The Journal of Open Source Software

Φ_{ML}: A Science-oriented Math and Neural Network Library for Jax, PyTorch, TensorFlow & NumPy

Philipp Holl [1] **and Nils Thuerey**[1]

1 Technical University of Munich

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Marcel Stimberg

Reviewers:

- @wandeln
- @chaoming0625
- @gauravbokil8

Submitted: 11 August 2023
Published: unpublished

In partnership with

AMERICAN ASTRONOMICAL SOCIETY

This article and software are linked with research article DOI 10.3847/xxxxx <- update this with the DOI from AAS once you know it., published in the Astrophysical Journal <- The name of the AAS journal..

Summary

Φ_{ML} is a math and neural network library designed for science applications. It enables users to quickly evaluate many network architectures on their data sets, perform (sparse) linear and non-linear optimization, and write differentiable simulations that scale to *n* dimensions. Φ_{ML} is compatible with Jax, PyTorch, TensorFlow and NumPy, and user code can be executed on all of these backends. The project is hosted at https://github.com/tum-pbs/PhiML under the MIT license.

Statement of need

Machine learning (ML) has become an essential tool for scientific research. In recent years, ML has been used to make significant advances in a wide range of scientific fields, including chemistry (Butler et al., 2018), materials science (Wei et al., 2019), weather and climate prediction (Bochenek & Ustrnul, 2022; Rolnick et al., 2022), computational fluid dynamics (Brunton et al., 2020), drug discovery (Jumper et al., 2021; Vamathevan et al., 2019), astrophysics (De La Calleja & Fuentes, 2004; Ntampaka et al., 2015; Petroff et al., 2020), geology (Rodriguez-Galiano et al., 2015), and many more. The use of ML for scientific applications is still in its early stages, but it has the potential to revolutionize the way that science is done. ML can help researchers to make new discoveries and insights that were previously impossible.

ML in science sets itself apart from other ML applications by a number of features.

- The dynamics of the observed system is often (partially) known and can be explicitly simulated. Making use of this knowledge has been shown to improve results when training ML models (Raissi et al., 2019; Um et al., 2020).
- Data typically represent objects or signals that exist in space and time. Data dimensions are interpretable, e.g. vector components, time series, *n*-dimensional lattices.
- Information transfer is usually local, resulting in sparsity in the dependency matrix between objects (particles, elements or cells).
- A high numerical accuracy is desirable, often requiring 64-bit floating point calculations.

However, current machine learning frameworks have limited support for these features, or they are cumbersome to use. Φ_{ML} is a scientific computing library based on Python 3 (Van Rossum & Drake, 2009) that aims to address these issues and simplify scientific code in the process. It consists of a high-level NumPy-like API geared towards writing easy-to-read and scalable simulation code, as well as a neural network API designed to allow users to quickly iterate over many network architectures and hyperparameter settings. Similar to eagerpy (Rauber et al., 2020), Φ_{ML} integrates with Jax (Bradbury et al., 2018), PyTorch (Paszke et al., 2019), TensorFlow (Abadi et al., 2016) and NumPy (Harris et al., 2020), providing a custom Tensor class. However, unlike eagerpy, Φ_{ML}'s `Tensor` adds additional functionality to make user code

more concise and easier to read.

Φ_{ML} has been in development since 2019 as part of the Φ_{Flow} (Holl et al., 2020) project where it originated as a unified API for TensorFlow and NumPy, used to run differentiable fluid simulations. With Φ_{Flow} version 2.0 and consecutive releases, Φ_{ML} underwent a drastic overhaul. A major issue with the previous API, and in fact all popular ML APIs, is the need for reshaping, which can quickly get out of hand for physical simulations. The work towards automatic reshaping sparked most of the changes that have been made to the library since.

We will first explain the design principles underlying Φ_{ML}'s development, before detailing the major design decisions and resulting architecture. For a list of supported features, see the GitHub homepage.

Design Principles

Here, we lay out our goals in developing Φ_{ML}, which serve as the foundation for the design.

Reusability

Simulation code based on Φ_{ML} should be able to run in many settings without modification. The dynamics of a system, e.g. governed by a partial differential equations, are often formulated in a dimension-agnostic manner. Simulation code implementing these dynamics should also exhibit that property. Most simulations use some form of discretization, such as particles or grids. Simulation code written for one such discretization should be easy to port to another appropriate one.

Compatibility

There are many toolkits and libraries extending ML frameworks with specialized functionality. These are generally only available for a certain framework, be it TensorFlow, PyTorch or Jax. Φ_{ML} users should be free to choose whatever framework they desire without modifying their simulation code. Additionally, simulations should be able to run on GPUs and CPUs and be vectorizable without modification. Φ_{ML} should support Linux, Windows and Mac.

Usability

Φ_{ML} should be easy to learn and use. To achieve this, the API should be intuitive with expressively named functions matching existing frameworks where possible. User code as well as built-in simulation functionality should be easy to read, i.e. concise and expressive. We give a more detailed explanation of easy-to-read code below.

Maintainability

Users should be able to read and understand all high-level source code of Φ_{ML}. All relevant framework functions should undergo continuous testing to ensure patches do not break existing code. When installing Φ_{ML}, users should be able to check the installation status and get hints as to how to solve potential issues.

Performance

Code using Φ_{ML} should be able to make use of hardware accelerators (GPUs, TPUs) where possible. During development, we prioritize rapid code iterations over execution speed but the completed code should run as fast as if written directly against the chosen ML library.

The Journal of Open Source Software

Major Design Decisions

Support for Jax, PyTorch, TensorFlow & NumPy

A large fraction of scientific code is re-written one or multiple times due to different preferences in programing languages and libraries. To avoid this as much as possible and reach a large audience, we decided to make Φ_{ML} compatible with all major Python-based ML libraries as well as NumPy, which they all integrate with. To realize this, we employ the adapter pattern (Freeman et al., 2004), creating an abstract `Backend` class with adapter subclasses for NumPy, TensorFlow, PyTorch and Jax. This API operates directly on backend-specific tensors, and we use it to implement low-level functions, such as linear algebra routines and neighborhood search. However, writing code that actually runs with all backends requires advanced knowledge of all backends due to the subtle differences between them. PyTorch, for example, does not allow negative steps in tensor slices and TensorFlow does not support assigning values to slices.

Custom `Tensor` class

The differences between the backends motivate us to provide a `Tensor` class that handles consistently across all backends. It also enables most of the additional functionality described below, making it easier to write reusable code. A Φ_{ML} tensor wraps and extends a tensor from one of the supported backends. To operate efficiently on Φ_{ML} tensors, we include a NumPy-like public API while relegating the `Backend` API to internal use. The public API takes in Φ_{ML} tenors, determines the appropriate `Backend`, and calls the corresponding low-level function. Since all backend-specific tensors are represented by the same `Tensor` class in Φ_{ML}, code written against Φ_{ML}'s public API is backend-agnostic. Data can also be passed between backends, internally using the tensor sharing functionality of DLPack (al., 2017) when possible. This way, an easy-to-use PyTorch network can interact with a Jax simulation for performance but also with an identical PyTorch simulation to facilitate debugging.

Named dimensions

In Φ_{ML}, dimensions are not referenced by their index but by name instead. We make dimension names mandatory for all dimensions, forcing users to explicitly document the meaning of each dimension upon creation. The name information gets preserved by tensor manipulations and can be inspected at any later point, e.g. by printing it or using a debugger. Named dimensions are also present in other numerics libraries, such as pandas (McKinney, 2010), xarray (Hoyer & Hamman, 2017), einops (Rogozhnikov, 2018), and are available for PyTorch as an add-on (NLP, 2019). However, these libraries make dimension names optional and, consequently, cannot support them to the same extent that Φ_{ML} can, preventing mainstream adoption. In Φ_{ML}, dimension names are one part of a carefully-designed set of tools, making them more intuitive and useful than in previous libraries. For instance, Φ_{ML} introduces the convenience slicing syntax `tensor.dim_name[start:stop:step]`, replacing the less readable slices `tensor[..., start:stop:step, :]`, and supports dimension names in all functions as first-class citizens. While naming dimensions adds a small amount of additional code, this is easily outweighed by the gains in readability and ease of debugging. Furthermore, dimension names enable automatic reshaping, which eliminates the need for reshaping operations in user code, often significantly reducing the amount of required boilerplate code.

Automatic reshaping

Named dimensions make it possible to perform reshaping, transposing, squeezing and unsqueezing operations completely under-the-hood. Φ_{ML} realizes this by aligning equally-named dimensions. Take the operation `a + b` where `a` has dimensions (x, y) and `b` has (y, z). Then Φ_{ML} will expand `a` by z and `b` by x so that both arguments have the common shape (x,y,z) before adding them. This automatic reshaping eliminates the vast majority of shape-related errors as user code is agnostic to the dimension order by default.

Element names along dimensions

In addition to naming dimensions, Φ_{ML} also supports naming slices or *items* along dimensions. This is optional but highly recommended for dimensions that enumerate interpretable quantities, such as vector components (x, y, z). UnifyML can then check at runtime that the component order is consistent, i.e. that no vector (z, y, x) is added to an (x, y, z)-ordered quantity. Additionally, the slicing syntax becomes more readable when using item names, e.g. `tensor.vector['x']` instead of the traditional `tensor[:, 0, ...]` or `tensor[:, -1, ...]` (PyTorch dimension order).

Non-uniform tensors

With some data structures, such as staggered grids, the number of elements along one or multiple dimensions can be variable. We will refer to tensors holding such data as *non-uniform* tensors, but they are also known as *ragged* or *nested* tensors. Users will often pad the missing elements with zeros to make the data easier to handle but this can lead to problems down the line. Instead, Φ_{ML} automatically creates non-uniform tensors when stacking tensors with non-matching shapes. The `shape` attribute of a non-uniform tensor stores its exact layout, allowing users to operate on non-uniform shapes like on regular shapes, e.g. allocating new memory with `zeros(non_uniform_shape)`.

Unified functional math

For differentiation, just-in-time compilation and iterative solves, we adopt a function-based approach similar to Jax. This is different from TensorFlow, where gradients are tracked via Python context managers, and PyTorch, where gradients are attached to tensors. Φ_{ML} unifies these different paradigms, providing unified function operations that run with all backends. For example, `math.functional_gradient(f)` returns a function that computes the gradient of `f` and, to solve a sparse system of linear equations, users simply supply a Python function and the desired output of that function.

Dimension types

In all backend libraries, tensor operations act only on certain dimensions, determined by the dimension index in the shape. This behaviour is generally not consistent for all backend libraries. Consider the response to extra leading dimensions in PyTorch:

- Most functions treat leading dimensions as batch dimensions, i.e. they deal with slices independently, either sequentially or in parallel.
- Some functions like `bincount` do not allow extra dimensions.
- Reduction functions like `sum` or `prod` reduce leading dimensions by default.
- Some functions, such as `histogram`, flatten all input dimensions.
- Some functions, such as `pad`, only allow a certain number of leading dimensions.

Φ_{ML} solves these issues by assigning a type to each dimension. Each of the five allowed types, *batch*, *spatial*, *instance*, *channel*, and *dual*, determines how math functions act on dimensions of that type. Spatial operations like `fft` only act on spatial dimensions and *all* functions accept tensors with any number of batch dimensions which are always preserved in the operation. Importantly, the order of dimensions is irrelevant to all math functions, only the types matter. For an explanation of all dimension types and further advantages of this system, see the online documentation.

Floating-point precision by context

Specifying the floating point precision can be a major headache in computing libraries. NumPy automatically up-casts data types (`bool` → `int` → `float` → `complex`) and floating point precision (16 bit → 32 bit → 64 bit). This can cause unintentional data type conversions

when trying to run code with a different precision, as new arrays are FP64 by default. To avoid these issues, TensorFlow has completely disabled automatic type conversion and Jax has disabled FP64 by default. Φ_{ML} solves the data type problem by enabling automatic casting but determining the desired floating point precision from the operation context rather than the data types of its inputs. The precision can be set globally or specified locally via context managers. All operations automatically convert tensors of non-matching data types. This avoids data-type-related problems and errors, as well as making user code more concise and cohesive.

Lazy stacking

Simulations often perform component-wise operations separately if there is no function achieving the desired effect with a single call, like computing the x, y and z-component of a velocity field in three lines. This often leads users to declare separate variables for the components to avoid repeated tensor stacking and slicing. However, this clutters the code and prevents it from being dimension-agnostic. Instead, Φ_{ML} performs lazy stacking by default, i.e. memory is only allocated once the stacked data is required as a block. Consequently, functions can unstack the components, operate on them individually, and restack them, without worrying about unnecessary memory allocations. This system also facilitates stacking tracer tensors, which cannot be done eagerly.

Just-in-time compilation

While the previous features allow for concise, expressive and flexible code, the added abstraction layer and shape tracking induces an additional performance overhead. To avoid this in production, Φ_{ML} supports just-in-time (JIT) compilation for PyTorch, TensorFlow and Jax. Once compiled, only the tensor operations are executed, eliminating all Python-based overhead.

Sparse matrices from linear functions

Solving linear systems of equations is a key requirement in both particle and grid-based simulations. Since the physical influence is typically limited to neighboring sample points or particles, the resulting linear systems are often sparse. Constructing such sparse matrices by hand yields code that is hard to understand and debug as well as limited to specific boundary conditions. Instead, Φ_{ML} lets users specify linear systems with a linear Python function, like with matrix-free solvers. However, these functions often consist of many individual operations, which makes it inefficient to call them at each solver iteration. To avoid this overhead, Φ_{ML} can convert most linear and affine functions to sparse matrices so that solvers can perform the matrix multiplication in a single operation. When JIT-compiling a simulation that includes a linear solve, the matrix generation will be performed during the initial tracing of the function, assuming the sparsity pattern is constant.

Compute device from Inputs

Like PyTorch, Φ_{ML} executes operations on the device where the tensors are allocated. This prevents unintentional copies of tensors as users have to explicitly declare transfer operations. This is unlike TensorFlow, where context managers can be used to specify the target device for code blocks.

Custom CUDA Operatorions

Φ_{ML} provides custom CUDA kernels for specific operations that could bottleneck simulations, such as grid sampling for TensorFlow or linear solves. If available, these will be used automatically in place of the fallback Python implementation.

Acknowledgements

We would like to thank Robin Greif, Kartik Bali, Elias Djossou and Brener Ramos for their contributions, as well as everyone who contributed to the project on GitHub.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., & others. (2016). Tensorflow: A system for large-scale machine learning. *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 265–283.

al., T. C. et. (2017). DLPack: Open in memory tensor structure. In *GitHub repository*. https://github.com/dmlc/dlpack; GitHub.

Bochenek, B., & Ustrnul, Z. (2022). Machine learning in weather prediction and climate analyses—applications and perspectives. *Atmosphere*, *13*(2), 180. https://doi.org/10.3390/atmos13020180

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs* (Version 0.2.5). http://github.com/google/jax

Brunton, S. L., Noack, B. R., & Koumoutsakos, P. (2020). Machine learning for fluid mechanics. *Annual Review of Fluid Mechanics*, *52*, 477–508. https://doi.org/10.1146/annurev-fluid-010719-060214

Butler, K. T., Davies, D. W., Cartwright, H., Isayev, O., & Walsh, A. (2018). Machine learning for molecular and materials science. *Nature*, *559*(7715), 547–555.

De La Calleja, J., & Fuentes, O. (2004). Machine learning and image analysis for morphological galaxy classification. *Monthly Notices of the Royal Astronomical Society*, *349*(1), 87–93. https://doi.org/10.1111/j.1365-2966.2004.07442.x

Freeman, E., Robson, E., Bates, B., & Sierra, K. (2004). *Head first design patterns: A brain-friendly guide*. " O'Reilly Media, Inc.".

Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H. van, Brett, M., Haldane, A., Río, J. F. del, Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Holl, P., Koltun, V., & Thuerey, N. (2020). Learning to control pdes with differentiable physics. *arXiv Preprint arXiv:2001.07457*.

Hoyer, S., & Hamman, J. (2017). Xarray: N-D labeled arrays and datasets in Python. *Journal of Open Research Software*, *5*(1). https://doi.org/10.5334/jors.148

Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., Tunyasuvunakool, K., Bates, R., Žídek, A., Potapenko, A., & others. (2021). Highly accurate protein structure prediction with AlphaFold. *Nature*, *596*(7873), 583–589.

McKinney, Wes. (2010). Data Structures for Statistical Computing in Python. In Stéfan van der Walt & Jarrod Millman (Eds.), *Proceedings of the 9th Python in Science Conference* (pp. 56–61). https://doi.org/10.25080/Majora-92bf1922-00a

NLP, H. (2019). *NamedTensor*. https://github.com/harvardnlp/NamedTensor.

Ntampaka, M., Trac, H., Sutherland, D. J., Battaglia, N., Póczos, B., & Schneider, J. (2015). A machine learning approach for dynamical mass measurements of galaxy clusters. *The Astrophysical Journal*, *803*(2), 50. https://doi.org/10.1088/0004-637X/803/2/50

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., & others. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, *32*.

Petroff, M. A., Addison, G. E., Bennett, C. L., & Weiland, J. L. (2020). Full-sky cosmic microwave background foreground cleaning using machine learning. *The Astrophysical Journal*, *903*(2), 104. https://doi.org/10.3847/1538-4357/abb9a7

Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, *378*, 686–707. https://doi.org/10.1016/j.jcp.2018.10.045

Rauber, J., Bethge, M., & Brendel, W. (2020). EagerPy: Writing code that works natively with PyTorch, TensorFlow, JAX, and NumPy. *arXiv Preprint arXiv:2008.04175*. https://eagerpy.jonasrauber.de

Rodriguez-Galiano, V., Sanchez-Castillo, M., Chica-Olmo, M., & Chica-Rivas, M. (2015). Machine learning predictive models for mineral prospectivity: An evaluation of neural networks, random forest, regression trees and support vector machines. *Ore Geology Reviews*, *71*, 804–818. https://doi.org/10.1016/j.oregeorev.2015.01.001

Rogozhnikov, A. (2018). *Einops*. https://github.com/arogozhnikov/einops.

Rolnick, D., Donti, P. L., Kaack, L. H., Kochanski, K., Lacoste, A., Sankaran, K., Ross, A. S., Milojevic-Dupont, N., Jaques, N., Waldman-Brown, A., & others. (2022). Tackling climate change with machine learning. *ACM Computing Surveys (CSUR)*, *55*(2), 1–96.

Um, K., Brand, R., Fei, Y. R., Holl, P., & Thuerey, N. (2020). Solver-in-the-loop: Learning from differentiable physics to interact with iterative pde-solvers. *Advances in Neural Information Processing Systems*, *33*, 6111–6122.

Vamathevan, J., Clark, D., Czodrowski, P., Dunham, I., Ferran, E., Lee, G., Li, B., Madabhushi, A., Shah, P., Spitzer, M., & others. (2019). Applications of machine learning in drug discovery and development. *Nature Reviews Drug Discovery*, *18*(6), 463–477.

Van Rossum, G., & Drake, F. L. (2009). *Python 3 reference manual*. CreateSpace. ISBN: 1441412697

Wei, J., Chu, X., Sun, X.-Y., Xu, K., Deng, H.-X., Chen, J., Wei, Z., & Lei, M. (2019). Machine learning in materials science. *InfoMat*, *1*(3), 338–358.