Reducing the overhead of coupled ML models between Python and Fortran: an application to Gravity Wave Parameterizations

Extended abstract

Jack Atkinson[1], Simon Clifford[1], David Connelly[2], Chris Edsall[1], Athena Elafrou[3], Ed Gerber[2], Laura Mansfield[4], Dominic Orchard[1,5], Aditi Sheshadri[4], Y. Qiang Sun[6], and Minah Yang[2]

[1]University of Cambridge, [2]New York University, [3]NVIDIA, [4]Stanford University, [5]University of Kent, [6]Rice University

Machine learning (ML) has recently been demonstrated as a viable approach to achieving one or both of higher computational performance and higher predictive performance in climate models. One notable use is for subgrid models (so-called "parameterizations"), where a neural network is trained against observational data or against a higher resolution dynamical model. In an era where scaling of hardware performance has slowed, ML models provide a way to push through performance barriers [2]. This approach typically then leads to a technical software engineering challenge of *language interoperation*: Python is the most popular language for building ML models due to libraries just as TensorFlow, PyTorch, and scikit-learn, whilst Fortran remains the language *du jour* for GCMs and other intermediate models. Implementations of Fortran and Python have different data models and so aligning the two languages requires some care.

Interoperability libraries such as `forpy` [5] can be used where the data structures of Python can be accessed from Fortran. The `forpy` package establishes native Python types that are accessible from Fortran. The standard `forpy` workflow involves importing relevant Python modules to Fortran and initialising objects (e.g, a PyTorch neural network model class) for inference. Fortran arrays are then passed in and out of the `forpy` interface, with some necessary restructuring.

In this work, we ask, is the `forpy` method the best approach? We present approaches for TensorFlow and PyTorch that avoid the use of `forpy` and show that higher performance can be achieved via more direct couplings between machine learning libraries and Fortran. Each approach has its unique characteristics however, based on the target library, although they share the common goal of overcoming overhead in more general coupling approaches.

We demonstrate our technique in the context of modelling atmospheric gravity waves. Atmospheric gravity waves are forced by disturbances in a stratified flow, such as orography (mountains), convection, and fronts. They propagate away from the source, primarily upwards, until wave breaking occurs, at which point they deposit momentum. This transfer of momentum from the lower to middle and upper atmosphere is one of the main drivers of middle atmosphere circulation. However, because gravity waves span a wide range of spatial scales, from tens to thousands of km, a large portion of their spectrum is not resolved explicitly by the numerical scheme of a climate model, which are limited to resolutions of e.g 100-300 km. Instead, climate models represent subgrid-scale gravity waves through parameterizations. A test for the effectiveness of gravity wave parameterizations (GWP) is the replication of the Quasi-Biennial Oscillation (QBO). The QBO is the main mode of variability in the equatorial stratosphere [3], featuring alternating westerly and easterly winds (between 5-100 hPa) that propagate downwards with time. The QBO is fairly consistent across the observational record with a period of $\sim$28 months. In the absence of gravity wave parameterizations, climate models have not produced a spontaneous QBO. Recently, Espinosa et al. [4] reproduced the QBO in an atmospheric model using a data-driven approach instead of the well-known GWP scheme of Alexander and Dunkerton [1]. The implementation leverages `forpy` and TensorFlow; we use this as a starting point for evaluating our techniques.

Fortran to TensorFlow coupling TensorFlow's core code is written in C++. There are three official APIs: C++, Python, and a C API. To interface directly to Fortran we use the C API, taking advantage of modern Fortran's `iso_c_binding` intrinsic module. We built the `fortran-tf-lib` library, a Fortran wrapper around the C API which performs some of the necessary translations between Fortran and C, such as string handling. It is not expected that users will build or train models directly from Fortran, so our library exposes only the functions necessary to load and infer a previously trained and saved model.

The TensorFlow C API is designed with the expectation that clients can construct and parse Protocol Buffers (protobufs) (Google's interchange format). Most of the time it is possible to use the API to generate and consume the appropriate buffers. However, there are two occasions when the client must parse buffers within the model: (1) loading the model and (2) forming the linkage between tensors and the model's inputs. There are no functions within the API to assist the user however: the user must either write extra protobuf parsing code, or have pre-knowledge of the model's parameters.

To overcome this hurdle and aid coupling, we developed `process_model`, a tool that examines TensorFlow SavedModel representations of an ML network, extracts the necessary parameters from the model, and code-generates a Fortran module that can load and run the models using the Fortran TensorFlow library. The tool uses the TensorFlow Python API to extract the tag sets present in the models, which are needed to load models, and the GraphOperation names and indices, which are needed for inference. The Fortran module provides procedures to load the models once, into module scope variables. There are helper routines to associate Fortran arrays with appropriately shaped and typed TensorFlow tensors. The inference routines have the GraphOperation names and indices hard-coded. Finally, a de-initialisation routine releases memory held by the TensorFlow library.

The code produced is well-commented and documented, as it is intended to be flexible and easy to modify to fit the user's needs. Using the generated code, the user will make one procedure call each to load then infer a model with one more call per tensor association. This will usually be far less code than the `forpy` approach, and without the inherent issues of interfacing Python and Fortran.

In the GWP, early tests show a speedup of around $3\times$ when comparing only the time spent in the `forpy` coupling code and model evaluation to time spent in the TensorFlow direct coupling code. Since the model evaluation will be identical once execution reaches the TensorFlow C++ framework we can say this speedup correlates to a performance increase when using the direct coupling. These early results may be affected by some non-optimal decisions in the `forpy` coupling code.

Fortran to PyTorch coupling PyTorch is another popular library for building ML models. PyTorch is commonly written and run using Python, but models can also be written in TorchScript, a statically-typed subset of Python. PyTorch provides methods to convert Python models to TorchScript by *tracing* (tracking tensor operations) or *scripting* (conversion of Python code into TorchScript). Once a model has been converted to TorchScript it may be compiled and interpreted using a Just-In-Time (JIT) compiler.

Like TensorFlow, PyTorch provides a C++ API capable of interfacing and calling TorchScript code. We adopt a similar approach to that for TensorFlow above, interfacing between TorchScript and Fortran using this API. We built the `fortran-pytorch-lib` library, a Fortran wrapper around the C++ API (via an intermediate C wrapper) using the modern Fortran `iso_c_binding` intrinsic module. This library performs the required translations for loading and inferring previously trained and saved TorchScript models. Thus we avoid direct use of Python and leverage the significant advantages of shared memory between Fortran and the TorchScript model.

The `fortran-pytorch-lib` library is well-commented and documented, with examples to guide users with a variety of possible use-cases. Early tests also indicate a speedup when compared to `forpy` coupling code, with comparisons between the PyTorch and TensorFlow coupling routines in progress.

Next steps Our next steps are to show benchmarking results of running the GWP using the `forpy` and direct coupling methods, coupling to TensorFlow and PyTorch models. The models will have the same internal structure and parameters. We will discuss strategies for processing the data as it passes between the GWP and the ML framework, such as when to reshape or normalise the data.

References

[1] M. J. Alexander and T. J. Dunkerton. "A Spectral Parameterization of Mean-Flow Forcing due to Breaking Gravity Waves". EN. In: *Journal of the Atmospheric Sciences* 56.24 (Dec. 1999), pp. 4167–4182. DOI: `10.1175/1520-0469(1999)056<4167:ASPOMF>2.0.CO;2`.

[2] V Balaji. "Climbing down Charney's ladder: machine learning and the post-Dennard era of computational climate science". In: *Philosophical Transactions of the Royal Society A* 379.2194 (2021).

[3] Mauro Dall'Amico et al. "Stratospheric temperature trends: impact of ozone variability and the QBO". In: *Climate Dynamics* 34.2-3 (Feb. 2010), pp. 381–398. DOI: `10.1007/s00382-009-0604-x`.

[4] Zachary I. Espinosa et al. "Machine Learning Gravity Wave Parameterization Generalizes to Capture the QBO and Response to Increased CO $_2$". In: *Geophysical Research Letters* 49.8 (Apr. 2022). ISSN: 0094-8276, 1944-8007. DOI: `10.1029/2022GL098174`.

[5] Elias Rabel et al. URL: `https://github.com/ylikx/forpy`.