Computer Physics Communications 296 (2024) 109052

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Computer Programs in Physics

RoseNNa: A performant, portable library for neural network inference with application to computational fluid dynamics [☆,☆☆]

Ajay Bati, Spencer H. Bryngelson [*]

School of Computational Science & Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA

ARTICLE INFO

Keywords:
Computational fluid dynamics
Neural Networks
Inference
Partial differential equation solvers

ABSTRACT

The rise of neural network-based machine learning ushered in high-level libraries, including TensorFlow and PyTorch, to support their functionality. Computational fluid dynamics (CFD) researchers have benefited from this trend and produced powerful neural networks that promise shorter simulation times. For example, multilayer perceptrons (MLPs) and Long Short Term Memory (LSTM) recurrent-based (RNN) architectures can represent sub-grid physical effects, like turbulence. Implementing neural networks in CFD solvers is challenging because the programming languages used for machine learning and CFD are mostly non-overlapping, We present the roseNNa library, which bridges the gap between neural network inference and CFD. RoseNNa is a non-invasive, lightweight (1000 lines), and performant tool for neural network inference, with focus on the smaller networks used to augment PDE solvers, like those of CFD, which are typically written in C/C++ or Fortran. RoseNNa accomplishes this by automatically converting trained models from typical neural network training packages into a high-performance Fortran library with C and Fortran APIs. This reduces the effort needed to access trained neural networks and maintains performance in the PDE solvers that CFD researchers build and rely upon. Results show that RoseNNa reliably outperforms PyTorch (Python) and libtorch (C++) on MLPs and LSTM RNNs with less than 100 hidden layers and 100 neurons per layer, even after removing the overhead cost of API calls. Speedups range from a factor of about 10 and 2 faster than these established libraries for the smaller and larger ends of the neural network size ranges tested.

Program summary
Program Title: RoseNNa
CPC Library link to program files: https://doi.org/10.17632/srbrfx8k74.1
Developer's repository link: https://github.com/comp-physics/roseNNa
Licensing provisions: MIT
Programming language: Fortran90 and Python
Nature of problem: Neural network applications in computational fluid dynamics exhibit high promise. However, they are often not deployed on HPC systems due to limited deep-learning support in Fortran or C/C++. Previous attempts to solve this issue rely on significant manual intervention. This leaves researchers with no practical solution for conveniently porting their trained models for use in performant HPC codebases.
Solution method: RoseNNa converts trained neural networks from popular machine learning frameworks to Fortran/C code for inference, requiring minimal manual intervention. Metaprogramming enables automatic code creation, and ONNX ensures flexible machine learning library support.

☆ The review of this paper was arranged by Prof. W. Jong.
☆☆ This paper and its associated computer program are available via the Computer Physics Communications homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).
* Corresponding author.
E-mail address: shb@gatech.edu (S.H. Bryngelson).

https://doi.org/10.1016/j.cpc.2023.109052
Received 30 July 2023; Received in revised form 1 December 2023; Accepted 6 December 2023
Available online 14 December 2023

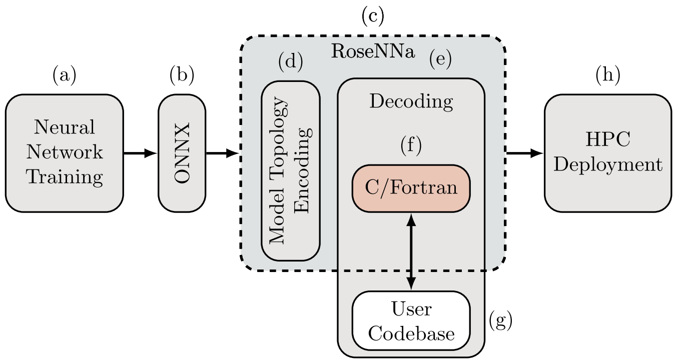

Fig. 1. RoseNNa (c) is a neural network converter that integrates into the inference process. It encodes the ONNX-converted neural network and transforms it into performant Fortran code with C and Fortran APIs. The user provides the components outside of (c).

1. Introduction

Deep learning has received considerable attention due to the availability of data and increasing computational power. Computational fluid dynamics (CFD) practitioners have been developing neural-network-based models to enhance traditional closures models and numerical methods. For example, Fukami et al. [1] implemented a convolutional autoencoder and multilayer perceptron (MLP) to speedup turbulence simulations, and Zhu et al. [2] showed how multiple artificial neural networks (ANNs) can model turbulence at high Reynolds numbers. Laubscher and Rousseau [3] used variational autoencoders and MLPs to predict cell-by-cell distributions of temperature, velocity, and species mass fractions for turbulent jet diffusion flames. Pirnia et al. [4] used an ANN to predict drag force on smaller particles and their results showed promise in improving drag force weighting in the coarse-grid method. Lastly, Baymani et al. [5] evaluated the capabilities of their feed-forward neural network by examining the electroosmotic flow through a two-dimensional microchannel. They found their MLP model was a fast solution to the Navier-Stokes equations. Of course, there are many other such examples. These trained models show promising results but are often not integrated into high-performance solvers to deploy the model at scale. Since the neural networks are typically constructed via Python-based learning libraries like PyTorch, it is unclear how to most efficiently introduce them into CFD and other PDE solvers written in low-level languages like C and Fortran.

Researchers have proposed solutions to bridge the gap between the Python and HPC domains for deep learning. Currently, the most frequently updated Fortran framework for this task is neural-Fortran [6], which supports building, training, and model parallelism in Fortran. However, porting pre-trained neural networks to Fortran using this framework would require understanding their deep learning documentation, manually rewriting the model's architecture, and transferring the trained model's parameters. Other attempts to solve the lack of deep learning support in HPC codebases also focus on manually specifying the architecture or converting neural network models from a single Python library to an HPC-amenable language. For example, Fortran–Keras Bridge (FKB) [7] is derived from neural-Fortran and specializes in Keras-based models, NEURBT [8] focuses on neural networks for classification, FANN [9] describes a C library for multilayer feed-forward networks, and SAGRAD [10] (like NEURBT) implements training in Fortran77.

We avoid these drawbacks via a fast and non-intrusive automatic conversion tool. This manuscript presents an open-source library called RoseNNa that achieves these tasks. RoseNNa is available under the MIT license at github.com/comp-physics/roseNNa. As shown in Fig. 1, RoseNNa encodes pre-trained neural networks from common machine learning libraries via an ONNX-backend and uses fypp, a Python-to-Fortran metaprogramming language, to generate the library.

RoseNNa targets inference of the artificial neural networks used for PDE- and CFD-based modeling, supporting commonly used architectures and activation functions in these areas. These network architectures were revealed via a literature survey of about 25 papers that used neural networks for modeling and numerics tasks. This survey found that MLP implementations consisted of at most 6 hidden layers, and 85% of them have fewer than 15 hidden layers (or dimensionality), and the remaining fraction having fewer than 100 neurons per layer.[1] We also reviewed 10 articles using LSTM architectures for similar purposes.[2] 90% of implementations use fewer than 64-time steps in the memory layer and a have a hidden dimension smaller than 32. These numbers provide the architectures that RoseNNa should support with high performance. The results are also consistent with expectations: PDE solvers on discretized grids with many elements that require many iterations (or time steps) cannot afford the evaluation of large neural networks since they involve relatively many floating point operations.

This manuscript discusses the methodology surrounding RoseNNa and example applications to CFD. Section 2 introduces the architecture of RoseNNa that enables its flexibility and speed. Section 3 describes its user-friendly interface and design, and Section 4 discusses RoseNNa's performance results against Python-based libraries for popular CFD architectures. We conclude in section 5 with a discussion of the primary results and use cases of the RoseNNa tool.

2. Design strategy

2.1. Design options

RoseNNa follows two main processes: read and interpret the Python-native model (encode, Fig. 1 (d)) and reconstruct it in Fortran/C (decode, Fig. 1 (e)). The tool decomposes key aspects of a neural network to define its structure: trained weights (values and dimensions), layer functionality, activation functions, and the order of layer connections. Using this encoded information, RoseNNa can reconstruct the functionality of a neural network in Fortran/C.

Users first convert their model to a unified format via ONNX [11] (Fig. 1 (b)), a library that provides interoperability between machine learning libraries, including sklearn [12], PyTorch [13], TensorFlow [14], and Caffe [15]. These libraries share common characteristics in their intermediary representations and the functionality of a neural network model. Still, they differ in their layer encoding. ONNX unifies these differences.

Currently, RoseNNa must support only a subset of all possible neural network layers (here: LSTMs, convolutions, pooling layers, and MLPs) and their features and activation functions (here: Tanh, ReLU, Sigmoid).

[1] Search terms: "mlp turbulence modeling," "mlp cfd solver," "multilayer perceptron computational fluid dynamics".

[2] Search terms: "subgrid closures rnn," "lstm rnn cfd solver," "lstm turbulence closure," "lstm rnn rans cfd".

During the "Encoding" process, these layers and their features are detected, and hyperparameters and their ordering are recorded. RoseNNa only proceeds if the detected input features are supported. Otherwise, a log indicating the unsupported feature is sent to standard output.

The ONNX-interpreted model is decoded using fypp (Fig. 1 (f)), a Python-based pre-processor for Fortran codes. The activation functions, model layers, and data structures holding model weights are first extracted via fypp and then stored in RoseNNa, specifically in Fortran code. This ensures no speed is lost to reading in needed values while conducting inference. ONNX encodes the model's structure while RoseNNa restores its graph interpretation using fypp.

Alternative solutions to Python-to-HPC model conversion are also viable. For example, Python functionality can be integrated into Fortran by running an instance of Python or exposing a model's outputs through APIs. These attempts, however, are susceptible to cascading overhead time issues. The library we present, RoseNNa, removes this overhead and enables quick HPC deployment, features that CFD practitioners require for running simulations. The power of RoseNNa comes from its internal management of neural networks, ONNX backend, and Fortran/C support.

Like established linear algebra libraries like BLAS and LAPACK, RoseNNa is a Fortran library. This is an appropriate fit since the library's focus is fast evaluation of rather simple mathematical functions, like small matrix–matrix and matrix–vector products. In addition, with recent updates to the language, Fortran can be readily linked to C, which is also often used for these applications. Fortran compilers are well optimized and can efficiently handle small matrix–matrix multiplies. We found that the optimized Python-based inference speeds for smaller model architectures are similar to the speeds seen in RoseNNa, as shown in Fig. 2 and Fig. 3.

2.2. ONNX (encoding)

Open Neural Network Exchange (ONNX) [11] is an open-source artificial intelligence (AI/ML) ecosystem that allows for interoperability between preexisting machine learning libraries and provides inference optimizations. During the pre-processing stage, RoseNNa encodes the neural network model. This entails parsing and storing each layer's order, weights, dimensions, and other functionality in the library. Any supported feature is recorded, regardless of the size of the input model. We use ONNX to unify differences between neural network model interpretations and establish a common parser that can be optimized at compile time. Users can convert their model to the ONNX due to its widespread interoperability support. ONNX is often used in research and industry. For example, Someki et al. [16] used ONNX to unify functionality support and Moreno et al. [17] converted a PyTorch model to a TensorFlow graph for compatibility with testing software. Like Rodriguez and Dassatti [18], RoseNNa reconstructs models from deep-learning libraries, enabling model designers to keep their native framework.

2.3. Metaprogramming (decoding)

The transition from model topology encoding Fig. 1 (d) to the decoding stage in Fig. 1 (e) is performed by a Python-based Fortran pre-processor called fypp [19]. In our implementation, fypp translates a neural network's properties into Fortran code *before* compile-time, thus exposing compiler optimizations. This decoding process is unique to each neural network, and so is re-run for different neural network models. The basic functionality of fypp is sufficient to extend RoseNNa with new features. Fypp is well maintained and is updated due to its wide usage in the Fortran community, though even if it were not, a deprecated fypp version would suffice for the RoseNNa's purposes.

After interpreting the Python-native model, we store its features and important variable definitions in fypp files. This encoding process stores the layers, activation functions, and weight parameters while preserving their order. We record these layers' specific options, including whether transposing is required and hyperparameter constants. We designed the tool to record an arbitrary-sized input, looping over each detected component and recording their options. We store the pre-processed information in a text file. RoseNNa tracks changes in matrix shapes, allowing it to define variables with their appropriate dimensions in Fortran explicitly. It also stores the output names of each layer so they can be referenced during the decoding phase in Fortran. To increase readability, these output names are only defined when the input undergoes dimension changes.

The decoding stage (Fig. 1 (f)) references each component described above, also looping over the recorded features. Using fypp, the layers are called in order with their respective weights, constants, and other supplementary options. The encoding and decoding process does not have a well-defined upper-limit on input size, limited implicitly by the hardware memory. As such, RoseNNa can handle hardware-supported inputs, in part owing to fypp's ability to process arbitrary-length Python code.

2.4. RoseNNa capabilities

RoseNNa was designed to support a broad range of neural network architectures in CFD. As discussed in section 1, these primarily include MLPs and LSTM RNNs. RoseNNa also supports other architectures, such as convolutional and pooling layers, which are generally popular and could become more broadly used in CFD solvers in the future. RoseNNA was built and tested using CPU execution, currently supporting single- and double-precision models. One can expand RoseNNa for different architectures, activation functions, model precision, and other cases as needed by following the RoseNNa contributor's guide. Adding these new features to the tool requires only a basic understanding of the architecture functionality and how ONNX encodes it.

2.5. Test suite

While the conversion process may compile and a given input runs through the converted model successfully, we confirm that the pre-processing and feature functionality is correct. For this, we use continuous integration (CI) testing to ensure the converted model output matches the outputs of the original Python-native model. Any GitHub pull request or commits trigger a CI run of the test suite to ensure previous functionality remains intact and added functionality is correct.

At the time of writing, RoseNNa has 17 tests, each executed via CI with each pull request and commit. RoseNNa tests the core functionality of each layer (LSTM, MLP, Maxpool, Avgpool, and Convolutions). Then, it creates cases to test different hyperparameters of these layers, including size, bias, and stride. Composition tests are also included, combining different layers and activation functions. We advise users who are extending RoseNNa's functionality to follow the test suite documentation and add core, option, and composition tests to ensure pre-processing and internal functionality are correctly implemented.

3. User interface

3.1. Using RoseNNa

The user will have access to all files that make up the library. RoseNNa is designed for straightforward and non-intrusive integration in existing codebases. As described in the pipeline of Fig. 1 (b), the only required input to RoseNNa is an ONNX-format pre-trained neural network model. Simple pre-processing using the metaprogramming language fypp reconstructs the neural network, creating a custom Fortran file with an organized subroutine defining the model's structure. Compiling all core files and the fypp-transcribed file creates a library that can be linked with an existing code (Fig. 1 (g)).

```
#: if tup[0] == 'Gemm'
!===Gemm Layer===
call linear_layer(${tup[1][0]}$, &
  linLayers(${layer_dict[tup[0]]}$), &
  ${1-tup[1][1]}$)
```

Listing 1: Fypp code to generate a linear layer.

$\xrightarrow{\text{fypp}}$

```
!===Gemm Layer===
call linear_layer(input, &
  linLayers(1),0)
```

Listing 2: Corresponding Fortran.

```
program example
  use rosenna !import the library
  implicit none
  real(c_double), dimension(1,2) :: inputs
  real(c_double), dimension(1,3) :: output

  inputs = reshape((/1.0, 1.0/), (/1, 2/), order=[2, 1])
  call initialize() !initialize/load in the weights
  call use_model(inputs, output) !conduct inference,
      store output
end program
```

Listing 3: Example Fortran90+ program invoking RoseNNa.

Using RoseNNa in C, except for defining headers for function calls, follows the same procedure. To use RoseNNa one imports it, which automatically reads and initializes the parameters encoded from the trained neural network, and then calls the model's forward subroutine with the same inputs as the native model. Listing 3 shows this lightweight approach.

3.2. Error handling

RoseNNa supports a subset of neural network capabilities, as mentioned in section 2. These include MLP, LSTM, Convolutional, and Max/AvgPool layers and Tanh, ReLU, and Sigmoid activation functions. While functionality can be readily extended, unsupported features detected during the Model Topology Encoding stage will be noticed by RoseNNa and raised as an error. These are explicit errors, with the user immediately alerted of an unsupported feature. There are, however, implicit errors that may occur when the encoded model's predicted outputs are not the same as its Python-native model counterpart (the input). This can be associated with incorrect pre-processing or implementation of an internal feature. Users should write additional test cases for their converted models to reduce the possibility of raising errors.

4. Results

4.1. Flexibility and portability

With only a few library calls, RoseNNa can be readily integrated into existing programs. It can interface with commonly used machine learning libraries and be linked to Fortran and C, the most popular languages in CFD. RoseNNa can dynamically reconstruct neural networks and avoids any manual intervention. RoseNNa can also represent attributes of deep learning models: Layers, activation functions, important constants, and more. RoseNNa caters to smaller neural networks (MLPs and LSTMs) and supports around 90% of the most popular architectures and activation functions used in CFD research. Its simple user interface and ability to interpret ONNX-format models enable the conversion of a massive pool of promising neural networks in CFD.

4.2. Performance on example cases

We run tests on CFD's most commonly used architectures, LSTMs and MLPs, to compare inference performance differences between RoseNNa and PyTorch. We compare RoseNNa's performance to that of PyTorch because it is a representative and widely used deep learning library.

We use a single CPU core and thread for the testing. We run 100 tests in PyTorch on a single thread for each data point using randomly initialized weights. The same models curated in PyTorch were converted to and tested in RoseNNa. Then, we took the ratio of the medians of the 100 RoseNNa and 100 PyTorch times. This process was repeated 25 times for each point in Figs. 2 to 4. Each test was curated under the same hardware setting: A single thread of an Intel Xeon Gold 6226 CPU.

Note that we anticipate these results could look different on GPU hardware, and, surely, inference times would be lower for both PyTorch and RoseNNa. However, it is challenging to ascertain these differences since RoseNNa is intended to be invoked at the individual grid cell level (or a glob of grid cells). As such, the performance depends strongly on how the user handles the GPU kernels that compute other PDE-relevant operations at each grid cell. In practice, the GPU threads will likely be saturated and RoseNNa, or any other neural network inference library, would operate on a single GPU thread.

Results for MLPs are important due to their widespread usage in CFD solvers. Their straightforward architectures and computation also allow for unproblematic conversions. Most MLPs used in CFD are shallow to enable reasonable computation runtimes. Based on this and the results of our literature survey, MLPs used to solve large PDE systems like those of CFD fall within the axis limits of Fig. 2.

Fig. 2 shows that tests fall under a RoseNNa-to-PyTorch time ratio of one, indicating RoseNNa's quicker inference speeds. The ratio stays near one even for large examples such as 50 neurons and a depth of 100, which is uncommon for CFD applications. We further tested RoseNNa's inference speeds against a different PyTorch backend for consistency and to ensure we compared against the fastest version of PyTorch. Therefore, Fig. 2 (b) represents the same tests run on PyTorch with an OpenBLAS backend instead of MKL. RoseNNa is 10% faster (averaged over all 25 test cases) using this backend, but the results still fall below a one-time ratio for most CFD use cases. However, with OpenBLAS, larger architectures entail increasingly slower times.

Small-scale LSTM–RNN architectures are also often used in CFD applications. Fig. 3 shows tests conducted at different depths and hidden dimension sizes to demonstrate where RoseNNa falls compared to PyTorch inference speeds. Most CFD-based LSTMs' architectures are located below the one RoseNNa-to-PyTorch time ratio. Compared with an OpenBLAS implementation of PyTorch, RoseNNa seems to be 10% faster on average. Larger architectures lead to a slower inference time ratio as expected.

Fig. 2 and Fig. 3 incorporate published examples of LSTMs and MLPs. All four test cases lie in the bottom left corner of the graph since they are shallow architectures. A simple conversion from PyTorch or TensorFlow to ONNX allowed us to pass the model through RoseNNa's pipeline. Despite their shallow architectures, these papers reported promising results across CFD modeling tasks broadly. For MLPs, Zhang et al. [20] proposed combining an artificial neural network with a flamelet-generated manifold to solve a memory issue. Zhou et al. [21] developed a new SGS model for large-eddy simulation (LES), showing significant improvements over the conventional models. For LSTMs, Srinivasan et al. [22] found this architecture outperformed MLPs in predicting turbulent statistics in temporally evolving turbulent flows. Lastly, Li et al. [23] uses LSTMs to develop a reduced-order modeling of a wind-bridge interaction system.

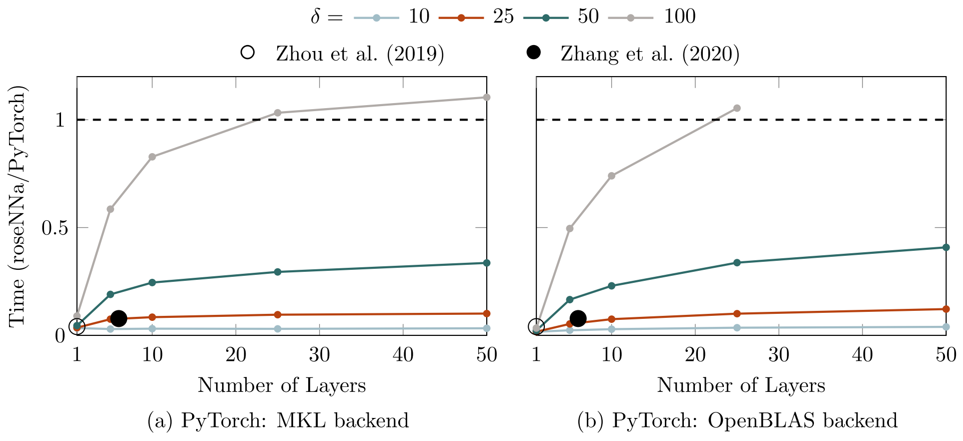

Fig. 2. Multilayer perceptron (MLP) time comparison (RoseNNa versus PyTorch). δ represents a specific hidden size (neurons per layer), and the x-axis represents the depth (number of hidden layers). Random activation functions (ReLu, Tanh, Sigmoid) were chosen for each MLP and assigned to each hidden layer.

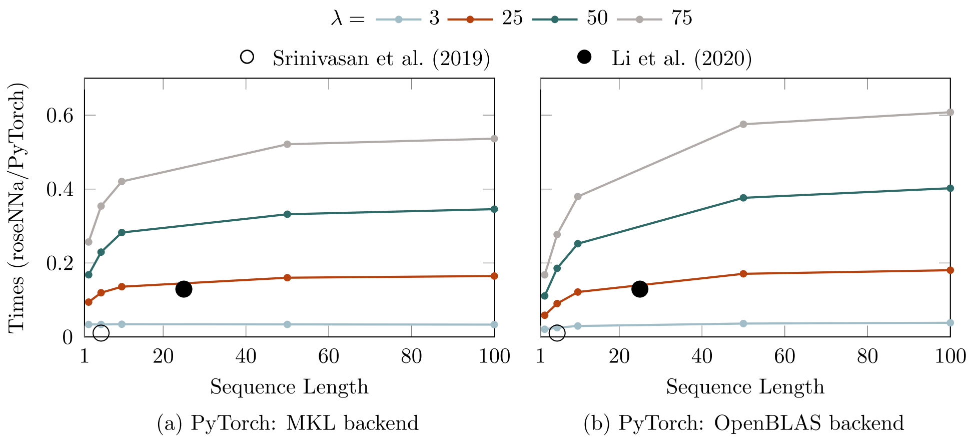

Fig. 3. Long Short-Term Memory (LSTM) time comparison (RoseNNa/PyTorch). The horizontal axis is the number of time steps (depth), and λ is the hidden dimension size. All the typical operations and activation functions were incorporated into the timing of the LSTM cells.

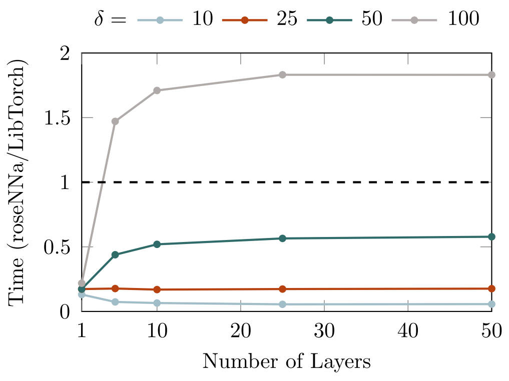

Fig. 4. Multilayer perceptron (MLP) model time comparison (RoseNNa/libtorch). δ is the hidden size, and the horizontal axis is the number of layers. Libtorch is PyTorch's C++ API. The same scheme for testing the RoseNNa to PyTorch speed ratio for MLPs was used for these tests.

4.3. Comparison to a lower-level implementation

Another approach to reducing Python overhead is to use a library's C/C++ API if supported. For example, PyTorch has a (beta) fully-native C++ API called libtorch that provides access to most PyTorch functionality [13]. Fig. 4 represents comparison tests run on the same architectures as Fig. 2 but against libtorch. Most architecture sizes are inferred faster via RoseNNa, in particular the smaller ones relevant to CFD simulation. Larger neural networks, most of which are outside the CFD scope, are still slower but near RoseNNa's speed, even for sizes as large as 15 layers of 100 neurons. Libtorch makes up some of the RoseNNa–PyTorch speed difference for larger cases, but there are still potential issues with relying upon the Torch C++ API (and other exposed backend APIs). For example, libtorch support is liable to change, which is stated directly on the Torch website. It also only provides a C++ API, thus requiring more work, like a shim layer, for use in Fortran codebases than RoseNNa.

Python-based overhead might explain part of the time discrepancy of Fig. 2 and Fig. 3, but the main contribution towards the speedup is RoseNNa's compile-time optimization and Fortran implementation. These results show RoseNNa's computationally viability for ML-enhanced CFD. For the larger architectures in Figs. 2 to 4 that were slower in RoseNNa, one can implement large matrix–matrix multiplies and other expensive calls via optimized linear algebra libraries like BLAS/LAPACK. However, based on our literature survey, these larger neural networks fall outside the CFD (and PDE-solver) scope RoseNNa focuses on. With no external dependencies, the RoseNNa library is lightweight and can be readily incorporated into existing PDE solvers.

5. Conclusions

This paper describes the design, application, and viability of RoseNNa, a neural network conversion tool for CFD codebases. It can encode a neural network's features using ONNX, a Python-based library we use to unify machine learning libraries. With a Python-powered preprocessor, fypp, RoseNNa decodes the model in Fortran. We present this tool as an alternative to manually defining neural networks in Fortran or re-implementing existing libraries' ML features. In three speed comparison benchmarks we conducted (RoseNNa/MKL, RoseNNa/OpenBLAS, RoseNNa/libtorch), RoseNNa's application in the CFD domain seemed to be promising and a more reliable alternative to low-level implementations of PyTorch, TensorFlow, or other Python-based machine learning libraries. RoseNNa supports many popular features and establishes a streamlined process for increasing its breadth.

RoseNNa presents useful benefits for neural network conversion and inference. First, it supports the conversion from Python machine-learning libraries via ONNX. It is also simple to use. As shown in Listing 3, a few API calls enable inference. Lastly, RoseNNa is a lightweight tool, enabling integration and minimal intrusiveness in existing and (potentially large) CFD codebases. The library is compiled for the neural network and linked to existing code.

RoseNNa's future lies in improving performance, adding functionality to existing architectures, and expanding to new, popular features. With the pipeline of Fig. 1, contributors can incorporate any needed feature. We have created an in-depth manuscript about our current methodology and how new contributions can be feasibly integrated (accessible at github.com/comp-physics/roseNNa). We provide steps describing which files to modify and examples, with documentation, of their functions and variables. Any new changes can be verified via the testing pipeline, allowing contributors to add new features efficiently.

CRediT authorship contribution statement

Ajay Bati: Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Spencer H. Bryngelson:** Conceptualization, Funding acquisition, Methodology, Project administration, Software, Supervision, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Spencer Bryngelson reports financial support was provided by Office of Naval Research.

Acknowledgements

This work used Bridges2 at the Pittsburgh Supercomputing Center through allocation TG-PHY210084 (PI Spencer Bryngelson) from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296. SHB also acknowledges the resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. SHB acknowledges support from the Office of the Naval Research under grant N00014-22-1-2519 (PM Dr. Julie Young). This research was supported in part through research cyberinfrastructure resources and services provided by the Partnership for an Advanced Computing Environment (PACE) at the Georgia Institute of Technology, Atlanta, Georgia, USA.

References

[1] K. Fukami, T. Nakamura, K. Fukagata, Convolutional neural network based hierarchical autoencoder for nonlinear mode decomposition of fluid field data, Phys. Fluids 32 (2020) 095110.
[2] L. Zhu, W. Zhang, X. Sun, Y. Liu, X. Yuan, Turbulence closure for high Reynolds number airfoil flows by deep neural networks, Aerosp. Sci. Technol. 110 (2021) 106452.
[3] R. Laubscher, P. Rousseau, An integrated approach to predict scalar fields of a simulated turbulent jet diffusion flame using multiple fully connected variational autoencoders and mlp networks, Appl. Soft Comput. 101 (2021) 107074.
[4] P. Pirnia, F. Duhaime, Y. Ethier, J.-S. Dubé, Drag force calculations in polydisperse dem simulations with the coarse-grid method: influence of the weighting method and improved predictions through artificial neural networks, Transp. Porous Media 129 (2019) 837–853.
[5] M. Baymani, S. Effati, H. Niazmand, A. Kerayechian, Artificial neural network method for solving the Navier–Stokes equations, Neural Comput. Appl. 26 (2015) 765–773.
[6] M. Curcic, A Parallel Fortran Framework for Neural Networks and Deep Learning, ACM SIGPLAN Fortran Forum, vol. 38, ACM, New York, NY, USA, 2019, pp. 4–21.
[7] J. Ott, M. Pritchard, N. Best, E. Linstead, M. Curcic, P. Baldi, A Fortran–Keras deep learning bridge for scientific computing, Sci. Program. 2020 (2020) 1–13.
[8] J. Bernal, J. Bernal, NEURBT: A Program for Computing Neural Networks for Classification Using Batch Learning, US Department of Commerce, National Institute of Standards and Technology, 2015.
[9] S. Nissen, Implementation of a fast artificial neural network library (FANN), Report in: Department of Computer Science University of Copenhagen (DIKU), vol. 31, 2003, p. 26.
[10] J. Bernal, J. Torres-Jimenez, SAGRAD: a program for neural network training with simulated annealing and the conjugate gradient method, J. Res. Natl. Inst. Stand. Technol. 120 (2015) 113.
[11] J. Bai, F. Lu, K. Zhang, et al., ONNX: Open neural network exchange, https://github.com/onnx/onnx, 2019.
[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al., Scikit-learn: machine learning in python, J. Mach. Learn. Res. 12 (2011) 2825–2830.
[13] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, PyTorch: an imperative style, high-performance deep learning library, Adv. Neural Inf. Process. Syst. 32 (2019).
[14] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., TensorFlow: a system for large-scale machine learning, in: 12th USENIX Symposium on Operating Systems Design and Implementation, in: OSDI, vol. 16, 2016, pp. 265–283.
[15] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, T. Darrell, Caffe: convolutional architecture for fast feature embedding, in: Proceedings of the 22nd ACM International Conference on Multimedia, 2014, pp. 675–678.
[16] M. Someki, Y. Higuchi, T. Hayashi, S. Watanabe, ESPnet-ONNX: bridging a gap between research and production, in: 2022 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC), IEEE, 2022, pp. 420–427.
[17] E.A. Moreno, O. Cerri, J.M. Duarte, H.B. Newman, T.Q. Nguyen, A. Periwal, M. Pierini, A. Serikova, M. Spiropulu, J.-R. Vlimant, JEDI-net: a jet identification algorithm based on interaction networks, Eur. Phys. J. C 80 (2020) 1–15.
[18] O. Rodriguez, A. Dassatti, Deep learning inference in GNU radio with ONNX, in: Proceedings of the GNU Radio Conference, vol. 5, 2020.
[19] B. Aradi, B. Aradi, O. Schütt, haraldkl, aradi/fypp: Release 3.0, 2020.
[20] Y. Zhang, S. Xu, S. Zhong, X.-S. Bai, H. Wang, M. Yao, Large eddy simulation of spray combustion using flamelet generated manifolds combined with artificial neural networks, Energy AI 2 (2020) 100021.
[21] Z. Zhou, G. He, S. Wang, G. Jin, Subgrid-scale model for large-eddy simulation of isotropic turbulent flows using an artificial neural network, Comput. Fluids 195 (2019) 104319.
[22] P.A. Srinivasan, L. Guastoni, H. Azizpour, P. Schlatter, R. Vinuesa, Predictions of turbulent shear flows using deep neural networks, Phys. Rev. Fluids 4 (2019) 054603.
[23] T. Li, T. Wu, Z. Liu, Nonlinear unsteady bridge aerodynamics: reduced-order modeling based on deep LSTM networks, J. Wind Eng. Ind. Aerodyn. 198 (2020) 104116.