Data Aggregator for Physics Informed Neural Networks in NVIDIA Modulus Framework

Matheus Scramignon[1], Alvaro L. G. A. Coutinho[2], Marta Mattoso[1]

[1]*Dept. of Computer and Systems Engineering, COPPE, Federal University of Rio de Janeiro*
Avenida Horácio Macedo 2030, CT, Room H319, Rio de Janeiro, RJ 21941-914, Brazil
mthlima@cos.ufrj.br, marta@cos.ufrj.br
[2]*Dept. of Civil Engineering, COPPE, Federal University of Rio de Janeiro*
Av. Athos da Silveira Ramos, 149, CT, Room B101, Rio de Janeiro, RJ 1941-909, Brazil
alvaro@coc.ufrj.br

Abstract. The development of Physics Informed Neural Networks (PINNs) has been receiving considerable attention lately. PINNs incorporate the partial derivative equations that describe the physical behavior of a natural or engineered system in the loss functions of neural networks. This model family represents a new paradigm for the solutions of PDEs for both forward and inverse problems. Different frameworks that aim to facilitate the production and training of such models are currently being provided, and Modulus is one of the available frameworks that has been gaining ground recently. In any case, despite the capability of these packages to assist the construction of PINNs, it is important to consider a viable data analysis strategy for the experiments. This work presents the *Modulus Aggregator* tool, which is developed to support the data analysis expert in the hyperparameter configuration of multiple models produced, with a strategy for the aggregation of results. The aggregation tool complements the *TensorBoard* visualization toolkit and takes advantage of the native directory structure of a Modulus experiment. The experiment of a wave propagation shows the potential to assist the analysis of results and the possibility of automating the extraction and filtering activities of trained models in a scenario of a significant amount of data.

Keywords: Physics-informed Neural Networks, Data analysis, Hyperparameter

1 Introduction

The growing popularity of Neural Networks (NNs) brought undeniable advances in a considerable wide range of different areas and problems in which traditional Machine Learning (ML) techniques are not able to settle successfully, such as computer vision [1], voice recognition [2], automated translation [3] and many others. Nevertheless, for scientific applications such as fluid dynamics, wave propagation, and quantum mechanics, the need to model the underlying physics with NNs without the constraints of high-cost simulations to produce synthetic data and sparse field data, a new model family has begun to draw attention.

The Physics Informed Neural Networks (PINNs) are NNs aimed at solving Partial Differential Equations (PDEs) that characterize forward and inverse problems by coupling the governing equations of a physical system in their loss function. Many software frameworks are being developed to facilitate the design, training, and implementation of PINNs and have become available for specialists. One of the frameworks that have been gaining ground, especially for complex applications, is the NVIDIA Modulus [1], mainly because of its robust toolset for PINNs implementation.

Despite the interesting built-in functionalities to support PINNs implementation, there is limited support to analyze and manage the results of several models that could be potentially produced in a batch of different experiments. This necessity led to the development of the Modulus Aggregator, a tool integrated with the Modulus framework built to complement the functionalities related to data processing and analysis to provide flexibility in the comparison of multiple models generated in a given experiment. This paper presents the Modulus Aggregator

[1] `https://developer.nvidia.com/modulus`

as an alternative to assist Modulus users in the data analysis of results, with a practical evaluation of the tool in a seismic wave propagation experiment.

The remainder of this paper is organized as follows. Section 2 defines the problem solved by PINNs. Section 3 briefly describes the NVIDIA Modulus Framework. Section 4 presents the Modulus Aggregator and defines its software architecture. Section 5 details the experiment for evaluating the aggregation tool. Section 6 summarizes our main findings.

2 Physics Informed Neural Networks

Consider the problem of solving the following equations:

$$\mathcal{L}(\mathbf{u}(t;\mathbf{x});\lambda) = \mathbf{f}(t;\mathbf{x}) \qquad \mathbf{x} \in \Omega, t \in [0,T] \tag{1}$$

$$\mathcal{B}(\mathbf{u}(t;\mathbf{x})) = \mathbf{g}(t;\mathbf{x}) \qquad \mathbf{x} \in \partial\Omega \tag{2}$$

where the equations 1 and 2 are defined in $\Omega \in R^D$ domain, with its boundary in $\partial\Omega$. The solution $\mathbf{u}(t;\mathbf{x})$ depends on $t \in [0,T]$, which represents the temporal variable, and $\mathbf{x} = [x_1, x_2, ..., x_D]$, that constitutes the spatial variable defined in Ω. Moreover, $\mathcal{L}$ represents the differential (non-linear) operator of the problem, λ the physical parameters, and $\mathbf{f}$ the function related to the problem's data. Finally, $\mathcal{B}$ corresponds to the operator representing the initial and boundary conditions (for example, *Dirichlet*, *Robin* or periodic) and $\mathbf{g}$ is the specified function in the boundary.

The trained PINN approximates $\mathbf{u}$ from a set of parameters θ (representing the neural network-related parameters in a forward problem, and the physical related λ parameters in a inverse problem) obtained through training, so that $(\mathbf{u}_\theta(t;\mathbf{x}) - \mathbf{u}(t;\mathbf{x}) = \mathcal{R}(\mathbf{x},t,\theta))$. In this case, $\mathcal{R}$ represents the residual between the approximated and the real solutions, which must be minimized.

Therefore, the PINN must minimize the following loss function:

$$\mathbf{L}(\theta) = w_d\mathbf{L}_d(\theta) + w_u\mathbf{L}_u(\theta) \tag{3}$$

where $\mathbf{L}_u$ represents the loss function related to the PDE's approximation that aims at minimizing $\mathcal{R}(\mathbf{x},t,\theta)$, and $\mathbf{L}_u$ corresponds to the loss function related to the problem's data, whether they are derived from initial or boundary conditions, or even from collected data. The weights w_d and w_u balance the total loss function, and θ are the network parameters. In this way, the objective is to find $\theta^* = \arg\min(\mathbf{L}(\theta))$, where θ^* minimizes the loss function defined in equation 3.

There are several software packages to facilitate the generation and training of PINNs. These packages take advantage of the great computational efficiency of standard optimization methods such as automatic differentiation by backpropagation that are implemented in deep neural networks frameworks like TensorFlow [4], Keras [5] and PyTorch [6]. In that sense, it is worth mentioning PINNs frameworks such as *DeepXDE* [7], *SciANN* [8] and *NeuroDiffEQ* [9], that, despite their popularity for designing and training PINNs, do not offer a consistent set of features and network architectures to support solving problems defined in more complex geometrical domains and to analyze experiment results natively via *TensorBoard* for instance, as Modulus does.

3 NVIDIA Modulus

The NVIDIA Modulus, previously known as SimNet [10], is a PINN development framework that aims to offer both academy and industry a solid option for designing and training physics-based neural networks. It presents itself as a robust toolset for implementing PINNs. Modulus supports advanced Neural Networks architectures and provides for the definition and parameterization of complex geometries by constructive solid geometry (CSG) and Tessellated geometry (TSG) modules. In NVIDIA Modulus documentation page [2] it is possible to access different

[2] `https://docs.nvidia.com/deeplearning/modulus/`

kinds of resources, such as the Modulus documentation, Modulus code examples, the Modulus GitHub, and also a self-paced course to learn more details on how to design train PINNs on the framework.

Despite advanced available functionalities, there are some limitations regarding model results comparison [11]. For example, the *TensorBoard* package, which is totally integrated with Modulus, provides the user with interesting features to visually analyze the progress of trained PINNs during different training phases. However, it lacks some features to aggregate results on demand and to compare the models with previously set metrics quantitatively. Accessing results data as data frames through the *TensorBoard Dataframe API* is the only possibility. Nevertheless, it is required that the user uploads the whole Modulus experiment in the *TensorBoard.dev* platform, which publicizes the experiment results. Uploading the results is not feasible in many research projects and industry development applications scenarios. With this, it is interesting that these limitations could be mapped in an adequately integrated software package that could be directly used with Modulus. That is the idea behind the development of the Modulus Aggregator.

4 Modulus Aggregator

The Modulus Aggregator is a tool developed to assist the data analysis expert in the hyperparameter configuration in a scenario of multiple trained models in a Modulus Experiment. This is conducted with a strategy to aggregate results to complement some of the mapped limitations found on the Modulus framework related to multiple models' quantitative results analysis process. It also takes advantage of the native directory structure of a Modulus experiment to minimally affect the generated artifacts, referred to as the multiple trained PINNs. This software is developed as an open-source Python package that can be installed in a Modulus development environment for aggregation and exportation of data relative to different generated models produced in an experiment repository.

The Modulus Aggregator software architecture, shown in Figure 1, is divided into four main modules that play particular roles in the experiment data aggregation. The first module is responsible for capturing the data results of the Modulus experiment repository by accessing its particular directory structure to capture and register the results. Next, the Post-Processing module is called to organize all the results related, for example, to different chosen metrics and optionally different record frequencies for validation, training, and monitoring related quantities (it is possible to customize the record frequencies for each registered metric in *Modulus*). Then, the Aggregation of Results module takes place, aiming to put together all the post-processed results in a structure that can be properly extracted. Finally, the Data Extraction module extracts the aggregated data to a local directory of the experiment repository in order for it to be accessed later by the user with a Python Pandas-friendly format.

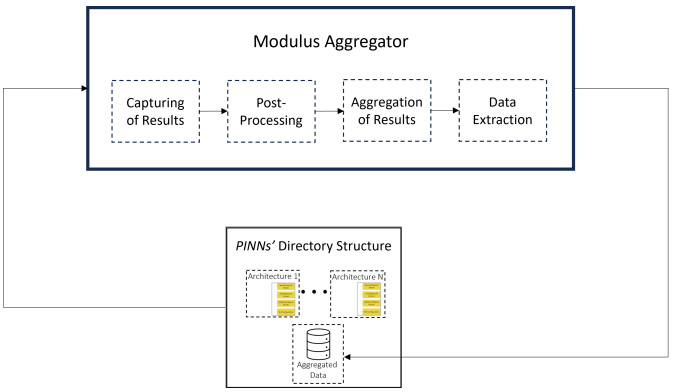

Figure 1. Modulus Aggregator's Software Architecture.

The Modulus Aggregator is built as a light package to extract results of the experiment without affecting the Modulus experimentation framework. The package provides a set of CLI (Command Line Interface) commands to aggregate and extract the results of the experiments. The commands are available at the *Modulus Aggregator* [3] GitHub. These commands can also be used in a script-like program to automate the aggregation and extraction of the results during the Modulus experiment. This is the framework used to evaluate the tool during an experiment that produced considerable data, further detailed in Section 5.

[3] `https://github.com/mthlimao/modulus_aggregator`

5 Experimental Evaluation

The Modulus Aggregator was used in a scenario of multiple models generation and a considerable amount of data in order for the tool to be evaluated in its capacity to assist the data analyst to better orient the hyperparameter configuration. With a view to provide options to a better analysis and to also allow the extraction and filtration of models, the aggregator tool was used complementing some of the features already available in TensorBoard. The experiment was conducted in the HPC environment provided by the *Santos Dumont* supercomputer [4].

5.1 Modulus Experiment

The conducted experiment is a wave propagation in a 2D domain with a *Ricker* source. The physical and configuration details of the problem are further addressed in [5]. For this experiment, the *multirun* mode available in *Modulus* is employed for the training. This mode allows the user to run the experiment routine with multiple combinations of hyperparameters with only one command. Values for the number of hidden layers (2, 3, 4, 5, and 6), layers size (128, 256, and 512), optimizers (*Adam*, *Sgd* and *Rmsprop*) and learning rate ($1e^{-05}$, $3e^{-05}$, $1e^{-04}$, $3e^{-04}$ e $1e^{-03}$) are varied. With this parameterization, 225 different models were generated, using a storage space of 8.78 GB. The experiment took approximately 120 hours to be concluded. No parallelization technique is used in this particular study. The exact command to run the experiment is in Figure 2.

```
$ python3 wave_2d.py -m arch.fully_connected.layer_size=128,256,512
arch.fully_connected.nr_layers=2,3,4,5,6 optimizer=sgd,adam,rmsprop
optimizer.lr=0.001,0.0003,0.0001,0.00003,0.00001
```

Figure 2. Command line for the Wave propagation experiment.

5.2 The Use of Modulus Aggregator

The tool is used in model filtering by a set of pre-fixed metrics of interest to reduce the amount of data to be analyzed. In this scenario, a simple Python script is utilized to define such metrics, filter the models and, at last, save the best models for later analysis. Figure 3 illustrates the utilization scheme of the aggregation tool. For the problem at hand, the following metrics for filtering are chosen:

- *Train/loss_aggregated*
- *Train/loss_c*
- *Train/loss_open_boundary*
- *Train/loss_u*
- *Train/loss_wave_equation*
- *Validators/VAL_0012/l2_relative_error_u*
- *Validators/Velocity/l2_relative_error_c*

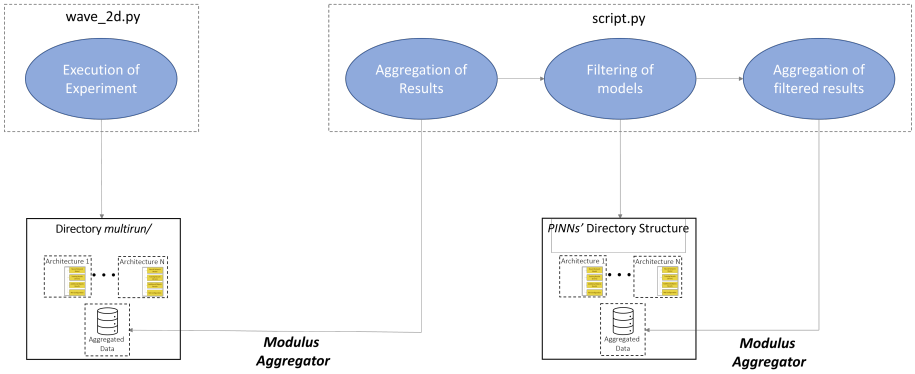

Figure 3. Utilization scheme of *Modulus Aggregator*.

[4] https://sdumont.lncc.br

[5] https://docs.nvidia.com/deeplearning/modulus/modulus-v2209/user_guide/foundational/2d_wave_equation

Table 1. Best filtered models.

	Hyper-parameters	**Best Metric**
A	(128, 5, 0.001, *adam*)	*Train/loss_aggregated*, *Train/loss_u*
B	(128, 6, 0.0001, *sgd*)	*Train/loss_open_boundary*
C	(256, 6, 0.001, *adam*)	*Train/loss_c*, *Validators/Velocity/l2_relative_error_c*
D	(512, 5, 0.0003, *adam*)	*Validators/VAL_0012/l2_relative_error_u*
E	(512, 5, 0.00003, *sgd*)	*Train/loss_wave_equation*

This way, only the best models for each of the chosen metrics are selected. The filtering process can reduce the space of analysis from a total of 225 models to only five. This represents a reduction of approximately 99.2% of the storage space to be locally analyzed (originally from 8.78Gb to 70.5 Mb), resulting in a much cleaner analysis environment in *TensorBoard*, for example, of the validation errors as the Figure 4 shows.

(a) Plots with no models filtering.

(b) Plots with models filtering.

Figure 4. Validation errors plots in *Tensorboard* before and after filtering with Modulus Aggregator.

5.3 Results Analysis

Table 1 presents the models, their associated hyperparameters (values of layer size, number of layers, initial learning rate, and optimizer, respectively), and the metrics that filtered each of the listed models. All the results analysis studies are then conducted with the aggregated results databases produced by the *Modulus Aggregator* tool, which are stored in the *multirun/* and *multirun_filtered/* directories, as illustrated in Figure 3.

One can note in Table 1 that two of the five selected models (A and C) simultaneously have the best performance for two metrics. Besides, three of the five models use the *Adam* optimizer (A, C and D) in the training. Based on this, conducting an analysis of results via *TensorBoard* is possible. Figure 5 shows the temporal evolution for training (*Train/loss_c* and *Train/loss_u*) and validation (*Validators/Velocity/l2_relative_error_c*) metrics.

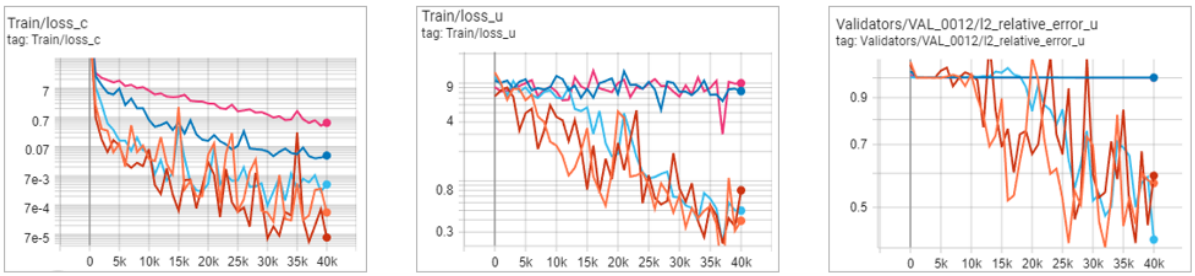

Figure 5. Temporal evolution for training and validation metrics.

It is possible to see in Figure 5 that, mainly for the plots related to the *Train/loss_u* e *Validators/Velocity/l2_relative_error_c*, three of the five models (the ones trained with the *Adam* optimizer) present better perfor-

mance, including a downward trend, for the error metrics related to u. This behavior is also observed for all other metrics, including the validation metrics (excluding *Train/loss_open_boundary* and *Train/loss_wave_equation*).

It is also possible to notice that these three models present different numbers of layers (5 and 6) and layer sizes (128, 256, and 512). This way, in this study, it is opted for the simplest possible architecture (5 layers and 128 neurons) for analyzing the influence of other hyperparameters in the validation metrics. In this sense, an investigation is carried out for the validation errors related to the last training step for the models. These metrics refer to the validation errors measured for the different time instants simulated in *Devito* [12] and utilized as references for ascertaining the quality of the trained PINNs. This study aims to evaluate which learning rate values could be used in its adjustment to train new models.

Figure 6 shows the variation of the validation errors for u based on the different *learning rate* values, with the simplest fixed architecture, using *Adam* optimizer. The produced plot indicated clearly that values of *learning rate* equal or smaller than 0.0001 entail a decreased performance for the validation errors for u. This can indicate to the analyst that greater *learning rate* values could preferably be employed for training new PINNs with *Modulus*.

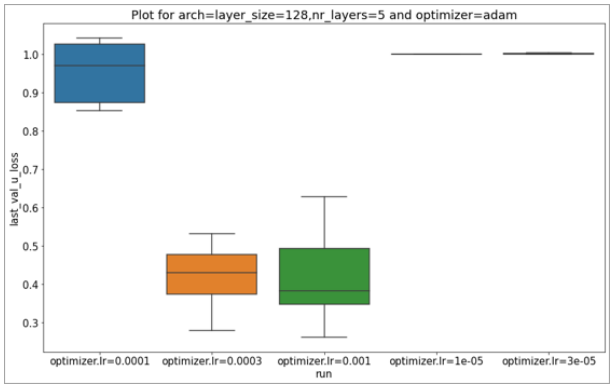

Figure 6. Validation Errors for u based on different values of *learning rate* for the fixed network architecture of 5 layers and 128 neurons with *Adam* optimizer.

Figure 7 illustrates the same type of plot produced with the validation errors for $\mathbf{u}$ and for the last training step based on the different optimizers, also with the same network architecture. For this second analysis, the *learning rate* value is fixed to 0.001. For this architecture and *learning rate*, the *Adam* optimizer also presents the best performance. In generating new models, adopting the *Adam* optimizer would be a justifiable option.

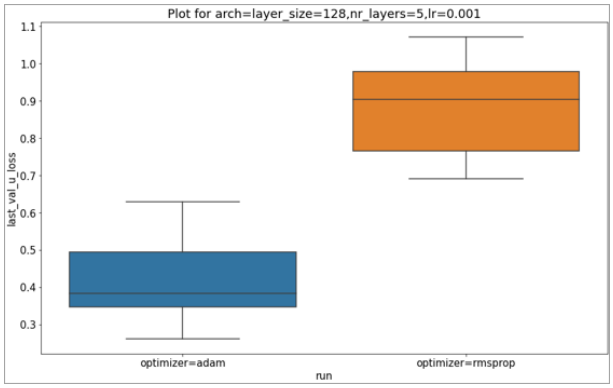

Figure 7. Validation Errors for u based on different optimizers, with a fixed network architecture of 5 layers and 128 neurons with *learning rate* equal to 0.001.

6 Conclusions

The Modulus Aggregator is developed as a library to be installed in the native development environment of the framework Modulus, and it acts as a CLI command package that allows the user to extract and aggregate the experiment results, mainly for a scenario of multiple model training. This paper shows that the aggregation tool was capable of helping in the analysis of results by reducing the hyperparameter space by aggregating data results and filtering the best models based on a set of pre-fixed metrics of interest. This way, the tool not only adds the

potential to increase the efficiency of the analyses of model results in *TensorBoard*, but it also adds the potential to reduce the physical data storage. It also can contribute to the automation of such a process, which is currently not possible with the exclusive use of the *TensorFlow data frame API*, given the necessity of publicizing the developed experiment, which is a complicated process to automate and is simply unfeasible in specific scenarios of research and development. There are still some limitations to be handled, so the tool can add more flexibility to the data extraction process. For instance, a feature that automatically filters models based on a set of metrics is perfectly feasible. Also, the possibility of handling metadata related to the use of computational resources, like memory and CPU, can further enrich an investigation by the analyst in a particular experiment.

Acknowledgements. This work is funded by CNPq, CAPES - Finance Code 001, and FAPERJ's Thematic Network in AI for Renewable Energy and Climate Change. We acknowledge the National Laboratory for Scientific Computing (LNCC/MCTI, Brazil) for providing HPC resources of the SDumont supercomputer.

Authorship statement. The authors hereby confirm that they are the sole liable persons responsible for the authorship of this work, and that all material that has been herein included as part of the present paper is either the property (and authorship) of the authors, or has the permission of the owners to be included here.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.
[2] S. Zhang, R. Liu, X. Tao, and X. Zhao. Deep Cross-Corpus Speech Emotion Recognition: Recent Advances and Perspectives. *Frontiers in Neurorobotics*, vol. 15, 2021.
[3] F. Suárez Bonilla and F. Ruiz Ugalde. Automatic translation of spanish natural language commands to control robot comands based on lstm neural network. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pp. 125–131, 2019.
[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A system for large-scale machine learning. *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283, 2016.
[5] F. Chollet and others. Keras. Publisher: GitHub, 2015.
[6] 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, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, d'F. Alché-Buc, E. Fox, and R. Garnett, eds, *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.
[7] L. Lu, X. Meng, Z. Mao, and G. Karniadakis. DeepXDE: A deep learning library for solving differential equations. *SIAM Review*, vol. 63, n. 1, pp. 208–228, 2021.
[8] E. Haghighat and R. Juanes. SciANN: A Keras/Tensorflow wrapper for scientific computations and physics-informed deep learning using artificial neural networks. *Computer Methods in Applied Mechanics and Engineering*, vol. 373, pp. 113552. arXiv:2005.08803 [cs], 2021.
[9] F. Chen, D. Sondak, P. Protopapas, M. Mattheakis, S. Liu, D. Agarwal, and M. Di Giovanni. NeuroDiffEq: A Python package for solving differential equations with neural networks. *Journal of Open Source Software*, vol. 5, pp. 1931, 2020.
[10] O. Hennigh, S. Narasimhan, M. Nabian, A. Subramaniam, K. Tangsali, Z. Fang, M. Rietmann, W. Byeon, and S. Choudhry. NVIDIA SimNet™: An AI-Accelerated Multi-Physics Simulation Framework. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12746 LNCS, pp. 447–461. ISBN: 9783030779764, 2021.
[11] T. Spinner, U. Schlegel, H. Schäfer, and M. El-Assady. explainer: A visual analytics framework for interactive and explainable machine learning. *IEEE transactions on visualization and computer graphics*, vol. 26, n. 1, pp. 1064–1074, 2019.
[12] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P. A. Witte, F. J. Herrmann, P. Velesko, and G. J. Gorman. Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration. *Geoscientific Model Development*, vol. 12, n. 3, pp. 1165–1187, 2019.