FastVPINNs: An efficient tensor-based Python library for solving partial differential equations using hp-Variational Physics Informed Neural Networks

Thivin Anandh [1¶], **Divij Ghose** [1], **and Sashikumaar Ganesan** [1¶]

1 Department of Computational and Data Sciences, Indian Institute of Science, Bangalore, India ¶ Corresponding author

DOI: 10.21105/joss.06764

Software

- Review
- Repository
- Archive

Editor: Patrick Diehl

Reviewers:

- @pescap
- @ziyiyin97

Submitted: 17 May 2024
Published: 30 July 2024

Introduction

Partial differential equations (PDEs) are essential in modeling physical phenomena such as heat transfer, electromagnetics and fluid dynamics. The current progress in the field of scientific machine learning (SciML) has resulted in incorporating deep learning and data-driven methods to solve PDEs. Let us consider a two-dimensional Poisson equation as an example, defined on the domain Ω, which reads as follows:

$$-\nabla^2 u(x) = f(x), \quad \Omega \in \mathbb{R}^2, \tag{1}$$
$$u(x) = g(x), \quad x \in \partial\Omega. \tag{2}$$

Here, $x \in \Omega$ is the spatial coordinate, $u(x)$ is the solution of the PDE, $f(x)$ is a known source term, and $g(x)$ is the value of the solution on the domain boundary, $\partial\Omega$.

A neural network is a parametric function of x, denoted as $u_{\text{NN}}(x; W, b)$. In this context, W and b represent the weights and biases of the network. When the neural network consists of h hidden layers, with the i^{th} layer containing n_i neurons, the mathematical representation of the function takes the following shape:

$$u_{\text{NN}}(x; W, b) = l \circ \mathrm{T}^{(h)} \circ \mathrm{T}^{(h-1)} \ldots \circ \mathrm{T}^{1}(x).$$

Here, $l : \mathbb{R}^{n_h} \to \mathbb{R}$ is a linear mapping in the output layer and $\mathrm{T}^{(i)}(\cdot) = \sigma(W^{(i)} \times \cdot + b^{(i)})$ is a non-linear mapping in the i^{th} layer $(i = 1, 2, \cdots, h)$, with the non-linear activation function σ and the weights $W^{(i)}$ and biases $b^{(i)}$ of the respective layers.

Physics-informed neural networks (PINNs), introduced by Raissi et al. (2019), work by incorporating the governing equations of the physical systems and the boundary conditions as physics-based constrains to train the neural networks. The core idea of PINNs lies in the ability of obtaining the gradients of the solutions with respect to the input using the automatic differention routines available within deep learning frameworks such as TensorFlow (Abadi et al., 2015). The loss function for the PINNs can be written as follows:

$$L_p(W, b) = \frac{1}{N_T} \sum_{t=1}^{N_T} \left((-\nabla^2 u_{\text{NN}}(x_t; W, b) - f(x_t)) \right)^2,$$
$$L_b(W, b) = \frac{1}{N_D} \sum_{d=1}^{N_D} \left(u_{\text{NN}}(x_d; W, b) - g(x_d) \right)^2,$$
$$L_{\text{PINN}}(W, b) = L_p + \tau L_b.$$

Here, the output of the neural network, $u_{\text{NN}}(x; W, b)$, is used to approximate the unknown solution. In addition, N_T is the total number of training points in the interior of the domain Ω, N_D is the total number of training points on the boundary $\partial\Omega$, τ is a scaling factor applied to control the penalty on the boundary term, and $L_{\text{PINN}}(W, b)$ is the loss function of the PINNs.

Variational Physics informed neural networks (VPINNs) (Kharazmi et al., 2019) are an extension of PINNs, where the weak form of the PDE is used in the loss function of the neural network. The weak form of the PDE is obtained by multiplying the PDE with a test function, integrating over the domain, and applying integration by parts to the higher order derivative terms. The method of hp-VPINNs (Kharazmi et al., 2021) was subsequently developed to increase the accuracy using h-refinement (increasing number of elements) and p-refinement (increasing the number of test functions). The domain Ω is divided into an array of non-overlapping cells, labeled as K_k, where $k = 1, 2, \dots, \texttt{N\_elem}$, ensuring that the complete union $\bigcup_{k=1}^{\texttt{N\_elem}} K_k = \Omega$ covers the entire domain Ω. The hp-VPINNs framework utilizes specific test functions v_k, where k ranges from 1 to N_elem, that are localized and defined within individual non-overlapping element across the domain.

$$v_k = \begin{cases} v^p \neq 0, & \text{over } K_k, \\ 0, & \text{elsewhere.} \end{cases}$$

The loss function of hp-VPINNs with `N_elem` elements in the domain can be written as follows:

$$\begin{aligned} L_v(W, b) &= \frac{1}{\texttt{N\_elem}} \sum_{k=1}^{\texttt{N\_elem}} \left(\int_{K_k} \nabla u_{\text{NN}}(x; W, b) \cdot \nabla v_k dK \; - \int_{K_k} f v_k \, dK \right)^2, \\ L_b(W, b) &= \frac{1}{N_D} \sum_{d=1}^{N_D} \left(u_{\text{NN}}(x; W, b) - g(x) \right)^2, \\ L_{\text{VPINN}}(W, b) &= L_v + \tau L_b. \end{aligned}$$

Where, K_k is the k^{th} element in the domain and v_k is the test function in the respective element. Further, $L_v(W, b)$ is the weak form PDE residual and $L_{\text{VPINN}}(W, b)$ is the loss function of the hp-VPINNs. For more information on the derivation of the weak form of the PDE and the loss function of hp-VPINNs, refer to Anandh et al. (2024) and Ganesan & Tobiska (2017).

Statement of need

The existing implementation of hp-VPINNs framework (Kharazmi, 2023) suffers from two major challenges. One is the inabilty of the framework to handle complex geometries and the other is the increased training time associated with the increase in number of elements within the domain. In the work Anandh et al. (2024), we presented FastVPINNs, which addresses both of these challenges. FastVPINNs handles complex geometries by using bilinear transformation, and it uses a tensor-based loss computation to reduce the dependency of training time on number of elements. The current implementation of FastVPINNs can acheive an speed-up of up to 100 times when compared with the existing implementation of hp-VPINNs. We have also shown that with proper hyperparameter selection, FastVPINNs can outperform PINNs both in terms of accuracy and training time, especially for problems with high frequency solutions.

In this work, we present the Python based implementation of the novel FastVPINNs framework which is built using TensorFlow-v2.0 (Abadi et al., 2015). FastVPINNs provides an elegant API for users to solve both forward and inverse problems for PDEs like the Poisson, Helmholtz, and Convection-Diffusion equations. With the current level of API abstraction, users should be able to solve PDEs with less than six API calls as shown in the minimal working example section. The framework is well-documented with examples, which can enable users to get started with the framework with ease. As per the authors' knowledge, FastVPINNs is the first open-source implementation of the hp-VPINNs framework with tensor-based loss computation and support for complex geometries.

The ability of the framework to allow users to train a hp-VPINNs to solve a PDE both faster and with minimal code, can result in widespread application of this method on several real-world problems, which often require complex geometries with a large number of elements within the domain.

Modules

The FastVPINNs framework consists of five core modules, which are Geometry, FE, Data, Physics, and Model.

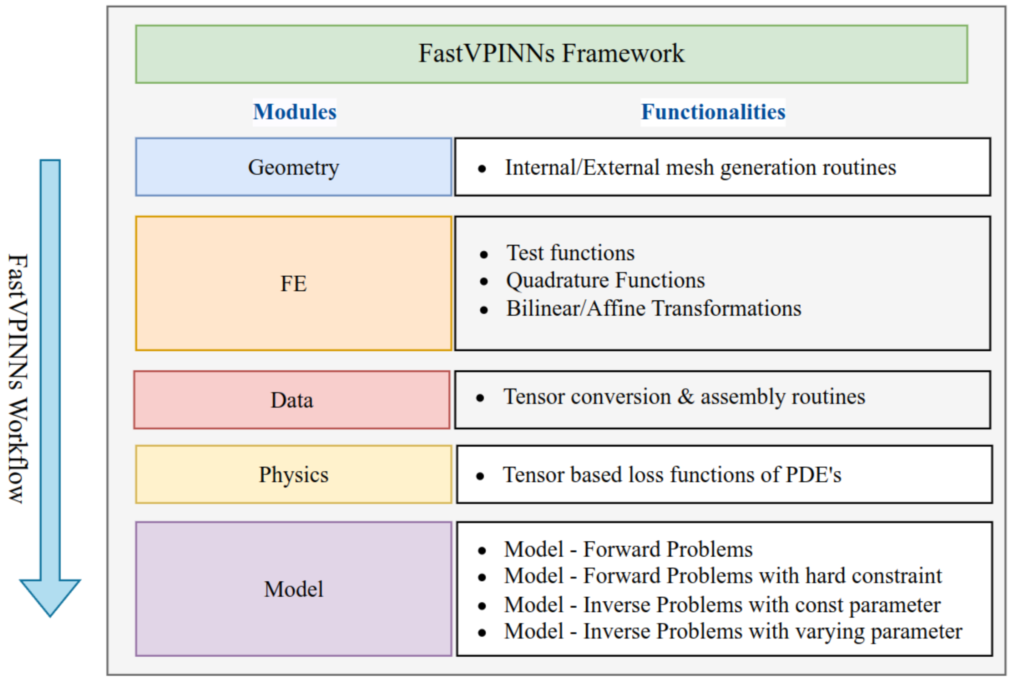
FastVPINNs Framework
Modules
Functionalities
Geometry
Internal/External mesh generation routines
FE
Test functions
Quadrature Functions
Bilinear/Affine Transformations
Data
Tensor conversion & assembly routines
Physics
Tensor based loss functions of PDE's
Model
Model - Forward Problems
Model - Forward Problems with hard constraint
Model - Inverse Problems with const parameter
Model - Inverse Problems with varying parameter
FastVPINNs Workflow

Figure 1: FastVPINNs Modules

Geometry Module

This module provides the functionality to define the geometry of the domain. The user can either generate a quadrilateral mesh internally or read an external `.mesh` file. The module also provides the functionality to obtain boundary points for complex geometries.

FE Module

The FE module is responsible for handling the finite element test functions and their gradients. The module's functionality can be broadly classified into into four categories:

- **Test functions**: Provides the test function values and its gradients for a given reference element. In our framework, we use Lagrange polynomials as the test functions.
- **Quadrature functions**: Provides the quadrature points and weights for a given element based on the quadrature order and the quadrature method selected by the user. This method will be used for performing the numerical integration of the weak form residual of the PDE.

- **Transformation functions**: Provides the implementation of transformation routines such as affine transformation and bilinear transformation. This can be used to transform the test function values and gradients from the reference element to the actual element.
- **Finite Element Setup**: Provides the functionality to set up the test functions, quadrature rules and the transformation for every element and save them in a common class to access them. Further, it also hosts functions to plot the mesh with quadrature points, assign boundary values based on the boundary points obtained from the geometry module and calculate the forcing term in the residual.

Remark: The module is named FE (Finite Element) Module because of its similarities with classical FEM routines, such as test functions, numerical quadratures, and transformations. However, we would like to state that our framework is not an FEM solver, but an hp-VPINNs solver.

Data Module:

The Data module collects data from all modules that are required for training and converts them to a tensor data type with the user specified precision (for example, `tf.float32` or `tf.float64`). It also assembles the test function values and gradients to form a three-dimensional tensor, which will be used during the loss computation.

Physics Module:

This module contains functions that are used to compute the variational loss function for different PDEs. These functions accept the test function tensors and the predicted gradients from the neural network along with the forcing matrix to compute the PDE residuals using tensor-based operations.

Model Module:

This module contains custom subclasses of the `tensorflow.keras.Model` class, which are used to train the neural network. The module provides the functionality to train the neural network using the tensor based loss computation and also provides the functionality to predict the solution of the PDE for a given input.

Minimal Working Example

With the higher level of abstraction provided by the FastVPINNs framework, users can solve a PDE with just six API calls. A Minimal working example to solve the Poisson equation using the FastVPINNs framework can be found here. The example files with detailed documentation can be found in the Tutorials section of the documentation.

Testing

The FastVPINNs framework has a strong testing suite, which tests the core functionalities of the framework. The testing suite is built using the `pytest` framework and is integrated with the continuous integration pipeline provided by Github Actions. The testing functionalites can be broadly classified into the three categories:

- **Unit Testing**: Covers the testing of individual modules and their functionalities.
- **Integration Testing**: Covers the overall flow of the framework. Different PDEs are solved with different parameters to check if the accuracy after training is within the acceptable limits. This ensures that the collection of modules work together as expected.

- **Compatibility Testing**: The framework is tested with different versions of Python such as 3.8, 3.9, 3.10, 3.11 and on different versions of OS such as Ubuntu, MacOS, and Windows to ensure the compatibility of the framework across different platforms.

Acknowledgements

We thank Shell Research, India, for providing partial funding for this project. We are thankful to the MHRD Grant No.STARS-1/388 (SPADE) for partial support.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., … Zheng, X. (2015). *TensorFlow: Large-scale machine learning on heterogeneous systems*. https://doi.org/10.48550/arXiv.1603.04467

Anandh, T., Ghose, D., Jain, H., & Ganesan, S. (2024). *FastVPINNs: Tensor-driven acceleration of VPINNs for complex geometries*. https://doi.org/10.48550/arXiv.2404.12063

Ganesan, S., & Tobiska, L. (2017). *Finite elements: Theory and algorithms*. Cambridge University Press. https://doi.org/10.1017/9781108235013

Kharazmi, E. (2023). *hp-VPINNs: High-performance variational physics-informed neural networks*. https://github.com/ehsankharazmi/hp-VPINNs

Kharazmi, E., Zhang, Z., & Karniadakis, G. E. (2019). Variational physics-informed neural networks for solving partial differential equations. *arXiv Preprint arXiv:1912.00873*. https://doi.org/10.48550/arXiv.1912.00873

Kharazmi, E., Zhang, Z., & Karniadakis, G. E. (2021). hp-VPINNs: Variational physics-informed neural networks with domain decomposition. *Computer Methods in Applied Mechanics and Engineering*, *374*, 113547. https://doi.org/10.1016/j.cma.2020.113547

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