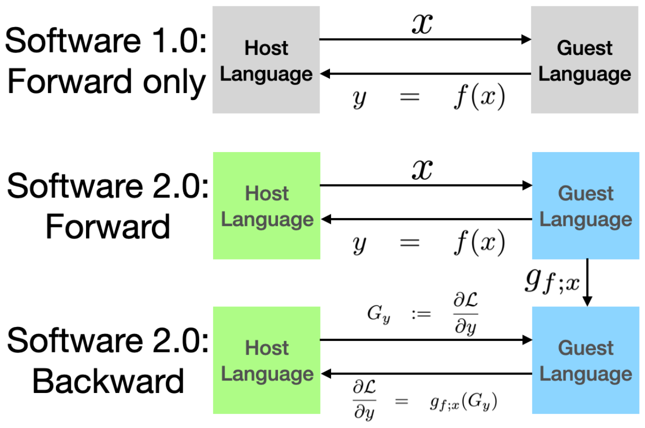

Figure 1: The mechanisms of foreign function interfaces for software 1.0 and software 2.0 are shown. Using our foreign function interface, we can call a simulator written in Julia as part of a PyTorch model, and have the automatic differentiation work across the language boundary.

To accomplish differentiable interoperability, we must look at how the automatic differentiation primitives are encoded in each framework. We have a function f, inputs x, and outputs y, where x and y may be arbitrary arrays. The chain rule gives that the gradients of these quantities with respect to some forward scalar $\mathcal{L}$ computed from y are given by:

$$y = f(x) \tag{1}$$

$$G_y := \frac{\partial \mathcal{L}}{\partial y} \tag{2}$$

$$G_x := \frac{\partial \mathcal{L}}{\partial x} \tag{3}$$

$$G_x = g_{f;x}(G_y) = G_y \cdot \frac{\partial f}{\partial x} \tag{4}$$

In Zygote (Innes et al. 2019), which is currently the main automatic differentiation tool in Julia, the vector-Jacobian product which encapsulates the backwards differentiation of a single function can be accessed using the 'pullback' function; this returns both y and the closure $g_{f;x}$ which gives an efficient evaluation of the gradient at point x; the pullback of f at x. In PyTorch, a differentiable operation is encoded using a forward method evaluating f, a backward method giving g_f, and a context object which is allowed to encapsulate information from the forward pass in order to efficiently evaluate g_f at x.

Dovetailing these two representations is thus fairly simple – see Figure 1. In the forward pass, we generate both the output y and the pullback $g_{f;x}$. The pullback is then stored on the PyTorch context object, and passed the gradient g_y. The backward pass, nearly trivial, then only needs to extract the pullback object from the context, and evaluate it on the gradient and return the result as a PyTorch tensor.

To make the interface generic and user friendly, we have wrapped this procedure for arbitrary Julia set-up code and functions to extract. However, this procedure has some complexity because of the structure of PyTorch differentiable functions: They are built using a class object which has only static methods, and instances of that class are not built by the user. Because instances are not built, manufacturing multiple connections between Julia and PyTorch cannot be accomplished by using instances which store the needed information to call a given Julia function. While initially, we hoped to solve this problem using class arguments (using either metaclasses or the `__init_subclass__` hook in Python), it is not straightforward to do so because PyTorch already uses metaclass mechanisms and is not prepared to accept class arguments. So instead, we employ a class factory pattern; `julia2pytorch` constructs an inner class, and the references to the Julia code which the class runs are stored as a closure over the function scope containing the inner class's definition. This cleanly solves the problem of encapsulating the function f within the class definition, without violating PyTorch's constraint that differentiable functions are implemented with static methods.

Examples

Simple function

To illustrate our approach, we begin with a simple example. Listing 1 show the code for this example. It involves a function, f, that can take either 1 or two arguments. In Julia's multiple dispatch parlance, f is a function with two methods. For the 1-argument method, the function f multiplies the argument, x, by a constant matrix, A. In this case, the Jacobian is simply the matrix A. The example subsequently verifies (line 22 in the listing) that PyTorch correctly computes the gradient of a function involving f, which is implemented in Julia.

In the second part of the listing, the code verifies that a gradient involving the 2-argument method of f is also correct (line 31 in the listing). This indicates that this approach is compatible with Julia's multiple dispatch functionality, which is a key feature in Julia.

Our tests scaling this example up to bigger matrices indicate that the overhead of passing data between the two languages is negligible. This indicates that using an automatically differentiated foreign function is almost as good as using a native function. Of course, the real value of this approach is better illustrated by the next example where reimplementing the foreign function in PyTorch would require a non-trivial amount of developer time. The low overhead indicates there is no real benefit to reimplementing – the foreign function is effectively as good.

Scientific Simulator Workflow: Flow through Fractures

Using our language-bridging approach we implemented an example where Pytorch-lightning (Falcon and The PyTorch Lightning team 2019), a popular deep learning framework in python performs forward and backward passes through a single-phase flow in porous media model implemented in DPFEHM (O'Malley et al. 2022), a subsurface physics simulator written in Julia. DPFEHM is able to solve the forward

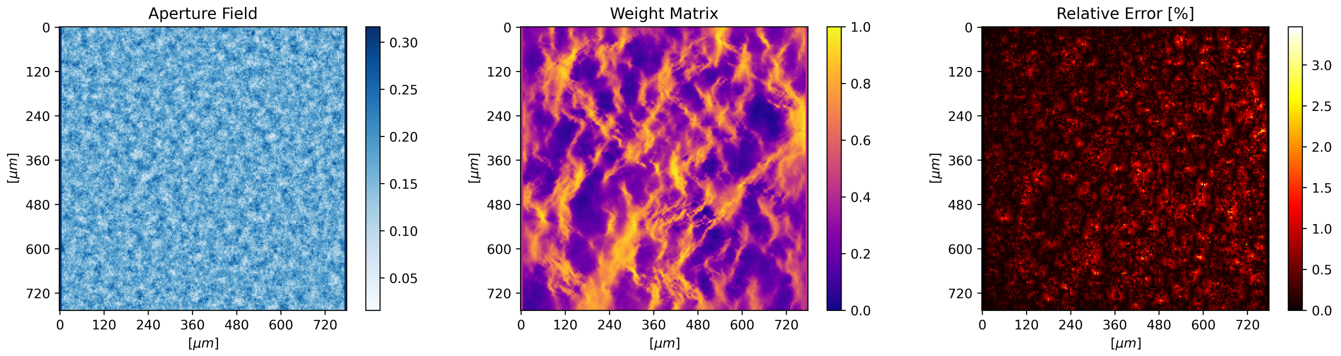

Figure 2: **Left)** Aperture field of the fracture. **Middle)** Resulting weight matrix W after training. **Right)** Pixel-wise relative error vs fine-scale simulations (the average error is less than 1%).

problem in milliseconds, nevertheless, it relies on the fact that the effective properties that are passed as input faithfully represent the domain characteristics. In the case of fluid flow through fractures, there is no functional relationship that accurately upscales the medium's characteristics (fracture aperture Ap) to permeability k. The local cubic law ($k = Ap^2/12$) is a common equation used. Nevertheless, since it stems from the assumption of a fracture being a pair of parallel plates, it fails in the context of real fractures by always overestimating k. The end goal of our model is to learn this relationship. This workflow is shown in Figure 3.

To approximate this relationship, we initialize a matrix of weights W that learns the right parameters to modify the cubic law solution. We train these weights to match a property of interest (in this case, the pressure field p) from fine-scale simulations (obtained via the lattice-Boltzmann method) by minimizing the mean square error (MSE) of the fields:

$$p = \mathrm{LBM}(\mathrm{domain}) \tag{5}$$
$$\hat{k} = Ap^2/12 \tag{6}$$
$$\hat{p} = \mathrm{DPFEHM}(\hat{k} \cdot W) \tag{7}$$
$$\mathcal{L} = \mathrm{MSE}(p, \hat{p}). \tag{8}$$

Our results are shown in Figure 2. Computing the pressure field p via the lattice-Boltzmann methods takes around 5 hours in a workstation, whereas DPFEHM is able to solve the forward problem in milliseconds. Hence, a model that is able to combine the LBM accuracy with DPFEHM speed is of great interest. This approach differs from purely data-driven workflows, since in this paradigm, the solution is provided by a full-physics simulation (DPFEHM), in comparison to a machine-learning model, which is untrustworthy and, e.g., susceptible to fail without warning.

```
setup_code = """
                import Zygote
                A = [1 2; 3 4]
                f(x, y) = A * x + y
                f(x) = A * x
             """

func_name = "f" # the name of the julia
                # function we want to be
                # able to differentiate
                                # in PyTorch

f = julia2pytorch(func_name, setup_code)
x = torch.zeros(2).requires_grad_(True)
y = torch.zeros(2).requires_grad_(True)

f_eval = f(x)
g = grad(f_eval.sum(), [x])
print("Is d/dx[f(x)] correct?")
g = g[0].detach().numpy()
t = (g == [4, 6]).all()
print(t) # prints "True"

f_eval = f(x, y)
g = grad(f_eval.sum(), [x, y])
print("Is d/d[x, y][f(x, y)] correct?")
g0 = g[0].detach().numpy()
g1 = g[1].detach().numpy()
t0 = (g0 == [4, 6]).all()
t1 = (g1 == [1, 1]).all()
print(t0 and t1) # prints "True"
```

Listing 1: A simple example illustrating interlingual automatic differentiation between PyTorch and Julia.

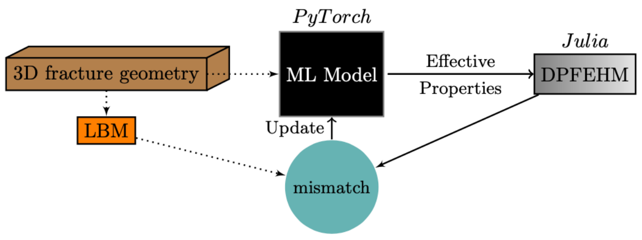

Figure 3: The proposed workflow for scale-bridging using differentiable programming.

Conclusion

Scientific computing has benefited greatly from the ability for codes written in different languages to interact through foreign function interfaces. As differentiability becomes important with the rise of machine learning, we must revisit these interfaces so that differentiation can occur across the language boundary. We illustrated that this is possible by connecting a PyTorch machine learning workflow to a differentiable subsurface flow simulator written in Julia.

This demonstration utilized two languages/frameworks (Python/PyTorch and Julia/Zygote) that have prioritized making software differentiable. This made it possible to utilize the general ‘julia2pytorch‘ factory with minimal effort from the user – only specifying some setup code and providing the name of the function. Scientific simulators have not traditionally been written to be differentiable, so this automation is not possible in many cases. However, these old-school simulators may support adjoint methods. Support for adjoint methods would open the door to bringing these simulators into a differentiable workflow (Karra, Ahmmed, and Mudunuru 2021), though this is more cumbersome.

References

Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. 2016. TensorFlow: a system for Large-Scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 265–283.

Backus, J. 1978. The history of Fortran I, II, and III. *ACM Sigplan Notices*, 13(8): 165–180.

Bedrunka, M. C.; Wilde, D.; Kliemank, M.; Reith, D.; Foysi, H.; and Krämer, A. 2021. Lettuce: Pytorch-based lattice boltzmann framework. In *International Conference on High Performance Computing*, 40–55. Springer.

Bezanson, J.; Edelman, A.; Karpinski, S.; and Shah, V. B. 2017. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1): 65–98.

Falcon, W.; and The PyTorch Lightning team. 2019. PyTorch Lightning.

Innes, M.; Edelman, A.; Fischer, K.; Rackauckas, C.; Saba, E.; Shah, V. B.; and Tebbutt, W. 2019. A differentiable programming system to bridge machine learning and scientific computing. *arXiv preprint arXiv:1907.07587*.

Karra, S.; Ahmmed, B.; and Mudunuru, M. K. 2021. AdjointNet: Constraining machine learning models with physics-based codes. *arXiv preprint arXiv:2109.03956*.

O’Malley, D.; Greer, S. Y.; Pachalieva, A.; Hao, W.; Harp, D.; and Vesselinov, V. V. 2022. DPFEHM: a differentiable subsurface physics simulator. https://github.com/OrchardLANL/DPFEHM.jl/blob/master/paper.md. [Online; accessed 12-September-2022].

Pachalieva, A.; O’Malley, D.; Harp, D. R.; and Viswanathan, H. 2022. Physics-informed machine learning with differentiable programming for heterogeneous underground reservoir pressure management. *arXiv preprint arXiv:2206.10718*.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

Ritchie, D. M.; Johnson, S. C.; Lesk, M.; Kernighan, B.; et al. 1978. The C programming language. *Bell Sys. Tech. J*, 57(6): 1991–2019.

Strang, G. 2007. *Computational science and engineering*, volume 791. Wellesley-Cambridge Press Wellesley.

Stroustrup, B. 2013. *The C++ programming language*. Pearson Education.

Van Rossum, G.; and Drake, F. L. 2009. *Python 3 reference manual*. CreateSpace.

Wu, H.; O’Malley, D.; Golden, J. K.; and Vesselinov, V. V. 2022. Inverse analysis with variational autoencoders: a comparison of shallow and deep networks. *Journal of Machine Learning for Modeling and Computing*, 3(2).