Neural Network in Fortran and Comparison with PyTorch

Hiu Wang Lau, Pak Hong Lau, Zhiping Zhang

hiulau@student.ethz.ch

paklam@student.ethz.ch

zhangzhip@student.ethz.ch

High Performance Computing for Weather and Climate Modeling

ETH Zurich, Switzerland

Spring Semester 2025

---

1 Introduction

Applications of Artificial Intelligence (AI) deep-learning models have grown significantly in recent decades. Their amount of usage, model sizes and complexity have also increased significantly, which led to the inference time of AI models becoming a major bottleneck in real-time utilization of deep learning AI models [1]. It is therefore important to identify the various processes and operations during model inference that could potentially be optimized to reduce the inference time.

PyTorch is one of the most widely used frameworks for deep learning, and Python is the dominant programming language in this field [2]. This popularity can be attributed to its open-source nature, simplicity, and the vast ecosystem of tools it supports. However, from the perspective of high-performance computing (HPC), it is worth questioning whether this combination is truly the optimal choice, and will frameworks in other scientific computing languages perform better than PyTorch.

A complete inference workflow typically involves several stages, including data loading, model loading, and actual inference. The choice of programming language and its underlying data structures may significantly affect the efficiency of the first two stages. For the inference stage itself, the core computations are convolution operations, which are similar to the stencil programming discussed in this course. The performance in this stage is therefore influenced by multiple factors, such as the type of computing device and the degree of parallelization employed. By comparing different programming languages and frameworks, we can develop a deeper understanding of the high-performance computing mechanisms underlying neural network inference.

In this work, we first implemented a neural network model in Fortran, and then compared its inference performance with PyTorch. Some other factors, such as the device type and the depth of the neural network, were also explored through experiments. The detailed implementation is described in Section 2, and the results are shown in Section 3. Finally, we discuss about explanations for the differences in performance in Section 4, and make brief conclusions in Section 5.

2 Data and Method

2.1 UNet Temperature field extrapolation model

We will be using a custom Convolution Neural Network (CNN) based deep learning model with UNet architecture [3] to conduct inference time experiments. The model extrapolates a single two-dimensional two-meter surface temperature (t2m) field given a time series of five t2m fields in the past.

2.1.1 Model Architecture

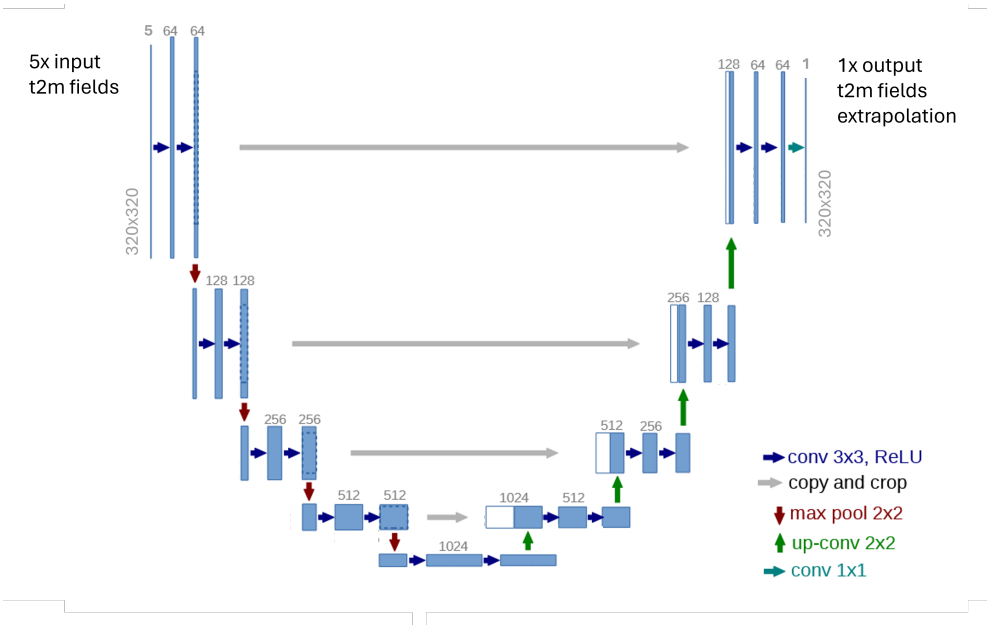

Fig. 1. Illustration of the UNet structure with depth=4. Numbers at the top of each filter represent the number of feature maps. Modified from the figure in [3].

The UNet deep-learning model consists of a contracting path and an expanding path respectively. The contracting path refers to a series of down-convolution (DownConv) modules that apply two 3x3 convolution layers, each with its corresponding rectified linear unit (ReLU) activation layer and a 2x2 max pooling (downsampling) layer, the number of feature channels doubles for each DownConv module. Similarly, the expanding path refers to a series of up-convolution (UpConv) modules that first applies a 2x2 upsampling convolution, then the output is concatenated with feature maps from the DownConv module of the same depth before applying two additional 3x3 convolutions, each with its own ReLU activation layer. Finally, at the greatest depth, the contracting and expanding paths are connected by two pairs of Convolution-ReLU layers [3].

The depth of the UNet model can be changed given the number of pairs of UpConv and DownConv modules. We set up three UNet models with varying depths and, therefore, different numbers of parameters to test for their inference times:

	D3	D5	D7
Depth (number of UpConv-DownConv pairs)	3	5	7
Number of parameters	1864001	31032897	497670721

2.1.2 Training Data Pre-processing

Two-meter temperature field time series data from the COSMO-2 Hourly Operational Analysis dataset [4] will be used as training and validation data. The data is available at a grid resolution of 2.2 km at 1-hour intervals. The dimension of a single temperature field is originally 390*582, which is cropped to 320*320 to be used in training.

2.1.3 Training and Validation

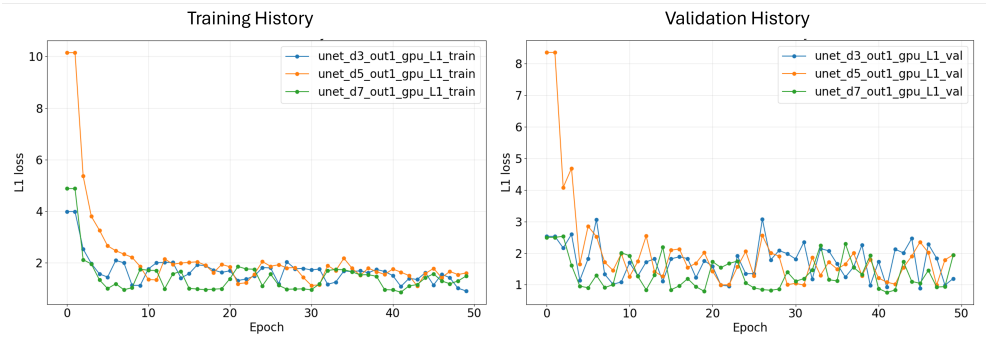

Fig. 2. Training and validation history. All models are trained for 50 epochs at a learning rate of 1e-4 and a batch size of 32 using mean absolute error (MAE) i.e. L1 loss.

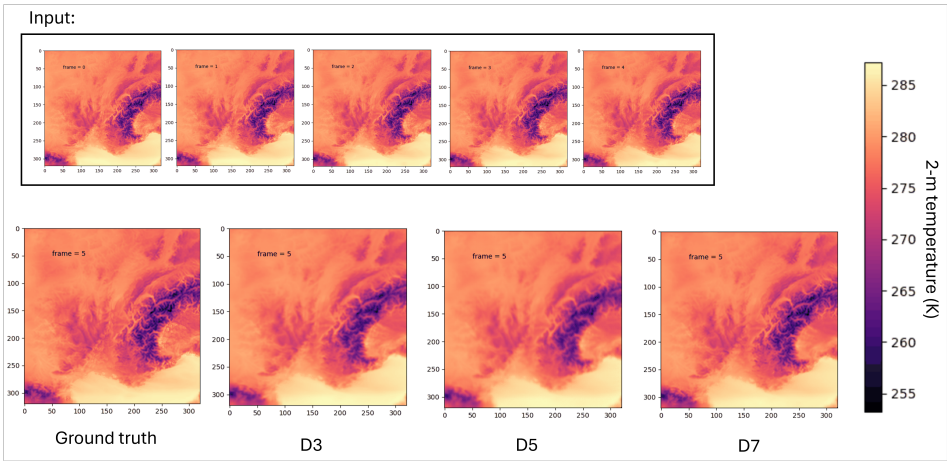

Fig. 3. Example of an input instance (top row) and the resulting extrapolated t2m fields (bottom row) and the corresponding ground truth.

From figure 2, we see that all three models successfully converged to a training L1 loss of around 1 °C and a validation loss of around 1.5 °C, the largest model D7 achieved the lowest validation loss of 0.9 °C. Meanwhile, in Figure 3, we can see that all models were able to predict the lower surface temperature over snow cover in the Swiss Alps and the meridional temperature gradients over the entire domain reasonably well. However, the model with deeper depth (D7) tends to produce less blurred predictions with more details, which demonstrated the benefit of increasing the size of deep learning models.

2.2 Libraries

2.2.1 PyTorch

PyTorch is an open source machine learning library that allows users to train and deploy machine learning models using Python [2]. Its modular structure of models provides a flexible and convenient tool for developers to design and modify models with different architectures. All the models in this project were trained using PyTorch.

2.2.2 LibTorch

LibTorch is the C++ backend of PyTorch [5]. Using the optimized C++ code implementations of LibTorch, PyTorch lets developers take advantage of the performance of C++ while using easy-to-learn Python. The models trained in PyTorch can be saved in a format called TorchScript, which can be directly loaded and inferred by LibTorch in native C++. This means any library, in any programming language, that wraps LibTorch can do PyTorch model inference in their language.

2.2.3 FTorch

FTorch is a library made for using PyTorch models directly in Fortran [6], without involving a Python program. Essentially, it is a wrapper of LibTorch written in Fortran, so users can load the models using Fortran code via the C++ backend.

2.3 Method

In this project, we will compare the run time of using a neural network in Fortran (by FTorch) and Python (by PyTorch). The model used is already described in Section 2.1. After training, the model is then converted to TorchScript format using the helper script provided by FTorch. We will load models D3, D5, and D7 in Fortran and Python respectively. After that, inference

will be done on both CPU and GPU. Since there are small fluctuations between inference times, we will do inference 100 times and take the average run time, which we call one "run".

First, we will perform 10 runs each (5 GPU, 5 CPU) using model D5 on Python and Fortran to compare the inference, data loading, and model loading times between the two languages on different devices. The input data are the same for all runs to keep the comparison fair. We will also do 10 warm-up forward passes before measuring the run time to get a more consistent result. For GPU runs, synchronization is also needed.

The performance between models with different depths will also be examined. On models D3, D5, and D7, one GPU run and one CPU run will be done on each of them. Inference times and model loading times will be measured. The measurement of data loading time will not be repeated here because this process depends only on the data loaded, the processes and hardware that load the data, but not the architecture of the model.

3 Results

Figures 4 and 5 show the box plots for the distribution of inference times. For GPU, it is shown that the inference time is of the magnitude of 0.001s, and Fortran is faster than Python. It is statistically significant that the mean inference times are different between them (with p-value $\sim O(10^{-150})$). For CPU, the inference times are of the magnitude of 0.1 s, three of the runs show a faster inference time. However, the means between Python and Fortran are not significantly different (with p-value = 0.330). Although not shown, the output of inferences for both Python and Fortran are the same, proving the portability of the model in TorchScript.

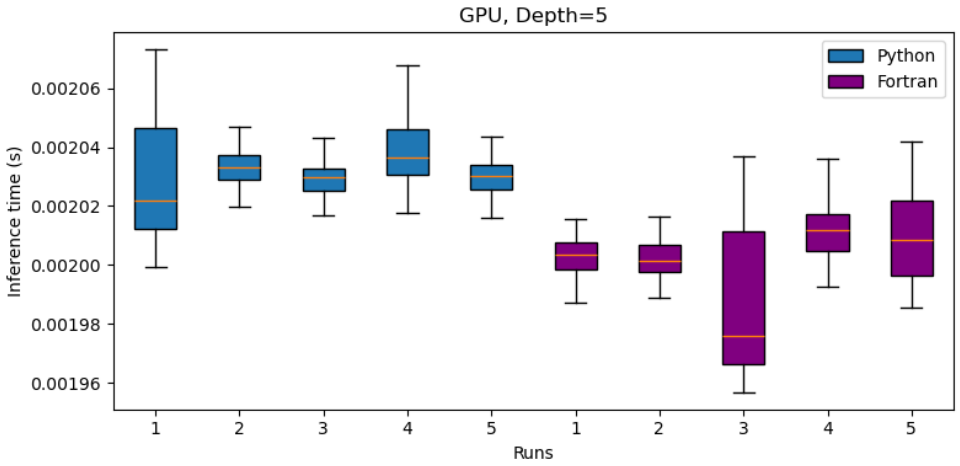

Fig. 4. The distribution of inference time for the model D5 using GPU. Each run contains 100 inference passes.

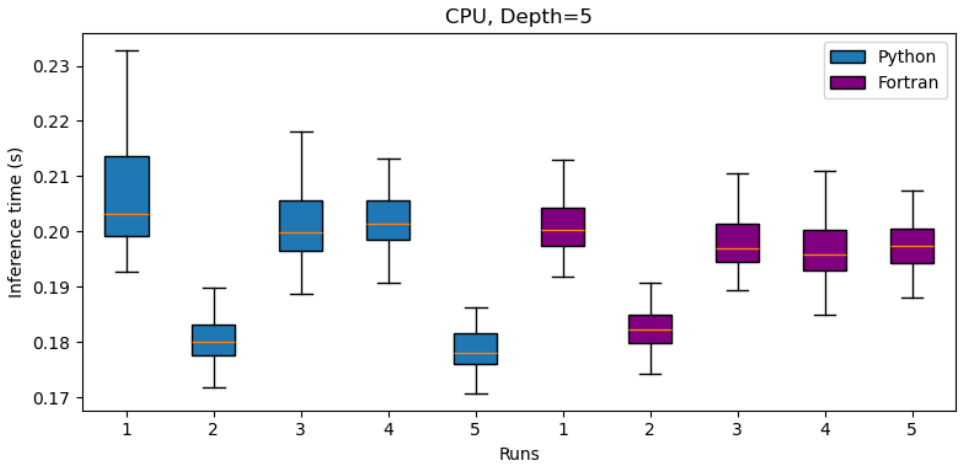

Fig. 5. The distribution of inference time for the model D5 using CPU. Each run contains 100 inference passes.

Figure 6 shows the inference times for models of different depths. The inference time increases with the depth of the model. On CPU, the inference time between forward passes fluctuates a lot, showing the need to take an average of multiple passes for the measurement of run time. For D7 in CPU, the difference in inference time between Python and Fortran is also larger than in other cases.

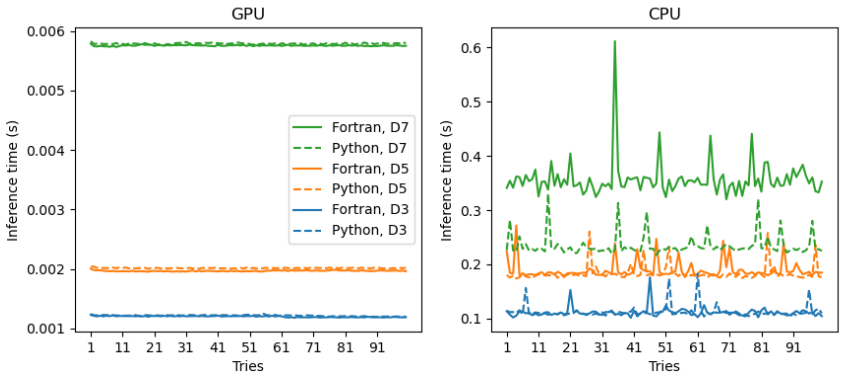

Fig. 6. Inference time for models of depths 3, 5, 7 in Fortran (solid lines) and Python (dotted lines), running on GPU (left plot) and CPU (right plot) respectively.

Figures 7 and 8 show the data loading time and the model loading time for the runs as in Figures 4 and 5. Fortran loads the data to GPU slower, but the times are similar when loading into CPU. For model loading, Fortran is slower on both GPU and CPU, and the difference is much larger on CPU.

Figure 9 displays the model loading time for models of different depths. Model D7 requires much more time to load the model, and Python loads the model faster than Fortran in all cases. For Python itself, the model loading time is similar between GPU and CPU.

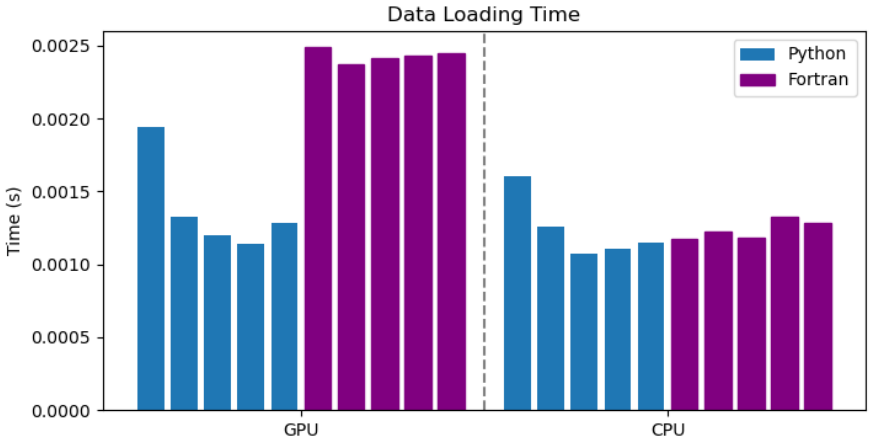

Fig. 7. The time used for loading input data from the file to memory. Same runs as in figures 4 and 5.

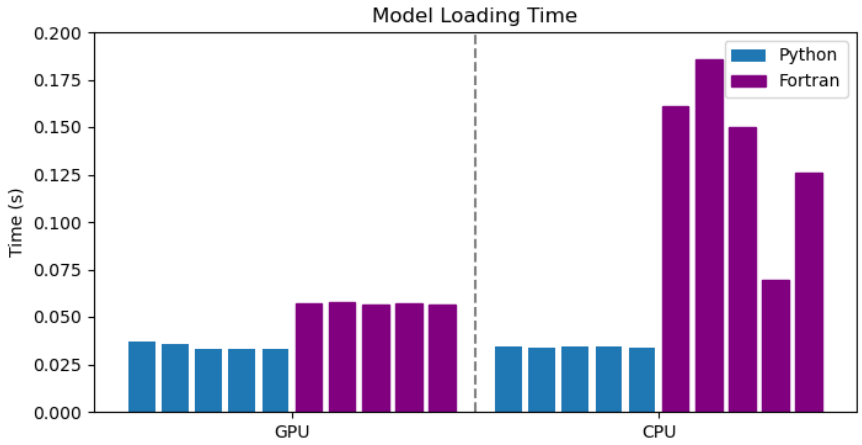

Fig. 8. The time used for loading the trained model from file to memory. Same runs as in figures 4 and 5.

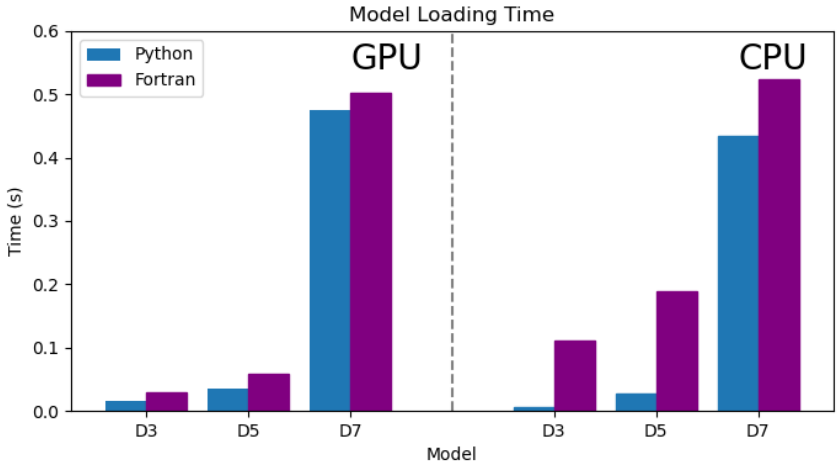

Fig. 9. The time for loading a trained model into memory for models D3, D5, D7, on GPU (left plot) and CPU (right plot).

4 Discussion

4.1 Inference

4.1.1 Model Depth = 5

From Figure 4 and Figure 5, we can see that the inference times of PyTorch and FTorch are similar. When inference is performed on GPU, the Fortran inference time is slightly lower than that of Python with significant confidence, but the difference is relatively small compared to the average inference time (around 0.002 seconds). When inference is conducted on CPU, there is no significant difference between Fortran and Python. This similarity can be attributed to the fact that both FTorch and PyTorch use the C++ library LibTorch to perform inference. In FTorch, this call path is straightforward and easy to trace. In the FTorch source code, the inference subroutine `torch_model_forward()` is bound to the C++ function `torch_jit_module_forward()`, which calls `torch::jit::script::Module::forward()` in LibTorch. In comparison, the binding to C++ in PyTorch is much more complex, passing through multiple layers of encapsulation before reaching LibTorch. This more complicated binding process in Python may introduce some overhead, slightly slowing down the inference. Nevertheless, the main inference time is still dominated by the convolution operations executed by LibTorch.

The inference process of our neural network is generally efficient, with an average inference time of 0.002s on GPU. What high-performance computing techniques does LibTorch use to optimize this process? We tried to look deeper into the LibTorch code. LibTorch is still not the lowest-level computation library. When performing CNN inference, LibTorch internally calls the convolution operator provided by the ATen library, which defines a wide range of tensor operations. The convolution operator in ATen is highly optimized for each convolution case. By using functions such as `xnnpack_use_convolution2d()` and `use_cudnn_depthwise()`, the library can assign the convolution task to different backends depending on the hardware and configuration. Many of these specialized backends are closed-source, such as cuDNN and MKL, but we can still investigate high-performance computing optimizations from the open-source library XNNPACK. In XNNPACK, there are many functions specialized for specific convolution cases. To boost execution speed, small loops such as the 3×3 kernel traversal are fully unrolled, and multiple fast paths are added to handle different boundary conditions like padding, stride, and tile edges. This loop unrolling makes the corresponding function files very large and long.

The difference in inference performance between CPU and GPU is predictable. Inference on CUDA is about 100 times faster than inference on CPU. This performance gap can be

attributed to GPU's massive parallelism, high memory bandwidth, and specialized deep learning libraries. Another difference between CPU and GPU inference lies in the variation of inference time. We observe that the variance of inference time on CPU is much larger than on GPU. This phenomenon becomes more significant in Figure 6. This instability can be explained by the fact that CPU resources are shared across many system processes, leading to fluctuations in scheduling and memory access. In contrast, GPUs are designed with a relatively single-purpose focus on parallelization, and therefore are less disrupted.

4.1.2 Different Model Depths

The inference time at different depths is shown in Figure 6. On both GPU and CPU, the inference time increases significantly as the depth grows. This result indicates that the complexity of our neural network has surpassed the parallelization limit of GPU, requiring more time to complete inference. A deeper network also implies a higher synchronization cost, since deeper layers must wait to receive data from the previous ones. We also observe that the relative advantage of GPU over CPU becomes weaker for the D7 network. This may be because the increased depth requires more space in the GPU cache, and for the D7 network, the required space exceeds the cache threshold.

4.2 Data Loading

Data loading is one of the prerequisite processes for inference. From Figure 7, we can see that Fortran exhibits generally similar data loading times on CPU compared to Python. However, importing the data to GPU adds more overhead for Fortran, indicating that PyTorch is more optimized in tensor transportation between GPU and CPU than FTorch. To make the comparison fair, the loaded data is a `.dat` file pre-exported from the PyTorch dataloader, since FTorch does not provide a direct interface to the Python dataloader. The preparation of the dataloader itself takes around 0.5 seconds, which makes it an important part of the overall inference workflow. It shows that the incomplete ecosystem in Fortran is a disadvantage compared to Python.

4.3 Model Loading

Model loading is another necessary prerequisite for inference. When loading the same TorchScript model, FTorch is slightly slower than PyTorch on GPU, but the difference is larger on CPU. Because both frameworks call LibTorch at the lower level, this further suggests that the more complex C++ binding in PyTorch may help optimize the model loading process. It should

be noted that as the network depth increases, model loading can become very time-consuming and may even dominate the overall inference time.

5 Conclusion

In conclusion, the Fortran-based neural network framework FTorch exhibits similar inference computing times to PyTorch, due to their shared core library, LibTorch. Switching the device from CPU to GPU can significantly boost performance for both frameworks. However, the incomplete ecosystem of Fortran may introduce additional overhead in data loading and model loading, making Fortran less favorable for deep learning applications.

The efficiency of FTorch, PyTorch, and LibTorch can be attributed to the highly optimized operators defined in specialized HPC computing libraries such as cuDNN and XNNPACK. For each specific computation, the program assigns the task to a suitable library based on the configuration and hardware. These libraries contain many highly optimized operation functions, often with unrolled loops. Additionally, the complex C++ binding in PyTorch may also contribute to optimizing the overall inference process, but the detailed mechanisms were not explored in this work.

References

[1] Tarek Abdelzaher et al. "The bottlenecks of AI: challenges for embedded and real-time research in a data-centric age". In: *Real-Time Systems* (June 2025).

[2] Jason Ansel et al. "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation". In: *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, Apr. 2024. DOI: 10.1145/3620665.3640366. URL: https://docs.pytorch.org/assets/pytorch2-2.pdf.

[3] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. arXiv: 1505.04597 [cs.CV]. URL: https://arxiv.org/abs/1505.04597.

[4] MeteoSwiss. *COSMO-2/KENDA Hourly Operational Analysis*. data retrieved from ETH Zurich, https://doi.org/10.3929/ethz-b-000720460. 2025.

[5] The Linux Foundation. *PyTorch C++ API*. 2025. URL: https://docs.pytorch.org/cppdocs/.

[6] Jack Atkinson et al. "FTorch: a library for coupling PyTorch models to Fortran". In: *Journal of Open Source Software* 10.107 (Mar. 2025), p. 7602. DOI: 10.21105/joss.07602. URL: https://joss.theoj.org/papers/10.21105/joss.07602.